The Java® Language Specification

Java SE 11 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
Daniel Smith

2018-08-21
# Table of Contents

1 **Introduction** 1  
   1.1 Organization of the Specification 2  
   1.2 Example Programs 6  
   1.3 Notation 6  
   1.4 Relationship to Predefined Classes and Interfaces 7  
   1.5 Feedback 7  
   1.6 References 7  

2 **Grammars** 9  
   2.1 Context-Free Grammars 9  
   2.2 The Lexical Grammar 9  
   2.3 The Syntactic Grammar 10  
   2.4 Grammar Notation 10  

3 **Lexical Structure** 15  
   3.1 Unicode 15  
   3.2 Lexical Translations 16  
   3.3 Unicode Escapes 17  
   3.4 Line Terminators 19  
   3.5 Input Elements and Tokens 19  
   3.6 White Space 20  
   3.7 Comments 21  
   3.8 Identifiers 22  
   3.9 Keywords 24  
   3.10 Literals 25  
      3.10.1 Integer Literals 25  
      3.10.2 Floating-Point Literals 32  
      3.10.3 Boolean Literals 35  
      3.10.4 Character Literals 35  
      3.10.5 String Literals 36  
      3.10.6 Escape Sequences for Character and String Literals 38  
      3.10.7 The Null Literal 39  
   3.11 Separators 40  
   3.12 Operators 40  

4 **Types, Values, and Variables** 41  
   4.1 The Kinds of Types and Values 41  
   4.2 Primitive Types and Values 42  
      4.2.1 Integral Types and Values 43
4.2.2 Integer Operations 43
4.2.3 Floating-Point Types, Formats, and Values 45
4.2.4 Floating-Point Operations 48
4.2.5 The boolean Type and boolean Values 51
4.3 Reference Types and Values 52
4.3.1 Objects 53
4.3.2 The Class Object 56
4.3.3 The Class String 56
4.3.4 When Reference Types Are the Same 57
4.4 Type Variables 57
4.5 Parameterized Types 59
4.5.1 Type Arguments of Parameterized Types 60
4.5.2 Members and Constructors of Parameterized Types 63
4.6 Type Erasure 64
4.7 Reifiable Types 65
4.8 Raw Types 66
4.9 Intersection Types 70
4.10 Subtyping 71
4.10.1 Subtyping among Primitive Types 71
4.10.2 Subtyping among Class and Interface Types 72
4.10.3 Subtyping among Array Types 73
4.10.4 Least Upper Bound 73
4.10.5 Type Projections 76
4.11 Where Types Are Used 78
4.12 Variables 83
4.12.1 Variables of Primitive Type 83
4.12.2 Variables of Reference Type 83
4.12.3 Kinds of Variables 85
4.12.4 final Variables 87
4.12.5 Initial Values of Variables 89
4.12.6 Types, Classes, and Interfaces 91

5 Conversions and Contexts 95
5.1 Kinds of Conversion 98
5.1.1 Identity Conversion 98
5.1.2 Widening Primitive Conversion 98
5.1.3 Narrowing Primitive Conversion 100
5.1.4 Widening and Narrowing Primitive Conversion 103
5.1.5 Widening Reference Conversion 103
5.1.6 Narrowing Reference Conversion 103
5.1.6.1 Allowed Narrowing Reference Conversion 104
5.1.6.2 Checked and Unchecked Narrowing Reference Conversions 105
5.1.6.3 Narrowing Reference Conversions at Run Time 105
5.1.7 Boxing Conversion 107
5.1.8 Unboxing Conversion 109
5.1.9 Unchecked Conversion 110
5.1.10 Capture Conversion 111
5.1.11 String Conversion 113
5.1.12 Forbidden Conversions 114
5.1.13 Value Set Conversion 114
5.2 Assignment Contexts 115
5.3 Invocation Contexts 120
5.4 String Contexts 122
5.5 Casting Contexts 122
5.6 Numeric Contexts 128
5.6.1 Unary Numeric Promotion 129
5.6.2 Binary Numeric Promotion 130

6 Names 133
6.1 Declarations 134
6.2 Names and Identifiers 141
6.3 Scope of a Declaration 143
6.4 Shadowing and Obscuring 147
6.4.1 Shadowing 149
6.4.2 Obscuring 152
6.5 Determining the Meaning of a Name 153
6.5.1 Syntactic Classification of a Name According to Context 154
6.5.2 Reclassification of Contextually Ambiguous Names 157
6.5.3 Meaning of Module Names and Package Names 160
6.5.3.1 Simple Package Names 160
6.5.3.2 Qualified Package Names 160
6.5.4 Meaning of PackageOrTypeName 160
6.5.4.1 Simple PackageOrTypeName 160
6.5.4.2 Qualified PackageOrTypeName 160
6.5.5 Meaning of Type Names 161
6.5.5.1 Simple Type Names 161
6.5.5.2 Qualified Type Names 161
6.5.6 Meaning of Expression Names 162
6.5.6.1 Simple Expression Names 162
6.5.6.2 Qualified Expression Names 163
6.5.7 Meaning of Method Names 165
6.5.7.1 Simple Method Names 165
6.6 Access Control 166
6.6.1 Determining Accessibility 168
6.6.2 Details on protected Access 172
6.6.2.1 Access to a protected Member 172
6.6.2.2 Access to a protected Constructor 173
6.7 Fully Qualified Names and Canonical Names 174

7 Packages and Modules 177
7.1 Package Members 178
7.2 Host Support for Modules and Packages 179
7.3 Compilation Units 182
7.4 Package Declarations 183
7.4.1 Named Packages 184
7.4.2 Unnamed Packages 184
7.4.3 Package Observability and Visibility 185
7.5 Import Declarations 186
7.5.1 Single-Type-Import Declarations 187
7.5.2 Type-Import-on-Demand Declarations 189
7.5.3 Single-Static-Import Declarations 190
7.5.4 Static-Import-on-Demand Declarations 191
7.6 Top Level Type Declarations 192
7.7 Module Declarations 195
7.7.1 Dependences 197
7.7.2 Exported and Opened Packages 200
7.7.3 Service Consumption 201
7.7.4 Service Provision 201
7.7.5 Unnamed Modules 202
7.7.6 Observability of a Module 203

8 Classes 205
8.1 Class Declarations 207
8.1.1 Class Modifiers 207
  8.1.1.1 abstract Classes 208
  8.1.1.2 final Classes 210
  8.1.1.3 strictfp Classes 210
8.1.2 Generic Classes and Type Parameters 210
8.1.3 Inner Classes and Enclosing Instances 213
8.1.4 Superclasses and Subclasses 216
8.1.5 Superinterfaces 218
8.1.6 Class Body and Member Declarations 222
8.2 Class Members 222
8.3 Field Declarations 227
  8.3.1 Field Modifiers 232
    8.3.1.1 static Fields 232
    8.3.1.2 final Fields 235
    8.3.1.3 transient Fields 235
    8.3.1.4 volatile Fields 236
  8.3.2 Field Initialization 237
  8.3.3 Restrictions on Field References in Initializers 239
8.4 Method Declarations 242
  8.4.1 Formal Parameters 243
  8.4.2 Method Signature 247
  8.4.3 Method Modifiers 248
    8.4.3.1 abstract Methods 248
    8.4.3.2 static Methods 250
    8.4.3.3 final Methods 250
    8.4.3.4 native Methods 251
    8.4.3.5 strictfp Methods 252
8.4.3.6 **synchronized** Methods 252
8.4.4 Generic Methods 253
8.4.5 Method Result 254
8.4.6 Method Throws 255
8.4.7 Method Body 256
8.4.8 Inheritance, Overriding, and Hiding 257
  8.4.8.1 Overriding (by Instance Methods) 258
  8.4.8.2 Hiding (by Class Methods) 262
  8.4.8.3 Requirements in Overriding and Hiding 263
  8.4.8.4 Inheriting Methods with Override-Equivalent Signatures 267
8.4.9 Overloading 268
8.5 Member Type Declarations 271
  8.5.1 Static Member Type Declarations 272
8.6 Instance Initializers 272
8.7 Static Initializers 272
8.8 Constructor Declarations 273
  8.8.1 Formal Parameters 274
  8.8.2 Constructor Signature 275
  8.8.3 Constructor Modifiers 275
  8.8.4 Generic Constructors 276
  8.8.5 Constructor Throws 276
  8.8.6 The Type of a Constructor 277
  8.8.7 Constructor Body 277
  8.8.7.1 Explicit Constructor Invocations 278
  8.8.8 Constructor Overloading 282
  8.8.9 Default Constructor 282
  8.8.10 Preventing Instantiation of a Class 284
8.9 Enum Types 284
  8.9.1 Enum Constants 285
  8.9.2 Enum Body Declarations 286
  8.9.3 Enum Members 288

9 **Interfaces** 295
9.1 Interface Declarations 296
  9.1.1 Interface Modifiers 296
    9.1.1.1 **abstract** Interfaces 297
    9.1.1.2 **strictfp** Interfaces 297
  9.1.2 Generic Interfaces and Type Parameters 297
  9.1.3 Superinterfaces and Subinterfaces 298
  9.1.4 Interface Body and Member Declarations 300
9.2 Interface Members 300
9.3 Field (Constant) Declarations 301
  9.3.1 Initialization of Fields in Interfaces 303
9.4 Method Declarations 304
  9.4.1 Inheritance and Overriding 305
    9.4.1.1 Overriding (by Instance Methods) 307
11.2.2 Exception Analysis of Statements 367
11.2.3 Exception Checking 368
11.3 Run-Time Handling of an Exception 370

12 Execution 375
12.1 Java Virtual Machine Startup 375
  12.1.1 Load the Class Test 376
  12.1.2 Link Test: Verify, Prepare, (Optionally) Resolve 376
  12.1.3 Initialize Test: Execute Initializers 377
  12.1.4 Invoke Test.main 378
12.2 Loading of Classes and Interfaces 378
  12.2.1 The Loading Process 379
12.3 Linking of Classes and Interfaces 380
  12.3.1 Verification of the Binary Representation 380
  12.3.2 Preparation of a Class or Interface Type 381
  12.3.3 Resolution of Symbolic References 381
12.4 Initialization of Classes and Interfaces 383
  12.4.1 When Initialization Occurs 383
  12.4.2 Detailed Initialization Procedure 386
12.5 Creation of New Class Instances 388
12.6 Finalization of Class Instances 391
  12.6.1 Implementing Finalization 393
  12.6.2 Interaction with the Memory Model 394
12.7 Unloading of Classes and Interfaces 396
12.8 Program Exit 397

13 Binary Compatibility 399
13.1 The Form of a Binary 400
13.2 What Binary Compatibility Is and Is Not 406
13.3 Evolution of Packages and Modules 407
13.4 Evolution of Classes 408
  13.4.1 abstract Classes 408
  13.4.2 final Classes 409
  13.4.3 public Classes 409
  13.4.4 Superclasses and Superinterfaces 409
  13.4.5 Class Type Parameters 411
  13.4.6 Class Body and Member Declarations 411
  13.4.7 Access to Members and Constructors 413
  13.4.8 Field Declarations 414
  13.4.9 final Fields and static Constant Variables 416
  13.4.10 static Fields 417
  13.4.11 transient Fields 417
  13.4.12 Method and Constructor Declarations 417
  13.4.13 Method and Constructor Type Parameters 418
  13.4.14 Method and Constructor Formal Parameters 419
  13.4.15 Method Result Type 420
  13.4.16 abstract Methods 420
13.4.17 final Methods 421
13.4.18 native Methods 421
13.4.19 static Methods 422
13.4.20 synchronized Methods 422
13.4.21 Method and Constructor Throws 422
13.4.22 Method and Constructor Body 422
13.4.23 Method and Constructor Overloading 422
13.4.24 Method Overriding 424
13.4.25 Static Initializers 424
13.4.26 Evolution of Enums 424
13.5 Evolution of Interfaces 424
13.5.1 public Interfaces 424
13.5.2 Superinterfaces 425
13.5.3 Interface Members 425
13.5.4 Interface Type Parameters 425
13.5.5 Field Declarations 426
13.5.6 Interface Method Declarations 426
13.5.7 Evolution of Annotation Types 427
14 Blocks and Statements 429
14.1 Normal and Abrupt Completion of Statements 429
14.2 Blocks 431
14.3 Local Class Declarations 431
14.4 Local Variable Declaration Statements 432
14.4.1 Local Variable Declarators and Types 434
14.4.2 Execution of Local Variable Declarations 435
14.5 Statements 436
14.6 The Empty Statement 438
14.7 Labeled Statements 438
14.8 Expression Statements 439
14.9 The if Statement 440
14.9.1 The if-then Statement 441
14.9.2 The if-then-else Statement 441
14.10 The assert Statement 442
14.11 The switch Statement 445
14.12 The while Statement 449
14.12.1 Abrupt Completion of while Statement 449
14.13 The do Statement 450
14.13.1 Abrupt Completion of do Statement 451
14.14 The for Statement 452
14.14.1 The basic for Statement 452
14.14.1.1 Initialization of for Statement 453
14.14.1.2 Iteration of for Statement 453
14.14.1.3 Abrupt Completion of for Statement 454
14.14.2 The enhanced for statement 455
14.15 The break Statement 458
14.16 The continue Statement 460
14.17 The return Statement 462
14.18 The throw Statement 464
14.19 The synchronized Statement 466
14.20 The try statement 467
  14.20.1 Execution of try-catch 470
  14.20.2 Execution of try-finally and try-catch-finally 471
  14.20.3 try-with-resources 473
    14.20.3.1 Basic try-with-resources 476
    14.20.3.2 Extended try-with-resources 479
14.21 Unreachable Statements 479

15 Expressions 487
  15.1 Evaluation, Denotation, and Result 487
  15.2 Forms of Expressions 488
  15.3 Type of an Expression 489
  15.4 FP-strict Expressions 490
  15.5 Expressions and Run-Time Checks 490
  15.6 Normal and Abrupt Completion of Evaluation 492
  15.7 Evaluation Order 494
    15.7.1 Evaluate Left-Hand Operand First 494
    15.7.2 Evaluate Operands before Operation 496
    15.7.3 Evaluation Respects Parentheses and Precedence 497
    15.7.4 Argument Lists are Evaluated Left-to-Right 498
    15.7.5 Evaluation Order for Other Expressions 499
  15.8 Primary Expressions 499
    15.8.1 Lexical Literals 500
    15.8.2 Class Literals 501
    15.8.3 this 502
    15.8.4 Qualified this 503
    15.8.5 Parenthesized Expressions 503
  15.9 Class Instance Creation Expressions 504
    15.9.1 Determining the Class being Instantiated 506
    15.9.2 Determining Enclosing Instances 507
    15.9.3 Choosing the Constructor and its Arguments 509
    15.9.4 Run-Time Evaluation of Class Instance Creation Expressions 513
    15.9.5 Anonymous Class Declarations 514
      15.9.5.1 Anonymous Constructors 515
  15.10 Array Creation and Access Expressions 516
    15.10.1 Array Creation Expressions 516
    15.10.2 Run-Time Evaluation of Array Creation Expressions 517
    15.10.3 Array Access Expressions 521
    15.10.4 Run-Time Evaluation of Array Access Expressions 521
  15.11 Field Access Expressions 524
    15.11.1 Field Access Using a Primary 524
    15.11.2 Accessing Superclass Members using super 527
  15.12 Method Invocation Expressions 529
15.12.1 Compile-Time Step 1: Determine Class or Interface to Search 530
15.12.2 Compile-Time Step 2: Determine Method Signature 532
  15.12.2.1 Identify Potentially Applicable Methods 538
  15.12.2.2 Phase 1: Identify Matching Arity Methods Applicable by Strict Invocation 541
  15.12.2.3 Phase 2: Identify Matching Arity Methods Applicable by Loose Invocation 542
  15.12.2.4 Phase 3: Identify Methods Applicable by Variable Arity Invocation 543
  15.12.2.5 Choosing the Most Specific Method 543
  15.12.2.6 Method Invocation Type 547
15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate? 548
15.12.4 Run-Time Evaluation of Method Invocation 551
  15.12.4.1 Compute Target Reference (If Necessary) 551
  15.12.4.2 Evaluate Arguments 553
  15.12.4.3 Check Accessibility of Type and Method 554
  15.12.4.4 Locate Method to Invoke 555
  15.12.4.5 Create Frame, Synchronize, Transfer Control 559
15.13 Method Reference Expressions 561
  15.13.1 Compile-Time Declaration of a Method Reference 564
  15.13.2 Type of a Method Reference 569
  15.13.3 Run-Time Evaluation of Method References 571
15.14 Postfix Expressions 574
  15.14.1 Expression Names 575
  15.14.2 Postfix Increment Operator ++ 575
  15.14.3 Postfix Decrement Operator -- 575
15.15 Unary Operators 576
  15.15.1 Prefix Increment Operator ++ 578
  15.15.2 Prefix Decrement Operator -- 578
  15.15.3 Unary Plus Operator + 579
  15.15.4 Unary Minus Operator − 579
  15.15.5 Bitwise Complement Operator ~ 580
  15.15.6 Logical Complement Operator ! 580
15.16 Cast Expressions 580
15.17 Multiplicative Operators 582
  15.17.1 Multiplication Operator * 583
  15.17.2 Division Operator / 584
  15.17.3 Remainder Operator % 585
15.18 Additive Operators 588
  15.18.1 String Concatenation Operator + 588
  15.18.2 Additive Operators (+ and −) for Numeric Types 591
15.19 Shift Operators 593
15.20 Relational Operators 594
  15.20.1 Numerical Comparison Operators <, <=, >, and >= 594
  15.20.2 Type Comparison Operator instanceof 596
15.21 Equality Operators 597
  15.21.1 Numerical Equality Operators == and != 597
15.21.2 Boolean Equality Operators == and != 598
15.21.3 Reference Equality Operators == and != 599

15.22 Bitwise and Logical Operators 599
15.22.1 Integer Bitwise Operators &, ^, and | 600
15.22.2 Boolean Logical Operators &, ^, and | 601

15.23 Conditional-And Operator && 601
15.24 Conditional-Or Operator || 602
15.25 Conditional Operator ? : 603
15.25.1 Boolean Conditional Expressions 610
15.25.2 Numeric Conditional Expressions 610
15.25.3 Reference Conditional Expressions 611

15.26 Assignment Operators 612
15.26.1 Simple Assignment Operator = 613
15.26.2 Compound Assignment Operators 619

15.27 Lambda Expressions 625
15.27.1 Lambda Parameters 627
15.27.2 Lambda Body 631
15.27.3 Type of a Lambda Expression 634
15.27.4 Run-Time Evaluation of Lambda Expressions 636

15.28 Constant Expressions 637
16 Definite Assignment 639
16.1 Definite Assignment and Expressions 645
16.1.1 Boolean Constant Expressions 645
16.1.2 Conditional-And Operator && 645
16.1.3 Conditional-Or Operator || 646
16.1.4 Logical Complement Operator ! 646
16.1.5 Conditional Operator ? : 646
16.1.6 Conditional Operator ? : 647
16.1.7 Other Expressions of Type boolean 647
16.1.8 Assignment Expressions 647
16.1.9 Operators ++ and -- 648
16.1.10 Other Expressions 648

16.2 Definite Assignment and Statements 650
16.2.1 Empty Statements 650
16.2.2 Blocks 650
16.2.3 Local Class Declaration Statements 651
16.2.4 Local Variable Declaration Statements 651
16.2.5 Labeled Statements 652
16.2.6 Expression Statements 652
16.2.7 if Statements 652
16.2.8 assert Statements 653
16.2.9 switch Statements 653
16.2.10 while Statements 654
16.2.11 do Statements 654
16.2.12 for Statements 654
16.2.12.1 Initialization Part of for Statement 655
16.2.12.2 Incrementation Part of for Statement 656
16.2.13 break, continue, return, and throw Statements 656
16.2.14 synchronized Statements 656
16.2.15 try Statements 657
16.3 Definite Assignment and Parameters 658
16.4 Definite Assignment and Array Initializers 658
16.5 Definite Assignment and Enum Constants 659
16.6 Definite Assignment and Anonymous Classes 659
16.7 Definite Assignment and Member Types 660
16.8 Definite Assignment and Static Initializers 660
16.9 Definite Assignment, Constructors, and Instance Initializers 661

17 Threads and Locks 663
17.1 Synchronization 664
17.2 Wait Sets and Notification 664
17.2.1 Wait 665
17.2.2 Notification 666
17.2.3 Interruptions 667
17.2.4 Interactions of Waits, Notification, and Interruption 667
17.3 Sleep and Yield 668
17.4 Memory Model 669
17.4.1 Shared Variables 672
17.4.2 Actions 672
17.4.3 Programs and Program Order 673
17.4.4 Synchronization Order 674
17.4.5 Happens-before Order 675
17.4.6 Executions 678
17.4.7 Well-Formed Executions 679
17.4.8 Executions and Causality Requirements 679
17.4.9 Observable Behavior and Nonterminating Executions 682
17.5 final Field Semantics 684
17.5.1 Semantics of final Fields 686
17.5.2 Reading final Fields During Construction 686
17.5.3 Subsequent Modification of final Fields 687
17.5.4 Write-Protected Fields 688
17.6 Word Tearing 689
17.7 Non-Atomic Treatment of double and long 690

18 Type Inference 691
18.1 Concepts and Notation 692
18.1.1 Inference Variables 692
18.1.2 Constraint Formulas 693
18.1.3 Bounds 693
18.2 Reduction 695
18.2.1 Expression Compatibility Constraints 695
18.2.2 Type Compatibility Constraints 700
18.2.3 Subtyping Constraints 701
18.2.4 Type Equality Constraints 702
18.2.5 Checked Exception Constraints 704
18.3 Incorporation 706
  18.3.1 Complementary Pairs of Bounds 707
  18.3.2 Bounds Involving Capture Conversion 707
18.4 Resolution 708
18.5 Uses of Inference 710
  18.5.1 Invocation Applicability Inference 710
  18.5.2 Invocation Type Inference 712
    18.5.2.1 Poly Method Invocation Compatibility 712
    18.5.2.2 Additional Argument Constraints 715
  18.5.3 Functional Interface Parameterization Inference 719
  18.5.4 More Specific Method Inference 720

19 Syntax 723

A Limited License Grant 751
THE Java® programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features.

The Java programming language is strongly and statically typed. This specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

The Java programming language is a relatively high-level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C’s free or C++’s delete). High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way.

1.1 Organization of the Specification

Chapter 2 describes grammars and the notation used to present the lexical and syntactic grammars for the language.

Chapter 3 describes the lexical structure of the Java programming language, which is based on C and C++. The language is written in the Unicode character set. It supports the writing of Unicode characters on systems that support only ASCII.

Chapter 4 describes types, values, and variables. Types are subdivided into primitive types and reference types.

The primitive types are defined to be the same on all machines and in all implementations, and are various sizes of two's-complement integers, single- and double-precision IEEE 754 standard floating-point numbers, a boolean type, and a Unicode character char type. Values of the primitive types do not share state.

Reference types are the class types, the interface types, and the array types. The reference types are implemented by dynamically created objects that are either instances of classes or arrays. Many references to each object can exist. All objects (including arrays) support the methods of the class Object, which is the (single) root of the class hierarchy. A predefined String class supports Unicode character strings. Classes exist for wrapping primitive values inside of objects. In many cases, wrapping and unwrapping is performed automatically by the compiler (in which case, wrapping is called boxing, and unwrapping is called unboxing). Class and interface declarations may be generic, that is, they may be parameterized by other reference types. Such declarations may then be invoked with specific type arguments.

Variables are typed storage locations. A variable of a primitive type holds a value of that exact primitive type. A variable of a class type can hold a null reference or a reference to an object whose type is that class type or any subclass of that class type. A variable of an interface type can hold a null reference or a reference to an instance of any class that implements the interface. A variable of an array type can hold a null reference or a reference to an array. A variable of class type Object can hold a null reference or a reference to any object, whether class instance or array.

Chapter 5 describes conversions and numeric promotions. Conversions change the compile-time type and, sometimes, the value of an expression. These conversions include the boxing and unboxing conversions between primitive types and reference types. Numeric promotions are used to convert the operands of a numeric operator to a common type where an operation can be performed. There are no
loopholes in the language; casts on reference types are checked at run time to ensure type safety.

Chapter 6 describes declarations and names, and how to determine what names mean (that is, which declaration a name denotes). The Java programming language does not require classes and interfaces, or their members, to be declared before they are used. Declaration order is significant only for local variables, local classes, and the order of field initializers in a class or interface. Recommended naming conventions that make for more readable programs are described here.

Chapter 7 describes the structure of a program, which is organized into packages. The members of a package are classes, interfaces, and subpackages. Packages, and consequently their members, have names in a hierarchical name space; the Internet domain name system can usually be used to form unique package names. Compilation units contain declarations of the classes and interfaces that are members of a given package, and may import classes and interfaces from other packages to give them short names.

Packages may be grouped into modules that serve as building blocks in the construction of very large programs. The declaration of a module specifies which other modules (and thus packages, and thus classes and interfaces) are required in order to compile and run code in its own packages.

The Java programming language supports limitations on external access to the members of packages, classes, and interfaces. The members of a package may be accessible solely by other members in the same package, or by members in other packages of the same module, or by members of packages in different modules. Similar constraints apply to the members of classes and interfaces.

Chapter 8 describes classes. The members of classes are classes, interfaces, fields (variables) and methods. Class variables exist once per class. Class methods operate without reference to a specific object. Instance variables are dynamically created in objects that are instances of classes. Instance methods are invoked on instances of classes; such instances become the current object this during their execution, supporting the object-oriented programming style.

Classes support single inheritance, in which each class has a single superclass. Each class inherits members from its superclass, and ultimately from the class Object. Variables of a class type can reference an instance of that class or of any subclass of that class, allowing new types to be used with existing methods, polymorphically.

Classes support concurrent programming with synchronized methods. Methods declare the checked exceptions that can arise from their execution, which allows compile-time checking to ensure that exceptional conditions are handled. Objects
can declare a `finalize` method that will be invoked before the objects are discarded by the garbage collector, allowing the objects to clean up their state.

For simplicity, the language has neither declaration "headers" separate from the implementation of a class nor separate type and class hierarchies.

A special form of classes, enums, support the definition of small sets of values and their manipulation in a type safe manner. Unlike enumerations in other languages, enums are objects and may have their own methods.

Chapter 9 describes interfaces. The members of interfaces are classes, interfaces, constant fields, and methods. Classes that are otherwise unrelated can implement the same interface. A variable of an interface type can contain a reference to any object that implements the interface.

Classes and interfaces support multiple inheritance from interfaces. A class that implements one or more interfaces may inherit instance methods from both its superclass and its superinterfaces.

Annotation types are specialized interfaces used to annotate declarations. Such annotations are not permitted to affect the semantics of programs in the Java programming language in any way. However, they provide useful input to various tools.

Chapter 10 describes arrays. Array accesses include bounds checking. Arrays are dynamically created objects and may be assigned to variables of type `Object`. The language supports arrays of arrays, rather than multidimensional arrays.

Chapter 11 describes exceptions, which are nonresuming and fully integrated with the language semantics and concurrency mechanisms. There are three kinds of exceptions: checked exceptions, run-time exceptions, and errors. The compiler ensures that checked exceptions are properly handled by requiring that a method or constructor can result in a checked exception only if the method or constructor declares it. This provides compile-time checking that exception handlers exist, and aids programming in the large. Most user-defined exceptions should be checked exceptions. Invalid operations in the program detected by the Java Virtual Machine result in run-time exceptions, such as `NullPointerException`. Errors result from failures detected by the Java Virtual Machine, such as `OutOfMemoryError`. Most simple programs do not try to handle errors.

Chapter 12 describes activities that occur during execution of a program. A program is normally stored as binary files representing compiled classes and interfaces. These binary files can be loaded into a Java Virtual Machine, linked to other classes and interfaces, and initialized.
After initialization, class methods and class variables may be used. Some classes may be instantiated to create new objects of the class type. Objects that are class instances also contain an instance of each superclass of the class, and object creation involves recursive creation of these superclass instances.

When an object is no longer referenced, it may be reclaimed by the garbage collector. If an object declares a finalizer, the finalizer is executed before the object is reclaimed to give the object a last chance to clean up resources that would not otherwise be released. When a class is no longer needed, it may be unloaded.

Chapter 13 describes binary compatibility, specifying the impact of changes to types on other types that use the changed types but have not been recompiled. These considerations are of interest to developers of types that are to be widely distributed, in a continuing series of versions, often through the Internet. Good program development environments automatically recompile dependent code whenever a type is changed, so most programmers need not be concerned about these details.

Chapter 14 describes blocks and statements, which are based on C and C++. The language has no goto statement, but includes labeled break and continue statements. Unlike C, the Java programming language requires boolean (or Boolean) expressions in control-flow statements, and does not convert types to boolean implicitly (except through unboxing), in the hope of catching more errors at compile time. A synchronized statement provides basic object-level monitor locking. A try statement can include catch and finally clauses to protect against non-local control transfers.

Chapter 15 describes expressions. This document fully specifies the (apparent) order of evaluation of expressions, for increased determinism and portability. Overloaded methods and constructors are resolved at compile time by picking the most specific method or constructor from those which are applicable.

Chapter 16 describes the precise way in which the language ensures that local variables are definitely set before use. While all other variables are automatically initialized to a default value, the Java programming language does not automatically initialize local variables in order to avoid masking programming errors.

Chapter 17 describes the semantics of threads and locks, which are based on the monitor-based concurrency originally introduced with the Mesa programming language. The Java programming language specifies a memory model for shared-memory multiprocessors that supports high-performance implementations.

Chapter 18 describes a variety of type inference algorithms used to test applicability of generic methods and to infer types in a generic method invocation.
Chapter 19 presents a syntactic grammar for the language.

1.2 Example Programs

Most of the example programs given in the text are ready to be executed and are similar in form to:

```java
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
        System.out.println();
    }
}
```

On a machine with the Oracle JDK installed, this class, stored in the file `Test.java`, can be compiled and executed by giving the commands:

```
javac Test.java
java Test Hello, world.
```

producing the output:

```
Hello, world.
```

1.3 Notation

Throughout this specification we refer to classes and interfaces drawn from the Java SE Platform API. Whenever we refer to a class or interface (other than those declared in an example) using a single identifier \( N \), the intended reference is to the class or interface named \( N \) in the package `java.lang`. We use the canonical name (§6.7) for classes or interfaces from packages other than `java.lang`.

Non-normative information, designed to clarify the specification, is given in smaller, indented text.

This is non-normative information. It provides intuition, rationale, advice, examples, etc.

The type system of the Java programming language occasionally relies on the notion of a substitution. The notation \([F_1 := T_1, \ldots, F_n := T_n]\) denotes substitution of \( F_i \) by \( T_i \) for \( 1 \leq i \leq n \).
1.4 Relationship to Predefined Classes and Interfaces

As noted above, this specification often refers to classes of the Java SE Platform API. In particular, some classes have a special relationship with the Java programming language. Examples include classes such as Object, Class, ClassLoader, String, Thread, and the classes and interfaces in package java.lang.reflect, among others. This specification constrains the behavior of such classes and interfaces, but does not provide a complete specification for them. The reader is referred to the Java SE Platform API documentation.

Consequently, this specification does not describe reflection in any detail. Many linguistic constructs have analogs in the Core Reflection API (java.lang.reflect) and the Language Model API (javax.lang.model), but these are generally not discussed here. For example, when we list the ways in which an object can be created, we generally do not include the ways in which the Core Reflection API can accomplish this. Readers should be aware of these additional mechanisms even though they are not mentioned in the text.

1.5 Feedback

Readers are invited to report technical errors and ambiguities in The Java® Language Specification to jls-jvms-spec-comments@openjdk.java.net. Questions concerning the behavior of javac (the reference compiler for the Java programming language), and in particular its conformance to this specification, may be sent to compiler-dev@openjdk.java.net.

1.6 References


CHAPTER 2

Grammars

THIS chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program.

2.1 Context-Free Grammars

A context-free grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished nonterminal, called the goal symbol, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2 The Lexical Grammar

A lexical grammar for the Java programming language is given in §3 (Lexical Structure). This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol Input (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5).

These input elements, with white space (§3.6) and comments (§3.7) discarded, form the terminal symbols for the syntactic grammar for the Java programming language and are called tokens (§3.5). These tokens are the identifiers (§3.8),
keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the Java programming language.

## 2.3 The Syntactic Grammar

The syntactic grammar for the Java programming language is given in Chapters 4, 6-10, 14, and 15. This grammar has as its terminal symbols the tokens defined by the lexical grammar. It defines a set of productions, starting from the goal symbol `CompilationUnit` (§7.3), that describe how sequences of tokens can form syntactically correct programs.

For convenience, the syntactic grammar is presented all together in Chapter 19.

## 2.4 Grammar Notation

Terminal symbols are shown in fixed width font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative definitions for the nonterminal then follow on succeeding lines.

For example, the syntactic production:

\[
\text{IfThenStatement}: \quad \text{if ( Expression ) Statement}
\]

states that the nonterminal \textit{IfThenStatement} represents the token \textit{if}, followed by a left parenthesis token, followed by an \textit{Expression}, followed by a right parenthesis token, followed by a \textit{Statement}.

The syntax \{x\} on the right-hand side of a production denotes zero or more occurrences of \textit{x}.

For example, the syntactic production:

\[
\text{ArgumentList}: \quad \text{Argument } \{, \text{Argument}\}
\]
states that an ArgumentList consists of an Argument, followed by zero or more occurrences of a comma and an Argument. The result is that an ArgumentList may contain any positive number of arguments.

The syntax \([x]\) on the right-hand side of a production denotes zero or one occurrences of \(x\). That is, \(x\) is an optional symbol. The alternative which contains the optional symbol actually defines two alternatives: one that omits the optional symbol and one that includes it.

This means that:

```
BreakStatement:
    break [Identifier] ;
```

is a convenient abbreviation for:

```
BreakStatement:
    break ;
    break Identifier ;
```

As another example, it means that:

```
BasicForStatement:
    for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement
```

is a convenient abbreviation for:

```
BasicForStatement:
    for ( ; [Expression] ; [ForUpdate] ) Statement
    for ( ForInit ; [Expression] ; [ForUpdate] ) Statement
```

which in turn is an abbreviation for:

```
BasicForStatement:
    for ( ; [ForUpdate] ) Statement
    for ( ; Expression ; [ForUpdate] ) Statement
    for ( ForInit ; [ForUpdate] ) Statement
    for ( ForInit ; Expression ; [ForUpdate] ) Statement
```

which in turn is an abbreviation for:
BasicForStatement:
  for (; ; ) Statement
  for (; ; ForUpdate ) Statement
  for (; Expression ; ) Statement
  for (; Expression ; ForUpdate ) Statement
  for ( ForInit ; ; ) Statement
  for ( ForInit ; ; ForUpdate ) Statement
  for ( ForInit ; Expression ; ) Statement
  for ( ForInit ; Expression ; ForUpdate ) Statement

so the nonterminal BasicForStatement actually has eight alternative right-hand sides.

A very long right-hand side may be continued on a second line by clearly indenting the second line.

For example, the syntactic grammar contains this production:

NormalClassDeclaration:
  { ClassModifier } class TypeIdentifier [ TypeParameters ]
  [ Superclass ] [ Superinterfaces ] ClassBody

which defines one right-hand side for the nonterminal NormalClassDeclaration.

The phrase (one of) on the right-hand side of a production signifies that each of the symbols on the following line or lines is an alternative definition.

For example, the lexical grammar contains the production:

ZeroToThree:
  (one of)
  0 1 2 3

which is merely a convenient abbreviation for:

ZeroToThree:
  0
  1
  2
  3

When an alternative in a production appears to be a token, it represents the sequence of characters that would make up such a token.

Thus, the production:

BooleanLiteral:
  (one of)
  true false
is shorthand for:

```plaintext
BooleanLiteral:
  true
  false
```

The right-hand side of a production may specify that certain expansions are not permitted by using the phrase "but not" and then indicating the expansions to be excluded.

For example:

```plaintext
Identifier:
  IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral
```

Finally, a few nonterminals are defined by a narrative phrase in roman type where it would be impractical to list all the alternatives.

For example:

```plaintext
RawInputCharacter:
  any Unicode character
```
Chapter 3

Lexical Structure

This chapter specifies the lexical structure of the Java programming language. Programs are written in Unicode (§3.1), but lexical translations are provided (§3.2) so that Unicode escapes (§3.3) can be used to include any Unicode character using only ASCII characters. Line terminators are defined (§3.4) to support the different conventions of existing host systems while maintaining consistent line numbers.

The Unicode characters resulting from the lexical translations are reduced to a sequence of input elements (§3.5), which are white space (§3.6), comments (§3.7), and tokens. The tokens are the identifiers (§3.8), keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the syntactic grammar.

3.1 Unicode

Programs are written using the Unicode character set. Information about this character set and its associated character encodings may be found at http://www.unicode.org/.

The Java SE Platform tracks the Unicode Standard as it evolves. The precise version of Unicode used by a given release is specified in the documentation of the class Character.

Versions of the Java programming language prior to JDK 1.1 used Unicode 1.1.5. Upgrades to newer versions of the Unicode Standard occurred in JDK 1.1 (to Unicode 2.0), JDK 1.1.7 (to Unicode 2.1), Java SE 1.4 (to Unicode 3.0), Java SE 5.0 (to Unicode 4.0), Java SE 7 (to Unicode 6.0), Java SE 8 (to Unicode 6.2), Java SE 9 (to Unicode 8.0), and Java SE 11 (to Unicode 10.0).

The Unicode standard was originally designed as a fixed-width 16-bit character encoding. It has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now U+0000
to U+10FFFF, using the hexadecimal \( U+n \) notation. Characters whose code points are greater than U+FFFF are called supplementary characters. To represent the complete range of characters using only 16-bit units, the Unicode standard defines an encoding called UTF-16. In this encoding, supplementary characters are represented as pairs of 16-bit code units, the first from the high-surrogates range, (U+D800 to U+DBFF), the second from the low-surrogates range (U+DC00 to U+DFFF). For characters in the range U+0000 to U+FFFF, the values of code points and UTF-16 code units are the same.

The Java programming language represents text in sequences of 16-bit code units, using the UTF-16 encoding.

Some APIs of the Java SE Platform, primarily in the `Character` class, use 32-bit integers to represent code points as individual entities. The Java SE Platform provides methods to convert between 16-bit and 32-bit representations.

This specification uses the terms `code point` and `UTF-16 code unit` where the representation is relevant, and the generic term `character` where the representation is irrelevant to the discussion.

Except for comments (§3.7), identifiers, and the contents of character and string literals (§3.10.4, §3.10.5), all input elements (§3.5) in a program are formed only from ASCII characters (or Unicode escapes (§3.3) which result in ASCII characters).

ASCII (ANSI X3.4) is the American Standard Code for Information Interchange. The first 128 characters of the Unicode UTF-16 encoding are the ASCII characters.

### 3.2 Lexical Translations

A raw Unicode character stream is translated into a sequence of tokens, using the following three lexical translation steps, which are applied in turn:

1. A translation of Unicode escapes (§3.3) in the raw stream of Unicode characters to the corresponding Unicode character. A Unicode escape of the form `\uxxxx`, where `xxxx` is a hexadecimal value, represents the UTF-16 code unit whose encoding is `xxxx`. This translation step allows any program to be expressed using only ASCII characters.

2. A translation of the Unicode stream resulting from step 1 into a stream of input characters and line terminators (§3.4).
3. A translation of the stream of input characters and line terminators resulting from step 2 into a sequence of input elements (§3.5) which, after white space (§3.6) and comments (§3.7) are discarded, comprise the tokens (§3.5) that are the terminal symbols of the syntactic grammar (§2.3).

The longest possible translation is used at each step, even if the result does not ultimately make a correct program while another lexical translation would. There is one exception: if lexical translation occurs in a type context (§4.11) and the input stream has two or more consecutive > characters that are followed by a non- character, then each > character must be translated to the token for the numerical comparison operator >.

The input characters a--b are tokenized (§3.5) as a, --, b, which is not part of any grammatically correct program, even though the tokenization a, -, -, b could be part of a grammatically correct program.

Without the rule for > characters, two consecutive > brackets in a type such as List<List<String>> would be tokenized as the signed right shift operator >>, while three consecutive > brackets in a type such as List<List<List<String>>> would be tokenized as the unsigned right shift operator >>>. Worse, the tokenization of four or more consecutive > brackets in a type such as List<List<List<List<String>>>> would be ambiguous, as various combinations of >, >>, and >>> tokens could represent the >>>> characters.

### 3.3 Unicode Escapes

A compiler for the Java programming language ("Java compiler") first recognizes Unicode escapes in its input, translating the ASCII characters \u followed by four hexadecimal digits to the UTF-16 code unit (§3.1) for the indicated hexadecimal value, and passing all other characters unchanged. Representing supplementary characters requires two consecutive Unicode escapes. This translation step results in a sequence of Unicode input characters.

```
UnicodeInputCharacter:  
  UnicodeEscape  
  RawInputCharacter

UnicodeEscape:  
  \ UnicodeMarker HexDigit HexDigit HexDigit HexDigit

UnicodeMarker:  
  \{u\}
```
3.3 Unicode Escapes

**HexDigit:**
(one of)
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

**RawInputCharacter:**
any Unicode character

The \, u, and hexadecimal digits here are all ASCII characters.

In addition to the processing implied by the grammar, for each raw input character that is a backslash \, input processing must consider how many other \ characters contiguously precede it, separating it from a non-\ character or the start of the input stream. If this number is even, then the \ is eligible to begin a Unicode escape; if the number is odd, then the \ is not eligible to begin a Unicode escape.

For example, the raw input "\\u2122=\\u2122" results in the eleven characters " \ 2 1 2 2 = ™ " (\\u2122 is the Unicode encoding of the character ™).

If an eligible \ is not followed by u, then it is treated as a **RawInputCharacter** and remains part of the escaped Unicode stream.

If an eligible \ is followed by u, or more than one u, and the last u is not followed by four hexadecimal digits, then a compile-time error occurs.

The character produced by a Unicode escape does not participate in further Unicode escapes.

For example, the raw input \u005cu005a results in the six characters \ u 0 0 5 a, because 005c is the Unicode value for \. It does not result in the character Z, which is Unicode character 005a, because the \ that resulted from the \u005c is not interpreted as the start of a further Unicode escape.

The Java programming language specifies a standard way of transforming a program written in Unicode into ASCII that changes a program into a form that can be processed by ASCII-based tools. The transformation involves converting any Unicode escapes in the source text of the program to ASCII by adding an extra u - for example, \uxxxx becomes \uuxxxx - while simultaneously converting non-ASCII characters in the source text to Unicode escapes containing a single u each.

This transformed version is equally acceptable to a Java compiler and represents the exact same program. The exact Unicode source can later be restored from this ASCII form by converting each escape sequence where multiple u's are present to a sequence of Unicode characters with one fewer u, while simultaneously converting each escape sequence with a single u to the corresponding single Unicode character.
A Java compiler should use the \uxxxx notation as an output format to display Unicode characters when a suitable font is not available.

### 3.4 Line Terminators

A Java compiler next divides the sequence of Unicode input characters into lines by recognizing *line terminators*.

**LineTerminator:**
- the ASCII LF character, also known as "newline"
- the ASCII CR character, also known as "return"
- the ASCII CR character followed by the ASCII LF character

**InputCharacter:**
- *UnicodeInputCharacter* but not CR or LF

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, not two.

A line terminator specifies the termination of the // form of a comment (§3.7).

The lines defined by line terminators may determine the line numbers produced by a Java compiler.

The result is a sequence of line terminators and input characters, which are the terminal symbols for the third step in the tokenization process.

### 3.5 Input Elements and Tokens

The input characters and line terminators that result from escape processing (§3.3) and then input line recognition (§3.4) are reduced to a sequence of *input elements*.

**Input:**

{InputElement} [Sub]

**InputElement:**
- *WhiteSpace*
- *Comment*
- *Token*
Token:
  Identifier
  Keyword
  Literal
  Separator
  Operator

Sub:
  the ASCII SUB character, also known as "control-Z"

Those input elements that are not white space or comments are tokens. The tokens are the terminal symbols of the syntactic grammar (§2.3).

White space (§3.6) and comments (§3.7) can serve to separate tokens that, if adjacent, might be tokenized in another manner. For example, the ASCII characters - and = in the input can form the operator token -= (§3.12) only if there is no intervening white space or comment.

As a special concession for compatibility with certain operating systems, the ASCII SUB character (\u001a, or control-Z) is ignored if it is the last character in the escaped input stream.

Consider two tokens $x$ and $y$ in the resulting input stream. If $x$ precedes $y$, then we say that $x$ is to the left of $y$ and that $y$ is to the right of $x$.

For example, in this simple piece of code:

```java
class Empty {
}
```

we say that the } token is to the right of the { token, even though it appears, in this two-dimensional representation, downward and to the left of the { token. This convention about the use of the words left and right allows us to speak, for example, of the right-hand operand of a binary operator or of the left-hand side of an assignment.

3.6 White Space

White space is defined as the ASCII space character, horizontal tab character, form feed character, and line terminator characters (§3.4).
3.7 Comments

There are two kinds of comments:

- /* text */
  
  A *traditional comment*: all the text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).

- // text
  
  An *end-of-line comment*: all the text from the ASCII characters // to the end of the line is ignored (as in C++).

Comment:
  
  TraditionalComment
  EndOfLineComment

TraditionalComment:
  /* CommentTail

CommentTail:
  * CommentTailStar
  NotStar CommentTail

CommentTailStar:
  /
  * CommentTailStar
  NotStarNotSlash CommentTail

NotStar:
  InputCharacter but not *
  LineTerminating

WhiteSpace:
  the ASCII SP character, also known as "space"
  the ASCII HT character, also known as "horizontal tab"
  the ASCII FF character, also known as "form feed"

LineTerminating
3.8 Identifiers

An identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter.

Identifier:
- IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
- JavaLetter {JavaLetterOrDigit}

JavaLetter:
  - any Unicode character that is a "Java letter"

JavaLetterOrDigit:
  - any Unicode character that is a "Java letter-or-digit"

A "Java letter" is a character for which the method Character.isJavaIdentifierStart(int) returns true.
A "Java letter-or-digit" is a character for which the method
Character.isJavaIdentifierPart(int) returns true.

The "Java letters" include uppercase and lowercase ASCII Latin letters A–Z (\u0041–\u005a), and a–z (\u0061–\u007a), and, for historical reasons, the ASCII dollar sign ($, or \u0024) and underscore (_, or \u005f). The dollar sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems. The underscore may be used in identifiers formed of two or more characters, but it cannot be used as a one-character identifier due to being a keyword.

The "Java digits" include the ASCII digits 0–9 (\u0030–\u0039).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§3.9), boolean literal (§3.10.3), or the null literal (§3.10.7), or a compile-time error occurs.

Two identifiers are the same only if, after ignoring characters that are ignorable, the identifiers have the same Unicode character for each letter or digit. An ignorable character is a character for which the method Character.isIdentifierIgnorable(int) returns true. Identifiers that have the same external appearance may yet be different.

For example, the identifiers consisting of the single letters LATIN CAPITAL LETTER A (A, \u0041), LATIN SMALL LETTER A (a, \u0061), GREEK CAPITAL LETTER ALPHA (A, \u0391), CYRILLIC SMALL LETTER A (a, \u0430) and MATHEMATICAL BOLD ITALIC SMALL A (a, \ud835\udc82) are all different.

Unicode composite characters are different from their canonical equivalent decomposed characters. For example, a LATIN CAPITAL LETTER A ACUTE (Á, \u00c1) is different from a LATIN CAPITAL LETTER A (A, \u0041) immediately followed by a NON-SPACING ACUTE (´, \u0301) in identifiers. See The Unicode Standard, Section 3.11 "Normalization Forms".

Examples of identifiers are:

- String
- i3
- ἀφετη
- MAX_VALUE
- isLetterOrDigit
A type identifier is an identifier that is not the character sequence `var`.

**TypeIdentifier:**

`Identifier` but not `var`

Type identifiers are used in certain contexts involving the declaration or use of types. For example, the name of a class must be a `TypeIdentifier`, so it is illegal to declare a class named `var` (§8.1).

### 3.9 Keywords

51 character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers (§3.8).

**Keyword:**

(one of)

<table>
<thead>
<tr>
<th>abstract</th>
<th>continue</th>
<th>for</th>
<th>new</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>default</td>
<td>if</td>
<td>package</td>
<td>synchronized</td>
</tr>
<tr>
<td>boolean</td>
<td>do</td>
<td>goto</td>
<td>private</td>
<td>this</td>
</tr>
<tr>
<td>break</td>
<td>double</td>
<td>implements</td>
<td>protected</td>
<td>throw</td>
</tr>
<tr>
<td>byte</td>
<td>else</td>
<td>import</td>
<td>public</td>
<td>throws</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>instanceof</td>
<td>return</td>
<td>transient</td>
</tr>
<tr>
<td>catch</td>
<td>extends</td>
<td>int</td>
<td>short</td>
<td>try</td>
</tr>
<tr>
<td>char</td>
<td>final</td>
<td>interface</td>
<td>static</td>
<td>void</td>
</tr>
<tr>
<td>class</td>
<td>finally</td>
<td>long</td>
<td>strictfp</td>
<td>volatile</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>native</td>
<td>super</td>
<td>while</td>
</tr>
</tbody>
</table>

_ (underscore)

The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

A variety of character sequences are sometimes assumed, incorrectly, to be keywords:

- `true` and `false` are not keywords, but rather boolean literals (§3.10.3).
- `null` is not a keyword, but rather the null literal (§3.10.7).
- `var` is not a keyword, but rather an identifier with special meaning as the type of a local variable declaration (§14.4, §14.14.1, §14.14.2, §14.20.3) and the type of a lambda formal parameter (§15.27.1).
A further ten character sequences are restricted keywords: open, module, requires, transitive, exports, opens, to, uses, provides, and with. These character sequences are tokenized as keywords solely where they appear as terminals in the ModuleDeclaration, ModuleDirective, and RequiresModifier productions (§7.7). They are tokenized as identifiers everywhere else, for compatibility with programs written before the introduction of restricted keywords. There is one exception: immediately to the right of the character sequence requires in the ModuleDirective production, the character sequence transitive is tokenized as a keyword unless it is followed by a separator, in which case it is tokenized as an identifier.

### 3.10 Literals

A literal is the source code representation of a value of a primitive type (§4.2), the String type (§4.3.3), or the null type (§4.1).

- **Literal:**
  - IntegerLiteral
  - FloatingPointLiteral
  - BooleanLiteral
  - CharacterLiteral
  - StringLiteral
  - NullLiteral

### 3.10.1 Integer Literals

An integer literal may be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2).

- **IntegerLiteral:**
  - DecimalIntegerLiteral
  - HexIntegerLiteral
  - OctalIntegerLiteral
  - BinaryIntegerLiteral

- **DecimalIntegerLiteral:**
  - DecimalNumeral [IntegerTypeSuffix]

- **HexIntegerLiteral:**
  - HexNumeral [IntegerTypeSuffix]
OctalIntegerLiteral:
OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix:
(one of)
1 L

An integer literal is of type long if it is suffixed with an ASCII letter L or l (ell); otherwise it is of type int (§4.2.1).

The suffix L is preferred, because the letter l (ell) is often hard to distinguish from the digit 1 (one).

Underscores are allowed as separators between digits that denote the integer.

In a hexadecimal or binary literal, the integer is only denoted by the digits after the 0x or 0b characters and before any type suffix. Therefore, underscores may not appear immediately after 0x or 0b, or after the last digit in the numeral.

In a decimal or octal literal, the integer is denoted by all the digits in the literal before any type suffix. Therefore, underscores may not appear before the first digit or after the last digit in the numeral. Underscores may appear after the initial 0 in an octal numeral (since 0 is a digit that denotes part of the integer) and after the initial non-zero digit in a non-zero decimal literal.
A decimal numeral is either the single ASCII digit 0, representing the integer zero, or consists of an ASCII digit from 1 to 9 optionally followed by one or more ASCII digits from 0 to 9 interspersed with underscores, representing a positive integer.

\[
\begin{align*}
\text{DecimalNumeral:} \\
&\quad 0 \\
&\quad \text{NonZeroDigit} \ [\text{Digits}] \\
&\quad \text{NonZeroDigit Underscores Digits} \\
\text{NonZeroDigit:} \\
&\quad \text{(one of)} \\
&\quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \\
\text{Digits:} \\
&\quad \text{Digit} \\
&\quad \text{Digit} \ [\text{DigitsAndUnderscores}] \ \text{Digit} \\
\text{Digit:} \\
&\quad 0 \\
&\quad \text{NonZeroDigit} \\
\text{DigitsAndUnderscores:} \\
&\quad \text{DigitOrUnderscore} \ \{\text{DigitOrUnderscore}\} \\
\text{DigitOrUnderscore:} \\
&\quad \text{Digit} \\
&\quad _{} \\
\text{Underscores:} \\
&\quad _{} \{\_\} 
\end{align*}
\]
A hexadecimal numeral consists of the leading ASCII characters 0x or 0X followed by one or more ASCII hexadecimal digits interspersed with underscores, and can represent a positive, zero, or negative integer.

Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

\[
\text{HexNumeral:}\n\begin{align*}
0 \times \text{HexDigits} \\
0 \times \text{HexDigits}
\end{align*}
\]

\[
\text{HexDigits:}\n\begin{align*}
\text{HexDigit} \\
\text{HexDigit} [\text{HexDigitsAndUnderscores}] \text{HexDigit}
\end{align*}
\]

\[
\text{HexDigit:}\n\begin{align*}
\text{(one of)} \\
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
\end{align*}
\]

\[
\text{HexDigitsAndUnderscores:}\n\begin{align*}
\text{HexDigitOrUnderscore} \{\text{HexDigitOrUnderscore}\}
\end{align*}
\]

\[
\text{HexDigitOrUnderscore:}\n\begin{align*}
\text{HexDigit} \\
-
\end{align*}
\]

The HexDigit production above comes from §3.3.
An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 interspersed with underscores, and can represent a positive, zero, or negative integer.

OctalNumeral:
  0 OctalDigits
  0 Underscores OctalDigits

OctalDigits:
  OctalDigit
  OctalDigit [OctalDigitsAndUnderscores] OctalDigit

OctalDigit:
  (one of)
  0 1 2 3 4 5 6 7

OctalDigitsAndUnderscores:
  OctalDigitOrUnderscore {OctalDigitOrUnderscore}

OctalDigitOrUnderscore:
  OctalDigit
  _

Note that octal numerals always consist of two or more digits, as 0 alone is always considered to be a decimal numeral - not that it matters much in practice, for the numerals 0, 00, and 0×0 all represent exactly the same integer value.
A binary numeral consists of the leading ASCII characters \(0\text{b}\) or \(0\text{B}\) followed by one or more of the ASCII digits 0 or 1 interspersed with underscores, and can represent a positive, zero, or negative integer.

\[
\text{BinaryNumeral:}
\begin{align*}
&0\text{b} \text{ BinaryDigits} \\
&0\text{B} \text{ BinaryDigits}
\end{align*}
\]

\[
\text{BinaryDigits:}
\begin{align*}
&\text{BinaryDigit} \\
&\text{BinaryDigit} [\text{BinaryDigitsAndUnderscores}] \text{ BinaryDigit}
\end{align*}
\]

\[
\text{BinaryDigit:}
\begin{align*}
&(\text{one of}) \\
&0 \ 1
\end{align*}
\]

\[
\text{BinaryDigitsAndUnderscores:}
\begin{align*}
&\text{BinaryDigitOrUnderscore} \ \{\text{BinaryDigitOrUnderscore}\}
\end{align*}
\]

\[
\text{BinaryDigitOrUnderscore:}
\begin{align*}
&\text{BinaryDigit} \\
&\
\end{align*}
\]
The largest decimal literal of type int is $2^{31}$. All decimal literals from 0 to $2^{31}$ may appear anywhere an int literal may appear. The decimal literal $2^{31}$ may appear only as the operand of the unary minus operator - (§15.15.4).

It is a compile-time error if the decimal literal $2^{31}$ appears anywhere other than as the operand of the unary minus operator; or if a decimal literal of type int is larger than $2^{31}$.

The largest positive hexadecimal, octal, and binary literals of type int - each of which represents the decimal value $2^{31} - 1$ - are respectively:

- $0x7fff_ffff$,
- $0177_7777_7777$ and
- $0b0111_1111_1111_1111_1111_1111_1111_1111$

The most negative hexadecimal, octal, and binary literals of type int - each of which represents the decimal value $-2^{31}$ - are respectively:

- $0x8000_0000$,
- $0200_0000_0000$ and
- $0b1000_0000_0000_0000_0000_0000_0000_0000$

The following hexadecimal, octal, and binary literals represent the decimal value $-1$:

- $0xffff_ffff$,
- $0377_7777_7777$ and
- $0b1111_1111_1111_1111_1111_1111_1111_1111$

It is a compile-time error if a hexadecimal, octal, or binary int literal does not fit in 32 bits.

The largest decimal literal of type long is $2^{63}$. All decimal literals from 0L to $2^{63}$ may appear anywhere a long literal may appear. The decimal literal $2^{63}$ may appear only as the operand of the unary minus operator - (§15.15.4).

It is a compile-time error if the decimal literal $2^{63}$ appears anywhere other than as the operand of the unary minus operator; or if a decimal literal of type long is larger than $2^{63}$. 
The largest positive hexadecimal, octal, and binary literals of type `long` - each of which represents the decimal value $9223372036854775807L \left(2^{63}-1\right)$ - are respectively:

- $0x7fff_{-}ffff_{-}ffff_{-}ffffL$,
- $07_{-}7777_{-}7777_{-}7777_{-}7777L$, and
- $0b0111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111L$

The most negative hexadecimal, octal, and binary literals of type `long` - each of which represents the decimal value $-9223372036854775808L \left(-2^{63}\right)$ - are respectively:

- $0x8000_{-}0000_{-}0000_{-}0000_{-}0000L$, and
- $010_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000L$, and
- $0b1000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000_{-}0000L$

The following hexadecimal, octal, and binary literals represent the decimal value $-1L$:

- $0xffff_{-}ffff_{-}ffff_{-}ffffL$,
- $017_{-}7777_{-}7777_{-}7777_{-}7777L$, and
- $0b1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111_{-}1111L$

It is a compile-time error if a hexadecimal, octal, or binary `long` literal does not fit in 64 bits.

Examples of `int` literals:

```
0  2  0372  0xDada_Cafe  1996  0x00_FF__00_FF
```

Examples of `long` literals:

```
01  0777L  0x100000000L  2_147_483_648L  0xC0B0L
```

### 3.10.2 Floating-Point Literals

A *floating-point literal* has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fraction part, an exponent, and a type suffix.

A floating-point literal may be expressed in decimal (base 10) or hexadecimal (base 16).
For decimal floating-point literals, at least one digit (in either the whole number or the fraction part) and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

For hexadecimal floating-point literals, at least one digit is required (in either the whole number or the fraction part), and the exponent is mandatory, and the float type suffix is optional. The exponent is indicated by the ASCII letter p or P followed by an optionally signed integer.

Underscores are allowed as separators between digits that denote the whole-number part, and between digits that denote the fraction part, and between digits that denote the exponent.

```
FloatingPointLiteral:
  DecimalFloatingPointLiteral
  HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
  Digits . [Digits] [ExponentPart] [FloatTypeSuffix]
  . Digits [ExponentPart] [FloatTypeSuffix]
  Digits ExponentPart [FloatTypeSuffix]
  Digits [ExponentPart] FloatTypeSuffix

ExponentPart:
  ExponentIndicator SignedInteger

ExponentIndicator:
  (one of)
  e E

SignedInteger:
  [Sign] Digits

Sign:
  (one of)
  + –

FloatTypeSuffix:
  (one of)
  f F d D
```
HexadecimalFloatingPointLiteral:
HexSignificand BinaryExponent [FloatTypeSuffix]

HexSignificand:
HexNumeral [ . ]
0 x [HexDigits] . HexDigits
0 x [HexDigits] . HexDigits

BinaryExponent:
BinaryExponentIndicator SignedInteger

BinaryExponentIndicator:
(one of)
p P

A floating-point literal is of type float if it is suffixed with an ASCII letter f or f; otherwise its type is double and it can optionally be suffixed with an ASCII letter D or d (§4.2.3).

The elements of the types float and double are those values that can be represented using the IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point formats, respectively.

The details of proper input conversion from a Unicode string representation of a floating-point number to the internal IEEE 754 binary floating-point representation are described for the methods valueOf of class Float and class Double of the package java.lang.

The largest positive finite literal of type float is \( 3.4028235e38f \).
The smallest positive finite non-zero literal of type float is \( 1.40e-45f \).
The largest positive finite literal of type double is \( 1.7976931348623157e308 \).
The smallest positive finite non-zero literal of type double is \( 4.9e-324 \).

It is a compile-time error if a non-zero floating-point literal is too large, so that on rounded conversion to its internal representation, it becomes an IEEE 754 infinity.

A program can represent infinities without producing a compile-time error by using constant expressions such as \( 1f/0f \) or \( -1d/0d \) or by using the predefined constants POSITIVE_INFINITY and NEGATIVE_INFINITY of the classes Float and Double.

It is a compile-time error if a non-zero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero.
A compile-time error does not occur if a non-zero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a non-zero denormalized number.

Predefined constants representing Not-a-Number values are defined in the classes `Float` and `Double` as `Float.NaN` and `Double.NaN`.

Examples of `float` literals:

```
1e1f   2.f   .3f   0f   3.14f   6.022137e+23f
```

Examples of `double` literals:

```
1e1    2.    .3    0.0    3.14    1e-9d    1e137
```

### 3.10.3 Boolean Literals

The `boolean` type has two values, represented by the `boolean literals` `true` and `false`, formed from ASCII letters.

```
BooleanLiteral:
    (one of)
    true false
```

A boolean literal is always of type `boolean` (§4.2.5).

### 3.10.4 Character Literals

A `character literal` is expressed as a character or an escape sequence (§3.10.6), enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is `\u0027`.)

```
CharacterLiteral:
    ' SingleCharacter ' 
    ' EscapeSequence ' 

SingleCharacter:
    InputCharacter but not ' or \ 
```

See §3.10.6 for the definition of `EscapeSequence`.

Character literals can only represent UTF-16 code units (§3.1), i.e., they are limited to values from `\u0000` to `\uffff`. Supplementary characters must be represented
either as a surrogate pair within a `char` sequence, or as an integer, depending on the API they are used with.

A character literal is always of type `char` (§4.2.1).

It is a compile-time error for the character following the `SingleCharacter` or `EscapeSequence` to be other than a `'`.

It is a compile-time error for a line terminator (§3.4) to appear after the opening `'` and before the closing `'`.

As specified in §3.4, the characters CR and LF are never an `InputCharacter`; each is recognized as constituting a `LineTerminator`.

The following are examples of `char` literals:

- `'a`
- `'\'
- `'\t`
- `'\n`
- `'\'`
- `'\u03a9`
- `'\uFFFF`
- `'\177`
- `'™`

Because Unicode escapes are processed very early, it is not correct to write `'\u000a'` for a character literal whose value is linefeed (LF); the Unicode escape `\u000a` is transformed into an actual linefeed in translation step 1 (§3.3) and the linefeed becomes a `LineTerminator` in step 2 (§3.4), and so the character literal is not valid in step 3. Instead, one should use the escape sequence `\n` (§3.10.6). Similarly, it is not correct to write `'\u000d'` for a character literal whose value is carriage return (CR). Instead, use `'\r'`.

In C and C++, a character literal may contain representations of more than one character, but the value of such a character literal is implementation-defined. In the Java programming language, a character literal always represents exactly one character.

### 3.10.5 String Literals

A string literal consists of zero or more characters enclosed in double quotes. Characters may be represented by escape sequences (§3.10.6) - one escape sequence for characters in the range U+0000 to U+FFFF, two escape sequences for the UTF-16 surrogate code units of characters in the range U+010000 to U+10FFFF.
**StringLiteral:**
" {StringCharacter} "

**StringCharacter:**
InputCharacter but not " or \
EscapeSequence

See §3.10.6 for the definition of EscapeSequence.

A string literal is always of type String (§4.3.3).

It is a compile-time error for a line terminator to appear after the opening " and before the closing matching ".

As specified in §3.4, the characters CR and LF are never an InputCharacter; each is recognized as constituting a LineTerminator.

A long string literal can always be broken up into shorter pieces and written as a (possibly parenthesized) expression using the string concatenation operator + (§15.18.1).

The following are examples of string literals:

```
""                    // the empty string
"\""                  // a string containing " alone
"This is a string"    // a string containing 16 characters
"This is a " +        // actually a string-valued constant expression, 
  "two-line string"    // formed from two string literals
```

Because Unicode escapes are processed very early, it is not correct to write "\u000a" for a string literal containing a single linefeed (LF): the Unicode escape \u000a is transformed into an actual linefeed in translation step 1 (§3.3) and the linefeed becomes a LineTerminator in step 2 (§3.4), and so the string literal is not valid in step 3. Instead, one should write "\n" (§3.10.6). Similarly, it is not correct to write "\u000d" for a string literal containing a single carriage return (CR). Instead, use "\r". Finally, it is not possible to write "\u0022" for a string literal containing a double quotation mark (").

A string literal is a reference to an instance of class String (§4.3.1, §4.3.3).

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern (§12.5).

**Example 3.10.5-1. String Literals**

The program consisting of the compilation unit (§7.3):

```java
package testPackage;
```
This example illustrates six points:

- String literals in the same class and package represent references to the same `String` object (§4.3.1).
- String literals in different classes in the same package represent references to the same `String` object.
- String literals in different classes in different packages likewise represent references to the same `String` object.
- Strings concatenated from constant expressions (§15.28) are computed at compile time and then treated as if they were literals.
- Strings computed by concatenation at run time are newly created and therefore distinct.
- The result of explicitly interning a computed string is the same `String` object as any pre-existing string literal with the same contents.

### 3.10.6 Escape Sequences for Character and String Literals

The character and string *escape sequences* allow for the representation of some nongraphic characters without using Unicode escapes, as well as the single quote, double quote, and backslash characters, in character literals (§3.10.4) and string literals (§3.10.5).
**Lexical Structure**

**3.10 Literals**

**EscapeSequence**:
- \b (backspace BS, Unicode \u0008)
- \t (horizontal tab HT, Unicode \u0009)
- \n (linefeed LF, Unicode \u000a)
- \f (form feed FF, Unicode \u000c)
- \r (carriage return CR, Unicode \u000d)
- " (double quote " , Unicode \u0022)
- ' (single quote ' , Unicode \u0027)
- \ (backslash \ , Unicode \u005c)

**OctalEscape** (octal value, Unicode \u0000 to \u00ff)

**OctalEscape**:
- \ OctalDigit
- \ OctalDigit OctalDigit
- \ ZeroToThree OctalDigit OctalDigit

**OctalDigit**:
(one of)
0 1 2 3 4 5 6 7

**ZeroToThree**:
(one of)
0 1 2 3

The **OctalDigit** production above comes from §3.10.1.

It is a compile-time error if the character following a backslash in an escape sequence is not an ASCII b, t, n, f, r, " , ' , \ , 0, 1, 2, 3, 4, 5, 6, or 7. The Unicode escape \u is processed earlier (§3.3).

Octal escapes are provided for compatibility with C, but can express only Unicode values \u0000 through \u00ff, so Unicode escapes are usually preferred.

### 3.10.7 The Null Literal

The null type has one value, the null reference, represented by the **null literal** null, which is formed from ASCII characters.

**NullLiteral**:

null

A null literal is always of the null type (§4.1).
3.11 Separators

Twelve tokens, formed from ASCII characters, are the separators (punctuators).

Separator:
(one of)
(   )   {   }   [   ]   ;   ,   .   . . . . .   @   ::

3.12 Operators

38 tokens, formed from ASCII characters, are the operators.

Operator:
(one of)
=   >   <   !   ~   ?   :   ->
==  >=  <=  !=  &&  ||  ++  --
+   -   *   /   &   |   ^   %   <<   >>   >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
CHAPTER 4

Types, Values, and Variables

The Java programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time.

The Java programming language is also a strongly typed language, because types limit the values that a variable (§4.12) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong static typing helps detect errors at compile time.

The types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types (§4.2) are the boolean type and the numeric types. The numeric types are the integral types byte, short, int, long, and char, and the floating-point types float and double. The reference types (§4.3) are class types, interface types, and array types. There is also a special null type. An object (§4.3.1) is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects. All objects, including arrays, support the methods of class Object (§4.3.2). String literals are represented by String objects (§4.3.3).

4.1 The Kinds of Types and Values

There are two kinds of types in the Java programming language: primitive types (§4.2) and reference types (§4.3). There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values (§4.2) and reference values (§4.3).

Type:

PrimitiveType
ReferenceType
There is also a special *null type*, the type of the expression `null` (§3.10.7, §15.8.1), which has no name.

Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type.

The null reference is the only possible value of an expression of null type.

The null reference can always be assigned or cast to any reference type (§5.2, §5.3, §5.5).

In practice, the programmer can ignore the null type and just pretend that `null` is merely a special literal that can be of any reference type.

### 4.2 Primitive Types and Values

A primitive type is predefined by the Java programming language and named by its reserved keyword (§3.9):

```
PrimitiveType:  
  {Annotation} NumericType  
  {Annotation} boolean  

NumericType:  
  IntegralType  
  FloatingPointType

IntegralType:  
  (one of)  
  byte short int long char

FloatingPointType:  
  (one of)  
  float double
```

Primitive values do not share state with other primitive values.

The *numeric types* are the integral types and the floating-point types.

The *integral types* are `byte`, `short`, `int`, and `long`, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and `char`, whose values are 16-bit unsigned integers representing UTF-16 code units (§3.1).
The floating-point types are float, whose values include the 32-bit IEEE 754 floating-point numbers, and double, whose values include the 64-bit IEEE 754 floating-point numbers.

The boolean type has exactly two values: true and false.

4.2.1 Integral Types and Values

The values of the integral types are integers in the following ranges:

- For byte, from -128 to 127, inclusive
- For short, from -32768 to 32767, inclusive
- For int, from -2147483648 to 2147483647, inclusive
- For long, from -9223372036854775808 to 9223372036854775807, inclusive
- For char, from ' \u0000 ' to ' \uffff ' inclusive, that is, from 0 to 65535

4.2.2 Integer Operations

The Java programming language provides a number of operators that act on integral values:

- The comparison operators, which result in a value of type boolean:
  - The numerical comparison operators <, <=, >, and >= (§15.20.1)
  - The numerical equality operators == and != (§15.21.1)
- The numerical operators, which result in a value of type int or long:
  - The unary plus and minus operators + and - (§15.15.3, §15.15.4)
  - The multiplicative operators *, /, and % (§15.17)
  - The additive operators + and - (§15.18)
  - The increment operator ++, both prefix (§15.15.1) and postfix (§15.14.2)
  - The decrement operator --, both prefix (§15.15.2) and postfix (§15.14.3)
  - The signed and unsigned shift operators <<, >>, and >>> (§15.19)
  - The bitwise complement operator ~ (§15.15.5)
  - The integer bitwise operators &, ^, and | (§15.22.1)
- The conditional operator ?: (§15.25)
• The cast operator (§15.16), which can convert from an integral value to a value of any specified numeric type

• The string concatenation operator + (§15.18.1), which, when given a String operand and an integral operand, will convert the integral operand to a String (the decimal form of a byte, short, int, or long operand, or the character of a char operand), and then produce a newly created String that is the concatenation of the two strings

Other useful constructors, methods, and constants are predefined in the classes Byte, Short, Integer, Long, and Character.

If an integer operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long, it is first widened (§5.1.5) to type long by numeric promotion (§5.6).

Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If either operand is not an int, it is first widened to type int by numeric promotion.

Any value of any integral type may be cast to or from any numeric type. There are no casts between integral types and the type boolean.

See §4.2.5 for an idiom to convert integer expressions to boolean.

The integer operators do not indicate overflow or underflow in any way.

An integer operator can throw an exception (§11 (Exceptions)) for the following reasons:

• Any integer operator can throw a NullPointerException if unboxing conversion (§5.1.8) of a null reference is required.

• The integer divide operator / (§15.17.2) and the integer remainder operator % (§15.17.3) can throw an ArithmeticException if the right-hand operand is zero.

• The increment and decrement operators ++ (§15.14.2, §15.15.1) and -- (§15.14.3, §15.15.2) can throw an OutOfMemoryError if boxing conversion (§5.1.7) is required and there is not sufficient memory available to perform the conversion.

Example 4.2.2-1. Integer Operations

class Test {
    public static void main(String[] args) {
        int i = 1000000;
        System.out.println(i * i);
This program produces the output:

\[-727379968\]
\[1000000000000\]

and then encounters an `ArithmeticException` in the division by \( l - i \), because \( l - i \) is zero. The first multiplication is performed in 32-bit precision, whereas the second multiplication is a long multiplication. The value \(-727379968\) is the decimal value of the low 32 bits of the mathematical result, \(1000000000000\), which is a value too large for type `int`.

### 4.2.3 Floating-Point Types, Formats, and Values

The floating-point types are `float` and `double`, which are conceptually associated with the single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative numbers that consist of a sign and magnitude, but also positive and negative zeros, positive and negative *infinities*, and special *Not-a-Number* values (hereafter abbreviated NaN). A NaN value is used to represent the result of certain invalid operations such as dividing zero by zero. NaN constants of both `float` and `double` type are predefined as `Float.NaN` and `Double.NaN`.

Every implementation of the Java programming language is required to support two standard sets of floating-point values, called the `float value set` and the `double value set`. In addition, an implementation of the Java programming language may support either or both of two extended-exponent floating-point value sets, called the `float-extended-exponent value set` and the `double-extended-exponent value set`. These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of expressions of type `float` or `double` (§5.1.13, §15.4).

The finite nonzero values of any floating-point value set can all be expressed in the form \( s \cdot m \cdot 2^{(e - N + 1)} \), where \( s \) is +1 or -1, \( m \) is a positive integer less than \( 2^N \), and \( e \) is an integer between \( E_{\min} = -(2^{K-1}-2) \) and \( E_{\max} = 2^{K-1}-1 \), inclusive, and where \( N \) and \( K \) are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, supposing that a
value \( v \) in a value set might be represented in this form using certain values for \( s, m, \) and \( e \), then if it happened that \( m \) were even and \( e \) were less than \( 2^{K-1} \), one could halve \( m \) and increase \( e \) by 1 to produce a second representation for the same value \( v \). A representation in this form is called *normalized* if \( m \geq 2^{N-1} \); otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that \( m \geq 2^{N-1} \), then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters \( N \) and \( K \) (and on the derived parameters \( E_{\text{min}} \) and \( E_{\text{max}} \)) for the two required and two optional floating-point value sets are summarized in Table 4.2.3-A.

**Table 4.2.3-A. Floating-point value set parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>float</th>
<th>float-extended-exponent</th>
<th>double</th>
<th>double-extended-exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>24</td>
<td>24</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>( K )</td>
<td>8</td>
<td>( \geq 11 )</td>
<td>11</td>
<td>( \geq 15 )</td>
</tr>
<tr>
<td>( E_{\text{max}} )</td>
<td>+127</td>
<td>( \geq +1023 )</td>
<td>+1023</td>
<td>( \geq +16383 )</td>
</tr>
<tr>
<td>( E_{\text{min}} )</td>
<td>-126</td>
<td>( \leq -1022 )</td>
<td>-1022</td>
<td>( \leq -16382 )</td>
</tr>
</tbody>
</table>

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant \( K \), whose value is constrained by Table 4.2.3-A; this value \( K \) in turn dictates the values for \( E_{\text{min}} \) and \( E_{\text{max}} \).

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also NaN values and the four values positive zero, negative zero, positive infinity, and negative infinity.

Note that the constraints in Table 4.2.3-A are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range of exponent values than the corresponding standard value set, but does not have more precision.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard. The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard. Note, however,
that the elements of the float-extended-exponent and double-extended-exponent value sets defined here do \textit{not} correspond to the values that can be represented using IEEE 754 single extended and double extended formats, respectively.

The float, float-extended-exponent, double, and double-extended-exponent value sets are not types. It is always correct for an implementation of the Java programming language to use an element of the float value set to represent a value of type \texttt{float}; however, it may be permissible in certain regions of code for an implementation to use an element of the float-extended-exponent value set instead. Similarly, it is always correct for an implementation to use an element of the double value set to represent a value of type \texttt{double}; however, it may be permissible in certain regions of code for an implementation to use an element of the double-extended-exponent value set instead.

Except for NaN, floating-point values are \textit{ordered}; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, positive and negative zero, positive finite nonzero values, and positive infinity.

IEEE 754 allows multiple distinct NaN values for each of its single and double floating-point formats. While each hardware architecture returns a particular bit pattern for NaN when a new NaN is generated, a programmer can also create NaNs with different bit patterns to encode, for example, retrospective diagnostic information.

For the most part, the Java SE Platform treats NaN values of a given type as though collapsed into a single canonical value, and hence this specification normally refers to an arbitrary NaN as though to a canonical value.

However, version 1.3 of the Java SE Platform introduced methods enabling the programmer to distinguish between NaN values: the \texttt{Float.floatToRawIntBits} and \texttt{Double.doubleToRawLongBits} methods. The interested reader is referred to the specifications for the \texttt{Float} and \texttt{Double} classes for more information.

Positive zero and negative zero compare equal; thus the result of the expression \texttt{0.0==-0.0} is \texttt{true} and the result of \texttt{0.0>-0.0} is \texttt{false}. But other operations can distinguish positive and negative zero; for example, \texttt{1.0/0.0} has the value positive infinity, while the value of \texttt{1.0/-0.0} is negative infinity.

NaN is \textit{unordered}, so:

- The numerical comparison operators \texttt{<}, \texttt{<=}, \texttt{>}, and \texttt{>=} return \texttt{false} if either or both operands are NaN (§15.20.1).

  In particular, \texttt{(x<y) == !(x>=y)} will be \texttt{false} if \texttt{x} or \texttt{y} is NaN.

- The equality operator \texttt{==} returns \texttt{false} if either operand is NaN.
• The inequality operator != returns true if either operand is NaN (§15.21.1).
  In particular, x!=x is true if and only if x is NaN.

4.2.4 Floating-Point Operations

The Java programming language provides a number of operators that act on floating-point values:

• The comparison operators, which result in a value of type boolean:
  – The numerical comparison operators <, <=, >, and >= (§15.20.1)
  – The numerical equality operators == and != (§15.21.1)

• The numerical operators, which result in a value of type float or double:
  – The unary plus and minus operators + and - (§15.15.3, §15.15.4)
  – The multiplicative operators *, /, and % (§15.17)
  – The additive operators + and - (§15.18.2)
  – The increment operator ++, both prefix (§15.15.1) and postfix (§15.14.2)
  – The decrement operator --, both prefix (§15.15.2) and postfix (§15.14.3)

• The conditional operator ? : (§15.25)

• The cast operator (§15.16), which can convert from a floating-point value to a value of any specified numeric type

• The string concatenation operator + (§15.18.1), which, when given a String operand and a floating-point operand, will convert the floating-point operand to a String representing its value in decimal form (without information loss), and then produce a newly created String by concatenating the two strings

Other useful constructors, methods, and constants are predefined in the classes Float, Double, and Math.

If at least one of the operands to a binary operator is of floating-point type, then the operation is a floating-point operation, even if the other is integral.

If at least one of the operands to a numerical operator is of type double, then the operation is carried out using 64-bit floating-point arithmetic, and the result of the numerical operator is a value of type double. If the other operand is not a double, it is first widened (§5.1.5) to type double by numeric promotion (§5.6).
Otherwise, the operation is carried out using 32-bit floating-point arithmetic, and
the result of the numerical operator is a value of type \texttt{float}. (If the other operand
is not a \texttt{float}, it is first widened to type \texttt{float} by numeric promotion.)

Any value of a floating-point type may be cast to or from any numeric type. There
are no casts between floating-point types and the type \texttt{boolean}.

See §4.2.5 for an idiom to convert floating-point expressions to \texttt{boolean}.

Operators on floating-point numbers behave as specified by IEEE 754 (with
the exception of the remainder operator (§15.17.3)). In particular, the Java
programming language requires support of IEEE 754 denormalized floating-point
numbers and gradual underflow, which make it easier to prove desirable properties
of particular numerical algorithms. Floating-point operations do not "flush to zero"
if the calculated result is a denormalized number.

The Java programming language requires that floating-point arithmetic behave
as if every floating-point operator rounded its floating-point result to the result
precision. Inexact results must be rounded to the representable value nearest to the
infinitely precise result; if the two nearest representable values are equally near,
the one with its least significant bit zero is chosen. This is the IEEE 754 standard’s
default rounding mode known as round to nearest.

The Java programming language uses round toward zero when converting a
floating value to an integer (§5.1.3), which acts, in this case, as though the number
were truncated, discarding the mantissa bits. Rounding toward zero chooses as its
result the format’s value closest to and no greater in magnitude than the infinitely
precise result.

A floating-point operation that overflows produces a signed infinity.

A floating-point operation that underflows produces a denormalized value or a
signed zero.

A floating-point operation that has no mathematically definite result produces NaN.

All numeric operations with NaN as an operand produce NaN as a result.

A floating-point operator can throw an exception (§11 (Exceptions)) for the
following reasons:

- Any floating-point operator can throw a \texttt{NullPointerException} if unboxing
  conversion (§5.1.8) of a null reference is required.

- The increment and decrement operators \texttt{++} (§15.14.2, §15.15.1) and \texttt{--}
  (§15.14.3, §15.15.2) can throw an \texttt{OutOfMemoryError} if boxing conversion
Example 4.2.4-1. Floating-point Operations

class Test {
    public static void main(String[] args) {
        // An example of overflow:
        double d = 1e308;
        System.out.print("overflow produces infinity: ");
        System.out.println(d + "*10==" + d*10);
        // An example of gradual underflow:
        d = 1e-305 * Math.PI;
        System.out.print("gradual underflow: " + d + "\n ");
        for (int i = 0; i < 4; i++)
            System.out.print(" " + (d /= 100000));
        System.out.println();
        // An example of NaN:
        System.out.print("0.0/0.0 is Not-a-Number: ");
        d = 0.0/0.0;
        System.out.println(d);
        // An example of inexact results and rounding:
        System.out.print("inexact results with float:");
        for (int i = 0; i < 100; i++) {
            float z = 1.0f / i;
            if (z * i != 1.0f)
                System.out.print(" " + i);
        }
        System.out.println();
        // Another example of inexact results and rounding:
        System.out.print("inexact results with double:");
        for (int i = 0; i < 100; i++) {
            double z = 1.0 / i;
            if (z * i != 1.0)
                System.out.print(" " + i);
        }
        System.out.println();
        // An example of cast to integer rounding:
        System.out.print("cast to int rounds toward 0: ");
        d = 12345.6;
        System.out.println((int)d + " " + (int)(-d));
    }
}

This program produces the output:
overflow produces infinity: 1.0E308*10==Infinity
gradual underflow: 3.141592653589793E-305
   3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
0.0/0.0 is Not-a-Number: NaN
inexact results with float: 0 41 47 55 61 82 83 94 97
inexact results with double: 0 49 98
cast to int rounds toward 0: 12345 -12345

This example demonstrates, among other things, that gradual underflow can result in a gradual loss of precision.

The results when i is 0 involve division by zero, so that z becomes positive infinity, and z * 0 is NaN, which is not equal to 1.0.

4.2.5 The boolean Type and boolean Values

The boolean type represents a logical quantity with two possible values, indicated by the literals true and false (§3.10.3).

The boolean operators are:

- The relational operators == and != (§15.21.2)
- The logical complement operator ! (§15.15.6)
- The logical operators &, ^, and | (§15.22.2)
- The conditional-and and conditional-or operators && (§15.23) and || (§15.24)
- The conditional operator ? : (§15.25)
- The string concatenation operator + (§15.18.1), which, when given a String operand and a boolean operand, will convert the boolean operand to a String (either "true" or "false"), and then produce a newly created String that is the concatenation of the two strings

Boolean expressions determine the control flow in several kinds of statements:

- The if statement (§14.9)
- The while statement (§14.12)
- The do statement (§14.13)
- The for statement (§14.14)

A boolean expression also determines which subexpression is evaluated in the conditional ? : operator (§15.25).

Only boolean and Boolean expressions can be used in control flow statements and as the first operand of the conditional operator ? :.
An integer or floating-point expression $x$ can be converted to a boolean value, following the C language convention that any nonzero value is true, by the expression $x!=0$.

An object reference $obj$ can be converted to a boolean value, following the C language convention that any reference other than null is true, by the expression $obj!=null$.

A boolean value can be converted to a String by string conversion (§5.4).

A boolean value may be cast to type boolean, Boolean, or Object (§5.5). No other casts on type boolean are allowed.

### 4.3 Reference Types and Values

There are four kinds of reference types: class types (§8.1), interface types (§9.1), type variables (§4.4), and array types (§10.1).

\[
\text{ReferenceType}:
\begin{align*}
&\text{ClassOrInterfaceType} \\
&\text{TypeVariable} \\
&\text{ArrayType}
\end{align*}
\]

\[
\text{ClassOrInterfaceType}:
\begin{align*}
&\text{ClassType} \\
&\text{InterfaceType}
\end{align*}
\]

\[
\text{ClassType}:
\begin{align*}
&\{\text{Annotation}\} \text{TypeIdentifier} \ [\text{TypeArguments}] \\
&\text{PackageName} . \{\text{Annotation}\} \text{TypeIdentifier} \ [\text{TypeArguments}] \\
&\text{ClassOrInterfaceType} . \{\text{Annotation}\} \text{TypeIdentifier} \ [\text{TypeArguments}]
\end{align*}
\]

\[
\text{InterfaceType}:
\begin{align*}
&\text{ClassType}
\end{align*}
\]

\[
\text{TypeVariable}:
\begin{align*}
&\{\text{Annotation}\} \text{TypeIdentifier}
\end{align*}
\]

\[
\text{ArrayType}:
\begin{align*}
&\text{PrimitiveType} \ Dims \\
&\text{ClassOrInterfaceType} \ Dims \\
&\text{TypeVariable} \ Dims
\end{align*}
\]
Dims:

[Annotation] [ ] {[Annotation] [ ]}

The sample code:

```java
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

declarates a class type `Point`, an interface type `Move`, and uses an array type `int[]` (an array of `int`) to declare the field `metrics` of the class `Point`.

A class or interface type consists of an identifier or a dotted sequence of identifiers, where each identifier is optionally followed by type arguments (§4.5.1). If type arguments appear anywhere in a class or interface type, it is a parameterized type (§4.5).

Each identifier in a class or interface type is classified as a package name or a type name (§6.5.1). Identifiers which are classified as type names may be annotated. If a class or interface type has the form `T.id` (optionally followed by type arguments), then `id` must be the simple name of an accessible member type of `T` (§6.6, §8.5, §9.5), or a compile-time error occurs. The class or interface type denotes that member type.

### 4.3.1 Objects

An object is a class instance or an array.

The reference values (often just references) are pointers to these objects, and a special null reference, which refers to no object.

A class instance is explicitly created by a class instance creation expression (§15.9).

An array is explicitly created by an array creation expression (§15.10.1).

Other expressions may implicitly create a class instance (§12.5) or an array (§10.6).

**Example 4.3.1-1. Object Creation**

```java
class Point {
    int x, y;
    Point() { System.out.println("default"); }
    Point(int x, int y) { this.x = x; this.y = y; }

    /* A Point instance is explicitly created at class initialization time: */
    static Point origin = new Point(0,0);
```
4.3 Reference Types and Values  TYPES, VALUES, AND VARIABLES

    /* A String can be implicitly created
       by a + operator: */
    public String toString() { return "(" + x + "," + y + ")"; }

    class Test {
        public static void main(String[] args) {
            /* A Point is explicitly created
               using newInstance: */
            Point p = null;
            try {
                p = (Point)Class.forName("Point").newInstance();
            } catch (Exception e) {
                System.out.println(e);
            }

            /* An array is implicitly created
               by an array constructor: */
            Point a[] = { new Point(0,0), new Point(1,1) };

            /* Strings are implicitly created
               by + operators: */
            System.out.println("p: " + p);
            System.out.println("a: { " + a[0] + ", " + a[1] + " }");

            /* An array is explicitly created
               by an array creation expression: */
            String sa[] = new String[2];
            sa[0] = "he"; sa[1] = "llol";
            System.out.println(sa[0] + sa[1]);
        }
    }

    This program produces the output:

    default
    p: (0,0)
    a: { (0,0), (1,1) }
    hello

The operators on references to objects are:

- Field access, using either a qualified name (§6.6) or a field access expression (§15.11)
- Method invocation (§15.12)
- The cast operator (§5.5, §15.16)
- The string concatenation operator + (§15.18.1), which, when given a String
  operand and a reference, will convert the reference to a String by invoking the
toString method of the referenced object (using "null" if either the reference
or the result of `toString` is a null reference), and then will produce a newly created `String` that is the concatenation of the two strings

- The `instanceof` operator (§15.20.2)
- The reference equality operators `==` and `!=` (§15.21.3)
- The conditional operator `?:` (§15.25).

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.

**Example 4.3.1-2. Primitive and Reference Identity**

```java
class Value { int val; }
class Test {
    public static void main(String[] args) {
        int i1 = 3;
        int i2 = i1;
        i2 = 4;
        System.out.print("i1==" + i1);
        System.out.println(" but i2==" + i2);
        Value v1 = new Value();
        v1.val = 5;
        Value v2 = v1;
        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" and v2.val==" + v2.val);
    }
}
```

This program produces the output:

```
i1==3 but i2==4
v1.val==5 and v2.val==6
```

because `v1.val` and `v2.val` reference the same instance variable (§4.12.3) in the one `Value` object created by the only `new` expression, while `i1` and `i2` are different variables.

Each object is associated with a monitor (§17.1), which is used by synchronized methods (§8.4.3) and the synchronized statement (§14.19) to provide control over concurrent access to state by multiple threads (§17 (Threads and Locks)).
4.3.2 The Class Object

The class Object is a superclass (§8.1.4) of all other classes.

All class and array types inherit (§8.4.8) the methods of class Object, which are summarized as follows:

- The method clone is used to make a duplicate of an object.
- The method equals defines a notion of object equality, which is based on value, not reference, comparison.
- The method finalize is run just before an object is destroyed (§12.6).
- The method getClass returns the Class object that represents the class of the object.

A Class object exists for each reference type. It can be used, for example, to discover the fully qualified name of a class, its members, its immediate superclass, and any interfaces that it implements.

The type of a method invocation expression of getClass is Class<? extends T|T>, where T is the class or interface that was searched for getClass (§15.12.1) and |T| denotes the erasure of T (§4.6).

A class method that is declared synchronized (§8.4.3.6) synchronizes on the monitor associated with the Class object of the class.

- The method hashCode is very useful, together with the method equals, in hashtables such as java.util.HashMap.
- The methods wait, notify, and notifyAll are used in concurrent programming using threads (§17.2).
- The method toString returns a String representation of the object.

4.3.3 The Class String

Instances of class String represent sequences of Unicode code points.

A String object has a constant (unchanging) value.

String literals (§3.10.5) are references to instances of class String.

The string concatenation operator + (§15.18.1) implicitly creates a new String object when the result is not a constant expression (§15.28).
4.3.4 When Reference Types Are the Same

Two reference types are the *same compile-time type* if they are declared in compilation units associated with the same module (§7.3), and they have the same binary name (§13.1), and their type arguments, if any, are the same, applying this definition recursively.

When two reference types are the same, they are sometimes said to be the *same class* or the *same interface*.

At run time, several reference types with the same binary name may be loaded simultaneously by different class loaders. These types may or may not represent the same type declaration. Even if two such types do represent the same type declaration, they are considered distinct.

Two reference types are the *same run-time type* if:

- They are both class or both interface types, are defined by the same class loader, and have the same binary name (§13.1), in which case they are sometimes said to be the *same run-time class* or the *same run-time interface*.
- They are both array types, and their component types are the same run-time type (§10 (Arrays)).

4.4 Type Variables

A *type variable* is an unqualified identifier used as a type in class, interface, method, and constructor bodies.

A type variable is introduced by the declaration of a *type parameter* of a generic class, interface, method, or constructor (§8.1.2, §9.1.2, §8.4.4, §8.8.4).

```
TypeParameter:
  {TypeParameterModifier} TypeIdentifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  extends TypeVariable
  extends ClassOrInterfaceType {AdditionalBound}
```
The scope of a type variable declared as a type parameter is specified in §6.3.

Every type variable declared as a type parameter has a *bound*. If no bound is declared for a type variable, `Object` is assumed. If a bound is declared, it consists of either:

- a single type variable $T$, or
- a class or interface type $T$ possibly followed by interface types $I_1 \& \ldots \& I_n$.

It is a compile-time error if any of the types $I_1 \ldots I_n$ is a class type or type variable.

The erasures (§4.6) of all constituent types of a bound must be pairwise different, or a compile-time error occurs.

A type variable must not at the same time be a subtype of two interface types which are different parameterizations of the same generic interface, or a compile-time error occurs.

The order of types in a bound is only significant in that the erasure of a type variable is determined by the first type in its bound, and that a class type or type variable may only appear in the first position.

The members of a type variable $X$ with bound $T \& I_1 \& \ldots \& I_n$ are the members of the intersection type (§4.9) $T \& I_1 \& \ldots \& I_n$ appearing at the point where the type variable is declared.

**Example 4.4-1. Members of a Type Variable**

```java
class C {
    public    void mCPublic()     ()
    protected void mCProtected() ()
    void mCPackage()     ()
    private   void mCPrivate()   ()
}

interface I {
    void mI();
}

class CT extends C implements I {
    public void mI() {}       
}

class Test {
```
```java
<T extends C & I> void test(T t) {
    t.mI();                      // OK
    t.mCPublic();                // OK
    t.mCProtected();             // OK
    t.mCPackage();               // OK
    t.mCPrivate();               // Compile-time error
}
```

The type variable $T$ has the same members as the intersection type $C \& I$, which in turn has the same members as the empty class $CT$, defined in the same scope with equivalent supertypes. The members of an interface are always public, and therefore always inherited (unless overridden). Hence $mI$ is a member of $CT$ and of $T$. Among the members of $C$, all but $mCPrivate$ are inherited by $CT$, and are therefore members of both $CT$ and $T$.

If $C$ had been declared in a different package than $T$, then the call to $mCPackage$ would give rise to a compile-time error, as that member would not be accessible at the point where $T$ is declared.

### 4.5 Parameterized Types

A class or interface declaration that is generic (§8.1.2, §9.1.2) defines a set of **parameterized types**.

A parameterized type is a class or interface type of the form $C<T_1,\ldots,T_n>$, where $C$ is the name of a generic type and $<T_1,\ldots,T_n>$ is a list of type arguments that denote a particular parameterization of the generic type.

A generic type has type parameters $F_1,\ldots,F_n$ with corresponding bounds $B_1,\ldots,B_n$. Each type argument $T_i$ of a parameterized type ranges over all types that are subtypes of all types listed in the corresponding bound. That is, for each bound type $S$ in $B_i$, $T_i$ is a subtype of $S[F_1:=T_1,\ldots,F_n:=T_n]$ (§4.10).

A parameterized type $C<T_1,\ldots,T_n>$ is **well-formed** if all of the following are true:

- $C$ is the name of a generic type.
- The number of type arguments is the same as the number of type parameters in the generic declaration of $C$.
- When subjected to capture conversion (§5.1.10) resulting in the type $C<X_1,\ldots,X_n>$, each type argument $X_i$ is a subtype of $S[F_1:=X_1,\ldots,F_n:=X_n]$ for each bound type $S$ in $B_i$.

It is a compile-time error if a parameterized type is not well-formed.
In this specification, whenever we speak of a class or interface type, we include the
generic version as well, unless explicitly excluded.

Two parameterized types are *provably distinct* if either of the following is true:

- They are parameterizations of distinct generic type declarations.
- Any of their type arguments are provably distinct.

Given the generic types in the examples of §8.1.2, here are some well-formed parameterized
types:

- Seq<String>
- Seq<Seq<String>>
- Seq<String>.Zipper<Integer>
- Pair<String,Integer>

Here are some incorrect parameterizations of those generic types:

- Seq<int> is illegal, as primitive types cannot be type arguments.
- Pair<String> is illegal, as there are not enough type arguments.
- Pair<String,String,String> is illegal, as there are too many type arguments.

A parameterized type may be an parameterization of a generic class or interface which
is nested. For example, if a non-generic class \( C \) has a generic member class \( D<T> \), then
\( C.D<Object> \) is a parameterized type. And if a generic class \( C<T> \) has a non-generic
member class \( D \), then the member type \( C<String>.D \) is a parameterized type, even though
the class \( D \) is not generic.

### 4.5.1 Type Arguments of Parameterized Types

Type arguments may be either reference types or wildcards. Wildcards are useful
in situations where only partial knowledge about the type parameter is required.

\[
\text{TypeArguments:} \\
< \text{TypeArgumentList} > 
\]

\[
\text{TypeArgumentList:} \\
\text{TypeArgument} \{ , \text{TypeArgument} \}
\]

\[
\text{TypeArgument:} \\
\text{ReferenceType} \\
\text{Wildcard}
\]
Wildcards may be given explicit bounds, just like regular type variable declarations. An upper bound is signified by the following syntax, where $B$ is the bound:

$$\? \text{ extends } B$$

Unlike ordinary type variables declared in a method signature, no type inference is required when using a wildcard. Consequently, it is permissible to declare lower bounds on a wildcard, using the following syntax, where $B$ is a lower bound:

$$\? \text{ super } B$$

The wildcard $\? \text{ extends } \text{Object}$ is equivalent to the unbounded wildcard $\?$. 

Two type arguments are **provably distinct** if one of the following is true:

- Neither argument is a type variable or wildcard, and the two arguments are not the same type.
- One type argument is a type variable or wildcard, with an upper bound (from capture conversion (§5.1.10), if necessary) of $S$; and the other type argument $T$ is not a type variable or wildcard; and neither $|S| <: |T|$ nor $|T| <: |S|$ (§4.8, §4.10).
- Each type argument is a type variable or wildcard, with upper bounds (from capture conversion, if necessary) of $S$ and $T$; and neither $|S| <: |T|$ nor $|T| <: |S|$.

A type argument $T_1$ is said to **contain** another type argument $T_2$, written $T_2 <= T_1$, if the set of types denoted by $T_2$ is provably a subset of the set of types denoted by $T_1$ under the reflexive and transitive closure of the following rules (where $<=$ denotes subtyping (§4.10)):

- $\? \text{ extends } T <= \? \text{ extends } S$ if $T <: S$
- $\? \text{ extends } T <= ?$
- $\? \text{ super } T <= \? \text{ super } S$ if $S <: T$
- $\? \text{ super } T <= ?$
- $\? \text{ super } T <= \? \text{ extends } \text{Object}$
The relationship of wildcards to established type theory is an interesting one, which we briefly allude to here. Wildcards are a restricted form of existential types. Given a generic type declaration \( G<T \text{ extends } B>, G<? > \) is roughly analogous to \( \text{Some } X <: B. G<X> \).

Historically, wildcards are a direct descendant of the work by Atsushi Igarashi and Mirko Viroli. Readers interested in a more comprehensive discussion should refer to *On Variance-Based Subtyping for Parametric Types* by Atsushi Igarashi and Mirko Viroli, in the *Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP 2002)*. This work itself builds upon earlier work by Kresten Thorup and Mads Torgersen (*Unifying Genericity*, ECOOP 99), as well as a long tradition of work on declaration based variance that goes back to Pierre America’s work on POOL (OOPSLA 89).

Wildcards differ in certain details from the constructs described in the aforementioned paper, in particular in the use of capture conversion (§5.1.10) rather than the close operation described by Igarashi and Viroli. For a formal account of wildcards, see *Wild FJ* by Mads Torgersen, Erik Ernst and Christian Plesner Hansen, in the 12th workshop on Foundations of Object Oriented Programming (FOOL 2005).

**Example 4.5.1-1. Unbounded Wildcards**

```java
import java.util.Collection;
import java.util.ArrayList;

class Test {
    static void printCollection(Collection<?> c) {
        // a wildcard collection
        for (Object o : c) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        Collection<String> cs = new ArrayList<String>();
        cs.add("hello");
        cs.add("world");
        printCollection(cs);
    }
}
```

Note that using `Collection<Object>` as the type of the incoming parameter, \( c \), would not be nearly as useful; the method could only be used with an argument expression that had type `Collection<Object>`, which would be quite rare. In contrast, the use of an unbounded wildcard allows any kind of collection to be passed as an argument.

Here is an example where the element type of an array is parameterized by a wildcard:
Example 4.5.1-2. Bounded Wildcards

```java
boolean addAll(Collection<? extends E> c)
```

Here, the method is declared within the interface `Collection<E>`, and is designed to add all the elements of its incoming argument to the collection upon which it is invoked. A natural tendency would be to use `Collection<E>` as the type of `c`, but this is unnecessarily restrictive. An alternative would be to declare the method itself to be generic:

```java
<T> boolean addAll(Collection<T> c)
```

This version is sufficiently flexible, but note that the type parameter is used only once in the signature. This reflects the fact that the type parameter is not being used to express any kind of interdependency between the type(s) of the argument(s), the return type and/or throws type. In the absence of such interdependency, generic methods are considered bad style, and wildcards are preferred.

```java
Reference(T referent, ReferenceQueue<? super T> queue)
```

Here, the referent can be inserted into any queue whose element type is a supertype of the type `T` of the referent; `T` is the lower bound for the wildcard.

### 4.5.2 Members and Constructors of Parameterized Types

Let `C` be a generic class or interface declaration with type parameters `A_1,...,A_n`, and let `C<T_1,...,T_n>` be a parameterization of `C` where, for `1 \leq i \leq n`, `T_i` is a type (rather than a wildcard). Then:

- Let `m` be a member or constructor declaration in `C`, whose type as declared is `T` (`§8.2, §8.8.6`).
  
  The type of `m` in `C<T_1,...,T_n>` is `T[A_1:=T_1,...,A_n:=T_n]`.

- Let `m` be a member or constructor declaration in `D`, where `D` is a class extended by `C` or an interface implemented by `C`. Let `D<U_1,...,U_k>` be the supertype of `C<T_1,...,T_n>` that corresponds to `D`.
  
  The type of `m` in `C<T_1,...,T_n>` is the type of `m` in `D<U_1,...,U_k>`.

If any of the type arguments in the parameterization of `C` are wildcards, then:

- The types of the fields, methods, and constructors in `C<T_1,...,T_n>` are the types of the fields, methods, and constructors in the capture conversion of `C<T_1,...,T_n>` (`§5.1.10`).
• Let \( D \) be a (possibly generic) class or interface declaration in \( C \). Then the type of \( D \) in \( C<T_1,\ldots,T_n> \) is \( D \) where, if \( D \) is generic, all type arguments are unbounded wildcards.

This is of no consequence, as it is impossible to access a member of a parameterized type without performing capture conversion, and it is impossible to use a wildcard after the keyword \( \text{new} \) in a class instance creation expression (§15.9).

The sole exception to the previous paragraph is when a nested parameterized type is used as the expression in an \( \text{instanceof} \) operator (§15.20.2), where capture conversion is not applied.

A static member that is declared in a generic type declaration must be referred to using the non-generic type that corresponds to the generic type (§6.1, §6.5.5.2, §6.5.6.2), or a compile-time error occurs.

In other words, it is illegal to refer to a static member declared in a generic type declaration by using a parameterized type.

### 4.6 Type Erasure

Type erasure is a mapping from types (possibly including parameterized types and type variables) to types (that are never parameterized types or type variables). We write \( |T| \) for the erasure of type \( T \). The erasure mapping is defined as follows:

• The erasure of a parameterized type (§4.5) \( G<T_1,\ldots,T_n> \) is \( |G| \).

• The erasure of a nested type \( T.C \) is \( |T|.C \).

• The erasure of an array type \( T[\] \) is \( |T|[\] \).

• The erasure of a type variable (§4.4) is the erasure of its leftmost bound.

• The erasure of every other type is the type itself.

Type erasure also maps the signature (§8.4.2) of a constructor or method to a signature that has no parameterized types or type variables. The erasure of a constructor or method signature \( s \) is a signature consisting of the same name as \( s \) and the erasures of all the formal parameter types given in \( s \).

The return type of a method (§8.4.5) and the type parameters of a generic method or constructor (§8.4.4, §8.8.4) also undergo erasure if the method or constructor's signature is erased.

The erasure of the signature of a generic method has no type parameters.
4.7 Reifiable Types

Because some type information is erased during compilation, not all types are available at run time. Types that are completely available at run time are known as reifiable types.

A type is reifiable if and only if one of the following holds:

- It refers to a non-generic class or interface type declaration.
- It is a parameterized type in which all type arguments are unbounded wildcards (§4.5.1).
- It is a raw type (§4.8).
- It is a primitive type (§4.2).
- It is an array type (§10.1) whose element type is reifiable.
- It is a nested type where, for each type \( T \) separated by a ",", \( T \) itself is reifiable.

For example, if a generic class \( X<T> \) has a generic member class \( Y<U> \), then the type \( X<T>.Y<U> \) is reifiable because \( X<T> \) is reifiable and \( Y<U> \) is reifiable. The type \( X<T>.Y<Object> \) is not reifiable because \( Y<Object> \) is not reifiable.

An intersection type is not reifiable.

The decision not to make all generic types reifiable is one of the most crucial, and controversial design decisions involving the type system of the Java programming language.

Ultimately, the most important motivation for this decision is compatibility with existing code. In a naive sense, the addition of new constructs such as generics has no implications for pre-existing code. The Java programming language, per se, is compatible with earlier versions as long as every program written in the previous versions retains its meaning in the new version. However, this notion, which may be termed language compatibility, is of purely theoretical interest. Real programs (even trivial ones, such as "Hello World") are composed of several compilation units, some of which are provided by the Java SE Platform (such as elements of java.lang or java.util). In practice, then, the minimum requirement is platform compatibility - that any program written for the prior version of the Java SE Platform continues to function unchanged in the new version.

One way to provide platform compatibility is to leave existing platform functionality unchanged, only adding new functionality. For example, rather than modify the existing Collections hierarchy in java.util, one might introduce a new library utilizing generics.

The disadvantages of such a scheme is that it is extremely difficult for pre-existing clients of the Collection library to migrate to the new library. Collections are used to exchange data between independently developed modules; if a vendor decides to switch to the new, generic, library, that vendor must also distribute two versions of their code, to be compatible...
with their clients. Libraries that are dependent on other vendors code cannot be modified to use generics until the supplier’s library is updated. If two modules are mutually dependent, the changes must be made simultaneously.

Clearly, platform compatibility, as outlined above, does not provide a realistic path for adoption of a pervasive new feature such as generics. Therefore, the design of the generic type system seeks to support migration compatibility. Migration compatibility allows the evolution of existing code to take advantage of generics without imposing dependencies between independently developed software modules.

The price of migration compatibility is that a full and sound reification of the generic type system is not possible, at least while the migration is taking place.

### 4.8 Raw Types

To facilitate interfacing with non-generic legacy code, it is possible to use as a type the erasure (§4.6) of a parameterized type (§4.5) or the erasure of an array type (§10.1) whose element type is a parameterized type. Such a type is called a raw type.

More precisely, a raw type is defined to be one of:

- The reference type that is formed by taking the name of a generic type declaration without an accompanying type argument list.
- An array type whose element type is a raw type.
- A non-static member type of a raw type $\mathcal{R}$ that is not inherited from a superclass or superinterface of $\mathcal{R}$.

A non-generic class or interface type is not a raw type.

To see why a non-static type member of a raw type is considered raw, consider the following example:

```java
class Outer<T>{
    T t;
    class Inner {
        T setOuterT(T t1) { t = t1; return t; }
    }
}
```

The type of the member(s) of `Inner` depends on the type parameter of `Outer`. If `Outer` is raw, `Inner` must be treated as raw as well, as there is no valid binding for `T`. 


This rule applies only to type members that are not inherited. Inherited type members that
depend on type variables will be inherited as raw types as a consequence of the rule that
the supertypes of a raw type are erased, described later in this section.

Another implication of the rules above is that a generic inner class of a raw type can itself
only be used as a raw type:

class Outer<T>{
   class Inner<S> {
      S s;
   }
}

It is not possible to access `Inner` as a partially raw type (a "rare" type):

```java
Outer.Inner<Double> x = null;  // illegal
Double d = x.s;
```

because `Outer` itself is raw, hence so are all its inner classes including `Inner`, and so it
is not possible to pass any type arguments to `Inner`.

The superclasses (respectively, superinterfaces) of a raw type are the erasures of the
superclasses (superinterfaces) of any of the parameterizations of the generic type.

The type of a constructor (§8.8), instance method (§8.4, §9.4), or non-static field
(§8.3) of a raw type `C` that is not inherited from its superclasses or superinterfaces
is the raw type that corresponds to the erasure of its type in the generic declaration
corresponding to `C`.

The type of a static method or static field of a raw type `C` is the same as its type
in the generic declaration corresponding to `C`.

It is a compile-time error to pass type arguments to a non-static type member of
a raw type that is not inherited from its superclasses or superinterfaces.

It is a compile-time error to attempt to use a type member of a parameterized type
as a raw type.

This means that the ban on "rare" types extends to the case where the qualifying type is
parameterized, but we attempt to use the inner class as a raw type:

```java
Outer<Integer>.Inner x = null; // illegal
```

This is the opposite of the case discussed above. There is no practical justification for this
half-baked type. In legacy code, no type arguments are used. In non-legacy code, we should
use the generic types correctly and pass all the required type arguments.
The supertype of a class may be a raw type. Member accesses for the class are treated as normal, and member accesses for the supertype are treated as for raw types. In the constructor of the class, calls to `super` are treated as method calls on a raw type.

The use of raw types is allowed only as a concession to compatibility of legacy code. The use of raw types in code written after the introduction of generics into the Java programming language is strongly discouraged. It is possible that future versions of the Java programming language will disallow the use of raw types.

To make sure that potential violations of the typing rules are always flagged, some accesses to members of a raw type will result in compile-time unchecked warnings. The rules for compile-time unchecked warnings when accessing members or constructors of raw types are as follows:

- At an assignment to a field: if the type of the `Primary` in the field access expression (§15.11) is a raw type, then a compile-time unchecked warning occurs if erasure changes the field's type.

- At an invocation of a method or constructor: if the type of the class or interface to search (§15.12.1) is a raw type, then a compile-time unchecked warning occurs if erasure changes any of the formal parameter types of the method or constructor.

- No compile-time unchecked warning occurs for a method call when the formal parameter types do not change under erasure (even if the return type and/or `throws` clause changes), for reading from a field, or for a class instance creation of a raw type.

Note that the unchecked warnings above are distinct from the unchecked warnings possible from narrowing reference conversion (§5.1.6), unchecked conversion (§5.1.9), method declarations (§8.4.1, §8.4.8.3), and certain expressions (§15.12.4.2, §15.13.2, §15.27.3).

The warnings here cover the case where a legacy consumer uses a generified library. For example, the library declares a generic class `Foo<T extends String>` that has a field `f` of type `Vector<T>`, but the consumer assigns a vector of integers to `e.f` where `e` has the raw type `Foo`. The legacy consumer receives a warning because it may have caused heap pollution (§4.12.2) for generified consumers of the generified library.

(Note that the legacy consumer can assign a `Vector<String>` from the library to its own `Vector` variable without receiving a warning. That is, the subtyping rules (§4.10.2) of the Java programming language make it possible for a variable of a raw type to be assigned a value of any of the type's parameterized instances.)

The warnings from unchecked conversion cover the dual case, where a generified consumer uses a legacy library. For example, a method of the library has the raw return type `Vector`, but the consumer assigns the result of the method invocation to a variable of type `Vector<String>`. This is unsafe, since the raw vector might have had a different element type than `String`, but is still permitted using unchecked conversion in order to enable
interfacing with legacy code. The warning from unchecked conversion indicates that the
generated consumer may experience problems from heap pollution at other points in the
program.

Example 4.8-1. Raw Types

class Cell<E> {
    E value;

    Cell(E v) { value = v; }
    E get() { return value; }
    void set(E v) { value = v; }

    public static void main(String[] args) {
        Cell x = new Cell<String>("abc");
        System.out.println(x.value);  // OK, has type Object
        System.out.println(x.get());  // OK, has type Object
        x.set("def");                 // unchecked warning
    }
}

Example 4.8-2. Raw Types and Inheritance

import java.util.*;
class NonGeneric {
    Collection<Number> myNumbers() { return null; }
}

abstract class RawMembers<T> extends NonGeneric
    implements Collection<String> {
    static Collection<NonGeneric> cng =
        new ArrayList<NonGeneric>();

    public static void main(String[] args) {
        RawMembers rw = null;
        Collection<Number> cn = rw.myNumbers();
            // OK
        Iterator<String> is = rw.iterator();
            // Unchecked warning
        Collection<NonGeneric> cnn = rw.cng;
            // OK, static member
    }
}

In this program (which is not meant to be run), RawMembers<T> inherits the method:

    Iterator<String> iterator()

from the Collection<String> superinterface. The raw type RawMembers inherits
iterator() from Collection, the erasure of Collection<String>, which means that
the return type of iterator() in RawMembers is Iterator. As a result, the attempt to
assign rw.iterator() to Iterator<String> requires an unchecked conversion, so a compile-time unchecked warning is issued.

In contrast, RawMembers inherits myNumbers() from the NonGeneric class whose erasure is also NonGeneric. Thus, the return type of myNumbers() in RawMembers is not erased, and the attempt to assign rw.myNumbers() to Collection<Number> requires no unchecked conversion, so no compile-time unchecked warning is issued.

Similarly, the static member cng retains its parameterized type even when accessed through a object of raw type. Note that access to a static member through an instance is considered bad style and is discouraged.

This example reveals that certain members of a raw type are not erased, namely static members whose types are parameterized, and members inherited from a non-generic supertype.

Raw types are closely related to wildcards. Both are based on existential types. Raw types can be thought of as wildcards whose type rules are deliberately unsound, to accommodate interaction with legacy code. Historically, raw types preceded wildcards; they were first introduced in GJ, and described in the paper Making the future safe for the past: Adding Genericity to the Java Programming Language by Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler, in Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 98), October 1998.

4.9 Intersection Types

An intersection type takes the form \( T_1 \& \ldots \& T_n \) \((n > 0)\), where \( T_i \) \((1 \leq i \leq n)\) are types.

Intersection types can be derived from type parameter bounds (§4.4) and cast expressions (§15.16); they also arise in the processes of capture conversion (§5.1.10) and least upper bound computation (§4.10.4).

The values of an intersection type are those objects that are values of all of the types \( T_i \) for \( 1 \leq i \leq n \).

Every intersection type \( T_1 \& \ldots \& T_n \) induces a notional class or interface for the purpose of identifying the members of the intersection type, as follows:

- For each \( T_i \) \((1 \leq i \leq n)\), let \( C_i \) be the most specific class or array type such that \( T_i \prec: C_i \). Then there must be some \( C_k \) such that \( C_k \prec: C_i \) for any \( i \) \((1 \leq i \leq n)\), or a compile-time error occurs.

- For \( 1 \leq j \leq n \), if \( T_j \) is a type variable, then let \( T_j' \) be an interface whose members are the same as the public members of \( T_j \); otherwise, if \( T_j \) is an interface, then let \( T_j' \) be \( T_j \).
• If $C_k$ is Object, a notional interface is induced; otherwise, a notional class is induced with direct superclass $C_k$. This class or interface has direct superinterfaces $T_1', ..., T_n'$ and is declared in the package in which the intersection type appears.

The members of an intersection type are the members of the class or interface it induces.

It is worth dwelling upon the distinction between intersection types and the bounds of type variables. Every type variable bound induces an intersection type. This intersection type is often trivial, consisting of a single type. The form of a bound is restricted (only the first element may be a class or type variable, and only one type variable may appear in the bound) to preclude certain awkward situations coming into existence. However, capture conversion can lead to the creation of type variables whose bounds are more general, such as array types.

### 4.10 Subtyping

The subtype and supertype relations are binary relations on types.

The supertypes of a type are obtained by reflexive and transitive closure over the direct supertype relation, written $S >_1 T$, which is defined by rules given later in this section. We write $S :> T$ to indicate that the supertype relation holds between $S$ and $T$.

$S$ is a proper supertype of $T$, written $S > T$, if $S :> T$ and $S \neq T$.

The subtypes of a type $T$ are all types $U$ such that $T$ is a supertype of $U$, and the null type. We write $T <: S$ to indicate that that the subtype relation holds between types $T$ and $S$.

$T$ is a proper subtype of $S$, written $T < S$, if $T <: S$ and $S \neq T$.

$T$ is a direct subtype of $S$, written $T <^1 S$, if $S >^1 T$.

Subtyping does not extend through parameterized types: $T <: S$ does not imply that $C<T> <: C<S>$.}

### 4.10.1 Subtyping among Primitive Types

The following rules define the direct supertype relation among the primitive types:

- $\text{double} >_1 \text{float}$
- $\text{float} >_1 \text{long}$
• long >₁ int
• int >₁ char
• int >₁ short
• short >₁ byte

4.10.2 Subtyping among Class and Interface Types

Given a non-generic type declaration \( c \), the direct supertypes of the type \( c \) are all of the following:

• The direct superclass of \( c \) (§8.1.4).
• The direct superinterfaces of \( c \) (§8.1.5).
• The type \( \text{Object} \), if \( c \) is an interface type with no direct superinterfaces (§9.1.3).

Given a generic type declaration \( c<_{F_1,...,F_n}> \ (n > 0) \), the direct supertypes of the raw type \( c \) (§4.8) are all of the following:

• The direct superclass of the raw type \( c \).
• The direct superinterfaces of the raw type \( c \).
• The type \( \text{Object} \), if \( c<_{F_1,...,F_n}> \) is a generic interface type with no direct superinterfaces (§9.1.2).

Given a generic type declaration \( c<_{F_1,...,F_n}> \ (n > 0) \), the direct supertypes of the generic type \( c<_{F_1,...,F_n}> \) are all of the following:

• The direct superclass of \( c<_{F_1,...,F_n}> \).
• The direct superinterfaces of \( c<_{F_1,...,F_n}> \).
• The type \( \text{Object} \), if \( c<_{F_1,...,F_n}> \) is a generic interface type with no direct superinterfaces.
• The raw type \( c \).

Given a generic type declaration \( c<_{F_1,...,F_n}> \ (n > 0) \), the direct supertypes of the parameterized type \( c<_{T_1,...,T_n}> \), where \( T_i \ (1 \leq i \leq n) \) is a type, are all of the following:

• \( D<_{U_1 \theta,...,U_k \theta}> \), where \( D<_{U_1,...,U_k}> \) is a generic type which is a direct supertype of the generic type \( c<_{F_1,...,F_n}> \) and \( \theta \) is the substitution \([F_1:=T_1, \ldots, F_n:=T_n]\).
• \( c<_{S_1,...,S_n}> \), where \( S_i \) contains \( T_i \ (1 \leq i \leq n) \) (§4.5.1).
The type **Object**, if $C<\ldots,F_n>$ is a generic interface type with no direct superinterfaces.

- The raw type $c$.

Given a generic type declaration $C<\ldots,F_n>$ ($n > 0$), the *direct supertypes* of the parameterized type $C<R_1,\ldots,R_n>$ where at least one of the $R_i$ (1 ≤ $i$ ≤ $n$) is a wildcard type argument, are the direct supertypes of the parameterized type $C<X_1,\ldots,X_n>$ which is the result of applying capture conversion to $C<R_1,\ldots,R_n>$ (§5.1.10).

The direct supertypes of an intersection type $T_1 \& \ldots \& T_n$ are $T_i$ (1 ≤ $i$ ≤ $n$).

The direct supertypes of a type variable are the types listed in its bound.

A type variable is a direct supertype of its lower bound.

The direct supertypes of the null type are all reference types other than the null type itself.

### 4.10.3 Subtyping among Array Types

The following rules define the direct supertype relation among array types:

- If $S$ and $T$ are both reference types, then $S[\ldots] >_1 T[\ldots]$ iff $S >_1 T$.

- **Object** >$_1$ **Object[]**

- **Cloneable** >$_1$ **Object[]**

- `java.io.Serializable` >$_1$ **Object[]**

- If $P$ is a primitive type, then:
  - **Object** >$_1$ $P[\ldots]$
  - **Cloneable** >$_1$ $P[\ldots]$
  - `java.io.Serializable` >$_1$ $P[\ldots]$

### 4.10.4 Least Upper Bound

The *least upper bound*, or "lub", of a set of reference types is a shared supertype that is more specific than any other shared supertype (that is, no other shared supertype is a subtype of the least upper bound). This type, lub($U_1, \ldots, U_k$), is determined as follows.

If $k = 1$, then the lub is the type itself: lub($U$) = $U$.

Otherwise:
4.10 Subtyping

• For each $u_i$ ($1 \leq i \leq k$):
  
  Let $\text{ST}(u_i)$ be the set of supertypes of $u_i$.
  
  Let $\text{EST}(u_i)$, the set of erased supertypes of $u_i$, be:
  
  \[
  \text{EST}(u_i) = \{ |w| \mid w \text{ in } \text{ST}(u_i) \} \text{ where } |w| \text{ is the erasure of } w.
  \]

  The reason for computing the set of erased supertypes is to deal with situations where the set of types includes several distinct parameterizations of a generic type.

  For example, given `List<String>` and `List<Object>`, simply intersecting the sets $\text{ST}(\text{List<String>}) = \{ \text{List<String>, Collection<String>, Object} \}$ and $\text{ST}(\text{List<Object>}) = \{ \text{List<Object>, Collection<Object>, Object} \}$ would yield a set $\{ \text{Object} \}$, and we would have lost track of the fact that the upper bound can safely be assumed to be a `List`.

  In contrast, intersecting $\text{EST}(\text{List<String>}) = \{ \text{List, Collection, Object} \}$ and $\text{EST}(\text{List<Object>}) = \{ \text{List, Collection, Object} \}$ yields $\{ \text{List, Collection, Object} \}$, which will eventually enable us to produce `List<??>`.

• Let $\text{EC}$, the erased candidate set for $u_1 \ldots u_k$, be the intersection of all the sets $\text{EST}(u_i)$ ($1 \leq i \leq k$).

• Let $\text{MEC}$, the minimal erased candidate set for $u_1 \ldots u_k$, be:
  
  \[
  \text{MEC} = \{ v \mid v \text{ in } \text{EC}, \text{ and for all } w \neq v \text{ in } \text{EC}, \text{ it is not the case that } w <: v \}
  \]

  Because we are seeking to infer more precise types, we wish to filter out any candidates that are supertypes of other candidates. This is what computing MEC accomplishes. In our running example, we had $\text{EC} = \{ \text{List, Collection, Object} \}$, so $\text{MEC} = \{ \text{List} \}$. The next step is to recover type arguments for the erased types in MEC.

• For any element $g$ of MEC that is a generic type:
  
  Let the "relevant" parameterizations of $g$, $\text{Relevant}(g)$, be:
  
  \[
  \text{Relevant}(g) = \{ v \mid 1 \leq i \leq k: v \text{ in } \text{ST}(u_i) \text{ and } v = g<...> \}
  \]

  In our running example, the only generic element of MEC is `List`, and $\text{Relevant(List)} = \{ \text{List<String>, List<Object>} \}$. We will now seek to find a type argument for `List` that contains (§4.5.1) both `String` and `Object`.

  This is done by means of the least containing parameterization (lcp) operation defined below. The first line defines lcp() on a set, such as $\text{Relevant(List)}$, as an operation on a list consisting of the elements of the set. The next line defines the operation on such a list as a pairwise reduction on the elements of the list. The third line is the definition of lcp() on pairs of parameterized types, which in turn relies on the notion of least containing type argument (lcta). lcta() is defined for all possible cases.
Let the "candidate" parameterization of \( \mathcal{G} \), Candidate(\( \mathcal{G} \)), be the most specific parameterization of the generic type \( \mathcal{G} \) that contains all the relevant parameterizations of \( \mathcal{G} \):

\[
\text{Candidate}(\mathcal{G}) = \text{lcp(Relevant}(\mathcal{G}))
\]

where lcp(), the least containing parameterization, is:

- \( \text{lcp}(S) = \text{lcp}(e_1, ..., e_n) \) where \( e_i \) (\( 1 \leq i \leq n \)) in \( S \)
- \( \text{lcp}(e_1, ..., e_n) = \text{lcp}(\text{lcp}(e_1, e_2), e_3, ..., e_n) \)
- \( \text{lcp}(G<X_1, ..., X_n>, G<Y_1, ..., Y_n>) = G<\text{lcta}(X_1), ..., \text{lcta}(X_n, Y_n)> \)
- \( \text{lcp}(G<X_1, ..., X_n>) = G<\text{lcta}(X_1), ..., \text{lcta}(X_n)> \)

and where lcta(), the least containing type argument, is: (assuming \( U \) and \( V \) are types)

- \( \text{lcta}(U, V) = U \text{ if } U = V, \text{ otherwise } ? \text{ extends } \text{lub}(U, V) \)
- \( \text{lcta}(U, ? \text{ extends } V) = ? \text{ extends } \text{lub}(U, V) \)
- \( \text{lcta}(U, ? \text{ super } V) = ? \text{ super } \text{glb}(U, V) \)
- \( \text{lcta}(? \text{ extends } U, ? \text{ extends } V) = ? \text{ extends } \text{lub}(U, V) \)
- \( \text{lcta}(? \text{ extends } U, ? \text{ super } V) = ? \text{ super } \text{glb}(U, V) \)
- \( \text{lcta}(U) = ? \text{ if } U\text{'s upper bound is Object, otherwise } ? \text{ extends } \text{lub}(U, \text{Object}) \)

and where glb() is as defined in §5.1.10.

- Let \( \text{lub}(U_1 \ldots U_k) \) be:

\[
\text{Best}(\bar{w}_1) \& \ldots \& \text{Best}(\bar{w}_r)
\]

where \( \bar{w}_i \) (\( 1 \leq i \leq r \)) are the elements of MEC, the minimal erased candidate set of \( U_1 \ldots U_k \);

and where, if any of these elements are generic, we use the candidate parameterization (so as to recover type arguments):

\[
\text{Best}(x) = \text{Candidate}(x) \text{ if } x \text{ is generic; } x \text{ otherwise.}
\]

Strictly speaking, this lub() function only approximates a least upper bound. Formally, there may exist some other type \( T \) such that all of \( U_1 \ldots U_k \) are subtypes of \( T \) and \( T \) is a subtype of \( \text{lub}(U_1, ..., U_k) \). However, a compiler for the Java programming language must implement lub() as specified above.
It is possible that the lub() function yields an infinite type. This is permissible, and a compiler for the Java programming language must recognize such situations and represent them appropriately using cyclic data structures.

The possibility of an infinite type stems from the recursive calls to lub(). Readers familiar with recursive types should note that an infinite type is not the same as a recursive type.

4.10.5 Type Projections

A synthetic type variable is a type variable introduced by the compiler during capture conversion (§5.1.10) or inference variable resolution (§18.4).

It is sometimes necessary to find a close supertype of a type, where that supertype does not mention certain synthetic type variables. This is achieved with an upward projection applied to the type.

Similarly, a downward projection may be applied to find a close subtype of a type, where that subtype does not mention certain synthetic type variables. Because such a type does not always exist, downward projection is a partial function.

These operations take as input a set of type variables that should no longer be referenced, referred to as the restricted type variables. When the operations recur, the set of restricted type variables is implicitly passed on to the recursive application.

The upward projection of a type $T$ with respect to a set of restricted type variables is defined as follows:

- If $T$ does not mention any restricted type variable, then the result is $T$.
- If $T$ is a restricted type variable, then the result is the upward projection of the upper bound of $T$.
- If $T$ is a parameterized class type or a parameterized interface type, $G<A_1,...,A_n>$, then the result is $G<A_1',...,A_n'>$ where, for $1 \leq i \leq n$, $A_i'$ is derived from $A_i$ as follows:
  - If $A_i$ does not mention any restricted type variable, then $A_i' = A_i$.
  - If $A_i$ is a type that mentions a restricted type variable, then let $U$ be the upward projection of $A_i$. $A_i'$ is a wildcard, defined by three cases:
    - If $U$ is not Object, and if either the declared bound of the $i$th parameter of $G$, $B_i$, mentions a type parameter of $G$, or $B_i$ is not a subtype of $U$, then $A_i'$ is an upper-bounded wildcard, $\exists$ extends $U$.  


Otherwise, if the downward projection of $A_i$ is $L$, then $A_i'$ is a lower-bounded wildcard, $\exists \text{ super } L$.

Otherwise, the downward projection of $A_i$ is undefined and $A_i'$ is an unbounded wildcard, $\exists$.

– If $A_i$ is an upper-bounded wildcard that mentions a restricted type variable, then let $U$ be the upward projection of the wildcard bound. $A_i'$ is an upper-bounded wildcard, $\exists \text{ extends } U$.

– If $A_i$ is a lower-bounded wildcard that mentions a restricted type variable, then if the downward projection of the wildcard bound is $L$, then $A_i'$ is a lower-bounded wildcard, $\exists \text{ super } L$; if the downward projection of the wildcard bound is undefined, then $A_i'$ is an unbounded wildcard, $\exists$.

• If $T$ is an array type, $S[]$, then the result is an array type whose component type is the upward projection of $S$.

• If $T$ is an intersection type, then the result is an intersection type. For each element, $S$, of $T$, the result has as an element the upward projection of $S$.

The downward projection of a type $T$ with respect to a set of restricted type variables is a partial function, defined as follows:

• If $T$ does not mention any restricted type variable, then the result is $T$.

• If $T$ is a restricted type variable, then if $T$ has a lower bound, and if the downward projection of that bound is $L$, the result is $L$; if $T$ has no lower bound, or if the downward projection of that bound is undefined, then the result is undefined.

• If $T$ is a parameterized class type or a parameterized interface type, $G<A_1, \ldots, A_n>$, then the result is $G<A_1', \ldots, A_n'>$; if, for $1 \leq i \leq n$, a type argument $A_i'$ can be derived from $A_i$ as follows; if not, the result is undefined:

  – If $A_i$ is does not mention a restricted type variable, then $A_i' = A_i$.

  – If $A_i$ is a type that mentions a restricted type variable, then $A_i'$ is undefined.

  – If $A_i$ is an upper-bounded wildcard that mentions a restricted type variable, then if the downward projection of the wildcard bound is $U$, then $A_i'$ is an upper-bounded wildcard, $\exists \text{ extends } U$; if the downward projection of the wildcard bound is undefined, then $A_i'$ is undefined.

  – If $A_i$ is a lower-bounded wildcard that mentions a restricted type variable, then let $L$ be the upward projection of the wildcard bound. $A_i'$ is a lower-bounded wildcard, $\exists \text{ super } L$. 
• If $T$ is an array type, $S[\cdot]$, then if the downward projection of $S$ is $S'$, the result is $S'[\cdot]$; if the downward projection of $S$ is undefined, then the result is undefined.

• If $T$ is an intersection type, then if the downward projection is defined for each element of $T$, the result is an intersection type whose elements are the downward projections of the elements of $T$; if the downward projection is undefined for any element of $T$, then the result is undefined.

Like lub (§4.10.4), upward projection and downward projection may produce infinite types, due to the recursion on type variable bounds.

### 4.11 Where Types Are Used

Types are used in most kinds of declaration and in certain kinds of expression. Specifically, there are 16 type contexts where types are used:

- In declarations:
  1. A type in the extends or implements clause of a class declaration (§8.1.4, §8.1.5, §8.5, §9.5)
  2. A type in the extends clause of an interface declaration (§9.1.3, §8.5, §9.5)
  3. The return type of a method (including the type of an element of an annotation type) (§8.4.5, §9.4, §9.6.1)
  4. A type in the throws clause of a method or constructor (§8.4.6, §8.8.5, §9.4)
  5. A type in the extends clause of a type parameter declaration of a generic class, interface, method, or constructor (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
  6. The type in a field declaration of a class or interface (including an enum constant) (§8.3, §9.3, §8.9.1)
  7. The type in a formal parameter declaration of a method, constructor, or lambda expression (§8.4.1, §8.8.1, §9.4, §15.27.1)
  8. The type of the receiver parameter of a method (§8.4)
  10. The type in an exception parameter declaration (§14.20)

- In expressions:
11. A type in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (§8.8.7.1, §15.9, §15.12)

12. In an unqualified class instance creation expression, as the class type to be instantiated (§15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (§15.9.5)

13. The element type in an array creation expression (§15.10.1)

14. The type in the cast operator of a cast expression (§15.16)

15. The type that follows the instanceof relational operator (§15.20.2)

16. In a method reference expression (§15.13), as the reference type to search for a member method or as the class type or array type to construct.

Also, types are used as:

- The element type of an array type in any of the above contexts; and
- A non-wildcard type argument, or a bound of a wildcard type argument, of a parameterized type in any of the above contexts.

Finally, there are three special terms in the Java programming language which denote the use of a type:

- An unbounded wildcard (§4.5.1)
- The . . . in the type of a variable arity parameter (§8.4.1), to indicate an array type
- The simple name of a type in a constructor declaration (§8.8), to indicate the class of the constructed object

The meaning of types in type contexts is given by:

- §4.2, for primitive types
- §4.4, for type parameters
- §4.5, for class and interface types that are parameterized, or appear either as type arguments in a parameterized type or as bounds of wildcard type arguments in a parameterized type
- §4.8, for class and interface types that are raw
- §4.9, for intersection types in the bounds of type parameters
- §6.5, for class and interface types in contexts where genericity is unimportant (§6.1)
• §10.1, for array types

Some type contexts restrict how a reference type may be parameterized:

• The following type contexts require that if a type is a parameterized reference type, it has no wildcard type arguments:
  – In an extends or implements clause of a class declaration (§8.1.4, §8.1.5)
  – In an extends clause of an interface declaration (§9.1.3)
  – In an unqualified class instance creation expression, as the class type to be instantiated (§15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (§15.9.5)
  – In a method reference expression (§15.13), as the reference type to search for a member method or as the class type or array type to construct.

In addition, no wildcard type arguments are permitted in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression or method reference expression (§8.8.7.1, §15.9, §15.12, §15.13).

• The following type contexts require that if a type is a parameterized reference type, it has only unbounded wildcard type arguments (i.e. it is a reifiable type):
  – As the element type in an array creation expression (§15.10.1)
  – As the type that follows the instanceof relational operator (§15.20.2)

• The following type contexts disallow a parameterized reference type altogether, because they involve exceptions and the type of an exception is non-generic (§6.1):
  – As the type of an exception that can be thrown by a method or constructor (§8.4.6, §8.8.5, §9.4)
  – In an exception parameter declaration (§14.20)

In any type context where a type is used, it is possible to annotate the keyword denoting a primitive type or the Identifier denoting the simple name of a reference type. It is also possible to annotate an array type by writing an annotation to the left of the ] at the desired level of nesting in the array type. Annotations in these locations are called type annotations, and are specified in §9.7.4. Here are some examples:

• @Foo int[] f; annotates the primitive type int
• int @Foo [] f; annotates the array type int[
• int @Foo [][] f; annotates the array type int[][]
• `int[] @Foo [] f;` annotates the array type `int[]` which is the component type of the array type `int[]`.

Five of the type contexts which appear in declarations occupy the same syntactic real estate as a number of declaration contexts (§9.6.4.1):

• The return type of a method (including the type of an element of an annotation type)
• The type in a field declaration of a class or interface (including an enum constant)
• The type in a formal parameter declaration of a method, constructor, or lambda expression
• The type in a local variable declaration
• The type in an exception parameter declaration

The fact that the same syntactic location in a program can be both a type context and a declaration context arises because the modifiers for a declaration immediately precede the type of the declared entity. §9.7.4 explains how an annotation in such a location is deemed to appear in a type context or a declaration context or both.

Example 4.11-1. Usage of a Type

```java
import java.util.Random;
import java.util.Collection;
import java.util.ArrayList;

class MiscMath<T extends Number> {
    int divisor;
    MiscMath(int divisor) { this.divisor = divisor; }
    float ratio(long l) {
        try {
            l /= divisor;
        } catch (Exception e) {
            if (e instanceof ArithmeticException)
                l = Long.MAX_VALUE;
            else
                l = 0;
        }
        return (float)l;
    }
    double gausser() {
        Random r = new Random();
        double[] val = new double[2];
        val[0] = r.nextGaussian();
        val[1] = r.nextGaussian();
        return (val[0] + val[1]) / 2;
    }
    Collection<Number> fromArray(Number[] na) {
        Collection<Number> cn = new ArrayList<Number>();
        for (Number n : na) cn.add(n);
        return cn;
    }
}```
4.11 Where Types Are Used

TYPES, VALUES, AND VARIABLES

```java
<S> void loop(S s) { this.<S>loop(s); }
```

In this example, types are used in declarations of the following:

- **Imported types** (§7.5); here the type `Random`, imported from the type `java.util.Random` of the package `java.util`, is declared
- **Fields**, which are the class variables and instance variables of classes (§8.3), and constants of interfaces (§9.3); here the field `divisor` in the class `MiscMath` is declared to be of type `int`
- **Method parameters** (§8.4.1); here the parameter `l` of the method `ratio` is declared to be of type `long`
- **Method results** (§8.4); here the result of the method `ratio` is declared to be of type `float`, and the result of the method `gausser` is declared to be of type `double`
- **Constructor parameters** (§8.8.1); here the parameter of the constructor for `MiscMath` is declared to be of type `int`
- **Local variables** (§14.4, §14.14); the local variables `r` and `val` of the method `gausser` are declared to be of types `Random` and `double[]` (array of `double`)
- **Exception parameters** (§14.20); here the exception parameter `e` of the `catch` clause is declared to be of type `Exception`
- **Type parameters** (§4.4); here the type parameter of `MiscMath` is a type variable `T` with the type `Number` as its declared bound
- In any declaration that uses a parameterized type; here the type `Number` is used as a type argument (§4.5.1) in the parameterized type `Collection<Number>`.

and in expressions of the following kinds:

- **Class instance creations** (§15.9); here a local variable `r` of method `gausser` is initialized by a class instance creation expression that uses the type `Random`
- **Generic class** (§8.1.2) instance creations (§15.9); here `Number` is used as a type argument in the expression `new ArrayList<Number>()`
- **Array creations** (§15.10.1); here the local variable `val` of method `gausser` is initialized by an array creation expression that creates an array of `double` with size 2
- **Generic method** (§8.4.4) or constructor (§8.8.4) invocations (§15.12); here the method `loop` calls itself with an explicit type argument `S`
- **Casts** (§15.16); here the `return` statement of the method `ratio` uses the `float` type in a cast
- **The `instanceof` operator** (§15.20.2); here the `instanceof` operator tests whether `e` is assignment-compatible with the type `ArithmeticException`
4.12 Variables

A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type (§4.2) or a reference type (§4.3).

A variable's value is changed by an assignment (§15.26) or by a prefix or postfix + (increment) or -- (decrement) operator (§15.14.2, §15.14.3, §15.15.1, §15.15.2).

Compatibility of the value of a variable with its type is guaranteed by the design of the Java programming language, as long as a program does not give rise to compile-time unchecked warnings (§4.12.2). Default values (§4.12.5) are compatible and all assignments to a variable are checked for assignment compatibility (§5.2), usually at compile time, but, in a single case involving arrays, a run-time check is made (§10.5).

4.12.1 Variables of Primitive Type

A variable of a primitive type always holds a primitive value of that exact primitive type.

4.12.2 Variables of Reference Type

A variable of a class type $T$ can hold a null reference or a reference to an instance of class $T$ or of any class that is a subclass of $T$.

A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.

Note that a variable is not guaranteed to always refer to a subtype of its declared type, but only to subclasses or subinterfaces of the declared type. This is due to the possibility of heap pollution discussed below.

If $T$ is a primitive type, then a variable of type "array of $T$" can hold a null reference or a reference to any array of type "array of $T$".

If $T$ is a reference type, then a variable of type "array of $T$" can hold a null reference or a reference to any array of type "array of $S" such that type $S$ is a subclass or subinterface of type $T$.

A variable of type Object[] can hold a reference to an array of any reference type.

A variable of type Object can hold a null reference or a reference to any object, whether it is an instance of a class or an array.
It is possible that a variable of a parameterized type will refer to an object that is not of that parameterized type. This situation is known as *heap pollution*.

Heap pollution can only occur if the program performed some operation involving a raw type that would give rise to a compile-time unchecked warning (§4.8, §5.1.6, §5.1.9, §8.4.1, §8.4.8.3, §8.4.8.4, §9.4.1.2, §15.12.4.2), or if the program aliases an array variable of non-reifiable element type through an array variable of a supertype which is either raw or non-generic.

For example, the code:

```java
List l = new ArrayList<Number>();
List<String> ls = l;  // Unchecked warning
```

gives rise to a compile-time unchecked warning, because it is not possible to ascertain, either at compile time (within the limits of the compile-time type checking rules) or at run time, whether the variable `l` does indeed refer to a `List<String>`.

If the code above is executed, heap pollution arises, as the variable `ls`, declared to be a `List<String>`, refers to a value that is not in fact a `List<String>`.

The problem cannot be identified at run time because type variables are not reified, and thus instances do not carry any information at run time regarding the type arguments used to create them.

In a simple example as given above, it may appear that it should be straightforward to identify the situation at compile time and give an error. However, in the general (and typical) case, the value of the variable `l` may be the result of an invocation of a separately compiled method, or its value may depend upon arbitrary control flow. The code above is therefore very atypical, and indeed very bad style.

Furthermore, the fact that `Object[]` is a supertype of all array types means that unsafe aliasing can occur which leads to heap pollution. For example, the following code compiles because it is statically type-correct:

```java
static void m(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;                // (1)
    String s = stringLists[0].get(0);  // (2)
}
```

Heap pollution occurs at (1) because a component in the `stringLists` array that should refer to a `List<String>` now refers to a `List<Integer>`. There is no way to detect this pollution in the presence of both a universal supertype (`Object[]`) and a non-reifiable type (the declared type of the formal parameter, `List<String>[]`). No unchecked warning is justified at (1); nevertheless, at run time, a `ClassCastException` will occur at (2).
A compile-time unchecked warning will be given at any invocation of the method above because an invocation is considered by the Java programming language's static type system to create an array whose element type, `List<String>`, is non-reifiable (§15.12.4.2). If and only if the body of the method was type-safe with respect to the variable arity parameter, then the programmer could use the `SafeVarargs` annotation to silence warnings at invocations (§9.6.4.7). Since the body of the method as written above causes heap pollution, it would be completely inappropriate to use the annotation to disable warnings for callers.

Finally, note that the `stringLists` array could be aliased through variables of types other than `Object[]`, and heap pollution could still occur. For example, the type of the array variable could be `java.util.Collection[]` - a raw element type - and the body of the method above would compile without warnings or errors and still cause heap pollution. And if the Java SE Platform defined, say, `Sequence` as a non-generic supertype of `List<T>`, then using `Sequence` as the type of `array` would also cause heap pollution.

The variable will always refer to an object that is an instance of a class that represents the parameterized type.

The value of `ls` in the example above is always an instance of a class that provides a representation of a `List`.

Assignment from an expression of a raw type to a variable of a parameterized type should only be used when combining legacy code which does not make use of parameterized types with more modern code that does.

If no operation that requires a compile-time unchecked warning to be issued takes place, and no unsafe aliasing occurs of array variables with non-reifiable element types, then heap pollution cannot occur. Note that this does not imply that heap pollution only occurs if a compile-time unchecked warning actually occurred. It is possible to run a program where some of the binaries were produced by a compiler for an older version of the Java programming language, or from sources that explicitly suppressed unchecked warnings. This practice is unhealthy at best.

Conversely, it is possible that despite executing code that could (and perhaps did) give rise to a compile-time unchecked warning, no heap pollution takes place. Indeed, good programming practice requires that the programmer satisfy herself that despite any unchecked warning, the code is correct and heap pollution will not occur.

4.12.3 Kinds of Variables

There are eight kinds of variables:

1. A **class variable** is a field declared using the keyword `static` within a class declaration (§8.3.1.1), or with or without the keyword `static` within an interface declaration (§9.3).

   A class variable is created when its class or interface is prepared (§12.3.2) and is initialized to a default value (§4.12.5). The class variable effectively ceases to exist when its class or interface is unloaded (§12.7).
2. An instance variable is a field declared within a class declaration without using the keyword static (§8.3.1.1).

If a class $\tau$ has a field $a$ that is an instance variable, then a new instance variable $a$ is created and initialized to a default value (§4.12.5) as part of each newly created object of class $\tau$ or of any class that is a subclass of $\tau$ (§8.1.4). The instance variable effectively ceases to exist when the object of which it is a field is no longer referenced, after any necessary finalization of the object (§12.6) has been completed.

3. Array components are unnamed variables that are created and initialized to default values (§4.12.5) whenever a new object that is an array is created (§10 (Arrays), §15.10.2). The array components effectively cease to exist when the array is no longer referenced.

4. Method parameters (§8.4.1) name argument values passed to a method.

For every parameter declared in a method declaration, a new parameter variable is created each time that method is invoked (§15.12). The new variable is initialized with the corresponding argument value from the method invocation. The method parameter effectively ceases to exist when the execution of the body of the method is complete.

5. Constructor parameters (§8.8.1) name argument values passed to a constructor.

For every parameter declared in a constructor declaration, a new parameter variable is created each time a class instance creation expression (§15.9) or explicit constructor invocation (§8.8.7) invokes that constructor. The new variable is initialized with the corresponding argument value from the creation expression or constructor invocation. The constructor parameter effectively ceases to exist when the execution of the body of the constructor is complete.

6. Lambda parameters (§15.27.1) name argument values passed to a lambda expression body (§15.27.2).

For every parameter declared in a lambda expression, a new parameter variable is created each time a method implemented by the lambda body is invoked (§15.12). The new variable is initialized with the corresponding argument value from the method invocation. The lambda parameter effectively ceases to exist when the execution of the lambda expression body is complete.

7. An exception parameter is created each time an exception is caught by a catch clause of a try statement (§14.20).
The new variable is initialized with the actual object associated with the exception (§11.3, §14.18). The exception parameter effectively ceases to exist when execution of the block associated with the `catch` clause is complete.

8. **Local variables** are declared by local variable declaration statements (§14.4).

Whenever the flow of control enters a block (§14.2) or `for` statement (§14.14), a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or `for` statement.

A local variable declaration statement may contain an expression which initializes the variable. The local variable with an initializing expression is not initialized, however, until the local variable declaration statement that declares it is executed. (The rules of definite assignment (§16 (Definite Assignment)) prevent the value of a local variable from being used before it has been initialized or otherwise assigned a value.) The local variable effectively ceases to exist when the execution of the block or `for` statement is complete.

Were it not for one exceptional situation, a local variable could always be regarded as being created when its local variable declaration statement is executed. The exceptional situation involves the `switch` statement (§14.11), where it is possible for control to enter a block but bypass execution of a local variable declaration statement. Because of the restrictions imposed by the rules of definite assignment (§16 (Definite Assignment)), however, the local variable declared by such a bypassed local variable declaration statement cannot be used before it has been definitely assigned a value by an assignment expression (§15.26).

**Example 4.12.3-1. Different Kinds of Variables**

```java
class Point {
    static int numPoints;   // numPoints is a class variable
    int x, y;               // x and y are instance variables
    int[] w = new int[10];  // w[0] is an array component
    int setX(int x) {       // x is a method parameter
        int oldx = this.x;  // oldx is a local variable
        this.x = x;
        return oldx;
    }
}
```

**4.12.4 final Variables**

A variable can be declared `final`. A `final` variable may only be assigned to once. It is a compile-time error if a `final` variable is assigned to unless it is definitely unassigned immediately prior to the assignment (§16 (Definite Assignment)).
Once a final variable has been assigned, it always contains the same value. If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object. This applies also to arrays, because arrays are objects; if a final variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

A blank final is a final variable whose declaration lacks an initializer.

A constant variable is a final variable of primitive type or type String that is initialized with a constant expression (§15.28). Whether a variable is a constant variable or not may have implications with respect to class initialization (§12.4.1), binary compatibility (§13.1), reachability (§14.21), and definite assignment (§16.1.1).

Three kinds of variable are implicitly declared final: a field of an interface (§9.3), a local variable declared as a resource of a try-with-resources statement (§14.20.3), and an exception parameter of a multi-catch clause (§14.20). An exception parameter of a uni-catch clause is never implicitly declared final, but may be effectively final.

Example 4.12.4-1. Final Variables

Declaring a variable final can serve as useful documentation that its value will not change and can help avoid programming errors. In this program:

```java
class Point {
    int x, y;
    int useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
```

the class Point declares a final class variable origin. The origin variable holds a reference to an object that is an instance of class Point whose coordinates are (0, 0). The value of the variable Point.origin can never change, so it always refers to the same Point object, the one created by its initializer. However, an operation on this Point object might change its state - for example, modifying its useCount or even, misleadingly, its x or y coordinate.

Certain variables that are not declared final are instead considered effectively final:

- A local variable whose declarator has an initializer (§14.4.2) is effectively final if all of the following are true:
  - It is not declared final.
It never occurs as the left hand side in an assignment expression (§15.26). (Note that the local variable declarator containing the initializer is not an assignment expression.)

– It never occurs as the operand of a prefix or postfix increment or decrement operator (§15.14, §15.15).

• A local variable whose declarator lacks an initializer is effectively final if all of the following are true:
  – It is not declared final.
  – Whenever it occurs as the left hand side in an assignment expression, it is definitely unassigned and not definitely assigned before the assignment; that is, it is definitely unassigned and not definitely assigned after the right hand side of the assignment expression (§16 (Definite Assignment)).
  – It never occurs as the operand of a prefix or postfix increment or decrement operator.

• A method, constructor, lambda, or exception parameter (§8.4.1, §8.8.1, §9.4, §15.27.1, §14.20) is treated, for the purpose of determining whether it is effectively final, as a local variable whose declarator has an initializer.

If a variable is effectively final, adding the final modifier to its declaration will not introduce any compile-time errors. Conversely, a local variable or parameter that is declared final in a valid program becomes effectively final if the final modifier is removed.

### 4.12.5 Initial Values of Variables

Every variable in a program must have a value before its value is used:

• Each class variable, instance variable, or array component is initialized with a default value when it is created (§15.9, §15.10.2):
  – For type byte, the default value is zero, that is, the value of (byte)0.
  – For type short, the default value is zero, that is, the value of (short)0.
  – For type int, the default value is zero, that is, 0.
  – For type long, the default value is zero, that is, 0L.
  – For type float, the default value is positive zero, that is, 0.0f.
  – For type double, the default value is positive zero, that is, 0.0d.
For type `char`, the default value is the null character, that is, `\u0000`.

For type `boolean`, the default value is `false`.

For all reference types (§4.3), the default value is `null`.

Each method parameter (§8.4.1) is initialized to the corresponding argument value provided by the invoker of the method (§15.12).

Each constructor parameter (§8.8.1) is initialized to the corresponding argument value provided by a class instance creation expression (§15.9) or explicit constructor invocation (§8.8.7).

An exception parameter (§14.20) is initialized to the thrown object representing the exception (§11.3, §14.18).

A local variable (§14.4, §14.14) must be explicitly given a value before it is used, by either initialization (§14.4) or assignment (§15.26), in a way that can be verified using the rules for definite assignment (§16 (Definite Assignment)).

**Example 4.12.5-1. Initial Values of Variables**

class Point {
    static int npoints;
    int x, y;
    Point root;
}

class Test {
    public static void main(String[] args) {
        System.out.println("npoints=");
        Point p = new Point();
        System.out.println("p.x=", p.x, ", p.y=", p.y);
        System.out.println("p.root=");
    }
}

This program prints:

npoints=0
p.x=0, p.y=0
p.root=null

illustrating the default initialization of `npoints`, which occurs when the class `Point` is prepared (§12.3.2), and the default initialization of `x`, `y`, and `root`, which occurs when a new `Point` is instantiated. See §12 (Execution) for a full description of all aspects of loading, linking, and initialization of classes and interfaces, plus a description of the instantiation of classes to make new class instances.
4.12.6 Types, Classes, and Interfaces

In the Java programming language, every variable and every expression has a type that can be determined at compile time. The type may be a primitive type or a reference type. Reference types include class types and interface types. Reference types are introduced by *type declarations*, which include class declarations (§8.1) and interface declarations (§9.1). We often use the term *type* to refer to either a class or an interface.

In the Java Virtual Machine, every object belongs to some particular class: the class that was mentioned in the creation expression that produced the object (§15.9), or the class whose *Class* object was used to invoke a reflective method to produce the object, or the *String* class for objects implicitly created by the string concatenation operator + (§15.18.1). This class is called the *class of the object*. An object is said to be an *instance* of its class and of all superclasses of its class.

Every array also has a class. The method *getClass*, when invoked for an array object, will return a class object (of class *Class*) that represents the *class of the array* (§10.8).

The compile-time type of a variable is always declared, and the compile-time type of an expression can be deduced at compile time. The compile-time type limits the possible values that the variable can hold at run time or the expression can produce at run time. If a run-time value is a reference that is not null, it refers to an object or array that has a class, and that class will necessarily be compatible with the compile-time type.

Even though a variable or expression may have a compile-time type that is an interface type, there are no instances of interfaces. A variable or expression whose type is an interface type can reference any object whose class implements (§8.1.5) that interface.

Sometimes a variable or expression is said to have a "run-time type". This refers to the class of the object referred to by the value of the variable or expression at run time, assuming that the value is not null.

The correspondence between compile-time types and run-time types is incomplete for two reasons:

1. At run time, classes and interfaces are loaded by the Java Virtual Machine using class loaders. Each class loader defines its own set of classes and interfaces. As a result, it is possible for two loaders to load an identical class or interface definition but produce distinct classes or interfaces at run time. Consequently, code that compiled correctly may fail at link time if the class loaders that load it are inconsistent.
2. Type variables (§4.4) and type arguments (§4.5.1) are not reified at run time. As a result, the same class or interface at run time represents multiple parameterized types (§4.5) from compile time. Specifically, all compile-time parameterizations of a given generic type (§8.1.2, §9.1.2) share a single run-time representation.

Under certain conditions, it is possible that a variable of a parameterized type refers to an object that is not of that parameterized type. This situation is known as heap pollution (§4.12.2). The variable will always refer to an object that is an instance of a class that represents the parameterized type.

Example 4.12.6.1. Type of a Variable versus Class of an Object

```java
interface Colorable {
    void setColor(byte r, byte g, byte b);
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
    byte r, g, b;
    public void setColor(byte rv, byte gv, byte bv) {
        r = rv; g = gv; b = bv;
    }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;
    }
}
```

In this example:

- The local variable `p` of the method `main` of class `Test` has type `Point` and is initially assigned a reference to a new instance of class `Point`.
- The local variable `cp` similarly has as its type `ColoredPoint`, and is initially assigned a reference to a new instance of class `ColoredPoint`.
- The assignment of the value of `cp` to the variable `p` causes `p` to hold a reference to a `ColoredPoint` object. This is permitted because `ColoredPoint` is a subclass of `Point`, so the class `ColoredPoint` is assignment-compatible (§5.2) with the type.
Point. A ColoredPoint object includes support for all the methods of a Point. In addition to its particular fields r, g, and b, it has the fields of class Point, namely x and y.

- The local variable c has as its type the interface type Colorable, so it can hold a reference to any object whose class implements Colorable; specifically, it can hold a reference to a ColoredPoint.

Note that an expression such as new Colorable() is not valid because it is not possible to create an instance of an interface, only of a class. However, the expression new Colorable() { public void setColor... } is valid because it declares an anonymous class (§15.9.5) that implements the Colorable interface.
EVERY expression written in the Java programming language either produces no result (§15.1) or has a type that can be deduced at compile time (§15.3). When an expression appears in most contexts, it must be compatible with a type expected in that context; this type is called the target type. For convenience, compatibility of an expression with its surrounding context is facilitated in two ways:

• First, for some expressions, termed poly expressions (§15.2), the deduced type can be influenced by the target type. The same expression can have different types in different contexts.

• Second, after the type of the expression has been deduced, an implicit conversion from the type of the expression to the target type can sometimes be performed.

If neither strategy is able to produce the appropriate type, a compile-time error occurs.

The rules determining whether an expression is a poly expression, and if so, its type and compatibility in a particular context, vary depending on the kind of context and the form of the expression. In addition to influencing the type of the expression, the target type may in some cases influence the run time behavior of the expression in order to produce a value of the appropriate type.

Similarly, the rules determining whether a target type allows an implicit conversion vary depending on the kind of context, the type of the expression, and, in one special case, the value of a constant expression (§15.28). A conversion from type \( S \) to type \( T \) allows an expression of type \( S \) to be treated at compile time as if it had type \( T \) instead. In some cases this will require a corresponding action at run time to check the validity of the conversion or to translate the run-time value of the expression into a form appropriate for the new type \( T \).
Example 5.0-1. Conversions at Compile Time and Run Time

• A conversion from type \texttt{Object} to type \texttt{Thread} requires a run-time check to make sure that the run-time value is actually an instance of class \texttt{Thread} or one of its subclasses; if it is not, an exception is thrown.

• A conversion from type \texttt{Thread} to type \texttt{Object} requires no run-time action; \texttt{Thread} is a subclass of \texttt{Object}, so any reference produced by an expression of type \texttt{Thread} is a valid reference value of type \texttt{Object}.

• A conversion from type \texttt{int} to type \texttt{long} requires run-time sign-extension of a 32-bit integer value to the 64-bit \texttt{long} representation. No information is lost.

• A conversion from type \texttt{double} to type \texttt{long} requires a non-trivial translation from a 64-bit floating-point value to the 64-bit integer representation. Depending on the actual run-time value, information may be lost.

The conversions possible in the Java programming language are grouped into several broad categories:

• Identity conversions
• Widening primitive conversions
• Narrowing primitive conversions
• Widening reference conversions
• Narrowing reference conversions
• Boxing conversions
• Unboxing conversions
• Unchecked conversions
• Capture conversions
• String conversions
• Value set conversions

There are six kinds of conversion contexts in which poly expressions may be influenced by context or implicit conversions may occur. Each kind of context has different rules for poly expression typing and allows conversions in some of the categories above but not others. The contexts are:

• Assignment contexts (§5.2, §15.26), in which an expression's value is bound to a named variable. Primitive and reference types are subject to widening, values may be boxed or unboxed, and some primitive constant expressions may be subject to narrowing. An unchecked conversion may also occur.
• **Strict invocation contexts** (§5.3, §15.9, §15.12), in which an argument is bound to a formal parameter of a constructor or method. Widening primitive, widening reference, and unchecked conversions may occur.

• **Loose invocation contexts** (§5.3, §15.9, §15.12), in which, like strict invocation contexts, an argument is bound to a formal parameter. Method or constructor invocations may provide this context if no applicable declaration can be found using only strict invocation contexts. In addition to widening and unchecked conversions, this context allows boxing and unboxing conversions to occur.

• **String contexts** (§5.4, §15.18.1), in which a value of any type is converted to an object of type `String`.

• **Casting contexts** (§5.5), in which an expression's value is converted to a type explicitly specified by a cast operator (§15.16). Casting contexts are more inclusive than assignment or loose invocation contexts, allowing any specific conversion other than a string conversion, but certain casts to a reference type are checked for correctness at run time.

• **Numeric contexts** (§5.6), in which the operands of a numeric operator may be widened to a common type so that an operation can be performed.

The term "conversion" is also used to describe, without being specific, any conversions allowed in a particular context. For example, we say that an expression that is the initializer of a local variable is subject to "assignment conversion", meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the assignment context.

**Example 5.0-2. Conversions In Various Contexts**

```java
class Test {
    public static void main(String[] args) {
        // Casting conversion (5.4) of a float literal to type int. Without the cast operator, this would be a compile-time error, because this is a narrowing conversion (5.1.3):
        int i = (int)12.5f;

        // String conversion (5.4) of i's int value:
        System.out.println("(int)12.5f==" + i);

        // Assignment conversion (5.2) of i's value to type float. This is a widening conversion (5.1.2):
        float f = i;

        // String conversion of f's float value:
        System.out.println("after float widening: " + f);

        // Numeric promotion (5.6) of i's value to type
```
5.1 Kinds of Conversion

Specific type conversions in the Java programming language are divided into 13 categories.

5.1.1 Identity Conversion

A conversion from a type to that same type is permitted for any type.

This may seem trivial, but it has two practical consequences. First, it is always permitted for an expression to have the desired type to begin with, thus allowing the simply stated rule that every expression is subject to conversion, if only a trivial identity conversion. Second, it implies that it is permitted for a program to include redundant cast operators for the sake of clarity.

5.1.2 Widening Primitive Conversion

19 specific conversions on primitive types are called the *widening primitive conversions*:

- byte to short, int, long, float, or double
• short to int, long, float, or double
• char to int, long, float, or double
• int to long, float, or double
• long to float or double
• float to double

A widening primitive conversion does not lose information about the overall magnitude of a numeric value in the following cases, where the numeric value is preserved exactly:

• from an integral type to another integral type
• from byte, short, or char to a floating point type
• from int to double
• from float to double in a strictfp expression (§15.4)

A widening primitive conversion from float to double that is not strictfp may lose information about the overall magnitude of the converted value.

A widening primitive conversion from int to float, or from long to float, or from long to double, may result in loss of precision - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).

A widening conversion of a signed integer value to an integral type T simply sign-extends the two's-complement representation of the integer value to fill the wider format.

A widening conversion of a char to an integral type T zero-extends the representation of the char value to fill the wider format.

Despite the fact that loss of precision may occur, a widening primitive conversion never results in a run-time exception (§11.1.1).

Example 5.1.2-1. Widening Primitive Conversion

```java
class Test {
    public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```
This program prints:

\[-46\]

thus indicating that information was lost during the conversion from type `int` to type `float` because values of type `float` are not precise to nine significant digits.

### 5.1.3 Narrowing Primitive Conversion

22 specific conversions on primitive types are called the *narrowing primitive conversions*:

- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`
- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

A narrowing primitive conversion from `double` to `float` is governed by the IEEE 754 rounding rules (§4.2.4). This conversion can lose precision, but also lose range, resulting in a `float` zero from a nonzero `double` and a `float` infinity from a finite `double`. A `double` NaN is converted to a `float` NaN and a `double` infinity is converted to the same-signed `float` infinity.

A narrowing conversion of a signed integer to an integral type `\( \tau \)` simply discards all but the `n` lowest order bits, where `n` is the number of bits used to represent type `\( \tau \)`. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

A narrowing conversion of a `char` to an integral type `\( \tau \)` likewise simply discards all but the `n` lowest order bits, where `n` is the number of bits used to represent type `\( \tau \)`. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the resulting value to be a negative number, even though chars represent 16-bit unsigned integer values.

A narrowing conversion of a floating-point number to an integral type `\( \tau \)` takes two steps:
1. In the first step, the floating-point number is converted either to a \texttt{long}, if \( T \) is \texttt{long}, or to an \texttt{int}, if \( T \) is \texttt{byte}, \texttt{short}, \texttt{char}, or \texttt{int}, as follows:

   - If the floating-point number is NaN (§4.2.3), the result of the first step of the conversion is an \texttt{int} or \texttt{long} 0.

   - Otherwise, if the floating-point number is not an infinity, the floating-point value is rounded to an integer value \( V \), rounding toward zero using IEEE 754 round-toward-zero mode (§4.2.3). Then there are two cases:
     a. If \( T\) is \texttt{long}, and this integer value can be represented as a \texttt{long}, then the result of the first step is the \texttt{long} value \( V \).
     b. Otherwise, if this integer value can be represented as an \texttt{int}, then the result of the first step is the \texttt{int} value \( V \).

   - Otherwise, one of the following two cases must be true:
     a. The value must be too small (a negative value of large magnitude or negative infinity), and the result of the first step is the smallest representable value of type \texttt{int} or \texttt{long}.
     b. The value must be too large (a positive value of large magnitude or positive infinity), and the result of the first step is the largest representable value of type \texttt{int} or \texttt{long}.

2. In the second step:
   - If \( T\) is \texttt{int} or \texttt{long}, the result of the conversion is the result of the first step.
   - If \( T\) is \texttt{byte}, \texttt{char}, or \texttt{short}, the result of the conversion is the result of a narrowing conversion to type \( T \) (§5.1.3) of the result of the first step.

Despite the fact that overflow, underflow, or other loss of information may occur, a narrowing primitive conversion never results in a run-time exception (§11.1.1).

Example 5.1.3-1. Narrowing Primitive Conversion

```java
class Test {
    public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: "+(long)fmin + ".." + (long)fmax);
        System.out.println("int: "+(int)fmin + ".." + (int)fmax);
        System.out.println("short: "+(short)fmin + ".." + (short)fmax);
        System.out.println("char: "+(int)(char)fmin + ".." + (int)(char)fmax);
    }
}
```
Kinds of Conversion

CONVERSIONS AND CONTEXTS

This program produces the output:

long: -922372036854775808..922372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1

The results for char, int, and long are unsurprising, producing the minimum and maximum representable values of the type.

The results for byte and short lose information about the sign and magnitude of the numeric values and also lose precision. The results can be understood by examining the low order bits of the minimum and maximum int. The minimum int is, in hexadecimal, 0x80000000, and the maximum int is 0x7fffffff. This explains the short results, which are the low 16 bits of these values, namely, 0x0000 and 0xffff; it explains the char results, which also are the low 16 bits of these values, namely, '\u0000' and '\uffff'; and it explains the byte results, which are the low 8 bits of these values, namely, 0x00 and 0xff.

Example 5.1.3-2. Narrowing Primitive Conversions that lose information

```java
class Test {
    public static void main(String[] args) {
        // A narrowing of int to short loses high bits:
        System.out.println("(short)0x12345678==0x" +
                            Integer.toHexString((short)0x12345678));
        // An int value too big for byte changes sign and magnitude:
        System.out.println("(byte)255==" + (byte)255);
        // A float value too big to fit gives largest int value:
        System.out.println("(int)1e20f==" + (int)1e20f);
        // A NaN converted to int yields zero:
        System.out.println("(int)NaN==" + (int)Float.NaN);
        // A double value too large for float yields infinity:
        System.out.println("(float)-1e100==" + (float)-1e100);
        // A double value too small for float underflows to zero:
        System.out.println("(float)1e-50==" + (float)1e-50);
    }
}
```

This program produces the output:
5.1.4 Widening and Narrowing Primitive Conversion

The following conversion combines both widening and narrowing primitive conversions:

- `byte` to `char`

First, the `byte` is converted to an `int` via widening primitive conversion (§5.1.2), and then the resulting `int` is converted to a `char` by narrowing primitive conversion (§5.1.3).

5.1.5 Widening Reference Conversion

A widening reference conversion exists from any reference type `S` to any reference type `T`, provided `S` is a subtype of `T` (§4.10).

Widening reference conversions never require a special action at run time and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

The null type is not a reference type (§4.1), and so a widening reference conversion does not exist from the null type to a reference type. However, many conversion contexts explicitly allow the null type to be converted to a reference type.

5.1.6 Narrowing Reference Conversion

A narrowing reference conversion treats expressions of a reference type `S` as expressions of a different reference type `T`, where `S` is not a subtype of `T`. The supported pairs of types are defined in §5.1.6.1. Unlike widening reference conversion, the types need not be directly related. However, there are restrictions that prohibit conversion between certain pairs of types when it can be statically proven that no value can be of both types.

A narrowing reference conversion may require a test at run time to validate that a value of type `S` is a legitimate value of type `T`. However, due to the lack of parameterized type information at run time, some conversions cannot be fully validated by a run time test; they are flagged at compile time (§5.1.6.2).
For conversions that can be fully validated by a run time test, and for certain conversions that involve parameterized type information but can still be partially validated at run time, a `ClassCastException` is thrown if the test fails (§5.1.6.3).

### 5.1.6.1 Allowed Narrowing Reference Conversion

A narrowing reference conversion exists from reference type \( S \) to reference type \( T \) if all of the following are true:

- \( S \) is not a subtype of \( T \) (§4.10)
- If there exists a parameterized type \( X \) that is a supertype of \( T \), and a parameterized type \( Y \) that is a supertype of \( S \), such that the erasures of \( X \) and \( Y \) are the same, then \( X \) and \( Y \) are not provably distinct (§4.5).

Using types from the `java.util` package as an example, no narrowing reference conversion exists from `ArrayList<String>` to `ArrayList<Object>`, or vice versa, because the type arguments `String` and `Object` are provably distinct. For the same reason, no narrowing reference conversion exists from `ArrayList<String>` to `List<Object>`, or vice versa. The rejection of provably distinct types is a simple static gate to prevent "stupid" narrowing reference conversions.

- One of the following cases applies:
  1. \( S \) and \( T \) are class types, and either \(|S| < : |T|\) or \(|T| < : |S|\).
  2. \( S \) and \( T \) are interface types.
  3. \( S \) is a class type, \( T \) is an interface type, and \( S \) does not name a `final` class (§8.1.1).
  4. \( S \) is a class type, \( T \) is an interface type, and \( S \) names a `final` class that implements the interface named by \( T \).
  5. \( S \) is an interface type, \( T \) is a class type, and \( T \) does not name a `final` class.
  6. \( S \) is an interface type, \( T \) is a class type, and \( T \) names a `final` class that implements the interface named by \( S \).
  7. \( S \) is the class type `Object` or the interface type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays (§10.8)), and \( T \) is an array type.
  8. \( S \) is an array type \( SC[] \), that is, an array of components of type \( SC \); \( T \) is an array type \( TC[] \), that is, an array of components of type \( TC \); and a narrowing reference conversion exists from \( SC \) to \( TC \).
  9. \( S \) is a type variable, and a narrowing reference conversion exists from the upper bound of \( S \) to \( T \).
10. $T$ is a type variable, and either a widening reference conversion or a narrowing reference conversion exists from $S$ to the upper bound of $T$.

11. $S$ is an intersection type $S_1 \& \ldots \& S_n$, and for all $i (1 \leq i \leq n)$, either a widening reference conversion or a narrowing reference conversion exists from $S_i$ to $T$.

12. $T$ is an intersection type $T_1 \& \ldots \& T_n$, and for all $i (1 \leq i \leq n)$, either a widening reference conversion or a narrowing reference conversion exists from $S$ to $T_i$.

5.1.6.2 Checked and Unchecked Narrowing Reference Conversions

A narrowing reference conversion is either checked or unchecked. These terms refer to the ability of the Java Virtual Machine to validate, or not, the type correctness of the conversion.

If a narrowing reference conversion is unchecked, then the Java Virtual Machine will not be able to fully validate its type correctness, possibly leading to heap pollution (§4.12.2). To flag this to the programmer, an unchecked narrowing reference conversion causes a compile-time unchecked warning, unless suppressed by @SuppressWarnings (§9.6.4.5). In contrast, if a narrowing reference conversion is not unchecked, then it is checked; the Java Virtual Machine will be able to fully validate its type correctness, so no warning is given at compile time.

The unchecked narrowing reference conversions are as follows:

- A narrowing reference conversion from a type $S$ to a parameterized class or interface type $T$ is unchecked, unless at least one of the following is true:
  - All of the type arguments of $T$ are unbounded wildcards.
  - $T <: S$, and $S$ has no subtype $X$ other than $T$ where the type arguments of $X$ are not contained in the type arguments of $T$.

- A narrowing reference conversion from a type $S$ to a type variable $T$ is unchecked.

- A narrowing reference conversion from a type $S$ to an intersection type $T_1 \& \ldots \& T_n$ is unchecked if there exists a $T_i (1 \leq i \leq n)$ such that $S$ is not a subtype of $T_i$ and a narrowing reference conversion from $S$ to $T_i$ is unchecked.

5.1.6.3 Narrowing Reference Conversions at Run Time

All checked narrowing reference conversions require a validity check at run time. Primarily, these conversions are to class and interface types that are not parameterized.
Some **unchecked** narrowing reference conversions require a validity check at run time. This depends on whether the unchecked narrowing reference conversion is **completely unchecked** or **partially unchecked**. A partially unchecked narrowing reference conversion requires a validity check at run time, while a completely unchecked narrowing reference conversion does not.

These terms refer to the compatibility of the types involved in the conversion when viewed as raw types. If the conversion is conceptually an "upcast", then the conversion is **completely unchecked**; no run time test is needed because the conversion is legal in the non-generic type system of the Java Virtual Machine. In contrast, if the conversion is conceptually a "downcast", then the conversion is **partially unchecked**; even in the non-generic type system of the Java Virtual Machine, a run time check is needed to test the compatibility of the (raw) types involved in the conversion.

Using types from the `java.util` package as an example, a conversion from `ArrayList<String>` to `Collection<T>` is completely unchecked, because the (raw) type `ArrayList` is a subtype of the (raw) type `Collection` in the Java Virtual Machine. In contrast, a conversion from `Collection<T>` to `ArrayList<String>` is partially unchecked, because the (raw) type `Collection` is not a subtype of the (raw) type `ArrayList` in the Java Virtual Machine.

The categorization of an unchecked narrowing reference conversion is as follows:

- An unchecked narrowing reference conversion from `S` to a non-intersection type `T` is completely unchecked if `|S| <: |T|`.

Otherwise, it is partially unchecked.

- An unchecked narrowing reference conversion from `S` to an intersection type `T_1 & ... & T_n` is completely unchecked if, for all `i` (1 ≤ `i` ≤ `n`), either `S <: T_i` or a narrowing reference conversion from `S` to `T_i` is completely unchecked.

Otherwise, it is partially unchecked.

The run time validity check for a checked or partially unchecked narrowing reference conversion is as follows:

- If the value at run time is `null`, then the conversion is allowed.

- Otherwise, let `R` be the class of the object referred to by the value, and let `T` be the erasure (§4.6) of the type being converted to. Then:
  - If `R` is an ordinary class (not an array class):
    - If `T` is a class type, then `R` must be either the same class as `T` (§4.3.4) or a subclass of `T`, or a `ClassCastException` is thrown.
    - If `T` is an interface type, then `R` must implement interface `T` (§8.1.5), or a `ClassCastException` is thrown.
If $T$ is an array type, then a `ClassCastException` is thrown.

- If $R$ is an interface:

  Note that $R$ cannot be an interface when these rules are first applied for any given conversion, but $R$ may be an interface if the rules are applied recursively because the run-time reference value may refer to an array whose element type is an interface type.

  If $T$ is a class type, then $T$ must be `Object` (§4.3.2), or a `ClassCastException` is thrown.

  If $T$ is an interface type, then $R$ must be either the same interface as $T$ or a subinterface of $T$, or a `ClassCastException` is thrown.

  If $T$ is an array type, then a `ClassCastException` is thrown.

- If $R$ is a class representing an array type $RC[]$, that is, an array of components of type $RC$:

  If $T$ is a class type, then $T$ must be `Object` (§4.3.2), or a `ClassCastException` is thrown.

  If $T$ is an interface type, then $T$ must be the type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays), or a `ClassCastException` is thrown.

  If $T$ is an array type $TC[]$, that is, an array of components of type $TC$, then a `ClassCastException` is thrown unless either $TC$ and $RC$ are the same primitive type, or $TC$ and $RC$ are reference types and are allowed by a recursive application of these run-time rules.

If the conversion is to an intersection type $T_1 \& \ldots \& T_n$, then for all $i$ ($1 \leq i \leq n$), any run-time check required for a conversion from $S$ to $T_i$ is also required for the conversion to the intersection type.

### 5.1.7 Boxing Conversion

Boxing conversion treats expressions of a primitive type as expressions of a corresponding reference type. Specifically, the following nine conversions are called the boxing conversions:

- From type `boolean` to type `Boolean`
- From type `byte` to type `Byte`
- From type `short` to type `Short`
5.1 Kinds of Conversion

CONVERSIONS AND CONTEXTS

- From type char to type Character
- From type int to type Integer
- From type long to type Long
- From type float to type Float
- From type double to type Double
- From the null type to the null type

This rule is necessary because the conditional operator (§15.25) applies boxing conversion to the types of its operands, and uses the result in further calculations.

At run time, boxing conversion proceeds as follows:

- If \( p \) is a value of type boolean, then boxing conversion converts \( p \) into a reference \( r \) of class and type Boolean, such that \( r\).booleanValue() == \( p \)
- If \( p \) is a value of type byte, then boxing conversion converts \( p \) into a reference \( r \) of class and type Byte, such that \( r\).byteValue() == \( p \)
- If \( p \) is a value of type char, then boxing conversion converts \( p \) into a reference \( r \) of class and type Character, such that \( r\).charValue() == \( p \)
- If \( p \) is a value of type short, then boxing conversion converts \( p \) into a reference \( r \) of class and type Short, such that \( r\).shortValue() == \( p \)
- If \( p \) is a value of type int, then boxing conversion converts \( p \) into a reference \( r \) of class and type Integer, such that \( r\).intValue() == \( p \)
- If \( p \) is a value of type long, then boxing conversion converts \( p \) into a reference \( r \) of class and type Long, such that \( r\).longValue() == \( p \)
- If \( p \) is a value of type float then:
  - If \( p \) is not NaN, then boxing conversion converts \( p \) into a reference \( r \) of class and type Float, such that \( r\).floatValue() evaluates to \( p \)
  - Otherwise, boxing conversion converts \( p \) into a reference \( r \) of class and type Float such that \( r\).isNaN() evaluates to true
- If \( p \) is a value of type double, then:
  - If \( p \) is not NaN, boxing conversion converts \( p \) into a reference \( r \) of class and type Double, such that \( r\).doubleValue() evaluates to \( p \)
  - Otherwise, boxing conversion converts \( p \) into a reference \( r \) of class and type Double such that \( r\).isNaN() evaluates to true
• If \( p \) is a value of any other type, boxing conversion is equivalent to an identity conversion (§5.1.1).

If the value \( p \) being boxed is the result of evaluating a constant expression (§15.28) of type boolean, char, short, int, or long, and the result is true, false, a character in the range '\u0000' to '\u007f' inclusive, or an integer in the range -128 to 127 inclusive, then let \( a \) and \( b \) be the results of any two boxing conversions of \( p \). It is always the case that \( a == b \).

Ideally, boxing a primitive value would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rule above is a pragmatic compromise, requiring that certain common values always be boxed into indistinguishable objects. The implementation may cache these, lazily or eagerly. For other values, the rule disallows any assumptions about the identity of the boxed values on the programmer’s part. This allows (but does not require) sharing of some or all of these references.

This ensures that in most common cases, the behavior will be the desired one, without imposing an undue performance penalty, especially on small devices. Less memory-limited implementations might, for example, cache all char and short values, as well as int and long values in the range of -32K to +32K.

A boxing conversion may result in an OutOfMemoryError if a new instance of one of the wrapper classes (Boolean, Byte, Character, Short, Integer, Long, Float, or Double) needs to be allocated and insufficient storage is available.

### 5.1.8 Unboxing Conversion

Unboxing conversion treats expressions of a reference type as expressions of a corresponding primitive type. Specifically, the following eight conversions are called the unboxing conversions:

• From type Boolean to type boolean
• From type Byte to type byte
• From type Short to type short
• From type Character to type char
• From type Integer to type int
• From type Long to type long
• From type Float to type float
• From type Double to type double

At run time, unboxing conversion proceeds as follows:
- If \( r \) is a reference of type `Boolean`, then unboxing conversion converts \( r \) into \( r\.booleanValue() \)
- If \( r \) is a reference of type `Byte`, then unboxing conversion converts \( r \) into \( r\.byteValue() \)
- If \( r \) is a reference of type `Character`, then unboxing conversion converts \( r \) into \( r\.charValue() \)
- If \( r \) is a reference of type `Short`, then unboxing conversion converts \( r \) into \( r\.shortValue() \)
- If \( r \) is a reference of type `Integer`, then unboxing conversion converts \( r \) into \( r\.intValue() \)
- If \( r \) is a reference of type `Long`, then unboxing conversion converts \( r \) into \( r\.longValue() \)
- If \( r \) is a reference of type `Float`, unboxing conversion converts \( r \) into \( r\.floatValue() \)
- If \( r \) is a reference of type `Double`, then unboxing conversion converts \( r \) into \( r\.doubleValue() \)
- If \( r \) is null, unboxing conversion throws a `NullPointerException`

A type is said to be *convertible to a numeric type* if it is a numeric type (§4.2), or it is a reference type that may be converted to a numeric type by unboxing conversion.

A type is said to be *convertible to an integral type* if it is an integral type, or it is a reference type that may be converted to an integral type by unboxing conversion.

### 5.1.9 Unchecked Conversion

Let \( \mathcal{G} \) name a generic type declaration with \( n \) type parameters.

There is an *unchecked conversion* from the raw class or interface type (§4.8) \( \mathcal{G} \) to any parameterized type of the form \( \mathcal{G}\langle T_1, \ldots, T_n \rangle \).

There is an *unchecked conversion* from the raw array type \( \mathcal{G}[] \) to any array type of the form \( \mathcal{G}\langle T_1, \ldots, T_n \rangle [[]] \). (The notation \([[]]_k \) indicates an array type of \( k \) dimensions.)

Use of an unchecked conversion causes a compile-time *unchecked warning* unless all type arguments \( T_i \) (\( 1 \leq i \leq n \)) are unbounded wildcards (§4.5.1), or the warning is suppressed by \@SuppressWarnings (§9.6.4.5).

Unchecked conversion is used to enable a smooth interoperation of legacy code, written before the introduction of generic types, with libraries that have undergone a conversion
to use genericity (a process we call generification). In such circumstances (most notably, clients of the Collections Framework in \texttt{java.util}), legacy code uses raw types (e.g. \texttt{Collection} instead of \texttt{Collection<String>}). Expressions of raw types are passed as arguments to library methods that use parameterized versions of those same types as the types of their corresponding formal parameters.

Such calls cannot be shown to be statically safe under the type system using generics. Rejecting such calls would invalidate large bodies of existing code, and prevent them from using newer versions of the libraries. This in turn, would discourage library vendors from taking advantage of genericity. To prevent such an unwelcome turn of events, a raw type may be converted to an arbitrary invocation of the generic type declaration to which the raw type refers. While the conversion is unsound, it is tolerated as a concession to practicality. An unchecked warning is issued in such cases.

### 5.1.10 Capture Conversion

Let \( G \) name a generic type declaration (§8.1.2, §9.1.2) with \( n \) type parameters \( A_1, \ldots, A_n \) with corresponding bounds \( U_1, \ldots, U_n \).

There exists a \textit{capture conversion} from a parameterized type \( G<T_1, \ldots, T_n> \) (§4.5) to a parameterized type \( G<S_1, \ldots, S_n> \), where, for \( 1 \leq i \leq n \):

- If \( T_i \) is a wildcard type argument (§4.5.1) of the form \(?\), then \( S_i \) is a fresh type variable whose upper bound is \( U_i[A_1:=S_1, \ldots, A_n:=S_n] \) and whose lower bound is the null type (§4.1).

- If \( T_i \) is a wildcard type argument of the form \(? \text{ extends } B_i\), then \( S_i \) is a fresh type variable whose upper bound is \( \text{glb}(B_i, U_i[A_1:=S_1, \ldots, A_n:=S_n]) \) and whose lower bound is the null type.

\( \text{glb}(V_1, \ldots, V_m) \) is defined as \( V_1 \& \ldots \& V_m \).

It is a compile-time error if, for any two classes (not interfaces) \( V_1 \) and \( V_3 \), \( V_1 \) is not a subclass of \( V_3 \) or vice versa.

- If \( T_i \) is a wildcard type argument of the form \(? \text{ super } B_i\), then \( S_i \) is a fresh type variable whose upper bound is \( U_i[A_1:=S_1, \ldots, A_n:=S_n] \) and whose lower bound is \( B_i \).

- Otherwise, \( S_i = T_i \).

Capture conversion on any type other than a parameterized type (§4.5) acts as an identity conversion (§5.1.1).

Capture conversion is not applied recursively.

Capture conversion never requires a special action at run time and therefore never throws an exception at run time.
Capture conversion is designed to make wildcards more useful. To understand the motivation, let's begin by looking at the method `java.util.Collections.reverse()`:

```java
public static void reverse(List<?> list);
```

The method reverses the list provided as a parameter. It works for any type of list, and so the use of the wildcard type `List<?>` as the type of the formal parameter is entirely appropriate.

Now consider how one would implement `reverse()`:

```java
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size() - i - 1));
    }
}
```

The implementation needs to copy the list, extract elements from the copy, and insert them into the original. To do this in a type-safe manner, we need to give a name, `T`, to the element type of the incoming list. We do this in the private service method `rev()`. This requires us to pass the incoming argument list, of type `List<?>`, as an argument to `rev()`. In general, `List<?>` is a list of unknown type. It is not a subtype of `List<T>`, for any type `T`. Allowing such a subtype relation would be unsound. Given the method:

```java
public static <T> void fill(List<T> l, T obj)
```

the following code would undermine the type system:

```java
List<String> ls = new ArrayList<String>();
List<?> l = ls;
Collections.fill(l, new Object());  // not legal - but assume it was!
String s = ls.get(0); // ClassCastException - ls contains Objects, not Strings.
```

So, without some special dispensation, we can see that the call from `reverse()` to `rev()` would be disallowed. If this were the case, the author of `reverse()` would be forced to write its signature as:

```java
public static <T> void reverse(List<T> list)
```

This is undesirable, as it exposes implementation information to the caller. Worse, the designer of an API might reason that the signature using a wildcard is what the callers of the API require, and only later realize that a type safe implementation was precluded.

The call from `reverse()` to `rev()` is in fact harmless, but it cannot be justified on the basis of a general subtyping relation between `List<?>` and `List<T>`. The call is harmless, because the incoming argument is doubtless a list of some type (albeit an unknown one). If we can capture this unknown type in a type variable `X`, we can infer `T` to be `X`. That is the essence of capture conversion. The specification of course must cope with complications,
like non-trivial (and possibly recursively defined) upper or lower bounds, the presence of multiple arguments etc.

Mathematically sophisticated readers will want to relate capture conversion to established type theory. Readers unfamiliar with type theory can skip this discussion - or else study a suitable text, such as *Types and Programming Languages* by Benjamin Pierce, and then revisit this section.

Here then is a brief summary of the relationship of capture conversion to established type theoretical notions. Wildcard types are a restricted form of existential types. Capture conversion corresponds loosely to an opening of a value of existential type. A capture conversion of an expression \( e \) can be thought of as an open of \( e \) in a scope that comprises the top level expression that encloses \( e \).

The classical open operation on existentials requires that the captured type variable must not escape the opened expression. The open that corresponds to capture conversion is always on a scope sufficiently large that the captured type variable can never be visible outside that scope. The advantage of this scheme is that there is no need for a close operation, as defined in the paper *On Variance-Based Subtyping for Parametric Types* by Atsushi Igarashi and Mirko Viroli, in the proceedings of the 16th European Conference on Object Oriented Programming (ECOOP 2002). For a formal account of wildcards, see *Wild FJ* by Mads Torgersen, Erik Ernst and Christian Plesner Hansen, in the 12th workshop on Foundations of Object Oriented Programming (FOOL 2005).

### 5.1.11 String Conversion

Any type may be converted to type `String` by *string conversion*.

A value \( x \) of primitive type \( T \) is first converted to a reference value as if by giving it as an argument to an appropriate class instance creation expression (§15.9):

- If \( T \) is `boolean`, then use `new Boolean(x)`.
- If \( T \) is `char`, then use `new Character(x)`.
- If \( T \) is `byte`, `short`, or `int`, then use `new Integer(x)`.
- If \( T \) is `long`, then use `new Long(x)`.
- If \( T \) is `float`, then use `new Float(x)`.
- If \( T \) is `double`, then use `new Double(x)`.

This reference value is then converted to type `String` by string conversion.

Now only reference values need to be considered:

- If the reference is `null`, it is converted to the string "null" (four ASCII characters n, u, l, l).
• Otherwise, the conversion is performed as if by an invocation of the `toString` method of the referenced object with no arguments; but if the result of invoking the `toString` method is `null`, then the string "null" is used instead.

The `toString` method is defined by the primordial class `Object` (§4.3.2). Many classes override it, notably `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `String`.

5.1.12 Forbidden Conversions

Any conversion that is not explicitly allowed is forbidden.

5.1.13 Value Set Conversion

Value set conversion is the process of mapping a floating-point value from one value set to another without changing its type.

Within an expression that is not FP-strict (§15.4), value set conversion provides choices to an implementation of the Java programming language:

• If the value is an element of the float-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the float value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).

• If the value is an element of the double-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the double value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).

Within an FP-strict expression (§15.4), value set conversion does not provide any choices; every implementation must behave in the same way:

• If the value is of type `float` and is not an element of the float value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.

• If the value is of type `double` and is not an element of the double value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.
Within an FP-strict expression, mapping values from the float-extended-exponent value set or double-extended-exponent value set is necessary only when a method is invoked whose declaration is not FP-strict and the implementation has chosen to represent the result of the method invocation as an element of an extended-exponent value set.

Whether in FP-strict code or code that is not FP-strict, value set conversion always leaves unchanged any value whose type is neither float nor double.

### 5.2 Assignment Contexts

Assignment contexts allow the value of an expression to be assigned (§15.26) to a variable; the type of the expression must be converted to the type of the variable.

Assignment contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a widening reference conversion followed by an unboxing conversion
- a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
- a boxing conversion (§5.1.7)
- a boxing conversion followed by a widening reference conversion
- an unboxing conversion (§5.1.8)
- an unboxing conversion followed by a widening primitive conversion

If, after the conversions listed above have been applied, the resulting type is a raw type (§4.8), an unchecked conversion (§5.1.9) may then be applied.

In addition, if the expression is a constant expression (§15.28) of type byte, short, char, or int:

- A narrowing primitive conversion may be used if the variable is of type byte, short, or char, and the value of the constant expression is representable in the type of the variable.
5.2 Assignment Contexts

CONVERSIONS AND CONTEXTS

- A narrowing primitive conversion followed by a boxing conversion may be used if the variable is of type Byte, Short, or Character, and the value of the constant expression is representable in the type byte, short, or char respectively.

The compile-time narrowing of constant expressions means that code such as:

```java
byte theAnswer = 42;
```

is allowed. Without the narrowing, the fact that the integer literal 42 has type int would mean that a cast to byte would be required:

```java
byte theAnswer = (byte)42;  // cast is permitted but not required
```

Finally, a value of the null type (the null reference is the only such value) may be assigned to any reference type, resulting in a null reference of that type.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation (§4.10).

An example of such an illegal chain would be:

```java
Integer, Comparable<Integer>, Comparable, Comparable<String>
```

The first three elements of the chain are related by widening reference conversion, while the last entry is derived from its predecessor by unchecked conversion. However, this is not a valid assignment conversion, because the chain contains two parameterized types, Comparable<Integer> and Comparable<String>, that are not subtypes.

If the type of an expression can be converted to the type of a variable by assignment conversion, we say the expression (or its value) is assignable to the variable or, equivalently, that the type of the expression is assignment compatible with the type of the variable.

If the type of the variable is float or double, then value set conversion (§5.1.13) is applied to the value v that is the result of the conversion(s):

- If v is of type float and is an element of the float-extended-exponent value set, then the implementation must map v to the nearest element of the float value set. This conversion may result in overflow or underflow.

- If v is of type double and is an element of the double-extended-exponent value set, then the implementation must map v to the nearest element of the double value set. This conversion may result in overflow or underflow.

The only exceptions that may arise from conversions in an assignment context are:
• A **ClassCastException** if, after the conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure (§4.6) of the type of the variable.

  This circumstance can only arise as a result of heap pollution (§4.12.2). In practice, implementations need only perform casts when accessing a field or method of an object of parameterized type when the erased type of the field, or the erased return type of the method, differ from its unerased type.

• An **OutOfMemoryError** as a result of a boxing conversion.

• A **NullPointerException** as a result of an unboxing conversion on a null reference.

• An **ArrayStoreException** in special cases involving array elements or field access (§10.5, §15.26.1).

**Example 5.2-1. Assignment for Primitive Types**

class Test {
    public static void main(String[] args) {
        short s = 12;       // narrow 12 to short
        float f = s;        // widen short to float
        System.out.print("f=");
        char c = '\u0123';
        long l = c;         // widen char to long
        System.out.print("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f;       // widen float to double
        System.out.print("d=" + d);
    }
}

This program produces the output:

```
f=12.0
l=0x123
d=1.2300000190734863
```

The following program, however, produces compile-time errors:

class Test {
    public static void main(String[] args) {
        short s = 123;
        char c = s;       // error: would require cast
        s = c;            // error: would require cast
    }
}

because not all **short** values are **char** values, and neither are all **char** values **short** values.
Example 5.2-2. Assignment for Reference Types

class Point { int x, y; }
class Point3D extends Point { int z; }
interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D();
        // OK because Point3D is a subclass of Point
        Point3D p3d = p;
        // Error: will require a cast because a Point
        // might not be a Point3D (even though it is,
        // dynamically, in this example.)

        // Assignments to variables of type Object:
        Object o = p;          // OK: any object to Object
        int[] a = new int[3];
        Object o2 = a;          // OK: an array to Object

        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;
        // OK: ColoredPoint implements Colorable

        // Assignments to variables of array type:
        byte[] b = new byte[4];
        a = b;
        // Error: these are not arrays of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;
        // OK: since we can assign a Point3D to a Point
        p3da = pa;
        // Error: (cast needed) since a Point
        // can't be assigned to a Point3D
    }
}

The following test program illustrates assignment conversions on reference values, but fails
to compile, as described in its comments. This example should be compared to the preceding
one.

class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
int color;
public void setColor(int color) { this.color = color; }
}

class Test {
public static void main(String[] args) {
    Point p = new Point();
    ColoredPoint cp = new ColoredPoint();
    // Okay because ColoredPoint is a subclass of Point:
    p = cp;
    // Okay because ColoredPoint implements Colorable:
    Colorable c = cp;
    // The following cause compile-time errors because
    // we cannot be sure they will succeed, depending on
    // the run-time type of p; a run-time check will be
    // necessary for the needed narrowing conversion and
    // must be indicated by including a cast:
    cp = p;    // p might be neither a ColoredPoint
               // nor a subclass of ColoredPoint
    c = p;     // p might not implement Colorable
}
}

Example 5.2-3. Assignment for Array Types

class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
public static void main(String[] args) {
    long[] veclong = new long[100];
    Object o = veclong;  // okay
    Long l = veclong;    // compile-time error
    short[] vecshort = veclong; // compile-time error
    Point[] pvec = new Point[100];
    ColoredPoint[] cpvec = new ColoredPoint[100];
    pvec = cpvec;        // okay
    pvec[0] = new Point();  // okay at compile time,
                           // but would throw an
                           // exception at run time
    cpvec = pvec;        // compile-time error
}
}

In this example:

- The value of veclong cannot be assigned to a Long variable, because Long is a class type other than Object. An array can be assigned only to a variable of a compatible array type, or to a variable of type Object, Cloneable or java.io.Serializable.
- The value of veclong cannot be assigned to vecshort, because they are arrays of primitive type, and short and long are not the same primitive type.
• The value of \texttt{cpvec} can be assigned to \texttt{pvec}, because any reference that could be the value of an expression of type \texttt{ColoredPoint} can be the value of a variable of type \texttt{Point}. The subsequent assignment of the new \texttt{Point} to a component of \texttt{pvec} then would throw an \texttt{ArrayStoreException} (if the program were otherwise corrected so that it could be compiled), because a \texttt{ColoredPoint} array cannot have an instance of \texttt{Point} as the value of a component.

• The value of \texttt{pvec} cannot be assigned to \texttt{cpvec}, because not every reference that could be the value of an expression of type \texttt{Point} can correctly be the value of a variable of type \texttt{ColoredPoint}. If the value of \texttt{pvec} at run time were a reference to an instance of \texttt{Point[]}, and the assignment to \texttt{cpvec} were allowed, a simple reference to a component of \texttt{cpvec}, say, \texttt{cpvec[0]}, could return a \texttt{Point}, and a \texttt{Point} is not a \texttt{ColoredPoint}. Thus to allow such an assignment would allow a violation of the type system. A cast may be used (§5.5, §15.16) to ensure that \texttt{pvec} references a \texttt{ColoredPoint[]}:

\begin{verbatim}
    cpvec = (ColoredPoint[])pvec;  // OK, but may throw an
    // exception at run time
\end{verbatim}

### 5.3 Invocation Contexts

Invocation contexts allow an argument value in a method or constructor invocation (§8.8.7.1, §15.9, §15.12) to be assigned to a corresponding formal parameter.

**Strict** invocation contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)

**Loose** invocation contexts allow a more permissive set of conversions, because they are only used for a particular invocation if no applicable declaration can be found using strict invocation contexts. Loose invocation contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a widening reference conversion followed by an unboxing conversion
- a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
- a boxing conversion (§5.1.7)
- a boxing conversion followed by widening reference conversion
• an unboxing conversion (§5.1.8)
• an unboxing conversion followed by a widening primitive conversion

If, after the conversions listed for an invocation context have been applied, the resulting type is a raw type (§4.8), an unchecked conversion (§5.1.9) may then be applied.

A value of the null type (the null reference is the only such value) may be assigned to any reference type.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation (§4.10).

If the type of an argument expression is either \texttt{float} or \texttt{double}, then value set conversion (§5.1.13) is applied after the conversion(s):

• If an argument value of type \texttt{float} is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.

• If an argument value of type \texttt{double} is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

The only exceptions that may arise in an invocation context are:

• A \texttt{ClassCastException} if, after the type conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure (§4.6) of the corresponding formal parameter type.

• An \texttt{OutOfMemoryError} as a result of a boxing conversion.

• A \texttt{NullPointerException} as a result of an unboxing conversion on a null reference.

Neither strict nor loose invocation contexts include the implicit narrowing of integer constant expressions which is allowed in assignment contexts. The designers of the Java programming language felt that including these implicit narrowing conversions would add additional complexity to the rules of overload resolution (§15.12.2).

Thus, the program:

```java
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String[] args) {
        System.out.println(m(12, 2));  // compile-time error
```
causes a compile-time error because the integer literals 12 and 2 have type \texttt{int}, so neither method \texttt{m} matches under the rules of overload resolution. A language that included implicit narrowing of integer constant expressions would need additional rules to resolve cases like this example.

5.4 String Contexts

String contexts apply only to an operand of the binary $+$ operator which is not a \texttt{String} when the other operand is a \texttt{String}.

The target type in these contexts is always \texttt{String}, and a string conversion ($\S 5.1.11$) of the non-\texttt{String} operand always occurs. Evaluation of the $+$ operator then proceeds as specified in $\S 15.18.1$.

5.5 Casting Contexts

Casting contexts allow the operand of a cast expression ($\S 15.16$) to be converted to the type explicitly named by the cast operator. Compared to assignment contexts and invocation contexts, casting contexts allow the use of more of the conversions defined in $\S 5.1$, and allow more combinations of those conversions.

If the expression is of a primitive type, then a casting context allows the use of one of the following:

• an identity conversion ($\S 5.1.1$)
• a widening primitive conversion ($\S 5.1.2$)
• a narrowing primitive conversion ($\S 5.1.3$)
• a widening and narrowing primitive conversion ($\S 5.1.4$)
• a boxing conversion ($\S 5.1.7$)
• a boxing conversion followed by a widening reference conversion ($\S 5.1.5$)

If the expression is of a reference type, then a casting context allows the use of one of the following:

• an identity conversion ($\S 5.1.1$)
• a widening reference conversion ($\S 5.1.5$)
• a widening reference conversion followed by an unboxing conversion
• a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
• a narrowing reference conversion (§5.1.6)
• a narrowing reference conversion followed by an unboxing conversion
• an unboxing conversion (§5.1.8)
• an unboxing conversion followed by a widening primitive conversion

If the expression has the null type, then the expression may be cast to any reference type.

If a casting context makes use of a narrowing reference conversion that is checked or partially unchecked (§5.1.6.2, §5.1.6.3), then a run time check will be performed on the class of the expression's value, possibly causing a `ClassCastException`. Otherwise, no run time check is performed.

Value set conversion (§5.1.13) is applied after the type conversion.

The following tables enumerate which conversions are used in certain casting contexts. Each conversion is signified by a symbol:

• - signifies no conversion allowed
• ≈ signifies identity conversion (§5.1.1)
• ω signifies widening primitive conversion (§5.1.2)
• η signifies narrowing primitive conversion (§5.1.3)
• ωη signifies widening and narrowing primitive conversion (§5.1.4)
• ↑ signifies widening reference conversion (§5.1.5)
• ↓ signifies narrowing reference conversion (§5.1.6)
• ⊕ signifies boxing conversion (§5.1.7)
• ⊗ signifies unboxing conversion (§5.1.8)

In the tables, a comma between symbols indicates that a casting context uses one conversion followed by another. The type `Object` means any reference type other than the eight wrapper classes `Boolean, Byte, Short, Character, Integer, Long, Float, Double`. 
### Table 5.5-A. Casting to primitive types

<table>
<thead>
<tr>
<th>To</th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>boolean</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>From↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>≈</td>
<td>ω</td>
<td>ωη</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>short</td>
<td>η</td>
<td>≈</td>
<td>η</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>char</td>
<td>η</td>
<td>η</td>
<td>≈</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>int</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>≈</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>long</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>≈</td>
<td>ω</td>
<td>ω</td>
<td>ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>float</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>≈</td>
<td>ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>double</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>η</td>
<td>≈</td>
<td>▼,ω</td>
<td>▼,ω</td>
</tr>
<tr>
<td>boolean</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>▼,ω</td>
</tr>
<tr>
<td>Byte</td>
<td>⊗</td>
<td>⊗,ω</td>
<td>-</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Short</td>
<td>-</td>
<td>⊗</td>
<td>-</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Character</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Integer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Long</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Float</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
<td>⊗,ω</td>
</tr>
<tr>
<td>Double</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Boolean</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>⊗</td>
</tr>
<tr>
<td>Object</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
<td>▼,ω</td>
</tr>
</tbody>
</table>
### Table 5.5-B. Casting to reference types

<table>
<thead>
<tr>
<th>To</th>
<th>Byte</th>
<th>Short</th>
<th>Character</th>
<th>Integer</th>
<th>Long</th>
<th>Float</th>
<th>Double</th>
<th>Boolean</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>From↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>short</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>char</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>int</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>long</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>float</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>double</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>boolean</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
<td>⫶</td>
</tr>
</tbody>
</table>

- ⫶: Convertible
- ⫶,↑: Convertible with implicit type conversion
- ⫶,↓: Convertible with explicit type conversion

<table>
<thead>
<tr>
<th>Type</th>
<th>Boolean</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>⫶</td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Character</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Integer</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Long</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Float</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Double</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Boolean</td>
<td>⫶</td>
<td>⫶</td>
</tr>
<tr>
<td>Object</td>
<td>⫶</td>
<td>⫶</td>
</tr>
</tbody>
</table>

- ⫶: Cannot cast to reference type
- ↑: Requires explicit type conversion

**Example 5.5-1. Casting for Reference Types**

```java
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point {}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p; // p might not reference an
        // object which is a ColoredPoint
        // or a subclass of ColoredPoint
        c = (Colorable)p; // p might not be Colorable
        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;    // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;    // compile-time error #2
    }
}
```

Here, the first compile-time error occurs because the class types `Long` and `Point` are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type `EndPoint` can never reference a value that implements the interface `Colorable`. This is because `EndPoint` is a `final` type, and a variable of a `final` type always holds a value of the same run-time type as its compile-time type. Therefore, the run-time type of variable `e` must be exactly the type `EndPoint`, and type `EndPoint` does not implement `Colorable`. 
Example 5.5-2. Casting for Array Types

class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}

interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}

class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: { ");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }\n")
    }
}

This program compiles without errors and produces the output:

    cpa: { (2,2)@12, (4,5)@24, null, null }
Example 5.5-3. Casting Incompatible Types at Run Time

```java
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];
        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;
        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
        c setColor(0);
    }
}
```

This program uses casts to compile, but it throws exceptions at run time, because the types are incompatible.

5.6 Numeric Contexts

Numeric contexts apply to the operands of an arithmetic operator. Numeric contexts allow the use of:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5) followed by an unboxing conversion
- a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
- an unboxing conversion (§5.1.8)
- an unboxing conversion followed by a widening primitive conversion

A numeric promotion is a process by which, given an arithmetic operator and its argument expressions, the arguments are converted to an inferred target type $T$. $T$
is chosen during promotion such that each argument expression can be converted to $T$ and the arithmetic operation is defined for values of type $T$.

The two kinds of numeric promotion are unary numeric promotion (§5.6.1) and binary numeric promotion (§5.6.2).

### 5.6.1 Unary Numeric Promotion

Some operators apply *[unary numeric promotion]* to a single operand, which must produce a value of a numeric type:

- If the operand is of compile-time type `Byte`, `Short`, `Character`, or `Integer`, it is subjected to unboxing conversion (§5.1.8). The result is then promoted to a value of type `int` by a widening primitive conversion (§5.1.2) or an identity conversion (§5.1.1).
- Otherwise, if the operand is of compile-time type `Long`, `Float`, or `Double`, it is subjected to unboxing conversion (§5.1.8).
- Otherwise, if the operand is of compile-time type `byte`, `short`, or `char`, it is promoted to a value of type `int` by a widening primitive conversion (§5.1.2).
- Otherwise, a unary numeric operand remains as is and is not converted.

After the conversion(s), if any, value set conversion (§5.1.13) is then applied.

Unary numeric promotion is performed on expressions in the following situations:

- Each dimension expression in an array creation expression (§15.10.1)
- The index expression in an array access expression (§15.10.3)
- The operand of a unary plus operator `+` (§15.15.3)
- The operand of a unary minus operator `-` (§15.15.4)
- The operand of a bitwise complement operator `~` (§15.15.5)
- Each operand, separately, of a shift operator `<<`, `>>`, or `>>>` (§15.19).

A `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long`.

**Example 5.6.1-1. Unary Numeric Promotion**

```java
class Test {
    public static void main(String[] args) {
        byte b = 2;
        int a[] = new int[b];  // dimension expression promotion
        char c = '\u0001';
    }
}
```
a[c] = 1;              // index expression promotion
a[0] = -c;             // unary - promotion
System.out.println("a: " + a[0] + "," + a[1])

b = -1;
int i = ~b;            // bitwise complement promotion
System.out.println("~0x" + Integer.toHexString(b)
+ "==0x" + Integer.toHexString(i));
i = b << 4L;           // shift promotion (left operand)
System.out.println("0x" + Integer.toHexString(b)
+ "<<4L==0x" + Integer.toHexString(i));
}
}

This program produces the output:

a: -1,1
~0xffffffff==0x0
0xffffffff<<4L==0xffffff0

5.6.2 Binary Numeric Promotion

When an operator applies binary numeric promotion to a pair of operands, each of which must denote a value that is convertible to a numeric type, the following rules apply, in order:

1. If any operand is of a reference type, it is subjected to unboxing conversion (§5.1.8).

2. Widening primitive conversion (§5.1.2) is applied to convert either or both operands as specified by the following rules:
   - If either operand is of type double, the other is converted to double.
   - Otherwise, if either operand is of type float, the other is converted to float.
   - Otherwise, if either operand is of type long, the other is converted to long.
   - Otherwise, both operands are converted to type int.

After the conversion(s), if any, value set conversion (§5.1.13) is then applied to each operand.

Binary numeric promotion is performed on the operands of certain operators:
- The multiplicative operators *, /, and % (§15.17)
- The addition and subtraction operators for numeric types + and - (§15.18.2)
- The numerical comparison operators <, <=, >, and >= (§15.20.1)
- The numerical equality operators == and != (§15.21.1)
• The integer bitwise operators &, ^, and | (§15.22.1)
• In certain cases, the conditional operator ? : (§15.25)

Example 5.6.2-1. Binary Numeric Promotion

```java
class Test {
    public static void main(String[] args) {
        int i = 0;
        float f = 1.0f;
        double d = 2.0;
        // First int*float is promoted to float*float, then
        // float==double is promoted to double==double:
        if (i * f == d) System.out.println("oops");

        // A char&byte is promoted to int&int:
        byte b = 0x1f;
        char c = 'G';
        int control = c & b;
        System.out.println(Integer.toHexString(control));

        // Here int:float is promoted to float:float:
        f = (b==0) ? i : 4.0f;
        System.out.println(1.0/f);
    }
}
```

This program produces the output:

```
7
0.25
```

The example converts the ASCII character 'G' to the ASCII control-G (BEL), by masking off all but the low 5 bits of the character. The 7 is the numeric value of this control character.
Names are used to refer to entities declared in a program.

A declared entity (§6.1) is a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable.

Names in programs are either *simple*, consisting of a single identifier, or *qualified*, consisting of a sequence of identifiers separated by "." tokens (§6.2).

Every declaration that introduces a name has a *scope* (§6.3), which is the part of the program text within which the declared entity can be referred to by a simple name.

A qualified name \( N.x \) may be used to refer to a *member* of a package or reference type, where \( N \) is a simple or qualified name and \( x \) is an identifier. If \( N \) names a package, then \( x \) is a member of that package, which is either a class or interface type or a subpackage. If \( N \) names a reference type or a variable of a reference type, then \( x \) names a member of that type, which is either a class, an interface, a field, or a method.

In determining the meaning of a name (§6.5), the context of the occurrence is used to disambiguate among packages, types, variables, and methods with the same name.

Access control (§6.6) can be specified in a class, interface, method, or field declaration to control when *access* to a member is allowed. Access is a different concept from scope. Access specifies the part of the program text within which the declared entity can be referred to by a qualified name. Access to a declared entity is also relevant in a field access expression (§15.11), a method invocation expression in which the method is not specified by a simple name (§15.12), a method reference expression (§15.13), or a qualified class instance creation expression (§15.9). In the absence of an access modifier, most declarations have package access, allowing
access anywhere within the package that contains its declaration; other possibilities are public, protected, and private.

Fully qualified and canonical names (§6.7) are also discussed in this chapter.

6.1 Declarations

A declaration introduces an entity into a program and includes an identifier (§3.8) that can be used in a name to refer to this entity. The identifier is constrained to be a type identifier when the entity being introduced is a class, interface, or type parameter.

A declared entity is one of the following:

- A module, declared in a module declaration (§7.7)
- A package, declared in a package declaration (§7.4)
- An imported type, declared in a single-type-import declaration or a type-import-on-demand declaration (§7.5.1, §7.5.2)
- An imported static member, declared in a single-static-import declaration or a static-import-on-demand declaration (§7.5.3, §7.5.4)
- A class, declared in a class type declaration (§8.1)
- An interface, declared in an interface type declaration (§9.1)
- A type parameter, declared as part of the declaration of a generic class, interface, method, or constructor (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
- A member of a reference type (§8.2, §9.2, §8.9.3, §9.6, §10.7), one of the following:
  - A member class (§8.5, §9.5)
  - A member interface (§8.5, §9.5)
  - An enum constant (§8.9)
  - A field, one of the following:
    - A field declared in a class type or enum type (§8.3, §8.9.2)
    - A field declared in an interface type or annotation type (§9.3, §9.6.1)
    - The field length, which is implicitly a member of every array type (§10.7)
  - A method, one of the following:
A method (abstract or otherwise) declared in a class type or enum type (§8.4, §8.9.2)

A method (abstract or otherwise) declared in an interface type or annotation type (§9.4, §9.6.1)

• A parameter, one of the following:
  – A formal parameter of a method or constructor of a class type or enum type (§8.4.1, §8.8.1, §8.9.2), or of a lambda expression (§15.27.1)
  – A formal parameter of an abstract method of an interface type or annotation type (§9.4, §9.6.1)
  – An exception parameter of an exception handler declared in a catch clause of a try statement (§14.20)

• A local variable, one of the following:
  – A local variable declared in a block (§14.4)
  – A local variable declared in a for statement (§14.14)

Constructors (§8.8) are also introduced by declarations, but use the name of the class in which they are declared rather than introducing a new name.

The declaration of a type which is not generic (class C ...) declares one entity: a non-generic type (C). A non-generic type is not a raw type, despite the syntactic similarity. In contrast, the declaration of a generic type (class C<T> ... or interface C<T> ...) declares two entities: a generic type (C<T>) and a corresponding non-generic type (C). In this case, the meaning of the term C depends on the context where it appears:

• If genericity is unimportant, as in the non-generic contexts identified below, the identifier C denotes the non-generic type C.

• If genericity is important, as in all contexts from §6.5 except the non-generic contexts, the identifier C denotes either:
  – The raw type C which is the erasure (§4.6) of the generic type C<T>; or
  – A parameterized type which is a particular parameterization (§4.5) of the generic type C<T>.

The 14 non-generic contexts are as follows:

1. In a uses or provides directive in a module declaration (§7.7.1)
2. In a single-type-import declaration (§7.5.1)
3. To the left of the . in a single-static-import declaration (§7.5.3)
4. To the left of the . in a static-import-on-demand declaration (§7.5.4)
5. To the left of the ( in a constructor declaration (§8.8)
6. After the @ sign in an annotation (§9.7)
7. To the left of .class in a class literal (§15.8.2)
8. To the left of .this in a qualified this expression (§15.8.4)
9. To the left of .super in a qualified superclass field access expression (§15.11.2)
10. To the left of .Identifier or .super.Identifier in a qualified method invocation expression (§15.12)
11. To the left of .super:: in a method reference expression (§15.13)
12. In a qualified expression name in a postfix expression or a try-with-resources statement (§15.14.1, §14.20.3)
13. In a throws clause of a method or constructor (§8.4.6, §8.8.5, §9.4)
14. In an exception parameter declaration (§14.20)

The first eleven non-generic contexts correspond to the first eleven syntactic contexts for a TypeName in §6.5.1. The twelfth non-generic context is where a qualified ExpressionName such as c.x may include a TypeName c to denote static member access. The common use of TypeName in these twelve contexts is significant: it indicates that these contexts involve a less-than-first-class use of a type. In contrast, the thirteenth and fourteenth non-generic contexts employ ClassType, indicating that throws and catch clauses use types in a first-class way, in line with, say, field declarations. The characterization of these two contexts as non-generic is due to the fact that an exception type cannot be parameterized.

Note that the ClassType production allows annotations, so it is possible to annotate the use of a type in a throws or catch clause, whereas the TypeName production disallows annotations, so it is not possible to annotate the name of a type in, say, a single-type-import declaration.

Naming Conventions

The class libraries of the Java SE Platform attempt to use, whenever possible, names chosen according to the conventions presented below. These conventions help to make code more readable and avoid certain kinds of name conflicts.

We recommend these conventions for use in all programs written in the Java programming language. However, these conventions should not be followed slavishly if long-held
conventional usage dictates otherwise. So, for example, the \texttt{sin} and \texttt{cos} methods of the class \texttt{java.lang.Math} have mathematically conventional names, even though these method names flout the convention suggested here because they are short and are not verbs.

\textit{Package Names and Module Names}

Developers should take steps to avoid the possibility of two published packages having the same name by choosing \textit{unique package names} for packages that are widely distributed. This allows packages to be easily and automatically installed and catalogued. This section specifies a suggested convention for generating such unique package names. Implementations of the Java SE Platform are encouraged to provide automatic support for converting a set of packages from local and casual package names to the unique name format described here.

If unique package names are not used, then package name conflicts may arise far from the point of creation of either of the conflicting packages. This may create a situation that is difficult or impossible for the user or programmer to resolve. The classes \texttt{ClassLoader} and \texttt{ModuleLayer} can be used to isolate packages with the same name from each other in those cases where the packages will have constrained interactions, but not in a way that is transparent to a naïve program.

You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as \texttt{oracle.com}. You then reverse this name, component by component, to obtain, in this example, \texttt{com.oracle}, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names. Such a convention might specify that certain package name components be division, department, project, machine, or login names.

\textbf{Example 6.1-1. Unique Package Names}

\begin{verbatim}
com.nighthacks.scrabble.dictionary
org.openjdk.compiler.source.tree
net.jcip.annotations
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
\end{verbatim}

The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top level domain names, such as \texttt{com}, \texttt{edu}, \texttt{gov}, \texttt{mil}, \texttt{net}, or \texttt{org}, or one of the English two-letter codes identifying countries as specified in \textit{ISO Standard 3166}.

In some cases, the Internet domain name may not be a valid package name. Here are some suggested conventions for dealing with these situations:

\begin{itemize}
  \item If the domain name contains a hyphen, or any other special character not allowed in an identifier (§3.8), convert it into an underscore.
  \item If any of the resulting package name components are keywords (§3.9), append an underscore to them.
\end{itemize}
• If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.

The name of a module should correspond to the name of its principal exported package. If a module does not have such a package, or if for legacy reasons it must have a name that does not correspond to one of its exported packages, then its name should still start with the reversed form of an Internet domain with which its author is associated.

Example 6.1-2. Unique Module Names

com.nighthacks.scrabble
org.openjdk.compiler
net.jcip.annotations

The first component of a package or module name must not be the identifier java. Package and module names that start with the identifier java are reserved for packages and modules of the Java SE Platform.

The name of a package or module is not meant to imply where the package or module is stored on the Internet. For example, a package named edu.cmu.cs.bovik.cheese is not necessarily obtainable from the host cmu.edu or cs.cmu.edu or bovik.cs.cmu.edu. The suggested convention for generating unique package and module names is merely a way to piggyback a package and module naming convention on top of an existing, widely known unique name registry instead of having to create a separate registry for package and module names.

Class and Interface Type Names

Names of class types should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized.

Example 6.1-3. Descriptive Class Names

ClassLoader
SecurityManager
Thread
Dictionary
BufferedInputStream

Likewise, names of interface types should be short and descriptive, not overly long, in mixed case with the first letter of each word capitalized. The name may be a descriptive noun or noun phrase, which is appropriate when an interface is used as if it were an abstract superclass, such as interfaces java.io.DataInput and java.io.DataOutput; or it may be an adjective describing a behavior, as for the interfaces Runnable and Cloneable.

Type Variable Names

Type variable names should be pithy (single character if possible) yet evocative, and should not include lower case letters. This makes it easy to distinguish type parameters from ordinary classes and interfaces.
Container types should use the name \( E \) for their element type. Maps should use \( K \) for the
type of their keys and \( V \) for the type of their values. The name \( X \) should be used for arbitrary
exception types. We use \( T \) for type, whenever there is not anything more specific about the
type to distinguish it. (This is often the case in generic methods.)

If there are multiple type parameters that denote arbitrary types, one should use letters
that neighbor \( T \) in the alphabet, such as \( S \). Alternately, it is acceptable to use numeric
subscripts (e.g., \( T_1, T_2 \)) to distinguish among the different type variables. In such cases, all
the variables with the same prefix should be subscripted.

If a generic method appears inside a generic class, it is a good idea to avoid using the
same names for the type parameters of the method and class, to avoid confusion. The same
applies to nested generic classes.

**Example 6.1-4. Conventional Type Variable Names**

```java
public class HashSet<E> extends AbstractSet<E> { ... }
public class HashMap<K,V> extends AbstractMap<K,V> { ... }
public class ThreadLocal<T> { ... }
public interface Functor<T, X extends Throwable> {
    T eval() throws X;
}
```

When type parameters do not fall conveniently into one of the categories mentioned, names
should be chosen to be as meaningful as possible within the confines of a single letter. The
names mentioned above (\( E, K, V, X, T \)) should not be used for type parameters that do not
fall into the designated categories.

**Method Names**

Method names should be verbs or verb phrases, in mixed case, with the first letter lowercase
and the first letter of any subsequent words capitalized. Here are some additional specific
conventions for method names:

- Methods to get and set an attribute that might be thought of as a variable \( V \) should be
  named `getV` and `setV`. An example is the methods `getPriority` and `setPriority`
of class `Thread`.

- A method that returns the length of something should be named `length`, as in class
  `String`.

- A method that tests a boolean condition \( V \) about an object should be named `isV`. An
  example is the method `isInterrupted` of class `Thread`.

- A method that converts its object to a particular format \( F \) should be named
  `toF`. Examples are the method `toString` of class `Object` and the methods
  `toLocaleString` and `toGMTString` of class `java.util.Date`.

Whenever possible and appropriate, basing the names of methods in a new class on names
in an existing class that is similar, especially a class from the Java SE Platform API, will
make it easier to use.
### Field Names

Names of fields that are not final should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Note that well-designed classes have very few public or protected fields, except for fields that are constants (static final fields).

Fields should have names that are nouns, noun phrases, or abbreviations for nouns.

Examples of this convention are the fields buf, pos, and count of the class java.io.ByteArrayInputStream and the field bytesTransferred of the class java.io.InterruptedIOException.

### Constant Names

The names of constants in interface types should be, and final variables of class types may conventionally be, a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech.

Examples of names for constants include MIN_VALUE, MAX_VALUE, MIN_RADIX, and MAX_RADIX of the class Character.

A group of constants that represent alternative values of a set, or, less frequently, masking bits in an integer value, are sometimes usefully specified with a common acronym as a name prefix.

For example:

```java
interface ProcessStates {
    int PS_RUNNING   = 0;
    int PS_SUSPENDED = 1;
}
```

### Local Variable and Parameter Names

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words, such as:

- Acronyms, that is the first letter of a series of words, as in cp for a variable holding a reference to a ColoredPoint
- Abbreviations, as in buf holding a pointer to a buffer of some kind
- Mnemonic terms, organized in some way to aid memory and understanding, typically by using a set of local variables with conventional names patterned after the names of parameters to widely used classes. For example:
  - in and out, whenever some kind of input and output are involved, patterned after the fields of System
- off and len, whenever an offset and length are involved, patterned after the parameters to the read and write methods of the interfaces DataInput and DataOutput of java.io

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type. Conventional one-character names are:

- b for a byte
- c for a char
- d for a double
- e for an Exception
- f for a float
- i, j, and k for ints
- l for a long
- o for an Object
- s for a String
- v for an arbitrary value of some type

Local variable or parameter names that consist of only two or three lowercase letters should not conflict with the initial country codes and domain names that are the first component of unique package names.

### 6.2 Names and Identifiers

A name is used to refer to an entity declared in a program. There are two forms of names: simple names and qualified names.

A **simple name** is a single identifier.

A **qualified name** consists of a name, a "." token, and an identifier.

In determining the meaning of a name (§6.5), the context in which the name appears is taken into account. The rules of §6.5 distinguish among contexts where a name must denote (refer to) a package (§6.5.3), a type (§6.5.5), a variable or value in an expression (§6.5.6), or a method (§6.5.7).

Packages and reference types have *members* which may be accessed by qualified names. As background for the discussion of qualified names and the determination of the meaning of names, see the descriptions of membership in §4.4, §4.5.2, §4.8, §4.9, §7.1, §8.2, §9.2, and §10.7.
Not all identifiers in a program are a part of a name. Identifiers are also used in the following situations:

- In declarations (§6.1), where an identifier may occur to specify the name by which the declared entity will be known.

- As labels in labeled statements (§14.7) and in break and continue statements (§14.15, §14.16) that refer to statement labels.

The identifiers used in labeled statements and their associated break and continue statements are completely separate from those used in declarations.

- In field access expressions (§15.11), where an identifier occurs after a "." token to indicate a member of the object denoted by the expression before the "." token, or the object denoted by the `super` or `TypeName.super` before the "." token.

- In some method invocation expressions (§15.12), wherever an identifier occurs after a "." token and before a "(" token to indicate a method to be invoked for the object denoted by the expression before the "." token, or the type denoted by the `TypeName` before the "." token, or the object denoted by the `super` or `TypeName.super` before the "." token.

- In some method reference expressions (§15.13), wherever an identifier occurs after a "::" token to indicate a method of the object denoted by the expression before the "::" token, or the type denoted by the `TypeName` before the "::" token, or the object denoted by the `super` or `TypeName.super` before the "::" token.

- In qualified class instance creation expressions (§15.9), where an identifier occurs to the right of the `new` token to indicate a type that is a member of the compile-time type of the expression preceding the `new` token.

- In element-value pairs of annotations (§9.7.1), to denote an element of the corresponding annotation type.

In this program:

```java
class Test {
    public static void main(String[] args) {
        Class c = System.out.getClass();
        System.out.println(c.toString().length() +
                           args[0].length() + args.length);
    }
}
```

the identifiers `Test`, `main`, and the first occurrences of `args` and `c` are not names. Rather, they are identifiers used in declarations to specify the names of the declared entities. The
names String, Class, System.out.getClass, System.out.println, c.toString, args, and args.length appear in the example.

The occurrence of length in args.length is a name because args.length is a qualified name (§6.5.6.2) and not a field access expression (§15.11). A field access expression, as well as a method invocation expression, a method reference expression, and a qualified class instance creation expression, uses an identifier rather than a name to denote the member of interest. Thus, the occurrence of length in args[0].length() is not a name, but rather an identifier appearing in a method invocation expression.

One might wonder why these kinds of expression use an identifier rather than a simple name, which is after all just an identifier. The reason is that a simple expression name is defined in terms of the lexical environment; that is, a simple expression name must be in the scope of a variable declaration (§6.5.6.1). On the other hand, field access, qualified method invocation, method references, and qualified class instance creation all refer to members whose names are not in the lexical environment. By definition, such names are bound only in the context provided by the Primary of the field access expression, method invocation expression, method reference expression, or class instance creation expression; or by the super of the field access expression, method invocation expression, or method reference expression; and so on. Thus, we denote such members with identifiers rather than simple names.

To complicate things further, a field access expression is not the only way to denote a field of an object. For parsing reasons, a qualified name is used to denote a field of an in-scope variable. (The variable itself is denoted with a simple name, alluded to above.) It is necessary for access control (§6.6) to apply to both denotations of a field.

### 6.3 Scope of a Declaration

The scope of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name, provided it is not shadowed (§6.4.1).

A declaration is said to be in scope at a particular point in a program if and only if the declaration's scope includes that point.

The scope of the declaration of an observable top level package (§7.4.3) is all observable compilation units associated with modules to which the package is uniquely visible (§7.4.3).

The declaration of a package that is not observable is never in scope.

The declaration of a subpackage is never in scope.

The package java is always in scope.

The scope of a type imported by a single-type-import declaration (§7.5.1) or a type-import-on-demand declaration (§7.5.2) is all the class and interface type
declarations (§7.6) in the compilation unit in which the import declaration appears, as well as any annotations on the module declaration or package declaration of the compilation unit.

The scope of a member imported by a single-static-import declaration (§7.5.3) or a static-import-on-demand declaration (§7.5.4) is all the class and interface type declarations in the compilation unit in which the import declaration appears, as well as any annotations on the module declaration or package declaration of the compilation unit.

The scope of a top level type (§7.6) is all type declarations in the package in which the top level type is declared.

The scope of a declaration of a member \( m \) declared in or inherited by a class type \( C \) (§8.1.6) is the entire body of \( C \), including any nested type declarations.

The scope of a declaration of a member \( m \) declared in or inherited by an interface type \( I \) (§9.1.4) is the entire body of \( I \), including any nested type declarations.

The scope of an enum constant \( c \) declared in an enum type \( T \) is the body of \( T \), and any case label of a switch statement whose expression is of enum type \( T \) (§14.11).

The scope of a formal parameter of a method (§8.4.1), constructor (§8.8.1), or lambda expression (§15.27) is the entire body of the method, constructor, or lambda expression.

The scope of a class's type parameter (§8.1.2) is the type parameter section of the class declaration, the type parameter section of any superclass or superinterface of the class declaration, and the class body.

The scope of an interface's type parameter (§9.1.2) is the type parameter section of the interface declaration, the type parameter section of any superinterface of the interface declaration, and the interface body.

The scope of a method's type parameter (§8.4.4) is the entire declaration of the method, including the type parameter section, but excluding the method modifiers.

The scope of a constructor's type parameter (§8.8.4) is the entire declaration of the constructor, including the type parameter section, but excluding the constructor modifiers.

The scope of a local class declaration immediately enclosed by a block (§14.2) is the rest of the immediately enclosing block, including its own class declaration.

The scope of a local class declaration immediately enclosed by a switch block statement group (§14.11) is the rest of the immediately enclosing switch block statement group, including its own class declaration.
The scope of a local variable declaration in a block (§14.4) is the rest of the block
in which the declaration appears, starting with its own initializer and including any
further declarators to the right in the local variable declaration statement.

The scope of a local variable declared in the ForInit part of a basic for statement
(§14.14.1) includes all of the following:

- Its own initializer
- Any further declarators to the right in the ForInit part of the for statement
- The Expression and ForUpdate parts of the for statement
- The contained Statement

The scope of a local variable declared in the FormalParameter part of an enhanced
for statement (§14.14.2) is the contained Statement.

The scope of a parameter of an exception handler that is declared in a catch clause
of a try statement (§14.20) is the entire block associated with the catch.

The scope of a variable declared in the ResourceSpecification of a try-with-
resources statement (§14.20.3) is from the declaration rightward over the remainder
of the ResourceSpecification and the entire try block associated with the try-with-
resources statement.

The translation of a try-with-resources statement implies the rule above.

**Example 6.3-1. Scope of Type Declarations**

These rules imply that declarations of class and interface types need not appear before uses
of the types. In the following program, the use of PointList in class Point is valid,
because the scope of the class declaration PointList includes both class Point and class
PointList, as well as any other type declarations in other compilation units of package
points.

```java
package points;
class Point {
    int x, y;
    PointList list;
    Point next;
}

class PointList {
    Point first;
}
```
Example 6.3-2. Scope of Local Variable Declarations

The following program causes a compile-time error because the initialization of local variable x is within the scope of the declaration of local variable x, but the local variable x does not yet have a value and cannot be used. The field x has a value of 0 (assigned when Test1 was initialized) but is a red herring since it is shadowed (§6.4.1) by the local variable x.

```java
class Test1 {
    static int x;
    public static void main(String[] args) {
        int x = x;
    }
}
```

The following program does compile:

```java
class Test2 {
    static int x;
    public static void main(String[] args) {
        int x = (x=2)*2;
        System.out.println(x);
    }
}
```

because the local variable x is definitely assigned (§16 (Definite Assignment)) before it is used. It prints:

```
4
```

In the following program, the initializer for three can correctly refer to the variable two declared in an earlier declarator, and the method invocation in the next line can correctly refer to the variable three declared earlier in the block.

```java
class Test3 {
    public static void main(String[] args) {
        System.out.print("2+1=");
        int two = 2, three = two + 1;
        System.out.println(three);
    }
}
```

This program produces the output:

```
2+1=3
```
6.4 Shadowing and Obscuring

A local variable (§14.4), formal parameter (§8.4.1, §15.27.1), exception parameter (§14.20), and local class (§14.3) can only be referred to using a simple name, not a qualified name (§6.2).

Some declarations are not permitted within the scope of a local variable, formal parameter, exception parameter, or local class declaration because it would be impossible to distinguish between the declared entities using only simple names.

For example, if the name of a formal parameter of a method could be redeclared as the name of a local variable in the method body, then the local variable would shadow the formal parameter and there would be no way to refer to the formal parameter - an undesirable outcome.

It is a compile-time error if the name of a formal parameter is used to declare a new variable within the body of the method, constructor, or lambda expression, unless the new variable is declared within a class declaration contained by the method, constructor, or lambda expression.

It is a compile-time error if the name of a local variable \( v \) is used to declare a new variable within the scope of \( v \), unless the new variable is declared within a class whose declaration is within the scope of \( v \).

It is a compile-time error if the name of an exception parameter is used to declare a new variable within the Block of the catch clause, unless the new variable is declared within a class declaration contained by the Block of the catch clause.

It is a compile-time error if the name of a local class \( c \) is used to declare a new local class within the scope of \( c \), unless the new local class is declared within another class whose declaration is within the scope of \( c \).

These rules allow redeclaration of a variable or local class in nested class declarations that occur in the scope of the variable or local class; such nested class declarations may be local classes (§14.3) or anonymous classes (§15.9). Thus, the declaration of a formal parameter, local variable, or local class may be shadowed in a class declaration nested within a method, constructor, or lambda expression; and the declaration of an exception parameter may be shadowed in a class declaration nested within the Block of the catch clause.

There are two design alternatives for handling name clashes created by lambda parameters and other variables declared in lambda expressions. One is to mimic class declarations: like local classes, lambda expressions introduce a new "level" for names, and all variable names outside the expression can be redeclared. Another is a "local" strategy: like catch clauses, for loops, and blocks, lambda expressions operate at the same "level" as the enclosing context, and local variables outside the expression cannot be shadowed. The above rules use the local strategy; there is no special dispensation that allows a variable declared in a lambda expression to shadow a variable declared in an enclosing method.
Note that the rule for local classes does not make an exception for a class of the same name declared within the local class itself. However, this case is prohibited by a separate rule: a class cannot have the same name as a class that encloses it (§8.1).

**Example 6.4-1. Attempted Shadowing Of A Local Variable**

Because a declaration of an identifier as a local variable of a method, constructor, or initializer block must not appear within the scope of a parameter or local variable of the same name, a compile-time error occurs for the following program:

```java
class Test1 {
    public static void main(String[] args) {
        int i;
        for (int i = 0; i < 10; i++)
            System.out.println(i);
    }
}
```

This restriction helps to detect some otherwise very obscure bugs. A similar restriction on shadowing of members by local variables was judged impractical, because the addition of a member in a superclass could cause subclasses to have to rename local variables. Related considerations make restrictions on shadowing of local variables by members of nested classes, or on shadowing of local variables by local variables declared within nested classes unattractive as well.

Hence, the following program compiles without error:

```java
class Test2 {
    public static void main(String[] args) {
        int i;
        class Local {
            {
                for (int i = 0; i < 10; i++)
                    System.out.println(i);
            }
        }
        new Local();
    }
}
```

On the other hand, local variables with the same name may be declared in two separate blocks or for statements, neither of which contains the other:

```java
class Test3 {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
        for (int i = 10; i > 0; i--)
            System.out.print(i + " ");
        System.out.println();
    }
}
```
This program compiles without error and, when executed, produces the output:

```
0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1
```

### 6.4.1 Shadowing

Some declarations may be *shadowed* in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity.

Shadowing is distinct from hiding (§8.3, §8.4.2, §8.5, §9.3, §9.5), which applies only to members which would otherwise be inherited but are not because of a declaration in a subclass. Shadowing is also distinct from obscuring (§6.4.2).

A declaration $d$ of a type named $n$ shadows the declarations of any other types named $n$ that are in scope at the point where $d$ occurs throughout the scope of $d$.

A declaration $d$ of a field or formal parameter named $n$ shadows, throughout the scope of $d$, the declarations of any other variables named $n$ that are in scope at the point where $d$ occurs.

A declaration $d$ of a local variable or exception parameter named $n$ shadows, throughout the scope of $d$, (a) the declarations of any other fields named $n$ that are in scope at the point where $d$ occurs, and (b) the declarations of any other variables named $n$ that are in scope at the point where $d$ occurs but are not declared in the innermost class in which $d$ is declared.

A declaration $d$ of a method named $n$ shadows the declarations of any other methods named $n$ that are in an enclosing scope at the point where $d$ occurs throughout the scope of $d$.

A package declaration never shadows any other declaration.

A type-import-on-demand declaration never causes any other declaration to be shadowed.

A static-import-on-demand declaration never causes any other declaration to be shadowed.

A single-type-import declaration $d$ in a compilation unit $c$ of package $p$ that imports a type named $n$ shadows, throughout $c$, the declarations of:

- any top level type named $n$ declared in another compilation unit of $p$
- any type named $n$ imported by a type-import-on-demand declaration in $c$
• any type named \( n \) imported by a static-import-on-demand declaration in \( c \)

A single-static-import declaration \( d \) in a compilation unit \( c \) of package \( p \) that imports a field named \( n \) shadows the declaration of any static field named \( n \) imported by a static-import-on-demand declaration in \( c \), throughout \( c \).

A single-static-import declaration \( d \) in a compilation unit \( c \) of package \( p \) that imports a method named \( n \) with signature \( s \) shadows the declaration of any static method named \( n \) with signature \( s \) imported by a static-import-on-demand declaration in \( c \), throughout \( c \).

A single-static-import declaration \( d \) in a compilation unit \( c \) of package \( p \) that imports a type named \( n \) shadows, throughout \( c \), the declarations of:

• any static type named \( n \) imported by a static-import-on-demand declaration in \( c \);

• any top level type (§7.6) named \( n \) declared in another compilation unit (§7.3) of \( p \);

• any type named \( n \) imported by a type-import-on-demand declaration (§7.5.2) in \( c \).

Example 6.4.1-1. Shadowing of a Field Declaration by a Local Variable Declaration

```java
class Test {
    static int x = 1;
    public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println("", Test.x=" + Test.x);
    }
}
```

This program produces the output:

```
x=0, Test.x=1
```

This program declares:

• a class `Test`

• a class (static) variable \( x \) that is a member of the class `Test`

• a class method `main` that is a member of the class `Test`

• a parameter `args` of the `main` method

• a local variable \( x \) of the `main` method

Since the scope of a class variable includes the entire body of the class (§8.2), the class variable \( x \) would normally be available throughout the entire body of the method `main`. 
In this example, however, the class variable $x$ is shadowed within the body of the method 
`main` by the declaration of the local variable $x$.

A local variable has as its scope the rest of the block in which it is declared (§6.3); in 
this case this is the rest of the body of the `main` method, namely its initializer "0" and the 
invocations of `System.out.print` and `System.out.println`.

This means that:

- The expression $x$ in the invocation of `print` refers to (denotes) the value of the local 
variable $x$.

- The invocation of `println` uses a qualified name (§6.6) `Test.x`, which uses the class 
type name `Test` to access the class variable $x$, because the declaration of `Test.x` is 
shadowed at this point and cannot be referred to by its simple name.

The keyword `this` can also be used to access a shadowed field $x$, using the form `this.x`. 
Indeed, this idiom typically appears in constructors (§8.8):

```java
class Pair {
    Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
}
```

Here, the constructor takes parameters having the same names as the fields to be initialized. 
This is simpler than having to invent different names for the parameters and is not too 
confusing in this stylized context. In general, however, it is considered poor style to have 
local variables with the same names as fields.

**Example 6.4.1-2. Shadowing of a Type Declaration by Another Type Declaration**

```java
import java.util.*;
class Vector {
    int val[] = { 1, 2 };
}
class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

The program compiles and prints:

```
1
```
using the class `Vector` declared here in preference to the generic class `java.util.Vector` (§8.1.2) that might be imported on demand.

### 6.4.2 Obscuring

A simple name may occur in contexts where it may potentially be interpreted as the name of a variable, a type, or a package. In these situations, the rules of §6.5 specify that a variable will be chosen in preference to a type, and that a type will be chosen in preference to a package. Thus, it is may sometimes be impossible to refer to a type or package via its simple name, even though its declaration is in scope and not shadowed. We say that such a declaration is *obscured*.

Obscuring is distinct from shadowing (§6.4.1) and hiding (§8.3, §8.4.8.2, §8.5, §9.3, §9.5).

There is no obscuring between the name of a module and the name of a variable, type, or package; thus, modules may share names with variables, types, and packages, though it is not necessarily recommended to name a module after a package it contains.

The naming conventions of §6.1 help reduce obscuring, but if it does occur, here are some notes about what you can do to avoid it.

When package names occur in expressions:

- If a package name is obscured by a field declaration, then *import* declarations (§7.5) can usually be used to make available the type names declared in that package.
- If a package name is obscured by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

The first component of a package name is normally not easily mistaken for a type name, as a type name normally begins with a single uppercase letter. (The Java programming language does not actually rely on case distinctions to determine whether a name is a package name or a type name.)

Obscuring involving class and interface type names is rare. Names of fields, parameters, and local variables normally do not obscure type names because they conventionally begin with a lowercase letter whereas type names conventionally begin with an uppercase letter.

Method names cannot obscure or be obscured by other names (§6.5.7).

Obscuring involving field names is rare; however:

- If a field name obscures a package name, then an *import* declaration (§7.5) can usually be used to make available the type names declared in that package.
- If a field name obscures a type name, then a fully qualified name for the type can be used unless the type name denotes a local class (§14.3).
- Field names cannot obscure method names.
- If a field name is shadowed by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

Obscuring involving constant names is rare:

- Constant names normally have no lowercase letters, so they will not normally obscure names of packages or types, nor will they normally shadow fields, whose names typically contain at least one lowercase letter.
- Constant names cannot obscure method names, because they are distinguished syntactically.

### 6.5 Determining the Meaning of a Name

The meaning of a name depends on the context in which it is used. The determination of the meaning of a name requires three steps:

- First, context causes a name syntactically to fall into one of seven categories: 
  - ModuleName
  - PackageName
  - TypeName
  - ExpressionName
  - MethodName
  - PackageOrTypeName
  - AmbiguousName

  Typename is less expressive than the other six categories, because it is denoted with TypeIdentifier, which excludes the character sequence var (§3.8).

- Second, a name that is initially classified by its context as an AmbiguousName or as a PackageOrTypeName is then reclassified to be a PackageName, TypeName, or ExpressionName.

- Third, the resulting category then dictates the final determination of the meaning of the name (or a compile-time error if the name has no meaning).

\[
\text{ModuleName:} \\
\quad \text{Identifier} \\
\quad \text{ModuleName} \ . \ \text{Identifier}
\]

\[
\text{PackageName:} \\
\quad \text{Identifier} \\
\quad \text{PackageName} \ . \ \text{Identifier}
\]
The use of context helps to minimize name conflicts between entities of different kinds. Such conflicts will be rare if the naming conventions described in §6.1 are followed. Nevertheless, conflicts may arise unintentionally as types developed by different programmers or different organizations evolve. For example, types, methods, and fields may have the same name. It is always possible to distinguish between a method and a field with the same name, since the context of a use always tells whether a method is intended.

6.5.1 Syntactic Classification of a Name According to Context

A name is syntactically classified as a `ModuleName` in these contexts:

- In a `requires` directive in a module declaration (§7.7.1)
- To the right of `to` in an `exports` or `opens` directive in a module declaration (§7.7.2)

A name is syntactically classified as a `PackageName` in these contexts:

- To the right of `exports` or `opens` in a module declaration
- To the left of the "." in a qualified `PackageName`

A name is syntactically classified as a `TypeName` in these contexts:

- The first eleven non-generic contexts (§6.1):
  1. In a `uses` or `provides` directive in a module declaration (§7.7.1)
2. In a single-type-import declaration (§7.5.1)
3. To the left of the . in a single-static-import declaration (§7.5.3)
4. To the left of the . in a static-import-on-demand declaration (§7.5.4)
5. To the left of the ( in a constructor declaration (§8.8)
6. After the @ sign in an annotation (§9.7)
7. To the left of .class in a class literal (§15.8.2)
8. To the left of .this in a qualified this expression (§15.8.4)
9. To the left of .super in a qualified superclass field access expression (§15.11.2)
10. To the left of .Identifier or .super.Identifier in a qualified method invocation expression (§15.12)
11. To the left of .super:: in a method reference expression (§15.13)

- As the Identifier or dotted Identifier sequence that constitutes any ReferenceType (including a ReferenceType to the left of the brackets in an array type, or to the left of the < in a parameterized type, or in a non-wildcard type argument of a parameterized type, or in an extends or super clause of a wildcard type argument of a parameterized type) in the 16 contexts where types are used (§4.11):
  1. In an extends or implements clause of a class declaration (§8.1.4, §8.1.5, §8.5, §9.5)
  2. In an extends clause of an interface declaration (§9.1.3)
  3. The return type of a method (§8.4, §9.4) (including the type of an element of an annotation type (§9.6.1))
  4. In the throws clause of a method or constructor (§8.4.6, §8.8.5, §9.4)
  5. In an extends clause of a type parameter declaration of a generic class, interface, method, or constructor (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
  6. The type in a field declaration of a class or interface (§8.3, §9.3)
  7. The type in a formal parameter declaration of a method, constructor, or lambda expression (§8.4.1, §8.8.1, §9.4, §15.27.1)
  8. The type of the receiver parameter of a method (§8.4)
10. A type in an exception parameter declaration (§14.20)

11. In an explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (§8.8.7.1, §15.9, §15.12)

12. In an unqualified class instance creation expression, either as the class type to be instantiated (§15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (§15.9.5)

13. The element type in an array creation expression (§15.10.1)

14. The type in the cast operator of a cast expression (§15.16)

15. The type that follows the `instanceof` relational operator (§15.20.2)

16. In a method reference expression (§15.13), as the reference type to search for a member method or as the class type or array type to construct.

The extraction of a `TypeName` from the identifiers of a `ReferenceType` in the 16 contexts above is intended to apply recursively to all sub-terms of the `ReferenceType`, such as its element type and any type arguments.

For example, suppose a field declaration uses the type `p.q.Foo[]`. The brackets of the array type are ignored, and the term `p.q.Foo` is extracted as a dotted sequence of `Identifiers` to the left of the brackets in an array type, and classified as a `TypeName`. A later step determines which of `p`, `q`, and `Foo` is a type name or a package name.

As another example, suppose a cast operator uses the type `p.q.Foo<? extends String>`. The term `p.q.Foo` is again extracted as a dotted sequence of `Identifier` terms, this time to the left of the `<` in a parameterized type, and classified as a `TypeName`. The term `String` is extracted as an `Identifier` in an `extends` clause of a wildcard type argument of a parameterized type, and classified as a `TypeName`.

A name is syntactically classified as an `ExpressionName` in these contexts:

- As the qualifying expression in a qualified superclass constructor invocation (§8.8.7.1)
- As the qualifying expression in a qualified class instance creation expression (§15.9)
- As the array reference expression in an array access expression (§15.10.3)
- As a `PostfixExpression` (§15.14)
- As the left-hand operand of an assignment operator (§15.26)
- As a `VariableAccess` in a `try`-with-resources statement (§14.20.3)
A name is syntactically classified as a *MethodName* in this context:
- Before the "(" in a method invocation expression (§15.12)

A name is syntactically classified as a *PackageOrTypeName* in these contexts:
- To the left of the "." in a qualified *TypeName*
- In a type-import-on-demand declaration (§7.5.2)

A name is syntactically classified as an *AmbiguousName* in these contexts:
- To the left of the "." in a qualified *ExpressionName*
- To the left of the rightmost "." that occurs before the "(" in a method invocation expression
- To the left of the "." in a qualified *AmbiguousName*
- In the default value clause of an annotation type element declaration (§9.6.2)
- To the right of an "=" in an an element-value pair (§9.7.1)
- To the left of :: in a method reference expression (§15.13)

The effect of syntactic classification is to restrict certain kinds of entities to certain parts of expressions:
- The name of a field, parameter, or local variable may be used as an expression (§15.14.1).
- The name of a method may appear in an expression only as part of a method invocation expression (§15.12).
- The name of a class or interface type may appear in an expression only as part of a class literal (§15.8.2), a qualified this expression (§15.8.4), a class instance creation expression (§15.9), an array creation expression (§15.10.1), a cast expression (§15.16), an instanceof expression (§15.20.2), an enum constant (§8.9), or as part of a qualified name for a field or method.
- The name of a package may appear in an expression only as part of a qualified name for a class or interface type.

### 6.5.2 Reclassification of Contextually Ambiguous Names

An *AmbiguousName* is then reclassified as follows.

If the *AmbiguousName* is a simple name, consisting of a single *Identifier*:
- If the *Identifier* appears within the scope (§6.3) of a local variable declaration (§14.4) or parameter declaration (§8.4.1, §8.8.1, §14.20) or field declaration (§8.3) with that name, then the *AmbiguousName* is reclassified as an *ExpressionName*. 
• Otherwise, if a field of that name is declared in the compilation unit (§7.3) containing the Identifier by a single-static-import declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4) then the AmbiguousName is reclassified as an ExpressionName.

• Otherwise, if the Identifier is a valid TypeIdentifier and appears within the scope (§6.3) of a top level class (§8 (Classes)) or interface type declaration (§9 (Interfaces)), a local class declaration (§14.3) or member type declaration (§8.5, §9.5) with that name, then the AmbiguousName is reclassified as a TypeName.

• Otherwise, if the Identifier is a valid TypeIdentifier and a type of that name is declared in the compilation unit (§7.3) containing the Identifier, either by a single-type-import declaration (§7.5.1), or by a type-import-on-demand declaration (§7.5.2), or by a single-static-import declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4), then the AmbiguousName is reclassified as a TypeName.

• Otherwise, the AmbiguousName is reclassified as a PackageName. A later step determines whether or not a package of that name actually exists.

If the AmbiguousName is a qualified name, consisting of a name, a " . ", and an Identifier, then the name to the left of the " . " is first reclassified, for it is itself an AmbiguousName. There is then a choice:

• If the name to the left of the " . " is reclassified as a PackageName, then:
  – If the Identifier is a valid TypeIdentifier, and there is a package whose name is the name to the left of the " . ", and that package contains a declaration of a type whose name is the same as the Identifier, then this AmbiguousName is reclassified as a TypeName.
  – Otherwise, this AmbiguousName is reclassified as a PackageName. A later step determines whether or not a package of that name actually exists.

• If the name to the left of the " . " is reclassified as a TypeName, then:
  – If the Identifier is the name of a method or field of the type denoted by TypeName, then this AmbiguousName is reclassified as an ExpressionName.
  – Otherwise, if the Identifier is a valid TypeIdentifier and is the name of a member type of the type denoted by TypeName, then this AmbiguousName is reclassified as a TypeName.
  – Otherwise, a compile-time error occurs.
• If the name to the left of the "." is reclassified as an ExpressionName, then this
AmbiguousName is reclassified as an ExpressionName. A later step determines
whether or not a member with the name Identifier actually exists.

The requirement that a potential type name be "a valid TypeIdentifier" prevents treating var
as a type name. It is usually redundant, because the rules for declarations already prevent
the introduction of types named var. However, in some cases, a compiler may find a binary
class named var, and we want to be clear that such classes can never be named. The simplest
solution is to consistently check for a valid TypeIdentifier.

Example 6.5.2-1. Reclassification of Contextually Ambiguous Names

Consider the following contrived "library code":

    package org.rpgpoet;
    import java.util.Random;
    public interface Music { Random[] wizards = new Random[4]; }

and then consider this example code in another package:

    package bazola;
    class Gabriel {
        static int n = org.rpgpoet.Music.wizards.length;
    }

First of all, the name org.rpgpoet.Music.wizards.length is classified as an
ExpressionName because it functions as a PostfixExpression. Therefore, each of the names:

 org.rpgpoet.Music.wizards
 org.rpgpoet.Music
 org.rpgpoet
 org

is initially classified as an AmbiguousName. These are then reclassified:

• The simple name org is reclassified as a PackageName (since there is no variable or
type named org in scope).

• Next, assuming that there is no class or interface named rpgpoet in any compilation unit
of package org (and we know that there is no such class or interface because package org
has a subpackage named rpgpoet), the qualified name org.rpgpoet is reclassified as a
PackageName.

• Next, because package org.rpgpoet has an accessible (§6.6) interface type named
Music, the qualified name org.rpgpoet.Music is reclassified as a TypeName.

• Finally, because the name org.rpgpoet.Music is a TypeName, the qualified name
org.rpgpoet.Music.wizards is reclassified as an ExpressionName.
6.5.3 **Meaning of Module Names and Package Names**

The module name M, whether simple or qualified, denotes the module (if any) with that name.

This section does not mandate a compile-time error if no module with that name is observable. Instead, the `requires` directive in a module declaration (§7.7.1) performs its own validation of the module name, while the `exports` and `opens` directives (§7.7.2) are tolerant of non-existent module names.

The meaning of a name classified as a `PackageName` is determined as follows.

6.5.3.1 **Simple Package Names**

If a package name consists of a single `Identifier`, then the identifier must occur in the scope of exactly one declaration of a top level package with this name (§6.3), and that package must be uniquely visible to the current module (§7.4.3), or a compile-time error occurs. The meaning of the package name is that package.

6.5.3.2 **Qualified Package Names**

If a package name is of the form `Q.Id`, then `Q` must also be a package name. The package name `Q.Id` names a package that is the member named `Id` within the package named by `Q`.

If `Q.Id` does not name a package that is uniquely visible to the current module (§7.4.3), then a compile-time error occurs.

6.5.4 **Meaning of PackageOrTypeName**

6.5.4.1 **Simple PackageOrTypeName**

If the `PackageOrTypeName`, `Q`, is a valid `TypeIdentifier` and occurs in the scope of a type named `Q`, then the `PackageOrTypeName` is reclassified as a `TypeName`.

Otherwise, the `PackageOrTypeName` is reclassified as a `PackageName`. The meaning of the `PackageOrTypeName` is the meaning of the reclassified name.

6.5.4.2 **Qualified PackageOrTypeName**

Given a qualified `PackageOrTypeName` of the form `Q.Id`, if `Id` is a valid `TypeIdentifier` and the type or package denoted by `Q` has a member type named `Id`, then the qualified `PackageOrTypeName` name is reclassified as a `TypeName`. 

160
Otherwise, it is reclassified as a *PackageName*. The meaning of the qualified *PackageOrTypeName* is the meaning of the reclassified name.

### 6.5.5 Meaning of Type Names

The meaning of a name classified as a *TypeName* is determined as follows.

#### 6.5.5.1 Simple Type Names

If a type name consists of a single *Identifier*, then the identifier must occur in the scope of exactly one declaration of a type with this name (§6.3), or a compile-time error occurs. The meaning of the type name is that type.

#### 6.5.5.2 Qualified Type Names

If a type name is of the form $Q.Id$, then $Q$ must be either the name of a type in a package uniquely visible to the current module, or the name of a package uniquely visible to the current module (§7.4.3).

If $Id$ names exactly one accessible type (§6.6) that is a member of the type or package denoted by $Q$, then the qualified type name denotes that type.

If $Id$ does not name a member type within $Q$ (§8.5, §9.5), or the member type named $Id$ within $Q$ is not accessible (§6.6), or $Id$ names more than one member type within $Q$, then a compile-time error occurs.

**Example 6.5.5.2-1. Qualified Type Names**

```java
class Test {
   public static void main(String[] args) {
      java.util.Date date =
         new java.util.Date(System.currentTimeMillis());
      System.out.println(date.toLocaleString());
   }
}
```

This program produced the following output the first time it was run:

```
Sun Jan 21 22:56:29 1996
```

In this example, the name `java.util.Date` must denote a type, so we first use the procedure recursively to determine if `java.util` is an accessible type or a package, which it is, and then look to see if the type `Date` is accessible in this package.
6.5.6 Meaning of Expression Names

The meaning of a name classified as an \textit{ExpressionName} is determined as follows.

6.5.6.1 Simple Expression Names

If an expression name consists of a single \textit{Identifier}, then there must be exactly one declaration denoting either a local variable, formal parameter, or field in scope at the point at which the \textit{Identifier} occurs. Otherwise, a compile-time error occurs.

If the declaration denotes an instance variable (§8.3.1.1), the expression name must appear within an instance method (§8.4.3.2), instance variable initializer (§8.3.2), instance initializer (§8.6), or constructor (§8.8). If the expression name appears within a class method, class variable initializer, or static initializer (§8.7), then a compile-time error occurs.

If the declaration declares a \textit{final} variable which is definitely assigned before the simple expression, the meaning of the name is the value of that variable. Otherwise, the meaning of the expression name is the variable declared by the declaration.

If the expression name appears in an assignment context, invocation context, or casting context, then the type of the expression name is the declared type of the field, local variable, or parameter after capture conversion (§5.1.10). Otherwise, the type of the expression name is the declared type of the field, local variable or parameter.

That is, if the expression name appears "on the right hand side", its type is subject to capture conversion. If the expression name is a variable that appears "on the left hand side", its type is not subject to capture conversion.

Example 6.5.6.1-1. Simple Expression Names

```java
class Test {
    static int v;
    static final int f = 3;
    public static void main(String[] args) {
        int i;
        i = 1;
        v = 2;
        f = 33; // compile-time error
        System.out.println(i + " " + v + " " + f);
    }
}
```

In this program, the names used as the left-hand-sides in the assignments to \texttt{i}, \texttt{v}, and \texttt{f} denote the local variable \texttt{i}, the field \texttt{v}, and the value of \texttt{f} (not the variable \texttt{f}, because \texttt{f} is a \texttt{final} variable). The example therefore produces an error at compile time because the
last assignment does not have a variable as its left-hand side. If the erroneous assignment is removed, the modified code can be compiled and it will produce the output:

```
1 2 3
```

### 6.5.6.2 Qualified Expression Names

If an expression name is of the form \( Q.Id \), then \( Q \) has already been classified as a package name, a type name, or an expression name.

If \( Q \) is a package name, then a compile-time error occurs.

If \( Q \) is a type name that names a class type (§8 (Classes)), then:

- If there is not exactly one accessible (§6.6) member of the class type that is a field named \( Id \), then a compile-time error occurs.
- Otherwise, if the single accessible member field is not a class variable (that is, it is not declared static), then a compile-time error occurs.
- Otherwise, if the class variable is declared final, then \( Q.Id \) denotes the value of the class variable.

  The type of the expression \( Q.Id \) is the declared type of the class variable after capture conversion (§5.1.10).

  If \( Q.Id \) appears in a context that requires a variable and not a value, then a compile-time error occurs.

  - Otherwise, \( Q.Id \) denotes the class variable.

    The type of the expression \( Q.Id \) is the declared type of the class variable after capture conversion (§5.1.10).

    Note that this clause covers the use of enum constants (§8.9), since these always have a corresponding final class variable.

If \( Q \) is a type name that names an interface type (§9 (Interfaces)), then:

- If there is not exactly one accessible (§6.6) member of the interface type that is a field named \( Id \), then a compile-time error occurs.

- Otherwise, \( Q.Id \) denotes the value of the field.

  The type of the expression \( Q.Id \) is the declared type of the field after capture conversion (§5.1.10).

  If \( Q.Id \) appears in a context that requires a variable and not a value, then a compile-time error occurs.
If $Q$ is an expression name, let $\tau$ be the type of the expression $Q$:

- If $\tau$ is not a reference type, a compile-time error occurs.
- If there is not exactly one accessible (§6.6) member of the type $\tau$ that is a field named $Id$, then a compile-time error occurs.
- Otherwise, if this field is any of the following:
  - A field of an interface type
  - A final field of a class type (which may be either a class variable or an instance variable)
  - The final field length of an array type (§10.7)

then $Q.Id$ denotes the value of the field, unless it appears in a context that requires a variable and the field is a definitely unassigned blank final field, in which case it yields a variable.

The type of the expression $Q.Id$ is the declared type of the field after capture conversion (§5.1.10).

If $Q.Id$ appears in a context that requires a variable and not a value, and the field denoted by $Q.Id$ is definitely assigned, then a compile-time error occurs.

- Otherwise, $Q.Id$ denotes a variable, the field $Id$ of class $\tau$, which may be either a class variable or an instance variable.

The type of the expression $Q.Id$ is the type of the field member after capture conversion (§5.1.10).

**Example 6.5.6.2-1. Qualified Expression Names**

```java
class Point {
    int x, y;
    static int nPoints;
}

class Test {
    public static void main(String[] args) {
        int i = 0;
        i.x++;        // compile-time error
        Point p = new Point();
        p.nPoints();  // compile-time error
    }
}
```

This program encounters two compile-time errors, because the int variable $i$ has no members, and because $nPoints$ is not a method of class $Point$. 
Example 6.5.6.2-2. Qualifying an Expression with a Type Name

Note that expression names may be qualified by type names, but not by types in general. A consequence is that it is not possible to access a class variable through a parameterized type. For example, given the code:

```java
class Foo<T> {
    public static int classVar = 42;
}
```

the following assignment is illegal:

```java
Foo<String>.classVar = 91; // illegal
```

Instead, one writes:

```java
Foo.classVar = 91;
```

This does not restrict the Java programming language in any meaningful way. Type parameters may not be used in the types of static variables, and so the type arguments of a parameterized type can never influence the type of a static variable. Therefore, no expressive power is lost. The type name `Foo` appears to be a raw type, but it is not; rather, it is the name of the non-generic type `Foo` whose static member is to be accessed (§6.1). Since there is no use of a raw type, there are no unchecked warnings.

### 6.5.7 Meaning of Method Names

The meaning of a name classified as a `MethodName` is determined as follows.

#### 6.5.7.1 Simple Method Names

A simple method name appears in the context of a method invocation expression (§15.12). The simple method name consists of a single `Identifier` which specifies the name of the method to be invoked. The rules of method invocation require that the `Identifier` either denotes a method that is in scope at the point of the method invocation, or denotes a method imported by a single-static-import declaration or static-import-on-demand declaration (§7.5.3, §7.5.4).

Example 6.5.7.1-1. Simple Method Names

The following program demonstrates the role of scoping when determining which method to invoke.

```java
class Super {
    void f2(String s) {}   // f2
    void f3(String s) {}   // f3
    void f3(int i1, int i2) {}  // f3
}
```
For the invocation \texttt{f1(0)}, only one method named \texttt{f1} is in scope. It is the method \texttt{Test.f1(int)}, whose declaration is in scope throughout the body of \texttt{Test} including the anonymous class declaration. §15.12.1 chooses to search in \texttt{class Test} since the anonymous class declaration has no member named \texttt{f1}. Eventually, \texttt{Test.f1(int)} is resolved.

For the invocation \texttt{f2(0)}, two methods named \texttt{f2} are in scope. First, the declaration of the method \texttt{Super.f2(String)} is in scope throughout the anonymous class declaration. Second, the declaration of the method \texttt{Test.f2(int)} is in scope throughout the body of \texttt{Test} including the anonymous class declaration. (Note that neither declaration shadows the other, because at the point where each is declared, the other is not in scope.) §15.12.1 chooses to search in \texttt{class Super} because it has a member named \texttt{f2}. However, \texttt{Super.f2(String)} is not applicable to \texttt{f2(0)}, so a compile-time error occurs. Note that \texttt{class Test} is not searched.

For the invocation \texttt{f3(0)}, three methods named \texttt{f3} are in scope. First and second, the declarations of the methods \texttt{Super.f3(String)} and \texttt{Super.f3(int,int)} are in scope throughout the anonymous class declaration. Third, the declaration of the method \texttt{Test.f3(int)} is in scope throughout the body of \texttt{Test} including the anonymous class declaration. §15.12.1 chooses to search in \texttt{class Super} because it has a member named \texttt{f3}. However, \texttt{Super.f3(String)} and \texttt{Super.f3(int,int)} are not applicable to \texttt{f3(0)}, so a compile-time error occurs. Note that \texttt{class Test} is not searched.

Choosing to search a nested class's superclass hierarchy before the lexically enclosing scope is called the "comb rule" (§15.12.1).

### 6.6 Access Control

The Java programming language provides mechanisms for \textit{access control}, to prevent the users of a package or class from depending on unnecessary details of the
implementation of that package or class. If access is permitted, then the accessed entity is said to be *accessible*.

Note that accessibility is a static property that can be determined at compile time; it depends only on types and declaration modifiers.

Qualified names are a means of access to members of packages and reference types. When the name of such a member is classified from its context (§6.5.1) as a qualified type name (denoting a member of a package or reference type, §6.5.5.2) or a qualified expression name (denoting a member of a reference type, §6.5.6.2), access control is applied.

For example, a single-type-import declaration uses a qualified type name (§7.5.1), so the named type must be accessible from the compilation unit containing the `import` declaration. As another example, a class declaration may use a qualified type name for a superclass (§8.1.5), so again the named type must be accessible.

Some obvious expressions are "missing" from context classification in §6.5.1: field access on a `Primary` (§15.11.1), method invocation on a `Primary` (§15.12), method reference via a `Primary` (§15.13), and the instantiated class in a qualified class instance creation (§15.9). Each of these expressions uses identifiers, rather than names, for the reason given in §6.2. Consequently, access control to members (whether fields, methods, or types) is applied explicitly by field access expressions, method invocation expressions, method reference expressions, and qualified class instance creation expressions. (Note that access to a field may also be denoted by a qualified name occurring as a postfix expression.)

In addition, many statements and expressions allow the use of types rather than type names. For example, a class declaration may use a parameterized type (§4.5) to denote a superclass. Because a parameterized type is not a qualified type name, it is necessary for the class declaration to explicitly perform access control for the denoted superclass. Consequently, most of the statements and expressions that provide contexts in §6.5.1 to classify a `TypeName` also perform their own access control checks.

Beyond access to members of a package or reference type, there is the matter of access to constructors of a reference type. Access control must be checked when a constructor is invoked explicitly or implicitly. Consequently, access control is checked by an explicit constructor invocation statement (§8.8.7.1) and by a class instance creation expression (§15.9.3). Such checks are necessary because §6.5.1 has no mention of explicit constructor invocation statements (because they reference constructor names indirectly) and is unaware of the distinction between the class type denoted by an unqualified class instance creation expression and a constructor of that class type. Also, constructors do not have qualified names, so we cannot rely on access control being checked during classification of qualified type names.

Accessibility affects inheritance of class members (§8.2), including hiding and method overriding (§8.4.8.1).
6.6.1 Determining Accessibility

- If a top level class or interface type is declared `public` and is a member of a package that is exported by a module, then the type may be accessed by any code in the same module, and by any code in another module to which the package is exported, provided that the compilation unit in which the type is declared is visible to that other module (§7.3).

- If a top level class or interface type is declared `public` and is a member of a package that is not exported by a module, then the type may be accessed by any code in the same module.

- If a top level class or interface type is declared with package access, then it may be accessed only from within the package in which it is declared.

A top level class or interface type declared without an access modifier implicitly has package access.

- A member (class, interface, field, or method) of a reference type, or a constructor of a class type, is accessible only if the type is accessible and the member or constructor is declared to permit access:
  - If the member or constructor is declared `public`, then access is permitted.
    All members of interfaces lacking access modifiers are implicitly `public`.
  - Otherwise, if the member or constructor is declared `protected`, then access is permitted only when one of the following is true:
    › Access to the member or constructor occurs from within the package containing the class in which the `protected` member or constructor is declared.
    › Access is correct as described in §6.6.2.
  - Otherwise, if the member or constructor is declared with package access, then access is permitted only when the access occurs from within the package in which the type is declared.
    A class member or constructor declared without an access modifier implicitly has package access.
  - Otherwise, the member or constructor is declared `private`, and access is permitted if and only if it occurs within the body of the top level type (§7.6) that encloses the declaration of the member or constructor.

- An array type is accessible if and only if its element type is accessible.
Example 6.6-1. Access Control

Consider the two compilation units:

```java
package points;
class PointVec { Point[] vec; }
```

and:

```java
package points;
public class Point {
    protected int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

which declare two class types in the package `points`:

- The class type `PointVec` is not `public` and not part of the public interface of the package `points`, but rather can be used only by other classes in the package.

- The class type `Point` is declared `public` and is available to other packages. It is part of the public interface of the package `points`.

- The methods `move`, `getX`, and `getY` of the class `Point` are declared `public` and so are available to any code that uses an object of type `Point`.

- The fields `x` and `y` are declared `protected` and are accessible outside the package `points` only in subclasses of class `Point`, and only when they are fields of objects that are being implemented by the code that is accessing them.

See §6.6.2 for an example of how the `protected` access modifier limits access.

Example 6.6-2. Access to `public` Fields, Methods, and Constructors

A `public` class member or constructor is accessible throughout the package where it is declared and from any other package, provided the package in which it is declared is observable (§7.4.3). For example, in the compilation unit:

```java
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
        moves++;
    }
    public static int moves = 0;
}
```
the public class `Point` has as public members the `move` method and the `moves` field. These public members are accessible to any other package that has access to package `points`. The fields `x` and `y` are not public and therefore are accessible only from within the package `points`.

**Example 6.6-3. Access to public and Non-public Classes**

If a class lacks the `public` modifier, access to the class declaration is limited to the package in which it is declared (§6.6). In the example:

```java
class Point {
    public int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
class PointList {
    Point next, prev;
}
```

two classes are declared in the compilation unit. The class `Point` is available outside the package `points`, while the class `PointList` is available for access only within the package. Thus a compilation unit in another package can access `points.Point`, either by using its fully qualified name:

```java
package pointsUser;
class Test1 {
    public static void main(String[] args) {
        points.Point p = new points.Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

or by using a single-type-import declaration (§7.5.1) that mentions the fully qualified name, so that the simple name may be used thereafter:

```java
package pointsUser;
import points.Point;
class Test2 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

However, this compilation unit cannot use or import `points.PointList`, which is not declared `public` and is therefore inaccessible outside package `points`.

170
Example 6.6-4. Access to Package-Access Fields, Methods, and Constructors

If none of the access modifiers public, protected, or private are specified, a class member or constructor has package access: it is accessible throughout the package that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other package.

If a public class has a method or constructor with package access, then this method or constructor is not accessible to or inherited by a subclass declared outside this package.

For example, if we have:

```java
package points;
public class Point {
    public int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    public void moveAlso(int dx, int dy) { move(dx, dy); }
}
```

then a subclass in another package may declare an unrelated move method, with the same signature (§8.4.2) and return type. Because the original move method is not accessible from package morepoints, super may not be used:

```java
package morepoints;
public class PlusPoint extends points.Point {
    public void move(int dx, int dy) {
        super.move(dx, dy);  // compile-time error
        moveAlso(dx, dy);
    }
}
```

Because move of Point is not overridden by move in PlusPoint, the method moveAlso in Point never calls the method move in PlusPoint. Thus if you delete the super.move call from PlusPoint and execute the test program:

```java
import points.Point;
import morepoints.PlusPoint;
class Test {
    public static void main(String[] args) {
        PlusPoint pp = new PlusPoint();
        pp.move(1, 1);
    }
}
```

it terminates normally. If move of Point were overridden by move in PlusPoint, then this program would recurse infinitely, until a StackOverflowError occurred.
6.6 Access Control

Example 6.6-5. Access to private Fields, Methods, and Constructors

A private class member or constructor is accessible only within the body of the top level class (§7.6) that encloses the declaration of the member or constructor. It is not inherited by subclasses. In the example:

```java
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; }
}
```

the private members ID, masterID, and setMasterID may be used only within the body of class Point. They may not be accessed by qualified names, field access expressions, or method invocation expressions outside the body of the declaration of Point.

See §8.8.10 for an example that uses a private constructor.

6.6.2 Details on protected Access

A protected member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object.

6.6.2.1 Access to a protected Member

Let C be the class in which a protected member is declared. Access is permitted only within the body of a subclass S of C.

A subclass S is regarded as being responsible for the implementation of objects of class C. Depending on C’s accessibility, S may be declared in the same package as C, or in different package of the same module as C, or in a package of a different module entirely.

In addition, access to an instance field or instance method is permitted based on the form of the qualified name, field access expression (§15.11), method invocation expression (§15.12), or method reference expression (§15.13):

- If the access is by (i) a qualified name of the form ExpressionName.Id or TypeName.Id, or (ii) a field access expression of the form Primary.Id, then access to the instance field Id is permitted if and only if the qualifying type is S or a subclass of S.

The qualifying type is the type of the ExpressionName or Primary, or the type denoted by TypeName.
• If the access is by (i) a method invocation expression of the form
  
  \texttt{ExpressionName.Id(...)} or \texttt{TypeName.Id(...)} or \texttt{Primary.Id(...)}, or (ii) a
  
  method reference expression of the form \texttt{ExpressionName :: Id} or \texttt{Primary :: Id}
  or \texttt{ReferenceType :: Id}, then access to the instance method \texttt{Id} is permitted
  if and only if the qualifying type is \texttt{S} or a subclass of \texttt{S}.

  The qualifying type is the type of the \texttt{ExpressionName} or \texttt{Primary}, or the type
  denoted by \texttt{TypeName} or \texttt{ReferenceType}.

  More information about access to protected members can be found in \textit{Checking Access}
  to Protected Members in the Java Virtual Machine} by Alessandro Coglio, in the \textit{Journal}

6.6.2.2 Access to a protected Constructor

Let \texttt{C} be the class in which a protected constructor is declared and let \texttt{S} be the
innermost class in whose declaration the use of the protected constructor occurs. Then:

• If the access is by a superclass constructor invocation \texttt{super(...)}, or a
  qualified superclass constructor invocation \texttt{E.super(...)}, where \texttt{E} is a Primary
  expression, then the access is permitted.

• If the access is by an anonymous class instance creation expression \texttt{new C(...)}
  {...}, or a qualified anonymous class instance creation expression \texttt{E.new C(...)} {...},
  where \texttt{E} is a Primary expression, then the access is permitted.

• If the access is by a simple class instance creation expression \texttt{new C(...)}, or a
  qualified class instance creation expression \texttt{E.new C(...)}, where \texttt{E} is a Primary
  expression, or a method reference expression \texttt{C :: new}, where \texttt{C} is a ClassType,
  then the access is not permitted. A protected constructor can be accessed by a
  class instance creation expression (that does not declare an anonymous class) or a
  method reference expression only from within the package in which it is defined.

Example 6.6.2-1. Access to protected Fields, Methods, and Constructors

Consider this example, where the \texttt{points} package declares:

```java
package points;
public class Point {
    protected int x, y;
    void warp(threePoint.Point3d a) {
        if (a.z > 0) // compile-time error: cannot access a.z
            a.delta(this);
    }
}
```
and the threePoint package declares:

```java
package threePoint;
import points.Point;
public class Point3d extends Point {
  protected int z;
  public void delta(Point p) {
    p.x += this.x; // compile-time error: cannot access p.x
    p.y += this.y; // compile-time error: cannot access p.y
  }
  public void delta3d(Point3d q) {
    q.x += this.x;
    q.y += this.y;
    q.z += this.z;
  }
}
```

A compile-time error occurs in the method delta here: it cannot access the protected members \( x \) and \( y \) of its parameter \( p \), because while Point3d (the class in which the references to fields \( x \) and \( y \) occur) is a subclass of Point (the class in which \( x \) and \( y \) are declared), it is not involved in the implementation of a Point (the type of the parameter \( p \)). The method delta3d can access the protected members of its parameter \( q \), because the class Point3d is a subclass of Point and is involved in the implementation of a Point3d.

The method delta could try to cast (§5.5, §15.16) its parameter to be a Point3d, but this cast would fail, causing an exception, if the class of \( p \) at run time were not Point3d.

A compile-time error also occurs in the method warp: it cannot access the protected member \( z \) of its parameter \( a \), because while the class Point (the class in which the reference to field \( z \) occurs) is involved in the implementation of a Point3d (the type of the parameter \( a \)), it is not a subclass of Point3d (the class in which \( z \) is declared).

### 6.7 Fully Qualified Names and Canonical Names

Every primitive type, named package, top level class, and top level interface has a fully qualified name:

- The fully qualified name of a primitive type is the keyword for that primitive type, namely byte, short, char, int, long, float, double, or boolean.

- The fully qualified name of a named package that is not a subpackage of a named package is its simple name.

- The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package, followed by ".", followed by the simple (member) name of the subpackage.
• The fully qualified name of a top level class or top level interface that is declared in an unnamed package is the simple name of the class or interface.

• The fully qualified name of a top level class or top level interface that is declared in a named package consists of the fully qualified name of the package, followed by "." , followed by the simple name of the class or interface.

Each member class, member interface, and array type may have a fully qualified name:

• A member class or member interface \( M \) of another class or interface \( C \) has a fully qualified name if and only if \( C \) has a fully qualified name.

  In that case, the fully qualified name of \( M \) consists of the fully qualified name of \( C \), followed by "." , followed by the simple name of \( M \).

• An array type has a fully qualified name if and only if its element type has a fully qualified name.

  In that case, the fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "[ ]".

A local class or anonymous class does not have a fully qualified name.

Every primitive type, named package, top level class, and top level interface has a canonical name:

• For every primitive type, named package, top level class, and top level interface, the canonical name is the same as the fully qualified name.

Each member class, member interface, and array type may have a canonical name:

• A member class or member interface \( M \) declared in another class or interface \( C \) has a canonical name if and only if \( C \) has a canonical name.

  In that case, the canonical name of \( M \) consists of the canonical name of \( C \), followed by "." , followed by the simple name of \( M \).

• An array type has a canonical name if and only if its component type has a canonical name.

  In that case, the canonical name of the array type consists of the canonical name of the component type of the array type followed by "[ ]".

A local class or anonymous class does not have a canonical name.

**Example 6.7-1. Fully Qualified Names**

• The fully qualified name of the type \( \text{long} \) is "long".
• The fully qualified name of the package `java.lang` is "java.lang" because it is subpackage `lang` of package `java`.

• The fully qualified name of the class `Object`, which is defined in the package `java.lang`, is "java.lang.Object".

• The fully qualified name of the interface `Enumeration`, which is defined in the package `java.util`, is "java.util.Enumeration".

• The fully qualified name of the type "array of double" is "double[]".

• The fully qualified name of the type "array of array of array of array of `String`" is "java.lang.String[][][][]".

In the code:

```java
package points;
class Point    { int x, y; }
class PointVec { Point[] vec; }
```

the fully qualified name of the type `Point` is "points.Point"; the fully qualified name of the type `PointVec` is "points.PointVec"; and the fully qualified name of the type of the field `vec` of class `PointVec` is "points.Point[]".

**Example 6.7-2. Fully Qualified Names v. Canonical Name**

The difference between a fully qualified name and a canonical name can be seen in code such as:

```java
package p;
class O1 { class I {} }
class O2 extends O1 {}
```

Both `p.O1.I` and `p.O2.I` are fully qualified names that denote the member class `I`, but only `p.O1.I` is its canonical name.
PROGRAMS are organized as sets of packages. The members of a package (§7.1) are class and interface types, which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

Each package has its own set of names for types, which helps to prevent name conflicts. The naming structure for packages is hierarchical.

If a set of packages is sufficiently cohesive, then the packages may be grouped into a module. A module categorizes some or all of its packages as exported, which means their types may be accessed from code outside the module. If a package is not exported by a module, then only code inside the module may access its types. Furthermore, if code in a module wishes to access the packages exported by another module, then the first module must explicitly depend on the second module. Thus, a module controls how its packages use other modules (by specifying dependences) and controls how other modules use its packages (by specifying which of its packages are exported).

Modules and packages may be stored in a file system or in a database (§7.2). Modules and packages that are stored in a file system may have certain constraints on the organization of their compilation units to allow a simple implementation to find module and type declarations easily.

Code in a compilation unit automatically has access to all types declared in its package and also automatically imports all of the public types declared in the predefined package java.lang.

A top level type is accessible (§6.6) outside the package that declares it only if the type is declared public. A top level type is accessible outside the module that declares it only if the type is declared public and is a member of an exported package. A type that is declared public but is not a member of an exported package is accessible only to code inside the module.
For small programs and casual development, a package can be unnamed (§7.4.2) or have a simple name, but if code is to be widely distributed, unique package names should be chosen using qualified names. This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program.

7.1 Package Members

The members of a package are its subpackages and all the top level class types (§7.6, §8 (Classes)) and top level interface types (§9 (Interfaces)) declared in all the compilation units (§7.3) of the package.

For example, in the Java SE Platform API:

- The package java has subpackages awt, applet, io, lang, net, and util, but no compilation units.
- The package java.awt has a subpackage named image, as well as a number of compilation units containing declarations of class and interface types.

If the fully qualified name (§6.7) of a package is $P$, and $Q$ is a subpackage of $P$, then $P.Q$ is the fully qualified name of the subpackage, and furthermore denotes a package.

A package may not contain two members of the same name, or a compile-time error results.

Here are some examples:

- Because the package java.awt has a subpackage image, it cannot (and does not) contain a declaration of a class or interface type named image.

- If there is a package named mouse and a member type Button in that package (which then might be referred to as mouse.Button), then there cannot be any package with the fully qualified name mouse.Button or mouse.Button.Click.

- If com.nighthacks.java.jag is the fully qualified name of a type, then there cannot be any package whose fully qualified name is either com.nighthacks.java.jag or com.nighthacks.java.jag.scrabble.

It is however possible for members of different packages to have the same simple name. For example, it is possible to declare a package:

```java
package vector;
public class Vector { Object[] vec; }
```
that has as a member a `public` class named `Vector`, even though the package `java.util` also declares a class named `Vector`. These two class types are different, reflected by the fact that they have different fully qualified names (§6.7). The fully qualified name of this example `Vector` is `vector.Vector`, whereas `java.util.Vector` is the fully qualified name of the `Vector` class included in the Java SE Platform. Because the package `vector` contains a class named `Vector`, it cannot also have a subpackage named `Vector`.

The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself other than the prohibition against a package having a subpackage with the same simple name as a top level type (§7.6) declared in that package.

For example, there is no special access relationship between a package named `oliver` and another package named `oliver.twist`, or between packages named `evelyn.wood` and `evelyn.waugh`. That is, the code in a package named `oliver.twist` has no better access to the types declared within package `oliver` than code in any other package.

### 7.2 Host Support for Modules and Packages

Each host system determines how modules, packages, and compilation units are created and stored.

Each host system determines which compilation units are *observable* in a particular compilation (§7.3). Each host system also determines which observable compilation units are *associated with* a module. The observability of compilation units associated with a module determines which modules are observable (§7.7.3) and which packages are visible within those modules (§7.4.3).

The host system is free to determine that a compilation unit which contains a module declaration is not, in fact, observable, and thus is not associated with the module declared therein. This enables a compiler to choose which directory on a `modulesourcepath` is "really" the embodiment of a given module. However, if the host system determines that a compilation unit which contains a module declaration is observable, then §7.4.3 mandates that the compilation unit must be associated with the module declared therein, and not with any other module.

The host system is free to determine that a compilation unit which contains a type declaration is (first) observable and (second) associated with an unnamed module or an automatic module - despite no declaration of an unnamed or automatic module existing in any compilation unit, observable or otherwise.

In simple implementations of the Java SE Platform, packages and compilation units may be stored in a local file system. Other implementations may store them using a distributed file system or some form of database.
If a host system stores packages and compilation units in a database, then the
database must not impose the optional restrictions (§7.6) on compilation units
permissible in file-based implementations.

For example, a system that uses a database to store packages may not enforce a maximum
of one public class or interface per compilation unit.

Systems that use a database must, however, provide an option to convert a
program to a form that obeys the restrictions, for purposes of export to file-based
implementations.

As an extremely simple example of storing packages in a file system, all the packages
and source and binary code in a project might be stored in a single directory and its
subdirectories. Each immediate subdirectory of this directory would represent a top level
package, that is, one whose fully qualified name consists of a single simple name. Each
further level of subdirectory would represent a subpackage of the package represented by
the containing directory, and so on.

The directory might contain the following immediate subdirectories:

```
com
gls
jag
java
wnj
```

where directory java would contain the Java SE Platform packages; the directories jag,
gle, and wnj might contain packages that three of the authors of this specification created
for their personal use and to share with each other within this small group; and the directory
com would contain packages procured from companies that used the conventions described
in §6.1 to generate unique names for their packages.

Continuing the example, the directory java would contain, among others, the following
subdirectories:

```
applet
awt
io
lang
net
util
```

corresponding to the packages java.applet, java.awt, java.io, java.lang,
java.net, and java.util that are defined as part of the Java SE Platform API.

Still continuing the example, if we were to look inside the directory util, we might see
the following files:
where each of the .java files contains the source for a compilation unit (§7.3) that contains the definition of a class or interface whose binary compiled form is contained in the corresponding .class file.

Under this simple organization of packages, an implementation of the Java SE Platform would transform a package name into a pathname by concatenating the components of the package name, placing a file name separator (directory indicator) between adjacent components.

For example, if this simple organization were used on an operating system where the file name separator is /, the package name:

```
jag.scrabble.board
```

would be transformed into the directory name:

```
jag/scrabble/board
```

A package name component or class name might contain a character that cannot correctly appear in a host file system's ordinary directory name, such as a Unicode character on a system that allows only ASCII characters in file names. As a convention, the character can be escaped by using, say, the @ character followed by four hexadecimal digits giving the numeric value of the character, as in the \uxxxx escape (§3.3).

Under this convention, the package name:

```
children.activities.crafts.papierMâché
```

which can also be written using full Unicode as:

```
children.activities.crafts.papierMâché
```

might be mapped to the directory name:

```
children/activities/crafts/papierM@00e2ch@00e9
```

If the @ character is not a valid character in a file name for some given host file system, then some other character that is not valid in a identifier could be used instead.
7.3 Compilation Units

CompilationUnit is the goal symbol (§2.1) for the syntactic grammar (§2.3) of Java programs. It is defined by the following production:

\[
\text{CompilationUnit:} \\
\quad \text{OrdinaryCompilationUnit} \\
\quad \text{ModularCompilationUnit}
\]

\[
\text{OrdinaryCompilationUnit:} \\
\quad \{ \text{PackageDeclaration} \} \{ \text{ImportDeclaration} \} \{ \text{TypeDeclaration} \}
\]

\[
\text{ModularCompilationUnit:} \\
\quad \{ \text{ImportDeclaration} \} \text{ ModuleDeclaration}
\]

An ordinary compilation unit consists of three parts, each of which is optional:

- A package declaration (§7.4), giving the fully qualified name (§6.7) of the package to which the compilation unit belongs. A compilation unit that has no package declaration is part of an unnamed package (§7.4.2).
- import declarations (§7.5) that allow types from other packages and static members of types to be referred to using their simple names.
- Top level type declarations (§7.6) of class and interface types.

A modular compilation unit consists of a module declaration (§7.7), optionally preceded by import declarations. The import declarations allow types from packages in this module and other modules, as well as static members of types, to be referred to using their simple names within the module declaration.

Every compilation unit implicitly imports every public type name declared in the predefined package java.lang, as if the declaration import java.lang.*; appeared at the beginning of each compilation unit immediately after any package declaration. As a result, the names of all those types are available as simple names in every compilation unit.

The host system determines which compilation units are observable, except for the compilation units in the predefined package java and its subpackages lang and io, which are all always observable.

Each observable compilation unit may be associated with a module, as follows:
• The host system may determine that an observable ordinary compilation unit is associated with a module chosen by the host system, except for the ordinary compilation units in the predefined package `java` and its subpackages `lang` and `io`, which are all associated with the `java.base` module.

• The host system must determine that an observable modular compilation unit is associated with the module declared by the modular compilation unit.

The observability of a compilation unit influences the observability of its package (§7.4.3), while the association of an observable compilation unit with a module influences the observability of that module (§7.7.6).

When compiling the modular and ordinary compilation units associated with a module $M$, the host system must respect the dependences specified in $M$’s declaration. Specifically, the host system must limit the ordinary compilation units that would otherwise be observable, to only those that are visible to $M$. The ordinary compilation units that are visible to $M$ are the observable ordinary compilation units associated with the modules that are read by $M$. The modules read by $M$ are given by the result of resolution, as described in the `java.lang.module` package specification, with $M$ as the only root module. The host system must perform resolution to determine the modules read by $M$; it is a compile-time error if resolution fails for any of the reasons described in the `java.lang.module` package specification.

The readability relation is reflexive, so $M$ reads itself, and thus all of the modular and ordinary compilation units associated with $M$ are visible to $M$.

The modules read by $M$ drive the packages that are uniquely visible to $M$ (§7.4.3), which in turn drives both the top level packages in scope and the meaning of package names for code in the modular and ordinary compilation units associated with $M$ (§6.3, §6.5.3, §6.5.5).

The rules above ensure that package/type names used in annotations in a modular compilation unit (in particular, annotations applied to the module declaration) are interpreted as if they appeared in an ordinary compilation unit associated with the module.

Types declared in different ordinary compilation units can refer to each other, circularly. A Java compiler must arrange to compile all such types at the same time.

### 7.4 Package Declarations

A package declaration appears within an ordinary compilation unit to indicate the package to which the compilation unit belongs.
7.4.1 Named Packages

A package declaration in an ordinary compilation unit specifies the name (§6.2) of the package to which the compilation unit belongs.

\[
\text{PackageDeclaration:} \\
\{\text{PackageModifier}\} \text{package Identifier } \{. \text{Identifier}\} ;
\]

The package name mentioned in a package declaration must be the fully qualified name of the package (§6.7).

The scope and shadowing of a package declaration is specified in §6.3 and §6.4.

The rules for annotation modifiers on a package declaration are specified in §9.7.4 and §9.7.5.

At most one annotated package declaration is permitted for a given package.

The manner in which this restriction is enforced must, of necessity, vary from implementation to implementation. The following scheme is strongly recommended for file-system-based implementations: The sole annotated package declaration, if it exists, is placed in a source file called package-info.java in the directory containing the source files for the package. This file does not contain the source for a class called package-info; indeed it would be illegal for it to do so, as package-info is not a legal identifier. Typically package-info.java contains only a package declaration, preceded immediately by the annotations on the package. While the file could technically contain the source code for one or more classes with package access, it would be very bad form.

It is recommended that package-info.java, if it is present, take the place of package.html for javadoc and other similar documentation generation systems. If this file is present, the documentation generation tool should look for the package documentation comment immediately preceding the (possibly annotated) package declaration in package-info.java. In this way, package-info.java becomes the sole repository for package-level annotations and documentation. If, in future, it becomes desirable to add any other package-level information, this file should prove a convenient home for this information.

7.4.2 Unnamed Packages

An ordinary compilation unit that has no package declaration is part of an unnamed package.

Unnamed packages are provided by the Java SE Platform principally for convenience when developing small or temporary applications or when just beginning development.
An unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package.

An implementation of the Java SE Platform must support at least one unnamed package. An implementation may support more than one unnamed package, but is not required to do so. Which ordinary compilation units are in each unnamed package is determined by the host system.

The host system must associate ordinary compilation units in an unnamed package with an unnamed module (§7.7.5), not a named module.

Example 7.4.2-1. Unnamed Package

The compilation unit:

```java
class FirstCall {
    public static void main(String[] args) {
        System.out.println("Mr. Watson, come here. " + "I want you.");
    }
}
```

defines a very simple compilation unit as part of an unnamed package.

In implementations of the Java SE Platform that use a hierarchical file system for storing packages, one typical strategy is to associate an unnamed package with each directory; only one unnamed package is observable at a time, namely the one that is associated with the "current working directory". The precise meaning of "current working directory" depends on the host system.

7.4.3 Package Observability and Visibility

A package is observable if and only if at least one of the following is true:

- An ordinary compilation unit containing a declaration of the package is observable (§7.3).
- A subpackage of the package is observable.

The packages `java`, `java.lang`, and `java.io` are always observable.

One can conclude this from the rule above and from the rules of observable compilation units, as follows. The predefined package `java.lang` declares the class `Object`, so the compilation unit for `Object` is always observable (§7.3). Hence, the `java.lang` package is observable, and the `java` package also. Furthermore, since `Object` is observable, the array type `Object[]` implicitly exists. Its superinterface `java.io.Serializable` (§10.1) also exists, hence the `java.io` package is observable.
A package is visible to a module \( M \) if and only if an ordinary compilation unit containing a declaration of the package is visible to \( M \).

Package visibility is meant to imply that a package is observable in a useful way to a given module. It is generally not useful to know that package \( P \) is observable merely because a subpackage \( P.Q \) is observable. For example, suppose \( P.Q \) is observable (in module \( M1 \)) and \( P.R \) is observable (in module \( M2 \)); then, \( P \) is observable, but where? In \( M1 \), or \( M2 \), or both? The question is redundant; during compilation of module \( N \) that requires only \( M1 \), it matters that \( P.Q \) is observable, but it does not matter that \( P \) is observable.

A package is uniquely visible to a module \( M \) if and only if one of the following holds:

- An ordinary compilation unit associated with \( M \) contains a declaration of the package, and \( M \) does not read any other module that exports the package to \( M \).
- No ordinary compilation unit associated with \( M \) contains a declaration of the package, and \( M \) reads exactly one other module that exports the package to \( M \).

### 7.5 Import Declarations

An import declaration allows a named type or a static member to be referred to by a simple name (§6.2) that consists of a single identifier.

Without the use of an appropriate import declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use a fully qualified name (§6.7).

**ImportDeclaration:**
- **SingleTypeImportDeclaration**
- **TypeImportOnDemandDeclaration**
- **SingleStaticImportDeclaration**
- **StaticImportOnDemandDeclaration**

- A single-type-import declaration (§7.5.1) imports a single named type, by mentioning its canonical name (§6.7).
- A type-import-on-demand declaration (§7.5.2) imports all the accessible types of a named type or named package as needed, by mentioning the canonical name of a type or package.
- A single-static-import declaration (§7.5.3) imports all accessible static members with a given name from a type, by giving its canonical name.
• A static-import-on-demand declaration (§7.5.4) imports all accessible static members of a named type as needed, by mentioning the canonical name of a type.

The scope and shadowing of a type or member imported by these declarations is specified in §6.3 and §6.4.

An import declaration makes types or members available by their simple names only within the compilation unit that actually contains the import declaration. The scope of the type(s) or member(s) introduced by an import declaration specifically does not include other compilation units in the same package, other import declarations in the current compilation unit, or a package declaration in the current compilation unit (except for the annotations of a package declaration).

7.5.1 Single-Type-Import Declarations

A single-type-import declaration imports a single type by giving its canonical name, making it available under a simple name in the module, class, and interface declarations of the compilation unit in which the single-type-import declaration appears.

    SingleTypeImportDeclaration:
      import TypeName ;

The TypeName must be the canonical name of a class type, interface type, enum type, or annotation type (§6.7).

The type must be either a member of a named package, or a member of a type whose outermost lexically enclosing type declaration (§8.1.3) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named type is not accessible (§6.6).

If two single-type-import declarations in the same compilation unit attempt to import types with the same simple name, then a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored.

If the type imported by the single-type-import declaration is declared in the compilation unit that contains the import declaration, the import declaration is ignored.

If a single-type-import declaration imports a type whose simple name is n, and the compilation unit also declares a top level type (§7.6) whose simple name is n, a compile-time error occurs.

If a compilation unit contains both a single-type-import declaration that imports a type whose simple name is n, and a single-static-import declaration (§7.5.3) that
imports a type whose simple name is \( n \), a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored.

**Example 7.5.1-1. Single-Type-Import**

```java
import java.util.Vector;
```

causes the simple name `Vector` to be available within the class and interface declarations in a compilation unit. Thus, the simple name `Vector` refers to the type declaration `Vector` in the package `java.util` in all places where it is not shadowed (§6.4.1) or obscured (§6.4.2) by a declaration of a field, parameter, local variable, or nested type declaration with the same name.

Note that the actual declaration of `java.util.Vector` is generic (§8.1.2). Once imported, the name `Vector` can be used without qualification in a parameterized type such as `Vector<String>`, or as the raw type `Vector`. A related limitation of the `import` declaration is that a nested type declared inside a generic type declaration can be imported, but its outer type is always erased.

**Example 7.5.1-2. Duplicate Type Declarations**

This program:

```java
import java.util.Vector;
class Vector { Object[] vec; }
```

causes a compile-time error because of the duplicate declaration of `Vector`, as does:

```java
import java.util.Vector;
import myVector.Vector;
```

where `myVector` is a package containing the compilation unit:

```java
package myVector;
public class Vector { Object[] vec; }
```

**Example 7.5.1-3. No Import of a Subpackage**

Note that an `import` declaration cannot import a subpackage, only a type.

For example, it does not work to try to import `java.util` and then use the name `util.Random` to refer to the type `java.util.Random`:

```java
import java.util;
class Test { util.Random generator; }
// incorrect: compile-time error
```
Example 7.5.1-4. Importing a Type Name that is also a Package Name

Package names and type names are usually different under the naming conventions described in §6.1. Nevertheless, in a contrived example where there is an unconventionally-named package `Vector`, which declares a public class whose name is `Mosquito`:

```java
package Vector;
public class Mosquito { int capacity; }
```

and then the compilation unit:

```java
package strange;
import java.util.Vector;
import Vector.Mosquito;
class Test {
    public static void main(String[] args) {
        System.out.println(new Vector().getClass());
        System.out.println(new Mosquito().getClass());
    }
}
```

the single-type-import declaration importing class `Vector` from package `java.util` does not prevent the package name `Vector` from appearing and being correctly recognized in subsequent `import` declarations. The example compiles and produces the output:

```
class java.util.Vector
class Vector.Mosquito
```

### 7.5.2 Type-Import-on-Demand Declarations

A `type-import-on-demand declaration` allows all accessible types of a named package or type to be imported as needed.

**TypeImportOnDemandDeclaration:**

```java
import PackageOrTypeName . * ;
```

The `PackageOrTypeName` must be the canonical name (§6.7) of a package, a class type, an interface type, an enum type, or an annotation type.

If the `PackageOrTypeName` denotes a type (§6.5.4), then the type must be either a member of a named package, or a member of a type whose outermost lexically enclosing type declaration (§8.1.3) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named package is not uniquely visible to the current module (§7.4.3), or if the named type is not accessible (§6.6).
It is not a compile-time error to name either `java.lang` or the named package of the current compilation unit in a type-import-on-demand declaration. The type-import-on-demand declaration is ignored in such cases.

Two or more type-import-on-demand declarations in the same compilation unit may name the same type or package. All but one of these declarations are considered redundant; the effect is as if that type was imported only once.

If a compilation unit contains both a type-import-on-demand declaration and a static-import-on-demand declaration (§7.5.4) that name the same type, the effect is as if the `static` member types of that type (§8.5, §9.5) were imported only once.

**Example 7.5.2-1. Type-Import-on-Demand**

```java
import java.util.*;
```

causes the simple names of all public types declared in the package `java.util` to be available within the class and interface declarations of the compilation unit. Thus, the simple name `Vector` refers to the type `Vector` in the package `java.util` in all places in the compilation unit where that type declaration is not shadowed (§6.4.1) or obscured (§6.4.2).

The declaration might be shadowed by a single-type-import declaration of a type whose simple name is `Vector`; by a type named `Vector` and declared in the package to which the compilation unit belongs; or any nested classes or interfaces.

The declaration might be obscured by a declaration of a field, parameter, or local variable named `Vector`.

(It would be unusual for any of these conditions to occur.)

### 7.5.3 Single-Static-Import Declarations

A single-static-import declaration imports all accessible `static` members with a given simple name from a type. This makes these `static` members available under their simple name in the module, class, and interface declarations of the compilation unit in which the single-static-import declaration appears.

**SingleStaticImportDeclaration:**

```java
import static TypeName . Identifier ;
```

The `TypeName` must be the canonical name (§6.7) of a class type, interface type, enum type, or annotation type.
The type must be either a member of a named package, or a member of a type whose outermost lexically enclosing type declaration (§8.1.3) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named type is not accessible (§6.6).

The Identifier must name at least one static member of the named type. It is a compile-time error if there is no static member of that name, or if all of the named members are not accessible.

It is permissible for one single-static-import declaration to import several fields or types with the same name, or several methods with the same name and signature. This occurs when the named type inherits multiple fields, member types, or methods, all with the same name, from its own supertypes.

If two single-static-import declarations in the same compilation unit attempt to import types with the same simple name, then a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored.

If a single-static-import declaration imports a type whose simple name is \( n \), and the compilation unit also declares a top level type (§7.6) whose simple name is \( n \), a compile-time error occurs.

If a compilation unit contains both a single-static-import declaration that imports a type whose simple name is \( n \), and a single-type-import declaration (§7.5.1) that imports a type whose simple name is \( n \), a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored.

### 7.5.4 Static-Import-on-Demand Declarations

A static-import-on-demand declaration allows all accessible static members of a named type to be imported as needed.

```
StaticImportOnDemandDeclaration:
    import static TypeName : * ;
```

The TypeName must be the canonical name (§6.7) of a class type, interface type, enum type, or annotation type.

The type must be either a member of a named package, or a member of a type whose outermost lexically enclosing type declaration (§8.1.3) is a member of a named package, or a compile-time error occurs.

It is a compile-time error if the named type is not accessible (§6.6).
Two or more static-import-on-demand declarations in the same compilation unit may name the same type; the effect is as if there was exactly one such declaration.

Two or more static-import-on-demand declarations in the same compilation unit may name the same member; the effect is as if the member was imported exactly once.

It is permissible for one static-import-on-demand declaration to import several fields or types with the same name, or several methods with the same name and signature. This occurs when the named type inherits multiple fields, member types, or methods, all with the same name, from its own supertypes.

If a compilation unit contains both a static-import-on-demand declaration and a type-import-on-demand declaration (§7.5.2) that name the same type, the effect is as if the static member types of that type (§8.5, §9.5) were imported only once.

7.6 Top Level Type Declarations

A top level type declaration declares a top level class type (§8 (Classes)) or a top level interface type (§9 (Interfaces)).

\[ TypeDeclaration: \]
\[ ClassDeclaration \]
\[ InterfaceDeclaration \]
\]

Extra ";" tokens appearing at the level of type declarations in a compilation unit have no effect on the meaning of the compilation unit. Stray semicolons are permitted in the Java programming language solely as a concession to C++ programmers who are used to placing ";" after a class declaration. They should not be used in new Java code.

In the absence of an access modifier, a top level type has package access: it is accessible only within ordinary compilation units of the package in which it is declared (§6.6.1). A type may be declared public to grant access to the type from code in other packages of the same module, and potentially from code in packages of other modules.

It is a compile-time error if a top level type declaration contains any one of the following access modifiers: protected, private, or static.

It is a compile-time error if the name of a top level type appears as the name of any other top level class or interface type declared in the same package.
The scope and shadowing of a top level type is specified in §6.3 and §6.4.
The fully qualified name of a top level type is specified in §6.7.

Example 7.6-1. Conflicting Top Level Type Declarations

```java
class test { import java.util.Vector; class Point { int x, y; } interface Point { // compile-time error #1 int getR(); int getTheta(); } class Vector { Point[] pts; } // compile-time error #2
```

Here, the first compile-time error is caused by the duplicate declaration of the name `Point` as both a class and an interface in the same package. A second compile-time error is the attempt to declare the name `Vector` both by a class type declaration and by a single-type-import declaration.

Note, however, that it is not an error for the name of a class to also name a type that otherwise might be imported by a type-import-on-demand declaration (§7.5.2) in the compilation unit (§7.3) containing the class declaration. Thus, in this program:

```java
class test { import java.util.*; class Vector {} // not a compile-time error
```

the declaration of the class `Vector` is permitted even though there is also a class `java.util.Vector`. Within this compilation unit, the simple name `Vector` refers to the class `test.Vector`, not to `java.util.Vector` (which can still be referred to by code within the compilation unit, but only by its fully qualified name).

Example 7.6-2. Scope of Top Level Types

```java
class Point { int x, y; // coordinates PointColor color; // color of this point Point next; // next point with this color static int nPoints; }
class PointColor { Point first; // first point with this color PointColor(int color) { this.color = color; } private int color; // color components }
```

This program defines two classes that use each other in the declarations of their class members. Because the class types `Point` and `PointColor` have all the type declarations...
in package points, including all those in the current compilation unit, as their scope, this program compiles correctly. That is, forward reference is not a problem.

**Example 7.6-3. Fully Qualified Names**

```java
class Point { int x, y; }
```

In this code, the class Point is declared in a compilation unit with no package declaration, and thus Point is its fully qualified name, whereas in the code:

```java
package vista;
class Point { int x, y; }
```

the fully qualified name of the class Point is vista.Point. (The package name vista is suitable for local or personal use; if the package were intended to be widely distributed, it would be better to give it a unique package name (§6.1.).)

An implementation of the Java SE Platform must keep track of types within packages by the combination of their enclosing module names and their binary names (§13.1). Multiple ways of naming a type must be expanded to binary names to make sure that such names are understood as referring to the same type.

For example, if a compilation unit contains the single-type-import declaration (§7.5.1):

```java
import java.util.Vector;
```

then within that compilation unit, the simple name Vector and the fully qualified name java.util.Vector refer to the same type.

If and only if packages are stored in a file system (§7.2), the host system may choose to enforce the restriction that it is a compile-time error if a type is not found in a file under a name composed of the type name plus an extension (such as .java or .jav) if either of the following is true:

- The type is referred to by code in other ordinary compilation units of the package in which the type is declared.
- The type is declared public (and therefore is potentially accessible from code in other packages).

This restriction implies that there must be at most one such type per compilation unit. This restriction makes it easy for a Java compiler to find a named class within a package. In practice, many programmers choose to put each class or interface type in its own compilation unit, whether or not it is public or is referred to by code in other compilation units.
For example, the source code for a public type `wet.sprocket.Toad` would be found in a file `Toad.java` in the directory `wet/sprocket`, and the corresponding object code would be found in the file `Toad.class` in the same directory.

### 7.7 Module Declarations

A module declaration specifies a new named module. A named module specifies *dependences* on other modules to define the universe of classes and interfaces available to its own code; and specifies which of its packages are *exported* or *opened* in order to populate the universe of classes and interfaces available to other modules which specify a dependence on it.

A "dependence" is what is expressed by a `requires` directive, independent of whether a module exists with the name specified by the directive. A "dependency" is the observable module enumerated by resolution (as described in the `java.lang.module` package specification) for a given `requires` directive. Generally, the rules of the Java programming language are more interested in dependences than dependencies.

#### ModuleDeclaration:

```
{ Annotation } [open] module Identifier { . Identifier }
{ { ModuleDirective } }
```

A module declaration introduces a module name that can be used in other module declarations to express relationships between modules. A module name consists of one or more Java identifiers (§3.8) separated by "." tokens.

There are two kinds of modules: *normal modules* and *open modules*. The kind of a module determines the nature of access to the module's types, and the members of those types, for code outside the module.

A normal module, without the `open` modifier, grants access at compile time and run time to types in only those packages which are explicitly exported.

An open module, with the `open` modifier, grants access at compile time to types in only those packages which are explicitly exported, but grants access at run time to types in all its packages, as if all packages had been exported.

For code outside a module (whether the module is normal or open), the access granted at compile time or run time to types in the module's exported packages is specifically to the public and protected types in those packages, and the public and protected members of those types (§6.6). No access is granted at compile time or run time to types, or their members, in packages which are not exported. Code inside the module may access public and protected types, and the public and
protected members of those types, in all packages in the module at both compile time and run time.

Distinct from access at compile time and access at run time, the Java SE Platform provides reflective access via the Core Reflection API (§1.4). A normal module grants reflective access to types in only those packages which are explicitly exported or explicitly opened (or both). An open module grants reflective access to types in all its packages, as if all packages had been opened.

For code outside a normal module, the reflective access granted to types in the module's exported (and not opened) packages is specifically to the public and protected types in those packages, and the public and protected members of those types. The reflective access granted to types in the module's opened packages (whether exported or not) is to all types in those packages, and all members of those types. No reflective access is granted to types, or their members, in packages which are not exported or opened. Code inside the module enjoys reflective access to all types, and all their members, in all packages in the module.

For code outside an open module, the reflective access granted to types in the module's opened packages (that is, all packages in the module) is to all types in those packages, and all members of those types. Code inside the module enjoys reflective access to all types, and all their members, in all packages in the module.

The directives of a module declaration specify the module's dependences on other modules (via requires, §7.7.1), the packages it makes available to other modules (via exports and opens, §7.7.2), the services it consumes (via uses, §7.7.3), and the services it provides (via provides, §7.7.4).

\[
\text{ModuleDirective:} \\
\quad \text{requires} \{\text{RequiresModifier}\} \text{ModuleName} ; \\
\quad \text{exports} \text{PackageName} \text{[to ModuleName[, ModuleName]]} ; \\
\quad \text{opens} \text{PackageName} \text{[to ModuleName[, ModuleName]]} ; \\
\quad \text{uses} \text{TypeName} ; \\
\quad \text{provides} \text{TypeName with TypeName[, TypeName]} ;
\]

\[
\text{RequiresModifier:} \\
\quad \text{(one of)} \\
\quad \text{transitive static}
\]

If and only if packages are stored in a file system (§7.2), the host system may choose to enforce the restriction that it is a compile-time error if a module declaration is not found in a file under a name composed of module-info plus an extension (such as .java or .jav).
To aid comprehension, it is customary, though not required, for a module declaration to group its directives, so that the `requires` directives which pertain to modules are visually distinct from the `exports` and `opens` directives which pertain to packages, and from the `uses` and `provides` directives which pertain to services. For example:

```java
module com.example.foo {
    requires com.example.foo.http;
    requires java.logging;
    requires transitive com.example.foo.network;

    exports com.example.foo.bar;
    exports com.example.foo.internal to com.example.foo.probe;

    opens com.example.foo.quux;
    opens com.example.foo.internal to com.example.foo.network,
        com.example.foo.probe;

    uses com.example.foo.spi.Intf;
    provides com.example.foo.spi.Intf with com.example.foo.Impl;
}
```

The `opens` directives can be avoided if the module is open:

```java
open module com.example.foo {
    requires com.example.foo.http;
    requires java.logging;
    requires transitive com.example.foo.network;

    exports com.example.foo.bar;
    exports com.example.foo.internal to com.example.foo.probe;

    uses com.example.foo.spi.Intf;
    provides com.example.foo.spi.Intf with com.example.foo.Impl;
}
```

Development tools for the Java programming language are encouraged to highlight `requires transitive` directives and unqualified `exports` directives, as these form the primary API of a module.

### 7.7.1 Dependences

The `requires` directive specifies the name of a module on which the current module has a dependence.

A `requires` directive must not appear in the declaration of the `java.base` module, or a compile-time error occurs, because it is the primordial module and has no dependences (§8.1.4).
If the declaration of a module does not express a dependence on the `java.base` module, and the module is not itself `java.base`, then the module has an implicitly declared dependence on the `java.base` module.

The `requires` keyword may be followed by the modifier `transitive`. This causes any module which `requires` the current module to have an implicitly declared dependence on the module specified by the `requires transitive` directive.

The `requires` keyword may be followed by the modifier `static`. This specifies that the dependence, while mandatory at compile time, is optional at run time.

If the declaration of a module expresses a dependence on the `java.base` module, and the module is not itself `java.base`, then it is a compile-time error if a modifier appears after the `requires` keyword.

It is a compile-time error if more than one `requires` directive in a module declaration specifies the same module name.

It is a compile-time error if resolution, as described in the `java.lang.module` package specification, with the current module as the only root module, fails for any of the reasons described in the `java.lang.module` package specification.

For example, if a `requires` directive specifies a module that is not observable, or if the current module directly or indirectly expresses a dependence on itself.

If resolution succeeds, then its result specifies the modules that are read by the current module. The modules read by the current module determine which ordinary compilation units are visible to the current module (§7.3). The types declared in those ordinary compilation units (and only those ordinary compilation units) may be accessible to code in the current module (§6.6).

The Java SE Platform distinguishes between named modules that are explicitly declared (that is, with a module declaration) and named modules that are implicitly declared (that is, automatic modules). However, the Java programming language does not surface the distinction: `requires` directives refer to named modules without regard for whether they are explicitly declared or implicitly declared.

While automatic modules are convenient for migration, they are unreliable in the sense that their names and exported packages may change when their authors convert them to explicitly declared modules. A Java compiler is encouraged to issue a warning if a `requires` directive refers to an automatic module. An especially strong warning is recommended if the `transitive` modifier appears in the directive.

**Example 7.1.1-1. Resolution of `requires transitive` directives**

Suppose there are four module declarations as follows:
module m.A {
    requires m.B;
}

module m.B {
    requires transitive m.C;
}

module m.C {
    requires transitive m.D;
}

module m.D {
    exports p;
}

where the package p exported by m.D is declared as follows:

package p;
public class Point {} 

and where a package client in module m.A refers to the public type Point in the exported package p:

package client;
import p.Point;
public class Test {
    public static void main(String[] args) {
        System.out.println(new Point());
    }
}

The modules may be compiled as follows, assuming that the current directory has one subdirectory per module, named after the module it contains:

javac --module-source-path . -d . --module m.D
javac --module-source-path . -d . --module m.C
javac --module-source-path . -d . --module m.B
javac --module-source-path . -d . --module m.A

The program client.Test may be run as follows:

java --module-path . --module m.A/client.Test

The reference from code in m.A to the exported public type Point in m.D is legal because m.A reads m.D, and m.D exports the package containing Point. Resolution determines that m.A reads m.D as follows:

• m.A requires m.B and therefore reads m.B.
• Since \( m.A \) reads \( m.B \), and since \( m.B \) requires transitive \( m.C \), resolution determines that \( m.A \) reads \( m.C \).

• Then, since \( m.A \) reads \( m.C \), and since \( m.C \) requires transitive \( m.D \), resolution determines that \( m.A \) reads \( m.D \).

In effect, a module may read another module through multiple levels of dependence, in order to support arbitrary amounts of refactoring. Once a module is released for someone to reuse (via requires), the module's author has committed to its name and API but is free to refactor its content into other modules which the original module reuses (via requires transitive) for the benefit of consumers. In the example above, package \( p \) may have been exported originally by \( m.B \) (thus, \( m.A \) requires \( m.B \)) but refactoring has caused some of \( m.B \)'s content to move into \( m.C \) and \( m.D \). By using a chain of requires transitive directives, the family of \( m.B, m.C, \) and \( m.D \) can preserve access to package \( p \) for code in \( m.A \) without forcing any changes to the requires directives of \( m.A \). Note that package \( p \) in \( m.D \) is not "re-exported" by \( m.C \) and \( m.B \); rather, \( m.A \) is made to read \( m.D \) directly.

### 7.7.2 Exported and Opened Packages

The **exports** directive specifies the name of a package to be exported by the current module. For code in other modules, this grants access at compile time and run time to the public and protected types in the package, and the public and protected members of those types (§6.6). It also grants reflective access to those types and members for code in other modules.

The **opens** directive specifies the name of a package to be opened by the current module. For code in other modules, this grants access at run time, but not compile time, to the public and protected types in the package, and the public and protected members of those types. It also grants reflective access to all types in the package, and all their members, for code in other modules.

It is a compile-time error if the package specified by exports is not declared by a compilation unit associated with the current module (§7.3).

It is permitted for opens to specify a package which is not declared by a compilation unit associated with the current module. (If the package should happen to be declared by an observable compilation unit associated with another module, the opens directive has no effect on that other module.)

It is a compile-time error if more than one exports directive in a module declaration specifies the same package name.

It is a compile-time error if more than one opens directive in a module declaration specifies the same package name.

It is a compile-time error if an opens directive appears in the declaration of an open module.
If an `exports` or `opens` directive has a `to` clause, then the directive is *qualified*; otherwise, it is *unqualified*. For a qualified directive, the public and protected types in the package, and their public and protected members, are accessible solely to code in the modules specified in the `to` clause. The modules specified in the `to` clause are referred to as *friends* of the current module. For an unqualified directive, these types and their members are accessible to code in any module.

It is permitted for the `to` clause of an `exports` or `opens` directive to specify a module which is not observable (§7.7.6).

It is a compile-time error if the `to` clause of a given `exports` directive specifies the same module name more than once.

It is a compile-time error if the `to` clause of a given `opens` directive specifies the same module name more than once.

### 7.7.3 Service Consumption

The `uses` directive specifies a *service* for which the current module may discover providers via `java.util.ServiceLoader`.

The service must be a class type, an interface type, or an annotation type. It is a compile-time error if a `uses` directive specifies an enum type (§8.9) as the service.

The service may be declared in the current module or in another module. If the service is not declared in the current module, then the service must be accessible to code in the current module (§6.6), or a compile-time error occurs.

It is a compile-time error if more than one `uses` directive in a module declaration specifies the same service.

### 7.7.4 Service Provision

The `provides` directive specifies a service for which the `with` clause specifies one or more *service providers* to `java.util.ServiceLoader`.

The service must be a class type, an interface type, or an annotation type. It is a compile-time error if a `provides` directive specifies an enum type (§8.9) as the service.

The service may be declared in the current module or in another module. If the service is not declared in the current module, then the service must be accessible to code in the current module (§6.6), or a compile-time error occurs.
Every service provider must be a class type or an interface type, that is public, and that is top level or nested static, or a compile-time error occurs.

Every service provider must be declared in the current module, or a compile-time error occurs.

If a service provider explicitly declares a public constructor with no formal parameters, or implicitly declares a public default constructor (§8.8.9), then that constructor is called the provider constructor.

If a service provider explicitly declares a public static method called provider with no formal parameters, then that method is called the provider method.

If a service provider has a provider method, then its return type must (i) either be declared in the current module, or be declared in another module and be accessible to code in the current module; and (ii) be a subtype of the service specified in the provides directive; or a compile-time error occurs.

While a service provider that is specified by a provides directive must be declared in the current module, its provider method may have a return type that is declared in another module. Also, note that when a service provider declares a provider method, the service provider itself need not be a subtype of the service.

If a service provider does not have a provider method, then that service provider must have a provider constructor and must be a subtype of the service specified in the provides directive, or a compile-time error occurs.

It is a compile-time error if more than one provides directive in a module declaration specifies the same service.

It is a compile-time error if the with clause of a given provides directive specifies the same service provider more than once.

### 7.7.5 Unnamed Modules

An observable ordinary compilation unit that the host system does not associate with a named module (§7.3) is associated with an unnamed module.

Unnamed modules are provided by the Java SE Platform in recognition of the fact that programs developed prior to Java SE 9 could not declare named modules. In addition, the reasons for the Java SE Platform providing unnamed packages (§7.4.2) are largely applicable to unnamed modules.

An implementation of the Java SE Platform must support at least one unnamed module. An implementation may support more than one unnamed module, but is
not required to do so. Which ordinary compilation units are associated with each
unnamed module is determined by the host system.

The host system may associate ordinary compilation units in a named package with
an unnamed module.

The rules for unnamed modules are designed to maximize their interoperation with
named modules, as follows:

- An unnamed module reads every observable module (§7.7.6).

  By virtue of the fact that an ordinary compilation unit associated with an unnamed
  module is observable, the associated unnamed module is observable. Thus, if the
  implementation of the Java SE Platform supports more than one unnamed module, every
  unnamed module is observable; and each unnamed module reads every unnamed module
  including itself.

  However, it is important to realize that the ordinary compilation units of an unnamed
  module are never visible to a named module (§7.3) because no requires directive can
  arrange for a named module to read an unnamed module. The Core Reflection API of
  the Java SE Platform may be used to arrange for a named module to read an unnamed
  module at run time.

- An unnamed module exports every package whose ordinary compilation units
  are associated with that unnamed module.

- An unnamed module opens every package whose ordinary compilation units are
  associated with that unnamed module.

### 7.7.6 Observability of a Module

A module is observable if at least one of the following is true:

- A modular compilation unit containing the declaration of the module is
  observable (§7.3).

- An ordinary compilation unit associated with the module is observable.
C L A S S declarations define new reference types and describe how they are implemented (§8.1).

A top level class is a class that is not a nested class.

A nested class is any class whose declaration occurs within the body of another class or interface.

This chapter discusses the common semantics of all classes - top level (§7.6) and nested (including member classes (§8.5, §9.5), local classes (§14.3) and anonymous classes (§15.9.5)). Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A named class may be declared abstract (§8.1.1.1) and must be declared abstract if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. A class may be declared final (§8.1.1.2), in which case it cannot have subclasses. If a class is declared public, then it can be referred to from code in any package of its module and potentially from code in other modules. Each class except Object is an extension of (that is, a subclass of) a single existing class (§8.1.4) and may implement interfaces (§8.1.5). Classes may be generic (§8.1.2), that is, they may declare type variables whose bindings may differ among different instances of the class.

Classes may be decorated with annotations (§9.7) just like any other kind of declaration.

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors (§8.1.6). The scope (§6.3) of a member (§8.2) is the entire body of the declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers (§6.6) public, protected, or private. The members of a class include both declared and inherited members (§8.2). Newly declared fields can hide fields declared in a
superclass or superinterface. Newly declared class members and interface members can hide class or interface members declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

Field declarations (§8.3) describe class variables, which are incarnated once, and instance variables, which are freshly incarnated for each instance of the class. A field may be declared final (§8.3.1.2), in which case it can be assigned to only once. Any field declaration may include an initializer.

Member class declarations (§8.5) describe nested classes that are members of the surrounding class. Member classes may be static, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes (§8.1.3).

Member interface declarations (§8.5) describe nested interfaces that are members of the surrounding class.

Method declarations (§8.4) describe code that may be invoked by method invocation expressions (§15.12). A class method is invoked relative to the class type; an instance method is invoked with respect to some particular object that is an instance of a class type. A method whose declaration does not indicate how it is implemented must be declared abstract. A method may be declared final (§8.4.3.3), in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent native code (§8.4.3.4). A synchronized method (§8.4.3.6) automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a synchronized statement (§14.19), thus allowing its activities to be synchronized with those of other threads (§17 (Threads and Locks)).

Method names may be overloaded (§8.4.9).

Instance initializers (§8.6) are blocks of executable code that may be used to help initialize an instance when it is created (§15.9).

Static initializers (§8.7) are blocks of executable code that may be used to help initialize a class.

Constructors (§8.8) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded (§8.8.8).
8.1 Class Declarations

A class declaration specifies a new named reference type.

There are two kinds of class declarations: normal class declarations and enum declarations.

\[
\text{ClassDeclaration:} \\
\text{NormalClassDeclaration} \\
\text{EnumDeclaration}
\]

\[
\text{NormalClassDeclaration:} \\
\{\text{ClassModifier}\} \text{ class TypeIdentifier [TypeParameters]} \\
\text{[Superclass] [Superinterfaces] ClassBody}
\]

The rules in this section apply to all class declarations, including enum declarations. However, special rules apply to enum declarations with regard to class modifiers, inner classes, and superclasses; these rules are stated in §8.9.

The \text{TypeIdentifier} in a class declaration specifies the name of the class.

It is a compile-time error if a class has the same simple name as any of its enclosing classes or interfaces.

The scope and shadowing of a class declaration is specified in §6.3 and §6.4.

8.1.1 Class Modifiers

A class declaration may include class modifiers.

\[
\text{ClassModifier:} \\
\text{(one of)} \\
\text{Annotation public protected private} \\
\text{abstract static final strictfp}
\]

The rules for annotation modifiers on a class declaration are specified in §9.7.4 and §9.7.5.

The access modifier public (§6.6) pertains only to top level classes (§7.6) and member classes (§8.5), not to local classes (§14.3) or anonymous classes (§15.9.5).

The access modifiers protected and private pertain only to member classes within a directly enclosing class declaration (§8.5).
The modifier \texttt{static} pertains only to member classes (§8.5.1), not to top level or local or anonymous classes.

It is a compile-time error if the same keyword appears more than once as a modifier for a class declaration, or if a class declaration has more than one of the access modifiers \texttt{public}, \texttt{protected}, and \texttt{private} (§6.6).

If two or more (distinct) class modifiers appear in a class declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for \texttt{ClassModifier}.

\textbf{8.1.1.1 abstract Classes}

An \texttt{abstract} class is a class that is incomplete, or to be considered incomplete.

It is a compile-time error if an attempt is made to create an instance of an \texttt{abstract} class using a class instance creation expression (§15.9.1).

A subclass of an \texttt{abstract} class that is not itself \texttt{abstract} may be instantiated, resulting in the execution of a constructor for the \texttt{abstract} class and, therefore, the execution of the field initializers for instance variables of that class.

A normal class may have \texttt{abstract} methods, that is, methods that are declared but not yet implemented (§8.4.3.1), only if it is an \texttt{abstract} class. It is a compile-time error if a normal class that is not \texttt{abstract} has an \texttt{abstract} method.

A class \texttt{C} has \texttt{abstract} methods if either of the following is true:

- Any of the member methods (§8.2) of \texttt{C} - either declared or inherited - is \texttt{abstract}.
- Any of \texttt{C}'s superclasses has an \texttt{abstract} method declared with package access, and there exists no method that overrides the \texttt{abstract} method from \texttt{C} or from a superclass of \texttt{C}.

It is a compile-time error to declare an \texttt{abstract} class type such that it is not possible to create a subclass that implements all of its \texttt{abstract} methods. This situation can occur if the class would have as members two \texttt{abstract} methods that have the same method signature (§8.4.2) but return types for which no type is return-type-substitutable with both (§8.4.5).

\textbf{Example 8.1.1.1-1. Abstract Class Declaration}

```java
abstract class Point {
    int x = 1, y = 1;
    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```
Here, a class `Point` is declared that must be declared abstract, because it contains a declaration of an abstract method named `alert`. The subclass of `Point` named `ColoredPoint` inherits the abstract method `alert`, so it must also be declared abstract. On the other hand, the subclass of `Point` named `SimplePoint` provides an implementation of `alert`, so it need not be abstract.

The statement:

```java
Point p = new Point();
```

would result in a compile-time error; the class `Point` cannot be instantiated because it is abstract. However, a `Point` variable could correctly be initialized with a reference to any subclass of `Point`, and the class `SimplePoint` is not abstract, so the statement:

```java
Point p = new SimplePoint();
```

would be correct. Instantiation of a `SimplePoint` causes the default constructor and field initializers for `x` and `y` of `Point` to be executed.

**Example 8.1.1.1-2. Abstract Class Declaration that Prohibits Subclasses**

```java
interface Colorable {
    void setColor(int color);
}
abstract class Colored implements Colorable {
    public abstract int setColor(int color);
}
```

These declarations result in a compile-time error: it would be impossible for any subclass of class `Colored` to provide an implementation of a method named `setColor`, taking one argument of type `int`, that can satisfy both abstract method specifications, because the one in interface `Colorable` requires the same method to return no value, while the one in class `Colored` requires the same method to return a value of type `int` (§8.4).

A class type should be declared abstract only if the intent is that subclasses can be created to complete the implementation. If the intent is simply to prevent instantiation of a class, the proper way to express this is to declare a constructor (§8.8.10) of no arguments, make it private, never invoke it, and declare no other constructors. A class of this form usually contains class methods and variables.
The class `Math` is an example of a class that cannot be instantiated; its declaration looks like this:

```java
public final class Math {
    private Math() { } // never instantiate this class
    . . . declarations of class variables and methods . . .
}
```

### 8.1.1.2 final Classes

A class can be declared `final` if its definition is complete and no subclasses are desired or required.

It is a compile-time error if the name of a `final` class appears in the `extends` clause (§8.1.4) of another class declaration; this implies that a `final` class cannot have any subclasses.

It is a compile-time error if a class is declared both `final` and `abstract`, because the implementation of such a class could never be completed (§8.1.1.1).

Because a `final` class never has any subclasses, the methods of a `final` class are never overridden (§8.4.8.1).

### 8.1.1.3 strictfp Classes

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the class declaration (including within variable initializers, instance initializers, static initializers, and constructors) be explicitly FP-strict (§15.4).

This implies that all methods declared in the class, and all nested types declared in the class, are implicitly `strictfp`.

### 8.1.2 Generic Classes and Type Parameters

A class is `generic` if it declares one or more type variables (§4.4).

These type variables are known as the `type parameters` of the class. The type parameter section follows the class name and is delimited by angle brackets.

```
TypeParameters:
   < TypeParameterList >

TypeParameterList:
   TypeParameter { , TypeParameter}
```

The following productions from §4.4 are shown here for convenience:
**TypeParameter:**

\{TypeParameterModifier\} TypeIdentifier \[TypeBound\]

**TypeParameterModifier:**

Annotation

**TypeBound:**

- extends TypeVariable
- extends ClassOrInterfaceType \{AdditionalBound\}

**AdditionalBound:**

& InterfaceType

The rules for annotation modifiers on a type parameter declaration are specified in §9.7.4 and §9.7.5.

In a class's type parameter section, a type variable \( \tau \) directly depends on a type variable \( \sigma \) if \( \sigma \) is the bound of \( \tau \), while \( \tau \) depends on \( \sigma \) if either \( \tau \) directly depends on \( \sigma \) or \( \tau \) directly depends on a type variable \( \upsilon \) that depends on \( \sigma \) (using this definition recursively). It is a compile-time error if a type variable in a class's type parameter section depends on itself.

The scope and shadowing of a class's type parameter is specified in §6.3 and §6.4.

A generic class declaration defines a set of parameterized types (§4.5), one for each possible parameterization of the type parameter section by type arguments. All of these parameterized types share the same class at run time.

For instance, executing the code:

```java
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

will result in the variable \( b \) holding the value \( \text{true} \).

It is a compile-time error if a generic class is a direct or indirect subclass of Throwable (§11.1.1).

This restriction is needed since the catch mechanism of the Java Virtual Machine works only with non-generic classes.

It is a compile-time error to refer to a type parameter of a generic class \( C \) in any of the following:

- the declaration of a static member of \( C \) (§8.3.1.1, §8.4.3.2, §8.5.1).
- the declaration of a static member of any type declaration nested within \( C \).
• a static initializer of \( C \) (§8.7), or
• a static initializer of any class declaration nested within \( C \).

Example 8.1.2-1. Mutually Recursive Type Variable Bounds

```java
interface ConvertibleTo<T> {
    T convert();
}
class ReprChange<T extends ConvertibleTo<S>,
    S extends ConvertibleTo<T>> {
    T t;
    void set(S s) { t = s.convert(); }
    S get() { return t.convert(); }
}
```

Example 8.1.2-2. Nested Generic Classes

```java
class Seq<T> {
    T head;
    Seq<T> tail;

    Seq() { this(null, null); }
    Seq(T head, Seq<T> tail) {
        this.head = head;
        this.tail = tail;
    }
    boolean isEmpty() { return tail == null; }

class Zipper<S> {
    Seq<Pair<T,S>> zip(Seq<S> that) {
        if (isEmpty() || that.isEmpty()) {
            return new Seq<Pair<T,S>>();
        } else {
            Seq<T>.Zipper<S> tailZipper =
                tail.new Zipper<S>();
            return new Seq<Pair<T,S>>(
                new Pair<T,S>(head, that.head),
                tailZipper.zip(that.tail));
        }
    }
}
class Pair<T, S> {
    T fst; S snd;
    Pair(T f, S s) { fst = f; snd = s; }
}
class Test {
    public static void main(String[] args) {
        Seq<String> strs =
            new Seq<String>(

"a",
new Seq<String>("b",
    new Seq<String>()));
Seq<Number> nums =
new Seq<Number>(
    new Integer(1),
    new Seq<Number>(new Double(1.5),
        new Seq<Number>()));

Seq<String>.Zipper<Number> zipper =
strs.new Zipper<Number>();
Seq<Pair<String,Number>> combined =
    zipper.zip(nums);
}
}

8.1.3 Inner Classes and Enclosing Instances

An inner class is a nested class that is not explicitly or implicitly declared static.
An inner class may be a non-static member class (§8.5), a local class (§14.3), or an anonymous class (§15.9.5). A member class of an interface is implicitly static (§9.5) so is never considered to be an inner class.

It is a compile-time error if an inner class declares a static initializer (§8.7).

It is a compile-time error if an inner class declares a member that is explicitly or implicitly static, unless the member is a constant variable (§4.12.4).

An inner class may inherit static members that are not constant variables even though it cannot declare them.

A nested class that is not an inner class may declare static members freely, in accordance with the usual rules of the Java programming language.

Example 8.1.3-1. Inner Class Declarations and Static Members

class HasStatic {
    static int j = 100;
}
class Outer {
    class Inner extends HasStatic {
        static final int x = 3;  // OK: constant variable
        static int y = 4;  // Compile-time error: an inner class
    }
    static class NestedButNotInner{
        static int z = 5;  // OK: not an inner class
    }
    interface NeverInner {}  // Interfaces are never inner
A statement or expression occurs in a static context if and only if the innermost method, constructor, instance initializer, static initializer, field initializer, or explicit constructor invocation statement enclosing the statement or expression is a static method, a static initializer, the variable initializer of a static variable, or an explicit constructor invocation statement (§8.8.7.1).

An inner class c is a direct inner class of a class or interface o if o is the immediately enclosing type declaration of c and the declaration of c does not occur in a static context.

A class c is an inner class of class or interface o if it is either a direct inner class of o or an inner class of an inner class of o.

It is unusual, but possible, for the immediately enclosing type declaration of an inner class to be an interface. This only occurs if the class is declared in a default method body (§9.4). Specifically, it occurs if an anonymous or local class is declared in a default method body, or a member class is declared in the body of an anonymous class that is declared in a default method body.

A class or interface o is the zeroth lexically enclosing type declaration of itself.

A class o is the n'th lexically enclosing type declaration of a class c if it is the immediately enclosing type declaration of the n-1'th lexically enclosing type declaration of c.

An instance i of a direct inner class c of a class or interface o is associated with an instance of o, known as the immediately enclosing instance of i. The immediately enclosing instance of an object, if any, is determined when the object is created (§15.9.2).

An object o is the zeroth lexically enclosing instance of itself.

An object o is the n'th lexically enclosing instance of an instance i if it is the immediately enclosing instance of the n-1'th lexically enclosing instance of i.

An instance of an inner class i whose declaration occurs in a static context has no lexically enclosing instances. However, if i is immediately declared within a static method or static initializer then i does have an enclosing block, which is the innermost block statement lexically enclosing the declaration of i.

For every superclass s of c which is itself a direct inner class of a class or interface so, there is an instance of so associated with i, known as the immediately enclosing instance of i with respect to s. The immediately enclosing instance of an object with respect to its class' direct superclass, if any, is determined when the superclass constructor is invoked via an explicit constructor invocation statement (§8.8.7.1).
When an inner class (whose declaration does not occur in a static context) refers to an instance variable that is a member of a lexically enclosing type declaration, the variable of the corresponding lexically enclosing instance is used.

Any local variable, formal parameter, or exception parameter used but not declared in an inner class must either be declared `final` or be effectively final (§4.12.4), or a compile-time error occurs where the use is attempted.

Any local variable used but not declared in an inner class must be definitely assigned (§16 (Definite Assignment)) before the body of the inner class, or a compile-time error occurs.

Similar rules on variable use apply in the body of a lambda expression (§15.27.2).

A blank `final` field (§4.12.4) of a lexically enclosing type declaration may not be assigned within an inner class, or a compile-time error occurs.

**Example 8.1.3-2. Inner Class Declarations**

```java
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i;  // Compile-time error
            int m = l;  // OK
        }
    }
    void foo() {
        class Local {  // A local class
            int j = i;
        }
    }
}
```

The declaration of class `LocalInStaticContext` occurs in a static context due to being within the static method `classMethod`. Instance variables of class `Outer` are not available within the body of a static method. In particular, instance variables of `Outer` are not available inside the body of `LocalInStaticContext`. However, local variables from the surrounding method may be referred to without error (provided they are declared `final` or are effectively final).

Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing type declaration. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing type declaration, the instance variable must be defined with respect to an enclosing instance of that declared type. For example, the class `Local` above has an enclosing instance of class `Outer`. As a further example:

```java
class WithDeepNesting {
```
boolean toBe;
WithDeepNesting(boolean b) { toBe = b; }

class Nested {
    boolean theQuestion;
    class DeeplyNested {
        DeeplyNested(){
            theQuestion = toBe || !toBe;
        }
    }
}

Here, every instance of WithDeepNesting.Nested.DeeplyNested has an enclosing instance of class WithDeepNesting.Nested (its immediately enclosing instance) and an enclosing instance of class WithDeepNesting (its 2nd lexically enclosing instance).

8.1.4 Superclasses and Subclasses

The optional extends clause in a normal class declaration specifies the direct superclass of the current class.

    Superclass:
        extends ClassType

The extends clause must not appear in the definition of the class Object, or a compile-time error occurs, because it is the primordial class and has no direct superclass.

The ClassType must name an accessible class type (§6.6), or a compile-time error occurs.

It is a compile-time error if the ClassType names a class that is final, because final classes are not allowed to have subclasses (§8.1.1.2).

It is a compile-time error if the ClassType names the class Enum or any invocation of Enum (§8.9).

If the ClassType has type arguments, it must denote a well-formed parameterized type (§4.5), and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

Given a (possibly generic) class declaration $C_{<F_1,...,F_n>}$ ($n \geq 0$, $C \neq Object$), the direct superclass of the class type $C_{<F_1,...,F_n>}$ is the type given in the extends clause of the declaration of $C$ if an extends clause is present, or Object otherwise.

Given a generic class declaration $C_{<F_1,...,F_n>}$ ($n > 0$), the direct superclass of the parameterized class type $C_{<T_1,...,T_n>}$, where $T_i$ ($1 \leq i \leq n$) is a type, is $D_{<U_1 \theta,...,U_k}$
CLASSES Class Declarations

θ>, where $D < U_1, \ldots, U_k >$ is the direct superclass of $C < F_1, \ldots, F_n >$ and θ is the substitution $[F_1 := T_1, \ldots, F_n := T_n]$.

A class is said to be a direct subclass of its direct superclass. The direct superclass is the class from whose implementation the implementation of the current class is derived.

The subclass relationship is the transitive closure of the direct subclass relationship. A class $A$ is a subclass of class $C$ if either of the following is true:

- $A$ is the direct subclass of $C$
- There exists a class $B$ such that $A$ is a subclass of $B$, and $B$ is a subclass of $C$, applying this definition recursively.

Class $C$ is said to be a superclass of class $A$ whenever $A$ is a subclass of $C$.

Example 8.1.4-1. Direct Superclasses and Subclasses

```java
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; }  // error
```

Here, the relationships are as follows:

- The class `Point` is a direct subclass of `Object`.
- The class `Object` is the direct superclass of the class `Point`.
- The class `ColoredPoint` is a direct subclass of class `Point`.
- The class `Point` is the direct superclass of class `ColoredPoint`.

The declaration of class `Colored3DPoint` causes a compile-time error because it attempts to extend the final class `ColoredPoint`.

Example 8.1.4-2. Superclasses and Subclasses

```java
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3DPoint extends ColoredPoint { int z; }
```

Here, the relationships are as follows:

- The class `Point` is a superclass of class `ColoredPoint`.
- The class `Point` is a superclass of class `Colored3DPoint`.
- The class `ColoredPoint` is a subclass of class `Point`.
- The class `ColoredPoint` is a superclass of class `Colored3DPoint`.
- The class `Colored3DPoint` is a subclass of class `ColoredPoint`.
8.1 Class Declarations

- The class Colored3dPoint is a subclass of class Point.

A class \( C \) directly depends on a type \( T \) if \( T \) is mentioned in the extends or implements clause of \( C \) either as a superclass or superinterface, or as a qualifier in the fully qualified form of a superclass or superinterface name.

A class \( C \) depends on a reference type \( T \) if any of the following is true:

- \( C \) directly depends on \( T \).
- \( C \) directly depends on an interface \( I \) that depends (§9.1.3) on \( T \).
- \( C \) directly depends on a class \( D \) that depends on \( T \) (using this definition recursively).

It is a compile-time error if a class depends on itself.

If circularly declared classes are detected at run time, as classes are loaded, then a ClassCircularityError is thrown (§12.2.1).

**Example 8.1.4-3. Class Depends on Itself**

class Point extends ColoredPoint { int x, y; }
class ColoredPoint extends Point { int color; }

This program causes a compile-time error because class Point depends on itself.

8.1.5 Superinterfaces

The optional implements clause in a class declaration lists the names of interfaces that are direct superinterfaces of the class being declared.

**Superinterfaces:**

\[
\text{implements } \text{InterfaceTypeList}
\]

**InterfaceTypeList:**

\[
\text{InterfaceType } \{, \ \text{InterfaceType}\}
\]

Each InterfaceType must name an accessible interface type (§6.6), or a compile-time error occurs.

If an InterfaceType has type arguments, it must denote a well-formed parameterized type (§4.5), and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.
It is a compile-time error if the same interface is mentioned as a direct superinterface more than once in a single implements clause. This is true even if the interface is named in different ways.

Example 8.1.5-1. Illegal Superinterfaces

```java
class Redundant implements java.lang.Cloneable, Cloneable {
    int x;
}
```

This program results in a compile-time error because the names `java.lang.Cloneable` and `Cloneable` refer to the same interface.

Given a (possibly generic) class declaration $C^{<F_1,...,F_n>}$ ($n \geq 0$, $C \neq Object$), the direct superinterfaces of the class type $C^{<F_1,...,F_n>}$ are the types given in the implements clause of the declaration of $C$, if an implements clause is present.

Given a generic class declaration $C^{<F_1,...,F_n>}$ ($n > 0$), the direct superinterfaces of the parameterized class type $C^{<T_1,...,T_n>}$, where $T_i$ ($1 \leq i \leq n$) is a type, are all types $I^{<U_1 \theta,...,U_k \theta>}$, where $I^{<U_1,...,U_k>}$ is a direct superinterface of $C^{<F_1,...,F_n>}$ and $\theta$ is the substitution $[F_1:=T_1, \ldots, F_n:=T_n]$.

An interface type $I$ is a superinterface of class type $C$ if any of the following is true:

- $I$ is a direct superinterface of $C$.
- $C$ has some direct superinterface $J$ for which $I$ is a superinterface, using the definition of "superinterface of an interface" given in §9.1.3.
- $I$ is a superinterface of the direct superclass of $C$.

A class can have a superinterface in more than one way.

A class is said to implement all its superinterfaces.

A class may not at the same time be a subtype of two interface types which are different parameterizations of the same generic interface (§9.1.2), or a subtype of a parameterization of a generic interface and a raw type naming that same generic interface, or a compile-time error occurs.

This requirement was introduced in order to support translation by type erasure (§4.6).

Example 8.1.5-2. Superinterfaces

```java
interface Colorable {
    void setColor(int color);
    int getColor();
}
enum Finish { MATTE, GLOSSY }
```
interface Paintable extends Colorable {
    void setFinish(Finish finish);
    Finish getFinish();
}

class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
    public int getColor() { return color; }
}
class PaintedPoint extends ColoredPoint implements Paintable {
    Finish finish;
    public void setFinish(Finish finish) {
        this.finish = finish;
    }
    public Finish getFinish() { return finish; }
}

Here, the relationships are as follows:

- The interface Paintable is a superinterface of class PaintedPoint.
- The interface Colorable is a superinterface of class ColoredPoint and of class PaintedPoint.
- The interface Paintable is a subinterface of the interface Colorable, and Colorable is a superinterface of Paintable, as defined in §9.1.3.

The class PaintedPoint has Colorable as a superinterface both because it is a superinterface of ColoredPoint and because it is a superinterface of Paintable.

Example 8.1.5-3. Illegal Multiple Inheritance of an Interface

interface I<T> {}
class B implements I<Integer> {}
class C extends B implements I<String> {}

Class C causes a compile-time error because it attempts to be a subtype of both I<Integer> and I<String>.

Unless the class being declared is abstract, all the abstract member methods of each direct superinterface must be implemented (§8.4.8.1) either by a declaration in this class or by an existing method declaration inherited from the direct superclass or a direct superinterface, because a class that is not abstract is not permitted to have abstract methods (§8.1.1.1).

Each default method (§9.4.3) of a superinterface of the class may optionally be overridden by a method in the class; if not, the default method is typically inherited and its behavior is as specified by its default body.
It is permitted for a single method declaration in a class to implement methods of more than one superinterface.

**Example 8.1.5-3. Implementing Methods of a Superinterface**

```java
interface Colorable {
    void setColor(int color);
    int getColor();
}

class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
    int color;
}
```

This program causes a compile-time error, because `ColoredPoint` is not an abstract class but fails to provide an implementation of methods `setColor` and `getColor` of the interface `Colorable`.

In the following program:

```java
interface Fish { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    public int getNumberOfScales() { return 91; }
}
```

the method `getNumberOfScales` in class `Tuna` has a name, signature, and return type that matches the method declared in interface `Fish` and also matches the method declared in interface `Piano`; it is considered to implement both.

On the other hand, in a situation such as this:

```java
interface Fish { int getNumberOfScales(); }
interface StringBass { double getNumberOfScales(); }
class Bass implements Fish, StringBass {
    // This declaration cannot be correct,
    // no matter what type is used.
    public ?? getNumberOfScales() { return 91; }
}
```

it is impossible to declare a method named `getNumberOfScales` whose signature and return type are compatible with those of both the methods declared in interface `Fish` and in interface `StringBass`, because a class cannot have multiple methods with the same signature and different primitive return types (§8.4). Therefore, it is impossible for a single class to implement both interface `Fish` and interface `StringBass` (§8.4.8).
8.1.6 Class Body and Member Declarations

A class body may contain declarations of members of the class, that is, fields (§8.3), methods (§8.4), classes (§8.5), and interfaces (§8.5).

A class body may also contain instance initializers (§8.6), static initializers (§8.7), and declarations of constructors (§8.8) for the class.

ClassBody:

\{
  \{ClassBodyDeclaration\}
\}

ClassBodyDeclaration:

ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:

FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

The scope and shadowing of a declaration of a member $m$ declared in or inherited by a class type $C$ is specified in §6.3 and §6.4.

If $C$ itself is a nested class, there may be definitions of the same kind (variable, method, or type) and name as $m$ in enclosing scopes. (The scopes may be blocks, classes, or packages.) In all such cases, the member $m$ declared in or inherited by $C$ shadows (§6.4.1) the other definitions of the same kind and name.

8.2 Class Members

The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.4), except in class Object, which has no direct superclass
- Members inherited from any direct superinterfaces (§8.1.5)
- Members declared in the body of the class (§8.1.6)
Members of a class that are declared *private* are not inherited by subclasses of that class.

Only members of a class that are declared *protected* or *public* are inherited by subclasses declared in a package other than the one in which the class is declared.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

We use the phrase *the type of a member* to denote:

- For a field, its type.
- For a method, an ordered 4-tuple consisting of:
  - type parameters: the declarations of any type parameters of the method member.
  - argument types: a list of the types of the arguments to the method member.
  - return type: the return type of the method member.
  - *throws* clause: exception types declared in the *throws* clause of the method member.

Fields, methods, and member types of a class type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5). However, this is discouraged as a matter of style.

**Example 8.2-1. Use of Class Members**

```java
class Point {
    int x, y;
    private Point() { reset(); }
    Point(int x, int y) { this.x = x; this.y = y; }
    private void reset() { this.x = 0; this.y = 0; }
}
class ColoredPoint extends Point {
    int color;
    void clear() { reset(); }  // error
}
class Test {
    public static void main(String[] args) {
        ColoredPoint c = new ColoredPoint(0, 0);  // error
        c.reset();  // error
    }
}
```

This program causes four compile-time errors.
One error occurs because `ColoredPoint` has no constructor declared with two `int` parameters, as requested by the use in `main`. This illustrates the fact that `ColoredPoint` does not inherit the constructors of its superclass `Point`.

Another error occurs because `ColoredPoint` declares no constructors, and therefore a default constructor for it is implicitly declared (§8.8.9), and this default constructor is equivalent to:

```java
ColoredPoint() { super(); }
```

which invokes the constructor, with no arguments, for the direct superclass of the class `ColoredPoint`. The error is that the constructor for `Point` that takes no arguments is `private`, and therefore is not accessible outside the class `Point`, even through a superclass constructor invocation (§8.8.7).

Two more errors occur because the method `reset` of class `Point` is `private`, and therefore is not inherited by class `ColoredPoint`. The method invocations in method `clear` of class `ColoredPoint` and in method `main` of class `Test` are therefore not correct.

**Example 8.2-2. Inheritance of Class Members with Package Access**

Consider the example where the `points` package declares two compilation units:

```java
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
```

and:

```java
package points;
public class Point3d extends Point {
    int z;
    public void move(int dx, int dy, int dz) {
        x += dx; y += dy; z += dz;
    }
}
```

and a third compilation unit, in another package, is:

```java
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        x += dx; y += dy; z += dz; w += dw; // compile-time errors
    }
}
```
Here both classes in the \texttt{points} package compile. The class \texttt{Point3d} inherits the fields \texttt{x} and \texttt{y} of class \texttt{Point}, because it is in the same package as \texttt{Point}. The class \texttt{Point4d}, which is in a different package, does not inherit the fields \texttt{x} and \texttt{y} of class \texttt{Point} or the field \texttt{z} of class \texttt{Point3d}, and so fails to compile.

A better way to write the third compilation unit would be:

\begin{verbatim}
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
\end{verbatim}

using the \texttt{move} method of the superclass \texttt{Point3d} to process \texttt{dx}, \texttt{dy}, and \texttt{dz}. If \texttt{Point4d} is written in this way, it will compile without errors.

\textbf{Example 8.2-3. Inheritance of public and protected Class Members}

Given the class \texttt{Point}:

\begin{verbatim}
package points;
public class Point {
    public int x, y;
    protected int useCount = 0;
    static protected int totalUseCount = 0;
    public void move(int dx, int dy) {
        x += dx; y += dy; useCount++; totalUseCount++;
    }
}
\end{verbatim}

the \texttt{public} and \texttt{protected} fields \texttt{x}, \texttt{y}, \texttt{useCount}, and \texttt{totalUseCount} are inherited in all subclasses of \texttt{Point}.

Therefore, this test program, in another package, can be compiled successfully:

\begin{verbatim}
class Test extends points.Point {
    public void moveBack(int dx, int dy) {
        x -= dx; y -= dy; useCount++; totalUseCount++;
    }
}
\end{verbatim}

\textbf{Example 8.2-4. Inheritance of private Class Members}

\begin{verbatim}
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
\end{verbatim}
8.2 Class Members

```java
void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz; totalMoves++; // error
    }
}
```

Here, the class variable `totalMoves` can be used only within the class `Point`; it is not inherited by the subclass `Point3d`. A compile-time error occurs because method `move` of class `Point3d` tries to increment `totalMoves`.

Example 8.2-5. Accessing Members of Inaccessible Classes

Even though a class might not be declared `public`, instances of the class might be available at run time to code outside the package in which it is declared by means of a `public` superclass or superinterface. An instance of the class can be assigned to a variable of such a `public` type. An invocation of a `public` method of the object referred to by such a variable may invoke a method of the class if it implements or overrides a method of the `public` superclass or superinterface. (In this situation, the method is necessarily declared `public`, even though it is declared in a class that is not `public`.)

Consider the compilation unit:

```java
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}
```

and another compilation unit of another package:

```java
package morePoints;
class Point3d extends points.Point {
    public int z;
    public void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz;
    }
    public void move(int dx, int dy) {
        move(dx, dy, 0);
    }
}
public class OnePoint {
    public static points.Point getOne() {
        return new Point3d();
    }
}
```
An invocation `morePoints.OnePoint.getOne()` in yet a third package would return a `Point3d` that can be used as a `Point`, even though the type `Point3d` is not available outside the package `morePoints`. The two-argument version of method `move` could then be invoked for that object, which is permissible because method `move` of `Point3d` is public (as it must be, for any method that overrides a public method must itself be public, precisely so that situations such as this will work out correctly). The fields `x` and `y` of that object could also be accessed from such a third package.

While the field `z` of class `Point3d` is public, it is not possible to access this field from code outside the package `morePoints`, given only a reference to an instance of class `Point3d` in a variable `p` of type `Point`. This is because the expression `p.z` is not correct, as `p` has type `Point` and class `Point` has no field named `z`; also, the expression `((Point3d)p).z` is not correct, because the class type `Point3d` cannot be referred to outside package `morePoints`.

The declaration of the field `z` as public is not useless, however. If there were to be, in package `morePoints`, a public subclass `Point4d` of the class `Point3d`:

```java
package morePoints;
public class Point4d extends Point3d {
    public int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
```

then class `Point4d` would inherit the field `z`, which, being public, could then be accessed by code in packages other than `morePoints`, through variables and expressions of the public type `Point4d`.

### 8.3 Field Declarations

The variables of a class type are introduced by `field declarations`. The declaration of a class type are introduced by *field declarations*.

**FieldDeclaration:**

```
{FieldModifier} UnannType VariableDeclaratorList ;
```

**VariableDeclaratorList:**

```
VariableDeclarator { , VariableDeclarator}
```

**VariableDeclarator:**

```
VariableDeclaratorId [= VariableInitializer]
```

**VariableDeclaratorId:**

```
Identifier [Dims]
```
VariableInitializer:
   Expression
   ArrayInitializer

UnannType:
   UnannPrimitiveType
   UnannReferenceType

UnannPrimitiveType:
   NumericType
   boolean

UnannReferenceType:
   UnannClassOrInterfaceType
   UnannTypeVariable
   UnannArrayType

UnannClassOrInterfaceType:
   UnannClassType
   UnannInterfaceType

UnannClassType:
   TypeIdentifier [TypeArguments]
   PackageName . {Annotation} TypeIdentifier [TypeArguments]
   UnannClassOrInterfaceType . {Annotation} TypeIdentifier
       [TypeArguments]

UnannInterfaceType:
   UnannClassType

UnannTypeVariable:
   TypeIdentifier

UnannArrayType:
   UnannPrimitiveType Dims
   UnannClassOrInterfaceType Dims
   UnannTypeVariable Dims

The following production from §4.3 is shown here for convenience:

Dims:
   {Annotation} [ ] {{Annotation} [ ]}
Each declarator in a FieldDeclaration declares one field. The Identifier in a declarator may be used in a name to refer to the field.

More than one field may be declared in a single FieldDeclaration by using more than one declarator; the FieldModifiers and UnannType apply to all the declarators in the declaration.

The FieldModifier clause is described in §8.3.1.

The declared type of a field is denoted by UnannType if no bracket pairs appear in UnannType and VariableDeclaratorId, and is specified by §10.2 otherwise.

The scope and shadowing of a field declaration is specified in §6.3 and §6.4.

It is a compile-time error for the body of a class declaration to declare two fields with the same name.

If a class declares a field with a certain name, then the declaration of that field is said to hide any and all accessible declarations of fields with the same name in superclasses, and superinterfaces of the class.

In this respect, hiding of fields differs from hiding of methods (§8.4.8.3), for there is no distinction drawn between static and non-static fields in field hiding whereas a distinction is drawn between static and non-static methods in method hiding.

A hidden field can be accessed by using a qualified name (§6.5.6.2) if it is static, or by using a field access expression that contains the keyword super (§15.11.2) or a cast to a superclass type.

In this respect, hiding of fields is similar to hiding of methods.

If a field declaration hides the declaration of another field, the two fields need not have the same type.

A class inherits from its direct superclass and direct superinterfaces all the non-private fields of the superclass and superinterfaces that are both accessible (§6.6) to code in the class and not hidden by a declaration in the class.

A private field of a superclass might be accessible to a subclass - for example, if both classes are members of the same class. Nevertheless, a private field is never inherited by a subclass.

It is possible for a class to inherit more than one field with the same name, either from its superclass and superinterfaces or from its superinterfaces alone. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such field by its simple name will result in a compile-time error, because the reference is ambiguous.
There might be several paths by which the same field declaration is inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

A value stored in a field of type float is always an element of the float value set (§4.2.3); similarly, a value stored in a field of type double is always an element of the double value set. It is not permitted for a field of type float to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a field of type double to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

**Example 8.3-1. Multiply Inherited Fields**

A class may inherit two or more fields with the same name, either from its superclass and a superinterface or from two superinterfaces. A compile-time error occurs on any attempt to refer to any ambiguously inherited field by its simple name. A qualified name or a field access expression that contains the keyword `super` (§15.11.2) may be used to access such fields unambiguously. In the program:

```java
interface Frob  { float v = 2.0f; }
class SuperTest { int   v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() { System.out.println(v); }
}
```

the class `Test` inherits two fields named `v`, one from its superclass `SuperTest` and one from its superinterface `Frob`. This in itself is permitted, but a compile-time error occurs because of the use of the simple name `v` in method `printV`: it cannot be determined which `v` is intended.

The following variation uses the field access expression `super.v` to refer to the field named `v` declared in class `SuperTest` and uses the qualified name `Frob.v` to refer to the field named `v` declared in interface `Frob`:

```java
interface Frob  { float v = 2.0f; }
class SuperTest { int   v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() {
        System.out.println((super.v + Frob.v)/2);
    }
}
```

It compiles and prints:
2.5

Even if two distinct inherited fields have the same type, the same value, and are both final, any reference to either field by simple name is considered ambiguous and results in a compile-time error. In the program:

```java
interface Color { int RED=0, GREEN=1, BLUE=2; }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN); // compile-time error
        System.out.println(RED);   // compile-time error
    }
}
```

it is not astonishing that the reference to GREEN should be considered ambiguous, because class Test inherits two different declarations for GREEN with different values. The point of this example is that the reference to RED is also considered ambiguous, because two distinct declarations are inherited. The fact that the two fields named RED happen to have the same type and the same unchanging value does not affect this judgment.

**Example 8.3-2. Re-inheritance of Fields**

If the same field declaration is inherited from an interface by multiple paths, the field is considered to be inherited only once. It may be referred to by its simple name without ambiguity. For example, in the code:

```java
interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {}  
class PaintedPoint extends ColoredPoint implements Paintable {
    int p = RED;
}
```

the fields RED, GREEN, and BLUE are inherited by the class PaintedPoint both through its direct superclass ColoredPoint and through its direct superinterface Paintable. The simple names RED, GREEN, and BLUE may nevertheless be used without ambiguity within the class PaintedPoint to refer to the fields declared in interface Colorable.
8.3 Field Declarations

8.3.1 Field Modifiers

FieldModifier:
(one of)
Annotation public protected private
static final transient volatile

The rules for annotation modifiers on a field declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration, or if a field declaration has more than one of the access modifiers public, protected, and private (§6.6).

If two or more (distinct) field modifiers appear in a field declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for FieldModifier.

8.3.1.1 static Fields

If a field is declared static, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized (§12.4).

A field that is not declared static (sometimes called a non-static field) is called an instance variable. Whenever a new instance of a class is created (§12.5), a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

Example 8.3.1.1-1. static Fields

```java
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
class Test {
    public static void main(String[] args) {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3;
        p.y = 3;
        p.useCount++;
        p.origin.useCount++;
        System.out.println(" + q.x + ", + q.y + ");
        System.out.println(q.useCount);
        System.out.println(q.origin == Point.origin);
    }
```
System.out.println(q.origin.useCount);
}
}

This program prints:

(2, 2)
0
true
1

showing that changing the fields x, y, and useCount of p does not affect the fields of q, because these fields are instance variables in distinct objects. In this example, the class variable origin of the class Point is referenced both using the class name as a qualifier, in Point.origin, and using variables of the class type in field access expressions ($15.11$), as in p.origin and q.origin. These two ways of accessing the origin class variable access the same object, evidenced by the fact that the value of the reference equality expression ($15.21.3$):

q.origin==Point.origin

is true. Further evidence is that the incrementation:

p.origin.useCount++;

causes the value of q.origin.useCount to be 1; this is so because p.origin and q.origin refer to the same variable.

Example 8.3.1-2. Hiding of Class Variables

class Point {
    static int x = 2;
}
class Test extends Point {
    static double x = 4.7;
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
}

This program produces the output:

4.7 2

because the declaration of x in class Test hides the definition of x in class Point, so class Test does not inherit the field x from its superclass Point. Within the declaration of class Test, the simple name x refers to the field declared within class Test. Code in class Test
may refer to the field \( x \) of class \( \text{Point} \) as \( \text{super.x} \) (or, because \( x \) is static, as \( \text{Point.x} \)). If the declaration of \( \text{Test.x} \) is deleted:

```java
class Point {
    static int x = 2;
}
class Test extends Point {
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
}
```

then the field \( x \) of class \( \text{Point} \) is no longer hidden within class \( \text{Test} \); instead, the simple name \( x \) now refers to the field \( \text{Point.x} \). Code in class \( \text{Test} \) may still refer to that same field as \( \text{super.x} \). Therefore, the output from this variant program is:

```
2 2
```

**Example 8.3.1-3. Hiding of Instance Variables**

```java
class Point {
    int x = 2;
}
class Test extends Point {
    double x = 4.7;
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

This program produces the output:

```
4.7 2
4.7 2
```

because the declaration of \( x \) in class \( \text{Test} \) hides the definition of \( x \) in class \( \text{Point} \), so class \( \text{Test} \) does not inherit the field \( x \) from its superclass \( \text{Point} \). It must be noted, however, that while the field \( x \) of class \( \text{Point} \) is not inherited by class \( \text{Test} \), it is nevertheless *implemented* by instances of class \( \text{Test} \). In other words, every instance of class \( \text{Test} \) contains two fields, one of type \( \text{int} \) and one of type \( \text{double} \). Both fields bear the name \( x \), but within the declaration of class \( \text{Test} \), the simple name \( x \) always refers to the field declared within class \( \text{Test} \). Code in instance methods of class \( \text{Test} \) may refer to the instance variable \( x \) of class \( \text{Point} \) as \( \text{super.x} \).
Code that uses a field access expression to access field \( x \) will access the field named \( x \) in the class indicated by the type of reference expression. Thus, the expression \( \text{sample}.x \) accesses a double value, the instance variable declared in class \( \text{Test} \), because the type of the variable \( \text{sample} \) is \( \text{Test} \), but the expression \( \text{((Point)sample)}.x \) accesses an int value, the instance variable declared in class \( \text{Point} \), because of the cast to type \( \text{Point} \).

If the declaration of \( x \) is deleted from class \( \text{Test} \), as in the program:

```java
class Point {
    static int x = 2;
}
class Test extends Point {
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

then the field \( x \) of class \( \text{Point} \) is no longer hidden within class \( \text{Test} \). Within instance methods in the declaration of class \( \text{Test} \), the simple name \( x \) now refers to the field declared within class \( \text{Point} \). Code in class \( \text{Test} \) may still refer to that same field as \( \text{super.x} \). The expression \( \text{sample}.x \) still refers to the field \( x \) within type \( \text{Test} \), but that field is now an inherited field, and so refers to the field \( x \) declared in class \( \text{Point} \). The output from this variant program is:

```
2 2
2 2
```

### 8.3.1.2 final Fields

A field can be declared `final` (§4.12.4). Both class and instance variables (static and non-static fields) may be declared `final`.

A blank `final` class variable must be definitely assigned by a static initializer of the class in which it is declared, or a compile-time error occurs (§8.7, §16.8).

A blank `final` instance variable must be definitely assigned and moreover not definitely unassigned at the end of every constructor of the class in which it is declared, or a compile-time error occurs (§8.8, §16.9).

### 8.3.1.3 transient Fields

Variables may be marked `transient` to indicate that they are not part of the persistent state of an object.
Example 8.3.1.3-1. Persistence of transient Fields

If an instance of the class Point:

```java
class Point {
    int x, y;
    transient float rho, theta;
}
```

were saved to persistent storage by a system service, then only the fields \(x\) and \(y\) would be saved. This specification does not specify details of such services; see the specification of `java.io.Serializable` for an example of such a service.

8.3.1.4 volatile Fields

The Java programming language allows threads to access shared variables (§17.1). As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that, conventionally, enforces mutual exclusion for those shared variables.

The Java programming language provides a second mechanism, volatile fields, that is more convenient than locking for some purposes.

A field may be declared volatile, in which case the Java Memory Model ensures that all threads see a consistent value for the variable (§17.4).

It is a compile-time error if a final variable is also declared volatile.

Example 8.3.1.4-1. volatile Fields

If, in the following example, one thread repeatedly calls the method `one` (but no more than `Integer.MAX_VALUE` times in all), and another thread repeatedly calls the method `two`:

```java
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i="); + i + " j=" + j);
    }
}
```

then method `two` could occasionally print a value for \(j\) that is greater than the value of \(i\), because the example includes no synchronization and, under the rules explained in §17.4, the shared values of \(i\) and \(j\) might be updated out of order.

One way to prevent this out-of-order behavior would be to declare methods `one` and `two` to be synchronized (§8.4.3.6):

```java
class Test {
```
This prevents method one and method two from being executed concurrently, and furthermore guarantees that the shared values of i and j are both updated before method one returns. Therefore method two never observes a value for j greater than that for i; indeed, it always observes the same value for i and j.

Another approach would be to declare i and j to be volatile:

```java
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

This allows method one and method two to be executed concurrently, but guarantees that accesses to the shared values for i and j occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text by each thread. Therefore, the shared value for j is never greater than that for i, because each update to i must be reflected in the shared value for i before the update to j occurs. It is possible, however, that any given invocation of method two might observe a value for j that is much greater than the value observed for i, because method one might be executed many times between the moment when method two fetches the value of i and the moment when method two fetches the value of j.

See §17.4 for more discussion and examples.

### 8.3.2 Field Initialization

If a declarator in a field declaration has a *variable initializer*, then the declarator has the semantics of an assignment (§15.26) to the declared variable.

If the declarator is for a class variable (that is, a *static field*), then the following rules apply to its initializer:

- It is a compile-time error if a reference by simple name to any instance variable occurs in the initializer.
- It is a compile-time error if the keyword `this` (§15.8.3) or the keyword `super` (§15.11.2, §15.12) occurs in the initializer.
- At run time, the initializer is evaluated and the assignment performed exactly once, when the class is initialized (§12.4.2).
Note that static fields that are constant variables (§4.12.4) are initialized before other static fields (§12.4.2). This also applies in interfaces (§9.3.1). When such fields are referenced by simple name, they will never be observed to have their default initial values (§4.12.5).

If the declarator is for an instance variable (that is, a field that is not static), then the following rules apply to its initializer:

- The initializer may refer by simple name to any class variable declared in or inherited by the class, even one whose declaration occurs to the right of the initializer (§3.5).

- The initializer may refer to the current object using the keyword this (§15.8.3) or the keyword super (§15.11.2, §15.12).

- At run time, the initializer is evaluated and the assignment performed each time an instance of the class is created (§12.5).

References from variable initializers to fields that may not yet be initialized are subject to additional restrictions, as specified in §8.3.3 and §16 (Definite Assignment).

Exception checking for a variable initializer in a field declaration is specified in §11.2.3.

Variable initializers are also used in local variable declaration statements (§14.4), where the initializer is evaluated and the assignment performed each time the local variable declaration statement is executed.

Example 8.3.2-1. Field Initialization

```java
class Point {
    int x = 1, y = 5;
}
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + "", " + p.y);
    }
}
```

This program produces the output:

```
1, 5
```

because the assignments to x and y occur whenever a new Point is created.
Example 8.3.2-2. Forward Reference to a Class Variable

```java
class Test {
   float f = j;
   static int j = 1;
}
```

This program compiles without error; it initializes `j` to `1` when class `Test` is initialized, and initializes `f` to the current value of `j` every time an instance of class `Test` is created.

8.3.3 Restrictions on Field References in Initializers

References to a field are sometimes restricted, even through the field is in scope. The following rules constrain forward references to a field (where the use textually precedes the field declaration) as well as self-reference (where the field is used in its own initializer).

For a reference by simple name to a class variable `f` declared in class or interface `C`, it is a compile-time error if:

- The reference appears either in a class variable initializer of `C` or in a static initializer of `C` (§8.7); and
- The reference appears either in the initializer of `f`'s own declarator or at a point to the left of `f`'s declarator; and
- The reference is not on the left hand side of an assignment expression (§15.26); and
- The innermost class or interface enclosing the reference is `C`.

For a reference by simple name to an instance variable `f` declared in class `C`, it is a compile-time error if:

- The reference appears either in an instance variable initializer of `C` or in an instance initializer of `C` (§8.6); and
- The reference appears in the initializer of `f`'s own declarator or at a point to the left of `f`'s declarator; and
- The reference is not on the left hand side of an assignment expression (§15.26); and
- The innermost class enclosing the reference is `C`.

Example 8.3.3-1. Restrictions on Field References

A compile-time error occurs for this program:
whereas the following program compiles without error:

```java
class Test2 {
    Test2() { k = 2; }
    int j = 1;
    int i = j;
    int k;
}
```
even though the constructor for Test2 (§8.8) refers to the field k that is declared three lines later.

The restrictions above are designed to catch, at compile time, circular or otherwise malformed initializations. Thus, both:

```java
class Z {
    static int i = j + 2;
    static int j = 4;
}
```
and:

```java
class Z {
    static { i = j + 2; }
    static int i, j;
    static { j = 4; }
}
```
result in compile-time errors. Accesses by methods are not checked in this way, so:

```java
class Z {
    static int peek() { return j; }
    static int i = peek();
    static int j = 1;
}
class Test {
    public static void main(String[] args) {
        System.out.println(Z.i);
    }
}
```
produces the output:

0
because the variable initializer for \( i \) uses the class method `peek` to access the value of the variable \( j \) before \( j \) has been initialized by its variable initializer, at which point it still has its default value (§4.12.5).

A more elaborate example is:

```java
class UseBeforeDeclaration {
    static {
        x = 100;
        // ok - assignment
        int y = x + 1;
        // error - read before declaration
        int v = x = 3;
        // ok - x at left hand side of assignment
        int z = UseBeforeDeclaration.x * 2;
        // ok - not accessed via simple name

        Object o = new Object() {
            void foo() {
                // ok - occurs in a different class
                x++; }
            // ok - occurs in a different class
        };
    }

    {
        j = 200;
        // ok - assignment
        j = j + 1;
        // error - right hand side reads before declaration
        int k = j = j + 1;
        // error - illegal forward reference to j
        int n = j = 300;
        // ok - j at left hand side of assignment
        int h = j++;
        // error - read before declaration
        int l = this.j * 3;
        // ok - not accessed via simple name

        Object o = new Object() {
            void foo() {
                // ok - occurs in a different class
                j = j + 1; }
            // ok - occurs in a different class
        };
    }

    int w = x = 3;
    // ok - x at left hand side of assignment
    int p = x;
    // ok - instance initializers may access static fields

    static int u =
```
(new Object() { int bar() { return x; } }).bar();
// ok - occurs in a different class
static int x;

int m = j = 4;
// ok - j at left hand side of assignment
int o =
  (new Object() { int bar() { return j; } }).bar();
// ok - occurs in a different class
int j;

8.4 Method Declarations

A method declares executable code that can be invoked, passing a fixed number of values as arguments.

MethodDeclaration:
  {MethodModifier} MethodHeader MethodBody

MethodHeader:
  Result MethodDeclarator [Throws]
  TypeParameters {Annotation} Result MethodDeclarator [Throws]

MethodDeclarator:
  Identifier ( [ReceiverParameter , ] [FormalParameterList] ) [Dims]

ReceiverParameter:
  {Annotation} UnannType {Identifier .}this

The following production from §4.3 is shown here for convenience:

Dims:
  {Annotation} [ ] {{Annotation} [ ]}

The FormalParameterList clause is described in §8.4.1, the MethodModifier clause in §8.4.3, the TypeParameters clause in §8.4.4, the Result clause in §8.4.5, the Throws clause in §8.4.6, and the MethodBody in §8.4.7.

The Identifier in a MethodDeclarator may be used in a name to refer to the method (§6.5.7.1, §15.12).

The scope and shadowing of a method declaration is specified in §6.3 and §6.4.
The receiver parameter is an optional syntactic device for an instance method or an inner class’s constructor. For an instance method, the receiver parameter represents the object for which the method is invoked. For an inner class's constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. In both cases, the receiver parameter exists solely to allow the type of the represented object to be denoted in source code, so that the type may be annotated (§9.7.4). The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable (§4.12.3), it is never bound to any value passed as an argument in a method invocation expression or class instance creation expression, and it has no effect whatsoever at run time.

A receiver parameter may appear either in the MethodDeclarator of an instance method or in the ConstructorDeclarator of a constructor of an inner class where the inner class is not declared in a static context (§8.1.3). If a receiver parameter appears in any other kind of method or constructor, then a compile-time error occurs.

The type and name of a receiver parameter are constrained as follows:

- In an instance method, the type of the receiver parameter must be the class or interface in which the method is declared, and the name of the receiver parameter must be this; otherwise, a compile-time error occurs.

- In an inner class's constructor, the type of the receiver parameter must be the class or interface which is the immediately enclosing type declaration of the inner class, and the name of the receiver parameter must be Identifier . this where Identifier is the simple name of the class or interface which is the immediately enclosing type declaration of the inner class; otherwise, a compile-time error occurs.

It is a compile-time error for the body of a class declaration to declare as members two methods with override-equivalent signatures (§8.4.2).

The declaration of a method that returns an array is allowed to place some or all of the bracket pairs that denote the array type after the formal parameter list. This syntax is supported for compatibility with early versions of the Java programming language. It is very strongly recommended that this syntax is not used in new code.

### 8.4.1 Formal Parameters

The formal parameters of a method or constructor, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type (optionally preceded by the final modifier and/or one or more annotations)
and an identifier (optionally followed by brackets) that specifies the name of the parameter.

If a method or constructor has no formal parameters, and no receiver parameter, then an empty pair of parentheses appears in the declaration of the method or constructor.

\[
\text{FormalParameterList:} \quad \text{FormalParameter} \{ , \text{FormalParameter} \}
\]

\[
\text{FormalParameter:} \quad \{\text{VariableModifier}\} \text{UnannType VariableDeclaratorId VariableArityParameter}
\]

\[
\text{VariableArityParameter:} \quad \{\text{VariableModifier}\} \text{UnannType \{Annotation\} ... Identifier}
\]

\[
\text{VariableModifier:} \quad \text{Annotation} \quad \text{final}
\]

The following productions from §8.3 and §4.3 are shown here for convenience:

\[
\text{VariableDeclaratorId:} \quad \text{Identifier [Dims]}
\]

\[
\text{Dims:} \quad \{\text{Annotation}\} \{\} \{\text{Annotation}\} \{\}
\]

A formal parameter of a method or constructor may be a \textit{variable arity parameter}, indicated by an ellipsis following the type. At most one variable arity parameter is permitted for a method or constructor. It is a compile-time error if a variable arity parameter appears anywhere in the list of parameter specifiers except the last position.

In the grammar for \texttt{VariableArityParameter}, note that the ellipsis (\ldots) is a token unto itself (§3.11). It is possible to put whitespace between it and the type, but this is discouraged as a matter of style.

If the last formal parameter is a variable arity parameter, the method is a \textit{variable arity method}. Otherwise, it is a \textit{fixed arity method}.

The rules for annotation modifiers on a formal parameter declaration and on a receiver parameter are specified in §9.7.4 and §9.7.5.
It is a compile-time error if `final` appears more than once as a modifier for a formal parameter declaration.

The scope and shadowing of a formal parameter is specified in §6.3 and §6.4.

It is a compile-time error for a method or constructor to declare two formal parameters with the same name. (That is, their declarations mention the same `Identifier`.)

It is a compile-time error if a formal parameter that is declared `final` is assigned to within the body of the method or constructor.

The declared type of a formal parameter depends on whether it is a variable arity parameter:

- If the formal parameter is not a variable arity parameter, then the declared type is denoted by `UnannType` if no bracket pairs appear in `UnannType` and `VariableDeclaratorId`, and specified by §10.2 otherwise.

- If the formal parameter is a variable arity parameter, then the declared type is an array type specified by §10.2.

If the declared type of a variable arity parameter has a non-reifiable element type (§4.7), then a compile-time unchecked warning occurs for the declaration of the variable arity method, unless the method is annotated with `@SafeVarargs` (§9.6.4.7) or the warning is suppressed by `@SuppressWarnings` (§9.6.4.5).

When the method or constructor is invoked (§15.12), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method or constructor. The `Identifier` that appears in the `FormalParameter` may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

Invocations of a variable arity method may contain more actual argument expressions than formal parameters. All the actual argument expressions that do not correspond to the formal parameters preceding the variable arity parameter will be evaluated and the results stored into an array that will be passed to the method invocation (§15.12.4.2).

A method's or constructor's formal parameter of type `float` always contains an element of the float value set (§4.2.3); similarly, a method's or constructor's formal parameter of type `double` always contains an element of the double value set. It is not permitted for a method's or constructor's formal parameter of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a method's or constructor's formal parameter
of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

Where an actual argument expression corresponding to a parameter variable is not FP-strict (§15.4), evaluation of that actual argument expression is permitted to use intermediate values drawn from the appropriate extended-exponent value sets. Prior to being stored in the parameter variable, the result of such an expression is mapped to the nearest value in the corresponding standard value set by being subjected to invocation conversion (§5.3).

Here are some examples of receiver parameters in instance methods and inner classes' constructors:

```java
class Test {
    Test(/* ?? ?? */) {}
    // No receiver parameter is permitted in the constructor of
    // a top level class, as there is no conceivable type or name.

    void m(Test this) {}
    // OK: receiver parameter in an instance method

    static void n(Test this) {}
    // Illegal: receiver parameter in a static method

class A {
    A(Test Test.this) {}
    // OK: the receiver parameter represents the instance
    // of Test which immediately encloses the instance
    // of A being constructed.

    void m(A this) {}
    // OK: the receiver parameter represents the instance
    // of A for which A.m() is invoked.

    class B {
        B(Test.A A.this) {}
        // OK: the receiver parameter represents the instance
        // of A which immediately encloses the instance of B
        // being constructed.

        void m(Test.A.B this) {}
        // OK: the receiver parameter represents the instance
        // of B for which B.m() is invoked.
    }
}
```

B's constructor and instance method show that the type of the receiver parameter may be denoted with a qualified `TypeName` like any other type; but that the name of the receiver parameter in an inner class's constructor must use the simple name of the enclosing class.
8.4.2 Method Signature

Two methods or constructors, M and N, have the same signature if they have the same name, the same type parameters (if any) (§8.4.4), and, after adapting the formal parameter types of N to the the type parameters of M, the same formal parameter types.

The signature of a method \( m_1 \) is a subsignature of the signature of a method \( m_2 \) if either:

- \( m_2 \) has the same signature as \( m_1 \), or
- the signature of \( m_1 \) is the same as the erasure (§4.6) of the signature of \( m_2 \).

Two method signatures \( m_1 \) and \( m_2 \) are override-equivalent iff either \( m_1 \) is a subsignature of \( m_2 \) or \( m_2 \) is a subsignature of \( m_1 \).

It is a compile-time error to declare two methods with override-equivalent signatures in a class.

Example 8.4.2-1. Override-Equivalent Signatures

```java
class Point {
    int x, y;
    abstract void move(int dx, int dy);
    void move(int dx, int dy) { x += dx; y += dy; }
}
```

This program causes a compile-time error because it declares two move methods with the same (and hence, override-equivalent) signature. This is an error even though one of the declarations is abstract.

The notion of subsignature is designed to express a relationship between two methods whose signatures are not identical, but in which one may override the other. Specifically, it allows a method whose signature does not use generic types to override any generified version of that method. This is important so that library designers may freely generify methods independently of clients that define subclasses or subinterfaces of the library.

Consider the example:

```java
class CollectionConverter {
    List toList(Collection c) {...}
}
class Overrider extends CollectionConverter {
    List toList(Collection c) {...}
}
```

Now, assume this code was written before the introduction of generics, and now the author of class CollectionConverter decides to generify the code, thus:
class CollectionConverter {
    <T> List<T> toList(Collection<T> c) {...}
}

Without special dispensation, Overrider.toList would no longer override CollectionConverter.toList. Instead, the code would be illegal. This would significantly inhibit the use of generics, since library writers would hesitate to migrate existing code.

8.4.3 Method Modifiers

MethodModifier:
(one of)
Annotation public protected private
abstract static final synchronized native strictfp

The rules for annotation modifiers on a method declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration, or if a method declaration has more than one of the access modifiers public, protected, and private (§6.6).

It is a compile-time error if a method declaration that contains the keyword abstract also contains any one of the keywords private, static, final, native, strictfp, or synchronized.

It is a compile-time error if a method declaration that contains the keyword native also contains strictfp.

If two or more (distinct) method modifiers appear in a method declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for MethodModifier.

8.4.3.1 abstract Methods

An abstract method declaration introduces the method as a member, providing its signature (§8.4.2), result (§8.4.5), and throws clause if any (§8.4.6), but does not provide an implementation (§8.4.7). A method that is not abstract may be referred to as a concrete method.

The declaration of an abstract method m must appear directly within an abstract class (call it A) unless it occurs within an enum declaration (§8.9); otherwise, a compile-time error occurs.
Every subclass of A that is not abstract (§8.1.1.1) must provide an implementation for m, or a compile-time error occurs.

An abstract class can override an abstract method by providing another abstract method declaration.

This can provide a place to put a documentation comment, to refine the return type, or to declare that the set of checked exceptions that can be thrown by that method, when it is implemented by its subclasses, is to be more limited.

An instance method that is not abstract can be overridden by an abstract method.

Example 8.4.3.1-1. Abstract/Abstract Method Overriding

```java
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
    BufferError() { super(); }
    BufferError(String s) { super(s); }
}
interface Buffer {
    char get() throws BufferEmpty, BufferError;
}
abstract class InfiniteBuffer implements Buffer {
    public abstract char get() throws BufferError;
}
```

The overriding declaration of method get in class InfiniteBuffer states that method get in any subclass of InfiniteBuffer never throws a BufferEmpty exception, putatively because it generates the data in the buffer, and thus can never run out of data.

Example 8.4.3.1-2. Abstract/Non-Abstract Overriding

We can declare an abstract class Point that requires its subclasses to implement toString if they are to be complete, instantiable classes:

```java
abstract class Point {
    int x, y;
    public abstract String toString();
}
```

This abstract declaration of toString overrides the non-abstract toString method of class Object. (Class Object is the implicit direct superclass of class Point.) Adding the code:

```java
class ColoredPoint extends Point {
    int color;
}
public String toString() {
    return super.toString() + " color " + color; // error
}
}

results in a compile-time error because the invocation super.toString() refers to method toString in class Point, which is abstract and therefore cannot be invoked. Method toString of class Object can be made available to class ColoredPoint only if class Point explicitly makes it available through some other method, as in:

abstract class Point {
    int x, y;
    public abstract String toString();
    protected String objString() { return super.toString(); }
}

class ColoredPoint extends Point {
    int color;
    public String toString() {
        return objString() + " color " + color; // correct
    }
}

8.4.3.2 static Methods

A method that is declared static is called a class method.

It is a compile-time error to use the name of a type parameter of any surrounding declaration in the header or body of a class method.

A class method is always invoked without reference to a particular object. It is a compile-time error to attempt to refer to the current object using the keyword this (§15.8.3) or the keyword super (§15.11.2).

A method that is not declared static is called an instance method, and sometimes called a non-static method.

An instance method is always invoked with respect to an object, which becomes the current object to which the keywords this and super refer during execution of the method body.

8.4.3.3 final Methods

A method can be declared final to prevent subclasses from overriding or hiding it.

It is a compile-time error to attempt to override or hide a final method.

A private method and all methods declared immediately within a final class (§8.1.1.2) behave as if they are final, since it is impossible to override them.
At run time, a machine-code generator or optimizer can "inline" the body of a final method, replacing an invocation of the method with the code in its body. The inlining process must preserve the semantics of the method invocation. In particular, if the target of an instance method invocation is null, then a NullPointerException must be thrown even if the method is inlined. A Java compiler must ensure that the exception will be thrown at the correct point, so that the actual arguments to the method will be seen to have been evaluated in the correct order prior to the method invocation.

Consider the example:

```java
final class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}

class Test {
    public static void main(String[] args) {
        Point[] p = new Point[100];
        for (int i = 0; i < p.length; i++) {
            p[i] = new Point();
            p[i].move(i, p.length-1-i);
        }
    }
}
```

Inlining the method `move` of class `Point` in method `main` would transform the `for` loop to the form:

```java
for (int i = 0; i < p.length; i++) {
    p[i] = new Point();
    Point pi = p[i];
    int j = p.length-1-i;
    pi.x += i;
    pi.y += j;
}
```

The loop might then be subject to further optimizations.

Such inlining cannot be done at compile time unless it can be guaranteed that `Test` and `Point` will always be recompiled together, so that whenever `Point` - and specifically its `move` method - changes, the code for `Test.main` will also be updated.

### 8.4.3.4 native Methods

A method that is native is implemented in platform-dependent code, typically written in another programming language such as C. The body of a native method is given as a semicolon only, indicating that the implementation is omitted, instead of a block (§8.4.7).

For example, the class `RandomAccessFile` of the package `java.io` might declare the following native methods:
package java.io;
public class RandomAccessFile
    implements DataOutput, DataInput {
    . . .
    public native void open(String name, boolean writeable)
        throws IOException;
    public native int readBytes(byte[] b, int off, int len)
        throws IOException;
    public native void writeBytes(byte[] b, int off, int len)
        throws IOException;
    public native long getFilePointer() throws IOException;
    public native void seek(long pos) throws IOException;
    public native long length() throws IOException;
    public native void close() throws IOException;
}

8.4.3.5 strictfp Methods
The effect of the strictfp modifier is to make all float or double expressions
within the method body be explicitly FP-strict (§15.4).

8.4.3.6 synchronized Methods
A synchronized method acquires a monitor (§17.1) before it executes.
For a class (static) method, the monitor associated with the Class object for the
method's class is used.
For an instance method, the monitor associated with this (the object for which the
method was invoked) is used.

Example 8.4.3.6-1. synchronized Monitors
These are the same monitors that can be used by the synchronized statement (§14.19).
Thus, the code:

class Test {
    int count;
    synchronized void bump() {
        count++;
    }
    static int classCount;
    static synchronized void classBump() {
        classCount++;
    }
}

has exactly the same effect as:
```java
class BumpTest {
    int count;
    void bump() {
        synchronized (this) { count++; }
    }
    static int classCount;
    static void classBump() {
        try {
            synchronized (Class.forName("BumpTest")) {
                classCount++;
            }
        } catch (ClassNotFoundException e) {} 
    }
}
```

**Example 8.4.3-2. synchronized Methods**

```java
public class Box {
    private Object boxContents;
    public synchronized Object get() {
        Object contents = boxContents;
        boxContents = null;
        return contents;
    }
    public synchronized boolean put(Object contents) {
        if (boxContents != null) return false;
        boxContents = contents;
        return true;
    }
}
```

This program defines a class which is designed for concurrent use. Each instance of the class `Box` has an instance variable `boxContents` that can hold a reference to any object. You can put an object in a `Box` by invoking `put`, which returns `false` if the box is already full. You can get something out of a `Box` by invoking `get`, which returns a null reference if the box is empty.

If `put` and `get` were not synchronized, and two threads were executing methods for the same instance of `Box` at the same time, then the code could misbehave. It might, for example, lose track of an object because two invocations to `put` occurred at the same time.

### 8.4.4 Generic Methods

A method is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the method. The form of the type parameter section of a generic method is identical to the type parameter section of a generic class (§8.1.2).

A generic method declaration defines a set of methods, one for each possible invocation of the type parameter section by type arguments. Type arguments may
not need to be provided explicitly when a generic method is invoked, as they can often be inferred (§18 (Type Inference)).

The scope and shadowing of a method’s type parameter is specified in §6.3.

Two methods or constructors $M$ and $N$ have the same type parameters if both of the following are true:

- $M$ and $N$ have same number of type parameters (possibly zero).
- Where $A_1, \ldots, A_n$ are the type parameters of $M$ and $B_1, \ldots, B_n$ are the type parameters of $N$, let $\theta = [B_1 := A_1, \ldots, B_n := A_n]$. Then, for all $i$ ($1 \leq i \leq n$), the bound of $A_i$ is the same type as $\theta$ applied to the bound of $B_i$.

Where two methods or constructors $M$ and $N$ have the same type parameters, a type mentioned in $N$ can be adapted to the type parameters of $M$ by applying $\theta$, as defined above, to the type.

### 8.4.5 Method Result

The result of a method declaration either declares the type of value that the method returns (the return type), or uses the keyword `void` to indicate that the method does not return a value.

\[
\text{Result:}
\begin{align*}
\text{UnannType} \\
\text{void}
\end{align*}
\]

If the result is not `void`, then the return type of a method is denoted by `UnannType` if no bracket pairs appear after the formal parameter list, and is specified by §10.2 otherwise.

Return types may vary among methods that override each other if the return types are reference types. The notion of return-type-substitutability supports covariant returns, that is, the specialization of the return type to a subtype.

A method declaration $d_1$ with return type $R_1$ is return-type-substitutable for another method $d_2$ with return type $R_2$ iff any of the following is true:

- If $R_1$ is `void` then $R_2$ is `void`.
- If $R_1$ is a primitive type then $R_2$ is identical to $R_1$.
- If $R_1$ is a reference type then one of the following is true:
  - $R_1$, adapted to the type parameters of $d_2$ (§8.4.4), is a subtype of $R_2$. 

– $R_1$ can be converted to a subtype of $R_2$ by unchecked conversion (§5.1.9).

– $d_1$ does not have the same signature as $d_2$ (§8.4.2), and $R_1 = |R_2|$. 

An unchecked conversion is allowed in the definition, despite being unsound, as a special allowance to allow smooth migration from non-generic to generic code. If an unchecked conversion is used to determine that $R_1$ is return-type-substitutable for $R_2$, then $R_1$ is necessarily not a subtype of $R_2$ and the rules for overriding (§8.4.8.3, §9.4.1) will require a compile-time unchecked warning.

8.4.6 Method Throws

A `throws` clause is used to denote any checked exception classes (§11.1.1) that the statements in a method or constructor body can throw (§11.2.2).

Throws:

```
throws ExceptionTypeList
```

ExceptionTypeList:

```
ExceptionType { , ExceptionType}
```

ExceptionType:

```
ClassType
TypeVariable
```

It is a compile-time error if an `ExceptionType` mentioned in a `throws` clause is not a subtype (§4.10) of `Throwable`.

Type variables are allowed in a `throws` clause even though they are not allowed in a `catch` clause (§14.20).

It is permitted but not required to mention unchecked exception classes (§11.1.1) in a `throws` clause.

The relationship between a `throws` clause and the exception checking for a method or constructor body is specified in §11.2.3.

Essentially, for each checked exception that can result from execution of the body of a method or constructor, a compile-time error occurs unless its exception type or a supertype of its exception type is mentioned in a `throws` clause in the declaration of the method or constructor.

The requirement to declare checked exceptions allows a Java compiler to ensure that code for handling such error conditions has been included. Methods or constructors that fail to handle exceptional conditions thrown as checked exceptions in their bodies will normally cause compile-time errors if they lack proper exception types in their `throws` clauses. The
Java programming language thus encourages a programming style where rare and otherwise truly exceptional conditions are documented in this way.

The relationship between the throws clause of a method and the throws clauses of overridden or hidden methods is specified in §8.4.8.3.

**Example 8.4.6-1. Type Variables as Thrown Exception Types**

```java
import java.io.FileNotFoundException;
interface PrivilegedExceptionAction<E extends Exception> {
    void run() throws E;
}

class AccessController {
    public static <E extends Exception>
    Object doPrivileged(PrivilegedExceptionAction<E> action) throws E {
        action.run();
        return "success";
    }
}

class Test {
    public static void main(String[] args) {
        try {
            AccessController.doPrivileged(
                new PrivilegedExceptionAction<FileNotFoundException>() {
                    public void run() throws FileNotFoundException {
                        // ... delete a file ...
                    }
                });
            } catch (FileNotFoundException f) { /* Do something */ }
    }
}
```

### 8.4.7 Method Body

A *method body* is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation.

**MethodBody:**

```
Block
;
```

The body of a method must be a semicolon if the method is abstract or native (§8.4.3.1, §8.4.3.4). More precisely:

- It is a compile-time error if a method declaration is either abstract or native and has a block for its body.
- It is a compile-time error if a method declaration is neither abstract nor native and has a semicolon for its body.
If an implementation is to be provided for a method declared void, but the implementation requires no executable code, the method body should be written as a block that contains no statements: "{}".

The rules for return statements in a method body are specified in §14.17.

If a method is declared to have a return type (§8.4.5), then a compile-time error occurs if the body of the method can complete normally (§14.1).

In other words, a method with a return type must return only by using a return statement that provides a value return; the method is not allowed to "drop off the end of its body". See §14.17 for the precise rules about return statements in a method body.

It is possible for a method to have a return type and yet contain no return statements. Here is one example:

class DizzyDean {
    int pitch() { throw new RuntimeException("90 mph?!"}; }
}

8.4.8 Inheritance, Overriding, and Hiding

A class \( C \) inherits from its direct superclass all concrete methods \( m \) (both static and instance) of the superclass for which all of the following are true:

- \( m \) is a member of the direct superclass of \( C \).
- \( m \) is public, protected, or declared with package access in the same package as \( C \).
- No method declared in \( C \) has a signature that is a subsignature (§8.4.2) of the signature of \( m \).

A class \( C \) inherits from its direct superclass and direct superinterfaces all abstract and default (§9.4) methods \( m \) for which all of the following are true:

- \( m \) is a member of the direct superclass or a direct superinterface, \( D \), of \( C \).
- \( m \) is public, protected, or declared with package access in the same package as \( C \).
- No method declared in \( C \) has a signature that is a subsignature (§8.4.2) of the signature of \( m \).
- No concrete method inherited by \( C \) from its direct superclass has a signature that is a subsignature of the signature of \( m \).
• There exists no method \( m' \) that is a member of the direct superclass or a direct superinterface, \( D' \), of \( C (m \text{ distinct from } m', D \text{ distinct from } D') \), such that \( m' \) overrides from \( D' \) (§8.4.8.1, §9.4.1.1) the declaration of the method \( m \).

A class does not inherit private or static methods from its superinterfaces.

Note that methods are overridden or hidden on a signature-by-signature basis. If, for example, a class declares two public methods with the same name (§8.4.9), and a subclass overrides one of them, the subclass still inherits the other method.

**Example 8.4.8-1. Inheritance**

```
interface I1 {
    int foo();
}

interface I2 {
    int foo();
}

abstract class Test implements I1, I2 {}
```

Here, the abstract class `Test` inherits the abstract method `foo` from interface `I1` and also the abstract method `foo` from interface `I2`. The key question in determining the inheritance of `foo` from `I1` is: does the method `foo` in `I2` override "from `I2" (§9.4.1.1) the method `foo` in `I1"? No, because `I1` and `I2` are not subinterfaces of each other. Thus, from the viewpoint of class `Test`, the inheritance of `foo` from `I1` is unfettered; similarly for the inheritance of `foo` from `I2`. Per §8.4.8.4, class `Test` can inherit both `foo` methods; obviously it must be declared abstract, or else override both abstract `foo` methods with a concrete method.

Note that it is possible for an inherited concrete method to prevent the inheritance of an abstract or default method. (The concrete method will override the abstract or default method "from \( C'\), per §8.4.8.1 and §9.4.1.1.) Also, it is possible for one supertype method to prevent the inheritance of another supertype method if the former "already" overrides the latter - this is the same as the rule for interfaces (§9.4.1), and prevents conflicts in which multiple default methods are inherited and one implementation is clearly meant to supersede the other.

### 8.4.8.1 Overriding (by Instance Methods)

An instance method \( m_C \) declared in or inherited by class \( C \), overrides from \( C \) another method \( m_A \) declared in class \( A \), iff all of the following are true:

• \( C \) is a subclass of \( A \).
• \( C \) does not inherit \( m_A \).
• The signature of \( m_C \) is a subsignature (§8.4.2) of the signature of \( m_A \).
• One of the following is true:
- \( m_A \) is public.
- \( m_A \) is protected.
- \( m_A \) is declared with package access in the same package as \( C \), and either \( C \) declares \( m_C \) or \( m_A \) is a member of the direct superclass of \( C \).
- \( m_A \) is declared with package access and \( m_C \) overrides \( m_A \) from some superclass of \( C \).
- \( m_A \) is declared with package access and \( m_C \) overrides a method \( m' \) from \( C \) (\( m' \) distinct from \( m_C \) and \( m_A \)), such that \( m' \) overrides \( m_A \) from some superclass of \( C \).

If \( m_C \) is non-abstract and overrides from \( C \) an abstract method \( m_A \), then \( m_C \) is said to implement \( m_A \) from \( C \).

It is a compile-time error if the overridden method, \( m_A \), is a static method.

In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for an instance variable to hide a static variable.

An instance method \( m_C \) declared in or inherited by class \( C \), overrides from \( C \) another method \( m_I \) declared in interface \( I \), iff all of the following are true:

- \( I \) is a superinterface of \( C \).
- \( m_I \) is not static.
- \( C \) does not inherit \( m_I \).
- The signature of \( m_C \) is a subsignature (§8.4.2) of the signature of \( m_I \).
- \( m_I \) is public.

The signature of an overriding method may differ from the overridden one if a formal parameter in one of the methods has a raw type, while the corresponding parameter in the other has a parameterized type. This accommodates migration of pre-existing code to take advantage of generics.

The notion of overriding includes methods that override another from some subclass of their declaring class. This can happen in two ways:

- A concrete method in a generic superclass can, under certain parameterizations, have the same signature as an abstract method in that class. In this case, the concrete method is inherited and the abstract method is not (as described above). The inherited method should then be considered to override its abstract peer from \( C \). (This scenario is complicated by package access: if \( C \) is in a different package, then \( m_A \) would not have been inherited anyway, and should not be considered overridden.)
- A method inherited from a class can override a superinterface method. (Happily, package access is not a concern here.)
An overridden method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super`. A qualified name or a cast to a superclass type is not effective in attempting to access an overridden method.

In this respect, overriding of methods differs from hiding of fields.

The presence or absence of the `strictfp` modifier has absolutely no effect on the rules for overriding methods and implementing abstract methods. For example, it is permitted for a method that is not FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

Example 8.4.8.1-1. Overriding

```java
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
    }
}
```

Here, the class `SlowPoint` overrides the declarations of method `move` of class `Point` with its own `move` method, which limits the distance that the point can move on each invocation of the method. When the `move` method is invoked for an instance of class `SlowPoint`, the overriding definition in class `SlowPoint` will always be called, even if the reference to the `SlowPoint` object is taken from a variable whose type is `Point`.

Example 8.4.8.1-2. Overriding

Overriding makes it easy for subclasses to extend the behavior of an existing class, as shown in this example:

```java
import java.io.OutputStream;
import java.io.IOException;

class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o = o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws IOException {
        if (pos == buf.length) flush();
        buf[pos++] = (byte)c;
    }
}
public void putstr(String s) throws IOException {
    for (int i = 0; i < s.length(); i++)
        putchar(s.charAt(i));
}
public void flush() throws IOException {
    o.write(buf, 0, pos);
    pos = 0;
}
}
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n') flush();
    }
}
class Test {
    public static void main(String[] args) throws IOException {
        LineBufferOutput lbo = new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}

This program produces the output:

lbo
print
lbo

The class BufferOutput implements a very simple buffered version of an OutputStream, flushing the output when the buffer is full or flush is invoked. The subclass LineBufferOutput declares only a constructor and a single method putchar, which overrides the method putchar of BufferOutput. It inherits the methods putstr and flush from class BufferOutput.

In the putchar method of a LineBufferOutput object, if the character argument is a newline, then it invokes the flush method. The critical point about overriding in this example is that the method putstr, which is declared in class BufferOutput, invokes the putchar method defined by the current object this, which is not necessarily the putchar method declared in class BufferOutput.

Thus, when putstr is invoked in main using the LineBufferOutput object lbo, the invocation of putchar in the body of the putstr method is an invocation of the putchar of the object lbo, the overriding declaration of putchar that checks for a newline. This allows a subclass of BufferOutput to change the behavior of the putstr method without redefining it.

Documentation for a class such as BufferOutput, which is designed to be extended, should clearly indicate what is the contract between the class and its subclasses, and should clearly indicate that subclasses may override the putchar method in this way.
The implementor of the BufferOutput class would not, therefore, want to change the implementation of putstr in a future implementation of BufferOutput not to use the method putchar, because this would break the pre-existing contract with subclasses. See the discussion of binary compatibility in §13 (Binary Compatibility), especially §13.2.

8.4.8.2 Hiding (by Class Methods)

If a class C declares or inherits a static method m, then m is said to hide any method m', where the signature of m is a subsignature (§8.4.2) of the signature of m', in the superclasses and superinterfaces of C that would otherwise be accessible (§6.6) to code in C.

It is a compile-time error if a static method hides an instance method.

In this respect, hiding of methods differs from hiding of fields (§8.3), for it is permissible for a static variable to hide an instance variable. Hiding is also distinct from shadowing (§6.4.1) and obscuring (§6.4.2).

A hidden method can be accessed by using a qualified name or by using a method invocation expression (§15.12) that contains the keyword super or a cast to a superclass type.

In this respect, hiding of methods is similar to hiding of fields.

Example 8.4.8.2-1. Invocation of Hidden Class Methods

A class (static) method that is hidden can be invoked by using a reference whose type is the class that actually contains the declaration of the method. In this respect, hiding of static methods is different from overriding of instance methods. The example:

class Super {
    static String greeting() { return "Goodnight"; }
    String name() { return "Richard"; }
}
class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}

produces the output:

Goodnight, Dick
because the invocation of `greeting` uses the type of `s`, namely `Super`, to figure out, at compile time, which class method to invoke, whereas the invocation of `name` uses the class of `s`, namely `Sub`, to figure out, at run time, which instance method to invoke.

### 8.4.8.3 Requirements in Overriding and Hiding

If a method declaration $d_1$ with return type $R_1$ overrides or hides the declaration of another method $d_2$ with return type $R_2$, then $d_1$ must be return-type-substitutable (§8.4.5) for $d_2$, or a compile-time error occurs.

This rule allows for covariant return types - refining the return type of a method when overriding it.

If $R_1$ is not a subtype of $R_2$, then a compile-time unchecked warning occurs, unless suppressed by `@SuppressWarnings` (§9.6.4.5).

A method that overrides or hides another method, including methods that implement abstract methods defined in interfaces, may not be declared to throw more checked exceptions than the overridden or hidden method.

In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for a field to hide a field of another type.

More precisely, suppose that $B$ is a class or interface, and $A$ is a superclass or superinterface of $B$, and a method declaration $m_2$ in $B$ overrides or hides a method declaration $m_1$ in $A$. Then:

- If $m_2$ has a `throws` clause that mentions any checked exception types, then $m_1$ must have a `throws` clause, or a compile-time error occurs.
- For every checked exception type listed in the `throws` clause of $m_2$, that same exception class or one of its supertypes must occur in the erasure (§4.6) of the `throws` clause of $m_1$; otherwise, a compile-time error occurs.
- If the unerased `throws` clause of $m_1$ does not contain a supertype of each exception type in the `throws` clause of $m_2$ (adapted, if necessary, to the type parameters of $m_1$), then a compile-time unchecked warning occurs, unless suppressed by `@SuppressWarnings` (§9.6.4.5).

It is a compile-time error if a type declaration $T$ has a member method $m_1$ and there exists a method $m_2$ declared in $T$ or a supertype of $T$ such that all of the following are true:

- $m_1$ and $m_2$ have the same name.
- $m_2$ is accessible (§6.6) from $T$. 
• The signature of $m_1$ is not a subsignature (§8.4.2) of the signature of $m_2$.

• The signature of $m_1$ or some method $m_1$ overrides (directly or indirectly) has the same erasure as the signature of $m_2$ or some method $m_2$ overrides (directly or indirectly).

These restrictions are necessary because generics are implemented via erasure. The rule above implies that methods declared in the same class with the same name must have different erasures. It also implies that a type declaration cannot implement or extend two distinct invocations of the same generic interface.

The access modifier of an overriding or hiding method must provide at least as much access as the overridden or hidden method, as follows:

• If the overridden or hidden method is public, then the overriding or hiding method must be public; otherwise, a compile-time error occurs.

• If the overridden or hidden method is protected, then the overriding or hiding method must be protected or public; otherwise, a compile-time error occurs.

• If the overridden or hidden method has package access, then the overriding or hiding method must not be private; otherwise, a compile-time error occurs.

Note that a private method cannot be overridden or hidden in the technical sense of those terms. This means that a subclass can declare a method with the same signature as a private method in one of its superclasses, and there is no requirement that the return type or throws clause of such a method bear any relationship to those of the private method in the superclass.

Example 8.4.8.3-1. Covariant Return Types

The following declarations are legal in the Java programming language from Java SE 5.0 onwards:

```java
class C implements Cloneable {
    C copy() throws CloneNotSupportedException {
        return (C)clone();
    }
}
class D extends C implements Cloneable {
    D copy() throws CloneNotSupportedException {
        return (D)clone();
    }
}
```

The relaxed rule for overriding also allows one to relax the conditions on abstract classes implementing interfaces.
Example 8.4.8.3-2. Unchecked Warning from Return Type

Consider:

```java
class StringSorter {
    // turns a collection of strings into a sorted list
    List toList(Collection c) {...}
}
```

and assume that someone subclasses `StringSorter`:

```java
class Overrider extends StringSorter {
    List toList(Collection c) {...}
}
```

Now, at some point the author of `StringSorter` decides to generify the code:

```java
class StringSorter {
    // turns a collection of strings into a sorted list
    List<String> toList(Collection<String> c) {...}
}
```

An unchecked warning would be given when compiling `Overrider` against the new definition of `StringSorter` because the return type of `Overrider.toList` is `List`, which is not a subtype of the return type of the overridden method, `List<String>`.

Example 8.4.8.3-3. Incorrect Overriding because of `throws`

This program uses the usual and conventional form for declaring a new exception type, in its declaration of the class `BadPointException`:

```java
class BadPointException extends Exception {
    BadPointException() { super(); }
    BadPointException(String s) { super(s); }
}
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
    void move(int dx, int dy) throws BadPointException {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

The program results in a compile-time error, because the override of method `move` in class `CheckedPoint` declares that it will throw a checked exception that the `move` in class `Point` has not declared. If this were not considered an error, an invoker of the method `move` on
a reference of type Point could find the contract between it and Point broken if this exception were thrown.

Removing the throws clause does not help:

```java
class CheckedPoint extends Point {
    void move(int dx, int dy) {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

A different compile-time error now occurs, because the body of the method move cannot throw a checked exception, namely BadPointException, that does not appear in the throws clause for move.

**Example 8.4.8.3-4. Erasure Affects Overriding**

A class cannot have two member methods with the same name and type erasure:

```java
class C<T> {
    T id (T x) {...}
}
class D extends C<String> {
    Object id(Object x) {...}
}
```

This is illegal since D.id(Object) is a member of D, C<String>.id(String) is declared in a supertype of D, and:

- The two methods have the same name, id
- C<String>.id(String) is accessible to D
- The signature of D.id(Object) is not a subsignature of that of C<String>.id(String)
- The two methods have the same erasure

Two different methods of a class may not override methods with the same erasure:

```java
class C<T> {
    T id(T x) {...}
}
interface I<T> {
    T id(T x);
}
class D extends C<String> implements I<Integer> {
    public String  id(String x) {...}
    public Integer id(Integer x) {...}
}
```
This is also illegal, since `D.id(String)` is a member of `D`, `D.id(Integer)` is declared in `D`, and:

- The two methods have the same name, `id`
- `D.id(Integer)` is accessible to `D`
- The two methods have different signatures (and neither is a subsignature of the other)
- `D.id(String)` overrides `C<String>.id(String)` and `D.id(Integer)` overrides `I.id(Integer)` yet the two overridden methods have the same erasure

### 8.4.8.4 Inheriting Methods with Override-Equivalent Signatures

It is possible for a class to inherit multiple methods with override-equivalent signatures (§8.4.2).

It is a compile-time error if a class `C` inherits a concrete method whose signature is override-equivalent with another method inherited by `C`.

It is a compile-time error if a class `C` inherits a default method whose signature is override-equivalent with another method inherited by `C`, unless there exists an `abstract` method declared in a superclass of `C` and inherited by `C` that is override-equivalent with the two methods.

This exception to the strict default-abstract and default-default conflict rules is made when an `abstract` method is declared in a superclass: the assertion of abstract-ness coming from the superclass hierarchy essentially trumps the default method, making the default method act as if it were `abstract`. However, the `abstract` method from a class does not override the default method(s), because interfaces are still allowed to refine the `signature` of the `abstract` method coming from the class hierarchy.

Note that the exception does not apply if all override-equivalent `abstract` methods inherited by `C` were declared in interfaces.

Otherwise, the set of override-equivalent methods consists of at least one `abstract` method and zero or more default methods; then the class is necessarily an `abstract` class and is considered to inherit all the methods.

One of the inherited methods must be return-type-substitutable for every other inherited method; otherwise, a compile-time error occurs. (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration is inherited from an interface. This fact causes no difficulty and never, of itself, results in a compile-time error.
8.4 Method Declarations

8.4.9 Overloading

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be overloaded.

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the throws clauses of two methods with the same name, unless their signatures are override-equivalent.

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked (§15.12.2). If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup (§15.12.4).

Example 8.4.9-1. Overloading

```java
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+","+y+");"; }
}
```

Here, the class `Point` has two members that are methods with the same name, `move`. The overloaded `move` method of class `Point` chosen for any particular method invocation is determined at compile time by the overloading resolution procedure given in §15.12.

In total, the members of the class `Point` are the `float` instance variables `x` and `y` declared in `Point`, the two declared `move` methods, the declared `toString` method, and the members that `Point` inherits from its implicit direct superclass `Object` (§4.3.2), such as the method `hashCode`. Note that `Point` does not inherit the `toString` method of class `Object` because that method is overridden by the declaration of the `toString` method in class `Point`.

Example 8.4.9-2. Overloading, Overriding, and Hiding

```java
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
}
```
Here, the class RealPoint hides the declarations of the int instance variables \( x \) and \( y \) of class Point with its own float instance variables \( x \) and \( y \), and overrides the method \textit{move} of class Point with its own \textit{move} method. It also overloads the name \textit{move} with another method with a different signature (§8.4.2).

In this example, the members of the class RealPoint include the instance variable \textit{color} inherited from the class Point, the float instance variables \( x \) and \( y \) declared in RealPoint, and the two \textit{move} methods declared in RealPoint.

Which of these overloaded \textit{move} methods of class RealPoint will be chosen for any particular method invocation will be determined at compile time by the overloading resolution procedure described in §15.12.

This following program is an extended variation of the preceding program:

```java
class Point {
    int x = 0, y = 0, color;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
}
```

Here, the class Point provides methods \texttt{getX} and \texttt{getY} that return the values of its fields \( x \) and \( y \); the class RealPoint then overrides these methods by declaring methods with the same signature. The result is two errors at compile time, one for each method, because the return types do not match; the methods in class Point return values of type \texttt{int}, but the wanna-be overriding methods in class RealPoint return values of type \texttt{float}.

This program corrects the errors of the preceding program:

```java
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    int getX() { return (int)Math.floor(x); }
}
```
int getY() { return (int)Math.floor(y); }
}

Here, the overriding methods `getX` and `getY` in class `RealPoint` have the same return types as the methods of class `Point` that they override, so this code can be successfully compiled.

Consider, then, this test program:

class Test {
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        Point p = rp;
        rp.move(1.71828f, 4.14159f);
        p.move(1, -1);
        show(p.x, p.y);
        show(rp.x, rp.y);
        show(p.getX(), p.getY());
        show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
        System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
        System.out.println("(" + x + ", " + y + ")");
    }
}

The output from this program is:

(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)

The first line of output illustrates the fact that an instance of `RealPoint` actually contains the two integer fields declared in class `Point`; it is just that their names are hidden from code that occurs within the declaration of class `RealPoint` (and those of any subclasses it might have). When a reference to an instance of class `RealPoint` in a variable of type `Point` is used to access the field `x`, the integer field `x` declared in class `Point` is accessed. The fact that its value is zero indicates that the method invocation `p.move(1, -1)` did not invoke the method `move` of class `Point`; instead, it invoked the overriding method `move` of class `RealPoint`.

The second line of output shows that the field access `rp.x` refers to the field `x` declared in class `RealPoint`. This field is of type `float`, and this second line of output accordingly displays floating-point values. Incidentally, this also illustrates the fact that the method name `show` is overloaded; the types of the arguments in the method invocation dictate which of the two definitions will be invoked.
The last two lines of output show that the method invocations `p.getX()` and `rp.getX()` each invoke the `getX` method declared in class `RealPoint`. Indeed, there is no way to invoke the `getX` method of class `Point` for an instance of class `RealPoint` from outside the body of `RealPoint`, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding.

### 8.5 Member Type Declarations

A **member class** is a class whose declaration is directly enclosed in the body of another class or interface declaration (§8.1.6, §9.1.4).

A **member interface** is an interface whose declaration is directly enclosed in the body of another class or interface declaration (§8.1.6, §9.1.4).

The accessibility of a member type declaration in a class is specified by its access modifier, or by §6.6 if lacking an access modifier.

It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration in a class, or if a member type declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6.)

The scope and shadowing of a member type is specified in §6.3 and §6.4.

If a class declares a member type with a certain name, then the declaration of that type is said to **hide** any and all accessible declarations of member types with the same name in superclasses and superinterfaces of the class.

In this respect, hiding of member types is similar to hiding of fields (§8.3).

A class inherits from its direct superclass and direct superinterfaces all the non-private member types of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

It is possible for a class to inherit more than one member type with the same name, either from its superclass and superinterfaces or from its superinterfaces alone. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such member type by its simple name will result in a compile-time error, because the reference is ambiguous.

There might be several paths by which the same member type declaration is inherited from an interface. In such a situation, the member type is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.
8.5.1 Static Member Type Declarations

The static keyword may modify the declaration of a member type \( C \) within the body of a non-inner class or interface \( T \). Its effect is to declare that \( C \) is not an inner class. Just as a static method of \( T \) has no current instance of \( T \) in its body, \( C \) also has no current instance of \( T \), nor does it have any lexically enclosing instances.

It is a compile-time error if a static class contains a usage of a non-static member of an enclosing class.

A member interface is implicitly static (§9.1.1). It is permitted for the declaration of a member interface to redundantly specify the static modifier.

8.6 Instance Initializers

An instance initializer declared in a class is executed when an instance of the class is created (§12.5, §15.9, §8.8.7.1).

\[
\text{InstanceInitializer:}
\]
\[
\text{Block}
\]

It is a compile-time error if an instance initializer cannot complete normally (§14.21).

It is a compile-time error if a return statement (§14.17) appears anywhere within an instance initializer.

An instance initializer is permitted to refer to the current object using the keyword this (§15.8.3) or the keyword super (§15.11.2, §15.12), and to use any type variables in scope.

Restrictions on how an instance initializer may refer to instance variables, even when the instance variables are in scope, are specified in §8.3.3).

Exception checking for an instance initializer is specified in §11.2.3.

8.7 Static Initializers

A static initializer declared in a class is executed when the class is initialized (§12.4.2). Together with any field initializers for class variables (§8.3.2), static initializers may be used to initialize the class variables of the class.
**StaticInitializer:**

static Block

It is a compile-time error if a static initializer cannot complete normally (§14.21).

It is a compile-time error if a return statement (§14.17) appears anywhere within a static initializer.

It is a compile-time error if the keyword this (§15.8.3) or the keyword super (§15.11, §15.12) or any type variable declared outside the static initializer, appears anywhere within a static initializer.

Restrictions on how a static initializer may refer to class variables, even when the class variables are in scope, are specified in §8.3.3.

Exception checking for a static initializer is specified in §11.2.3.

### 8.8 Constructor Declarations

A constructor is used in the creation of an object that is an instance of a class (§12.5, §15.9).

**ConstructorDeclaration:**

{ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

**ConstructorDeclarator:**

[TypeParameters] SimpleName ( [ReceiverParameter , ] [FormalParameterList] )

**SimpleName:**

TypeIdentifier

The rules in this section apply to constructors in all class declarations, including enum declarations. However, special rules apply to enum declarations with regard to constructor modifiers, constructor bodies, and default constructors; these rules are stated in §8.9.2.

The SimpleName in the ConstructorDeclarator must be the simple name of the class that contains the constructor declaration, or a compile-time error occurs.

In all other respects, a constructor declaration looks just like a method declaration that has no result (§8.4.5).
Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

Constructors are invoked by class instance creation expressions (§15.9), by the conversions and concatenations caused by the string concatenation operator + (§15.18.1), and by explicit constructor invocations from other constructors (§8.8.7). Access to constructors is governed by access modifiers (§6.6), so it is possible to prevent class instantiation by declaring an inaccessible constructor (§8.8.10).

Constructors are never invoked by method invocation expressions (§15.12).

Example 8.8-1. Constructor Declarations

```java
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

8.8.1 Formal Parameters

The formal parameters of a constructor are identical in syntax and semantics to those of a method (§8.4.1).

The constructor of a non-private inner member class implicitly declares, as the first formal parameter, a variable representing the immediately enclosing instance of the class (§15.9.2, §15.9.3).

The rationale for why only this kind of class has an implicitly declared constructor parameter is subtle. The following explanation may be helpful:

1. In a class instance creation expression for a non-private inner member class, §15.9.2 specifies the immediately enclosing instance of the member class. The member class may have been emitted by a compiler which is different than the compiler of the class instance creation expression. Therefore, there must be a standard way for the compiler of the creation expression to pass a reference (representing the immediately enclosing instance) to the member class's constructor. Consequently, the Java programming language deems in this section that a non-private inner member class's constructor implicitly declares an initial parameter for the immediately enclosing instance. §15.9.3 specifies that the instance is passed to the constructor.

2. In a class instance creation expression for a local class (not in a static context) or anonymous class, §15.9.2 specifies the immediately enclosing instance of the local/anonymous class. The local/anonymous class is necessarily emitted by the same compiler as the class instance creation expression. That compiler can represent the immediately enclosing instance however it wishes. There is no need for the Java programming language to implicitly declare a parameter in the local/anonymous class's constructor.
3. In a class instance creation expression for an anonymous class, and where the anonymous class's superclass is either inner or local (not in a static context), §15.9.2 specifies the anonymous class's immediately enclosing instance with respect to the superclass. This instance must be transmitted from the anonymous class to its superclass, where it will serve as the immediately enclosing instance. Since the superclass may have been emitted by a compiler which is different than the compiler of the class instance creation expression, it is necessary to transmit the instance in a standard way, by passing it as the first argument to the superclass's constructor. Note that the anonymous class itself is necessarily emitted by the same compiler as the class instance creation expression, so it would be possible for the compiler to transmit the immediately enclosing instance with respect to the superclass to the anonymous class however it wishes, before the anonymous class passes the instance to the superclass's constructor. However, for consistency, the Java programming language deems in §15.9.5.1 that, in some circumstances, an anonymous class's constructor implicitly declares an initial parameter for the immediately enclosing instance with respect to the superclass.

The fact that a non-private inner member class may be accessed by a different compiler than compiled it, whereas a local or anonymous class is always accessed by the same compiler that compiled it, explains why the binary name of a non-private inner member class is defined to be predictable but the binary name of a local or anonymous class is not (§13.1).

### 8.8.2 Constructor Signature

It is a compile-time error to declare two constructors with override-equivalent signatures (§8.4.2) in a class.

It is a compile-time error to declare two constructors whose signatures have the same erasure (§4.6) in a class.

### 8.8.3 Constructor Modifiers

(ConstructorModifier:

(one of)

Annotation public protected private)

The rules for annotation modifiers on a constructor declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier in a constructor declaration, or if a constructor declaration has more than one of the access modifiers public, protected, and private (§6.6).

In a normal class declaration, a constructor declaration with no access modifiers has package access.
If two or more (distinct) method modifiers appear in a method declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for MethodModifier.

Unlike methods, a constructor cannot be abstract, static, final, native, strictfp, or synchronized:

• A constructor is not inherited, so there is no need to declare it final.
• An abstract constructor could never be implemented.
• A constructor is always invoked with respect to an object, so it makes no sense for a constructor to be static.
• There is no practical need for a constructor to be synchronized, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work.
• The lack of native constructors is an arbitrary language design choice that makes it easy for an implementation of the Java Virtual Machine to verify that superclass constructors are always properly invoked during object creation.
• The inability to declare a constructor as strictfp (in contrast to a method (§8.4.3)) is an intentional language design choice; it effectively ensures that a constructor is FP-strict if and only if its class is FP-strict (§15.4).

### 8.8.4 Generic Constructors

A constructor is generic if it declares one or more type variables (§4.4).

These type variables are known as the type parameters of the constructor. The form of the type parameter section of a generic constructor is identical to the type parameter section of a generic class (§8.1.2).

It is possible for a constructor to be generic independently of whether the class the constructor is declared in is itself generic.

A generic constructor declaration defines a set of constructors, one for each possible invocation of the type parameter section by type arguments. Type arguments may not need to be provided explicitly when a generic constructor is invoked, as they can often be inferred (§18 (Type Inference)).

The scope and shadowing of a constructor's type parameter is specified in §6.3 and §6.4.

### 8.8.5 Constructor Throws

The throws clause for a constructor is identical in structure and behavior to the throws clause for a method (§8.4.6).
### 8.8.6 The Type of a Constructor

The type of a constructor consists of its signature and the exception types given by its `throws` clause.

### 8.8.7 Constructor Body

The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass (§8.8.7.1).

`ConstructorBody:`

\[
\begin{cases}
\text{[ExplicitConstructorInvocation]} \ [\text{BlockStatements}]
\end{cases}
\]

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving `this`.

If a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class `Object`, then the constructor body implicitly begins with a superclass constructor invocation "`super();`", an invocation of the constructor of its direct superclass that takes no arguments.

Except for the possibility of explicit constructor invocations, and the prohibition on explicitly returning a value (§14.17), the body of a constructor is like the body of a method (§8.4.7).

A `return` statement (§14.17) may be used in the body of a constructor if it does not include an expression.

#### Example 8.8.7-1. Constructor Bodies

```java
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}

class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, WHITE);
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```
Here, the first constructor of ColoredPoint invokes the second, providing an additional argument; the second constructor of ColoredPoint invokes the constructor of its superclass Point, passing along the coordinates.

8.8.7.1 Explicit Constructor Invocations

ExplicitConstructorInvocation:

\[
[\text{TypeArguments}] \text{this} (\ [\text{ArgumentList}] \ ) \ ;
\]
\[
[\text{TypeArguments}] \text{super} (\ [\text{ArgumentList}] \ ) \ ;
\]
\[
\text{ExpressionName} . [\text{TypeArguments}] \text{super} (\ [\text{ArgumentList}] \ ) \ ;
\]
\[
\text{Primary} . [\text{TypeArguments}] \text{super} (\ [\text{ArgumentList}] \ ) \ ;
\]

The following productions from §4.5.1 and §15.12 are shown here for convenience:

TypeArguments:

\[
< \text{TypeArgumentList} >
\]

ArgumentList:

\[
\text{Expression} \ {,} \ \text{Expression}
\]

Explicit constructor invocation statements are divided into two kinds:

- **Alternate constructor invocations** begin with the keyword this (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.

- **Superclass constructor invocations** begin with either the keyword super (possibly prefaced with explicit type arguments) or a Primary expression or an ExpressionName. They are used to invoke a constructor of the direct superclass. They are further divided:
  - **Unqualified superclass constructor invocations** begin with the keyword super (possibly prefaced with explicit type arguments).
  - **Qualified superclass constructor invocations** begin with a Primary expression or an ExpressionName. They allow a subclass constructor to explicitly specify the newly created object's immediately enclosing instance with respect to the direct superclass (§8.1.3). This may be necessary when the superclass is an inner class.

An explicit constructor invocation statement in a constructor body may not refer to any instance variables or instance methods or inner classes declared in this class or any superclass, or use this or super in any expression; otherwise, a compile-time error occurs.
This prohibition on using the current instance explains why an explicit constructor invocation statement is deemed to occur in a static context (§8.1.3).

If `TypeArguments` is present to the left of `this` or `super`, then it is a compile-time error if any of the type arguments are wildcards (§4.5.1).

Let `c` be the class being instantiated, and let `s` be the direct superclass of `c`.

If a superclass constructor invocation statement is unqualified, then:

- If `s` is an inner member class, but `s` is not a member of a class enclosing `c`, then a compile-time error occurs.

If a superclass constructor invocation statement is qualified, then:

- If `s` is not an inner class, or if the declaration of `s` occurs in a static context, then a compile-time error occurs.

- Otherwise, let `p` be the `Primary` expression or the `ExpressionName` immediately preceding ".super", and let `o` be the immediately enclosing class of `s`. It is a compile-time error if the type of `p` is not `o` or a subclass of `o`, or if the type of `p` is not accessible (§6.6).

The exception types that an explicit constructor invocation statement can throw are specified in §11.2.2.

Evaluation of an alternate constructor invocation statement proceeds by first evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

Evaluation of a superclass constructor invocation statement proceeds as follows:

1. Let `i` be the instance being created. The immediately enclosing instance of `i` with respect to `s` (if any) must be determined:

   - If `s` is not an inner class, or if the declaration of `s` occurs in a static context, then no immediately enclosing instance of `i` with respect to `s` exists.

   - If the superclass constructor invocation is unqualified, then `s` is necessarily a local class or an inner member class.

   If `s` is a local class, then let `o` be the immediately enclosing type declaration of `s`.

   If `s` is an inner member class, then let `o` be the innermost enclosing class of `c` of which `s` is a member.

   Let `n` be an integer (`n ≥ 1`) such that `o` is the `n`'th lexically enclosing type declaration of `c`. 
The immediately enclosing instance of \( i \) with respect to \( S \) is the \( n \)'th lexically enclosing instance of \( \text{this} \).

While it may be the case that \( S \) is a member of \( C \) due to inheritance, the zeroth lexically enclosing instance of \( \text{this} \) (that is, \( \text{this itself} \)) is never used as the immediately enclosing instance of \( i \) with respect to \( S \).

- If the superclass constructor invocation is qualified, then the Primary expression or the ExpressionName immediately preceding ".super", \( p \), is evaluated.

  If \( p \) evaluates to null, a NullPointerException is raised, and the superclass constructor invocation completes abruptly.

  Otherwise, the result of this evaluation is the immediately enclosing instance of \( i \) with respect to \( S \).

2. After determining the immediately enclosing instance of \( i \) with respect to \( S \) (if any), evaluation of the superclass constructor invocation statement proceeds by evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

3. Finally, if the superclass constructor invocation statement completes normally, then all instance variable initializers of \( C \) and all instance initializers of \( C \) are executed. If an instance initializer or instance variable initializer \( I \) textually precedes another instance initializer or instance variable initializer \( J \), then \( I \) is executed before \( J \).

Execution of instance variable initializers and instance initializers is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided implicitly. (An alternate constructor invocation does not perform this additional implicit execution.)

Example 8.8.7.1-1. Restrictions on Explicit Constructor Invocation Statements

If the first constructor of ColoredPoint in the example from §8.8.7 were changed as follows:

```java
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
```


then a compile-time error would occur, because the instance variable color cannot be used by an explicit constructor invocation statement.

Example 8.8.7.1-2. Qualified Superclass Constructor Invocation

In the code below, ChildOfInner has no lexically enclosing type declaration, so an instance of ChildOfInner has no enclosing instance. However, the superclass of ChildOfInner (Inner) has a lexically enclosing type declaration (Outer), and an instance of Inner must have an enclosing instance of Outer. The enclosing instance of Outer is set when an instance of Inner is created. Therefore, when we create an instance of ChildOfInner, which is implicitly an instance of Inner, we must provide the enclosing instance of Outer via a qualified superclass invocation statement in ChildOfInner’s constructor. The instance of Outer is called the immediately enclosing instance of ChildOfInner with respect to Inner.

class Outer {
    class Inner {}  
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner() { (new Outer()).super(); }
}

Perhaps surprisingly, the same instance of Outer may serve as the immediately enclosing instance of ChildOfInner with respect to Inner for multiple instances of ChildOfInner. These instances of ChildOfInner are implicitly linked to the same instance of Outer. The program below achieves this by passing an instance of Outer to the constructor of ChildOfInner, which uses the instance in a qualified superclass constructor invocation statement. The rules for an explicit constructor invocation statement do not prohibit using formal parameters of the constructor that contains the statement.

class Outer {
    int secret = 5;
    class Inner {
        int getSecret() { return secret; }
        void setSecret(int s) { secret = s; }
    }
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner(Outer x) { x.super(); }
}

public class Test {
    public static void main(String[] args) {
Outer x = new Outer();
ChildOfInner a = new ChildOfInner(x);
ChildOfInner b = new ChildOfInner(x);
System.out.println(b.getSecret());
a.setSecret(6);
System.out.println(b.getSecret());
}
}

This program produces the output:

5
6

The effect is that manipulation of instance variables in the common instance of Outer is visible through references to different instances of ChildOfInner, even though such references are not aliases in the conventional sense.

8.8.8 Constructor Overloading

Overloading of constructors is identical in behavior to overloading of methods (§8.4.9). The overloading is resolved at compile time by each class instance creation expression (§15.9).

8.8.9 Default Constructor

If a class contains no constructor declarations, then a default constructor is implicitly declared. The form of the default constructor for a top level class, member class, or local class is as follows:

• The default constructor has the same access modifier as the class, unless the class lacks an access modifier, in which case the default constructor has package access (§6.6).

• The default constructor has no formal parameters, except in a non-private inner member class, where the default constructor implicitly declares one formal parameter representing the immediately enclosing instance of the class (§8.8.1, §15.9.2, §15.9.3).

• The default constructor has no throws clauses.

• If the class being declared is the primordial class Object, then the default constructor has an empty body. Otherwise, the default constructor simply invokes the superclass constructor with no arguments.

The form of the default constructor for an anonymous class is specified in §15.9.5.1.
It is a compile-time error if a default constructor is implicitly declared but the superclass does not have an accessible constructor that takes no arguments and has no throws clause.

Example 8.8.9-1. Default Constructors

The declaration:

```java
public class Point {
    int x, y;
}
```

is equivalent to the declaration:

```java
public class Point {
    int x, y;
    public Point() { super(); }
}
```

where the default constructor is public because the class Point is public.

Example 8.8.9-2. Accessibility of Constructors v. Classes

The rule that the default constructor of a class has the same accessibility as the class itself is simple and intuitive. Note, however, that this does not imply that the constructor is accessible whenever the class is accessible. Consider:

```java
package p1;
public class Outer {
    protected class Inner {}
}
package p2;
class SonOfOuter extends p1.Outer {
    void foo() {
        new Inner();  // compile-time access error
    }
}
```

The default constructor for Inner is protected. However, the constructor is protected relative to Inner, while Inner is protected relative to Outer. So, Inner is accessible in SonOfOuter, since it is a subclass of Outer. Inner's constructor is not accessible in SonOfOuter, because the class SonOfOuter is not a subclass of Inner! Hence, even though Inner is accessible, its default constructor is not.
8.9.10 Preventing Instantiation of a Class

A class can be designed to prevent code outside the class declaration from creating instances of the class by declaring at least one constructor, to prevent the creation of a default constructor, and by declaring all constructors to be private (§6.6.1).

A public class can likewise prevent the creation of instances outside its package by declaring at least one constructor, to prevent creation of a default constructor with public access, and by declaring no constructor that is public or protected (§6.6.2).

Example 8.8.10-1. Preventing Instantiation via Constructor Accessibility

class ClassOnly {
    private ClassOnly() { }
    static String just = "only the lonely";
}

Here, the class ClassOnly cannot be instantiated, while in the following code:

class PackageOnly {
    PackageOnly() { }
    String[] justDesserts = { "cheesecake", "ice cream" };  
}

class just {
    package just;
    public class PackageOnly {
        PackageOnly() { }
        String[] justDesserts = { "cheesecake", "ice cream" };
    }
}

The public class PackageOnly can be instantiated only within the package just, in which it is declared. This restriction would also apply if the constructor of PackageOnly was protected, although in that case, it would be possible for code in other packages to instantiate subclasses of PackageOnly.

8.9 Enum Types

An enum declaration specifies a new enum type, a special kind of class type.

EnumDeclaration:
{ClassModifier} enum TypeIdentifier [Superinterfaces] EnumBody

It is a compile-time error if an enum declaration has the modifier abstract or final.

An enum declaration is implicitly final unless it contains at least one enum constant that has a class body (§8.9.1).

A nested enum type is implicitly static. It is permitted for the declaration of a nested enum type to redundantly specify the static modifier.
This implies that it is impossible to declare an enum type in the body of an inner class (§8.1.3), because an inner class cannot have static members except for constant variables.

It is a compile-time error if the same keyword appears more than once as a modifier for an enum declaration, or if an enum declaration has more than one of the access modifiers public, protected, and private (§6.6).

The direct superclass of an enum type \( E \) is \( \text{Enum}<E> \) (§8.1.4).

An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1).

In addition to the compile-time error, three further mechanisms ensure that no instances of an enum type exist beyond those defined by its enum constants:

- The final clone method in Enum ensures that enum constants can never be cloned.
- Reflective instantiation of enum types is prohibited.
- Special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization.

### 8.9.1 Enum Constants

The body of an enum declaration may contain enum constants. An enum constant defines an instance of the enum type.

\[
\text{EnumBody:} \\
\{ \text{[EnumConstantList]} [,] \text{[EnumBodyDeclarations]} \}
\]

\[
\text{EnumConstantList:} \\
\text{EnumConstant} [, \text{EnumConstant}}
\]

\[
\text{EnumConstant:} \\
\{\text{[EnumConstantModifier]} \text{Identifier} [\{ [\text{ArgumentList}] \} ] \text{[ClassBody]} \}
\]

\[
\text{EnumConstantModifier:} \\
\text{Annotation}
\]

The following production from §15.12 is shown here for convenience:

\[
\text{ArgumentList:} \\
\text{Expression} [, \text{Expression}]
\]

The rules for annotation modifiers on an enum constant declaration are specified in §9.7.4 and §9.7.5.
The Identifier in a EnumConstant may be used in a name to refer to the enum constant.

The scope and shadowing of an enum constant is specified in §6.3 and §6.4.

An enum constant may be followed by arguments, which are passed to the constructor of the enum when the constant is created during class initialization as described later in this section. The constructor to be invoked is chosen using the normal rules of overload resolution (§15.12.2). If the arguments are omitted, an empty argument list is assumed.

The optional class body of an enum constant implicitly defines an anonymous class declaration (§15.9.5) that extends the immediately enclosing enum type. The class body is governed by the usual rules of anonymous classes; in particular it cannot contain any constructors. Instance methods declared in these class bodies may be invoked outside the enclosing enum type only if they override accessible methods in the enclosing enum type (§8.4.8).

It is a compile-time error for the class body of an enum constant to declare an abstract method.

Because there is only one instance of each enum constant, it is permitted to use the == operator in place of the equals method when comparing two object references if it is known that at least one of them refers to an enum constant.

The equals method in Enum is a final method that merely invokes super.equals on its argument and returns the result, thus performing an identity comparison.

### 8.9.2 Enum Body Declarations

In addition to enum constants, the body of an enum declaration may contain constructor and member declarations as well as instance and static initializers.

*EnumBodyDeclarations:*

```
; {ClassBodyDeclaration}
```

The following productions from §8.1.6 are shown here for convenience:

*ClassBodyDeclaration:*

```
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration
```
Any constructor or member declarations in the body of an enum declaration apply to the enum type exactly as if they had been present in the body of a normal class declaration, unless explicitly stated otherwise.

It is a compile-time error if a constructor declaration in an enum declaration is public or protected (§6.6).

It is a compile-time error if a constructor declaration in an enum declaration contains a superclass constructor invocation statement (§8.8.7.1).

It is a compile-time error to refer to a static field of an enum type from a constructor, instance initializer, or instance variable initializer of the enum type, unless the field is a constant variable (§4.12.4).

In an enum declaration, a constructor declaration with no access modifiers is private.

In an enum declaration with no constructor declarations, a default constructor is implicitly declared. The default constructor is private, has no formal parameters, and has no throws clause.

In practice, a compiler is likely to mirror the Enum type by declaring String and int parameters in the default constructor of an enum type. However, these parameters are not specified as "implicitly declared" because different compilers do not need to agree on the form of the default constructor. Only the compiler of an enum type knows how to instantiate the enum constants; other compilers can simply rely on the implicitly declared public static fields of the enum type (§8.9.3) without regard for how those fields were initialized.

It is a compile-time error if an enum declaration $E$ has an abstract method $m$ as a member, unless $E$ has at least one enum constant and all of $E$’s enum constants have class bodies that provide concrete implementations of $m$.

It is a compile-time error for an enum declaration to declare a finalizer (§12.6). An instance of an enum type may never be finalized.

**Example 8.9.2-1. Enum Body Declarations**

```java
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }
    private final int value;
}
```
public int value() { return value; }
}

Each enum constant arranges for a different value in the field `value`, passed in via a constructor. The field represents the value, in cents, of an American coin. Note that there are no restrictions on the parameters that may be declared by an enum type's constructor.

**Example 8.9.2-2. Restriction On Enum Constant Self-Reference**

Without the rule on `static` field access, apparently reasonable code would fail at run time due to the initialization circularity inherent in enum types. (A circularity exists in any class with a "self-typed" `static` field.) Here is an example of the sort of code that would fail:

```java
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;
    Color() { colorMap.put(toString(), this); }

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
}
```

Static initialization of this enum would throw a `NullPointerException` because the `static` variable `colorMap` is uninitialized when the constructors for the enum constants run. The restriction above ensures that such code cannot be compiled. However, the code can easily be refactored to work properly:

```java
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
    static {
        for (Color c : Color.values())
            colorMap.put(c.toString(), c);
    }
}
```

The refactored version is clearly correct, as static initialization occurs top to bottom.

### 8.9.3 Enum Members

The members of an enum type $E$ are all of the following:

- Members declared in the body of the declaration of $E$. 

• Members inherited from `Enum<E>`.

• For each enum constant `c` declared in the body of the declaration of `E`, `E` has an implicitly declared `public static final` field of type `E` that has the same name as `c`. The field has a variable initializer which instantiates `E` and passes any arguments of `c` to the constructor chosen for `E`. The field has the same annotations as `c` (if any).

These fields are implicitly declared in the same order as the corresponding enum constants, before any `static` fields explicitly declared in the body of the declaration of `E`.

An enum constant is said to be *created* when the corresponding implicitly declared field is initialized.

• The following implicitly declared methods:

```java
/**
 * Returns an array containing the constants of this enum
type, in the order they're declared. This method may be
* used to iterate over the constants as follows:
 * 
 *     for(E c : E.values())
 *         System.out.println(c);
 * 
 * @return an array containing the constants of this enum
type, in the order they're declared
 */
public static E[] values();

/**
 * Returns the enum constant of this type with the specified
* name. The string must match exactly an identifier used to declare
* an enum constant in this type. (Extraneous whitespace
* characters are not permitted.)
 * 
 * @return the enum constant with the specified name
 * @throws IllegalArgumentException if this enum type has no
* constant with the specified name
 */
public static E valueOf(String name);
```

It follows that the declaration of enum type `E` cannot contain fields that conflict with the implicitly declared fields corresponding to `E`'s enum constants, nor contain methods that conflict with implicitly declared methods or override `final` methods of class `Enum<E>`.

**Example 8.9.3-1. Iterating Over Enum Constants With An Enhanced for Loop**

```java
public class Test {
    enum Season { WINTER, SPRING, SUMMER, FALL }
}
Example 8.9.3-2. Switching Over Enum Constants

A switch statement (§14.11) is useful for simulating the addition of a method to an enum type from outside the type. This example "adds" a color method to the Coin type from §8.9.2, and prints a table of coins, their values, and their colors.

class Test {
    enum CoinColor { COPPER, NICKEL, SILVER }
    static CoinColor color(Coin c) {
        switch (c) {
            case PENNY:
                return CoinColor.COPPER;
            case NICKEL:
                return CoinColor.NICKEL;
            case DIME: case QUARTER:
                return CoinColor.SILVER;
            default:
                throw new AssertionError("Unknown coin: " + c);
        }
    }
    public static void main(String[] args) {
        for (Coin c : Coin.values())
            System.out.println(c + "	" + color(c) + "	" + c.value());
    }
}

This program produces the output:

<table>
<thead>
<tr>
<th>Coin</th>
<th>Value</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>PENNY</td>
<td>1</td>
<td>COPPER</td>
</tr>
<tr>
<td>NICKEL</td>
<td>5</td>
<td>NICKEL</td>
</tr>
<tr>
<td>DIME</td>
<td>10</td>
<td>SILVER</td>
</tr>
<tr>
<td>QUARTER</td>
<td>25</td>
<td>SILVER</td>
</tr>
</tbody>
</table>
Example 8.9.3-3. Enum Constants with Class Bodies

    enum Operation {
        PLUS {
            double eval(double x, double y) { return x + y; }
        },
        MINUS {
            double eval(double x, double y) { return x - y; }
        },
        TIMES {
            double eval(double x, double y) { return x * y; }
        },
        DIVIDED_BY {
            double eval(double x, double y) { return x / y; }
        }
    // Each constant supports an arithmetic operation
    abstract double eval(double x, double y);
    
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
    }
    
Class bodies attach behaviors to the enum constants. The program produces the output:

    java Operation 2.0 4.0
    2.0 PLUS 4.0 = 6.0
    2.0 MINUS 4.0 = -2.0
    2.0 TIMES 4.0 = 8.0
    2.0 DIVIDED_BY 4.0 = 0.5

This pattern is much safer than using a switch statement in the base type (Operation), as the pattern precludes the possibility of forgetting to add a behavior for a new constant (since the enum declaration would cause a compile-time error).

Example 8.9.3-4. Multiple Enum Types

In the following program, a playing card class is built atop two simple enums.

    import java.util.List;
    import java.util.ArrayList;
    class Card implements Comparable<Card>, java.io.Serializable {
        public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
8.9 Enum Types

public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

private final Rank rank;
private final Suit suit;
public Rank rank() { return rank; }
public Suit suit() { return suit; }

private Card(Rank rank, Suit suit) {
    if (rank == null || suit == null)
        throw new NullPointerException(rank + ", " + suit);
    this.rank = rank;
    this.suit = suit;
}

public String toString() { return rank + " of " + suit; }

// Primary sort on suit, secondary sort on rank
public int compareTo(Card c) {
    int suitCompare = suit.compareTo(c.suit);
    return (suitCompare != 0 ?
        suitCompare :
        rank.compareTo(c.rank));
}

private static final List<Card> prototypeDeck =
    new ArrayList<Card>(52);

static {
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            prototypeDeck.add(new Card(rank, suit));
}

// Returns a new deck
public static List<Card> newDeck() {
    return new ArrayList<Card>(prototypeDeck);
}

The following program exercises the Card class. It takes two integer parameters on the
command line, representing the number of hands to deal and the number of cards in each
hand:

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
            System.out.println(deck.get(i).toString());
    }
}
System.out.println(dealHand(deck, cardsPerHand));
}

/**
 * Returns a new ArrayList consisting of the last n
 * elements of deck, which are removed from deck.
 * The returned list is sorted using the elements'
 * natural ordering.
 */
public static <E extends Comparable<E>>
ArrayList<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    ArrayList<E> hand = new ArrayList<E>(handView);
    handView.clear();
    Collections.sort(hand);
    return hand;
}

The program produces the output:

java Deal 4 3
[DEUCE of CLUBS, SEVEN of CLUBS, QUEEN of DIAMONDS]
[NINE of HEARTS, FIVE of SPADES, ACE of SPADES]
[THREE of HEARTS, SIX of HEARTS, TEN of SPADES]
[TEN of CLUBS, NINE of DIAMONDS, THREE of SPADES]
AN interface declaration introduces a new reference type whose members are classes, interfaces, constants, and methods. This type has no instance variables, and typically declares one or more abstract methods; otherwise unrelated classes can implement the interface by providing implementations for its abstract methods. Interfaces may not be directly instantiated.

A nested interface is any interface whose declaration occurs within the body of another class or interface.

A top level interface is an interface that is not a nested interface.

We distinguish between two kinds of interfaces - normal interfaces and annotation types.

This chapter discusses the common semantics of all interfaces - normal interfaces, both top level (§7.6) and nested (§8.5, §9.5), and annotation types (§9.6). Details that are specific to particular kinds of interfaces are discussed in the sections dedicated to these constructs.

Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object.

An interface may be declared to be a direct extension of one or more other interfaces, meaning that it inherits all the member types, instance methods, and constants of the interfaces it extends, except for any members that it may override or hide.

A class may be declared to directly implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing a superclass.
A variable whose declared type is an interface type may have as its value a
reference to any instance of a class which implements the specified interface. It is
not sufficient that the class happen to implement all the abstract methods of the
interface; the class or one of its superclasses must actually be declared to implement
the interface, or else the class is not considered to implement the interface.

9.1 Interface Declarations

An interface declaration specifies a new named reference type. There are two
kinds of interface declarations - normal interface declarations and annotation type
declarations (§9.6).

\[
\text{InterfaceDeclaration:} \\
\quad \text{NormalInterfaceDeclaration} \\
\quad \text{AnnotationTypeDeclaration}
\]

\[
\text{NormalInterfaceDeclaration:} \\
\quad \{\text{InterfaceModifier}\} \text{ interface TypeIdentifier [TypeParameters]} \\
\quad \text{[ExtendsInterfaces] InterfaceBody}
\]

The TypeIdentifier in an interface declaration specifies the name of the interface.
It is a compile-time error if an interface has the same simple name as any of its
enclosing classes or interfaces.

The scope and shadowing of an interface declaration is specified in §6.3 and §6.4.

9.1.1 Interface Modifiers

An interface declaration may include interface modifiers.

\[
\text{InterfaceModifier:} \\
\quad \text{(one of)} \\
\quad \text{Annotation public protected private} \\
\quad \text{abstract static strictfp}
\]

The rules for annotation modifiers on an interface declaration are specified in §9.7.4
and §9.7.5.

The access modifier public (§6.6) pertains to every kind of interface declaration.
The access modifiers protected and private pertain only to member interfaces whose declarations are directly enclosed by a class declaration (§8.5.1).

The modifier static pertains only to member interfaces (§8.5.1, §9.5), not to top level interfaces (§7.6).

It is a compile-time error if the same keyword appears more than once as a modifier for an interface declaration, or if a interface declaration has more than one of the access modifiers public, protected, and private (§6.6).

If two or more (distinct) interface modifiers appear in an interface declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for InterfaceModifier.

### 9.1.1.1 abstract Interfaces

Every interface is implicitly abstract.

This modifier is obsolete and should not be used in new programs.

### 9.1.1.2 strictfp Interfaces

The effect of the strictfp modifier is to make all float or double expressions within the interface declaration be explicitly FP-strict (§15.4).

This implies that all methods declared in the interface, and all nested types declared in the interface, are implicitly strictfp.

### 9.1.2 Generic Interfaces and Type Parameters

An interface is generic if it declares one or more type variables (§4.4).

These type variables are known as the type parameters of the interface. The type parameter section follows the interface name and is delimited by angle brackets.

The following productions from §8.1.2 and §4.4 are shown here for convenience:

\[
\text{TypeParameters:} \\
< \text{TypeParameterList} > \\
\text{TypeParameterList:} \\
\quad \text{TypeParameter} \{, \text{TypeParameter}\} \\
\text{TypeParameter:} \\
\quad \{\text{TypeParameterModifier}\} \text{TypeIdentifier} [\text{TypeBound}] \\
\text{TypeParameterModifier:} \\
\quad \text{Annotation}
\]
9.1 Interface Declarations

TypeBound:
  extends TypeVariable
  extends ClassOrInterfaceType {AdditionalBound}

AdditionalBound:
  & InterfaceType

The rules for annotation modifiers on a type parameter declaration are specified in §9.7.4 and §9.7.5.

In an interface's type parameter section, a type variable \( \tau \) directly depends on a type variable \( s \) if \( s \) is the bound of \( \tau \), while \( \tau \) depends on \( s \) if either \( \tau \) directly depends on \( s \) or \( \tau \) directly depends on a type variable \( u \) that depends on \( s \) (using this definition recursively). It is a compile-time error if a type variable in a interface's type parameter section depends on itself.

The scope and shadowing of an interface's type parameter is specified in §6.3.

It is a compile-time error to refer to a type parameter of a generic interface \( I \) anywhere in the declaration of a static member of \( I \) (§9.3, §9.4, §9.5).

A generic interface declaration defines a set of parameterized types (§4.5), one for each possible parameterization of the type parameter section by type arguments. All of these parameterized types share the same interface at run time.

9.1.3 Superinterfaces and Subinterfaces

If an extends clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the member types, instance methods, and constants of each of the other named interfaces.

These other named interfaces are the direct superinterfaces of the interface being declared.

Any class that implements the declared interface is also considered to implement all the interfaces that this interface extends.

ExtendsInterfaces:
  extends InterfaceTypeList

The following production from §8.1.5 is shown here for convenience:

InterfaceTypeList:
  InterfaceType {, InterfaceType}

Each InterfaceType in the extends clause of an interface declaration must name an accessible interface type (§6.6), or a compile-time error occurs.
If an InterfaceType has type arguments, it must denote a well-formed parameterized type (§4.5), and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

Given a (possibly generic) interface declaration $I_{F_1,...,F_n}$ $(n \geq 0)$, the direct superinterfaces of the interface type $I_{F_1,...,F_n}$ are the types given in the extends clause of the declaration of $I$, if an extends clause is present.

Given a generic interface declaration $I_{F_1,...,F_n}$ $(n > 0)$, the direct superinterfaces of the parameterized interface type $I_{T_1,...,T_n}$, where $T_i (1 \leq i \leq n)$ is a type, are all types $J_{U_1 \theta,..,U_k \theta}$, where $J_{U_1,...,U_k}$ is a direct superinterface of $I_{F_1,...,F_n}$ and $\theta$ is the substitution $[F_1 := T_1, \ldots, F_n := T_n]$.

The superinterface relationship is the transitive closure of the direct superinterface relationship. An interface $K$ is a superinterface of interface $I$ if either of the following is true:

- $K$ is a direct superinterface of $I$.
- There exists an interface $J$ such that $K$ is a superinterface of $J$, and $J$ is a superinterface of $I$, applying this definition recursively.

Interface $I$ is said to be a subinterface of interface $K$ whenever $K$ is a superinterface of $I$.

While every class is an extension of class Object, there is no single interface of which all interfaces are extensions.

An interface $I$ directly depends on a type $T$ if $T$ is mentioned in the extends clause of $I$ either as a superinterface or as a qualifier in the fully qualified form of a superinterface name.

An interface $I$ depends on a reference type $T$ if any of the following is true:

- $I$ directly depends on $T$.
- $I$ directly depends on a class $C$ that depends on $T$ (§8.1.5).
- $I$ directly depends on an interface $J$ that depends on $T$ (using this definition recursively).

It is a compile-time error if an interface depends on itself.

If circularly declared interfaces are detected at run time, as interfaces are loaded, then a ClassCircularityError is thrown (§12.2.1).
9.2 Interface Members

The members of an interface type are:

- Members declared in the body of the interface (§9.1.4).
- Members inherited from any direct superinterfaces (§9.1.3).
- If an interface has no direct superinterfaces, then the interface implicitly declares a public abstract member method \( m \) with signature \( s \), return type \( r \), and throws clause \( t \) corresponding to each public instance method \( m \) with signature \( s \), return type \( r \), and throws clause \( t \) declared in Object (§4.3.2), unless an abstract method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.

It is a compile-time error if the interface explicitly declares such a method \( m \) in the case where \( m \) is declared to be final in Object.

It is a compile-time error if the interface explicitly declares a method with a signature that is override-equivalent (§8.4.2) to a public method of Object, but which has a different return type, or an incompatible throws clause, or is not abstract.

The interface inherits, from the interfaces it extends, all members of those interfaces, except for (i) fields, classes, and interfaces that it hides, (ii) abstract
methods and default methods that it overrides (§9.4.1), (iii) private methods, and (iv) static methods.

Fields, methods, and member types of an interface type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5). However, this is discouraged as a matter of style.

9.3 Field (Constant) Declarations

ConstantDeclaration:
\{ConstantModifier\} UnannType VariableDeclaratorList ;  

ConstantModifier:
(one of)  
Annotation public static final

See §8.3 for UnannType. The following productions from §4.3 and §8.3 are shown here for convenience:

VariableDeclaratorList:
VariableDeclarator { , VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [ Dims]

Dims:
{ Annotation} [ ] {Annotation} [ ]

VariableInitializer:
Expression
ArrayInitializer

The rules for annotation modifiers on an interface field declaration are specified in §9.7.4 and §9.7.5.

Every field declaration in the body of an interface is implicitly public, static, and final. It is permitted to redundantly specify any or all of these modifiers for such fields.

It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration.
If two or more (distinct) field modifiers appear in a field declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for *ConstantModifier*.

The declared type of a field is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and is specified by §10.2 otherwise.

The scope and shadowing of an interface field declaration is specified in §6.3 and §6.4.

It is a compile-time error for the body of an interface declaration to declare two fields with the same name.

If the interface declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in superinterfaces of the interface.

It is possible for an interface to inherit more than one field with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to any such field by its simple name will result in a compile-time error, because the reference is ambiguous.

There might be several paths by which the same field declaration is inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

**Example 9.3-1. Ambiguous Inherited Fields**

If two fields with the same name are inherited by an interface because, for example, two of its direct superinterfaces declare fields with that name, then a single ambiguous member results. Any use of this ambiguous member will result in a compile-time error. In the program:

```java
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
interface PrintColors extends BaseColors {
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

the interface *LotsOfColors* inherits two fields named YELLOW. This is all right as long as the interface does not contain any reference by simple name to the field YELLOW. (Such a reference could occur within a variable initializer for a field.)
Even if interface PrintColors were to give the value 3 to YELLOW rather than the value 8, a reference to field YELLOW within interface LotsOfColors would still be considered ambiguous.

Example 9.3-2. Multiply Inherited Fields

If a single field is inherited multiple times from the same interface because, for example, both this interface and one of this interface’s direct superinterfaces extend the interface that declares the field, then only a single member results. This situation does not in itself cause a compile-time error.

In the previous example, the fields RED, GREEN, and BLUE are inherited by interface LotsOfColors in more than one way, through interface RainbowColors and also through interface PrintColors, but the reference to field RED in interface LotsOfColors is not considered ambiguous because only one actual declaration of the field RED is involved.

9.3.1 Initialization of Fields in Interfaces

Every declarator in a field declaration of an interface must have a variable initializer, or a compile-time error occurs.

The initializer need not be a constant expression (§15.28).

It is a compile-time error if the initializer of an interface field uses the simple name of the same field or another field whose declaration occurs textually later in the same interface.

It is a compile-time error if the keyword this (§15.8.3) or the keyword super (§15.11.2, §15.12) occurs in the initializer of an interface field, unless the occurrence is within the body of an anonymous class (§15.9.5).

At run time, the initializer is evaluated and the field assignment performed exactly once, when the interface is initialized (§12.4.2).

Note that interface fields that are constant variables (§4.12.4) are initialized before other interface fields. This also applies to static fields that are constant variables in classes (§8.3.2). Such fields will never be observed to have their default initial values (§4.12.5), even by devious programs.

Example 9.3.1-1. Forward Reference to a Field

```java
interface Test {
    float f = j;
    int j = 1;
    int k = k + 1;
}
```

This program causes two compile-time errors, because j is referred to in the initialization of f before j is declared, and because the initialization of k refers to k itself.
9.4 Method Declarations

InterfaceMethodDeclaration:
{InterfaceMethodModifier} MethodHeader MethodBody

InterfaceMethodModifier:
(one of)
Annotation public private
abstract default static strictfp

The following productions from §8.4, §8.4.5, and §8.4.7 are shown here for convenience:

MethodHeader:
    Result MethodDeclarator [Throws]
    TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
    UnannType
    void

MethodDeclarator:
    Identifier ( [ReceiverParameter , ] [FormalParameterList] ) [Dims]

MethodBody:
    Block
    ;

The rules for annotation modifiers on an interface method declaration are specified in §9.7.4 and §9.7.5.

A method in the body of an interface may be declared public or private (§6.6). If no access modifier is given, the method is implicitly public. It is permitted, but discouraged as a matter of style, to redundantly specify the public modifier for a method declaration in an interface.

A default method is an instance method declared in an interface with the default modifier. Its body is always represented by a block, which provides a default implementation for any class that implements the interface without overriding the method. Default methods are distinct from concrete methods (§8.4.3.1), which are declared in classes, and from private interface methods, which are neither inherited nor overridden.

An interface can declare static methods, which are invoked without reference to a particular object. static interface methods are distinct from default methods, which are instance methods.
It is a compile-time error to use the name of a type parameter of any surrounding declaration in the header or body of a \texttt{static} method of an interface.

The effect of the \texttt{strictfp} modifier is to make all float or double expressions within the body of a default or \texttt{static} method be explicitly FP-strict (§15.4).

An interface method lacking a \texttt{private}, \texttt{default}, or \texttt{static} modifier is implicitly \texttt{abstract}. Its body is represented by a semicolon, not a block. It is permitted, but discouraged as a matter of style, to redundantly specify the \texttt{abstract} modifier for such a method declaration.

Note that an interface method may not be declared with \texttt{protected} or package access, or with the modifiers \texttt{final}, \texttt{synchronized}, or \texttt{native}.

It is a compile-time error if the same keyword appears more than once as a modifier for an interface method declaration, or if an interface method declaration has more than one of the access modifiers \texttt{public} and \texttt{private} (§6.6).

It is a compile-time error if an interface method declaration has more than one of the keywords \texttt{abstract}, \texttt{default}, or \texttt{static}.

It is a compile-time error if an interface method declaration that contains the keyword \texttt{private} also contains the keyword \texttt{abstract} or \texttt{default}. It is permitted for an interface method declaration to contain both \texttt{private} and \texttt{static}.

It is a compile-time error if an interface method declaration that contains the keyword \texttt{abstract} also contains the keyword \texttt{strictfp}.

It is a compile-time error for the body of an interface to declare, explicitly or implicitly, two methods with override-equivalent signatures (§8.4.2). However, an interface may inherit several \texttt{abstract} methods with such signatures (§9.4.1).

A method declared in an interface may be generic. The rules for type parameters of a generic method in an interface are the same as for a generic method in a class (§8.4.4).

\section{Inheritance and Overriding}

An interface \(I\) \textit{inherits} from its direct superinterfaces all \texttt{abstract} and default methods \(m\) for which all of the following are true:

\begin{itemize}
  \item \(m\) is a member of a direct superinterface, \(J\), of \(I\).
  \item No method declared in \(I\) has a signature that is a subsignature (§8.4.2) of the signature of \(m\).
\end{itemize}
• There exists no method \( m' \) that is a member of a direct superinterface, \( \mathcal{J} \), of \( I \) (\( m \) distinct from \( m' \), \( \mathcal{J} \) distinct from \( \mathcal{J}' \)), such that \( m' \) overrides from \( \mathcal{J}' \) the declaration of the method \( m \).

Note that methods are overridden on a signature-by-signature basis. If, for example, an interface declares two public methods with the same name (§9.4.2), and a subinterface overrides one of them, the subinterface still inherits the other method.

The third clause above prevents a subinterface from re-inheriting a method that has already been overridden by another of its superinterfaces. For example, in this program:

```java
interface Top {
    default String name() { return "unnamed"; }
}
interface Left extends Top {
    default String name() { return getClass().getName(); }
}
interface Right extends Top {}
interface Bottom extends Left, Right {}
```

Right inherits name() from Top, but Bottom inherits name() from Left, not Right. This is because name() from Left overrides the declaration of name() in Top.

An interface does not inherit private or static methods from its superinterfaces.

If an interface \( I \) declares a private or static method \( m \), and the signature of \( m \) is a subsignature of a public instance method \( m' \) in a superinterface of \( I \), and \( m' \) would otherwise be accessible to code in \( I \), then a compile-time error occurs.

In essence, a static method in an interface cannot hide an instance method in a superinterface. This is similar to the rule in §8.4.8.2 whereby a static method in a class cannot hide an instance method in a superclass or superinterface. Note that the rule in §8.4.8.2 speaks of a class that "declares or inherits a static method", whereas the rule above speaks only of an interface that "declares a static method", since an interface cannot inherit a static method. Also note that the rule in §8.4.8.2 allows hiding of both instance and static methods in superclasses/superinterfaces, whereas the rule above considers only public instance methods in superinterfaces.

Along the same lines, a private method in an interface cannot override an instance method - whether public or private - in a superinterface. This is similar to the rules in §8.4.8.1 and §8.4.8.3 whereby a private method in a class cannot override any instance method in a superclass or superinterface, because §8.4.8.1 requires the overridden method to be non-private and §8.4.8.3 requires the overriding method to provide at least as much access as the overridden method. In summary, only public methods in interfaces can be overridden, and only by public methods in subinterfaces or in implementing classes.
9.4.1.1 Overriding (by Instance Methods)

An instance method $m_I$ declared in or inherited by interface $I$, overrides from $I$ another instance method $m_J$ declared in interface $J$, iff all of the following are true:

- $I$ is a subinterface of $J$.
- $I$ does not inherit $m_J$.
- The signature of $m_I$ is a subsignature (§8.4.2) of the signature of $m_J$.
- $m_J$ is public.

The presence or absence of the `strictfp` modifier has absolutely no effect on the rules for overriding methods. For example, it is permitted for a method that is not FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

An overridden default method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super` qualified by a superinterface name.

9.4.1.2 Requirements in Overriding

The relationship between the return type of an interface method and the return types of any overridden interface methods is specified in §8.4.8.3.

The relationship between the `throws` clause of an interface method and the `throws` clauses of any overridden interface methods is specified in §8.4.8.3.

The relationship between the signature of an interface method and the signatures of any overridden interface methods is specified in §8.4.8.3.

The relationship between the accessibility of an interface method and the accessibility of any overridden interface methods is specified in §8.4.8.3.

It is a compile-time error if a default method is override-equivalent with a non-private method of the class `Object`, because any class implementing the interface will inherit its own implementation of the method.

The prohibition against declaring one of the `Object` methods as a default method may be surprising. There are, after all, cases like `java.util.List` in which the behavior of `toString` and `equals` are precisely defined. The motivation becomes clearer, however, when some broader design decisions are understood:

- First, methods inherited from a superclass are allowed to override methods inherited from superinterfaces (§8.4.8.1). So, every implementing class would automatically override an interface's `toString` default. This is longstanding behavior in the Java programming language. It is not something we wish to change with the design of default methods, because that would conflict with the goal of allowing interfaces to
unobtrusively evolve, only providing default behavior when a class doesn’t already have it through the class hierarchy.

- Second, interfaces do not inherit from `Object`, but rather implicitly declare many of the same methods as `Object` (§9.2). So, there is no common ancestor for the `toString` declared in `Object` and the `toString` declared in an interface. At best, if both were candidates for inheritance by a class, they would conflict. Working around this problem would require awkward commingling of the class and interface inheritance trees.

- Third, use cases for declaring `Object` methods in interfaces typically assume a linear interface hierarchy; the feature does not generalize very well to multiple inheritance scenarios.

- Fourth, the `Object` methods are so fundamental that it seems dangerous to allow an arbitrary superinterface to silently add a default method that changes their behavior.

An interface is free, however, to define another method that provides behavior useful for classes that override the `Object` methods. For example, the `java.util.List` interface could declare an `elementString` method that produces the string described by the contract of `toString`; implementors of `toString` in classes could then delegate to this method.

### 9.4.1.3 Inheriting Methods with Override-Equivalent Signatures

It is possible for an interface to inherit several methods with override-equivalent signatures (§8.4.2).

If an interface \( I \) inherits a default method whose signature is override-equivalent with another method inherited by \( I \), then a compile-time error occurs. (This is the case whether the other method is abstract or default.)

Otherwise, all the inherited methods are abstract, and the interface is considered to inherit all the methods.

One of the inherited methods must be return-type-substitutable for every other inherited method, or else a compile-time error occurs. (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration is inherited from an interface. This fact causes no difficulty and never, of itself, results in a compile-time error.

Naturally, when two different default methods with matching signatures are inherited by a subinterface, there is a behavioral conflict. We actively detect this conflict and notify the programmer with an error, rather than waiting for the problem to arise when a concrete class is compiled. The error can be avoided by declaring a new method that overrides, and thus prevents the inheritance of, all conflicting methods.

Similarly, when an abstract method and a default method with matching signatures are inherited by a subinterface, we produce an error. In this case, it would be possible to give priority to one or the other - perhaps we would assume that the default method provides a reasonable implementation for the abstract method. But this is risky, since other than
the coincidental name and signature, we have no reason to believe that the default method behaves consistently with the abstract method's contract - the default method may not have even existed when the subinterface was originally developed. It is safer in this situation to ask the user to actively assert that the default implementation is appropriate (via an overriding declaration).

In contrast, the longstanding behavior for inherited concrete methods in classes is that they override abstract methods declared in interfaces (see §8.4.8). The same argument about potential contract violation applies here, but in this case there is an inherent imbalance between classes and interfaces. We prefer, in order to preserve the independent nature of class hierarchies, to minimize class-interface clashes by simply giving priority to concrete methods.

### 9.4.2 Overloading

If two methods of an interface (whether both declared in the same interface, or both inherited by an interface, or one declared and one inherited) have the same name but different signatures that are not override-equivalent (§8.4.2), then the method name is said to be *overloaded*.

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the `throws` clauses of two methods with the same name but different signatures that are not override-equivalent.

**Example 9.4.2-1. Overloading an abstract Method Declaration**

```java
interface PointInterface {
    void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
    void move(float dx, float dy);
    void move(double dx, double dy);
}
```

Here, the method named `move` is overloaded in interface `RealPointInterface` with three different signatures, two of them declared and one inherited. Any non-abstract class that implements interface `RealPointInterface` must provide implementations of all three method signatures.

### 9.4.3 Interface Method Body

A default method has a block body. This block of code provides an implementation of the method in the event that a class implements the interface but does not provide its own implementation of the method.

A `private` or `static` method also has a block body, which provides the implementation of the method.
It is a compile-time error if an interface method declaration is `abstract` (explicitly or implicitly) and has a block for its body.

It is a compile-time error if an interface method declaration is `default`, `private`, or `static`, and has a semicolon for its body.

It is a compile-time error for the body of a `static` method to attempt to reference the current object using the keyword `this` or the keyword `super`.

The rules for `return` statements in a method body are specified in §14.17.

If a method is declared to have a return type (§8.4.5), then a compile-time error occurs if the body of the method can complete normally (§14.1).

### 9.5 Member Type Declarations

Interfaces may contain member type declarations (§8.5).

Every member type declaration in the body of an interface is implicitly `public` and `static`. It is permitted to redundantly specify either or both of these modifiers.

It is a compile-time error if a member type declaration in an interface has the modifier `protected` or `private`.

It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration in an interface.

If an interface declares a member type with a certain name, then the declaration of that type is said to `hide` any and all accessible declarations of member types with the same name in superinterfaces of the interface.

An interface inherits from its direct superinterfaces all the non-private member types of the superinterfaces that are both accessible to code in the interface and not hidden by a declaration in the interface.

It is possible for an interface to inherit more than one member type with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to any such member type by its simple name will result in a compile-time error, because the reference is ambiguous.

There might be several paths by which the same member type declaration is inherited from an interface. In such a situation, the member type is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.
9.6 Annotation Types

An *annotation type declaration* specifies a new *annotation type*, a special kind of interface type. To distinguish an annotation type declaration from a normal interface declaration, the keyword `interface` is preceded by an at-sign (`@`).

*AnnotationTypeDeclaration*:

```
{InterfaceModifier} @ interface TypeIdentifier AnnotationTypeBody
```

Note that the at-sign (`@`) and the keyword `interface` are distinct tokens. It is possible to separate them with whitespace, but this is discouraged as a matter of style.

The rules for annotation modifiers on an annotation type declaration are specified in §9.7.4 and §9.7.5.

The `TypeIdentifier` in an annotation type declaration specifies the name of the annotation type.

It is a compile-time error if an annotation type has the same simple name as any of its enclosing classes or interfaces.

The direct superinterface of every annotation type is `java.lang.annotation.Annotation`.

By virtue of the `AnnotationTypeDeclaration` syntax, an annotation type declaration cannot be generic, and no `extends` clause is permitted.

A consequence of the fact that an annotation type cannot explicitly declare a superclass or superinterface is that a subclass or subinterface of an annotation type is never itself an annotation type. Similarly, `java.lang.annotation.Annotation` is not itself an annotation type.

An annotation type inherits several members from `java.lang.annotation.Annotation`, including the implicitly declared methods corresponding to the instance methods of `Object`, yet these methods do not define elements of the annotation type (§9.6.1).

Because these methods do not define elements of the annotation type, it is illegal to use them in annotations of that type (§9.7). Without this rule, we could not ensure that elements were of the types representable in annotations, or that accessor methods for them would be available.

Unless explicitly modified herein, all of the rules that apply to normal interface declarations apply to annotation type declarations.
For example, annotation types share the same namespace as normal class and interface types; and annotation type declarations are legal wherever interface declarations are legal, and have the same scope and accessibility.

9.6.1 Annotation Type Elements

The body of an annotation type declaration may contain method declarations, each of which defines an element of the annotation type. An annotation type has no elements other than those defined by the methods it explicitly declares.

AnnotationTypeBody:
   { [AnnotationTypeMemberDeclaration] }

AnnotationTypeMemberDeclaration:
   AnnotationTypeElementDeclaration
   ConstantDeclaration
   ClassDeclaration
   InterfaceDeclaration
   ;

AnnotationTypeElementDeclaration:
   [AnnotationTypeElementModifier] UnannType Identifier () [Dims]
   [DefaultValue] ;

AnnotationTypeElementModifier:
   (one of)
   Annotation public
   abstract

By virtue of the AnnotationTypeElementDeclaration production, a method declaration in an annotation type declaration cannot have formal parameters, type parameters, or a throws clause. The following production from §4.3 is shown here for convenience:

Dims:
   {[Annotation] [ ] {[Annotation] [ ]}}

By virtue of the AnnotationTypeElementModifier production, a method declaration in an annotation type declaration cannot be default or static. Thus, an annotation type cannot declare the same variety of methods as a normal interface type. Note that it is still possible for an annotation type to inherit a default method from its implicit superinterface, java.lang.annotation.Annotation, though no such default method exists as of Java SE 11.

By convention, the only AnnotationTypeElementModifiers that should be present on an annotation type element are annotations.
The return type of a method declared in an annotation type must be one of the following, or a compile-time error occurs:

- A primitive type
- String
- `Class` or an invocation of `Class` (§4.5)
- An enum type
- An annotation type
- An array type whose component type is one of the preceding types (§10.1).

This rule precludes elements with nested array types, such as:

```java
@interface Verboten {
    String[][] value();
}
```

The declaration of a method that returns an array is allowed to place the bracket pair that denotes the array type after the empty formal parameter list. This syntax is supported for compatibility with early versions of the Java programming language. It is very strongly recommended that this syntax is not used in new code.

It is a compile-time error if any method declared in an annotation type has a signature that is override-equivalent to that of any `public` or `protected` method declared in class `Object` or in the interface `java.lang.annotation.Annotation`.

It is a compile-time error if an annotation type declaration \( T \) contains an element of type \( T \), either directly or indirectly.

For example, this is illegal:

```java
@interface SelfRef { SelfRef value(); }
```

and so is this:

```java
@interface Ping { Pong value(); }
@interface Pong { Ping value(); }
```

An annotation type with no elements is called a *marker annotation type*.

An annotation type with one element is called a *single-element annotation type*.

By convention, the name of the sole element in a single-element annotation type is `value`. Linguistic support for this convention is provided by single-element annotations (§9.7.3).
Example 9.6.1-1. Annotation Type Declaration

The following annotation type declaration defines an annotation type with several elements:

```java
/**
 * Describes the "request-for-enhancement" (RFE)
 * that led to the presence of the annotated API element.
 */
@interface RequestForEnhancement {
    int id();        // Unique ID number associated with RFE
    String synopsis();  // Synopsis of RFE
    String engineer();  // Name of engineer who implemented RFE
    String date();      // Date RFE was implemented
}
```

Example 9.6.1-2. Marker Annotation Type Declaration

The following annotation type declaration defines a marker annotation type:

```java
/**
 * An annotation with this type indicates that the
 * specification of the annotated API element is
 * preliminary and subject to change.
 */
@interface Preliminary {}
```

Example 9.6.1-3. Single-Element Annotation Type Declarations

The convention that a single-element annotation type defines an element called `value` is illustrated in the following annotation type declaration:

```java
/**
 * Associates a copyright notice with the annotated API element.
 */
@interface Copyright {
    String value();
}
```

The following annotation type declaration defines a single-element annotation type whose sole element has an array type:

```java
/**
 * Associates a list of endorsers with the annotated class.
 */
@interface Endorsers {
    String[] value();
}
```

The following annotation type declaration shows a `Class`-typed element whose value is constrained by a bounded wildcard:
interface Formatter {}

// Designates a formatter to pretty-print the annotated class
@interface PrettyPrinter {
  Class<? extends Formatter> value();
}

The following annotation type declaration contains an element whose type is also an
annotation type:

/**
 * Indicates the author of the annotated program element.
 */
@interface Author {
  Name value();
}

/**
 * A person's name. This annotation type is not designed
 * to be used directly to annotate program elements, but to
 * define elements of other annotation types.
 */
@interface Name {
  String first();
  String last();
}

The grammar for annotation type declarations permits other element declarations besides
method declarations. For example, one might choose to declare a nested enum for use in
conjunction with an annotation type:

@interface Quality {
  enum Level { BAD, INDIFFERENT, GOOD }
  Level value();
}

### 9.6.2 Defaults for Annotation Type Elements

An annotation type element may have a *default value*, specified by following the
element's (empty) parameter list with the keyword `default` and an `ElementValue` (**§9.7.1**).

**DefaultValue:**

```
default ElementValue
```

It is a compile-time error if the type of the element is not commensurate (**§9.7**) with
the default value specified.

Default values are not compiled into annotations, but rather applied dynamically
at the time annotations are read. Thus, changing a default value affects annotations
even in classes that were compiled before the change was made (presuming these annotations lack an explicit value for the defaulted element).

Example 9.6.2-1. Annotation Type Declaration With Default Values

Here is a refinement of the RequestForEnhancement annotation type from §9.6.1:

```java
@interface RequestForEnhancementDefault {
    int id();       // No default - must be specified in
                    // each annotation
    String synopsis(); // No default - must be specified in
                        // each annotation
    String engineer() default "[unassigned]";
    String date()      default "[unimplemented]";
}
```

9.6.3 Repeatable Annotation Types

An annotation type \( T \) is repeatable if its declaration is (meta-)annotated with an @Repeatable annotation (§9.6.4.8) whose value element indicates a containing annotation type of \( T \).

An annotation type \( TC \) is a containing annotation type of \( T \) if all of the following are true:

1. \( TC \) declares a \texttt{value()} method whose return type is \( T[] \).
2. Any methods declared by \( TC \) other than \texttt{value()} have a default value.
3. \( TC \) is retained for at least as long as \( T \), where retention is expressed explicitly or implicitly with the @Retention annotation (§9.6.4.2). Specifically:
   - If the retention of \( TC \) is \texttt{java.lang.annotation.RetentionPolicy.SOURCE}, then the retention of \( T \) is \texttt{java.lang.annotation.RetentionPolicy.SOURCE}.
   - If the retention of \( TC \) is \texttt{java.lang.annotation.RetentionPolicy.CLASS}, then the retention of \( T \) is either \texttt{java.lang.annotation.RetentionPolicy.CLASS} or \texttt{java.lang.annotation.RetentionPolicy.SOURCE}.
   - If the retention of \( TC \) is \texttt{java.lang.annotation.RetentionPolicy.RUNTIME}, then the retention of \( T \) is \texttt{java.lang.annotation.RetentionPolicy.SOURCE}, \texttt{java.lang.annotation.RetentionPolicy.CLASS}, or \texttt{java.lang.annotation.RetentionPolicy.RUNTIME}.
4. $T$ is applicable to at least the same kinds of program element as $TC$ (§9.6.4.1). Specifically, if the kinds of program element where $T$ is applicable are denoted by the set $m_1$, and the kinds of program element where $TC$ is applicable are denoted by the set $m_2$, then each kind in $m_2$ must occur in $m_1$, except that:

- If the kind in $m_2$ is `java.lang.annotation.ElementType.ANNOTATION_TYPE`, then at least one of `java.lang.annotation.ElementType.ANNOTATION_TYPE` or `java.lang.annotation.ElementType.TYPE` or `java.lang.annotation.ElementType.TYPE_USE` must occur in $m_1$.

- If the kind in $m_2$ is `java.lang.annotation.ElementType.TYPE`, then at least one of `java.lang.annotation.ElementType.TYPE` or `java.lang.annotation.ElementType.TYPE_USE` must occur in $m_1$.

- If the kind in $m_2$ is `java.lang.annotation.ElementType.TYPE_PARAMETER`, then at least one of `java.lang.annotation.ElementType.TYPE_PARAMETER` or `java.lang.annotation.ElementType.TYPE_USE` must occur in $m_1$.

This clause implements the policy that an annotation type may be repeatable on only some of the kinds of program element where it is applicable.

5. If the declaration of $T$ has a (meta-)annotation that corresponds to `java.lang.annotation.Documented`, then the declaration of $TC$ must have a (meta-)annotation that corresponds to `java.lang.annotation.Documented`.

Note that it is permissible for $TC$ to be `@Documented` while $T$ is not `@Documented`.

6. If the declaration of $T$ has a (meta-)annotation that corresponds to `java.lang.annotation.Inherited`, then the declaration of $TC$ must have a (meta)-annotation that corresponds to `java.lang.annotation.Inherited`.

Note that it is permissible for $TC$ to be `@Inherited` while $T$ is not `@Inherited`.

It is a compile-time error if an annotation type $T$ is (meta-)annotated with an `@Repeatable` annotation whose value element indicates a type which is not a containing annotation type of $T$.

**Example 9.6.3-1. Ill-formed Containing Annotation Type**

Consider the following declarations:

```java
import java.lang.annotation.Repeatable;

@Repeatable(FooContainer.class)
@interface Foo {}
```
@interface FooContainer { Object[] value(); }

Compiling the Foo declaration produces a compile-time error because Foo uses @Repeatable to attempt to specify FooContainer as its containing annotation type, but FooContainer is not in fact a containing annotation type of Foo. (The return type of FooContainer.value() is not Foo[].)

The @Repeatable annotation cannot be repeated, so only one containing annotation type can be specified by a repeatable annotation type.

Allowing more than one containing annotation type to be specified would cause an undesirable choice at compile time, when multiple annotations of the repeatable annotation type are logically replaced with a container annotation (§9.7.5).

An annotation type can be the containing annotation type of at most one annotation type.

This is implied by the requirement that if the declaration of an annotation type \( T \) specifies a containing annotation type of \( TC \), then the value() method of \( TC \) has a return type involving \( T \), specifically \( T[] \).

An annotation type cannot specify itself as its containing annotation type.

This is implied by the requirement on the value() method of the containing annotation type. Specifically, if an annotation type \( A \) specified itself (via @Repeatable) as its containing annotation type, then the return type of \( A \)'s value() method would have to be \( A[] \); but this would cause a compile-time error since an annotation type cannot refer to itself in its elements (§9.6.1). More generally, two annotation types cannot specify each other to be their containing annotation types, because cyclic annotation type declarations are illegal.

An annotation type \( TC \) may be the containing annotation type of some annotation type \( T \) while also having its own containing annotation type \( TC' \). That is, a containing annotation type may itself be a repeatable annotation type.

Example 9.6.3-2. Restricting Where Annotations May Repeat

An annotation whose type declaration indicates a target of java.lang.annotation.ElementType.TYPE can appear in at least as many locations as an annotation whose type declaration indicates a target of java.lang.annotation.ElementType.ANNOTATION_TYPE. For example, given the following declarations of repeatable and containing annotation types:

import java.lang.annotationTargetException;
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;

@Target(ElementType.TYPE)
@Repeatable(FooContainer.class)
@interface Foo {}

@Target(ElementType.ANNOTATION_TYPE)
@interface FooContainer {
    Foo[] value();
}

@Foo can appear on any type declaration while @FooContainer can appear on only annotation type declarations. Therefore, the following annotation type declaration is legal:

@Foo @Foo
@interface Anno {}

while the following interface declaration is illegal:

@Foo @Foo
interface Intf {}

More broadly, if Foo is a repeatable annotation type and FooContainer is its containing annotation type, then:

- If Foo has no @Target meta-annotation and FooContainer has no @Target meta-annotation, then @Foo may be repeated on any program element which supports annotations.
- If Foo has no @Target meta-annotation but FooContainer has an @Target meta-annotation, then @Foo may only be repeated on program elements where @FooContainer may appear.
- If Foo has an @Target meta-annotation, then in the judgment of the designers of the Java programming language, FooContainer must be declared with knowledge of the Foo's applicability. Specifically, the kinds of program element where FooContainer may appear must logically be the same as, or a subset of, Foo's kinds.

For example, if Foo is applicable to field and method declarations, then FooContainer may legitimately serve as Foo's containing annotation type if FooContainer is applicable to just field declarations (preventing @Foo from being repeated on method declarations). But if FooContainer is applicable only to formal parameter declarations, then FooContainer was a poor choice of containing annotation type by Foo because @FooContainer cannot be implicitly declared on some program elements where @Foo is repeated.

Similarly, if Foo is applicable to field and method declarations, then FooContainer cannot legitimately serve as Foo's containing annotation type if FooContainer is applicable to field and parameter declarations. While it would be possible to take the intersection of the program elements and make Foo repeatable on field declarations only, the presence of additional program elements for FooContainer indicates that FooContainer was not designed as a containing annotation type for Foo. It would therefore be dangerous for Foo to rely on it.
Example 9.6.3-3. A Repeatable Containing Annotation Type

The following declarations are legal:

```java
import java.lang.annotation.Repeatable;

// Foo: Repeatable annotation type
@Repeatable(FooContainer.class)
@interface Foo { int value(); }

// FooContainer: Containing annotation type of Foo
// Also a repeatable annotation type itself
@Repeatable(FooContainerContainer.class)
@interface FooContainer { Foo[] value(); }

// FooContainerContainer: Containing annotation type of FooContainer
@interface FooContainerContainer { FooContainer[] value(); }
```

Thus, an annotation whose type is a containing annotation type may itself be repeated:

```java
@FooContainer({@Foo(1)}) @FooContainer({@Foo(2)})
class Test {}
```

An annotation type which is both repeatable and containing is subject to the rules on mixing annotations of repeatable annotation type with annotations of containing annotation type (§9.7.5). For example, it is not possible to write multiple `@Foo` annotations alongside multiple `@FooContainer` annotations, nor is it possible to write multiple `@FooContainer` annotations alongside multiple `@FooContainerContainer` annotations. However, if the `FooContainerContainer` type was itself repeatable, then it would be possible to write multiple `@Foo` annotations alongside multiple `@FooContainerContainer` annotations.

### 9.6.4 Predefined Annotation Types

Several annotation types are predefined in the libraries of the Java SE Platform. Some of these predefined annotation types have special semantics. These semantics are specified in this section. This section does not provide a complete specification for the predefined annotations contained here in; that is the role of the appropriate API specifications. Only those semantics that require special behavior on the part of a Java compiler or Java Virtual Machine implementation are specified here.

#### 9.6.4.1 @Target

An annotation of type `java.lang.annotation.Target` is used on the declaration of an annotation type `T` to specify the contexts in which `T` is applicable. `java.lang.annotation.Target` has a single element, `value`, of type `java.lang.annotation.ElementType[]`, to specify contexts.
Annotation types may be applicable in *declaration contexts*, where annotations apply to declarations, or in *type contexts*, where annotations apply to types used in declarations and expressions.

There are nine declaration contexts, each corresponding to an enum constant of `java.lang.annotation.ElementType`:

1. Module declarations (§7.7)
   
   Corresponds to `java.lang.annotation.ElementType.MODULE`

2. Package declarations (§7.4.1)
   
   Corresponds to `java.lang.annotation.ElementType.PACKAGE`

3. Type declarations: class, interface, enum, and annotation type declarations (§8.1.1, §9.1.1, §8.5, §9.5, §8.9, §9.6)
   
   Corresponds to `java.lang.annotation.ElementType.TYPE`
   
   Additionally, annotation type declarations correspond to `java.lang.annotation.ElementType.ANNOTATION_TYPE`

4. Method declarations (including elements of annotation types) (§8.4.3, §9.4, §9.6.1)
   
   Corresponds to `java.lang.annotation.ElementType.METHOD`

5. Constructor declarations (§8.8.3)
   
   Corresponds to `java.lang.annotation.ElementType.CONSTRUCTOR`

6. Type parameter declarations of generic classes, interfaces, methods, and constructors (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
   
   Corresponds to `java.lang.annotation.ElementType.TYPE_PARAMETER`

7. Field declarations (including enum constants) (§8.3.1, §9.3, §8.9.1)
   
   Corresponds to `java.lang.annotation.ElementType.FIELD`

8. Formal and exception parameter declarations (§8.4.1, §9.4, §14.20)
   
   Corresponds to `java.lang.annotation.ElementType.PARAMETER`

   
   Corresponds to `java.lang.annotation.ElementType.LOCAL_VARIABLE`
There are 16 type contexts (§4.11), all represented by the enum constant `TYPE_USE` of `java.lang.annotation.ElementType`.

It is a compile-time error if the same enum constant appears more than once in the value element of an annotation of type `java.lang.annotation.Target`.

If an annotation of type `java.lang.annotation.Target` is not present on the declaration of an annotation type $T$, then $T$ is applicable in all declaration contexts except type parameter declarations, and in no type contexts.

These contexts are the syntactic locations where annotations were allowed in Java SE 7.

9.6.4.2 @Retention

Annotations may be present only in source code, or they may be present in the binary form of a class or interface. An annotation that is present in the binary form may or may not be available at run time via the reflection libraries of the Java SE Platform. The annotation type `java.lang.annotation.Retention` is used to choose among these possibilities.

If an annotation $a$ corresponds to a type $T$, and $T$ has a (meta-)annotation $m$ that corresponds to `java.lang.annotation.Retention`, then:

- If $m$ has an element whose value is `java.lang.annotation.RetentionPolicy.SOURCE`, then a Java compiler must ensure that $a$ is not present in the binary representation of the class or interface in which $a$ appears.
- If $m$ has an element whose value is `java.lang.annotation.RetentionPolicy.CLASS` or `java.lang.annotation.RetentionPolicy.RUNTIME`, then a Java compiler must ensure that $a$ is represented in the binary representation of the class or interface in which $a$ appears, unless $a$ annotates a local variable declaration or $a$ annotates a formal parameter declaration of a lambda expression.

An annotation on the declaration of a local variable, or on the declaration of a formal parameter of a lambda expression, is never retained in the binary representation. In contrast, an annotation on the type of a local variable, or on the type of a formal parameter of a lambda expression, is retained in the binary representation if the annotation type specifies a suitable retention policy.

Note that it is not illegal for an annotation type to be meta-annotated with `@Target(java.lang.annotation.ElementType.LOCAL_VARIABLE)` and `@Retention(java.lang.annotation.RetentionPolicy.CLASS)` or `@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)`.
If \( m \) has an element whose value is `java.lang.annotation.RetentionPolicy.RUNTIME`, the reflection libraries of the Java SE Platform must make \( a \) available at run time.

If \( T \) does not have a (meta-)annotation \( m \) that corresponds to `java.lang.annotation.Retention`, then a Java compiler must treat \( T \) as if it does have such a meta-annotation \( m \) with an element whose value is `java.lang.annotation.RetentionPolicy.CLASS`.

9.6.4.3 \@Inherited

The annotation type `java.lang.annotation.Inherited` is used to indicate that annotations on a class \( C \) corresponding to a given annotation type are inherited by subclasses of \( C \).

9.6.4.4 \@Override

Programmers occasionally overload a method declaration when they mean to override it, leading to subtle problems. The annotation type `Override` supports early detection of such problems.

The classic example concerns the `equals` method. Programmers write the following in class `Foo`:

```java
public boolean equals(Foo that) { ... }
```

when they mean to write:

```java
public boolean equals(Object that) { ... }
```

This is perfectly legal, but class `Foo` inherits the `equals` implementation from `Object`, which can cause some subtle bugs.

If a method declaration in type \( T \) is annotated with `@Override`, but the method does not override from \( T \) a method declared in a supertype of \( T \) (§8.4.8.1, §9.4.1.1), or is not override-equivalent to a public method of `Object` (§4.3.2, §8.4.2), then a compile-time error occurs.

This behavior differs from Java SE 5.0, where `@Override` only caused a compile-time error if applied to a method that implemented a method from a superinterface that was not also present in a superclass.

The clause about overriding a public method is motivated by use of `@Override` in an interface. Consider the following type declarations:
class Foo { @Override public int hashCode() { .. } }
interface Bar { @Override int hashCode(); }

The use of @Override in the class declaration is legal by the first clause, because Foo.hashCode overrides from Foo the method Object.hashCode.

For the interface declaration, consider that while an interface does not have Object as a supertype, an interface does have public abstract members that correspond to the public members of Object (§9.2). If an interface chooses to declare them explicitly (that is, to declare members that are override-equivalent to public methods of Object), then the interface is deemed to override them, and use of @Override is allowed.

However, consider an interface that attempts to use @Override on a clone method: (finalize could also be used in this example)

interface Quux { @Override Object clone(); }

Because Object.clone is not public, there is no member called clone implicitly declared in Quux. Therefore, the explicit declaration of clone in Quux is not deemed to "implement" any other method, and it is erroneous to use @Override. (The fact that Quux.clone is public is not relevant.)

In contrast, a class declaration that declares clone is simply overriding Object.clone, so is able to use @Override:

class Beep { @Override protected Object clone() { .. } }

9.6.4.5 @SuppressWarnings

Java compilers are increasingly capable of issuing helpful "lint-like" warnings. To encourage the use of such warnings, there should be some way to disable a warning in a part of the program when the programmer knows that the warning is inappropriate.

The annotation type SuppressWarnings supports programmer control over warnings otherwise issued by a Java compiler. It defines a single element that is an array of String.

If a declaration is annotated with @SuppressWarnings(value = \{S\_1, \ldots, S\_k\}), then a Java compiler must suppress (that is, not report) any warning specified by one of S\_1 \ldots S\_k if that warning would have been generated as a result of the annotated declaration or any of its parts.

The three kinds of warnings defined by the Java programming language are specified using the following strings:

- Unchecked warnings (§4.8, §5.1.6, §5.1.9, §8.4.1, §8.4.8.3, §15.12.4.2, §15.13.2, §15.27.3) are specified by the string "unchecked".
• Deprecation warnings (§9.6.4.6) are specified by the string "deprecation".

• Removal warnings (§9.6.4.6) are specified by the string "removal".

For other kinds of warnings, compiler vendors should document the strings they support for @SuppressWarnings. Vendors are encouraged to cooperate to ensure that the same names work across multiple compilers.

9.6.4.6  @Deprecated

Programmers are sometimes discouraged from using certain program elements (modules, types, fields, methods, and constructors) because they are dangerous or because a better alternative exists. The annotation type Deprecated allows a compiler to warn about uses of these program elements.

A deprecated program element is a module, type, field, method, or constructor whose declaration is annotated with @Deprecated. The manner in which a program element is deprecated depends on the value of the forRemoval element of the annotation:

• If forRemoval=false (the default), then the program element is ordinarily deprecated.

An ordinarily deprecated program element is not intended to be removed in a future release, but programmers should nevertheless migrate away from using it.

• If forRemoval=true, then the program element is terminally deprecated.

A terminally deprecated program element is intended to be removed in a future release. Programmers should stop using it or risk source and binary incompatibilities (§13.2) when upgrading to a newer release.

A Java compiler must produce a deprecation warning when an ordinarily deprecated program element is used (overridden, invoked, or referenced by name) in the declaration of a program element (whether explicitly or implicitly declared), unless:

• The use is within a declaration that is itself deprecated, either ordinarily or terminally; or

• The use is within a declaration that is annotated to suppress deprecation warnings; or

• The use and declaration are both within the same outermost class; or

• The use is within an import declaration that imports the ordinarily deprecated type or member.
• The use is within an `exports` or `opens` directive (§7.7.2).

A Java compiler must produce a *removal warning* when a terminally deprecated program element is used (overridden, invoked, or referenced by name) in the declaration of a program element (whether explicitly or implicitly declared), unless:

• The use is within a declaration that is annotated to suppress removal warnings; or
• The use and declaration are both within the same outermost class; or
• The use is within an `import` declaration that imports the terminally deprecated type or member.
• The use is within an `exports` or `opens` directive.

Terminal deprecation is sufficiently urgent that the use of a terminally deprecated element will cause a removal warning *even if the using element is itself deprecated*, since there is no guarantee that both elements will be removed at the same time. To dismiss the warning but continue using the element, the programmer must manually acknowledge the risk via an `@SuppressWarnings` annotation.

No deprecation warning or removal warning is produced when:

• a local variable or formal parameter is used (referenced by name), even if the declaration of the local variable or formal parameter is annotated with `@Deprecated`.

• the name of a package is used (referenced by a qualified type name, or an `import` declaration, or an `exports` or `opens` directive), even if the declaration of the package is annotated with `@Deprecated`.

• the name of a module is used by a qualified `exports` or `opens` directive, even if the declaration of the friend module is annotated with `@Deprecated`.

A module declaration that exports or opens a package is usually controlled by the same programmer or team that controls the package's declaration. As such, there is little benefit in warning that the package declaration is annotated with `@Deprecated` when the package is exported or opened by the module declaration. In contrast, a module declaration that exports or opens a package to a friend module is usually not controlled by the same programmer or team that controls the friend module. Simply exporting or opening the package does not make the module declaration rely on the friend module, so there is little value in warning if the friend module is deprecated; the programmer of the module declaration would almost always wish to suppress such a warning.

The only implicit declaration that can cause a deprecation warning or removal warning is a container annotation (§9.7.5). Namely, if $T$ is a repeatable annotation type and $TC$ is its containing annotation type, and $TC$ is deprecated, then repeating the $@T$ annotation will cause a warning. The warning is due to the implicit $@TC$ container annotation. It is
strongly discouraged to deprecate a containing annotation type without deprecating the corresponding repeatable annotation type.

9.6.4.7 @SafeVarargs

A variable arity parameter with a non-reifiable element type (§4.7) can cause heap pollution (§4.12.2) and give rise to compile-time unchecked warnings (§5.1.9). Such warnings are uninformative if the body of the variable arity method is well-behaved with respect to the variable arity parameter.

The annotation type SafeVarargs, when used to annotate a method or constructor declaration, makes a programmer assertion that prevents a Java compiler from reporting unchecked warnings for the declaration or invocation of a variable arity method or constructor where the compiler would otherwise do so due to the variable arity parameter having a non-reifiable element type.

The annotation @SafeVarargs has non-local effects because it suppresses unchecked warnings at method invocation expressions, in addition to an unchecked warning pertaining to the declaration of the variable arity method itself (§8.4.1). In contrast, the annotation @SuppressWarnings("unchecked") has local effects because it only suppresses unchecked warnings pertaining to the declaration of a method.

The canonical target for @SafeVarargs is a method like java.util.Collections.addAll, whose declaration starts with:

```java
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

The variable arity parameter has declared type T[], which is non-reifiable. However, the method fundamentally just reads from the input array and adds the elements to a collection, both of which are safe operations with respect to the array. Therefore, any compile-time unchecked warnings at method invocation expressions for java.util.Collections.addAll are arguably spurious and uninformative. Applying @SafeVarargs to the method declaration prevents generation of these unchecked warnings at the method invocation expressions.

It is a compile-time error if a fixed arity method or constructor declaration is annotated with the annotation @SafeVarargs.

It is a compile-time error if a variable arity method declaration that is neither static nor final nor private is annotated with the annotation @SafeVarargs.

Since @SafeVarargs is only applicable to static methods, final and/or private instance methods, and constructors, the annotation is not usable where method overriding occurs. Annotation inheritance only works for annotations on classes (not on methods, interfaces, or constructors), so an @SafeVarargs-style annotation cannot be passed through instance methods in classes or through interfaces.
9.6.4.8  @Repeatable

The annotation type `java.lang.annotation.Repeatable` is used on the declaration of a *repeatable annotation type* to indicate its containing annotation type (§9.6.3).

> Note that an `@Repeatable` meta-annotation on the declaration of `T`, indicating `TC`, is *not* sufficient to make `TC` the containing annotation type of `T`. There are numerous well-formedness rules for `TC` to be considered the containing annotation type of `T`.

9.6.4.9  @FunctionalInterface

The annotation type `FunctionalInterface` is used to indicate that an interface is meant to be a functional interface (§9.8). It facilitates early detection of inappropriate method declarations appearing in or inherited by an interface that is meant to be functional.

It is a compile-time error if an interface declaration is annotated with `@FunctionalInterface` but is not, in fact, a functional interface.

Because some interfaces are functional incidentally, it is not necessary or desirable that all declarations of functional interfaces be annotated with `@FunctionalInterface`.

9.7  Annotations

An *annotation* is a marker which associates information with a program construct, but has no effect at run time. An annotation denotes a specific invocation of an annotation type (§9.6) and usually provides values for the elements of that type.

There are three kinds of annotations. The first kind is the most general, while the other kinds are merely shorthands for the first kind.

*Annotation:*

- `NormalAnnotation`
- `MarkerAnnotation`
- `SingleElementAnnotation`

Normal annotations are described in §9.7.1, marker annotations in §9.7.2, and single element annotations in §9.7.3. Annotations may appear at various syntactic locations in a program, as described in §9.7.4. The number of annotations of the
same type that may appear at a location is determined by their type, as described in §9.7.5.

9.7.1 Normal Annotations

A normal annotation specifies the name of an annotation type and optionally a list of comma-separated element-value pairs. Each pair contains an element value that is associated with an element of the annotation type (§9.6.1).

\[
\text{NormalAnnotation:} \\
\text{\hspace{1em} @ TypeName ( [ElementValuePairList] )} \\
\text{ElementValuePairList:} \\
\quad \text{ElementValuePair} \{, \text{ ElementValuePair } \}
\]

\[
\text{ElementValuePair:} \\
\quad \text{Identifier} = \text{ ElementValue }
\]

\[
\text{ElementValue:} \\
\quad \text{ConditionalExpression} \\
\quad \text{ElementValueArrayInitializer} \\
\quad \text{Annotation}
\]

\[
\text{ElementValueArrayInitializer:} \\
\quad ( \ [\text{ElementValueList}] [, ] )
\]

\[
\text{ElementValueList:} \\
\quad \text{ElementValue} \{, \text{ ElementValue } \}
\]

Note that the at-sign (@) is a token unto itself (§3.11). It is possible to put whitespace between it and the TypeName, but this is discouraged as a matter of style.

The TypeName specifies the annotation type corresponding to the annotation. The annotation is said to be "of" that type.

The TypeName must name an accessible annotation type (§6.6), or a compile-time error occurs.

The Identifier in an element-value pair must be the simple name of one of the elements (that is, methods) of the annotation type, or a compile-time error occurs.

The return type of this method defines the element type of the element-value pair.
If the element type is an array type, then it is not required to use curly braces to specify the element value of the element-value pair. If the element value is not an `ElementValueArrayInitializer`, then an array value whose sole element is the element value is associated with the element. If the element value is an `ElementValueArrayInitializer`, then the array value represented by the `ElementValueArrayInitializer` is associated with the element.

It is a compile-time error if the element type is not commensurate with the element value. An element type $\tau$ is commensurate with an element value $v$ if and only if one of the following is true:

- $\tau$ is an array type $E[]$, and either:
  - If $v$ is a `ConditionalExpression` or an `Annotation`, then $v$ is commensurate with $E$; or
  - If $v$ is an `ElementValueArrayInitializer`, then each element value that $v$ contains is commensurate with $E$.

An `ElementValueArrayInitializer` is similar to a normal array initializer (§10.6), except that an `ElementValueArrayInitializer` may syntactically contain annotations as well as expressions and nested initializers. However, nested initializers are not semantically legal in an `ElementValueArrayInitializer` because they are never commensurate with array-typed elements in annotation type declarations (nested array types not permitted).

- $\tau$ is not an array type, and the type of $v$ is assignment compatible (§5.2) with $\tau$, and:
  - If $\tau$ is a primitive type or `String`, then $v$ is a constant expression (§15.28).
  - If $\tau$ is `Class` or an invocation of `Class` (§4.5), then $v$ is a class literal (§15.8.2).
  - If $\tau$ is an enum type (§8.9), then $v$ is an enum constant (§8.9.1).
  - $v$ is not null.

Note that if $\tau$ is not an array type or an annotation type, the element value must be a `ConditionalExpression` (§15.25). The use of `ConditionalExpression` rather than a more general production like `Expression` is a syntactic trick to prevent assignment expressions as element values. Since an assignment expression is not a constant expression, it cannot be a commensurate element value for a primitive or `String`-typed element.

Formally, it is invalid to speak of an `ElementValue` as FP-strict (§15.4) because it might be an annotation or a class literal. Still, we can speak informally of `ElementValue` as FP-strict when it is either a constant expression or an array of constant expressions or an annotation whose element values are (recursively) found to be constant expressions; after all, every constant expression is FP-strict.
A normal annotation must contain an element-value pair for every element of the corresponding annotation type, except for those elements with default values, or a compile-time error occurs.

A normal annotation may, but is not required to, contain element-value pairs for elements with default values.

It is customary, though not required, that element-value pairs in an annotation are presented in the same order as the corresponding elements in the annotation type declaration.

An annotation on an annotation type declaration is known as a *meta-annotation*.

An annotation of type \( \tau \) may appear as a meta-annotation on the declaration of type \( \tau \) itself. More generally, circularities in the transitive closure of the "annotates" relation are permitted.

For example, it is legal to annotate the declaration of an annotation type \( S \) with a meta-annotation of type \( \tau \), and to annotate \( \tau \)'s own declaration with a meta-annotation of type \( S \). The pre-defined annotation types contain several such circularities.

**Example 9.7.1-1. Normal Annotations**

Here is an example of a normal annotation using the annotation type from §9.6.1:

```java
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

Here is an example of a normal annotation that takes advantage of default values, using the annotation type from §9.6.2:

```java
@RequestForEnhancement(
    id       = 4561414,
    synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

### 9.7.2 Marker Annotations

A *marker annotation* is a shorthand designed for use with marker annotation types (§9.6.1).
9.7 Annotations

MarkerAnnotation:
@TypeName

It is shorthand for the normal annotation:

@TypeName()

It is legal to use marker annotations for annotation types with elements, so long as all the elements have default values (§9.6.2).

Example 9.7.2-1. Marker Annotations

Here is an example using the Preliminary marker annotation type from §9.6.1:

@Preliminary public class TimeTravel {

9.7.3 Single-Element Annotations

A single-element annotation, is a shorthand designed for use with single-element annotation types (§9.6.1).

SingleElementAnnotation:
@TypeName(ElementValue)

It is shorthand for the normal annotation:

@TypeName(value = ElementValue)

It is legal to use single-element annotations for annotation types with multiple elements, so long as one element is named value and all other elements have default values (§9.6.2).

Example 9.7.3-1. Single-Element Annotations

The following annotations all use the single-element annotation types from §9.6.1.

Here is an example of a single-element annotation:

@Copyright("2002 Yoyodyne Propulsion Systems, Inc.")
public class OscillationOverthruster {

Here is an example of an array-valued single-element annotation:

@Endorsers("Children", "Unscrupulous dentists")
public class Lollipop {

...}
Here is an example of a single-element array-valued single-element annotation: (note that the curly braces are omitted)

```java
@Endorsers("Epicurus")
public class Pleasure { ... }
```

Here is an example of a single-element annotation with a `Class`-typed element whose value is constrained by a bounded wildcard.

```java
class GorgeousFormatter implements Formatter { ... }

@PrettyPrinter(GorgeousFormatter.class)
public class Petunia { ... }

// Illegal; String is not a subtype of Formatter
@PrettyPrinter(String.class)
public class Begonia { ... }
```

Here is an example with of a single-element annotation that contains a normal annotation:

```java
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

Here is an example of a single-element annotation that uses an enum type defined inside the annotation type:

```java
@Quality(Quality.Level.GOOD)
public class Karma { ... }
```

### 9.7.4 Where Annotations May Appear

A **declaration annotation** is an annotation that applies to a declaration, and whose own type is applicable in the declaration context (§9.6.4.1) represented by that declaration; or an annotation that applies to a class, interface, enum, annotation type, or type parameter declaration, and whose own type is applicable in type contexts (§4.11).

A **type annotation** is an annotation that applies to a type (or any part of a type), and whose own type is applicable in type contexts.

For example, given the field declaration:

```java
@Foo int f;
```

@Foo is a declaration annotation on f if Foo is meta-annotated by @Target(ElementType.FIELD), and a type annotation on int if Foo is meta-annotated by @Target(ElementType.TYPE_USE). It is possible for @Foo to be both a declaration annotation and a type annotation simultaneously.
Type annotations can apply to an array type or any component type thereof (§10.1).
For example, assuming that A, B, and C are annotation types meta-annotated with
@Target(ElementType.TYPE_USE), then given the field declaration:

```java
@C int @A [] @B [] f;
```

@A applies to the array type `int[][]`, @B applies to its component type `int[]`, and @C
applies to the element type `int`. For more examples, see §10.2.

An important property of this syntax is that, in two declarations that differ only in the
number of array levels, the annotations to the left of the type refer to the same type. For
example, @C applies to the type `int` in all of the following declarations:

```java
@C int f;
@C int[] f;
@C int[][] f;
```

It is customary, though not required, to write declaration annotations before all other
modifiers, and type annotations immediately before the type to which they apply.

It is possible for an annotation to appear at a syntactic location in a program where
it could plausibly apply to a declaration, or a type, or both. This can happen in any
of the five declaration contexts where modifiers immediately precede the type of
the declared entity:

- Method declarations (including elements of annotation types)
- Constructor declarations
- Field declarations (including enum constants)
- Formal and exception parameter declarations
- Local variable declarations (including loop variables of `for` statements and
  resource variables of `try-with-resources` statements)

The grammar of the Java programming language unambiguously treats annotations
at these locations as modifiers for a declaration (§8.3), but that is purely a syntactic
matter. Whether an annotation applies to the declaration or to the type of the
declared entity - and thus, whether the annotation is a declaration annotation or a
type annotation - depends on the applicability of the annotation's type:

- If the annotation's type is applicable in the declaration context corresponding to
  the declaration, and not in type contexts, then the annotation is deemed to apply
  only to the declaration.
- If the annotation's type is applicable in type contexts, and not in the declaration
  context corresponding to the declaration, then the annotation is deemed to apply
  only to the type which is closest to the annotation.
• If the annotation's type is applicable in the declaration context corresponding to the declaration and in type contexts, then the annotation is deemed to apply to both the declaration and the type which is closest to the annotation.

In the second and third cases above, the type which is closest to the annotation is determined as follows:

• If the annotation appears before a `void` method declaration or a local variable declaration that uses `var` (§14.4, §14.14.2, §14.20.3), then there is no closest type. If the annotation's type is deemed to apply only to the type which is closest to the annotation, a compile-time error occurs.

• If the annotation appears before a constructor declaration, then the closest type is the type of the newly constructed object. The type of the newly constructed object is the fully qualified name of the type immediately enclosing the constructor declaration. Within that fully qualified name, the annotation applies to the simple type name indicated by the constructor declaration.

• In all other cases, the closest type is the type written in source code for the declared entity; if that type is an array type, then the element type is deemed to be closest to the annotation.

For example, in the field declaration `@Foo public static String f;`, the type which is closest to `@Foo` is `String`. (If the type of the field declaration had been written as `java.lang.String`, then `java.lang.String` would be the type closest to `@Foo`, and later rules would prohibit a type annotation from applying to the package name `java`.) In the generic method declaration `@Foo <T> int[] m() { ... }`, the type written for the declared entity is `int[]`, so `@Foo` applies to the element type `int`.

Local variable declarations which do not use `var` are similar to formal parameter declarations of lambda expressions, in that both allow declaration annotations and type annotations in source code, but only the type annotations can be stored in the class file.

It is a compile-time error if an annotation of type \( \tau \) is syntactically a modifier for:

• a module declaration, but \( \tau \) is not applicable to module declarations.

• a package declaration, but \( \tau \) is not applicable to package declarations.

• a class, interface, or enum declaration, but \( \tau \) is not applicable to type declarations or type contexts; or an annotation type declaration, but \( \tau \) is not applicable to annotation type declarations or type declarations or type contexts.

• a method declaration (including an element of an annotation type), but \( \tau \) is not applicable to method declarations or type contexts.

• a constructor declaration, but \( \tau \) is not applicable to constructor declarations or type contexts.
• a type parameter declaration of a generic class, interface, method, or constructor, but \( T \) is not applicable to type parameter declarations or type contexts.

• a field declaration (including an enum constant), but \( T \) is not applicable to field declarations or type contexts.

• a formal or exception parameter declaration, but \( T \) is not applicable to either formal and exception parameter declarations or type contexts.

• a receiver parameter, but \( T \) is not applicable to type contexts.

• a local variable declaration (including a loop variable of a \texttt{for} statement or a resource variable of a \texttt{try}-with-resources statement), but \( T \) is not applicable to local variable declarations or type contexts.

Five of these nine clauses mention "... or type contexts" because they characterize the five syntactic locations where an annotation could plausibly apply either to a declaration or to the type of a declared entity. Furthermore, two of the nine clauses - for class, interface, enum, and annotation type declarations, and for type parameter declarations - mention "... or type contexts" because it may be convenient to apply an annotation whose type is meta-annotated with \texttt{@Target(ElementType.TYPE_USE)} (thus, applicable in type contexts) to a type declaration.

A type annotation is \textit{admissible} if both of the following are true:

• The simple name to which the annotation is closest is classified as a \textit{TypeName}, not a \textit{PackageName}.

• If the simple name to which the annotation is closest is followed by "." and another \textit{TypeName} - that is, the annotation appears as \texttt{@Foo T.U} - then \( U \) denotes an inner class of \( T \).

The intuition behind the second clause is that if \texttt{Outer.this} is legal in a nested class enclosed by \texttt{Outer}, then \texttt{Outer} may be annotated because it represents the type of some object at run time. On the other hand, if \texttt{Outer.this} is not legal - because the class where it appears has no enclosing instance of \texttt{Outer} at run time - then \texttt{Outer} may not be annotated because it is logically just a name, akin to components of a package name in a fully qualified type name.

For example, in the following program, it is not possible to write \texttt{A.this} in the body of \texttt{B}, as \texttt{B} has no lexically enclosing instances (8.5.1). Therefore, it is not possible to apply \texttt{@Foo} to \texttt{A} in the type \texttt{A.B}, because \texttt{A} is logically just a name, not a type.

```java
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    class A {
        static class B {}
    }
}
```
@Foo A.B x;  // Illegal
}

On the other hand, in the following program, it is possible to write C.this in the body of D. Therefore, it is possible to apply @Foo to C in the type C.D, because C represents the type of some object at run time.

@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    static class C {
        class D {}
    }
    @Foo C.D x;  // Legal
}

Finally, note that the second clause looks only one level deeper in a qualified type. This is because a static class may only be nested in a top level class or another static nested class. It is not possible to write a nest like:

@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    class E {
        class F {
            static class G {}
        }
    }
    @Foo E.F.G x;
}

Assume for a moment that the nest was legal. In the type of field x, E and F would logically be names qualifying G, as E.F.this would be illegal in the body of G. Then, @Foo should not be legal next to E. Technically, however, @Foo would be admissible next to E because the next deepest term F denotes an inner class; but this is moot as the class nest is illegal in the first place.

It is a compile-time error if an annotation of type $\tau$ applies to the outermost level of a type in a type context, and $\tau$ is not applicable in type contexts or the declaration context (if any) which occupies the same syntactic location.

It is a compile-time error if an annotation of type $\tau$ applies to a part of a type (that is, not the outermost level) in a type context, and $\tau$ is not applicable in type contexts.
It is a compile-time error if an annotation of type $T$ applies to a type (or any part of a type) in a type context, and $T$ is applicable in type contexts, and the annotation is not admissible.

For example, assume an annotation type $TA$ which is meta-annotated with just @Target(ElementType.TYPE_USE). The terms $@TA$ java.lang.Object and java.@TA lang.Object are illegal because the simple name to which $@TA$ is closest is classified as a package name. On the other hand, java.lang.$@TA$ Object is legal.

Note that the illegal terms are illegal "everywhere". The ban on annotating package names applies broadly: to locations which are solely type contexts, such as class ... extends $@TA$ java.lang.Object (...), and to locations which are both declaration and type contexts, such as $@TA$ java.lang.Object $f$; (There are no locations which are solely declaration contexts where a package name could be annotated, as class, package, and type parameter declarations use only simple names.)

If $TA$ is additionally meta-annotated with @Target(ElementType.FIELD), then the term $@TA$ java.lang.Object is legal in locations which are both declaration and type contexts, such as a field declaration $@TA$ java.lang.Object $f$; Here, $@TA$ is deemed to apply to the declaration of $f$ (and not to the type java.lang.Object) because $TA$ is applicable in the field declaration context.

### 9.7.5 Multiple Annotations of the Same Type

It is a compile-time error if multiple annotations of the same type $T$ appear in a declaration context or type context, unless $T$ is repeatable (§9.6.3) and both $T$ and the containing annotation type of $T$ are applicable in the declaration context or type context (§9.6.4.1).

It is customary, though not required, for multiple annotations of the same type to appear contiguously.

If a declaration context or type context has multiple annotations of a repeatable annotation type $T$, then it is as if the context has no explicitly declared annotations of type $T$ and one implicitly declared annotation of the containing annotation type of $T$.

The implicitly declared annotation is called the container annotation, and the multiple annotations of type $T$ which appeared in the context are called the base annotations. The elements of the (array-typed) value element of the container annotation are all the base annotations in the left-to-right order in which they appeared in the context.

It is a compile-time error if, in a declaration context or type context, there are multiple annotations of a repeatable annotation type $T$ and any annotations of the containing annotation type of $T$. 
In other words, it is not possible to repeat annotations where an annotation of the same type as their container also appears. This prohibits obtuse code like:

```java
@Foo(0) @Foo(1) @FooContainer({@Foo(2)})
class A {}
```

If this code was legal, then multiple levels of containment would be needed: first the annotations of type `Foo` would be contained by an implicitly declared container annotation of type `FooContainer`, then that annotation and the explicitly declared annotation of type `FooContainer` would be contained in yet another implicitly declared annotation. This complexity is undesirable in the judgment of the designers of the Java programming language. Another approach, treating the annotations of type `Foo` as if they had occurred alongside `@Foo(2)` in the explicit `@FooContainer` annotation, is undesirable because it could change how reflective programs interpret the `@FooContainer` annotation.

It is a compile-time error if, in a declaration context or type context, there is one annotation of a repeatable annotation type `T` and multiple annotations of the containing annotation type of `T`.

This rule is designed to allow the following code:

```java
@Foo(1) @FooContainer({@Foo(2)})
class A {}
```

With only one annotation of the repeatable annotation type `Foo`, no container annotation is implicitly declared, even if `FooContainer` is the containing annotation type of `Foo`. However, repeating the annotation of type `FooContainer`, as in:

```java
@Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)})
class A {}
```

is prohibited, even if `FooContainer` is repeatable with a containing annotation type of its own. It is obtuse to repeat annotations which are themselves containers when an annotation of the underlying repeatable type is present.

### 9.8 Functional Interfaces

A functional interface is an interface that has just one abstract method (aside from the methods of `Object`), and thus represents a single function contract. This "single" method may take the form of multiple abstract methods with override-equivalent signatures inherited from superinterfaces; in this case, the inherited methods logically represent a single method.

For an interface `I`, let `M` be the set of abstract methods that are members of `I` that do not have the same signature as any `public` instance method of the class `Object`
Then, \( I \) is a *functional interface* if there exists a method \( m \) in \( M \) for which both of the following are true:

- The signature of \( m \) is a subsignature (§8.4.2) of every method's signature in \( M \).
- \( m \) is return-type-substitutable (§8.4.5) for every method in \( M \).

In addition to the usual process of creating an interface instance by declaring and instantiating a class (§15.9), instances of functional interfaces can be created with method reference expressions and lambda expressions (§15.13, §15.27).

The definition of *functional interface* excludes methods in an interface that are also public methods in Object. This is to allow functional treatment of an interface like java.util.Comparator<T> that declares multiple abstract methods of which only one is really "new" - int compare(T, T). The other boolean equals(Object) - is an explicit declaration of an abstract method that would otherwise be implicitly declared in the interface (§9.2) and automatically implemented by every class that implements the interface.

Note that if non-public methods of Object, such as clone(), are explicitly declared in an interface as public, they are not automatically implemented by every class that implements the interface. The implementation inherited from Object is protected while the interface method is public, so the only way to implement the interface would be for a class to override the non-public Object method with a public method.

**Example 9.8-1. Functional Interfaces**

A simple example of a functional interface is:

```java
interface Runnable {
    void run();
}
```

The following interface is not functional because it declares nothing which is not already a member of Object:

```java
interface NonFunc {
    boolean equals(Object obj);
}
```

However, its subinterface can be functional by declaring an *abstract* method which is not a member of Object:

```java
interface Func extends NonFunc {
    int compare(String o1, String o2);
}
```

Similarly, the well known interface java.util.Comparator<T> is functional because it has one *abstract* non-Object method:
interface Comparator<T> {
    boolean equals(Object obj);
    int compare(T o1, T o2);
}

The following interface is not functional because while it only declares one abstract method which is not a member of Object, it declares two abstract methods which are not public members of Object:

interface Foo {
    int m();
    Object clone();
}

Example 9.8-2. Functional Interfaces and Erasure

In the following interface hierarchy, Z is a functional interface because while it inherits two abstract methods which are not members of Object, they have the same signature, so the inherited methods logically represent a single method:

interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {} 

Similarly, Z is a functional interface in the following interface hierarchy because Y.m is a subsignature of X.m and is return-type-substitutable for X.m:

interface X { Iterable m(Iterable<String> arg); }
interface Y { Iterable<String> m(Iterable arg); }
interface Z extends X, Y {} 

The definition of functional interface respects the fact that an interface cannot have two members which are not subsignatures of each other, yet have the same erasure (§9.4.1.2). Thus, in the following three interface hierarchies where Z causes a compile-time error, Z is not a functional interface: (because none of its abstract members are subsignatures of all other abstract members)

interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {} 

interface X { int m(Iterable<String> arg, Class c); }
interface Y { int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {} 

interface X<T> { void m(T arg); }
interface Y<T> { void m(T arg); }
interface Z<A, B> extends X<A>, Y<B> {} 

Similarly, the definition of "functional interface" respects the fact that an interface may only have methods with override-equivalent signatures if one is return-type-substitutable
for all the others. Thus, in the following interface hierarchy where \( Z \) causes a compile-time error, \( Z \) is not a functional interface: (because none of its abstract members are return-type-substitutable for all other abstract members)

```java
interface X { long m(); }
interface Y { int m(); }
interface Z extends X, Y {}
```

In the following example, the declarations of \( \text{Foo}<T,N> \) and \( \text{Bar} \) are legal: in each, the methods called \( m \) are not subsignatures of each other, but do have different erasures. Still, the fact that the methods in each are not subsignatures means \( \text{Foo}<T,N> \) and \( \text{Bar} \) are not functional interfaces. However, \( \text{Baz} \) is a functional interface because the methods it inherits from \( \text{Foo}<\text{Integer},\text{Integer}> \) have the same signature and so logically represent a single method.

```java
interface Foo<T, N extends Number> {
    void m(T arg);
    void m(N arg);
}
interface Bar extends Foo<String, Integer> {}
interface Baz extends Foo<Integer, Integer> {}  
```

Finally, the following examples demonstrate the same rules as above, but with generic methods:

```java
interface Exec { <T> T execute(Action<T> a); }  // Functional
interface X { <T> T execute(Action<T> a); }
interface Y { <S> S execute(Action<S> a); }
interface Exec extends X, Y {}  // Functional: signatures are logically "the same"
interface X { <T> T execute(Action<T> a); }
interface Y { <S,T> S execute(Action<S> a); }
interface Exec extends X, Y {}  // Error: different signatures, same erasure
```

**Example 9.8-3. Generic Functional Interfaces**

Functional interfaces can be generic, such as \( \text{java.util.function.Predicate}<T> \). Such a functional interface may be parameterized in a way that produces distinct abstract methods - that is, multiple methods that cannot be legally overridden with a single declaration. For example:

```java
interface I { Object m(Class c); }
interface J<S> { S m(Class<?> c); }
interface K<T> { T m(Class<?> c); }
interface Functional<S,T> extends I, J<S>, K<T> {}  
```
Functional\langle S, T \rangle is a functional interface - \( I.m \) is return-type-substitutable for \( J.m \) and \( K.m \) - but the functional interface type Functional\langle String, Integer \rangle clearly cannot be implemented with a single method. However, other parameterizations of Functional\langle S, T \rangle which are functional interface types are possible.

The declaration of a functional interface allows a functional interface type to be used in a program. There are four kinds of functional interface type:

- The type of a non-generic (§6.1) functional interface
- A parameterized type that is a parameterization (§4.5) of a generic functional interface
- The raw type (§4.8) of a generic functional interface
- An intersection type (§4.9) that induces a notional functional interface

In special circumstances, it is useful to treat an intersection type as a functional interface type. Typically, this will look like an intersection of a functional interface type with one or more marker interface types, such as Runnable & java.io.Serializable. Such an intersection can be used in casts (§15.16) that force a lambda expression to conform to a certain type. If one of the interface types in the intersection is java.io.Serializable, special run-time support for serialization is triggered (§15.27.4).

### 9.9 Function Types

The function type of a functional interface \( I \) is a method type (§8.2) that can be used to override (§8.4.8) the abstract method(s) of \( I \).

Let \( M \) be the set of abstract methods defined for \( I \). The function type of \( I \) consists of the following:

- Type parameters, formal parameter types, and return type:
  
  Let \( m \) be a method in \( M \) with:

  1. a signature that is a subsignature of every method's signature in \( M \); and
  2. a return type \( R \) (possibly void), where either \( R \) is the same as every method's return type in \( M \), or \( R \) is a reference type and is a subtype of every method's return type in \( M \) (after adapting for any type parameters (§8.4.4) if the two methods have the same signature).

  If no such method exists, then let \( m \) be a method in \( M \) with:

  1. a signature that is a subsignature of every method's signature in \( M \); and
2. a return type such that \( m \) is return-type-substitutable (§8.4.5) for every method in \( M \).

The function type's type parameters, formal parameter types, and return type are as given by \( m \).

• throws clause:

The function type's throws clause is derived from the throws clauses of the methods in \( M \), as follows:

1. If the function type is generic, the throws clauses are first adapted to the type parameters of the function type (§8.4.4).

   If the function type is not generic but at least one method in \( M \) is generic, the throws clauses are first erased.

2. Then, the function type's throws clause includes every type \( E \) which satisfies the following constraints:

   - \( E \) is mentioned in one of the throws clauses.

   - For each throws clause, \( E \) is a subtype of some type named in that clause.

When some return types in \( M \) are raw and others are not, the definition of a function type tries to choose the most specific type, if possible. For example, if the return types are LinkedList and LinkedList<String>, then the latter is immediately chosen as the function type's return type. When there is no most specific type, the definition compensates by finding the most substitutable return type. For example, if there is a third return type, List<?>, then it is not the case that one of the return types is a subtype of every other (as raw LinkedList is not a subtype of List<?>): instead, LinkedList<String> is chosen as the function type's return type because it is return-type-substitutable for both LinkedList and List<?>.

The goal driving the definition of a function type's thrown exception types is to support the invariant that a method with the resulting throws clause could override each abstract method of the functional interface. Per §8.4.6, this means the function type cannot throw "more" exceptions than any single method in the set \( M \), so we look for as many exception types as possible that are "covered" by every method's throws clause.

The function type of a functional interface type is specified as follows:

• The function type of the type of a non-generic functional interface \( I \) is simply the function type of the functional interface \( I \), as defined above.

• The function type of a parameterized functional interface type \( I<A_1...A_n> \), where \( A_1...A_n \) are types and the corresponding type parameters of \( I \) are \( P_1...P_n \), is derived by applying the substitution \([P_i:=A_i, \ldots, P_n:=A_n]\) to the function type of the generic functional interface \( I<P_1...P_n> \).
• The function type of a parameterized functional interface type $I_{<A_1...A_n>}$, where one or more of $A_1...A_n$ is a wildcard, is the function type of the non-wildcard parameterization of $I$, $I_{<T_1...T_n>}$. The non-wildcard parameterization is determined as follows.

Let $P_1...P_n$ be the type parameters of $I$ with corresponding bounds $B_1...B_n$. For all $i$ $(1 \leq i \leq n)$, $T_i$ is derived according to the form of $A_i$:

– If $A_i$ is a type, then $T_i = A_i$.

– If $A_i$ is a wildcard, and the corresponding type parameter’s bound, $B_i$, mentions one of $P_1...P_n$, then $T_i$ is undefined and there is no function type.

– Otherwise:
  » If $A_i$ is an unbound wildcard $?$, then $T_i = B_i$.
  » If $A_i$ is an upper-bounded wildcard $？$ extends $U_i$, then $T_i = \text{glb}(U_i, B_i)$ ($§5.1.10$).
  » If $A_i$ is a lower-bounded wildcard $？$ super $L_i$, then $T_i = L_i$.

• The function type of the raw type of a generic functional interface $I_{<...>}$ is the erasure of the function type of the generic functional interface $I_{<...>}$.

• The function type of an intersection type that induces a notional functional interface is the function type of the notional functional interface.

**Example 9.9-1. Function Types**

Given the following interfaces:

```java
interface X { void m() throws IOException; }
interface Y { void m() throws EOFException; }
interface Z { void m() throws ClassNotFound; }
```

the function type of:

```java
interface XY extends X, Y {
}
```

is:

```java
() -> void throws EOFException
```

while the function type of:

```java
interface XYZ extends X, Y, Z {
}
```

is:
Function Types

Given the following interfaces:

```java
interface A {
    List<String> foo(List<String> arg)
        throws IOException, SQLTransientException;
}
interface B {
    List foo(List<String> arg)
        throws EOFException, SQLException, TimeoutException;
}
interface C {
    List foo(List arg) throws Exception;
}
```

the function type of:

```java
interface D extends A, B {}
```

is:

```java
(List<String>) -> List<String>
    throws EOFException, SQLTransientException
```

while the function type of:

```java
interface E extends A, B, C {}
```

is:

```java
(List) -> List throws EOFException, SQLTransientException
```

The function type of a functional interface is defined nondeterministically: while the signatures in $M$ are "the same", they may be syntactically different (HashMap.Entry and Map.Entry, for example); the return type may be a subtype of every other return type, but there may be other return types that are also subtypes (List<?> and List<?> extends Object, for example); and the order of thrown types is unspecified. These distinctions are subtle, but they can sometimes be important. However, function types are not used in the Java programming language in such a way that the nondeterminism matters. Note that the return type and throws clause of a "most specific method" are also defined nondeterministically when there are multiple abstract methods (§15.12.2.5).

When a generic functional interface is parameterized by wildcards, there are many different instantiations that could satisfy the wildcard and produce different function types. For example, each of Predicate<Integer> (function type Integer -> boolean), Predicate<Number> (function type Number -> boolean), and Predicate<Object> (function type Object -> boolean) is a Predicate<? super Integer>. Sometimes, it is possible to known from the context, such as the parameter types of a lambda expression, which function type is intended (§15.27.3). Other times, it is necessary to pick one; in
these circumstances, the bounds are used. (This simple strategy cannot guarantee that the resulting type will satisfy certain complex bounds, so not all complex cases are supported.)

Example 9.9-2. Generic Function Types

A function type may be generic, as a functional interface's abstract method may be generic. For example, in the following interface hierarchy:

```java
interface G1 {
    <E extends Exception> Object m() throws E;
}
interface G2 {
    <F extends Exception> String m() throws Exception;
}
interface G extends G1, G2 {}
```

the function type of `G` is:

```java
<F extends Exception> () -> String throws F
```

A generic function type for a functional interface may be implemented by a method reference expression (§15.13), but not by a lambda expression (§15.27) as there is no syntax for generic lambda expressions.
CHAPTER 10

Arrays

In the Java programming language, arrays are objects (§4.3.1), are dynamically created, and may be assigned to variables of type Object (§4.3.2). All methods of class Object may be invoked on an array.

An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be empty. The variables contained in an array have no names; instead they are referenced by array access expressions that use non-negative integer index values. These variables are called the components of the array. If an array has $n$ components, we say $n$ is the length of the array; the components of the array are referenced using integer indices from 0 to $n - 1$, inclusive.

All the components of an array have the same type, called the component type of the array. If the component type of an array is $\tau$, then the type of the array itself is written $\tau[]$.

The value of an array component of type float is always an element of the float value set (§4.2.3); similarly, the value of an array component of type double is always an element of the double value set. It is not permitted for the value of an array component of type float to be an element of the float-extended-exponent value set that is not also an element of the float value set, nor for the value of an array component of type double to be an element of the double-extended-exponent value set that is not also an element of the double value set.

The component type of an array may itself be an array type. The components of such an array may contain references to subarrays. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the element type of the original array, and the components at this level of the data structure are called the elements of the original array.
There are some situations in which an element of an array can be an array: if the element type is `Object` or `Cloneable` or `java.io.Serializable`, then some or all of the elements may be arrays, because any array object can be assigned to any variable of these types.

### 10.1 Array Types

Array types are used in declarations and in cast expressions (§15.16).

An array type is written as the name of an element type followed by some number of empty pairs of square brackets `[]`. The number of bracket pairs indicates the depth of array nesting.

Each bracket pair in an array type may be annotated by type annotations (§9.7.4). An annotation applies to the bracket pair (or ellipsis, in a variable arity parameter declaration) that follows it.

The element type of an array may be any type, whether primitive or reference. In particular:

- Arrays with an interface type as the element type are allowed.
  
  An element of such an array may have as its value a null reference or an instance of any type that implements the interface.

- Arrays with an `abstract` class type as the element type are allowed.
  
  An element of such an array may have as its value a null reference or an instance of any subclass of the `abstract` class that is not itself `abstract`.

An array's length is not part of its type.

The supertypes of an array type are specified in §4.10.3.

The supertype relation for array types is not the same as the superclass relation. The direct supertype of `Integer[]` is `Number[]` according to §4.10.3, but the direct superclass of `Integer[]` is `Object` according to the `Class` object for `Integer[]` (§10.8). This does not matter in practice, because `Object` is also a supertype of all array types.

### 10.2 Array Variables

A variable of array type holds a reference to an object. Declaring a variable of array type does not create an array object or allocate any space for array components. It
creates only the variable itself, which can contain a reference to an array. However, the initializer part of a declarator (§8.3, §9.3, §14.4.1) may create an array, a reference to which then becomes the initial value of the variable.

Example 10.2-1. Declarations of Array Variables

```java
int[] ai;        // array of int
short[][] as;    // array of array of short
short s,         // scalar short
short[][] aas[]; // array of array of short
Object[] ao,     // array of Object
otherAo;         // array of Object
Collection<?>[] ca; // array of Collection of unknown type
```

The declarations above do not create array objects. The following are examples of declarations of array variables that do create array objects:

```java
Exception ae[] = new Exception[3];
Object aao[][] = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ', 'S', 't', 'r', 'i', 'n', 'g' };
String[] aas = { "array", "of", "String", };```

The array type of a variable depends on the bracket pairs that may appear as part of the type at the beginning of a variable declaration, or as part of the declarator for the variable, or both. Specifically, in the declaration of a field, formal parameter, or local variable (§8.3, §8.4.1, §9.3, §9.4, §14.4.1, §14.14.2, §15.27.1), the array type of the variable is denoted by:

- the element type that appears at the beginning of the declaration; then,
- any bracket pairs that follow the variable's Identifier in the declarator (not applicable for a variable arity parameter); then,
- any bracket pairs that appear in the type at the beginning of the declaration (where the ellipsis of a variable arity parameter is treated as a bracket pair).

The return type of a method (§8.4.5) may be an array type. The precise array type depends on the bracket pairs that may appear as part of the type at the beginning of the method declaration, or after the method's formal parameter list, or both. The array type is denoted by:

- the element type that appears in the Result; then,
- any bracket pairs that follow the formal parameter list; then,
- any bracket pairs that appear in the Result.
We do not recommend "mixed notation" in array variable declarations, where bracket pairs appear on both the type and in declarators; nor in method declarations, where bracket pairs appear both before and after the formal parameter list.

Example 10.2-2. Array Variables and Array Types

The local variable declaration statement:

```java
byte[] rowvector, colvector, matrix[];
```

is equivalent to:

```java
byte rowvector[], colvector[], matrix[][];
```

because the array type of each local variable is unchanged. Similarly, the local variable declaration statement:

```java
int a, b[], c[][];
```

is equivalent to the series of declaration statements:

```java
int a;
int[] b;
int[][] c;
```

Brackets are allowed in declarators as a nod to the tradition of C and C++. The general rules for variable declaration, however, permit brackets to appear on both the type and in declarators, so that the local variable declaration statement:

```java
float[][][] f, g, h[]; // Yechh!
```

is equivalent to the series of declarations:

```java
float[][][] f;
float[][] g;
float[] h;
```

Because of how array types are formed, the following parameter declarations have the same array type:

```java
void m(int @A [] @B [] x) {}  
void n(int @A [] @B ... y) {}  
```

And perhaps surprisingly, the following field declarations have the same array type:

```java
int @A [] f @B [];
int @B [] @A [] g;
```
Once an array object is created, its length never changes. To make an array variable refer to an array of different length, a reference to a different array must be assigned to the variable.

A single variable of array type may contain references to arrays of different lengths, because an array's length is not part of its type.

If an array variable \(v\) has type \(A[\]\), where \(A\) is a reference type, then \(v\) can hold a reference to an instance of any array type \(B[\]\), provided \(B\) can be assigned to \(A\) (§5.2). This may result in a run-time exception on a later assignment; see §10.5 for a discussion.

### 10.3 Array Creation

An array is created by an array creation expression (§15.10.1) or an array initializer (§10.6).

An array creation expression specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting. The array's length is available as a final instance variable \texttt{length}.

An array initializer creates an array and provides initial values for all its components.

### 10.4 Array Access

A component of an array is accessed by an array access expression (§15.10.3) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by \([\) and \(]\), as in \(A[i]\).

All arrays are 0-origin. An array with length \(n\) can be indexed by the integers 0 to \(n-1\).

**Example 10.4-1. Array Access**

```java
class Gauss {
    public static void main(String[] args) {
        int[] ia = new int[101];
        for (int i = 0; i < ia.length; i++) ia[i] = i;
        int sum = 0;
        for (int e : ia) sum += e;
        System.out.println(sum);
    }
}
```
This program produces the output:

5050

The program declares a variable `ia` that has type array of `int`, that is, `int[]`. The variable `ia` is initialized to reference a newly created array object, created by an array creation expression (§15.10.1). The array creation expression specifies that the array should have 101 components. The length of the array is available using the field `length`, as shown. The program fills the array with the integers from 0 to 100, sums these integers, and prints the result.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion (§5.6.1) and become `int` values.

An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown (§15.10.4).

### 10.5 Array Store Exception

For an array whose type is `A[]`, where `A` is a reference type, an assignment to a component of the array is checked at run time to ensure that the value being assigned is assignable to the component.

If the type of the value being assigned is not assignment-compatible (§5.2) with the component type, an `ArrayStoreException` is thrown.

If the component type of an array were not reifiable (§4.7), the Java Virtual Machine could not perform the store check described in the preceding paragraph. This is why an array creation expression with a non-reifiable element type is forbidden (§15.10.1). One may declare a variable of an array type whose element type is non-reifiable, but assignment of the result of an array creation expression to the variable will necessarily cause an unchecked warning (§5.1.9).

**Example 10.5-1. ArrayStoreException**

```java
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    public static void main(String[] args) {
```
ColoredPoint[] cpa = new ColoredPoint[10];
Point[] pa = cpa;
System.out.println(pa[1] == null);
try {
    pa[0] = new Point();
} catch (ArrayStoreException e) {
    System.out.println(e);
}
}

This program produces the output:

true
java.lang.ArrayStoreException: Point

The variable \texttt{pa} has type \texttt{Point[]} and the variable \texttt{cpa} has as its value a reference to an object of type \texttt{ColoredPoint[]}. A \texttt{ColoredPoint} can be assigned to a \texttt{Point}; therefore, the value of \texttt{cpa} can be assigned to \texttt{pa}.

A reference to this array \texttt{pa}, for example, testing whether \texttt{pa[1]} is \texttt{null}, will not result in a run-time type error. This is because the element of the array of type \texttt{ColoredPoint[]} is a \texttt{ColoredPoint}, and every \texttt{ColoredPoint} can stand in for a \texttt{Point}, since \texttt{Point} is the superclass of \texttt{ColoredPoint}.

On the other hand, an assignment to the array \texttt{pa} can result in a run-time error. At compile time, an assignment to an element of \texttt{pa} is checked to make sure that the value assigned is a \texttt{Point}. But since \texttt{pa} holds a reference to an array of \texttt{ColoredPoint}, the assignment is valid only if the type of the value assigned at run time is, more specifically, a \texttt{ColoredPoint}.

The Java Virtual Machine checks for such a situation at run time to ensure that the assignment is valid; if not, an \texttt{ArrayStoreException} is thrown.

10.6 Array Initializers

An \textit{array initializer} may be specified in a field declaration ($\S8.3$, $\S9.3$) or local variable declaration ($\S14.4$), or as part of an array creation expression ($\S15.10.1$), to create an array and provide some initial values.

\textit{ArrayInitializer:}
\[ \{ \text{VariableInitializerList} \} \]

\textit{VariableInitializerList:}
\[ \text{VariableInitializer} \rightarrow \text{VariableInitializer} \{ , \} \]

The following production from $\S8.3$ is shown here for convenience:
An array initializer is written as a comma-separated list of expressions, enclosed by braces `{ and `}`.

A trailing comma may appear after the last expression in an array initializer and is ignored.

Each variable initializer must be assignment-compatible (§5.2) with the array's component type, or a compile-time error occurs.

It is a compile-time error if the component type of the array being initialized is not reifiable (§4.7).

The length of the array to be constructed is equal to the number of variable initializers immediately enclosed by the braces of the array initializer. Space is allocated for a new array of that length. If there is insufficient space to allocate the array, evaluation of the array initializer completes abruptly by throwing an `OutOfMemoryError`. Otherwise, a one-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.12.5).

The variable initializers immediately enclosed by the braces of the array initializer are then executed from left to right in the textual order they occur in the source code. The n'th variable initializer specifies the value of the n-1'th array component. If execution of a variable initializer completes abruptly, then execution of the array initializer completes abruptly for the same reason. If all the variable initializer expressions complete normally, the array initializer completes normally, with the value of the newly initialized array.

If the component type is an array type, then the variable initializer specifying a component may itself be an array initializer; that is, array initializers may be nested. In this case, execution of the nested array initializer constructs and initializes an array object by recursive application of the algorithm above, and assigns it to the component.

Example 10.6-1. Array Initializers

```java
class Test {
    public static void main(String[] args) {
        int ia[][] = { {1, 2}, null };
        for (int[] ea : ia) {
            for (int e: ea) {
                System.out.println(e);
            }
        }
    }
}
```
This program produces the output:

```
1
2
```

before causing a `NullPointerException` in trying to index the second component of the array `ia`, which is a null reference.

### 10.7 Array Members

The members of an array type are all of the following:

- **The public final field** `length`, which contains the number of components of the array. `length` may be positive or zero.

- **The public method** `clone`, which overrides the method of the same name in class `Object` and throws no checked exceptions. The return type of the `clone` method of an array type `T[]` is `T[]`.

  A clone of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared.

- **All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method.**

  See §9.6.4.4 for another situation where the difference between public and non-public methods of `Object` requires special care.

An array thus has the same public fields and methods as the following class:

```java
class A<T> implements Cloneable, java.io.Serializable {
    public final int length = X;
    public T[] clone() {
        try {
            return (T[])super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.getMessage());
        }
    }
}
```

Note that the cast to `T[]` in the code above would generate an unchecked warning (§5.1.9) if arrays were really implemented this way.
10.8 Class Objects for Arrays

Example 10.7-1. Arrays Are Cloneable

```java
class Test1 {
    public static void main(String[] args) {
        int ia1[] = { 1, 2 };
        int ia2[] = ia1.clone();
        System.out.print((ia1 == ia2) + " ");
        ia1[1]++;
        System.out.println(ia2[1]);
    }
}
```

This program produces the output:

```
false 2
```

showing that the components of the arrays referenced by ia1 and ia2 are different variables.

Example 10.7-2. Shared Subarrays After A Clone

The fact that subarrays are shared when a multidimensional array is cloned is shown by this program:

```java
class Test2 {
    public static void main(String[] args) throws Throwable {
        int ia[][] = { {1,2}, null };
        int ja[][] = ia.clone();
        System.out.print((ia == ja) + " ");
        System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
    }
}
```

This program produces the output:

```
false true
```

showing that the int[] array that is ia[0] and the int[] array that is ja[0] are the same array.

10.8 Class Objects for Arrays

Every array has an associated Class object, shared with all other arrays with the same component type.

Although an array type is not a class, the Class object of every array acts as if:
• The direct superclass of every array type is Object.
• Every array type implements the interfaces Cloneable and java.io.Serializable.

**Example 10.8-1. Class Object Of Array**

```java
class Test1 {
    public static void main(String[] args) {
        int[] ia = new int[3];
        System.out.println(ia.getClass());
        System.out.println(ia.getClass().getSuperclass());
        for (Class<?> c : ia.getClass().getInterfaces())
            System.out.println("Superinterface: " + c);
    }
}
```

This program produces the output:

```
class [I
class java.lang.Object
Superinterface: interface java.lang.Cloneable
Superinterface: interface java.io.Serializable
```

where the string "[I" is the run-time type signature for the Class object "array with component type int".

**Example 10.8-2. Array Class Objects Are Shared**

```java
class Test2 {
    public static void main(String[] args) {
        int[] ia = new int[3];
        int[] ib = new int[6];
        System.out.println(ia == ib);
        System.out.println(ia.getClass() == ib.getClass());
    }
}
```

This program produces the output:

```
false
true
```

While `ia` and `ib` refer to different arrays, the result of the comparison of the `Class` objects demonstrates that all arrays whose components are of type `int` are instances of the same array type (namely `int[]`).
10.9 An Array of Characters Is Not a String

In the Java programming language, unlike C, an array of char is not a String, and neither a String nor an array of char is terminated by '\u0000' (the NUL character).

A String object is immutable, that is, its contents never change, while an array of char has mutable elements.

The method toCharArray in class String returns an array of characters containing the same character sequence as a String. The class StringBuffer implements useful methods on mutable arrays of characters.
When a program violates the semantic constraints of the Java programming language, the Java Virtual Machine signals this error to the program as an exception.

An example of such a violation is an attempt to index outside the bounds of an array. Some programming languages and their implementations react to such errors by peremptorily terminating the program; other programming languages allow an implementation to react in an arbitrary or unpredictable way. Neither of these approaches is compatible with the design goals of the Java SE Platform: to provide portability and robustness.

Instead, the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer.

An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred.

Programs can also throw exceptions explicitly, using throw statements (§14.18).

Explicit use of throw statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value -1 where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both.

Every exception is represented by an instance of the class Throwable or one of its subclasses (§11.1). Such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by catch clauses of try statements (§14.20).
During the process of throwing an exception, the Java Virtual Machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception (§11.2). If no such handler is found, then the exception may be handled by one of a hierarchy of uncaught exception handlers (§11.3) - thus every effort is made to avoid letting an exception go unhandled.

The exception mechanism of the Java SE Platform is integrated with its synchronization model (§17.1), so that monitors are unlocked as synchronized statements (§14.19) and invocations of synchronized methods (§8.4.3.6, §15.12) complete abruptly.

11.1 The Kinds and Causes of Exceptions

11.1.1 The Kinds of Exceptions

An exception is represented by an instance of the class Throwable (a direct subclass of Object) or one of its subclasses.

Throwable and all its subclasses are, collectively, the exception classes.

The classes Exception and Error are direct subclasses of Throwable:

- Exception is the superclass of all the exceptions from which ordinary programs may wish to recover.

  The class RuntimeException is a direct subclass of Exception. RuntimeException is the superclass of all the exceptions which may be thrown for many reasons during expression evaluation, but from which recovery may still be possible.

  RuntimeException and all its subclasses are, collectively, the run-time exception classes.

- Error is the superclass of all the exceptions from which ordinary programs are not ordinarily expected to recover.

  Error and all its subclasses are, collectively, the error classes.

The unchecked exception classes are the run-time exception classes and the error classes.
The *checked exception classes* are all exception classes other than the unchecked exception classes. That is, the checked exception classes are `Throwable` and all its subclasses other than `RuntimeException` and its subclasses and `Error` and its subclasses.

Programs can use the pre-existing exception classes of the Java SE Platform API in `throw` statements, or define additional exception classes as subclasses of `Throwable` or of any of its subclasses, as appropriate. To take advantage of compile-time checking for exception handlers (§11.2), it is typical to define most new exception classes as checked exception classes, that is, as subclasses of `Exception` that are not subclasses of `RuntimeException`.

The class `Error` is a separate subclass of `Throwable`, distinct from `Exception` in the class hierarchy, to allow programs to use the idiom `} catch (Exception e) {
` (§11.2.3) to catch all exceptions from which recovery may be possible without catching errors from which recovery is typically not possible.

Note that a subclass of `Throwable` cannot be generic (§8.1.2).

### 11.1.2 The Causes of Exceptions

An exception is thrown for one of three reasons:

- A `throw` statement (§14.18) was executed.
- An abnormal execution condition was synchronously detected by the Java Virtual Machine, namely:
  - evaluation of an expression violates the normal semantics of the Java programming language (§15.6), such as an integer divide by zero.
  - an error occurs while loading, linking, or initializing part of the program (§12.2, §12.3, §12.4); in this case, an instance of a subclass of `LinkageError` is thrown.
  - an internal error or resource limitation prevents the Java Virtual Machine from implementing the semantics of the Java programming language; in this case, an instance of a subclass of `VirtualMachineError` is thrown.

These exceptions are not thrown at an arbitrary point in the program, but rather at a point where they are specified as a possible result of an expression evaluation or statement execution.

- An asynchronous exception occurred (§11.1.3).
11.1.3 Asynchronous Exceptions

Most exceptions occur synchronously as a result of an action by the thread in which they occur, and at a point in the program that is specified to possibly result in such an exception. An asynchronous exception is, by contrast, an exception that can potentially occur at any point in the execution of a program.

Asynchronous exceptions occur only as a result of:

- An invocation of the (deprecated) `stop` method of class `Thread` or `ThreadGroup`.

  The (deprecated) `stop` methods may be invoked by one thread to affect another thread or all the threads in a specified thread group. They are asynchronous because they may occur at any point in the execution of the other thread or threads.

- An internal error or resource limitation in the Java Virtual Machine that prevents it from implementing the semantics of the Java programming language. In this case, the asynchronous exception that is thrown is an instance of a subclass of `VirtualMachineError`.

  Note that `StackOverflowError`, a subclass of `VirtualMachineError`, may be thrown synchronously by method invocation (§15.12.4.5) as well as asynchronously due to native method execution or Java Virtual Machine resource limitations. Similarly, `OutOfMemoryError`, another subclass of `VirtualMachineError`, may be thrown synchronously during class instance creation (§15.9.4, §12.5), array creation (§15.10.2, §10.6), class initialization (§12.4.2), and boxing conversion (§5.1.7), as well as asynchronously.

The Java SE Platform permits a small but bounded amount of execution to occur before an asynchronous exception is thrown.

Asynchronous exceptions are rare, but proper understanding of their semantics is necessary if high-quality machine code is to be generated.

The delay noted above is permitted to allow optimized code to detect and throw these exceptions at points where it is practical to handle them while obeying the semantics of the Java programming language. A simple implementation might poll for asynchronous exceptions at the point of each control transfer instruction. Since a program has a finite size, this provides a bound on the total delay in detecting an asynchronous exception. Since no asynchronous exception will occur between control transfers, the code generator has some flexibility to reorder computation between control transfers for greater performance. The paper *Polling Efficiently on Stock Hardware* by Marc Feeley, *Proc. 1993 Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, pp. 179-187, is recommended as further reading.
11.2 Compile-Time Checking of Exceptions

The Java programming language requires that a program contains handlers for checked exceptions which can result from execution of a method or constructor (§8.4.6, §8.8.5). This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled. For each checked exception which is a possible result, the throws clause for the method or constructor must mention the class of that exception or one of the superclasses of the class of that exception (§11.2.3).

The checked exception classes (§11.1.1) named in the throws clause are part of the contract between the implementor and user of the method or constructor. The throws clause of an overriding method may not specify that this method will result in throwing any checked exception which the overridden method is not permitted, by its throws clause, to throw (§8.4.8.3). When interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a throws clause that is compatible with all the overridden declarations (§9.4.1).

The unchecked exception classes (§11.1.1) are exempted from compile-time checking.

Error classes are exempted because they can occur at many points in the program and recovery from them is difficult or impossible. A program declaring such exceptions would be cluttered, pointlessly. Sophisticated programs may yet wish to catch and attempt to recover from some of these conditions.

Run-time exception classes are exempted because, in the judgment of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs. Many of the operations and constructs of the Java programming language can result in exceptions at run time. The information available to a Java compiler, and the level of analysis a compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irritation to programmers.

For example, certain code might implement a circular data structure that, by construction, can never involve null references; the programmer can then be certain that a NullPointerException cannot occur, but it would be difficult for a Java compiler to prove it. The theorem-proving technology that is needed to establish such global properties of data structures is beyond the scope of this specification.

We say that a statement or expression can throw an exception class \( E \) if, according to the rules in §11.2.1 and §11.2.2, the execution of the statement or expression can result in an exception of class \( E \) being thrown.
We say that a `catch` clause *can catch* its catchable exception class(es):

- The catchable exception class of a uni-*catch* clause is the declared type of its exception parameter (§14.20).
- The catchable exception classes of a multi-*catch* clause are the alternatives in the union that denotes the type of its exception parameter.

### 11.2.1 Exception Analysis of Expressions

A class instance creation expression (§15.9) can throw an exception class \( E \) iff either:

- The expression is a qualified class instance creation expression and the qualifying expression can throw \( E \); or
- Some expression of the argument list can throw \( E \); or
- \( E \) is one of the exception types of the invocation type of the chosen constructor (§15.12.2.6); or
- The class instance creation expression includes a `ClassBody`, and some instance initializer or instance variable initializer in the `ClassBody` can throw \( E \).

A method invocation expression (§15.12) can throw an exception class \( E \) iff either:

- The method invocation expression is of the form `Primary . [TypeArguments] Identifier` and the `Primary` expression can throw \( E \); or
- Some expression of the argument list can throw \( E \); or
- \( E \) is one of the exception types of the invocation type of the chosen method (§15.12.2.6).

A lambda expression (§15.27) can throw no exception classes.

For every other kind of expression, the expression can throw an exception class \( E \) iff one of its immediate subexpressions can throw \( E \).

Note that a method reference expression (§15.13) of the form `Primary : : [TypeArguments] Identifier` can throw an exception class if the `Primary` subexpression can throw an exception class. In contrast, a lambda expression can throw nothing, and has no immediate subexpressions on which to perform exception analysis. It is the *body* of a lambda expression, containing expressions and statements, that can throw exception classes.
11.2.2 Exception Analysis of Statements

A `throw` statement (§14.18) whose thrown expression has static type `E` and is not a final or effectively final exception parameter can throw `E` or any exception class that the thrown expression can throw.

For example, the statement `throw new java.io.FileNotFoundException();` can throw `java.io.FileNotFoundException` only. Formally, it is not the case that it "can throw" a subclass or superclass of `java.io.FileNotFoundException`.

A `throw` statement whose thrown expression is a final or effectively final exception parameter of a `catch` clause `C` can throw an exception class `E` iff:

- `E` is an exception class that the `try` block of the `try` statement which declares `C` can throw; and
- `E` is assignment compatible with any of `C`'s catchable exception classes; and
- `E` is not assignment compatible with any of the catchable exception classes of the `catch` clauses declared to the left of `C` in the same `try` statement.

A `try` statement (§14.20) can throw an exception class `E` iff either:

- The `try` block can throw `E`, or an expression used to initialize a resource (in a `try-with-resources` statement) can throw `E`, or the automatic invocation of the `close()` method of a resource (in a `try-with-resources` statement) can throw `E`, and `E` is not assignment compatible with any catchable exception class of any `catch` clause of the `try` statement, and either no `finally` block is present or the `finally` block can complete normally; or
- Some `catch` block of the `try` statement can throw `E` and either no `finally` block is present or the `finally` block can complete normally; or
- A `finally` block is present and can throw `E`.

An explicit constructor invocation statement (§8.8.7.1) can throw an exception class `E` iff either:

- Some expression of the constructor invocation's parameter list can throw `E`; or
- `E` is determined to be an exception class of the `throws` clause of the constructor that is invoked (§15.12.2.6).

Any other statement `S` can throw an exception class `E` iff an expression or statement immediately contained in `S` can throw `E`.
11.2.3 Exception Checking

It is a compile-time error if a method or constructor body can throw some exception class $E$ when $E$ is a checked exception class and $E$ is not a subclass of some class declared in the throws clause of the method or constructor.

It is a compile-time error if a lambda body can throw some exception class $E$ when $E$ is a checked exception class and $E$ is not a subclass of some class declared in the throws clause of the function type targeted by the lambda expression.

It is a compile-time error if a class variable initializer (§8.3.2) or static initializer (§8.7) of a named class or interface can throw a checked exception class.

It is a compile-time error if an instance variable initializer (§8.3.2) or instance initializer (§8.6) of a named class can throw a checked exception class, unless the named class has at least one explicitly declared constructor and the exception class or one of its superclasses is explicitly declared in the throws clause of each constructor.

Note that no compile-time error is due if an instance variable initializer or instance initializer of an anonymous class (§15.9.5) can throw an exception class. In a named class, it is the responsibility of the programmer to propagate information about which exception classes can be thrown by initializers, by declaring a suitable throws clause on any explicit constructor declaration. This relationship between the checked exception classes thrown by a class’s initializers and the checked exception classes declared by a class’s constructors is assured for an anonymous class declaration, because no explicit constructor declarations are possible and a Java compiler always generates a constructor with a suitable throws clause for the anonymous class declaration based on the checked exception classes that its initializers can throw.

It is a compile-time error if a catch clause can catch checked exception class $E_1$ and it is not the case that the try block corresponding to the catch clause can throw a checked exception class that is a subclass or superclass of $E_1$, unless $E_1$ is Exception or a superclass of Exception.

It is a compile-time error if a catch clause can catch an exception class $E_1$ and a preceding catch clause of the immediately enclosing try statement can catch $E_1$ or a superclass of $E_1$.

A Java compiler is encouraged to issue a warning if a catch clause can catch checked exception class $E_1$ and the try block corresponding to the catch clause can throw checked exception class $E_2$, where $E_2 <: E_1$, and a preceding catch clause of the immediately enclosing try statement can catch checked exception class $E_3$, where $E_2 <: E_3 <: E_1$.

Example 11.2.3-1. Catching Checked Exceptions

```java
import java.io.*;
```
class StaticallyThrownExceptionsIncludeSubtypes {
    public static void main(String[] args) {
        try {
            throw new FileNotFoundException();
        } catch (IOException ioe) {
            // "catch IOException" catches IOException
            // and any subtype.
        }

        try {
            throw new FileNotFoundException();
            // Statement "can throw" FileNotFoundException.
            // It is not the case that statement "can throw"
            // a subtype or supertype of FileNotFoundException.
        } catch (FileNotFoundException fnfe) {
            // ... Handle exception ...
        } catch (IOException ioe) {
            // Legal, but compilers are encouraged to give
            // warnings as of Java SE 7, because all subtypes of
            // IOException that the try block "can throw" have
            // already been caught by the prior catch clause.
        }

        try {
            m();
            // m's declaration says "throws IOException", so
            // m "can throw" IOException. It is not the case
            // that m "can throw" a subtype or supertype of
            // IOException (e.g. Exception).
        } catch (FileNotFoundException fnfe) {
            // Legal, because the dynamic type of the exception
            // might be FileNotFoundException.
        } catch (IOException ioe) {
            // Legal, because the dynamic type of the exception
            // might be a different subtype of IOException.
        } catch (Throwable t) {
            // Can always catch Throwable.
        }
    }

    static void m() throws IOException {
        throw new FileNotFoundException();
    }
}

By the rules above, each alternative in a multi-catch clause (§14.20) must be able to catch some exception class thrown by the try block and uncaught by previous catch clauses. For example, the second catch clause below would cause a compile-time error because exception analysis determines that SubclassOfFoo is already caught by the first catch clause:
11.3 Run-Time Handling of an Exception

When an exception is thrown (§14.18), control is transferred from the code that caused the exception to the nearest dynamically enclosing catch clause, if any, of a try statement (§14.20) that can handle the exception.

A statement or expression is dynamically enclosed by a catch clause if it appears within the try block of the try statement of which the catch clause is a part, or if the caller of the statement or expression is dynamically enclosed by the catch clause.

The caller of a statement or expression depends on where it occurs:

- If within a method, then the caller is the method invocation expression (§15.12) that was executed to cause the method to be invoked.

- If within a constructor or an instance initializer or the initializer for an instance variable, then the caller is the class instance creation expression (§15.9) or the method invocation of newInstance that was executed to cause an object to be created.

- If within a static initializer or an initializer for a static variable, then the caller is the expression that used the class or interface so as to cause it to be initialized (§12.4).

Whether a particular catch clause can handle an exception is determined by comparing the class of the object that was thrown to the catchable exception classes of the catch clause. The catch clause can handle the exception if one of its catchable exception classes is the class of the exception or a superclass of the class of the exception.

Equivalently, a catch clause will catch any exception object that is an instanceof (§15.20.2) one of its catchable exception classes.

The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception; execution then continues by executing the block of that catch clause. The code that caused the exception is never resumed.
All exceptions (synchronous and asynchronous) are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated.

If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program.

If no catch clause that can handle an exception can be found, then the current thread (the thread that encountered the exception) is terminated. Before termination, all finally clauses are executed and the uncaught exception is handled according to the following rules:

- If the current thread has an uncaught exception handler set, then that handler is executed.

- Otherwise, the method uncaughtException is invoked for the ThreadGroup that is the parent of the current thread. If the ThreadGroup and its parent ThreadGroups do not override uncaughtException, then the default handler’s uncaughtException method is invoked.

In situations where it is desirable to ensure that one block of code is always executed after another, even if that other block of code completes abruptly, a try statement with a finally clause (§14.20.2) may be used.

If a try or catch block in a try-finally or try-catch-finally statement completes abruptly, then the finally clause is executed during propagation of the exception, even if no matching catch clause is ultimately found.

If a finally clause is executed because of abrupt completion of a try block and the finally clause itself completes abruptly, then the reason for the abrupt completion of the try block is discarded and the new reason for abrupt completion is propagated from there.

The exact rules for abrupt completion and for the catching of exceptions are specified in detail with the specification of each statement in §14 (Blocks and Statements) and for expressions in §15 (Expressions) (especially §15.6).

Example 11.3-1. Throwing and Catching Exceptions

The following program declares an exception class TestException. The main method of class Test invokes the thrower method four times, causing exceptions to be thrown three of the four times. The try statement in method main catches each exception that the thrower throws. Whether the invocation of thrower completes normally or abruptly, a message is printed describing what happened.
class TestException extends Exception {
    TestException() { super(); }
    TestException(String s) { super(s); }
}

class Test {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                thrower(arg);
                System.out.println("Test "+ arg + 
                        " didn't throw an exception");
            } catch (Exception e) {
                System.out.println("Test "+ arg + 
                        " threw a " + e.getClass() + 
                        "\n                with message: " + 
                        e.getMessage());
            }
        }
    }
}

static int thrower(String s) throws TestException {
    try {
        if (s.equals("divide")) {
            int i = 0;
            return i/i;
        }
        if (s.equals("null")) {
            s = null;
            return s.length();
        }
        if (s.equals("test")) {
            throw new TestException("Test message");
        }
        return 0;
    } finally {
        System.out.println("[thrower(" + s + "] done");
    }
}

If we execute the program, passing it the arguments:

    divide null not test

it produces the output:
Test "divide" threw a class java.lang.ArithmeticException with message: / by zero
Test "null" threw a class java.lang.NullPointerException with message: null
Test "not" didn't throw an exception
Test "test" threw a class TestException with message: Test message

The declaration of the method thrower must have a throws clause because it can throw instances of TestException, which is a checked exception class (§11.1.1). A compile-time error would occur if the throws clause were omitted.

Notice that the finally clause is executed on every invocation of thrower, whether or not an exception occurs, as shown by the "[thrower(...) done]" output that occurs for each invocation.
This chapter specifies activities that occur during execution of a program. It is organized around the life cycle of the Java Virtual Machine and of the classes, interfaces, and objects that form a program.

The Java Virtual Machine starts up by loading a specified class or interface, then invoking the method `main` in this specified class or interface. Section §12.1 outlines the loading, linking, and initialization steps involved in executing `main`, as an introduction to the concepts in this chapter. Further sections specify the details of loading (§12.2), linking (§12.3), and initialization (§12.4).

The chapter continues with a specification of the procedures for creation of new class instances (§12.5); and finalization of class instances (§12.6). It concludes by describing the unloading of classes (§12.7) and the procedure followed when a program exits (§12.8).

### 12.1 Java Virtual Machine Startup

The Java Virtual Machine starts execution by invoking the method `main` of some specified class or interface, passing it a single argument which is an array of strings. In the examples in this specification, this first class is typically called `Test`.

The precise semantics of Java Virtual Machine startup are given in Chapter 5 of *The Java Virtual Machine Specification, Java SE 11 Edition*. Here we present an overview of the process from the viewpoint of the Java programming language.

The manner in which the initial class or interface is specified to the Java Virtual Machine is beyond the scope of this specification, but it is typical, in host environments that use command lines, for the fully qualified name of the class or interface to be specified as a command line argument and for following command
line arguments to be used as strings to be provided as the argument to the method main.

For example, in a UNIX implementation, the command line:

```
java Test reboot Bob Dot Enzo
```

will typically start a Java Virtual Machine by invoking method `main` of class `Test` (a class in an unnamed package), passing it an array containing the four strings "reboot", "Bob", "Dot", and "Enzo".

We now outline the steps the Java Virtual Machine may take to execute `Test`, as an example of the loading, linking, and initialization processes that are described further in later sections.

### 12.1.1 Load the Class `Test`

The initial attempt to execute the method `main` of class `Test` discovers that the class `Test` is not loaded - that is, that the Java Virtual Machine does not currently contain a binary representation for this class. The Java Virtual Machine then uses a class loader to attempt to find such a binary representation. If this process fails, then an error is thrown. This loading process is described further in §12.2.

### 12.1.2 Link `Test`: Verify, Prepare, (Optionally) Resolve

After `Test` is loaded, it must be initialized before `main` can be invoked. And `Test`, like all (class or interface) types, must be linked before it is initialized. Linking involves verification, preparation, and (optionally) resolution. Linking is described further in §12.3.

Verification checks that the loaded representation of `Test` is well-formed, with a proper symbol table. Verification also checks that the code that implements `Test` obeys the semantic requirements of the Java programming language and the Java Virtual Machine. If a problem is detected during verification, then an error is thrown. Verification is described further in §12.3.1.

Preparation involves allocation of static storage and any data structures that are used internally by the implementation of the Java Virtual Machine, such as method tables. Preparation is described further in §12.3.2.

Resolution is the process of checking symbolic references from `Test` to other classes and interfaces, by loading the other classes and interfaces that are mentioned and checking that the references are correct.
The resolution step is optional at the time of initial linkage. An implementation may resolve symbolic references from a class or interface that is being linked very early, even to the point of resolving all symbolic references from the classes and interfaces that are further referenced, recursively. (This resolution may result in errors from these further loading and linking steps.) This implementation choice represents one extreme and is similar to the kind of "static" linkage that has been done for many years in simple implementations of the C language. (In these implementations, a compiled program is typically represented as an "a.out" file that contains a fully-linked version of the program, including completely resolved links to library routines used by the program. Copies of these library routines are included in the "a.out" file.)

An implementation may instead choose to resolve a symbolic reference only when it is actively used; consistent use of this strategy for all symbolic references would represent the "laziest" form of resolution. In this case, if Test had several symbolic references to another class, then the references might be resolved one at a time, as they are used, or perhaps not at all, if these references were never used during execution of the program.

The only requirement on when resolution is performed is that any errors detected during resolution must be thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error. Using the "static" example implementation choice described above, loading and linkage errors could occur before the program is executed if they involved a class or interface mentioned in the class Test or any of the further, recursively referenced, classes and interfaces. In a system that implemented the "laziest" resolution, these errors would be thrown only when an incorrect symbolic reference is actively used.

The resolution process is described further in §12.3.3.

12.1.3 Initialize Test: Execute Initializers

In our continuing example, the Java Virtual Machine is still trying to execute the method main of class Test. This is permitted only if the class has been initialized (§12.4.1).

Initialization consists of execution of any class variable initializers and static initializers of the class Test, in textual order. But before Test can be initialized, its direct superclass must be initialized, as well as the direct superclass of its direct superclass, and so on, recursively. In the simplest case, Test has Object as its implicit direct superclass; if class Object has not yet been initialized, then it must
be initialized before Test is initialized. Class Object has no superclass, so the recursion terminates here.

If class Test has another class Super as its superclass, then Super must be initialized before Test. This requires loading, verifying, and preparing Super if this has not already been done and, depending on the implementation, may also involve resolving the symbolic references from Super and so on, recursively.

Initialization may thus cause loading, linking, and initialization errors, including such errors involving other types.

The initialization process is described further in §12.4.

12.1.4 Invoke Test.main

Finally, after completion of the initialization for class Test (during which other consequential loading, linking, and initializing may have occurred), the method main of Test is invoked.

The method main must be declared public, static, and void. It must specify a formal parameter (§8.4.1) whose declared type is array of String. Therefore, either of the following declarations is acceptable:

\[\text{public static void main(String[]} \text{args})\]
\[\text{public static void main(String... args)}\]

12.2 Loading of Classes and Interfaces

Loading refers to the process of finding the binary form of a class or interface type with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source code by a Java compiler, and constructing, from that binary form, a Class object to represent the class or interface.

The precise semantics of loading are given in Chapter 5 of The Java Virtual Machine Specification, Java SE 11 Edition. Here we present an overview of the process from the viewpoint of the Java programming language.

The binary format of a class or interface is normally the class file format described in The Java Virtual Machine Specification, Java SE 11 Edition cited above, but other formats are possible, provided they meet the requirements specified in §13.1.
The method `defineClass` of class `ClassLoader` may be used to construct `Class` objects from binary representations in the `class` file format.

Well-behaved class loaders maintain these properties:

- Given the same name, a good class loader should always return the same class object.

- If a class loader $L_1$ delegates loading of a class $C$ to another loader $L_2$, then for any type $T$ that occurs as the direct superclass or a direct superinterface of $C$, or as the type of a field in $C$, or as the type of a formal parameter of a method or constructor in $C$, or as a return type of a method in $C$, $L_1$ and $L_2$ should return the same `Class` object.

A malicious class loader could violate these properties. However, it could not undermine the security of the type system, because the Java Virtual Machine guards against this.

For further discussion of these issues, see *The Java Virtual Machine Specification, Java SE 11 Edition* and the paper *Dynamic Class Loading in the Java Virtual Machine*, by Sheng Liang and Gilad Bracha, in *Proceedings of OOPSLA ’98*, published as *ACM SIGPLAN Notices*, Volume 33, Number 10, October 1998, pages 36-44. A basic principle of the design of the Java programming language is that the run-time type system cannot be subverted by code written in the Java programming language, not even by implementations of such otherwise sensitive system classes as `ClassLoader` and `SecurityManager`.

### 12.2.1 The Loading Process

The loading process is implemented by the class `ClassLoader` and its subclasses.

Different subclasses of `ClassLoader` may implement different loading policies. In particular, a class loader may cache binary representations of classes and interfaces, prefetch them based on expected usage, or load a group of related classes together. These activities may not be completely transparent to a running application if, for example, a newly compiled version of a class is not found because an older version is cached by a class loader. It is the responsibility of a class loader, however, to reflect loading errors only at points in the program where they could have arisen without prefetching or group loading.

If an error occurs during class loading, then an instance of one of the following subclasses of class `LinkageError` will be thrown at any point in the program that (directly or indirectly) uses the type:

- `ClassCircularityError`: A class or interface could not be loaded because it would be its own superclass or superinterface (§8.1.4, §9.1.3, §13.4.4).
• **ClassFormatError**: The binary data that purports to specify a requested compiled class or interface is malformed.

• **NoClassDefFoundError**: No definition for a requested class or interface could be found by the relevant class loader.

Because loading involves the allocation of new data structures, it may fail with an **OutOfMemoryError**.

### 12.3 Linking of Classes and Interfaces

*Linking* is the process of taking a binary form of a class or interface type and combining it into the run-time state of the Java Virtual Machine, so that it can be executed. A class or interface type is always loaded before it is linked.

Three different activities are involved in linking: verification, preparation, and resolution of symbolic references.

The precise semantics of linking are given in Chapter 5 of *The Java Virtual Machine Specification, Java SE 11 Edition*. Here we present an overview of the process from the viewpoint of the Java programming language.

This specification allows an implementation flexibility as to when linking activities (and, because of recursion, loading) take place, provided that the semantics of the Java programming language are respected, that a class or interface is completely verified and prepared before it is initialized, and that errors detected during linkage are thrown at a point in the program where some action is taken by the program that might require linkage to the class or interface involved in the error.

For example, an implementation may choose to resolve each symbolic reference in a class or interface individually, only when it is used (lazy or late resolution), or to resolve them all at once while the class is being verified (static resolution). This means that the resolution process may continue, in some implementations, after a class or interface has been initialized.

Because linking involves the allocation of new data structures, it may fail with an **OutOfMemoryError**.

### 12.3.1 Verification of the Binary Representation

*Verification* ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other
instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java Virtual Machine language.

If an error occurs during verification, then an instance of the following subclass of class LinkageError will be thrown at the point in the program that caused the class to be verified:

- **VerifyError**: The binary definition for a class or interface failed to pass a set of required checks to verify that it obeys the semantics of the Java Virtual Machine language and that it cannot violate the integrity of the Java Virtual Machine. (See §13.4.2, §13.4.4, §13.4.9, and §13.4.17 for some examples.)

### 12.3.2 Preparation of a Class or Interface Type

*Preparation* involves creating the static fields (class variables and constants) for a class or interface and initializing such fields to the default values (§4.12.5). This does not require the execution of any source code; explicit initializers for static fields are executed as part of initialization (§12.4), not preparation.

Implementations of the Java Virtual Machine may precompute additional data structures at preparation time in order to make later operations on a class or interface more efficient. One particularly useful data structure is a "method table" or other data structure that allows any method to be invoked on instances of a class without requiring a search of superclasses at invocation time.

### 12.3.3 Resolution of Symbolic References

The binary representation of a class or interface references other classes and interfaces and their fields, methods, and constructors symbolically, using the binary names (§13.1) of the other classes and interfaces (§13.1). For fields and methods, these symbolic references include the name of the class or interface type of which the field or method is a member, as well as the name of the field or method itself, together with appropriate type information.

Before a symbolic reference can be used it must undergo resolution, wherein a symbolic reference is checked to be correct and, typically, replaced with a direct reference that can be more efficiently processed if the reference is used repeatedly.

If an error occurs during resolution, then an error will be thrown. Most typically, this will be an instance of one of the following subclasses of the class IncompatibleClassChangeError, but it may also be an instance of some other subclass of IncompatibleClassChangeError or even an instance of the class
IncompatibleClassChangeError itself. This error may be thrown at any point in
the program that uses a symbolic reference to the type, directly or indirectly:

- **IllegalAccessError**: A symbolic reference has been encountered that specifies
  a use or assignment of a field, or invocation of a method, or creation of an
  instance of a class, to which the code containing the reference does not have
  access because the field or method was declared with private, protected, or
  package access (not public), or because the class was not declared public in a
  package that is exported or opened to the code containing the reference.

  This can occur, for example, if a field that is originally declared public is changed to
  be private after another class that refers to the field has been compiled (§13.4.7); or
  if the package in which a public class is declared ceases to be exported by its module
  after another module that refers to the class has been compiled (§13.3).

- **InstantiationException**: A symbolic reference has been encountered that is used
  in class instance creation expression, but an instance cannot be created because
  the reference turns out to refer to an interface or to an abstract class.

  This can occur, for example, if a class that is originally not abstract is changed to
  be abstract after another class that refers to the class in question has been compiled
  (§13.4.1).

- **NoSuchFieldError**: A symbolic reference has been encountered that refers to a
  specific field of a specific class or interface, but the class or interface does not
  contain a field of that name.

  This can occur, for example, if a field declaration was deleted from a class after another
  class that refers to the field was compiled (§13.4.8).

- **NoSuchMethodError**: A symbolic reference has been encountered that refers to a
  specific method of a specific class or interface, but the class or interface does not
  contain a method of that signature.

  This can occur, for example, if a method declaration was deleted from a class after another
  class that refers to the method was compiled (§13.4.12).

Additionally, an UnsatisfiedLinkError, a subclass of LinkageError, may be
thrown if a class declares a native method for which no implementation can be
found. The error will occur if the method is used, or earlier, depending on what
kind of resolution strategy is being used by an implementation of the Java Virtual
Machine (§12.3).
12.4 Initialization of Classes and Interfaces

Initialization of a class consists of executing its static initializers and the initializers for static fields (class variables) declared in the class.

Initialization of an interface consists of executing the initializers for fields (constants) declared in the interface.

12.4.1 When Initialization Occurs

A class or interface type $\tau$ will be initialized immediately before the first occurrence of any one of the following:

- $\tau$ is a class and an instance of $\tau$ is created.
- A static method declared by $\tau$ is invoked.
- A static field declared by $\tau$ is assigned.
- A static field declared by $\tau$ is used and the field is not a constant variable (§4.12.4).

When a class is initialized, its superclasses are initialized (if they have not been previously initialized), as well as any superinterfaces (§8.1.5) that declare any default methods (§9.4.3) (if they have not been previously initialized). Initialization of an interface does not, of itself, cause initialization of any of its superinterfaces.

A reference to a static field (§8.3.1.1) causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface.

Invocation of certain reflective methods in class Class and in package java.lang.reflect also causes class or interface initialization.

A class or interface will not be initialized under any other circumstance.

Note that a compiler may generate synthetic default methods in an interface, that is, default methods that are neither explicitly nor implicitly declared (§13.1). Such methods will trigger the interface’s initialization despite the source code giving no indication that the interface should be initialized.

The intent is that a class or interface type has a set of initializers that put it in a consistent state, and that this state is the first state that is observed by other classes. The static initializers and class variable initializers are executed in textual order, and may not refer to class variables declared in the class whose declarations appear textually after the use, even though these class variables are in scope (§8.3.3).
This restriction is designed to detect, at compile time, most circular or otherwise malformed initializations.

The fact that initialization code is unrestricted allows examples to be constructed where the value of a class variable can be observed when it still has its initial default value, before its initializing expression is evaluated, but such examples are rare in practice. (Such examples can be also constructed for instance variable initialization (§12.5).) The full power of the Java programming language is available in these initializers; programmers must exercise some care. This power places an extra burden on code generators, but this burden would arise in any case because the Java programming language is concurrent (§12.4.2).

Example 12.4.1-1. Superclasses Are Initialized Before Subclasses

```java
class Super {
    static { System.out.print("Super "); }
}
class One {
    static { System.out.print("One "); }
}
class Two extends Super {
    static { System.out.print("Two "); }
}
class Test {
    public static void main(String[] args) {
        One o = null;
        Two t = new Two();
        System.out.println((Object)o == (Object)t);
    }
}
```

This program produces the output:

```
Super Two false
```

The class One is never initialized, because it not used actively and therefore is never linked to. The class Two is initialized only after its superclass Super has been initialized.

Example 12.4.1-2. Only The Class That Declares static Field Is Initialized

```java
class Super {
    static int taxi = 1729;
}
class Sub extends Super {
    static { System.out.print("Sub "); }
}
class Test {
    public static void main(String[] args) {
        System.out.println(Sub.taxi);
    }
}
```
This program prints only:

1729

because the class Sub is never initialized; the reference to Sub.taxi is a reference to a field actually declared in class Super and does not trigger initialization of the class Sub.

Example 12.4.1-3. Interface Initialization Does Not Initialize Superinterfaces

```java
interface I {
    int i = 1, ii = Test.out("ii", 2);
}
interface J extends I {
    int j = Test.out("j", 3), jj = Test.out("jj", 4);
}
interface K extends J {
    int k = Test.out("k", 5);
}
class Test {
    public static void main(String[] args) {
        System.out.println(J.i);
        System.out.println(K.j);
    }
    static int out(String s, int i) {
        System.out.println(s + "="+ i);
        return i;
    }
}
```

This program produces the output:

1
j=3
jj=4
3

The reference to J.i is to a field that is a constant variable (§4.12.4); therefore, it does not cause I to be initialized (§13.4.9).

The reference to K.j is a reference to a field actually declared in interface J that is not a constant variable; this causes initialization of the fields of interface J, but not those of its superinterface I, nor those of interface K.

Despite the fact that the name K is used to refer to field J of interface J, interface K is not initialized.
12.4.2 Detailed Initialization Procedure

Because the Java programming language is multithreaded, initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface; for example, a variable initializer in class $A$ might invoke a method of an unrelated class $B$, which might in turn invoke a method of class $A$. The implementation of the Java Virtual Machine is responsible for taking care of synchronization and recursive initialization by using the following procedure.

The procedure assumes that the `Class` object has already been verified and prepared, and that the `Class` object contains state that indicates one of four situations:

- This `Class` object is verified and prepared but not initialized.
- This `Class` object is being initialized by some particular thread $T$.
- This `Class` object is fully initialized and ready for use.
- This `Class` object is in an erroneous state, perhaps because initialization was attempted and failed.

For each class or interface $C$, there is a unique initialization lock $LC$. The mapping from $C$ to $LC$ is left to the discretion of the Java Virtual Machine implementation. The procedure for initializing $C$ is then as follows:

1. Synchronize on the initialization lock, $LC$, for $C$. This involves waiting until the current thread can acquire $LC$.
2. If the `Class` object for $C$ indicates that initialization is in progress for $C$ by some other thread, then release $LC$ and block the current thread until informed that the in-progress initialization has completed, at which time repeat this step.
3. If the `Class` object for $C$ indicates that initialization is in progress for $C$ by the current thread, then this must be a recursive request for initialization. Release $LC$ and complete normally.
4. If the `Class` object for $C$ indicates that $C$ has already been initialized, then no further action is required. Release $LC$ and complete normally.
5. If the `Class` object for $C$ is in an erroneous state, then initialization is not possible. Release $LC$ and throw a `NoClassDefFoundError`. 
6. Otherwise, record the fact that initialization of the Class object for C is in progress by the current thread, and release LC.

Then, initialize the static fields of C which are constant variables (§4.12.4, §8.3.2, §9.3.1).

7. Next, if C is a class rather than an interface, then let SC be its superclass and let SI_1, ..., SI_n be all superinterfaces of C that declare at least one default method. The order of superinterfaces is given by a recursive enumeration over the superinterface hierarchy of each interface directly implemented by C (in the left-to-right order of C's implements clause). For each interface I directly implemented by C, the enumeration recurs on I's superinterfaces (in the left-to-right order of I's extends clause) before returning I.

For each S in the list [ SC, SI_1, ..., SI_n ], if S has not yet been initialized, then recursively perform this entire procedure for S. If necessary, verify and prepare S first.

If the initialization of S completes abruptly because of a thrown exception, then acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete abruptly, throwing the same exception that resulted from initializing S.

8. Next, determine whether assertions are enabled (§14.10) for C by querying its defining class loader.

9. Next, execute either the class variable initializers and static initializers of the class, or the field initializers of the interface, in textual order, as though they were a single block.

10. If the execution of the initializers completes normally, then acquire LC, label the Class object for C as fully initialized, notify all waiting threads, release LC, and complete this procedure normally.

11. Otherwise, the initializers must have completed abruptly by throwing some exception E. If the class of E is not Error or one of its subclasses, then create a new instance of the class ExceptionInInitializerError, with E as the argument, and use this object in place of E in the following step. If a new instance of ExceptionInInitializerError cannot be created because an OutOfMemoryError occurs, then instead use an OutOfMemoryError object in place of E in the following step.

12. Acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.
An implementation may optimize this procedure by eliding the lock acquisition in step 1 (and release in step 4/5) when it can determine that the initialization of the class has already completed, provided that, in terms of the memory model, all happens-before orderings that would exist if the lock were acquired, still exist when the optimization is performed.

Code generators need to preserve the points of possible initialization of a class or interface, inserting an invocation of the initialization procedure just described. If this initialization procedure completes normally and the `Class` object is fully initialized and ready for use, then the invocation of the initialization procedure is no longer necessary and it may be eliminated from the code - for example, by patching it out or otherwise regenerating the code.

Compile-time analysis may, in some cases, be able to eliminate many of the checks that a type has been initialized from the generated code, if an initialization order for a group of related types can be determined. Such analysis must, however, fully account for concurrency and for the fact that initialization code is unrestricted.

### 12.5 Creation of New Class Instances

A new class instance is explicitly created when evaluation of a class instance creation expression (§15.9) causes a class to be instantiated.

A new class instance may be implicitly created in the following situations:

- Loading of a class or interface that contains a string literal (§3.10.5) may create a new `String` object to represent the literal. (This will not occur if a string denoting the same sequence of Unicode code points has previously been interned.)

- Execution of an operation that causes boxing conversion (§5.1.7). Boxing conversion may create a new object of a wrapper class (`Boolean, Byte, Short, Character, Integer, Long, Float, Double`) associated with one of the primitive types.

- Execution of a string concatenation operator `+` (§15.18.1) that is not part of a constant expression (§15.28) always creates a new `String` object to represent the result. String concatenation operators may also create temporary wrapper objects for a value of a primitive type.

- Evaluation of a method reference expression (§15.13.3) or a lambda expression (§15.27.4) may require that a new instance of a class that implements a functional interface type be created.

Each of these situations identifies a particular constructor (§8.8) to be called with specified arguments (possibly none) as part of the class instance creation process.
Whenever a new class instance is created, memory space is allocated for it with room for all the instance variables declared in the class type and all the instance variables declared in each superclass of the class type, including all the instance variables that may be hidden (§8.3).

If there is not sufficient space available to allocate memory for the object, then creation of the class instance completes abruptly with an `OutOfMemoryError`. Otherwise, all the instance variables in the new object, including those declared in superclasses, are initialized to their default values (§4.12.5).

Just before a reference to the newly created object is returned as the result, the indicated constructor is processed to initialize the new object using the following procedure:

1. Assign the arguments for the constructor to newly created parameter variables for this constructor invocation.

2. If this constructor begins with an explicit constructor invocation (§8.8.7.1) of another constructor in the same class (using `this`), then evaluate the arguments and process that constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason; otherwise, continue with step 5.

3. This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using `this`). If this constructor is for a class other than `Object`, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using `super`). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue with step 4.

4. Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception. Otherwise, continue with step 5.

5. Execute the rest of the body of this constructor. If that execution completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.

Unlike C++, the Java programming language does not specify altered rules for method dispatch during the creation of a new class instance. If methods are
invoked that are overridden in subclasses in the object being initialized, then these overriding methods are used, even before the new object is completely initialized.

**Example 12.5-1. Evaluation of Instance Creation**

```java
class Point {
    int x, y;
    Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        System.out.println(cp.color);
    }
}
```

Here, a new instance of `ColoredPoint` is created. First, space is allocated for the new `ColoredPoint`, to hold the fields `x`, `y`, and `color`. All these fields are then initialized to their default values (in this case, 0 for each field). Next, the `ColoredPoint` constructor with no arguments is first invoked. Since `ColoredPoint` declares no constructors, a default constructor of the following form is implicitly declared:

```
ColoredPoint() { super(); }
```

This constructor then invokes the `Point` constructor with no arguments. The `Point` constructor does not begin with an invocation of a constructor, so the Java compiler provides an implicit invocation of its superclass constructor of no arguments, as though it had been written:

```
Point() { super(); x = 1; y = 1; }
```

Therefore, the constructor for `Object` which takes no arguments is invoked.

The class `Object` has no superclass, so the recursion terminates here. Next, any instance initializers and instance variable initializers of `Object` are invoked. Next, the body of the constructor of `Object` that takes no arguments is executed. No such constructor is declared in `Object`, so the Java compiler supplies a default one, which in this special case is:

```
Object() { }
```

This constructor executes without effect and returns.

Next, all initializers for the instance variables of class `Point` are executed. As it happens, the declarations of `x` and `y` do not provide any initialization expressions, so no action is required for this step of the example. Then the body of the `Point` constructor is executed, setting `x` to 1 and `y` to 1.
Next, the initializers for the instance variables of class \texttt{ColoredPoint} are executed. This step assigns the value 0xFF00FF to \texttt{color}. Finally, the rest of the body of the \texttt{ColoredPoint} constructor is executed (the part after the invocation of \texttt{super}); there happen to be no statements in the rest of the body, so no further action is required and initialization is complete.

**Example 12.5-2. Dynamic Dispatch During Instance Creation**

```java
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}

class Test extends Super {
    int three = (int)Math.PI; // That is, 3
    void printThree() { System.out.println(three); }

    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

This program produces the output:

```
0
3
```

This shows that the invocation of \texttt{printThree} in the constructor for class \texttt{Super} does not invoke the definition of \texttt{printThree} in class \texttt{Super}, but rather invokes the overriding definition of \texttt{printThree} in class \texttt{Test}. This method therefore runs before the field initializers of \texttt{Test} have been executed, which is why the first value output is 0, the default value to which the field \texttt{three} of \texttt{Test} is initialized. The later invocation of \texttt{printThree} in method \texttt{main} invokes the same definition of \texttt{printThree}, but by that point the initializer for instance variable \texttt{three} has been executed, and so the value 3 is printed.

### 12.6 Finalization of Class Instances

The class \texttt{Object} has a protected method called \texttt{finalize}; this method can be overridden by other classes. The particular definition of \texttt{finalize} that can be invoked for an object is called the \texttt{finalizer} of that object. Before the storage for an object is reclaimed by the garbage collector, the Java Virtual Machine will invoke the finalizer of that object.

Finalizers provide a chance to free up resources that cannot be freed automatically by an automatic storage manager. In such situations, simply reclaiming the memory used by an object would not guarantee that the resources it held would be reclaimed.
The Java programming language does not specify how soon a finalizer will be invoked, except to say that it will happen before the storage for the object is reused.

The Java programming language does not specify which thread will invoke the finalizer for any given object.

It is important to note that many finalizer threads may be active (this is sometimes needed on large shared memory multiprocessors), and that if a large connected data structure becomes garbage, all of the finalize methods for every object in that data structure could be invoked at the same time, each finalizer invocation running in a different thread.

The Java programming language imposes no ordering on finalize method calls. Finalizers may be called in any order, or even concurrently.

As an example, if a circularly linked group of unfinalized objects becomes unreachable (or finalizer-reachable), then all the objects may become finalizable together. Eventually, the finalizers for these objects may be invoked, in any order, or even concurrently using multiple threads. If the automatic storage manager later finds that the objects are unreachable, then their storage can be reclaimed.

It is straightforward to implement a class that will cause a set of finalizer-like methods to be invoked in a specified order for a set of objects when all the objects become unreachable. Defining such a class is left as an exercise for the reader.

It is guaranteed that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is invoked.

If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

The completion of an object's constructor happens-before (§17.4.5) the execution of its finalize method (in the formal sense of happens-before).

The finalize method declared in class Object takes no action. The fact that class Object declares a finalize method means that the finalize method for any class can always invoke the finalize method for its superclass. This should always be done, unless it is the programmer's intent to nullify the actions of the finalizer in the superclass. (Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

For efficiency, an implementation may keep track of classes that do not override the finalize method of class Object, or override it in a trivial way.

For example:

```java
protected void finalize() throws Throwable {
    super.finalize();
}
```
We encourage implementations to treat such objects as having a finalizer that is not overridden, and to finalize them more efficiently, as described in §12.6.1.

A finalizer may be invoked explicitly, just like any other method.

The package java.lang.ref describes weak references, which interact with garbage collection and finalization. As with any API that has special interactions with the Java programming language, implementors must be cognizant of any requirements imposed by the java.lang.ref API. This specification does not discuss weak references in any way. Readers are referred to the API documentation for details.

### 12.6.1 Implementing Finalization

Every object can be characterized by two attributes: it may be reachable, finalizer-reachable, or unreachable, and it may also be unfinalized, finalizable, or finalized.

A reachable object is any object that can be accessed in any potential continuing computation from any live thread.

A finalizer-reachable object can be reached from some finalizable object through some chain of references, but not from any live thread.

An unreachable object cannot be reached by either means.

An unfinalized object has never had its finalizer automatically invoked.

A finalized object has had its finalizer automatically invoked.

A finalizable object has never had its finalizer automatically invoked, but the Java Virtual Machine may eventually automatically invoke its finalizer.

An object o is not finalizable until its constructor has invoked the constructor for object on o and that invocation has completed successfully (that is, without throwing an exception). Every pre-finalization write to a field of an object must be visible to the finalization of that object. Furthermore, none of the pre-finalization reads of fields of that object may see writes that occur after finalization of that object is initiated.

Optimizing transformations of a program can be designed that reduce the number of objects that are reachable to be less than those which would naively be considered reachable. For example, a Java compiler or code generator may choose to set a variable or parameter that will no longer be used to null to cause the storage for such an object to be potentially reclaimable sooner.
Another example of this occurs if the values in an object's fields are stored in registers. The program may then access the registers instead of the object, and never access the object again. This would imply that the object is garbage. Note that this sort of optimization is only allowed if references are on the stack, not stored in the heap.

For example, consider the Finalizer Guardian pattern:

```java
class Foo {
    private final Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
            /* finalize outer Foo object */
        }
    }
}
```

The finalizer guardian forces `super.finalize` to be called if a subclass overrides `finalize` and does not explicitly call `super.finalize`.

If these optimizations are allowed for references that are stored on the heap, then a Java compiler can detect that the `finalizerGuardian` field is never read, null it out, collect the object immediately, and call the finalizer early. This runs counter to the intent: the programmer probably wanted to call the Foo finalizer when the Foo instance became unreachable. This sort of transformation is therefore not legal: the inner class object should be reachable for as long as the outer class object is reachable.

Transformations of this sort may result in invocations of the `finalize` method occurring earlier than might be otherwise expected. In order to allow the user to prevent this, we enforce the notion that synchronization may keep the object alive. If an object's finalizer can result in synchronization on that object, then that object must be alive and considered reachable whenever a lock is held on it.

Note that this does not prevent synchronization elimination: synchronization only keeps an object alive if a finalizer might synchronize on it. Since the finalizer occurs in another thread, in many cases the synchronization could not be removed anyway.

### 12.6.2 Interaction with the Memory Model

It must be possible for the memory model (§17.4) to decide when it can commit actions that take place in a finalizer. This section describes the interaction of finalization with the memory model.

Each execution has a number of reachability decision points, labeled $d_i$. Each action either comes-before $d_i$ or comes-after $d_i$. Other than as explicitly mentioned, the comes-before ordering described in this section is unrelated to all other orderings in the memory model.

If $r$ is a read that sees a write $w$ and $r$ comes-before $d_i$, then $w$ must come-before $d_i$. 
If \( x \) and \( y \) are synchronization actions on the same variable or monitor such that \( so(x, y) \) (§17.4.4) and \( y \) comes-before \( di \), then \( x \) must come-before \( di \).

At each reachability decision point, some set of objects are marked as unreachable, and some subset of those objects are marked as finalizable. These reachability decision points are also the points at which references are checked, enqueued, and cleared according to the rules provided in the API documentation for the package `java.lang.ref`.

The only objects that are considered definitely reachable at a point \( di \) are those that can be shown to be reachable by the application of these rules:

- An object \( B \) is definitely reachable at \( di \) from static fields if there exists a write \( w1 \) to a static field \( v \) of a class \( C \) such that the value written by \( w1 \) is a reference to \( B \), the class \( C \) is loaded by a reachable classloader, and there does not exist a write \( w2 \) to \( v \) such that \( hb(w2, w1) \) is not true and both \( w1 \) and \( w2 \) come-before \( di \).

- An object \( B \) is definitely reachable from \( A \) at \( di \) if there is a write \( w1 \) to an element \( v \) of \( A \) such that the value written by \( w1 \) is a reference to \( B \) and there does not exist a write \( w2 \) to \( v \) such that \( hb(w2, w1) \) is not true and both \( w1 \) and \( w2 \) come-before \( di \).

- If an object \( C \) is definitely reachable from an object \( B \), and object \( B \) is definitely reachable from an object \( A \), then \( C \) is definitely reachable from \( A \).

If an object \( x \) is marked as unreachable at \( di \), then:

- \( x \) must not be definitely reachable at \( di \) from static fields; and

- All active uses of \( x \) in thread \( t \) that come-after \( di \) must occur in the finalizer invocation for \( x \) or as a result of thread \( t \) performing a read that comes-after \( di \) of a reference to \( x \); and

- All reads that come-after \( di \) that see a reference to \( x \) must see writes to elements of objects that were unreachable at \( di \), or see writes that came-after \( di \).

An action \( a \) is an active use of \( x \) if and only if at least one of the following is true:

- \( a \) reads or writes an element of \( x \)

- \( a \) locks or unlocks \( x \) and there is a lock action on \( x \) that happens-after the invocation of the finalizer for \( x \)

- \( a \) writes a reference to \( x \)

- \( a \) is an active use of an object \( y \), and \( x \) is definitely reachable from \( y \)

If an object \( x \) is marked as finalizable at \( di \), then:
12.7 Unloading of Classes and Interfaces

An implementation of the Java programming language may **unload** classes.

A class or interface may be unloaded if and only if its defining class loader may be reclaimed by the garbage collector as discussed in §12.6.

Classes and interfaces loaded by the bootstrap loader may not be unloaded.

Class unloading is an optimization that helps reduce memory use. Obviously, the semantics of a program should not depend on whether and how a system chooses to implement an optimization such as class unloading. To do otherwise would compromise the portability of programs. Consequently, whether a class or interface has been unloaded or not should be transparent to a program.

However, if a class or interface $C$ was unloaded while its defining loader was potentially reachable, then $C$ might be reloaded. One could never ensure that this would not happen. Even if the class was not referenced by any other currently loaded class, it might be referenced by some class or interface, $D$, that had not yet been loaded. When $D$ is loaded by $C$'s defining loader, its execution might cause reloading of $C$.

Reloading may not be transparent if, for example, the class has static variables (whose state would be lost), static initializers (which may have side effects), or native methods (which may retain static state). Furthermore, the hash value of the Class object is dependent on its identity. Therefore it is, in general, impossible to reload a class or interface in a completely transparent manner.

Since we can never guarantee that unloading a class or interface whose loader is potentially reachable will not cause reloading, and reloading is never transparent, but unloading must be transparent, it follows that one must not unload a class or interface while its loader is potentially reachable. A similar line of reasoning can be used to deduce that classes and interfaces loaded by the bootstrap loader can never be unloaded.

One must also argue why it is safe to unload a class $C$ if its defining class loader can be reclaimed. If the defining loader can be reclaimed, then there can never be any live references to it (this includes references that are not live, but might be resurrected by finalizers). This, in turn, can only be true if there are can never be any live references to any of the classes defined by that loader, including $C$, either from their instances or from code.

Class unloading is an optimization that is only significant for applications that load large numbers of classes and that stop using most of those classes after some time. A prime example of such an application is a web browser, but there are others. A characteristic of
such applications is that they manage classes through explicit use of class loaders. As a result, the policy outlined above works well for them.

Strictly speaking, it is not essential that the issue of class unloading be discussed by this specification, as class unloading is merely an optimization. However, the issue is very subtle, and so it is mentioned here by way of clarification.

12.8 Program Exit

A program terminates all its activity and exits when one of two things happens:

- All the threads that are not daemon threads terminate.
- Some thread invokes the `exit` method of class `Runtime` or class `System`, and the `exit` operation is not forbidden by the security manager.
DEVELOPMENT tools for the Java programming language should support automatic recompilation as necessary whenever source code is available. Particular implementations may also store the source and binary of types in a versioning database and implement a ClassLoader that uses integrity mechanisms of the database to prevent linkage errors by providing binary-compatible versions of types to clients.

Developers of packages and classes that are to be widely distributed face a different set of problems. In the Internet, which is our favorite example of a widely distributed system, it is often impractical or impossible to automatically recompile the pre-existing binaries that directly or indirectly depend on a type that is to be changed. Instead, this specification defines a set of changes that developers are permitted to make to a package or to a class or interface type while preserving (not breaking) compatibility with pre-existing binaries.

Within the framework of Release-to-Release Binary Compatibility in SOM (Forman, Conner, Danforth, and Raper, Proceedings of OOPSLA ’95), Java programming language binaries are binary compatible under all relevant transformations that the authors identify (with some caveats with respect to the addition of instance variables). Using their scheme, here is a list of some important binary compatible changes that the Java programming language supports:

- Reimplementing existing methods, constructors, and initializers to improve performance.
- Changing methods or constructors to return values on inputs for which they previously either threw exceptions that normally should not occur or failed by going into an infinite loop or causing a deadlock.
- Adding new fields, methods, or constructors to an existing class or interface.
- Deleting private fields, methods, or constructors of a class.
13.1 The Form of a Binary

Programs must be compiled either into the class file format specified by The Java Virtual Machine Specification, Java SE 11 Edition, or into a representation that can be mapped into that format by a class loader written in the Java programming language.

A class file corresponding to a class or interface declaration must have certain properties. A number of these properties are specifically chosen to support source code transformations that preserve binary compatibility. The required properties are:

- When an entire package is updated, deleting package access fields, methods, or constructors of classes and interfaces in the package.
- Reordering the fields, methods, or constructors in an existing type declaration.
- Moving a method upward in the class hierarchy.
- Reordering the list of direct superinterfaces of a class or interface.
- Inserting new class or interface types in the type hierarchy.

This chapter specifies minimum standards for binary compatibility guaranteed by all implementations. The Java programming language guarantees compatibility when binaries of classes and interfaces are mixed that are not known to be from compatible sources, but whose sources have been modified in the compatible ways described here. Note that we are discussing compatibility between releases of an application. A discussion of compatibility among releases of the Java SE Platform is beyond the scope of this chapter.

We encourage development systems to provide facilities that alert developers to the impact of changes on pre-existing binaries that cannot be recompiled.

This chapter first specifies some properties that any binary format for the Java programming language must have (§13.1). It next defines binary compatibility, explaining what it is and what it is not (§13.2). It finally enumerates a large set of possible changes to packages (§13.3), classes (§13.4), and interfaces (§13.5), specifying which of these changes are guaranteed to preserve binary compatibility and which are not.

Occasionally, references of the form: (JVMS §x.y) are used to indicate concepts from The Java Virtual Machine Specification, Java SE 11 Edition.
1. The class or interface must be named by its *binary name*, which must meet the following constraints:

   • The binary name of a top level type (§7.6) is its canonical name (§6.7).

   • The binary name of a member type (§8.5, §9.5) consists of the binary name of its immediately enclosing type, followed by $\$, followed by the simple name of the member.

   • The binary name of a local class (§14.3) consists of the binary name of its immediately enclosing type, followed by $\$, followed by a non-empty sequence of digits, followed by the simple name of the local class.

   • The binary name of an anonymous class (§15.9.5) consists of the binary name of its immediately enclosing type, followed by $\$, followed by a non-empty sequence of digits.

   • The binary name of a type variable declared by a generic class or interface (§8.1.2, §9.1.2) is the binary name of its immediately enclosing type, followed by $\$, followed by the simple name of the type variable.

   • The binary name of a type variable declared by a generic method (§8.4.4) is the binary name of the type declaring the method, followed by $\$, followed by the descriptor of the method (JVMS §4.3.3), followed by $\$, followed by the simple name of the type variable.

   • The binary name of a type variable declared by a generic constructor (§8.8.4) is the binary name of the type declaring the constructor, followed by $\$, followed by the descriptor of the constructor (JVMS §4.3.3), followed by $\$, followed by the simple name of the type variable.

2. A reference to another class or interface type must be symbolic, using the binary name of the type.

3. A reference to a field that is a constant variable (§4.12.4) must be resolved at compile time to the value $v$ denoted by the constant variable's initializer.

   If such a field is *static*, then no reference to the field should be present in the code in a binary file, including the class or interface which declared the field. Such a field must always appear to have been initialized (§12.4.2); the default initial value for the field (if different than $v$) must never be observed.

   If such a field is non-static, then no reference to the field should be present in the code in a binary file, except in the class containing the field. (It will be a class rather than an interface, since an interface has only static fields.) The class should have code to set the field's value to $v$ during instance creation (§12.5).
4. Given a legal expression denoting a field access in a class $c$, referencing a field named $f$ that is not a constant variable and is declared in a (possibly distinct) class or interface $d$, we define the *qualifying type of the field reference* as follows:

- If the expression is referenced by a simple name, then if $f$ is a member of the current class or interface, $c$, then let $T$ be $c$. Otherwise, let $T$ be the innermost lexically enclosing type declaration of which $f$ is a member. In either case, $T$ is the qualifying type of the reference.
- If the reference is of the form $TypeName.f$, where $TypeName$ denotes a class or interface, then the class or interface denoted by $TypeName$ is the qualifying type of the reference.
- If the expression is of the form $ExpressionName.f$ or $Primary.f$, then:
  - If the compile-time type of $ExpressionName$ or $Primary$ is an intersection type $V_1 \& \ldots \& V_n$ (§4.9), then the qualifying type of the reference is $V_1$.
  - Otherwise, the compile-time type of $ExpressionName$ or $Primary$ is the qualifying type of the reference.
- If the expression is of the form $super.f$, then the superclass of $c$ is the qualifying type of the reference.
- If the expression is of the form $TypeName.super.f$, then the superclass of the class denoted by $TypeName$ is the qualifying type of the reference.

The reference to $f$ must be compiled into a symbolic reference to the erasure (§4.6) of the qualifying type of the reference, plus the simple name of the field, $f$. The reference must also include a symbolic reference to the erasure of the declared type of the field so that the verifier can check that the type is as expected.

5. Given a method invocation expression or a method reference expression in a class or interface $c$, referencing a method named $m$ declared (or implicitly declared (§9.2)) in a (possibly distinct) class or interface $d$, we define the *qualifying type of the method invocation* as follows:

- If $D$ is $Object$ then the qualifying type of the expression is $Object$.
- Otherwise:
  - If the method is referenced by a simple name, then if $m$ is a member of the current class or interface $c$, let $T$ be $c$; otherwise, let $T$ be the innermost lexically enclosing type declaration of which $m$ is a member. In either case, $T$ is the qualifying type of the method invocation.
– If the expression is of the form `TypeName.m` or `ReferenceType::m`, then the type denoted by `TypeName` or `ReferenceType` is the qualifying type of the method invocation.

– If the expression is of the form `ExpressionName.m` or `Primary.m` or `ExpressionName::m` or `Primary::m`, then:

  › If the compile-time type of `ExpressionName` or `Primary` is an intersection type $V_1 \& \ldots \& V_n$ (§4.9), then the qualifying type of the method invocation is $V_1$.

  › Otherwise, the compile-time type of `ExpressionName` or `Primary` is the qualifying type of the method invocation.

– If the expression is of the form `super.m` or `super::m`, then the superclass of $C$ is the qualifying type of the method invocation.

– If the expression is of the form `TypeName.super.m` or `TypeName.super::m`, then if `TypeName` denotes a class $X$, the superclass of $X$ is the qualifying type of the method invocation; if `TypeName` denotes an interface $X$, $X$ is the qualifying type of the method invocation.

A reference to a method must be resolved at compile time to a symbolic reference to the erasure (§4.6) of the qualifying type of the invocation, plus the erasure of the signature (§8.4.2) of the method. The signature of a method must include all of the following as determined by §15.12.3:

• The simple name of the method

• The number of parameters to the method

• A symbolic reference to the type of each parameter

A reference to a method must also include either a symbolic reference to the erasure of the return type of the denoted method or an indication that the denoted method is declared `void` and does not return a value.

6. Given a class instance creation expression (§15.9) or an explicit constructor invocation statement (§8.8.7.1) or a method reference expression of the form `ClassType::new` (§15.13) in a class or interface $C$ referencing a constructor $m$ declared in a (possibly distinct) class or interface $D$, we define the qualifying type of the constructor invocation as follows:

• If the expression is of the form `new $D(...)` or `ExpressionName.new $D(...)` or `Primary.new $D(...)` or `$D::new`, then the qualifying type of the invocation is $D$. 
• If the expression is of the form `new D(...){...}` or `ExpressionName.new D(...){...}` or `Primary.new D(...){...}`, then the qualifying type of the expression is the compile-time type of the expression.

• If the expression is of the form `super(...)` or `ExpressionName.super(...)` or `Primary.super(...), then the qualifying type of the expression is the direct superclass of `C`.

• If the expression is of the form `this(...)`, then the qualifying type of the expression is `C`.

A reference to a constructor must be resolved at compile time to a symbolic reference to the erasure (§4.6) of the qualifying type of the invocation, plus the signature of the constructor (§8.8.2). The signature of a constructor must include both:

• The number of parameters of the constructor

• A symbolic reference to the type of each formal parameter

A binary representation for a class or interface must also contain all of the following:

1. If it is a class and is not `Object`, then a symbolic reference to the erasure of the direct superclass of this class.

2. A symbolic reference to the erasure of each direct superinterface, if any.

3. A specification of each field declared in the class or interface, given as the simple name of the field and a symbolic reference to the erasure of the type of the field.

4. If it is a class, then the erased signature of each constructor, as described above.

5. For each method declared in the class or interface (excluding, for an interface, its implicitly declared methods (§9.2)), its erased signature and return type, as described above.

6. The code needed to implement the class or interface:

   • For an interface, code for the field initializers and the implementation of each method with a block body (§9.4.3).

   • For a class, code for the field initializers, the instance and static initializers, the implementation of each method with a block body (§8.4.7), and the implementation of each constructor.
7. Every type must contain sufficient information to recover its canonical name (§6.7).

8. Every member type must have sufficient information to recover its source-level access modifier.

9. Every nested class and nested interface must have a symbolic reference to its immediately enclosing type (§8.1.3).

10. Every class must contain symbolic references to all of its member types (§8.5), and to all local and anonymous classes that appear in its methods, constructors, static initializers, instance initializers, and field initializers.

   Every interface must contain symbolic references to all of its member types (§9.5), and to all local and anonymous classes that appear in its default methods and field initializers.

11. A construct emitted by a Java compiler must be marked as synthetic if it does not correspond to a construct declared explicitly or implicitly in source code, unless the emitted construct is a class initialization method (JVMS §2.9).

12. A construct emitted by a Java compiler must be marked as mandated if it corresponds to a formal parameter declared implicitly in source code (§8.8.1, §8.8.9, §8.9.3, §15.9.5.1).

The following formal parameters are declared implicitly in source code:

- The first formal parameter of a constructor of a non-private inner member class (§8.8.1, §8.8.9).
- The first formal parameter of an anonymous constructor of an anonymous class whose superclass is inner or local (not in a static context) (§15.9.5.1).
- The formal parameter name of the valueOf method which is implicitly declared in an enum type (§8.9.3).

For reference, the following constructs are declared implicitly in source code, but are not marked as mandated because only formal parameters can be so marked in a class file (JVMS §4.7.24):

- Default constructors of classes and enum types (§8.8.9, §8.9.2)
- Anonymous constructors (§15.9.5.1)
- The values and valueOf methods of enum types (§8.9.3)
- Certain public fields of enum types (§8.9.3)
- Certain public methods of interfaces (§9.2)
- Container annotations (§9.7.5)
A class file corresponding to a module declaration must have the properties of a class file for a class whose binary name is module-info and which has no superclass, no superinterfaces, no fields, and no methods. In addition, the binary representation of the module must contain all of the following:

- A specification of the name of the module, given as a symbolic reference to the name indicated after module. Also, the specification must include whether the module is normal or open (§7.7).

- A specification of each dependence denoted by a requires directive, given as a symbolic reference to the name of the module indicated by the directive (§7.7.1). Also, the specification must include whether the dependence is transitive and whether the dependence is static.

- A specification of each package denoted by an exports or opens directive, given as a symbolic reference to the name of the package indicated by the directive (§7.7.2). Also, if the directive was qualified, the specification must give symbolic references to the names of the modules indicated by the directive’s to clause.

- A specification of each service denoted by a uses directive, given as a symbolic reference to the name of the type indicated by the directive (§7.7.3).

- A specification of the service providers denoted by a provides directive, given as symbolic references to the names of the types indicated by the directive’s with clause (§7.7.4). Also, the specification must give a symbolic reference to the name of the type indicated as the service by the directive.

The following sections discuss changes that may be made to class and interface type declarations without breaking compatibility with pre-existing binaries. Under the translation requirements given above, the Java Virtual Machine and its class file format support these changes. Any other valid binary format, such as a compressed or encrypted representation that is mapped back into class files by a class loader under the above requirements, will necessarily support these changes as well.

### 13.2 What Binary Compatibility Is and Is Not

A change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.

Binaries are compiled to rely on the accessible members and constructors of other classes and interfaces. To preserve binary compatibility, a class or interface should
treat its accessible members and constructors, their existence and behavior, as a *contract* with its users.

The Java programming language is designed to prevent additions to contracts and accidental name collisions from breaking binary compatibility. Specifically, addition of more methods overloading a particular method name does not break compatibility with pre-existing binaries. The method signature that the pre-existing binary will use for method lookup is chosen by the overload resolution algorithm at compile time (§15.12.2).

If the Java programming language had been designed so that the particular method to be executed was chosen at run time, then such an ambiguity might be detected at run time. Such a rule would imply that adding an additional overloaded method so as to make ambiguity possible at a call site could break compatibility with an unknown number of pre-existing binaries. See §13.4.23 for more discussion.

Binary compatibility is not the same as source compatibility. In particular, the example in §13.4.6 shows that a set of compatible binaries can be produced from sources that will not compile all together. This example is typical: a new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the pre-existing binary for that unchanged part of the source code retains the fully-qualified, previous meaning of the name. Producing a consistent set of source code requires providing a qualified name or field access expression corresponding to the previous meaning.

### 13.3 Evolution of Packages and Modules

A new top level class or interface type may be added to a package without breaking compatibility with pre-existing binaries, provided the new type does not reuse a name previously given to an unrelated type. If a new type reuses a name previously given to an unrelated type, then a conflict may result, since binaries for both types could not be loaded by the same class loader.

Changes in top level class and interface types that are not public and that are not a superclass or superinterface, respectively, of a public type, affect only types within the package in which they are declared. Such types may be deleted or otherwise changed, even if incompatibilities are otherwise described here, provided that the affected binaries of that package are updated together.

If a module that was declared to export or open a package is changed to not export or open the package, or to export or open the package to a different set of friends, then an `IllegalAccessError` is thrown if a pre-existing binary is linked that needs
but no longer has access to the public and protected types of the package. Such a change is not recommended for modules that have been widely distributed.

If a module was not declared to export or open a given package, then changing the module to export or open the package does not break compatibility with pre-existing binaries. However, changing the module to export the package may prevent the program from starting, since any module that reads the module may also read some other module that exports a package with the same name.

Adding a requires directive to a module declaration, or adding the transitive modifier to a requires directive, does not break compatibility with pre-existing binaries. However, it may prevent the program from starting, since the module may now read multiple modules that export packages with the same name.

Deleting a requires directive in a module declaration, or deleting the transitive modifier from a requires directive, may break compatibility with any pre-existing binary that relied on the directive or modifier for readability of a given module in the course of referencing types exported by that module. An IllegalAccessException may be thrown when such a reference from a pre-existing binary is linked.

Adding or deleting a uses or provides directive in a module declaration does not break compatibility with pre-existing binaries.

13.4 Evolution of Classes

This section describes the effects of changes to the declaration of a class and its members and constructors on pre-existing binaries.

13.4.1 abstract Classes

If a class that was not declared abstract is changed to be declared abstract, then pre-existing binaries that attempt to create new instances of that class will throw either an InstantiationError at link time, or (if a reflective method is used) an InstantiationException at run time; such a change is therefore not recommended for widely distributed classes.

Changing a class that is declared abstract to no longer be declared abstract does not break compatibility with pre-existing binaries.
13.4.2 final Classes

If a class that was not declared final is changed to be declared final, then a VerifyError is thrown if a binary of a pre-existing subclass of this class is loaded, because final classes can have no subclasses; such a change is not recommended for widely distributed classes.

Changing a class that is declared final to no longer be declared final does not break compatibility with pre-existing binaries.

13.4.3 public Classes

Changing a class that is not declared public to be declared public does not break compatibility with pre-existing binaries.

If a class that was declared public is changed to not be declared public, then an IllegalAccessError is thrown if a pre-existing binary is linked that needs but no longer has access to the class type; such a change is not recommended for widely distributed classes.

13.4.4 Superclasses and Superinterfaces

A ClassCircularityError is thrown at load time if a class would be a superclass of itself. Changes to the class hierarchy that could result in such a circularity when newly compiled binaries are loaded with pre-existing binaries are not recommended for widely distributed classes.

Changing the direct superclass or the set of direct superinterfaces of a class type will not break compatibility with pre-existing binaries, provided that the total set of superclasses or superinterfaces, respectively, of the class type loses no members.

If a change to the direct superclass or the set of direct superinterfaces results in any class or interface no longer being a superclass or superinterface, respectively, then linkage errors may result if pre-existing binaries are loaded with the binary of the modified class. Such changes are not recommended for widely distributed classes.

Example 13.4.4-1. Changing A Superclass

Suppose that the following test program:

class Hyper { char h = 'h'; }  
class Super extends Hyper { char s = 's'; }  
class Test extends Super {  
    public static void printH(Hyper h) {  
        System.out.println(h.h);  
    }  
}
Suppose that a new version of class `Super` is then compiled:

```java
class Super { char s = 's'; }
```

This version of class `Super` is not a subclass of `Hyper`. If we then run the existing binaries of `Hyper` and `Test` with the new version of `Super`, then a `VerifyError` is thrown at link time. The verifier objects because the result of `new Super()` cannot be passed as an argument in place of a formal parameter of type `Hyper`, because `Super` is not a subclass of `Hyper`.

It is instructive to consider what might happen without the verification step: the program might run and print:

```java
s
```

This demonstrates that without the verifier, the Java type system could be defeated by linking inconsistent binary files, even though each was produced by a correct Java compiler.

The lesson is that an implementation that lacks a verifier or fails to use it will not maintain type safety and is, therefore, not a valid implementation.

The requirement that alternatives in a multi-catch clause (§14.20) not be subclasses or superclasses of each other is only a source restriction. Assuming the following client code is legal:

```java
try {
    throwAorB();
} catch(ExceptionA | ExceptionB e) {
    ...
}
```

where `ExceptionA` and `ExceptionB` do not have a subclass/superclass relationship when the client is compiled, it is binary compatible with respect to the client for `ExceptionA` and `ExceptionB` to have such a relationship when the client is executed.

This is analogous to other situations where a class transformation that is binary compatible for a client might not be source compatible for the same client.
13.4.5 Class Type Parameters

Adding or removing a type parameter of a class does not, in itself, have any implications for binary compatibility.

If such a type parameter is used in the type of a field or method, that may have the normal implications of changing the aforementioned type.

Renaming a type parameter of a class has no effect with respect to pre-existing binaries.

Changing the first bound of a type parameter of a class may change the erasure (§4.6) of any member that uses that type parameter in its own type, and this may affect binary compatibility. The change of such a bound is analogous to the change of the first bound of a type parameter of a method or constructor (§13.4.13).

Changing any other bound has no effect on binary compatibility.

13.4.6 Class Body and Member Declarations

No incompatibility with pre-existing binaries is caused by adding an instance (respectively static) member that has the same name and accessibility (for fields), or same name and accessibility and signature and return type (for methods), as an instance (respectively static) member of a superclass or subclass. No error occurs even if the set of classes being linked would encounter a compile-time error.

Deleting a class member or constructor that is not declared private may cause a linkage error if the member or constructor is used by a pre-existing binary.

Example 13.4.6-1. Changing A Class Body

```java
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
class Test {
    public static void main(String[] args) {
        new Super().hello();
    }
}
```

This program produces the output:

```
hello from Super
```

Suppose that a new version of class Super is produced:
Then, recompiling Super and executing this new binary with the original binaries for Test and Hyper produces the output:

hello from Hyper

as expected.

The super keyword can be used to access a method declared in a superclass, bypassing any methods declared in the current class. The expression super . Identifier is resolved, at compile time, to a method m in the superclass s. If the method m is an instance method, then the method which is invoked at run time is the method with the same signature as m that is a member of the direct superclass of the class containing the expression involving super.

Example 13.4.6-2. Changing A Superclass

```java
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper {
}
class Test extends Super {
    public static void main(String[] args) {
        new Test().hello();
    }
    void hello() {
        super.hello();
    }
}
```

This program produces the output:

hello from Hyper

Suppose that a new version of class Super is produced:

```java
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
```

Then, if Super and Hyper are recompiled but not Test, then running the new binaries with the existing binary of Test produces the output:

hello from Super

as you might expect.
13.4.7 Access to Members and Constructors

Changing the declared access of a member or constructor to permit less access may break compatibility with pre-existing binaries, causing a linkage error to be thrown when these binaries are resolved. Less access is permitted if the access modifier is changed from package access to private access; from protected access to package or private access; or from public access to protected, package, or private access. Changing a member or constructor to permit less access is therefore not recommended for widely distributed classes.

Perhaps surprisingly, the binary format is defined so that changing a member or constructor to be more accessible does not cause a linkage error when a subclass (already) defines a method to have less access.

Example 13.4.7-1. Changing Accessibility

If the package points defines the class Point:

```java
package points;
public class Point {
    public int x, y;
    protected void print() {
        System.out.println("(" + x + "," + y + ")");
    }
}
```

used by the program:

```java
class Test extends points.Point {
    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
    protected void print() {
        System.out.println("Test");
    }
}
```

then these classes compile and Test executes to produce the output:

```
Test
```

If the method print in class Point is changed to be public, and then only the Point class is recompiled, and then executed with the previously existing binary for Test, then no linkage error occurs. This happens even though it is improper, at compile time, for a public method to be overridden by a protected method (as shown by the fact that the class Test could not be recompiled using this new Point class unless print in Test were changed to be public.)
13.4 Evolution of Classes

Allowing superclasses to change protected methods to be public without breaking binaries of pre-existing subclasses helps make binaries less fragile. The alternative, where such a change would cause a linkage error, would create additional binary incompatibilities.

13.4.8 Field Declarations

Widely distributed programs should not expose any fields to their clients. Apart from the binary compatibility issues discussed below, this is generally good software engineering practice. Adding a field to a class may break compatibility with pre-existing binaries that are not recompiled.

Assume a reference to a field \( f \) with qualifying type \( T \). Assume further that \( f \) is in fact an instance (respectively static) field declared in a superclass of \( T, S \), and that the type of \( f \) is \( X \).

If a new field of type \( X \) with the same name as \( f \) is added to a subclass of \( S \) that is a superclass of \( T \) or \( T \) itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following is true:

- The new field is less accessible than the old one.
- The new field is a static (respectively instance) field.

In particular, no linkage error will occur in the case where a class could no longer be recompiled because a field access previously referenced a field of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the field declared in a superclass.

**Example 13.4.8-1. Adding A Field Declaration**

```java
class Hyper { String h = "hyper"; }
class Super extends Hyper { String s = "super"; }
class Test {
    public static void main(String[] args) {
        System.out.println(new Super().h);
    }
}
```

This program produces the output:

```
hyper
```

Suppose a new version of class Super is produced:

```java
class Super extends Hyper {
    String s = "super";
```
Then, recompiling Hyper and Super, and executing the resulting new binaries with the old binary of Test produces the output:

```
hyper
```

The field h of Hyper is output by the original binary of Test. While this may seem surprising at first, it serves to reduce the number of incompatibilities that occur at run time. (In an ideal world, all source files that needed recompilation would be recompiled whenever any one of them changed, eliminating such surprises. But such a mass recompilation is often impractical or impossible, especially in the Internet. And, as was previously noted, such recompilation would sometimes require further changes to the source code.)

As another example, if the program:

```java
class Hyper { String h = "Hyper"; }
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

is compiled and executed, it produces the output:

Hyper

Suppose that a new version of class Super is then compiled:

```java
class Super extends Hyper { char h = 'h'; }
```

If the resulting binary is used with the existing binaries for Hyper and Test, then the output is still:

Hyper

even though compiling the source for these binaries:

```java
class Hyper { String h = "Hyper"; }
class Super extends Hyper { char h = 'h'; }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```
would result in a compile-time error, because the \texttt{h} in the source code for \texttt{main} would now be construed as referring to the \texttt{char} field declared in \texttt{Super}, and a \texttt{char} value can't be assigned to a \texttt{String}.

Deleting a field from a class will break compatibility with any pre-existing binaries that reference this field, and a \texttt{NoSuchFieldError} will be thrown when such a reference from a pre-existing binary is linked. Only \texttt{private} fields may be safely deleted from a widely distributed class.

For purposes of binary compatibility, adding or removing a field $f$ whose type involves type variables ($\S$4.4) or parameterized types ($\S$4.5) is equivalent to the addition (respectively, removal) of a field of the same name whose type is the erasure ($\S$4.6) of the type of $f$.

13.4.9 \textbf{final Fields and static Constant Variables}

If a field that was not declared \texttt{final} is changed to be declared \texttt{final}, then it can break compatibility with pre-existing binaries that attempt to assign new values to the field.

\textbf{Example 13.4.9-1. Changing A Variable To Be final}

\begin{verbatim}
 class Super { char s; }
 class Test extends Super {
   public static void main(String[] args) {
     Super x = new Super();
     x.s = 'a';
     System.out.println(x.s);
   }
 }
\end{verbatim}

This program produces the output:

\texttt{a}

Suppose that a new version of class \texttt{Super} is produced:

\begin{verbatim}
 class Super { final char s = 'b'; }
\end{verbatim}

If \texttt{Super} is recompiled but not \texttt{Test}, then running the new binary with the existing binary of \texttt{Test} results in a \texttt{IllegalAccessError}.

Deleting the keyword \texttt{final} or changing the value to which a field is initialized does not break compatibility with existing binaries.

If a field is a constant variable ($\S$4.12.4), and moreover is \texttt{static}, then deleting the keyword \texttt{final} or changing its value will not break compatibility with pre-
existing binaries by causing them not to run, but they will not see any new value for a usage of the field unless they are recompiled. This result is a side-effect of the decision to support conditional compilation (§14.21). (One might suppose that the new value is not seen if the usage occurs in a constant expression (§15.28) but is seen otherwise. This is not so; pre-existing binaries do not see the new value at all.)

The best way to avoid problems with "inconstant constants" in widely-distributed code is to use static constant variables only for values which truly are unlikely ever to change. Other than for true mathematical constants, we recommend that source code make very sparing use of static constant variables.

If the read-only nature of final is required, a better choice is to declare a private static variable and a suitable accessor method to get its value. Thus we recommend:

```java
private static int N;
public static int getN() { return N; }
```
rather than:

```java
public static final int N = ...;
```

There is no problem with:

```java
public static int N = ...;
```
if N need not be read-only.

### 13.4.10 static Fields

If a field that is not declared private was not declared static and is changed to be declared static, or vice versa, then a linkage error, specifically an IncompatibleClassChangeError, will result if the field is used by a pre-existing binary which expected a field of the other kind. Such changes are not recommended in code that has been widely distributed.

### 13.4.11 transient Fields

Adding or deleting a transient modifier of a field does not break compatibility with pre-existing binaries.

### 13.4.12 Method and Constructor Declarations

Adding a method or constructor declaration to a class will not break compatibility with any pre-existing binaries, even in the case where a type could no longer be
recompiled because an invocation previously referenced a method or constructor of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the method or constructor declared in a superclass.

Assume a reference to a method \( m \) with qualifying type \( T \). Assume further that \( m \) is in fact an instance (respectively static) method declared in a superclass of \( T, S \).

If a new method of type \( X \) with the same signature and return type as \( m \) is added to a subclass of \( S \) that is a superclass of \( T \) or \( T \) itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following is true:

- The new method is less accessible than the old one.
- The new method is a static (respectively instance) method.

Deleting a method or constructor from a class may break compatibility with any pre-existing binary that referenced this method or constructor; a NoSuchMethodError may be thrown when such a reference from a pre-existing binary is linked. Such an error will occur only if no method with a matching signature and return type is declared in a superclass.

If the source code for a non-inner class contains no declared constructors, then a default constructor with no parameters is implicitly declared (§8.8.9). Adding one or more constructor declarations to the source code of such a class will prevent this default constructor from being implicitly declared, effectively deleting a constructor, unless one of the new constructors also has no parameters, thus replacing the default constructor. The default constructor with no parameters is given the same access modifier as the class of its declaration, so any replacement should have as much or more access if compatibility with pre-existing binaries is to be preserved.

13.4.13 Method and Constructor Type Parameters

Adding or removing a type parameter of a method or constructor does not, in itself, have any implications for binary compatibility.

If such a type parameter is used in the type of the method or constructor, that may have the normal implications of changing the aforementioned type.

Renaming a type parameter of a method or constructor has no effect with respect to pre-existing binaries.
Changing the first bound of a type parameter of a method or constructor may change the erasure (§4.6) of any member that uses that type parameter in its own type, and this may affect binary compatibility. Specifically:

- If the type parameter is used as the type of a field, the effect is as if the field was removed and a field with the same name, whose type is the new erasure of the type variable, was added.
- If the type parameter is used as the type of any formal parameter of a method, but not as the return type, the effect is as if that method were removed, and replaced with a new method that is identical except for the types of the aforementioned formal parameters, which now have the new erasure of the type parameter as their type.
- If the type parameter is used as a return type of a method, but not as the type of any formal parameter of the method, the effect is as if that method were removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type parameter.
- If the type parameter is used as a return type of a method and as the type of one or more formal parameters of the method, the effect is as if that method were removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type parameter, and except for the types of the aforementioned formal parameters, which now have the new erasure of the type parameter as their types.

Changing any other bound has no effect on binary compatibility.

### 13.4.14 Method and Constructor Formal Parameters

Changing the name of a formal parameter of a method or constructor does not impact pre-existing binaries.

Changing the name of a method, or the type of a formal parameter to a method or constructor, or adding a parameter to or deleting a parameter from a method or constructor declaration creates a method or constructor with a new signature, and has the combined effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature (§13.4.12).

Changing the type of the last formal parameter of a method from \( T[\] \) to a variable arity parameter (§8.4.1) of type \( T \) (i.e. to \( T\ldots \)), and vice versa, does not impact pre-existing binaries.

For purposes of binary compatibility, adding or removing a method or constructor \( m \) whose signature involves type variables (§4.4) or parameterized types (§4.5)
is equivalent to the addition (respectively, removal) of an otherwise equivalent method whose signature is the erasure (§4.6) of the signature of \( m \).

### 13.4.15 Method Result Type

Changing the result type of a method, or replacing a result type with `void`, or replacing `void` with a result type, has the combined effect of deleting the old method and adding a new method with the new result type or newly `void` result (see §13.4.12).

For purposes of binary compatibility, adding or removing a method or constructor \( m \) whose return type involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of the an otherwise equivalent method whose return type is the erasure (§4.6) of the return type of \( m \).

### 13.4.16 abstract Methods

Changing a method that is declared `abstract` to no longer be declared `abstract` does not break compatibility with pre-existing binaries.

Changing a method that is not declared `abstract` to be declared `abstract` will break compatibility with pre-existing binaries that previously invoked the method, causing an `AbstractMethodError`.

**Example 13.4.16-1. Changing A Method To Be abstract**

```java
class Super { void out() { System.out.println("Out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println("Way ");
        t.out();
    }
}
```

This program produces the output:

```
Way
Out
```

Suppose that a new version of class `Super` is produced:

```java
abstract class Super {
    abstract void out();
}
```
If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in an `AbstractMethodError`, because class `Test` has no implementation of the method `out`, and is therefore is (or should be) abstract.

### 13.4.17 final Methods

Changing a method that is declared `final` to no longer be declared `final` does not break compatibility with pre-existing binaries.

Changing an instance method that is not declared `final` to be declared `final` may break compatibility with existing binaries that depend on the ability to override the method.

**Example 13.4.17-1. Changing A Method To Be final**

```java
class Super { void out() { System.out.println("out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        t.out();
    }
    void out() { super.out(); }
}
```

This program produces the output:

```
out
```

Suppose that a new version of class `Super` is produced:

```java
class Super { final void out() { System.out.println("!"); } }
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `VerifyError` because the class `Test` improperly tries to override the instance method `out`.

Changing a class (static) method that is not declared `final` to be declared `final` does not break compatibility with existing binaries, because the method could not have been overridden.

### 13.4.18 native Methods

Adding or deleting a `native` modifier of a method does not break compatibility with pre-existing binaries.

The impact of changes to types on pre-existing native methods that are not recompiled is beyond the scope of this specification and should be provided with
the description of an implementation. Implementations are encouraged, but not required, to implement `native` methods in a way that limits such impact.

### 13.4.19 Static Methods

If a method that is not declared `private` is also declared `static` (that is, a class method) and is changed to not be declared `static` (that is, to an instance method), or vice versa, then compatibility with pre-existing binaries may be broken, resulting in a linkage time error, namely an `IncompatibleClassChangeError`, if these methods are used by the pre-existing binaries. Such changes are not recommended in code that has been widely distributed.

### 13.4.20 Synchronized Methods

Adding or deleting a `synchronized` modifier of a method does not break compatibility with pre-existing binaries.

### 13.4.21 Method and Constructor Throws

Changes to the `throws` clause of methods or constructors do not break compatibility with pre-existing binaries; these clauses are checked only at compile time.

### 13.4.22 Method and Constructor Body

Changes to the body of a method or constructor do not break compatibility with pre-existing binaries.

The keyword `final` on a method does not mean that the method can be safely inlined; it means only that the method cannot be overridden. It is still possible that a new version of that method will be provided at link-time. Furthermore, the structure of the original program must be preserved for purposes of reflection.

Therefore, we note that a Java compiler cannot expand a method inline at compile time. In general we suggest that implementations use late-bound (run-time) code generation and optimization.

### 13.4.23 Method and Constructor Overloading

Adding new methods or constructors that overload existing methods or constructors does not break compatibility with pre-existing binaries. The signature to be used for each invocation was determined when these existing binaries were compiled;
therefore newly added methods or constructors will not be used, even if their signatures are both applicable and more specific than the signature originally chosen.

While adding a new overloaded method or constructor may cause a compile-time error the next time a class or interface is compiled because there is no method or constructor that is most specific (§15.12.2.5), no such error occurs when a program is executed, because no overload resolution is done at execution time.

**Example 13.4.23-1. Adding An Overloaded Method**

class Super {
    static void out(float f) {
        System.out.println("float");
    }
}
class Test {
    public static void main(String[] args) {
        Super.out(2);
    }
}

This program produces the output:

float

Suppose that a new version of class Super is produced:

class Super {
    static void out(float f) { System.out.println("float"); }
    static void out(int i)   { System.out.println("int");   }
}

If Super is recompiled but not Test, then running the new binary with the existing binary of Test still produces the output:

float

However, if Test is then recompiled, using this new Super, the output is then:

int

as might have been naively expected in the previous case.
13.4.24 Method Overriding

If an instance method is added to a subclass and it overrides a method in a superclass, then the subclass method will be found by method invocations in pre-existing binaries, and these binaries are not impacted.

If a class method is added to a class, then this method will not be found unless the qualifying type of the reference is the subclass type.

13.4.25 Static Initializers

Adding, deleting, or changing a static initializer (§8.7) of a class does not impact pre-existing binaries.

13.4.26 Evolution of Enums

Adding or reordering constants in an enum will not break compatibility with pre-existing binaries.

If a pre-existing binary attempts to access an enum constant that no longer exists, the client will fail at run time with a NoSuchFieldError. Therefore such a change is not recommended for widely distributed enums.

In all other respects, the binary compatibility rules for enums are identical to those for classes.

13.5 Evolution of Interfaces

This section describes the impact of changes to the declaration of an interface and its members on pre-existing binaries.

13.5.1 public Interfaces

Changing an interface that is not declared public to be declared public does not break compatibility with pre-existing binaries.

If an interface that is declared public is changed to not be declared public, then an IllegalAccessError is thrown if a pre-existing binary is linked that needs but no longer has access to the interface type, so such a change is not recommended for widely distributed interfaces.
13.5.2 Superinterfaces

Changes to the interface hierarchy cause errors in the same way that changes to the class hierarchy do, as described in §13.4.4. In particular, changes that result in any previous superinterface of a class no longer being a superinterface can break compatibility with pre-existing binaries, resulting in a VerifyError.

13.5.3 Interface Members

Adding an abstract, private, or static method to an interface does not break compatibility with pre-existing binaries.

Adding a field to a superinterface of C may hide a field inherited from a superclass of C. If the original reference was to an instance field, an IncompatibleClassChangeError will result. If the original reference was an assignment, an IllegalAccessError will result.

Deleting a member from an interface may cause linkage errors in pre-existing binaries.

Example 13.5.3-1. Deleting An Interface Member

interface I { void hello(); }
class Test implements I {
    public static void main(String[] args) {
        I anI = new Test();
        anI.hello();
    }
    public void hello() { System.out.println("hello"); }
}

This program produces the output:

hello

Suppose that a new version of interface I is compiled:

interface I {} 

If I is recompiled but not Test, then running the new binary with the existing binary for Test will result in a NoSuchMethodError.

13.5.4 Interface Type Parameters

The effects of changes to the type parameters of an interface are the same as those of analogous changes to the type parameters of a class.
13.5.5 Field Declarations

The considerations for changing field declarations in interfaces are the same as those for `static final` fields in classes, as described in §13.4.8 and §13.4.9.

13.5.6 Interface Method Declarations

The considerations for changing method declarations in interfaces include those for changing methods in classes, as described in §13.4.7, §13.4.14, §13.4.15, §13.4.19, §13.4.21, §13.4.22, and §13.4.23.

Adding a `default` method, or changing a method from `abstract` to `default`, does not break compatibility with pre-existing binaries, but may cause an `IncompatibleClassChangeError` if a pre-existing binary attempts to invoke the method. This error occurs if the qualifying type, T, is a subtype of two interfaces, I and J, where both I and J declare a `default` method with the same signature and result, and neither I nor J is a subinterface of the other.

In other words, adding a default method is a binary-compatible change because it does not introduce errors at link time, even if it introduces errors at compile time or invocation time. In practice, the risk of accidental clashes occurring by introducing a default method are similar to those associated with adding a new method to a non-final class. In the event of a clash, adding a method to a class is unlikely to trigger a `LinkageError`, but an accidental override of the method in a child can lead to unpredictable method behavior. Both changes can cause errors at compile time.

**Example 13.5.6-1. Adding A Default Method**

```java
interface Painter {
    default void draw() {
        System.out.println("Here's a picture...");
    }
}

interface Cowboy {
}

public class CowboyArtist implements Cowboy, Painter {
    public static void main(String... args) {
        new CowboyArtist().draw();
    }
}
```

This program produces the output:

Here's a picture...

Suppose that a default method is added to `Cowboy`:
interface Cowboy {
    default void draw() {
        System.out.println("Bang!");
    }
}

If Cowboy is recompiled but not CowboyArtist, then running the new binary with the existing binary for CowboyArtist will link without error but cause an IncompatibleClassChangeError when main attempts to invoke draw().

### 13.5.7 Evolution of Annotation Types

Annotation types behave exactly like any other interface. Adding or removing an element from an annotation type is analogous to adding or removing a method. There are important considerations governing other changes to annotation types, such as making an annotation type repeatable (§9.6.3), but these have no effect on the linkage of binaries by the Java Virtual Machine. Rather, such changes affect the behavior of reflective APIs that manipulate annotations. The documentation of these APIs specifies their behavior when various changes are made to the underlying annotation types.

Adding or removing annotations has no effect on the correct linkage of the binary representations of programs in the Java programming language.
THE sequence of execution of a program is controlled by *statements*, which are executed for their effect and do not have values.

Some statements *contain* other statements as part of their structure; such other statements are substatements of the statement. We say that statement $S$ *immediately contains* statement $U$ if there is no statement $T$ different from $S$ and $U$ such that $S$ contains $T$ and $T$ contains $U$. In the same manner, some statements contain expressions (§15 (*Expressions*)) as part of their structure.

The first section of this chapter discusses the distinction between normal and abrupt completion of statements (§14.1). Most of the remaining sections explain the various kinds of statements, describing in detail both their normal behavior and any special treatment of abrupt completion.

Blocks are explained first (§14.2), followed by local class declarations (§14.3) and local variable declaration statements (§14.4).

Next a grammatical maneuver that sidesteps the familiar "dangling else" problem (§14.5) is explained.

The last section (§14.21) of this chapter addresses the requirement that every statement be *reachable* in a certain technical sense.

### 14.1 Normal and Abrupt Completion of Statements

Every statement has a normal mode of execution in which certain computational steps are carried out. The following sections describe the normal mode of execution for each kind of statement.
If all the steps are carried out as described, with no indication of abrupt completion, the statement is said to complete normally. However, certain events may prevent a statement from completing normally:

- The `break` (§14.15), `continue` (§14.16), and `return` (§14.17) statements cause a transfer of control that may prevent normal completion of statements that contain them.

- Evaluation of certain expressions may throw exceptions from the Java Virtual Machine (§15.6). An explicit `throw` (§14.18) statement also results in an exception. An exception causes a transfer of control that may prevent normal completion of statements.

If such an event occurs, then execution of one or more statements may be terminated before all steps of their normal mode of execution have completed; such statements are said to complete abruptly.

An abrupt completion always has an associated reason, which is one of the following:

- A `break` with no label
- A `break` with a given label
- A `continue` with no label
- A `continue` with a given label
- A `return` with no value
- A `return` with a given value
- A `throw` with a given value, including exceptions thrown by the Java Virtual Machine

The terms "complete normally" and "complete abruptly" also apply to the evaluation of expressions (§15.6). The only reason an expression can complete abruptly is that an exception is thrown, because of either a `throw` with a given value (§14.18) or a run-time exception or error (§11 (Exceptions), §15.6).

If a statement evaluates an expression, abrupt completion of the expression always causes the immediate abrupt completion of the statement, with the same reason. All succeeding steps in the normal mode of execution are not performed.

Unless otherwise specified in this chapter, abrupt completion of a substatement causes the immediate abrupt completion of the statement itself, with the same reason, and all succeeding steps in the normal mode of execution of the statement are not performed.
Unless otherwise specified, a statement completes normally if all expressions it evaluates and all substatements it executes complete normally.

### 14.2 Blocks

A block is a sequence of statements, local class declarations, and local variable declaration statements within braces.

```
Block:
    { [BlockStatements] }
```

```
BlockStatements:
    BlockStatement {BlockStatement}
```

```
BlockStatement:
    LocalVariableDeclarationStatement
    ClassDeclaration
    Statement
```

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

### 14.3 Local Class Declarations

A local class is a nested class (§8 (Classes)) that is not a member of any class and that has a name (§6.2, §6.7).

All local classes are inner classes (§8.1.3).

Every local class declaration statement is immediately contained by a block (§14.2). Local class declaration statements may be intermixed freely with other kinds of statements in the block.

It is a compile-time error if a local class declaration contains any of the access modifiers public, protected, or private (§6.6), or the modifier static (§8.1.1).

The scope and shadowing of a local class declaration is specified in §6.3 and §6.4.
Example 14.3-1. Local Class Declarations

Here is an example that illustrates several aspects of the rules given above:

```java
class Global {
    class Cyclic {}

    void foo() {
        new Cyclic(); // create a Global.Cyclic
        class Cyclic extends Cyclic {} // circular definition
    }
}
```

The first statement of method `foo` creates an instance of the member class `Global.Cyclic` rather than an instance of the local class `Cyclic`, because the statement appears prior to the scope of the local class declaration.

The fact that the scope of a local class declaration encompasses its whole declaration (not only its body) means that the definition of the local class `Cyclic` is indeed cyclic because it extends itself rather than `Global.Cyclic`. Consequently, the declaration of the local class `Cyclic` is rejected at compile time.

Since local class names cannot be redeclared within the same method (or constructor or initializer, as the case may be), the second and third declarations of `Local` result in compile-time errors. However, `Local` can be redeclared in the context of another, more deeply nested, class such as `AnotherLocal`.

The final declaration of `Local` is legal, since it occurs outside the scope of any prior declaration of `Local`.

14.4 Local Variable Declaration Statements

A local variable declaration statement declares one or more local variable names.
LocalVariableDeclarationStatement:
   LocalVariableDeclaration;

LocalVariableDeclaration:
   {VariableModifier} LocalVariableType VariableDeclaratorList

LocalVariableType:
   UnannType
   var

See §8.3 for UnannType. The following productions from §4.3, §8.3, and §8.4.1 are shown here for convenience:

   VariableModifier:
   Annotation
   final

   VariableDeclaratorList:
   VariableDeclarator {, VariableDeclarator}

   VariableDeclarator:
   VariableDeclaratorId [= VariableInitializer]

   VariableDeclaratorId:
   Identifier [Dims]

   Dims:
   {Annotation} [ ] {{Annotation} [ ]}

   VariableInitializer:
   Expression
   ArrayInitializer

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

Apart from local variable declaration statements, a local variable can be declared by the header of a basic for statement (§14.14.1), an enhanced for statement (§14.14.2), or a try-with-resources statement (§14.20.3).

The rules for annotation modifiers on a local variable declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if final appears more than once as a modifier for a local variable declaration.

It is a compile-time error if the LocalVariableType is var and any of the following are true:
• More than one \textit{VariableDeclarator} is listed.
• The \textit{VariableDeclaratorId} has one or more bracket pairs.
• The \textit{VariableDeclarator} lacks an initializer.
• The initializer of the \textit{VariableDeclarator} is an \textit{ArrayInitializer}.
• The initializer of the \textit{VariableDeclarator} contains a reference to the variable.

\textbf{Example 14.4-1. Local Variables Declared With \texttt{var}}

The following code illustrates these rules restricting the use of \texttt{var}:

\begin{verbatim}
var a = 1;            // Legal
var b = 2, c = 3.0;   // Illegal: multiple declarators
var d[] = new int[4]; // Illegal: extra bracket pairs
var e;                // Illegal: no initializer
var f = { 6 };        // Illegal: array initializer
var g = (g = 7);      // Illegal: self reference in initializer
\end{verbatim}

These restrictions help to avoid confusion about the type being represented by \texttt{var}.

\section*{14.4.1 Local Variable Declarators and Types}

Each \textit{declarator} in a local variable declaration declares one local variable, whose
name is the \textit{Identifier} that appears in the declarator.

If the optional keyword \texttt{final} appears at the start of the declaration, the variable
being declared is a final variable (§4.12.4).

The declared type of a local variable is determined as follows:

• If \textit{LocalVariableType} is an \textit{UnannType}, and no bracket pairs appear in
  \textit{UnannType} or \textit{VariableDeclaratorId}, then \textit{UnannType} denotes the type of the
  local variable.

• If \textit{LocalVariableType} is an \textit{UnannType}, and bracket pairs appear in \textit{UnannType}
  or \textit{VariableDeclaratorId}, then the type of the local variable is specified by §10.2.

• If \textit{LocalVariableType} is \texttt{var}, then let $\tau$ be the type of the initializer expression
  when treated as if it did not appear in an assignment context, and were thus
  a standalone expression (§15.2). The type of the local variable is the upward
  projection of $\tau$ with respect to all synthetic type variables mentioned by $\tau$
  (§4.10.5).

  It is a compile-time error if $\tau$ is the null type.
Because the initializer is treated as if it did not appear in an assignment context, an error occurs if it is a lambda expression (§15.27) or a method reference expression (§15.13).

**Example 14.4.1-1. Type of Local Variables Declared With var**

The following code illustrates the typing of variables declared with `var`:

```java
var a = 1; // a has type 'int'
var b = java.util.List.of(1, 2); // b has type 'List<Integer>'
var c = "x".getClass(); // c has type 'Class<? extends String>'
// (see JLS 15.12.2.6)
var d = new Object() {}; // d has the type of the anonymous class
var e = (CharSequence & Comparable<String>) "x";
// e has type CharSequence & Comparable<String>
var f = () -> "hello"; // Illegal: lambda not in an assignment context
var g = null; // Illegal: null type
```

Note that some variables declared with `var` cannot be declared with an explicit type, because the type of the variable is not denotable.

Upward projection is applied to the type of the initializer when determining the type of the variable. If the type of the initializer contains capture variables, this projection maps the type of the initializer to a supertype that does not contain capture variables.

While it would be possible to allow the type of the variable to mention capture variables, by projecting them away we enforce an attractive invariant that the scope of a capture variable is never larger than the statement containing the expression whose type is captured. Informally, capture variables cannot "leak" into subsequent statements.

A local variable of type `float` always contains a value that is an element of the float value set (§4.2.3); similarly, a local variable of type `double` always contains a value that is an element of the double value set. It is not permitted for a local variable of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a local variable of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

The scope and shadowing of a local variable declaration is specified in §6.3 and §6.4.

### 14.4.2 Execution of Local Variable Declarations

A local variable declaration statement is an executable statement. Every time it is executed, the declarators are processed in order from left to right. If a declarator has an initializer, the initializer is evaluated and its value is assigned to the variable.
If a declarator does not have an initializer, then every reference to the variable must be preceded by execution of an assignment to the variable, or a compile-time error occurs by the rules of §16 (Definite Assignment).

Each initializer (except the first) is evaluated only if evaluation of the preceding initializer completes normally.

Execution of the local variable declaration completes normally only if evaluation of the last initializer completes normally.

If the local variable declaration contains no initializers, then executing it always completes normally.

14.5 Statements

There are many kinds of statements in the Java programming language. Most correspond to statements in the C and C++ languages, but some are unique.

As in C and C++, the if statement of the Java programming language suffers from the so-called "dangling else problem," illustrated by this misleadingly formatted example:

```java
if (door.isOpen())
    if (resident.isVisible())
        resident.greet("Hello!");
else door.bell.ring();  // A "dangling else"
```

The problem is that both the outer if statement and the inner if statement might conceivably own the else clause. In this example, one might surmise that the programmer intended the else clause to belong to the outer if statement.

The Java programming language, like C and C++ and many programming languages before them, arbitrarily decrees that an else clause belongs to the innermost if to which it might possibly belong. This rule is captured by the following grammar:

```
Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement
```
StatementNoShortIf:
   StatementWithoutTrailingSubstatement
   LabeledStatementNoShortIf
   IfThenElseStatementNoShortIf
   WhileStatementNoShortIf
   ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
   Block
   EmptyStatement
   ExpressionStatement
   AssertStatement
   SwitchStatement
   DoStatement
   BreakStatement
   ContinueStatement
   ReturnStatement
   SynchronizedStatement
   ThrowStatement
   TryStatement

The following productions from §14.9 are shown here for convenience:

   IfThenStatement:
      if ( Expression ) Statement

   IfThenElseStatement:
      if ( Expression ) StatementNoShortIf else Statement

   IfThenElseStatementNoShortIf:
      if ( Expression ) StatementNoShortIf else StatementNoShortIf

Statements are thus grammatically divided into two categories: those that might end in an if statement that has no else clause (a "short if statement") and those that definitely do not.

Only statements that definitely do not end in a short if statement may appear as an immediate substatement before the keyword else in an if statement that does have an else clause.

This simple rule prevents the "dangling else" problem. The execution behavior of a statement with the "no short if" restriction is identical to the execution behavior of the same kind of statement without the "no short if" restriction; the distinction is drawn purely to resolve the syntactic difficulty.
14.6 The Empty Statement

An empty statement does nothing.

*EmptyStatement:*

; Execution of an empty statement always completes normally.

14.7 Labeled Statements

Statements may have *label* prefixes.

*LabeledStatement:*

*Identifier* : *Statement*

*LabeledStatementNoShortIf:*

*Identifier* : *StatementNoShortIf*

The *Identifier* is declared to be the label of the immediately contained *Statement*.

Unlike C and C++, the Java programming language has no *goto* statement; identifier statement labels are used with *break* or *continue* statements (§14.15, §14.16) appearing anywhere within the labeled statement.

The scope of a label of a labeled statement is the immediately contained *Statement*.

It is a compile-time error if the name of a label of a labeled statement is used within the scope of the label as a label of another labeled statement.

There is no restriction against using the same identifier as a label and as the name of a package, class, interface, method, field, parameter, or local variable. Use of an identifier to label a statement does not obscure (§6.4.2) a package, class, interface, method, field, parameter, or local variable with the same name. Use of an identifier as a class, interface, method, field, local variable or as the parameter of an exception handler (§14.20) does not obscure a statement label with the same name.

A labeled statement is executed by executing the immediately contained *Statement*.

If the statement is labeled by an *Identifier* and the contained *Statement* completes abruptly because of a *break* with the same *Identifier*, then the labeled statement completes normally. In all other cases of abrupt completion of the *Statement*, the labeled statement completes abruptly for the same reason.
Example 14.7-1. Labels and Identifiers

The following code was taken from a version of the class `String` and its method `indexOf`, where the label was originally called `test`. Changing the label to have the same name as the local variable `i` does not obscure the label in the scope of the declaration of `i`. Thus, the code is valid.

class Test {
    char[] value;
    int offset, count;
    int indexOf(TestString str, int fromIndex) {
        char[] v1 = value, v2 = str.value;
        int max = offset + (count - str.count);
        int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
        i:
            for (int i = start; i <= max; i++) {
                int n = str.count, j = i, k = str.offset;
                while (n-- != 0) {
                    if (v1[j++] != v2[k++])
                        continue i;
                }
                return i - offset;
            }
        return -1;
    }
}

The identifier `max` could also have been used as the statement label; the label would not obscure the local variable `max` within the labeled statement.

14.8 Expression Statements

Certain kinds of expressions may be used as statements by following them with semicolons.

ExpressionStatement:
   StatementExpression ;
The `if` Statement

The `if` statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

```
IfThenStatement:
    if ( Expression ) Statement
```
IfThenElseStatement:
   if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
   if ( Expression ) StatementNoShortIf else StatementNoShortIf

The Expression must have type boolean or Boolean, or a compile-time error occurs.

14.9.1 The if-then Statement

An if-then statement is executed by first evaluating the Expression. If the result is of type boolean, it is subject to unboxing conversion (§5.1.8).

If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the if-then statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

- If the value is true, then the contained Statement is executed; the if-then statement completes normally if and only if execution of the Statement completes normally.
- If the value is false, no further action is taken and the if-then statement completes normally.

14.9.2 The if-then-else Statement

An if-then-else statement is executed by first evaluating the Expression. If the result is of type boolean, it is subject to unboxing conversion (§5.1.8).

If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, then the if-then-else statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

- If the value is true, then the first contained Statement (the one before the else keyword) is executed; the if-then-else statement completes normally if and only if execution of that statement completes normally.
- If the value is false, then the second contained Statement (the one after the else keyword) is executed; the if-then-else statement completes normally if and only if execution of that statement completes normally.
14.10 The assert Statement

An assertion is an assert statement containing a boolean expression. An assertion is either enabled or disabled. If an assertion is enabled, execution of the assertion causes evaluation of the boolean expression and an error is reported if the expression evaluates to false. If the assertion is disabled, execution of the assertion has no effect whatsoever.

AssertStatement:

\[
\text{assert Expression ;} \\
\text{assert Expression : Expression ;}
\]

To ease the presentation, the first Expression in both forms of the assert statement is referred to as Expression1. In the second form of the assert statement, the second Expression is referred to as Expression2.

It is a compile-time error if Expression1 does not have type boolean or Boolean.

It is a compile-time error if, in the second form of the assert statement, Expression2 is void (§15.1).

An assert statement that is executed after its class or interface has completed initialization is enabled if and only if the host system has determined that the top level class or interface that lexically contains the assert statement enables assertions.

Whether a top level class or interface enables assertions is determined no later than the earliest of (i) the initialization of the top level class or interface, and (ii) the initialization of any class or interface nested in the top level class or interface. Whether a top level class or interface enables assertions cannot be changed after it has been determined.

An assert statement that is executed before its class or interface has completed initialization is enabled.

This rule is motivated by a case that demands special treatment. Recall that the assertion status of a class is set no later than the time it is initialized. It is possible, though generally not desirable, to execute methods or constructors prior to initialization. This can happen when a class hierarchy contains a circularity in its static initialization, as in the following example:

```java
public class Foo {
    public static void main(String[] args) {
        Baz.testAsserts();
        // Will execute after Baz is initialized.
    }
```
class Bar {
    static {
        Baz.testAsserts();
        // Will execute before Baz is initialized!
    }
}

class Baz extends Bar {
    static void testAsserts() {
        boolean enabled = false;
        assert enabled = true;
        System.out.println("Asserts " +
                (enabled ? "enabled" : "disabled"));
    }
}

Invoking Baz.testAsserts() causes Baz to be initialized. Before this can happen, Bar
must be initialized. Bar's static initializer again invokes Baz.testAsserts(). Because
initialization of Baz is already in progress by the current thread, the second invocation
executes immediately, though Baz is not initialized (§12.4.2).

Because of the rule above, if the program above is executed without enabling assertions,
it must print:

    Asserts enabled
    Asserts disabled

A disabled assert statement does nothing. In particular, neither Expression1
nor Expression2 (if it is present) are evaluated. Execution of a disabled assert
statement always completes normally.

An enabled assert statement is executed by first evaluating Expression1. If the
result is of type Boolean, it is subject to unboxing conversion (§5.1.8).

If evaluation of Expression1 or the subsequent unboxing conversion (if any)
completes abruptly for some reason, the assert statement completes abruptly for
the same reason.

Otherwise, execution continues by making a choice based on the value of
Expression1:

- If the value is true, no further action is taken and the assert statement completes
  normally.
- If the value is false, the execution behavior depends on whether Expression2
  is present:
    - If Expression2 is present, it is evaluated. Then:
If the evaluation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.

If the evaluation completes normally, an `AssertionError` instance whose "detail message" is the resulting value of `Expression2` is created. Then:

- If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
- If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.

- If `Expression2` is not present, an `AssertionError` instance with no "detail message" is created. Then:
  - If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
  - If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.

Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.

Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects. Evaluating such a boolean expression should not affect any state that is visible after the evaluation is complete. It is not illegal for a boolean expression contained in an assertion to have a side effect, but it is generally inappropriate, as it could cause program behavior to vary depending on whether assertions were enabled or disabled.

In light of this, assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.

A secondary problem with using assertions for argument checking is that erroneous arguments should result in an appropriate run-time exception (such as `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception. Again, it is not illegal to use assertions for argument checking on public methods, but it is generally inappropriate. It is intended that `AssertionError` never be caught, but it is possible to do so, thus the rules for `try` statements should treat assertions appearing in a `try` block similarly to the current treatment of `throw` statements.
14.11 The switch Statement

The switch statement transfers control to one of several statements depending on the value of an expression.

\[
\text{SwitchStatement:} \\
\quad \text{switch ( Expression ) SwitchBlock}
\]

\[
\text{SwitchBlock:} \\
\quad \{ \{ \text{SwitchBlockStatementGroup} \} \{ \text{SwitchLabel} \} \}
\]

\[
\text{SwitchBlockStatementGroup:} \\
\quad \text{SwitchLabels BlockStatements}
\]

\[
\text{SwitchLabels:} \\
\quad \text{SwitchLabel [SwitchLabel]}
\]

\[
\text{SwitchLabel:} \\
\quad \text{case ConstantExpression :} \\
\quad \text{case EnumConstantName :} \\
\quad \text{default :}
\]

\[
\text{EnumConstantName:} \\
\quad \text{Identifier}
\]

The type of the Expression must be char, byte, short, int, Character, Byte, Short, Integer, String, or an enum type (§8.9), or a compile-time error occurs.

The body of a switch statement is known as a switch block. Any statement immediately contained by the switch block may be labeled with one or more switch labels, which are case or default labels. Every case label has a case constant, which is either a constant expression or the name of an enum constant. Switch labels and their case constants are said to be associated with the switch statement.

Given a switch statement, all of the following must be true or a compile-time error occurs:

- Every case constant associated with the switch statement must be assignment compatible with the type of the switch statement's Expression (§5.2).
- If the type of the switch statement's Expression is an enum type, then every case constant associated with the switch statement must be an enum constant of that type.
• No two of the case constants associated with the switch statement have the same value.

• No case constant associated with the switch statement is null.

• At most one default label is associated with the switch statement.

The prohibition against using null as a case constant prevents code being written that can never be executed. If the switch statement’s Expression is of a reference type, that is, String or a boxed primitive type or an enum type, then an exception will be thrown if the Expression evaluates to null at run time. In the judgment of the designers of the Java programming language, this is a better outcome than silently skipping the entire switch statement or choosing to execute the statements (if any) after the default label (if any).

A Java compiler is encouraged (but not required) to provide a warning if a switch on an enum-valued expression lacks a default label and lacks case labels for one or more of the enum’s constants. Such a switch will silently do nothing if the expression evaluates to one of the missing constants.

In C and C++ the body of a switch statement can be a statement and statements with case labels do not have to be immediately contained by that statement. Consider the simple loop:

```java
for (i = 0; i < n; ++i) foo();
```

where n is known to be positive. A trick known as Duff’s device can be used in C or C++ to unroll the loop, but this is not valid code in the Java programming language:

```java
int q = (n+7)/8;
switch (n%8) {
    case 0: do { foo(); // Great C hack, Tom,
                foo(); // but it's not valid here.
    case 7:      foo();
    case 6:      foo();
    case 5:      foo();
    case 4:      foo();
    case 3:      foo();
    case 2:      foo();
    case 1:      foo();
          } while (--q > 0);
}
```

Fortunately, this trick does not seem to be widely known or used. Moreover, it is less needed nowadays; this sort of code transformation is properly in the province of state-of-the-art optimizing compilers.

A switch statement is executed by first evaluating the Expression. If the Expression evaluates to null, a NullPointerException is thrown and the entire switch statement completes abruptly for that reason. Otherwise, if the result is of
type Character, Byte, Short, or Integer, it is subject to unboxing conversion (§5.1.8).

If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the switch statement completes abruptly for the same reason.

Otherwise, execution continues by comparing the value of the Expression with each case constant, and there is a choice:

- If one of the case constants is equal to the value of the expression, then we say that the case label matches. Equality is defined in terms of the == operator (§15.21) unless the value of the expression is a String, in which case equality is defined in terms of the equals method of class String.

All statements after the matching case label in the switch block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the matching case label, then the entire switch statement completes normally.

- If no case label matches but there is a default label, then all statements after the default label in the switch block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the default label, then the entire switch statement completes normally.

- If no case label matches and there is no default label, then no further action is taken and the switch statement completes normally.

If any statement immediately contained by the Block body of the switch statement completes abruptly, it is handled as follows:

- If execution of the Statement completes abruptly because of a break with no label, no further action is taken and the switch statement completes normally.

- If execution of the Statement completes abruptly for any other reason, the switch statement completes abruptly for the same reason.

The case of abrupt completion because of a break with a label is handled by the general rule for labeled statements (§14.7).

Example 14.11-1. Fall-Through in the switch Statement

As in C and C++, execution of statements in a switch block "falls through labels."

For example, the program:

```
class TooMany {
```
contains a switch block in which the code for each case falls through into the code for the next case. As a result, the program prints:

    many
too many
one too many

If code is not to fall through case to case in this manner, then break statements should be used, as in this example:

class TwoMany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.println("one");
            break;  // exit the switch
            case 2: System.out.println("two");
            break;  // exit the switch
            case 3: System.out.println("many");
            break;  // not needed, but good style
        }
    }
    public static void main(String[] args) {
        howMany(1);
        howMany(2);
        howMany(3);
    }
}

This program prints:

    one
two
many
14.12 The while Statement

The while statement executes an Expression and a Statement repeatedly until the value of the Expression is false.

WhileStatement:

while (Expression) Statement

WhileStatementNoShortIf:

while (Expression) StatementNoShortIf

The Expression must have type boolean or Boolean, or a compile-time error occurs.

A while statement is executed by first evaluating the Expression. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8).

If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the while statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

- If the value is true, then the contained Statement is executed. Then there is a choice:
  - If execution of the Statement completes normally, then the entire while statement is executed again, beginning by re-evaluating the Expression.
  - If execution of the Statement completes abruptly, see §14.12.1.

- If the (possibly unboxed) value of the Expression is false, no further action is taken and the while statement completes normally.

  If the (possibly unboxed) value of the Expression is false the first time it is evaluated, then the Statement is not executed.

14.12.1 Abrupt Completion of while Statement

Abrupt completion of the contained Statement is handled in the following manner:

- If execution of the Statement completes abruptly because of a break with no label, no further action is taken and the while statement completes normally.

- If execution of the Statement completes abruptly because of a continue with no label, then the entire while statement is executed again.
• If execution of the Statement completes abruptly because of a continue with label L, then there is a choice:
  – If the while statement has label L, then the entire while statement is executed again.
  – If the while statement does not have label L, the while statement completes abruptly because of a continue with label L.

• If execution of the Statement completes abruptly for any other reason, the while statement completes abruptly for the same reason.

  The case of abrupt completion because of a break with a label is handled by the general rule for labeled statements (§14.7).

### 14.13 The do Statement

The do statement executes a Statement and an Expression repeatedly until the value of the Expression is false.

$$\text{DoStatement:}$$
$$\text{do Statement while ( Expression ) ;}$$

The Expression must have type boolean or Boolean, or a compile-time error occurs.

A do statement is executed by first executing the Statement. Then there is a choice:

• If execution of the Statement completes normally, then the Expression is evaluated. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8).

  If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the do statement completes abruptly for the same reason.

  Otherwise, there is a choice based on the resulting value:
  – If the value is true, then the entire do statement is executed again.
  – If the value is false, no further action is taken and the do statement completes normally.

• If execution of the Statement completes abruptly, see §14.13.1.

  Executing a do statement always executes the contained Statement at least once.
14.13.1 Abrupt Completion of do Statement

Abrupt completion of the contained Statement is handled in the following manner:

- If execution of the Statement completes abruptly because of a break with no label, then no further action is taken and the do statement completes normally.

- If execution of the Statement completes abruptly because of a continue with no label, then the Expression is evaluated. Then there is a choice based on the resulting value:
  - If the value is true, then the entire do statement is executed again.
  - If the value is false, no further action is taken and the do statement completes normally.

- If execution of the Statement completes abruptly because of a continue with label $L$, then there is a choice:
  - If the do statement has label $L$, then the Expression is evaluated. Then there is a choice:
    - If the value of the Expression is true, then the entire do statement is executed again.
    - If the value of the Expression is false, no further action is taken and the do statement completes normally.
  - If the do statement does not have label $L$, the do statement completes abruptly because of a continue with label $L$.

- If execution of the Statement completes abruptly for any other reason, the do statement completes abruptly for the same reason.

The case of abrupt completion because of a break with a label is handled by the general rule for labeled statements (§14.7).

Example 14.13-1. The do Statement

The following code is one possible implementation of the toHexString method of class Integer:

```java
public static String toHexString(int i) {
    StringBuffer buf = new StringBuffer(8);
    do {
        buf.append(Character.forDigit(i & 0xF, 16));
        i >>>= 4;
    } while (i != 0);
    return buf.reverse().toString();
}
```
Because at least one digit must be generated, the do statement is an appropriate control structure.

14.14 The for Statement

The for statement has two forms:

- The basic for statement.
- The enhanced for statement

ForStatement:
  BasicForStatement
  EnhancedForStatement

ForStatementNoShortIf:
  BasicForStatementNoShortIf
  EnhancedForStatementNoShortIf

14.14.1 The basic for Statement

The basic for statement executes some initialization code, then executes an Expression, a Statement, and some update code repeatedly until the value of the Expression is false.

BasicForStatement:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement

BasicForStatementNoShortIf:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) StatementNoShortIf

ForInit:
  StatementExpressionList
  LocalVariableDeclaration

ForUpdate:
  StatementExpressionList

StatementExpressionList:
  StatementExpression { , StatementExpression}
The Expression must have type boolean or Boolean, or a compile-time error occurs.

The scope and shadowing of a local variable declared in the ForInit part of a basic for statement is specified in §6.3 and §6.4.

14.14.1.1 Initialization of for Statement

A for statement is executed by first executing the ForInit code:

• If the ForInit code is a list of statement expressions (§14.8), the expressions are evaluated in sequence from left to right; their values, if any, are discarded.

  If evaluation of any expression completes abruptly for some reason, the for statement completes abruptly for the same reason; any ForInit statement expressions to the right of the one that completed abruptly are not evaluated.

• If the ForInit code is a local variable declaration (§14.4), it is executed as if it were a local variable declaration statement appearing in a block.

  If execution of the local variable declaration completes abruptly for any reason, the for statement completes abruptly for the same reason.

• If the ForInit part is not present, no action is taken.

14.14.1.2 Iteration of for Statement

Next, a for iteration step is performed, as follows:

• If the Expression is present, it is evaluated. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8).

  If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly, the for statement completes abruptly for the same reason.

Otherwise, there is then a choice based on the presence or absence of the Expression and the resulting value if the Expression is present; see next bullet.

• If the Expression is not present, or it is present and the value resulting from its evaluation (including any possible unboxing) is true, then the contained Statement is executed. Then there is a choice:

  – If execution of the Statement completes normally, then the following two steps are performed in sequence:

    1. First, if the ForUpdate part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the
for statement completes abruptly for the same reason; any ForUpdate statement expressions to the right of the one that completed abruptly are not evaluated.

If the ForUpdate part is not present, no action is taken.

2. Second, another for iteration step is performed.
   – If execution of the Statement completes abruptly, see §14.14.1.3.

• If the Expression is present and the value resulting from its evaluation (including any possible unboxing) is false, no further action is taken and the for statement completes normally.

   If the (possibly unboxed) value of the Expression is false the first time it is evaluated, then the Statement is not executed.

If the Expression is not present, then the only way a for statement can complete normally is by use of a break statement.

14.14.1.3 Abrupt Completion of for Statement

Abrupt completion of the contained Statement is handled in the following manner:

• If execution of the Statement completes abruptly because of a break with no label, no further action is taken and the for statement completes normally.

• If execution of the Statement completes abruptly because of a continue with no label, then the following two steps are performed in sequence:
  1. First, if the ForUpdate part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded.

     If the ForUpdate part is not present, no action is taken.
  2. Second, another for iteration step is performed.

• If execution of the Statement completes abruptly because of a continue with label L, then there is a choice:
   – If the for statement has label L, then the following two steps are performed in sequence:
     1. First, if the ForUpdate part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded.

        If the ForUpdate is not present, no action is taken.
     2. Second, another for iteration step is performed.
– If the for statement does not have label L, the for statement completes abruptly because of a continue with label L.

• If execution of the Statement completes abruptly for any other reason, the for statement completes abruptly for the same reason.

Note that the case of abrupt completion because of a break with a label is handled by the general rule for labeled statements (§14.7).

14.14.2 The enhanced for statement

The enhanced for statement has the form:

```
EnhancedForStatement:
   for ( {VariableModifier} LocalVariableType VariableDeclaratorId : Expression )
   Statement

EnhancedForStatementNoShortIf:
   for ( {VariableModifier} LocalVariableType VariableDeclaratorId : Expression )
   StatementNoShortIf
```

The following productions from §4.3, §8.3, §8.4.1, and §14.4 are shown here for convenience:

```
VariableModifier:
   Annotation
   final

LocalVariableType:
   UnannType
   var

VariableDeclaratorId:
   Identifier [Dims]

Dims:
   [Annotation] [ ] [Annotation] [ ]
```

The header of the enhanced for statement declares a local variable, whose name is the identifier given by VariableDeclaratorId.

If the keyword final appears at the start of the declaration, the variable being declared is a final variable (§4.12.4).
It is a compile-time error if the `LocalVariableType` is `var` and the `VariableDeclaratorId` has one or more bracket pairs.

The type of the `Expression` must be a subtype of the raw type `Iterable` or an array type (§10.1), or a compile-time error occurs.

The type of the local variable is determined as follows:

- If `LocalVariableType` is an `UnannType`, and no bracket pairs appear in `UnannType` or `VariableDeclaratorId`, then `UnannType` denotes the type of the local variable.

- If `LocalVariableType` is an `UnannType`, and bracket pairs appear in `UnannType` or `VariableDeclaratorId`, then the type of the local variable is specified by §10.2.

- If `LocalVariableType` is `var`, then let $T$ be derived from the type of the `Expression`, as follows:
  - If the `Expression` has an array type, then $T$ is the component type of the array type.
  - Otherwise, if the `Expression` has a type that is a subtype of `Iterable<X>`, for some type $X$, then $T$ is $X$.
  - Otherwise, the `Expression` has a type that is a subtype of the raw type `Iterable`, and $T$ is `Object`.

The type of the local variable is the upward projection of $T$ with respect to all synthetic type variables mentioned by $T$ (§4.10.5).

The scope and shadowing of the local variable is specified in §6.3 and §6.4.

When an enhanced `for` statement is executed, the local variable is initialized, on each iteration of the loop, to successive elements of the array or `Iterable` produced by the expression. The precise meaning of the enhanced `for` statement is given by translation into a basic `for` statement, as follows:

- If the type of `Expression` is a subtype of `Iterable`, then the translation is as follows.

  If the type of `Expression` is a subtype of `Iterable<X>` for some type argument $X$, then let $I$ be the type `java.util.Iterator<X>`; otherwise, let $I$ be the raw type `java.util.Iterator`.

  The enhanced `for` statement is equivalent to a basic `for` statement of the form:
for (I #i = Expression.iterator(); #i.hasNext(); ) {
    {VariableModifier} TargetType Identifier =
    (TargetType) #i.next();
    Statement
}

#i is an automatically generated identifier that is distinct from any other identifiers (automatically generated or otherwise) that are in scope (§6.3) at the point where the enhanced for statement occurs.

If the declared type of the local variable in the header of the enhanced for statement is a reference type, then TargetType is that declared type; otherwise, TargetType is the upper bound of the capture conversion (§5.1.10) of the type argument of I, or Object if I is raw.

For example, this code:

List<? extends Integer> l = ...
for (float i : l) ...

will be translated to:

for (Iterator<Integer> #i = l.iterator(); #i.hasNext(); ) {
    float #i0 = (Integer)#i.next();
    ...

• Otherwise, the Expression necessarily has an array type, T[].

Let L₁ ... Lₘ be the (possibly empty) sequence of labels immediately preceding the enhanced for statement.

The enhanced for statement is equivalent to a basic for statement of the form:

T[] #a = Expression;
L₁: L₂: ... Lₘ:
for (int #i = 0; #i < #a.length; #i++) {
    {VariableModifier} TargetType Identifier = #a[#i];
    Statement
}

#a and #i are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the enhanced for statement occurs.

TargetType is the declared type of the local variable in the header of the enhanced for statement.
Example 14.14-1. Enhanced for And Arrays

The following program, which calculates the sum of an integer array, shows how enhanced for works for arrays:

```java
int sum(int[] a) {
    int sum = 0;
    for (int i : a) sum += i;
    return sum;
}
```

Example 14.14-2. Enhanced for And Unboxing Conversion

The following program combines the enhanced for statement with auto-unboxing to translate a histogram into a frequency table:

```java
Map<String, Integer> histogram = ...;
    double total = 0;
    for (int i : histogram.values())
        total += i;
    for (Map.Entry<String, Integer> e : histogram.entrySet())
        System.out.println(e.getKey() + " " + e.getValue() / total);
```

14.15 The break Statement

A break statement transfers control out of an enclosing statement.

```
BreakStatement: 
    break [Identifier] ;
```

A break statement with no label attempts to transfer control to the innermost enclosing switch, while, do, or for statement of the immediately enclosing method or initializer; this statement, which is called the break target, then immediately completes normally.

To be precise, a break statement with no label always completes abruptly, the reason being a break with no label.

If no switch, while, do, or for statement in the immediately enclosing method, constructor, or initializer contains the break statement, a compile-time error occurs.

A break statement with label Identifier attempts to transfer control to the enclosing labeled statement (§14.7) that has the same Identifier as its label; this statement,
which is called the *break target*, then immediately completes normally. In this case, the break target need not be a *switch*, *while*, *do*, or *for* statement.

To be precise, a *break* statement with label *Identifier* always completes abruptly, the reason being a *break* with label *Identifier*.

A *break* statement must refer to a label within the immediately enclosing method, constructor, initializer, or lambda body. There are no non-local jumps. If no labeled statement with *Identifier* as its label in the immediately enclosing method, constructor, initializer, or lambda body contains the *break* statement, a compile-time error occurs.

It can be seen, then, that a *break* statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any *try* statements (§14.20) within the break target whose *try* blocks or *catch* clauses contain the *break* statement, then any *finally* clauses of those *try* statements are executed, in order, innermost to outermost, before control is transferred to the break target. Abrupt completion of a *finally* clause can disrupt the transfer of control initiated by a *break* statement.

**Example 14.15-1. The *break* Statement**

In the following example, a mathematical graph is represented by an array of arrays. A graph consists of a set of nodes and a set of edges; each edge is an arrow that points from some node to some other node, or from a node to itself. In this example it is assumed that there are no redundant edges; that is, for any two nodes $P$ and $Q$, where $Q$ may be the same as $P$, there is at most one edge from $P$ to $Q$.

Nodes are represented by integers, and there is an edge from node $i$ to node $j$ for every $i$ and $j$ for which the array reference $edges[i][j]$ does not throw an *ArrayIndexOutOfBoundsException*.

The task of the method *loseEdges*, given integers $i$ and $j$, is to construct a new graph by copying a given graph but omitting the edge from node $i$ to node $j$, if any, and the edge from node $j$ to node $i$, if any:

```java
class Graph {
    int edges[][];
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
        for (int k = 0; k < n; ++k) {
            edgelist:
            {
                int z;
                search:
```
```java
{    
    if (k == i) {    
        for (z = 0; z < edges[k].length; ++z) {    
            if (edges[k][z] == j) break search;    
        }    
    } else if (k == j) {    
        for (z = 0; z < edges[k].length; ++z) {    
            if (edges[k][z] == i) break search;    
        }    
    }    

    // No edge to be deleted; share this list.    
    newedges[k] = edges[k];    
    break edgelist;    
} //search    

// Copy the list, omitting the edge at position z.    
int m = edges[k].length - 1;    
int ne[] = new int[m];    
System.arraycopy(edges[k], 0, ne, 0, z);    
System.arraycopy(edges[k], z+1, ne, z, m-z);    
newedges[k] = ne;    
} //edgelist    
    
    return new Graph(newedges);    
}
```

Note the use of two statement labels, edgelist and search, and the use of break statements. This allows the code that copies a list, omitting one edge, to be shared between two separate tests, the test for an edge from node \( i \) to node \( j \), and the test for an edge from node \( j \) to node \( i \).

### 14.16 The continue Statement

A continue statement may occur only in a while, do, or for statement; statements of these three kinds are called iteration statements. Control passes to the loop-continuation point of an iteration statement.

**ContinueStatement:**

```
continue [Identifier];
```

A continue statement with no label attempts to transfer control to the innermost enclosing while, do, or for statement of the immediately enclosing method, constructor, or initializer; this statement, which is called the continue target, then immediately ends the current iteration and begins a new one.
To be precise, such a `continue` statement always completes abruptly, the reason being a `continue` with no label.

If no `while`, `do`, or `for` statement of the immediately enclosing method, constructor, or initializer contains the `continue` statement, a compile-time error occurs.

A `continue` statement with label `Identifier` attempts to transfer control to the enclosing labeled statement (§14.7) that has the same `Identifier` as its label; that statement, which is called the `continue target`, then immediately ends the current iteration and begins a new one.

To be precise, a `continue` statement with label `Identifier` always completes abruptly, the reason being a `continue` with label `Identifier`.

The `continue target` must be a `while`, `do`, or `for` statement, or a compile-time error occurs.

A `continue` statement must refer to a label within the immediately enclosing method, constructor, initializer, or lambda body. There are no non-local jumps. If no labeled statement with `Identifier` as its label in the immediately enclosing method, constructor, initializer, or lambda body contains the `continue` statement, a compile-time error occurs.

It can be seen, then, that a `continue` statement always completes abruptly.

See the descriptions of the `while` statement (§14.12), `do` statement (§14.13), and `for` statement (§14.14) for a discussion of the handling of abrupt termination because of `continue`.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.20) within the `continue target` whose `try` blocks or `catch` clauses contain the `continue` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the `continue target`. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `continue` statement.

**Example 14.16-1. The `continue` Statement**

In the `Graph` class in §14.15, one of the `break` statements is used to finish execution of the entire body of the outermost `for` loop. This `break` can be replaced by a `continue` if the `for` loop itself is labeled:

```java
class Graph {
    int edges[][];
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
```
int[][] newedges = new int[n][];
edgelists:
    for (int k = 0; k < n; ++k) {
        int z;
        search:
        {
            if (k == i) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == j) break search;
                }
            } else if (k == j) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == i) break search;
                }
            }
        }
        // No edge to be deleted; share this list.
        newedges[k] = edges[k];
        continue edgelists;
    } //search

    // Copy the list, omitting the edge at position z.
    int m = edges[k].length - 1;
    int ne[] = new int[m];
    System.arraycopy(edges[k], 0, ne, 0, z);
    System.arraycopy(edges[k], z+1, ne, z, m-z);
    newedges[k] = ne;
} //edgelists
return new Graph(newedges);

Which to use, if either, is largely a matter of programming style.

14.17 The return Statement

A return statement returns control to the invoker of a method (§8.4, §15.12) or constructor (§8.8, §15.9).

ReturnStatement:
    return [Expression];

A return statement is contained in the innermost constructor, method, initializer, or lambda expression whose body encloses the return statement.

It is a compile-time error if a return statement is contained in an instance initializer or a static initializer (§8.6, §8.7).
A `return` statement with no `Expression` must be contained in one of the following, or a compile-time error occurs:

- A method that is declared, using the keyword `void`, not to return a value (§8.4.5)
- A constructor (§8.8.7)
- A lambda expression (§15.27)

A `return` statement with no `Expression` attempts to transfer control to the invoker of the method, constructor, or lambda body that contains it. To be precise, a `return` statement with no `Expression` always completes abruptly, the reason being a return with no value.

A `return` statement with an `Expression` must be contained in one of the following, or a compile-time error occurs:

- A method that is declared to return a value
- A lambda expression

The `Expression` must denote a variable or a value, or a compile-time error occurs.

When a `return` statement with an `Expression` appears in a method declaration, the `Expression` must be assignable (§5.2) to the declared return type of the method, or a compile-time error occurs.

A `return` statement with an `Expression` attempts to transfer control to the invoker of the method or lambda body that contains it; the value of the `Expression` becomes the value of the method invocation. More precisely, execution of such a `return` statement first evaluates the `Expression`. If the evaluation of the `Expression` completes abruptly for some reason, then the `return` statement completes abruptly for that reason. If evaluation of the `Expression` completes normally, producing a value \( V \), then the `return` statement completes abruptly, the reason being a return with value \( V \).

If the expression is of type `float` and is not FP-strict (§15.4), then the value may be an element of either the float value set or the float-extended-exponent value set (§4.2.3). If the expression is of type `double` and is not FP-strict, then the value may be an element of either the double value set or the double-extended-exponent value set.

It can be seen, then, that a `return` statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.20) within the method or constructor whose `try` blocks or `catch` clauses contain the `return` statement, then any `finally` clauses of those `try` statements will be executed, in order, innermost to outermost, before
control is transferred to the invoker of the method or constructor. Abrupt completion of a finally clause can disrupt the transfer of control initiated by a return statement.

14.18 The throw Statement

A throw statement causes an exception (§11 (Exceptions)) to be thrown. The result is an immediate transfer of control (§11.3) that may exit multiple statements and multiple constructor, instance initializer, static initializer and field initializer evaluations, and method invocations until a try statement (§14.20) is found that catches the thrown value. If no such try statement is found, then execution of the thread (§17 (Threads and Locks)) that executed the throw is terminated (§11.3) after invocation of the uncaughtException method for the thread group to which the thread belongs.

ThrowStatement:

throw Expression ;

The Expression in a throw statement must either denote a variable or value of a reference type which is assignable (§5.2) to the type Throwable, or denote the null reference, or a compile-time error occurs.

The reference type of the Expression will always be a class type (since no interface types are assignable to Throwable) which is not parameterized (since a subclass of Throwable cannot be generic (§8.1.2)).

At least one of the following three conditions must be true, or a compile-time error occurs:

• The type of theExpression is an unchecked exception class (§11.1.1) or the null type (§4.1).

• The throw statement is contained in the try block of a try statement (§14.20) and it is not the case that the try statement can throw an exception of the type of the Expression. (In this case we say the thrown value is caught by the try statement.)

• The throw statement is contained in a method or constructor declaration and the type of the Expression is assignable (§5.2) to at least one type listed in the throws clause (§8.4.6, §8.8.5) of the declaration.

The exception types that a throw statement can throw are specified in §11.2.2.

A throw statement first evaluates the Expression. Then:
• If evaluation of the Expression completes abruptly for some reason, then the
  throw completes abruptly for that reason.

• If evaluation of the Expression completes normally, producing a non-null value
  V, then the throw statement completes abruptly, the reason being a throw with
  value V.

• If evaluation of the Expression completes normally, producing a null value, then
  an instance V' of class NullPointerException is created and thrown instead of
  null. The throw statement then completes abruptly, the reason being a throw
  with value V'.

   It can be seen, then, that a throw statement always completes abruptly.

If there are any enclosing try statements (§14.20) whose try blocks contain the
throw statement, then any finally clauses of those try statements are executed
as control is transferred outward, until the thrown value is caught. Note that abrupt
completion of a finally clause can disrupt the transfer of control initiated by a
throw statement.

If a throw statement is contained in a method declaration or a lambda expression,
but its value is not caught by some try statement that contains it, then the invocation
of the method completes abruptly because of the throw.

If a throw statement is contained in a constructor declaration, but its value is not
caught by some try statement that contains it, then the class instance creation
expression that invoked the constructor will complete abruptly because of the
throw (§15.9.4).

If a throw statement is contained in a static initializer (§8.7), then a compile-time
check (§11.2.3) ensures that either its value is always an unchecked exception or
its value is always caught by some try statement that contains it. If at run time,
despite this check, the value is not caught by some try statement that contains the
throw statement, then the value is rethrown if it is an instance of class Error or one
of its subclasses; otherwise, it is wrapped in an ExceptionInInitializerError
object, which is then thrown (§12.4.2).

If a throw statement is contained in an instance initializer (§8.6), then a compile-
time check (§11.2.3) ensures that either its value is always an unchecked exception
or its value is always caught by some try statement that contains it, or the type
of the thrown exception (or one of its superclasses) occurs in the throws clause of
every constructor of the class.

   By convention, user-declared throwable types should usually be declared to be subclasses
   of class Exception, which is a subclass of class Throwable (§11.1.1).
14.19 The synchronized Statement

A synchronized statement acquires a mutual-exclusion lock (§17.1) on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.

\textit{SynchronizedStatement:}
\begin{verbatim}
synchronized ( Expression ) Block
\end{verbatim}

The type of \textit{Expression} must be a reference type, or a compile-time error occurs.

A synchronized statement is executed by first evaluating the \textit{Expression}. Then:

- If evaluation of the \textit{Expression} completes abruptly for some reason, then the synchronized statement completes abruptly for the same reason.

- Otherwise, if the value of the \textit{Expression} is \texttt{null}, a \texttt{NullPointerException} is thrown.

- Otherwise, let the non-\texttt{null} value of the \textit{Expression} be \texttt{v}. The executing thread locks the monitor associated with \texttt{v}. Then the \textit{Block} is executed, and then there is a choice:
  - If execution of the \textit{Block} completes normally, then the monitor is unlocked and the synchronized statement completes normally.
  - If execution of the \textit{Block} completes abruptly for any reason, then the monitor is unlocked and the synchronized statement completes abruptly for the same reason.

The locks acquired by synchronized statements are the same as the locks that are acquired implicitly by synchronized methods (§8.4.3.6). A single thread may acquire a lock more than once.

Acquiring the lock associated with an object does not in itself prevent other threads from accessing fields of the object or invoking un-synchronized methods on the object. Other threads can also use synchronized methods or the synchronized statement in a conventional manner to achieve mutual exclusion.

\textbf{Example 14.19-1. The synchronized Statement}

\begin{verbatim}
class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
    }
}
\end{verbatim}
This program produces the output:

```
made it!
```

Note that this program would deadlock if a single thread were not permitted to lock a monitor more than once.

### 14.20 The `try` statement

A `try` statement executes a block. If a value is thrown and the `try` statement has one or more `catch` clauses that can catch it, then control will be transferred to the first such `catch` clause. If the `try` statement has a `finally` clause, then another block of code is executed, no matter whether the `try` block completes normally or abruptly, and no matter whether a `catch` clause is first given control.

```
TryStatement:
  try Block Catches  try Block [Catches] Finally
  TryWithResourcesStatement

Catches:
  CatchClause {CatchClause}

CatchClause:
  catch (CatchFormalParameter) Block

CatchFormalParameter:
  {VariableModifier} CatchType VariableDeclaratorId

CatchType:
  UnannClassType [ | ClassType]

Finally:
  finally Block
```

See §8.3 for `UnannClassType`. The following productions from §4.3, §8.3, and §8.4.1 are shown here for convenience:
The *try* block immediately after the keyword *try* is called the *try block* of the *try* statement.

The *Block* immediately after the keyword *finally* is called the *finally block* of the *try* statement.

A *try* statement may have *catch* clauses, also called *exception handlers*.

A *catch* clause declares exactly one parameter, which is called an *exception parameter*.

It is a compile-time error if *final* appears more than once as a modifier for an exception parameter declaration.

The scope and shadowing of an exception parameter is specified in §6.3 and §6.4.

An exception parameter may denote its type as either a single class type or a union of two or more class types (called *alternatives*). The alternatives of a union are syntactically separated by `|`.

A *catch* clause whose exception parameter is denoted as a single class type is called a *uni*-*catch* clause.

A *catch* clause whose exception parameter is denoted as a union of types is called a *multi-catch* clause.

Each class type used in the denotation of the type of an exception parameter must be the class *Throwable* or a subclass of *Throwable*, or a compile-time error occurs.

It is a compile-time error if a type variable is used in the denotation of the type of an exception parameter.

It is a compile-time error if a union of types contains two alternatives $D_i$ and $D_j$ ($i \neq j$) where $D_i$ is a subtype of $D_j$ (§4.10.2).

The declared type of an exception parameter that denotes its type with a single class type is that class type.

The declared type of an exception parameter that denotes its type as a union with alternatives $D_1 \mid D_2 \mid ... \mid D_n$ is lub($D_1, D_2, ..., D_n$).
An exception parameter of a multi-\texttt{catch} clause is implicitly declared \texttt{final} if it is not explicitly declared \texttt{final}.

It is a compile-time error if an exception parameter that is implicitly or explicitly declared \texttt{final} is assigned to within the body of the \texttt{catch} clause.

An exception parameter of a uni-\texttt{catch} clause is never implicitly declared \texttt{final}, but it may be explicitly declared \texttt{final} or be effectively final (§4.12.4).

An implicitly \texttt{final} exception parameter is final by virtue of its declaration, while an effectively final exception parameter is (as it were) final by virtue of how it is used. An exception parameter of a multi-\texttt{catch} clause is implicitly declared \texttt{final}, so will never occur as the left-hand operand of an assignment operator, but it is \textit{not} considered effectively final.

If an exception parameter is effectively final (in a uni-\texttt{catch} clause) or implicitly final (in a multi-\texttt{catch} clause), then adding an explicit \texttt{final} modifier to its declaration will not introduce any compile-time errors. On the other hand, if the exception parameter of a uni-\texttt{catch} clause is explicitly declared \texttt{final}, then removing the \texttt{final} modifier may introduce compile-time errors because the exception parameter, now considered to be effectively final, can no longer be referenced by anonymous and local class declarations in the body of the \texttt{catch} clause. If there are no compile-time errors, it is possible to further change the program so that the exception parameter is re-assigned in the body of the \texttt{catch} clause and thus will no longer be considered effectively final.

The exception types that a \texttt{try} statement can throw are specified in §11.2.2.

The relationship of the exceptions thrown by the \texttt{try} block of a \texttt{try} statement and caught by the \texttt{catch} clauses (if any) of the \texttt{try} statement is specified in §11.2.3.

Exception handlers are considered in left-to-right order: the earliest possible \texttt{catch} clause accepts the exception, receiving as its argument the thrown exception object, as specified in §11.3.

A multi-\texttt{catch} clause can be thought of as a sequence of uni-\texttt{catch} clauses. That is, a \texttt{catch} clause where the type of the exception parameter is denoted as a union $D_1 | D_2 | ... | D_n$ is equivalent to a sequence of $n$ \texttt{catch} clauses where the types of the exception parameters are class types $D_1$, $D_2$, ..., $D_n$ respectively. In the \texttt{Block} of each of the $n$ \texttt{catch} clauses, the declared type of the exception parameter is \texttt{lub}(D_1, D_2, ..., D_n). For example, the following code:

\begin{verbatim}
try {
    ... throws ReflectiveOperationException ... 
}
catch (ClassNotFoundException | IllegalAccessException ex) {
    ... body ...
}
\end{verbatim}

is semantically equivalent to the following code:
try {
    ... throws ReflectiveOperationException ...
}
catch (final ClassNotFoundException ex1) {
    final ReflectiveOperationException ex = ex1;
    ... body ...
}
catch (final IllegalAccessException ex2) {
    final ReflectiveOperationException ex = ex2;
    ... body ...
}

where the multi-catch clause with two alternatives has been translated into two uni-catch clauses, one for each alternative. A Java compiler is neither required nor recommended to compile a multi-catch clause by duplicating code in this manner, since it is possible to represent the multi-catch clause in a class file without duplication.

A finally clause ensures that the finally block is executed after the try block and any catch block that might be executed, no matter how control leaves the try block or catch block. Handling of the finally block is rather complex, so the two cases of a try statement with and without a finally block are described separately (§14.20.1, §14.20.2).

A try statement is permitted to omit catch clauses and a finally clause if it is a try-with-resources statement (§14.20.3).

### 14.20.1 Execution of try-catch

A try statement without a finally block is executed by first executing the try block. Then there is a choice:

- If execution of the try block completes normally, then no further action is taken and the try statement completes normally.
- If execution of the try block completes abruptly because of a throw of a value \( v \), then there is a choice:
  - If the run-time type of \( v \) is assignment compatible with (§5.2) a catchable exception class of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value \( v \) is assigned to the parameter of the selected catch clause, and the Block of that catch clause is executed, and then there is a choice:
    - If that block completes normally, then the try statement completes normally.
If that block completes abruptly for any reason, then the try statement completes abruptly for the same reason.

- If the run-time type of \( v \) is not assignment compatible with a catchable exception class of any catch clause of the try statement, then the try statement completes abruptly because of a throw of the value \( v \).

- If execution of the try block completes abruptly for any other reason, then the try statement completes abruptly for the same reason.

**Example 14.20.1-1. Catching An Exception**

```java
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
    static void blowUp() throws BlewIt { throw new BlewIt(); }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (RuntimeException r) {
            System.out.println("Caught RuntimeException");
        } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
        }
    }
}
```

Here, the exception BlewIt is thrown by the method blowUp. The try-catch statement in the body of main has two catch clauses. The run-time type of the exception is BlewIt which is not assignable to a variable of type RuntimeException, but is assignable to a variable of type BlewIt, so the output of the example is:

Caught BlewIt

### 14.20.2 Execution of try-finally and try-catch-finally

A try statement with a finally block is executed by first executing the try block. Then there is a choice:

- If execution of the try block completes normally, then the finally block is executed, and then there is a choice:
  - If the finally block completes normally, then the try statement completes normally.
– If the finally block completes abruptly for reason \( S \), then the \texttt{try} statement completes abruptly for reason \( S \).

- If execution of the \texttt{try} block completes abruptly because of a \texttt{throw} of a value \( V \), then there is a choice:

– If the run-time type of \( V \) is assignment compatible with a catchable exception class of any \texttt{catch} clause of the \texttt{try} statement, then the first (leftmost) such \texttt{catch} clause is selected. The value \( V \) is assigned to the parameter of the selected \texttt{catch} clause, and the \texttt{Block} of that \texttt{catch} clause is executed. Then there is a choice:

  › If the \texttt{catch} block completes normally, then the \texttt{finally} block is executed. Then there is a choice:

    » If the \texttt{finally} block completes normally, then the \texttt{try} statement completes normally.

    » If the \texttt{finally} block completes abruptly for any reason, then the \texttt{try} statement completes abruptly for the same reason.

  › If the \texttt{catch} block completes abruptly for reason \( R \), then the \texttt{finally} block is executed. Then there is a choice:

    » If the \texttt{finally} block completes normally, then the \texttt{try} statement completes abruptly for reason \( R \).

    » If the \texttt{finally} block completes abruptly for reason \( S \), then the \texttt{try} statement completes abruptly for reason \( S \) (and reason \( R \) is discarded).

– If the run-time type of \( V \) is not assignment compatible with a catchable exception class of any \texttt{catch} clause of the \texttt{try} statement, then the \texttt{finally} block is executed. Then there is a choice:

  › If the \texttt{finally} block completes normally, then the \texttt{try} statement completes abruptly because of a \texttt{throw} of the value \( V \).

  › If the \texttt{finally} block completes abruptly for reason \( S \), then the \texttt{try} statement completes abruptly for reason \( S \) (and the \texttt{throw} of value \( V \) is discarded and forgotten).

- If execution of the \texttt{try} block completes abruptly for any other reason \( R \), then the \texttt{finally} block is executed, and then there is a choice:

  – If the \texttt{finally} block completes normally, then the \texttt{try} statement completes abruptly for reason \( R \).
- If the finally block completes abruptly for reason $S$, then the try statement completes abruptly for reason $S$ (and reason $R$ is discarded).

**Example 14.20.2-1. Handling An Uncaught Exception With finally**

```java
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}

class Test {
    static void blowUp() throws BlewIt {
        throw new NullPointerException();
    }

    public static void main(String[] args) {
        try {
            blowUp();
        } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
        } finally {
            System.out.println("Uncaught Exception");
        }
    }
}
```

This program produces the output:

```
Uncaught Exception
Exception in thread "main" java.lang.NullPointerException
    at Test.blowUp(Test.java:7)
    at Test.main(Test.java:11)
```

The NullPointerException (which is a kind of RuntimeException) that is thrown by method blowUp is not caught by the try statement in main, because a NullPointerException is not assignable to a variable of type BlewIt. This causes the finally clause to execute, after which the thread executing main, which is the only thread of the test program, terminates because of an uncaught exception, which typically results in printing the exception name and a simple backtrace. However, a backtrace is not required by this specification.

The problem with mandating a backtrace is that an exception can be created at one point in the program and thrown at a later one. It is prohibitively expensive to store a stack trace in an exception unless it is actually thrown (in which case the trace may be generated while unwinding the stack). Hence we do not mandate a back trace in every exception.

**14.20.3 try-with-resources**

A try-with-resources statement is parameterized with local variables (known as resources) that are initialized before execution of the try block and closed automatically, in the reverse order from which they were initialized, after execution
of the try block. catch clauses and a finally clause are often unnecessary when resources are closed automatically.

TryWithResourcesStatement:
   try ResourceSpecification Block [Catches] [Finally]

ResourceSpecification:
   ( ResourceList [;] )

ResourceList:
   Resource { ; Resource}

Resource:
   {VariableModifier} LocalVariableType Identifier = Expression
   VariableAccess

VariableAccess:
   ExpressionName
   FieldAccess

The following productions from §8.4.1 and §14.4 are shown here for convenience:

VariableModifier:
   Annotation
   final

LocalVariableType:
   UnannType
   var

A resource specification uses variables to denote resources for the try statement, either by declaring local variables with initializer expressions or by referring to suitable existing variables. An existing variable is referred to by either an expression name (§6.5.6) or a field access expression (§15.11).

It is a compile-time error for a resource specification to declare two variables with the same name.

It is a compile-time error if final appears more than once as a modifier for each variable declared in a resource specification.

A variable declared in a resource specification is implicitly declared final if it is not explicitly declared final (§4.12.4).
A resource denoted by an expression name or field access expression must be a final or effectively final variable that is definitely assigned before the try-with-resources statement (§16 (Definite Assignment)), or a compile-time error occurs.

It is a compile-time error if the LocalVariableType of a variable declared in a resource specification is var and the initializer expression contains a reference to the variable.

The type of a variable declared in a resource specification is determined as follows:

- If LocalVariableType is an UnannType, then UnannType denotes the type of the local variable.

- If LocalVariableType is var, then let $\tau$ be the type of the initializer expression when treated as if it did not appear in an assignment context, and were thus a standalone expression (§15.2). The type of the local variable is the upward projection of $\tau$ with respect to all synthetic type variables mentioned by $\tau$ (§4.10.5).

  It is a compile-time error if $\tau$ is the null type.

The type of a variable declared or referred to as a resource in a resource specification must be a subtype of AutoCloseable, or a compile-time error occurs.

The scope and shadowing of a variable declared in a resource specification is specified in §6.3 and §6.4.

Resources are initialized in left-to-right order. If a resource fails to initialize (that is, its initializer expression throws an exception), then all resources initialized so far by the try-with-resources statement are closed. If all resources initialize successfully, the try block executes as normal and then all non-null resources of the try-with-resources statement are closed.

Resources are closed in the reverse order from that in which they were initialized. A resource is closed only if it initialized to a non-null value. An exception from the closing of one resource does not prevent the closing of other resources. Such an exception is suppressed if an exception was thrown previously by an initializer, the try block, or the closing of a resource.

A try-with-resources statement whose resource specification indicates multiple resources is treated as if it were multiple try-with-resources statements, each of which has a resource specification that indicates a single resource. When a try-with-resources statement with $n$ resources ($n > 1$) is translated, the result is a try-with-resources statement with $n-1$ resources. After $n$ such translations, there are $n$ nested try-catch-finally statements, and the overall translation is complete.
14.20.3.1 Basic try-with-resources

A try-with-resources statement with no catch clauses or finally clause is called a basic try-with-resources statement.

If a basic try-with-resource statement is of the form:

```
try (VariableAccess ...)  
    Block
```

then the resource is first converted to a local variable declaration by the following translation:

```
try (T #r = VariableAccess ...)  
    Block
```

T is the type of the variable denoted by VariableAccess and #r is an automatically generated identifier that is distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the try-with-resources statement occurs. The try-with-resources statement is then translated according to the rest of this section.

The meaning of a basic try-with-resources statement of the form:

```
try ((VariableModifier) R Identifier = Expression ...)  
    Block
```

is given by the following translation to a local variable declaration and a try-catch-finally statement:
The `try` statement

```java
{ final (VariableModifierNoFinal) R Identifier = Expression;
  Throwable #primaryExc = null;
  try ResourceSpecification_tail Block
    catch (Throwable #t) {
      #primaryExc = #t;
      throw #t;
    } finally {
      if (Identifier != null) {
        if (#primaryExc != null) {
          try {
            Identifier.close();
          } catch (Throwable #suppressedExc) {
            #primaryExc.addSuppressed(#suppressedExc);
          } else {
            Identifier.close();
          }
        } else {
          Identifier.close();
        }
      }
    }
}
```

`{VariableModifierNoFinal}` is defined as `{VariableModifier}` without `final`, if present.

`#t`, `#primaryExc`, and `#suppressedExc` are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the `try`-with-resources statement occurs.

If the resource specification indicates one resource, then `ResourceSpecification_tail` is empty (and the `try-catch-finally` statement is not itself a `try`-with-resources statement).

If the resource specification indicates $n > 1$ resources, then `ResourceSpecification_tail` consists of the 2nd, 3rd, ..., $n$'th resources indicated in the resource specification, in the same order (and the `try-catch-finally` statement is itself a `try`-with-resources statement).

Reachability and definite assignment rules for the basic `try`-with-resources statement are implicitly specified by the translation above.

In a basic `try`-with-resources statement that manages a single resource:

- If the initialization of the resource completes abruptly because of a `throw` of a value $V$, then the `try`-with-resources statement completes abruptly because of a `throw` of the value $V$. 

• If the initialization of the resource completes normally, and the `try` block completes abruptly because of a `throw` of a value `V`, then:
  – If the automatic closing of the resource completes normally, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V`.
  – If the automatic closing of the resource completes abruptly because of a `throw` of a value `V2`, then the `try-with-resources` statement completes abruptly because of a `throw` of value `V` with `V2` added to the suppressed exception list of `V`.

• If the initialization of the resource completes normally, and the `try` block completes normally, and the automatic closing of the resource completes abruptly because of a `throw` of a value `V`, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V`.

In a basic `try-with-resources` statement that manages multiple resources:

• If the initialization of a resource completes abruptly because of a `throw` of a value `V`, then:
  – If the automatic closings of all successfully initialized resources (possibly zero) complete normally, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V`.
  – If the automatic closings of all successfully initialized resources (possibly zero) complete abruptly because of `throws` of values `V1...Vn`, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V` with any remaining values `V1...Vn` added to the suppressed exception list of `V`.

• If the initialization of all resources completes normally, and the `try` block completes abruptly because of a `throw` of a value `V`, then:
  – If the automatic closings of all initialized resources complete normally, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V`.
  – If the automatic closings of one or more initialized resources complete abruptly because of `throws` of values `V1...Vn`, then the `try-with-resources` statement completes abruptly because of a `throw` of the value `V` with any remaining values `V1...Vn` added to the suppressed exception list of `V`.

• If the initialization of every resource completes normally, and the `try` block completes normally, then:
  – If one automatic closing of an initialized resource completes abruptly because of a `throw` of value `V`, and all other automatic closings of initialized resources
complete normally, then the try-with-resources statement completes abruptly because of a throw of the value \( V \).

- If more than one automatic closing of an initialized resource completes abruptly because of throws of values \( V_1...V_n \), then the try-with-resources statement completes abruptly because of a throw of the value \( V_1 \) with any remaining values \( V_2...V_n \) added to the suppressed exception list of \( V_1 \) (where \( V_1 \) is the exception from the rightmost resource failing to close and \( V_n \) is the exception from the leftmost resource failing to close).

14.20.3.2 Extended try-with-resources

A try-with-resources statement with at least one catch clause and/or a finally clause is called an extended try-with-resources statement.

The meaning of an extended try-with-resources statement:

```java
try ResourceSpecification
    Block
[Catches]
[Finally]
```

is given by the following translation to a basic try-with-resources statement nested inside a try-catch or try-finally or try-catch-finally statement:

```java
try {
    try ResourceSpecification
        Block
}
[Catches]
[Finally]
```

The effect of the translation is to put the resource specification "inside" the try statement. This allows a catch clause of an extended try-with-resources statement to catch an exception due to the automatic initialization or closing of any resource.

Furthermore, all resources will have been closed (or attempted to be closed) by the time the finally block is executed, in keeping with the intent of the finally keyword.

14.21 Unreachable Statements

It is a compile-time error if a statement cannot be executed because it is unreachable.
This section is devoted to a precise explanation of the word "reachable." The idea is that there must be some possible execution path from the beginning of the constructor, method, instance initializer, or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of while, do, and for statements whose condition expression has the constant value true, the values of expressions are not taken into account in the flow analysis.

For example, a Java compiler will accept the code:

```java
{  
    int n = 5;  
    while (n > 7) k = 2;  
}
```

even though the value of n is known at compile time and in principle it can be known at compile time that the assignment to k can never be executed.

The rules in this section define two technical terms:

- whether a statement is reachable
- whether a statement can complete normally

The definitions here allow a statement to complete normally only if it is reachable. To shorten the description of the rules, the customary abbreviation "iff" is used to mean "if and only if."

A reachable break statement exits a statement if, within the break target, either there are no try statements whose try blocks contain the break statement, or there are try statements whose try blocks contain the break statement and all finally clauses of those try statements can complete normally.

This definition is based on the logic around "attempts to transfer control" in §14.15.

A continue statement continues a do statement if, within the do statement, either there are no try statements whose try blocks contain the continue statement, or there are try statements whose try blocks contain the continue statement and all finally clauses of those try statements can complete normally.

The rules are as follows:

- The block that is the body of a constructor, method, instance initializer, or static initializer is reachable.
- An empty block that is not a switch block can complete normally iff it is reachable.
A non-empty block that is not a switch block can complete normally iff the last statement in it can complete normally.

The first statement in a non-empty block that is not a switch block is reachable iff the block is reachable.

Every other statement $s$ in a non-empty block that is not a switch block is reachable iff the statement preceding $s$ can complete normally.

- A local class declaration statement can complete normally iff it is reachable.
- A local variable declaration statement can complete normally iff it is reachable.
- An empty statement can complete normally iff it is reachable.
- A labeled statement can complete normally if at least one of the following is true:
  - The contained statement can complete normally.
  - There is a reachable \texttt{break} statement that exits the labeled statement.

The contained statement is reachable iff the labeled statement is reachable.

- An expression statement can complete normally iff it is reachable.
- An \texttt{if-then} statement can complete normally iff it is reachable.
  
  The \texttt{then}-statement is reachable iff the \texttt{if-then} statement is reachable.

  An \texttt{if-then-else} statement can complete normally iff the \texttt{then}-statement can complete normally or the \texttt{else}-statement can complete normally.

  The \texttt{then}-statement is reachable iff the \texttt{if-then-else} statement is reachable.

  The \texttt{else}-statement is reachable iff the \texttt{if-then-else} statement is reachable.

  This handling of an \texttt{if} statement, whether or not it has an \texttt{else} part, is rather unusual. The rationale is given at the end of this section.

- An \texttt{assert} statement can complete normally iff it is reachable.
- A \texttt{switch} statement can complete normally iff at least one of the following is true:
  - The switch block is empty or contains only switch labels.
  - The last statement in the switch block can complete normally.
  - There is at least one switch label after the last switch block statement group.
  - The switch block does not contain a \texttt{default} label.
  - There is a reachable \texttt{break} statement that exits the \texttt{switch} statement.
• A switch block is reachable iff its switch statement is reachable.

• A statement in a switch block is reachable iff its switch statement is reachable and at least one of the following is true:
  – It bears a case or default label.
  – There is a statement preceding it in the switch block and that preceding statement can complete normally.

• A while statement can complete normally iff at least one of the following is true:
  – The while statement is reachable and the condition expression is not a constant expression (§15.28) with value true.
  – There is a reachable break statement that exits the while statement.

The contained statement is reachable iff the while statement is reachable and the condition expression is not a constant expression whose value is false.

• A do statement can complete normally iff at least one of the following is true:
  – The contained statement can complete normally and the condition expression is not a constant expression (§15.28) with value true.
  – The do statement contains a reachable continue statement with no label, and the do statement is the innermost while, do, or for statement that contains that continue statement, and the continue statement continues that do statement, and the condition expression is not a constant expression with value true.
  – The do statement contains a reachable continue statement with a label L, and the do statement has label L, and the continue statement continues that do statement, and the condition expression is not a constant expression with value true.
  – There is a reachable break statement that exits the do statement.

The contained statement is reachable iff the do statement is reachable.

• A basic for statement can complete normally iff at least one of the following is true:
  – The for statement is reachable, there is a condition expression, and the condition expression is not a constant expression (§15.28) with value true.
  – There is a reachable break statement that exits the for statement.

The contained statement is reachable iff the for statement is reachable and the condition expression is not a constant expression whose value is false.
• An enhanced for statement can complete normally iff it is reachable.
• A break, continue, return, or throw statement cannot complete normally.
• A synchronized statement can complete normally iff the contained statement can complete normally.
  The contained statement is reachable iff the synchronized statement is reachable.
• A try statement can complete normally iff both of the following are true:
  – The try block can complete normally or any catch block can complete normally.
  – If the try statement has a finally block, then the finally block can complete normally.
• The try block is reachable iff the try statement is reachable.
• A catch block C is reachable iff both of the following are true:
  – Either the type of C's parameter is an unchecked exception type or Exception or a superclass of Exception, or some expression or throw statement in the try block is reachable and can throw a checked exception whose type is assignable to the type of C's parameter. (An expression is reachable iff the innermost statement containing it is reachable.)
  – There is no earlier catch block A in the try statement such that the type of C's parameter is the same as or a subclass of the type of A's parameter.
• The Block of a catch block is reachable iff the catch block is reachable.
• If a finally block is present, it is reachable iff the try statement is reachable.

One might expect the if statement to be handled in the following manner:

• An if-then statement can complete normally iff at least one of the following is true:
  – The if-then statement is reachable and the condition expression is not a constant expression whose value is true.
  – The then-statement can complete normally.

The then-statement is reachable iff the if-then statement is reachable and the condition expression is not a constant expression whose value is false.

• An if-then-else statement can complete normally iff the then-statement can complete normally or the else-statement can complete normally.
The then-statement is reachable iff the if-then-else statement is reachable and the condition expression is not a constant expression whose value is false.

The else-statement is reachable iff the if-then-else statement is reachable and the condition expression is not a constant expression whose value is true.

This approach would be consistent with the treatment of other control structures. However, in order to allow the if statement to be used conveniently for "conditional compilation" purposes, the actual rules differ.

As an example, the following statement results in a compile-time error:

```java
while (false) { x=3; }
```

because the statement x=3; is not reachable; but the superficially similar case:

```java
if (false) { x=3; }
```

does not result in a compile-time error. An optimizing compiler may realize that the statement x=3; will never be executed and may choose to omit the code for that statement from the generated class file, but the statement x=3; is not regarded as "unreachable" in the technical sense specified here.

The rationale for this differing treatment is to allow programmers to define "flag" variables such as:

```java
static final boolean DEBUG = false;
```

and then write code such as:

```java
if (DEBUG) { x=3; }
```

The idea is that it should be possible to change the value of DEBUG from false to true or from true to false and then compile the code correctly with no other changes to the program text.

Conditional compilation comes with a caveat. If a set of classes that use a "flag" variable - or more precisely, any static constant variable (§4.12.4) - are compiled and conditional code is omitted, it does not suffice later to distribute just a new version of the class or interface that contains the definition of the flag. The classes that use the flag will not see its new value, so their behavior may be surprising. In essence, a change to the value of a flag is binary compatible with pre-existing binaries (no LinkageError occurs) but not behaviorally compatible.

Another reason for "inlining" values of static constant variables is because of switch statements. They are the only kind of statement that relies on constant expressions, namely that each case label of a switch statement must be a constant expression whose value is different than every other case label. case labels are often references to static constant variables so it may not be immediately obvious that all the labels have different values. If it
is proven that there are no duplicate labels at compile time, then inlining the values into the class file ensures there are no duplicate labels at run time either - a very desirable property.

Example 14.21-1. Conditional Compilation

If the example:

```java
class Flags { static final boolean DEBUG = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.DEBUG)
            System.out.println("DEBUG is true");
    }
}
```

is compiled and executed, it produces the output:

```
DEBUG is true
```

Suppose that a new version of class Flags is produced:

```java
class Flags { static final boolean DEBUG = false; }
```

If Flags is recompiled but not Test, then running the new binary with the existing binary of Test produces the output:

```
DEBUG is true
```

because DEBUG is a static constant variable, so its value could have been used in compiling Test without making a reference to the class Flags.

This behavior would also occur if Flags was an interface, as in the modified example:

```java
interface Flags { boolean DEBUG = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.DEBUG)
            System.out.println("DEBUG is true");
    }
}
```

In fact, because the fields of interfaces are always static and final, we recommend that only constant expressions be assigned to fields of interfaces. We note, but do not recommend, that if a field of primitive type of an interface may change, its value may be expressed idiomatically as in:

```java
interface Flags {
    boolean debug = Boolean.valueOf(true).booleanValue();
}
```
ensuring that this value is not a constant expression. Similar idioms exist for the other
primitive types.
CHAPTER 15

Expressions

Much of the work in a program is done by evaluating expressions, either for their side effects, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

This chapter specifies the meanings of expressions and the rules for their evaluation.

15.1 Evaluation, Denotation, and Result

When an expression in a program is evaluated (executed), the result denotes one of three things:

- A variable (§4.12) (in C, this would be called an lvalue)
- A value (§4.2, §4.3)
- Nothing (the expression is said to be void)

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the value of the expression.

Value set conversion (§5.1.13) is applied to the result of every expression that produces a value, including when the value of a variable of type float or double is used.

An expression denotes nothing if and only if it is a method invocation (§15.12) that invokes a method that does not return a value, that is, a method declared void (§8.4). Such an expression can be used only as an expression statement (§14.8) or as the single expression of a lambda body (§15.27.2), because every other context in which an expression can appear requires the expression to denote something. An
expression statement or lambda body that is a method invocation may also invoke a method that produces a result; in this case the value returned by the method is quietly discarded.

Evaluation of an expression can produce side effects, because expressions may contain embedded assignments, increment operators, decrement operators, and method invocations.

An expression occurs in either:

- The declaration of some (class or interface) type that is being declared: in a field initializer, in a static initializer, in an instance initializer, in a constructor declaration, in a method declaration, or in an annotation.
- An annotation on a package declaration or on a top level type declaration.

### 15.2 Forms of Expressions

Expressions can be broadly categorized into one of the following syntactic forms:

- Expression names (§6.5.6)
- Primary expressions (§15.8 - §15.13)
- Unary operator expressions (§15.14 - §15.16)
- Binary operator expressions (§15.17 - §15.24, and §15.26)
- Ternary operator expressions (§15.25)
- Lambda expressions (§15.27)

Precedence among operators is managed by a hierarchy of grammar productions. The lowest precedence operator is the arrow of a lambda expression (\(\to\)), followed by the assignment operators. Thus, all expressions are syntactically included in the `LambdaExpression` and `AssignmentExpression` nonterminals:

\[
\text{Expression:}
\begin{align*}
\text{LambdaExpression} \\
\text{AssignmentExpression}
\end{align*}
\]

When some expressions appear in certain contexts, they are considered *poly expressions*. The following forms of expressions may be poly expressions:

- Parenthesized expressions (§15.8.5)
• Class instance creation expressions (§15.9)
• Method invocation expressions (§15.12)
• Method reference expressions (§15.13)
• Conditional expressions (§15.25)
• Lambda expressions (§15.27)

The rules determining whether an expression of one of these forms is a poly expression are given in the individual sections that specify these forms of expressions.

Expressions that are not poly expressions are standalone expressions. Standalone expressions are expressions of the forms above when determined not to be poly expressions, as well as all expressions of all other forms. Expressions of all other forms are said to have a standalone form.

Some expressions have a value that can be determined at compile time. These are constant expressions (§15.28).

### 15.3 Type of an Expression

If an expression denotes a variable or a value, then the expression has a type known at compile time. The type of a standalone expression can be determined entirely from the contents of the expression; in contrast, the type of a poly expression may be influenced by the expression's target type (§5 (Conversions and Contexts)). The rules for determining the type of an expression are explained separately below for each kind of expression.

The value of an expression is assignment compatible (§5.2) with the type of the expression, unless heap pollution occurs (§4.12.2).

Likewise, the value stored in a variable is always compatible with the type of the variable, unless heap pollution occurs.

In other words, the value of an expression whose type is $\tau$ is always suitable for assignment to a variable of type $\tau$.

Note that an expression whose type is a class type $F$ that is declared final is guaranteed to have a value that is either a null reference or an object whose class is $F$ itself, because final types have no subclasses.
15.4 FP-strict Expressions

If the type of an expression is float or double, then there is a question as to what value set (§4.2.3) the value of the expression is drawn from. This is governed by the rules of value set conversion (§5.1.13); these rules in turn depend on whether or not the expression is FP-strict.

Every constant expression (§15.28) is FP-strict.

If an expression is not a constant expression, then consider all the class declarations, interface declarations, and method declarations that contain the expression. If any such declaration bears the strictfp modifier (§8.1.1.3, §8.4.3.5, §9.1.1.2), then the expression is FP-strict.

If a class, interface, or method, \( X \), is declared strictfp, then \( X \) and any class, interface, method, constructor, instance initializer, static initializer, or variable initializer within \( X \) is said to be FP-strict.

Note that an annotation's element value (§9.7) is always FP-strict, because it is always a constant expression.

It follows that an expression is not FP-strict if and only if it is not a constant expression and it does not appear within any declaration that has the strictfp modifier.

Within an FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats.

Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results; the net effect, roughly speaking, is that a calculation might produce "the correct answer" in situations where exclusive use of the float value set or double value set might result in overflow or underflow.

15.5 Expressions and Run-Time Checks

If the type of an expression is a primitive type, then the value of the expression is of that same primitive type.
If the type of an expression is a reference type, then the class of the referenced object, or even whether the value is a reference to an object rather than null, is not necessarily known at compile time. There are a few places in the Java programming language where the actual class of a referenced object affects program execution in a manner that cannot be deduced from the type of the expression. They are as follows:

- **Method invocation** (§15.12). The particular method used for an invocation o.m(...) is chosen based on the methods that are part of the class or interface that is the type of o. For instance methods, the class of the object referenced by the run-time value of o participates because a subclass may override a specific method already declared in a parent class so that this overriding method is invoked. (The overriding method may or may not choose to further invoke the original overridden m method.)

- **The instanceof operator** (§15.20.2). An expression whose type is a reference type may be tested using instanceof to find out whether the class of the object referenced by the run-time value of the expression may be converted to some other reference type.

- **Casting** (§15.16). The class of the object referenced by the run-time value of the operand expression might not be compatible with the type specified by the cast operator. For reference types, this may require a run-time check that throws an exception if the class of the referenced object, as determined at run time, cannot be converted to the target type.

- **Assignment to an array component of reference type** (§10.5, §15.13, §15.26.1). The type-checking rules allow the array type S[] to be treated as a subtype of T[] if S is a subtype of T, but this requires a run-time check for assignment to an array component, similar to the check performed for a cast.

- **Exception handling** (§14.20). An exception is caught by a catch clause only if the class of the thrown exception object is an instanceof the type of the formal parameter of the catch clause.

Situations where the class of an object is not statically known may lead to run-time type errors.

In addition, there are situations where the statically known type may not be accurate at run time. Such situations can arise in a program that gives rise to compile-time unchecked warnings. Such warnings are given in response to operations that cannot be statically guaranteed to be safe, and cannot immediately be subjected to dynamic checking because they involve non-reifiable types (§4.7). As a result, dynamic
checks later in the course of program execution may detect inconsistencies and result in run-time type errors.

A run-time type error can occur only in these situations:

- In a cast, when the actual class of the object referenced by the value of the operand expression is not compatible with the target type specified by the cast operator (§5.5, §15.16); in this case a ClassCastException is thrown.
- In an automatically generated cast introduced to ensure the validity of an operation on a non-reifiable type (§4.7).
- In an assignment to an array component of reference type, when the actual class of the object referenced by the value to be assigned is not compatible with the actual run-time component type of the array (§10.5, §15.13, §15.26.1); in this case an ArrayStoreException is thrown.
- When an exception is not caught by any catch clause of a try statement (§14.20); in this case the thread of control that encountered the exception first attempts to invoke an uncaught exception handler (§11.3) and then terminates.

15.6 Normal and Abrupt Completion of Evaluation

Every expression has a normal mode of evaluation in which certain computational steps are carried out. The following sections describe the normal mode of evaluation for each kind of expression.

If all the steps are carried out without an exception being thrown, the expression is said to complete normally.

If, however, evaluation of an expression throws an exception, then the expression is said to complete abruptly. An abrupt completion always has an associated reason, which is always a throw with a given value.

Run-time exceptions are thrown by the predefined operators as follows:

- A class instance creation expression (§15.9.4), array creation expression (§15.10.2), method reference expression (§15.13.3), array initializer expression (§10.6), string concatenation operator expression (§15.18.1), or lambda expression (§15.27.4) throws an OutOfMemoryError if there is insufficient memory available.
• An array creation expression (§15.10.2) throws a NegativeArraySizeException if the value of any dimension expression is less than zero.

• An array access expression (§15.10.4) throws a NullPointerException if the value of the array reference expression is null.

• An array access expression (§15.10.4) throws an ArrayIndexOutOfBoundsException if the value of the array index expression is negative or greater than or equal to the length of the array.

• A field access expression (§15.11) throws a NullPointerException if the value of the object reference expression is null.

• A method invocation expression (§15.12) that invokes an instance method throws a NullPointerException if the target reference is null.

• A cast expression (§15.16) throws a ClassCastException if a cast is found to be impermissible at run time.

• An integer division (§15.17.2) or integer remainder (§15.17.3) operator throws an ArithmeticException if the value of the right-hand operand expression is zero.

• An assignment to an array component of reference type (§15.26.1), a method invocation expression (§15.12), or a prefix or postfix increment (§15.14.2, §15.15.1) or decrement operator (§15.14.3, §15.15.2) may all throw an OutOfMemoryError as a result of boxing conversion (§5.1.7).

• An assignment to an array component of reference type (§15.26.1) throws an ArrayStoreException when the value to be assigned is not compatible with the component type of the array (§10.5).

A method invocation expression can also result in an exception being thrown if an exception occurs that causes execution of the method body to complete abruptly.

A class instance creation expression can also result in an exception being thrown if an exception occurs that causes execution of the constructor to complete abruptly.

Various linkage and virtual machine errors may also occur during the evaluation of an expression. By their nature, such errors are difficult to predict and difficult to handle.

If an exception occurs, then evaluation of one or more expressions may be terminated before all steps of their normal mode of evaluation are complete; such expressions are said to complete abruptly.
If evaluation of an expression requires evaluation of a subexpression, then abrupt completion of the subexpression always causes the immediate abrupt completion of the expression itself, with the same reason, and all succeeding steps in the normal mode of evaluation are not performed.

The terms "complete normally" and "complete abruptly" are also applied to the execution of statements (§14.1). A statement may complete abruptly for a variety of reasons, not just because an exception is thrown.

15.7 Evaluation Order

The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification. Code is usually clearer when each expression contains at most one side effect, as its outermost operation, and when code does not depend on exactly which exception arises as a consequence of the left-to-right evaluation of expressions.

15.7.1 Evaluate Left-Hand Operand First

The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated.

If the operator is a compound-assignment operator (§15.26.2), then evaluation of the left-hand operand includes both remembering the variable that the left-hand operand denotes and fetching and saving that variable's value for use in the implied binary operation.

If evaluation of the left-hand operand of a binary operator completes abruptly, no part of the right-hand operand appears to have been evaluated.

Example 15.7.1-1. Left-Hand Operand Is Evaluated First

In the following program, the * operator has a left-hand operand that contains an assignment to a variable and a right-hand operand that contains a reference to the same variable. The value produced by the reference will reflect the fact that the assignment occurred first.

```java
class Test1 {
    public static void main(String[] args) {
        int i = 2;
        int j = (i=3) * i;
        System.out.println(j);
    }
}
```
This program produces the output:

9

It is not permitted for evaluation of the * operator to produce 6 instead of 9.

**Example 15.7.1-2. Implicit Left-Hand Operand In Operator Of Compound Assignment**

In the following program, the two assignment statements both fetch and remember the value of the left-hand operand, which is 9, before the right-hand operand of the addition operator is evaluated, at which point the variable is set to 3.

```java
class Test2 {
    public static void main(String[] args) {
        int a = 9;
        a += (a = 3);  // first example
        System.out.println(a);
        int b = 9;
        b = b + (b = 3);  // second example
        System.out.println(b);
    }
}
```

This program produces the output:

12
12

It is not permitted for either assignment (compound for a, simple for b) to produce the result 6.

See also the example in §15.26.2.

**Example 15.7.1-3. Abrupt Completion of Evaluation of the Left-Hand Operand**

```java
class Test3 {
    public static void main(String[] args) {
        int j = 1;
        try {
            int i = forgetIt() / (j = 2);
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Now j = " + j);
        }
    }
    static int forgetIt() throws Exception {
        throw new Exception("I'm outta here!");
    }
}
```
This program produces the output:

```
java.lang.Exception: I'm outta here!
Now j = 1
```

That is, the left-hand operand `forgetIt()` of the operator `/` throws an exception before the right-hand operand is evaluated and its embedded assignment of 2 to `j` occurs.

### 15.7.2 Evaluate Operands before Operation

The Java programming language guarantees that every operand of an operator (except the conditional operators `&&`, `||`, and `? :`) appears to be fully evaluated before any part of the operation itself is performed.

If the binary operator is an integer division `/` (§15.17.2) or integer remainder `%` (§15.17.3), then its execution may raise an `ArithmeticException`, but this exception is thrown only after both operands of the binary operator have been evaluated and only if these evaluations completed normally.

#### Example 15.7.2-1. Evaluation of Operands Before Operation

```java
class Test {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            int i = 1 / (divisor * loseBig());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int loseBig() throws Exception {
        throw new Exception("Shuffle off to Buffalo!");
    }
}
```

This program produces the output:

```
java.lang.Exception: Shuffle off to Buffalo!
```

and not:

```
java.lang.ArithmeticException: / by zero
```

since no part of the division operation, including signaling of a divide-by-zero exception, may appear to occur before the invocation of `loseBig` completes, even though the implementation may be able to detect or infer that the division operation would certainly result in a divide-by-zero exception.
15.7.3 Evaluation Respects Parentheses and Precedence

The Java programming language respects the order of evaluation indicated explicitly by parentheses and implicitly by operator precedence.

An implementation of the Java programming language may not take advantage of algebraic identities such as the associative law to rewrite expressions into a more convenient computational order unless it can be proven that the replacement expression is equivalent in value and in its observable side effects, even in the presence of multiple threads of execution (using the thread execution model in §17 (Threads and Locks)), for all possible computational values that might be involved.

In the case of floating-point calculations, this rule applies also for infinity and not-a-number (NaN) values.

For example, !(x<y) may not be rewritten as x>=y, because these expressions have different values if either x or y is NaN or both are NaN.

Specifically, floating-point calculations that appear to be mathematically associative are unlikely to be computationally associative. Such computations must not be naively reordered.

For example, it is not correct for a Java compiler to rewrite 4.0*x*0.5 as 2.0*x; while roundoff happens not to be an issue here, there are large values of x for which the first expression produces infinity (because of overflow) but the second expression produces a finite result.

So, for example, the test program:

```java
strictfp class Test {
    public static void main(String[] args) {
        double d = 8e+307;
        System.out.println(4.0 * d * 0.5);
        System.out.println(2.0 * d);
    }
}
```

prints:

```
Infinity
1.6e+308
```

because the first expression overflows and the second does not.

In contrast, integer addition and multiplication are provably associative in the Java programming language.
For example $a+b+c$, where $a$, $b$, and $c$ are local variables (this simplifying assumption avoids issues involving multiple threads and volatile variables), will always produce the same answer whether evaluated as $(a+b)+c$ or $a+(b+c)$; if the expression $b+c$ occurs nearby in the code, a smart Java compiler may be able to use this common subexpression.

### 15.7.4 Argument Lists are Evaluated Left-to-Right

In a method or constructor invocation or class instance creation expression, argument expressions may appear within the parentheses, separated by commas. Each argument expression appears to be fully evaluated before any part of any argument expression to its right.

If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

**Example 15.7.4-1. Evaluation Order At Method Invocation**

```java
class Test1 {
    public static void main(String[] args) {
        String s = "going, ";
        print3(s, s, s = "gone");
    }
    static void print3(String a, String b, String c) {
        System.out.println(a + b + c);
    }
}
```

This program produces the output:

```
going, going, gone
```

because the assignment of the string "gone" to $s$ occurs after the first two arguments to $\text{print3}$ have been evaluated.

**Example 15.7.4-2. Abrupt Completion of Argument Expression**

```java
class Test2 {
    static int id;
    public static void main(String[] args) {
        try {
            test(id = 1, oops(), id = 3);
        } catch (Exception e) {
            System.out.println(e + ", id=", id);
        }
    }
    static int test(int a, int b, int c) {
        return a + b + c;
    }
    static int oops() throws Exception {
        throw new Exception("oops");
    }
}
```
This program produces the output:

    java.lang.Exception: oops, id=1

because the assignment of 3 to id is not executed.

15.7.5 Evaluation Order for Other Expressions

The order of evaluation for some expressions is not completely covered by these general rules, because these expressions may raise exceptional conditions at times that must be specified. See the detailed explanations of evaluation order for the following kinds of expressions:

- class instance creation expressions (§15.9.4)
- array creation expressions (§15.10.2)
- array access expressions (§15.10.4)
- method invocation expressions (§15.12.4)
- method reference expressions (§15.13.3)
- assignments involving array components (§15.26)
- lambda expressions (§15.27.4)

15.8 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, object creations, field accesses, method invocations, method references, and array accesses. A parenthesized expression is also treated syntactically as a primary expression.

Primary:

PrimaryNoNewArray
ArrayCreationExpression
PrimaryNoNewArray:
Literal
ClassLiteral
this
TypeName . this
( Expression )
ClassInstanceCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

This part of the grammar of the Java programming language is unusual, in two ways. First, one might expect simple names, such as names of local variables and method parameters, to be primary expressions. For technical reasons, names are grouped together with primary expressions a little later when postfix expressions are introduced (§15.14).

The technical reasons have to do with allowing left-to-right parsing of Java programs with only one-token lookahead. Consider the expressions \((z[3])\) and \((z[])\). The first is a parenthesized array access (§15.10.3) and the second is the start of a cast (§15.16). At the point that the look-ahead symbol is \([\)\), a left-to-right parse will have reduced the \(z\) to the nonterminal \(Name\). In the context of a cast we prefer not to have to reduce the name to a Primary, but if Name were one of the alternatives for Primary, then we could not tell whether to do the reduction (that is, we could not determine whether the current situation would turn out to be a parenthesized array access or a cast) without looking ahead two tokens, to the token following the \([\). The grammar presented here avoids the problem by keeping Name and Primary separate and allowing either in certain other syntax rules (those for ClassInstanceCreationExpression, MethodInvocation, ArrayAccess, and PostfixExpression, though not FieldAccess because it uses an identifier directly). This strategy effectively defers the question of whether a Name should be treated as a Primary until more context can be examined.

The second unusual feature avoids a potential grammatical ambiguity in the expression "new int[3][3]" which in Java always means a single creation of a multidimensional array, but which, without appropriate grammatical finesse, might also be interpreted as meaning the same as "(new int[3])[3]."

This ambiguity is eliminated by splitting the expected definition of Primary into Primary and PrimaryNoNewArray. (This may be compared to the splitting of Statement into Statement and StatementNoShortIf (§14.5) to avoid the "dangling else" problem.)

15.8.1 Lexical Literals

A literal (§3.10) denotes a fixed, unchanging value.

The following production from §3.10 is shown here for convenience:
The type of a literal is determined as follows:

- The type of an integer literal (§3.10.1) that ends with L or l is long (§4.2.1). The type of any other integer literal is int (§4.2.1).
- The type of a floating-point literal (§3.10.2) that ends with F or f is float and its value must be an element of the float value set (§4.2.3). The type of any other floating-point literal is double and its value must be an element of the double value set (§4.2.3).
- The type of a boolean literal (§3.10.3) is boolean (§4.2.5).
- The type of a character literal (§3.10.4) is char (§4.2.1).
- The type of a string literal (§3.10.5) is String (§4.3.3).
- The type of the null literal null (§3.10.7) is the null type (§4.1); its value is the null reference.

Evaluation of a lexical literal always completes normally.

### 15.8.2 Class Literals

A class literal is an expression consisting of the name of a class, interface, array, or primitive type, or the pseudo-type void, followed by a '.' and the token class.

```
ClassLiteral:  
  TypeName [[]].class  
  NumericType [[]].class  
  boolean [[]].class  
  void.class
```

The `TypeName` must denote a class or interface type that is accessible (§6.6). It is a compile-time error if the `TypeName` denotes a class or interface type that is not accessible, or denotes a type variable.

The type of `C.class`, where `C` is the name of a class, interface, or array type (§4.3), is `Class<C>`. 

501
The type of \( p . \text{class} \), where \( p \) is the name of a primitive type (§4.2), is \( \text{Class}<B> \), where \( B \) is the type of an expression of type \( p \) after boxing conversion (§5.1.7).

The type of \( \text{void.class} \) (§8.4.5) is \( \text{Class}<\text{Void}> \).

A class literal evaluates to the \( \text{Class} \) object for the named type (or for \( \text{void} \)) as defined by the defining class loader (§12.2) of the class of the current instance.

### 15.8.3 this

The keyword \textit{this} may be used only in the following contexts:

- in the body of an instance method or default method (§8.4.7, §9.4.3)
- in the body of a constructor of a class (§8.8.7)
- in an instance initializer of a class (§8.6)
- in the initializer of an instance variable of a class (§8.3.2)
- to denote a receiver parameter (§8.4)

If it appears anywhere else, a compile-time error occurs.

The keyword \textit{this} may be used in a lambda expression only if it is allowed in the context in which the lambda expression appears. Otherwise, a compile-time error occurs.

When used as a primary expression, the keyword \textit{this} denotes a value that is a reference to the object for which the instance method or default method was invoked (§15.12), or to the object being constructed. The value denoted by \textit{this} in a lambda body is the same as the value denoted by \textit{this} in the surrounding context.

The keyword \textit{this} is also used in explicit constructor invocation statements (§8.8.7.1).

The type of \textit{this} is the class or interface type \( T \) within which the keyword \textit{this} occurs.

Default methods provide the unique ability to access \textit{this} inside an interface. (All other interface methods are either \textit{abstract} or \textit{static}, so provide no access to \textit{this}.) As a result, it is possible for \textit{this} to have an interface type.

At run time, the class of the actual object referred to may be \( T \), if \( T \) is a class type, or a class that is a subtype of \( T \).

**Example 15.8.3-1. The this Expression**

```java
class IntVector {
    int[] v;
```
Here, the class `IntVector` implements a method `equals`, which compares two vectors. If the other vector is the same vector object as the one for which the `equals` method was invoked, then the check can skip the length and value comparisons. The `equals` method implements this check by comparing the reference to the other object to `this`.

### 15.8.4 Qualified `this`

Any lexically enclosing instance (§8.1.3) can be referred to by explicitly qualifying the keyword `this`.

Let \( T \) be the type denoted by `TypeName`. Let \( n \) be an integer such that \( T \) is the \( n \)’th lexically enclosing type declaration of the class or interface in which the qualified `this` expression appears.

The value of an expression of the form `TypeName.this` is the \( n \)’th lexically enclosing instance of `this`.

The type of the expression is \( T \).

It is a compile-time error if the expression occurs in a class or interface which is not an inner class of class \( T \) or \( T \) itself.

### 15.8.5 Parenthesized Expressions

A parenthesized expression is a primary expression whose type is the type of the contained expression and whose value at run time is the value of the contained expression. If the contained expression denotes a variable then the parenthesized expression also denotes that variable.

The use of parentheses affects only the order of evaluation, except for a corner case whereby \((-2147483648)\) and \((-9223372036854775808L)\) are legal but \(-(2147483648)\) and \-\((9223372036854775808L)\) are illegal.

This is because the decimal literals \(2147483648\) and \(9223372036854775808\) are allowed only as an operand of the unary minus operator (§3.10.1).
In particular, the presence or absence of parentheses around an expression does not (except for the case noted above) affect in any way:

- the choice of value set (§4.2.3) for the value of an expression of type float or double.
- whether a variable is definitely assigned, definitely assigned when true, definitely assigned when false, definitely unassigned, definitely unassigned when true, or definitely unassigned when false (§16 (Definite Assignment)).

If a parenthesized expression appears in a context of a particular kind with target type $\tau$ (§5 (Conversions and Contexts)), its contained expression similarly appears in a context of the same kind with target type $\tau$.

If the contained expression is a poly expression (§15.2), the parenthesized expression is also a poly expression. Otherwise, it is a standalone expression.

A poly parenthesized expression is compatible with a target type $\tau$ if its contained expression is compatible with $\tau$.

### 15.9 Class Instance Creation Expressions

A class instance creation expression is used to create new objects that are instances of classes.

```
ClassInstanceCreationExpression:
   UnqualifiedClassInstanceCreationExpression
   ExpressionName . UnqualifiedClassInstanceCreationExpression
   Primary . UnqualifiedClassInstanceCreationExpression

UnqualifiedClassInstanceCreationExpression:
   new [TypeArguments]
   ClassOrInterfaceTypeToInstantiate ( [ArgumentList] ) [ClassBody]

ClassOrInterfaceTypeToInstantiate:
   [Annotation] Identifier . [Annotation] Identifier
   [TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
   TypeArguments
   <>
```

The following production from §15.12 is shown here for convenience:
A class instance creation expression specifies a class to be instantiated, possibly followed by type arguments (§4.5.1) or a diamond ($<$>) if the class being instantiated is generic (§8.1.2), followed by (a possibly empty) list of actual value arguments to the constructor.

If the type argument list to the class is empty — the diamond form $<$> — the type arguments of the class are inferred. It is legal, though strongly discouraged as a matter of style, to have white space between the "$<$" and "$>$" of a diamond.

If the constructor is generic (§8.8.4), the type arguments to the constructor may similarly either be inferred or passed explicitly. If passed explicitly, the type arguments to the constructor immediately follow the keyword `new`.

It is a compile-time error if a class instance creation expression provides type arguments to a constructor but uses the diamond form for type arguments to the class.

This rule is introduced because inference of a generic class's type arguments may influence the constraints on a generic constructor's type arguments.

If `TypeArguments` is present immediately after `new`, or immediately before `,`, then it is a compile-time error if any of the type arguments are wildcards (§4.5.1).

The exception types that a class instance creation expression can throw are specified in §11.2.1.

Class instance creation expressions have two forms:

- **Unqualified class instance creation expressions** begin with the keyword `new`.
  
  An unqualified class instance creation expression may be used to create an instance of a class, regardless of whether the class is a top level (§7.6), member (§8.5, §9.5), local (§14.3), or anonymous class (§15.9.5).

- **Qualified class instance creation expressions** begin with a `Primary` expression or an `ExpressionName`.

  A qualified class instance creation expression enables the creation of instances of inner member classes and their anonymous subclasses.

Both unqualified and qualified class instance creation expressions may optionally end with a class body. Such a class instance creation expression declares an anonymous class (§15.9.5) and creates an instance of it.
A class instance creation expression is a poly expression (§15.2) if it uses the
diamond form for type arguments to the class, and it appears in an assignment
context or an invocation context (§5.2, §5.3). Otherwise, it is a standalone
expression.

We say that a class is *instantiated* when an instance of the class is created by a class
instance creation expression. Class instantiation involves determining the class
to be instantiated (§15.9.1), the enclosing instances (if any) of the newly created
instance (§15.9.2), and the constructor to be invoked to create the new instance
(§15.9.3).

### 15.9.1 Determining the Class being Instantiated

If `ClassOrInterfaceTypeToInstantiate` ends with `TypeArguments` (rather than `<>`),
then `ClassOrInterfaceTypeToInstantiate` must denote a well-formed parameterized
type (§4.5), or a compile-time error occurs.

If `ClassOrInterfaceTypeToInstantiate` ends with `<>`, but the type denoted by the
Identifier in `ClassOrInterfaceTypeToInstantiate` is not generic, then a compile-time
error occurs.

If the class instance creation expression ends in a class body, then the class being
instantiated is an anonymous class. Then:

- If the class instance creation expression is unqualified, then:

  The Identifier in `ClassOrInterfaceTypeToInstantiate` must denote either a class
  that is accessible, non-final, and not an enum type, or an interface that is
  accessible (§6.6). Otherwise a compile-time error occurs.

  If the Identifier in `ClassOrInterfaceTypeToInstantiate` denotes a class, `c`, then an
  anonymous direct subclass of `c` is declared. If `TypeArguments` is present, then `c`
  has type arguments given by `TypeArguments`; if `<>` is present, then `c` will have its
type arguments inferred in §15.9.3; otherwise, `c` has no type arguments. The body
  of the subclass is the `ClassBody` given in the class instance creation expression.
  The class being instantiated is the anonymous subclass.

  If the Identifier in `ClassOrInterfaceTypeToInstantiate` denotes an interface, `i`, then an
  anonymous direct subclass of `Object` that implements `i` is declared. If `TypeArguments`
  is present, then `i` has type arguments given by `TypeArguments`; if `<>` is present, then `i`
  will have its type arguments inferred in §15.9.3; otherwise, `i` has no type arguments. The body
  of the subclass is the `ClassBody` given in the class instance creation expression. The class being instantiated is the anonymous subclass.
• If the class instance creation expression is qualified, then:

The *Identifier* in *ClassOrInterfaceTypeToInstantiate* must unambiguously denote an inner class that is accessible, non-final, not an enum type, and a member of the compile-time type of the *Primary* expression or the *ExpressionName*. Otherwise, a compile-time error occurs.

Let the *Identifier* in *ClassOrInterfaceTypeToInstantiate* denote a class, C. An anonymous direct subclass of C is declared. If *TypeArguments* is present, then C has type arguments given by *TypeArguments*; if <> is present, then C will have its type arguments inferred in §15.9.3; otherwise, C has no type arguments. The body of the subclass is the *ClassBody* given in the class instance creation expression. The class being instantiated is the anonymous subclass.

If a class instance creation expression does not declare an anonymous class, then:

• If the class instance creation expression is unqualified, then:

The *Identifier* in *ClassOrInterfaceTypeToInstantiate* must denote a class that is accessible, non-abstract, and not an enum type. Otherwise, a compile-time error occurs.

The class being instantiated is specified by the *Identifier* in *ClassOrInterfaceTypeToInstantiate*. If *TypeArguments* is present, then the class has type arguments given by *TypeArguments*; if <> is present, then the class will have its type arguments inferred in §15.9.3; otherwise, the class has no type arguments.

• If the class instance creation expression is qualified, then:

The *ClassOrInterfaceTypeToInstantiate* must unambiguously denote an inner class that is accessible, non-abstract, not an enum type, and a member of the compile-time type of the *Primary* expression or the *ExpressionName*.

The class being instantiated is specified by the *Identifier* in *ClassOrInterfaceTypeToInstantiate*. If *TypeArguments* is present, then the class has type arguments given by *TypeArguments*; if <> is present, then the class will have its type arguments inferred in §15.9.3; otherwise, the class has no type arguments.

### 15.9.2 Determining Enclosing Instances

Let C be the class being instantiated, and let i be the instance being created. If C is an inner class, then i may have an *immediately enclosing instance* (§8.1.3), determined as follows:
• If \( c \) is an anonymous class, then:
  – If the class instance creation expression occurs in a static context, then \( i \) has no immediately enclosing instance.
  – Otherwise, the immediately enclosing instance of \( i \) is this.

• If \( c \) is a local class, then:
  – If \( c \) occurs in a static context, then \( i \) has no immediately enclosing instance.
  – Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
  – Otherwise, let \( o \) be the immediately enclosing class of \( c \). Let \( n \) be an integer such that \( o \) is the \( n \)'th lexically enclosing type declaration of the class in which the class instance creation expression appears.
    The immediately enclosing instance of \( i \) is the \( n \)'th lexically enclosing instance of this.

• If \( c \) is an inner member class, then:
  – If the class instance creation expression is unqualified, then:
    › If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    › Otherwise, if \( c \) is a member of a class enclosing the class in which the class instance creation expression appears, then let \( o \) be the immediately enclosing class of which \( c \) is a member. Let \( n \) be an integer such that \( o \) is the \( n \)'th lexically enclosing type declaration of the class in which the class instance creation expression appears.
      The immediately enclosing instance of \( i \) is the \( n \)'th lexically enclosing instance of this.
    › Otherwise, a compile-time error occurs.
  – If the class instance creation expression is qualified, then the immediately enclosing instance of \( i \) is the object that is the value of the Primary expression or the ExpressionName.

If \( c \) is an anonymous class, and its direct superclass \( s \) is an inner class, then \( i \) may have an immediately enclosing instance with respect to \( s \), determined as follows:

• If \( s \) is a local class, then:
  – If \( s \) occurs in a static context, then \( i \) has no immediately enclosing instance with respect to \( s \).
– Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.

– Otherwise, let \( \sigma \) be the immediately enclosing type declaration of \( S \). Let \( n \) be an integer such that \( \sigma \) is the \( n \)'th lexically enclosing type declaration of the class in which the class instance creation expression appears.

The immediately enclosing instance of \( i \) with respect to \( S \) is the \( n \)'th lexically enclosing instance of this.

• If \( S \) is an inner member class, then:

  – If the class instance creation expression is unqualified, then:
    
    › If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    
    › Otherwise, if \( S \) is a member of a class enclosing the class in which the class instance creation expression appears, then let \( \sigma \) be the immediately enclosing class of which \( S \) is a member. Let \( n \) be an integer such that \( \sigma \) is the \( n \)'th lexically enclosing type declaration of the class in which the class instance creation expression appears.

    The immediately enclosing instance of \( i \) with respect to \( S \) is the \( n \)'th lexically enclosing instance of this.

    › Otherwise, a compile-time error occurs.

  – If the class instance creation expression is qualified, then the immediately enclosing instance of \( i \) with respect to \( S \) is the object that is the value of the Primary expression or the ExpressionName.

15.9.3 Choosing the Constructor and its Arguments

Let \( c \) be the class being instantiated. To create an instance of \( c, i \), a constructor of \( c \) is chosen at compile time by the following rules.

First, the actual arguments to the constructor invocation are determined:

• If \( c \) is an anonymous class with direct superclass \( s \), then:

  – If \( s \) is not an inner class, or if \( s \) is a local class that occurs in a static context, then the arguments to the constructor are the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the expression.
- Otherwise, the first argument to the constructor is the immediately enclosing instance of i with respect to s (§15.9.2), and the subsequent arguments to the constructor are the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the class instance creation expression.

- If c is a local class or a private inner member class, then the arguments to the constructor are the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the class instance creation expression.

- If c is a non-private inner member class, then the first argument to the constructor is the immediately enclosing instance of i (§8.8.1, §15.9.2), and the subsequent arguments to its constructor are the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the class instance creation expression.

- Otherwise, the arguments to the constructor are the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the expression.

Second, a constructor of c and corresponding throws clause and return type are determined:

- If the class instance creation expression does not use <>, then:
  - If c is not an anonymous class, then:
    
    Let τ be the type denoted by c followed by any class type arguments in the expression. The process specified in §15.12.2, modified to handle constructors, is used to choose one of the constructors of τ and determine its throws clause.
    
    If there is no unique most-specific constructor in τ that is both applicable and accessible (§6.6), then a compile-time error occurs (as in method invocations).
    
    Otherwise, the return type corresponding to the chosen constructor is τ.
    
  - If c is an anonymous class, then:
    
    The process specified in §15.12.2, modified to handle constructors, is used to choose one of the constructors of the direct superclass of c and determine its throws clause.
    
    If there is no unique most-specific constructor in the direct superclass of c that is both applicable and accessible, then a compile-time error occurs (as in method invocations).
Otherwise, c’s anonymous constructor is chosen as the constructor of c
(§15.9.5.1). Its body consists of an explicit constructor invocation of the
constructor chosen in the direct superclass of c.

The throws clause of the chosen constructor includes the exceptions in the
throws clause of the constructor chosen in the direct superclass of c.

The return type corresponding to the chosen constructor is the anonymous
class type.

• If the class instance creation expression uses <>, then:

If c is not an anonymous class, let d be the same as c. If c is an anonymous class,
let d be the superclass or superinterface of c named by the class instance creation
expression.

If d is a class, let c₁...cₚ be the constructors of class D. If d is an interface, let
Cₖ...Cₘ be a singleton list (n = 1) containing the zero-argument constructor of
class Object.

A list of methods m₁...mₙ is defined for the purpose of overload resolution and type
argument inference. For all j (1 ≤ j ≤ n), m_j is defined in terms of c_j as follows:

– A substitution θ_j is first defined to instantiate the types in c_j.

  Let F₁...Fₚ be the type parameters of d, and let G₁...Gₚ be the type parameters
  (if any) of c_j. Let X₁...Xₚ and Y₁...Yₚ be type variables with distinct names that
  are not in scope in the body of d.

  θ_j is [F₁:=X₁, ..., Fₚ:=Xₚ, G₁:=Y₁, ..., Gₚ:=Yₚ].

– The type parameters of m_j are X₁...Xₚ, Y₁...Yₚ. The bound of each type parameter,
  if any, is θ_j applied to the corresponding type parameter bound in d or c_j.

– The return type of m_j is θ_j applied to D<F₁,...,Fₚ>.

– The (possibly empty) list of argument types of m_j is θ_j applied to the argument
types of c_j.

– The (possibly empty) list of thrown types of m_j is θ_j applied to the thrown
types of c_j.

– The modifiers of m_j are those of c_j.

– The name of m_j is #m, an automatically generated name that is distinct from all
  constructor and method names in d and is shared by m₁...mₙ.

– The body of m_j is irrelevant.
To choose a constructor, we temporarily consider $m_1...m_n$ to be members of $D$. One of $m_1...m_n$ is chosen, as determined by the class instance creation expression's argument expressions, using the process specified in §15.12.2.

If there is no unique most specific method that is both applicable and accessible, then a compile-time error occurs.

Otherwise, where $m_j$ is the chosen method:

- If $C$ is not an anonymous class, then $c_j$ is chosen as the constructor of $C$.
  
  The `throws` clause of the chosen constructor is the same as the `throws` clause determined for $m_j$.
  
  The return type corresponding to the chosen constructor is the return type determined for $m_j$ (§15.12.2.6).

- If $C$ is an anonymous class, then $C$’s anonymous constructor is chosen as the constructor of $C$. Its body consists of an explicit constructor invocation of $c_j$.
  
  The `throws` clause of the chosen constructor includes the exceptions in the `throws` clause determined for $m_j$.
  
  The return type corresponding to the chosen constructor is the anonymous class type.

If the class instance creation expression is a poly expression, then its compatibility with a target type is as determined by §18.5.2.1, using $m_j$ as the selected method $m$.

Testing for compatibility with a target type may occur multiple times before making a final determination of the class instance creation expression’s target type and the return type corresponding to the chosen constructor. For example, an enclosing method invocation expression may require testing the class instance creation expression for compatibility with different methods’ formal parameter types.

If $C$ is an anonymous class, then its superclass or superinterface (§15.9.5) is the return type determined for $m_j$ (§15.12.2.6).

It is a compile-time error if the superclass or superinterface, or any subexpression therein ("subexpression" includes type arguments of parameterized types, bounds of wildcard type arguments, and element types of array types, but excludes bounds of type variables), has one of the following forms:

- A type variable that was not declared as a type parameter (such as a type variable produced by capture conversion).

- An intersection type.
– A class or interface type, where the class or interface declaration is not accessible from the class or interface in which the class instance creation expression appears.

It is a compile-time error if an argument to a class instance creation expression is not compatible with its target type, as derived from the invocation type (§15.12.2.6).

If the compile-time declaration is applicable by variable arity invocation (§15.12.2.4), then where the last formal parameter type of the invocation type of the constructor is $F_n[]$, it is a compile-time error if the type which is the erasure of $F_n$ is not accessible at the point of invocation.

The type of the class instance creation expression is the return type corresponding to the chosen constructor, as defined above.

15.9.4 Run-Time Evaluation of Class Instance Creation Expressions

At run time, evaluation of a class instance creation expression is as follows.

First, if the class instance creation expression is a qualified class instance creation expression, the qualifying primary expression is evaluated. If the qualifying expression evaluates to `null`, a `NullPointerException` is raised, and the class instance creation expression completes abruptly. If the qualifying expression completes abruptly, the class instance creation expression completes abruptly for the same reason.

Next, space is allocated for the new class instance. If there is insufficient space to allocate the object, evaluation of the class instance creation expression completes abruptly by throwing an `OutOfMemoryError`.

The new object contains new instances of all the fields declared in the specified class type and all its superclasses. As each new field instance is created, it is initialized to its default value (§4.12.5).

Next, the actual arguments to the constructor are evaluated, left-to-right. If any of the argument evaluations completes abruptly, any argument expressions to its right are not evaluated, and the class instance creation expression completes abruptly for the same reason.

Next, the selected constructor of the specified class type is invoked. This results in invoking at least one constructor for each superclass of the class type. This process can be directed by explicit constructor invocation statements (§8.8) and is specified in detail in §12.5.
15.9

Class Instance Creation Expressions

EXPRESSIONS

The value of a class instance creation expression is a reference to the newly created
object of the specified class. Every time the expression is evaluated, a fresh object
is created.
Example 15.9.4-1. Evaluation Order and Out-Of-Memory Detection
If evaluation of a class instance creation expression finds there is insufficient memory to
perform the creation operation, then an OutOfMemoryError is thrown. This check occurs
before any argument expressions are evaluated.
So, for example, the test program:
class List {
int value;
List next;
static List head = new List(0);
List(int n) { value = n; next = head; head = this; }
}
class Test {
public static void main(String[] args) {
int id = 0, oldid = 0;
try {
for (;;) {
++id;
new List(oldid = id);
}
} catch (Error e) {
List.head = null;
System.out.println(e.getClass() + ", " + (oldid==id));
}
}
}

prints:
class java.lang.OutOfMemoryError, false

because the out-of-memory condition is detected before the argument expression oldid
= id is evaluated.
Compare this to the treatment of array creation expressions, for which the out-of-memory
condition is detected after evaluation of the dimension expressions (§15.10.2).

15.9.5 Anonymous Class Declarations
An anonymous class declaration is automatically derived from a class instance
creation expression by the Java compiler.
An anonymous class is never abstract (§8.1.1.1).
514


An anonymous class is never final (§8.1.2).

The fact that an anonymous class is not final is relevant in casting, in particular the narrowing reference conversion allowed for the cast operator (§5.5). It is also of interest in subclassing, in that it is impossible to declare a subclass of an anonymous class, despite an anonymous class being non-final, because an anonymous class cannot be named by an extends clause (§8.1.4).

An anonymous class is always an inner class (§8.1.3); it is never static (§8.1.1, §8.5.1).

The superclass or superinterface of an anonymous class is given by the class instance creation expression (§15.9.1), with type arguments inferred as necessary while choosing a constructor (§15.9.3).

If the class instance creation expression uses <> with an anonymous class, then for all non-private methods declared in the anonymous class body, it is as if the method declaration is annotated with @Override (§9.6.4.4).

When <> is used, the inferred type arguments may not be as anticipated by the programmer. Consequently, the supertype of the anonymous class may not be as anticipated, and methods declared in the anonymous class may not override supertype methods as intended. Treating such methods as if annotated with @Override (if they are not explicitly annotated with @Override) helps avoid silently incorrect programs.

15.9.5.1 Anonymous Constructors

An anonymous class cannot have an explicitly declared constructor. Instead, an anonymous constructor is implicitly declared for an anonymous class. The form of the anonymous constructor for an anonymous class C with direct superclass S is as follows:

• If S is not an inner class, or if S is a local class that occurs in a static context, then the anonymous constructor has one formal parameter for each actual argument to the class instance creation expression in which C is declared.

The actual arguments to the class instance creation expression are used to determine a constructor cs of S, as specified by §15.9.3. The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of cs.

The constructor body consists of an explicit constructor invocation (§8.8.7.1) of the form super(...), where the actual arguments are the formal parameters of the constructor, in the order they were declared. The superclass constructor to be invoked is cs.
• Otherwise, the first formal parameter of the constructor of \( C \) represents the value of the immediately enclosing instance of \( i \) with respect to \( S \) (§15.9.2, §15.9.3). The type of this parameter is the class type that immediately encloses the declaration of \( S \).

The constructor has an additional formal parameter for each actual argument to the class instance creation expression that declared the anonymous class. The \( n \)'th formal parameter \( e \) corresponds to the \( n-1 \)'th actual argument.

The actual arguments to the class instance creation expression are used to determine a constructor \( cs \) of \( S \), as specified by §15.9.3. The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of \( cs \).

The constructor body consists of an explicit constructor invocation (§8.8.7.1) of the form \( o.super(...) \), where \( o \) is the first formal parameter of the constructor, and the actual arguments are the subsequent formal parameters of the constructor, in the order they were declared. The superclass constructor to be invoked is \( cs \).

In all cases, the \texttt{throws} clause of an anonymous constructor must list all the checked exceptions thrown by the explicit superclass constructor invocation statement contained within the anonymous constructor, as specified in §15.9.3, and all checked exceptions thrown by any instance initializers or instance variable initializers of the anonymous class.

Note that it is possible for the signature of the anonymous constructor to refer to an inaccessible type (for example, if such a type occurred in the signature of the superclass constructor \( cs \)). This does not, in itself, cause any errors at either compile-time or run-time.

### 15.10 Array Creation and Access Expressions

#### 15.10.1 Array Creation Expressions

An array creation expression is used to create new arrays (§10 (Arrays)).

\[
\text{ArrayCreationExpression:}
\begin{align*}
\text{new PrimitiveType DimExprs [Dims]} \\
\text{new ClassOrInterfaceType DimExprs [Dims]} \\
\text{new PrimitiveType Dims ArrayInitializer} \\
\text{new ClassOrInterfaceType Dims ArrayInitializer}
\end{align*}
\]
An array creation expression creates an object that is a new array whose elements are of the type specified by the PrimitiveType or ClassOrInterfaceType.

It is a compile-time error if the ClassOrInterfaceType does not denote a reifiable type (§4.7). Otherwise, the ClassOrInterfaceType may name any named reference type, even an abstract class type (§8.1.1.1) or an interface type.

The rules above imply that the element type in an array creation expression cannot be a parameterized type, unless all type arguments to the parameterized type are unbounded wildcards.

The type of each dimension expression within a DimExpr must be a type that is convertible (§5.1.8) to an integral type, or a compile-time error occurs.

Each dimension expression undergoes unary numeric promotion (§5.6.1). The promoted type must be int, or a compile-time error occurs.

The type of the array creation expression is an array type that can denoted by a copy of the array creation expression from which the new keyword and every DimExpr expression and array initializer have been deleted.

For example, the type of the creation expression:

```
new double[3][3][]
```

is:

```
double[][][]
```
Array Creation and Access Expressions

initializer as described in §10.6. The value of the array initializer becomes the value of the array creation expression.

- Otherwise, there is no array initializer, and:
  - First, the dimension expressions are evaluated, left-to-right. If any of the expression evaluations completes abruptly, the expressions to the right of it are not evaluated.
  - Next, the values of the dimension expressions are checked. If the value of any DimExpr expression is less than zero, then a NegativeArraySizeException is thrown.
  - Next, space is allocated for the new array. If there is insufficient space to allocate the array, evaluation of the array creation expression completes abruptly by throwing an OutOfMemoryError.
  - Then, if a single DimExpr appears, a one-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.12.5).
  - Otherwise, if n DimExpr expressions appear, then array creation effectively executes a set of nested loops of depth n–1 to create the implied arrays of arrays.

A multidimensional array need not have arrays of the same length at each level.

Example 15.10.2-1. Array Creation Evaluation

In an array creation expression with one or more dimension expressions, each dimension expression is fully evaluated before any part of any dimension expression to its right. Thus:

```java
class Test1 {
    public static void main(String[] args) {
        int i = 4;
        int ia[][] = new int[i][i=3];
        System.out.println("[", ia.length + ",", ia[0].length + "]");
    }
}
```

prints:

```
[4,3]
```

because the first dimension is calculated as 4 before the second dimension expression sets i to 3.
If evaluation of a dimension expression completes abruptly, no part of any dimension expression to its right will appear to have been evaluated. Thus:

```java
class Test2 {
    public static void main(String[] args) {
        int[][] a = { { 00, 01 }, { 10, 11 } };
        int i = 99;
        try {
            a[val()] [i = 1]++;
        } catch (Exception e) {
            System.out.println(e + ", i=", i);
        }
    }
    static int val() throws Exception {
        throw new Exception("unimplemented");
    }
}
```

prints:

```
java.lang.Exception: unimplemented, i=99
```

because the embedded assignment that sets \textit{i} to 1 is never executed.

**Example 15.10.2-2. Multi-Dimensional Array Creation**

The declaration:

```java
float[][] matrix = new float[3][3];
```

is equivalent in behavior to:

```java
float[][] matrix = new float[3][];
for (int \textit{d} = 0; \textit{d} < matrix.length; \textit{d}++)
    matrix[\textit{d}] = new float[3];
```

and:

```java
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

is equivalent to:
with $d$, $d_1$, $d_2$, and $d_3$ replaced by names that are not already locally declared. Thus, a single new expression actually creates one array of length 6, 6 arrays of length 10, $6 \times 10 = 60$ arrays of length 8, and $6 \times 10 \times 8 = 480$ arrays of length 12. This example leaves the fifth dimension, which would be arrays containing the actual array elements (references to Age objects), initialized only to null references. These arrays can be filled in later by other code, such as:

```java
Age[] Hair = { new Age("quartz"), new Age("topaz")};
Aquarius[1][9][6][9] = Hair;
```

A triangular matrix may be created by:

```java
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
    triang[i] = new float[i+1];
```

If evaluation of an array creation expression finds there is insufficient memory to perform the creation operation, then an OutOfMemoryError is thrown. If the array creation expression does not have an array initializer, then this check occurs only after evaluation of all dimension expressions has completed normally. If the array creation expression does have an array initializer, then an OutOfMemoryError can occur when an object of reference type is allocated during evaluation of a variable initializer expression, or when space is allocated for an array to hold the values of a (possibly nested) array initializer.

**Example 15.10.2-3. OutOfMemoryError and Dimension Expression Evaluation**

```java
class Test3 {
    public static void main(String[] args) {
        int len = 0, oldlen = 0;
        Object[] a = new Object[0];
        try {
            for (;;) {
                ++len;
                Object[] temp = new Object[oldlen = len];
                temp[0] = a;
                a = temp;
            }
        } catch (Error e) {
            System.out.println(e + ", " + (oldlen==len));
        }
    }
}
This program produces the output:

```java
java.lang.OutOfMemoryError, true
```

because the out-of-memory condition is detected after the dimension expression oldlen = len is evaluated.

Compare this to class instance creation expressions (§15.9), which detect the out-of-memory condition before evaluating argument expressions (§15.9.4).

### 15.10.3 Array Access Expressions

An array access expression refers to a variable that is a component of an array.

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets).

Note that the array reference expression may be a name or any primary expression that is not an array creation expression (§15.10).

The type of the array reference expression must be an array type (call it \( T[\] \), an array whose components are of type \( T \)), or a compile-time error occurs.

The index expression undergoes unary numeric promotion (§5.6.1). The promoted type must be \( \text{int} \), or a compile-time error occurs.

The type of the array access expression is the result of applying capture conversion (§5.1.10) to \( T \).

The result of an array access expression is a variable of type \( T \), namely the variable within the array selected by the value of the index expression.

This resulting variable, which is a component of the array, is never considered final, even if the array reference expression denoted a final variable.

### 15.10.4 Run-Time Evaluation of Array Access Expressions

At run time, evaluation of an array access expression behaves as follows:
• First, the array reference expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason and the index expression is not evaluated.

• Otherwise, the index expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason.

• Otherwise, if the value of the array reference expression is \texttt{null}, then a \texttt{NullPointerException} is thrown.

• Otherwise, the value of the array reference expression indeed refers to an array. If the value of the index expression is less than zero, or greater than or equal to the array's \texttt{length}, then an \texttt{ArrayIndexOutOfBoundsException} is thrown.

• Otherwise, the result of the array access is the variable of type \(\tau\), within the array, selected by the value of the index expression.

**Example 15.10.4-1. Array Reference Is Evaluated First**

In an array access, the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated. For example, in the (admittedly monstrous) expression \(a[(a=b)[3]]\), the expression \(a\) is fully evaluated before the expression \((a=b)[3]\); this means that the original value of \(a\) is fetched and remembered while the expression \((a=b)[3]\) is evaluated. This array referenced by the original value of \(a\) is then subscripted by a value that is element 3 of another array (possibly the same array) that was referenced by \(b\) and is now also referenced by \(a\).

Thus, the program:

```java
class Test1 {
    public static void main(String[] args) {
        int[] a = { 11, 12, 13, 14 };  
        int[] b = { 0, 1, 2, 3 };       
        System.out.println(a[(a=b)[3]]); 
    }
}
```

prints:

```
14
```

because the monstrous expression's value is equivalent to \(a[b[3]]\) or \(a[3]\) or 14.

**Example 15.10.4-2. Abrupt Completion of Array Reference Evaluation**

If evaluation of the expression to the left of the brackets completes abruptly, no part of the expression within the brackets will appear to have been evaluated. Thus, the program:

```java
class Test2 {
```

public static void main(String[] args) {
    int index = 1;
    try {
        skedaddle()[index=2]++;
    } catch (Exception e) {
        System.out.println(e + ", index=\" + index);
    }
    }
}
static int[] skedaddle() throws Exception {
    throw new Exception("Ciao");
}

prints:
java.lang.Exception: Ciao, index=1

because the embedded assignment of 2 to index never occurs.

Example 15.10.4-3. null Array Reference

If the array reference expression produces null instead of a reference to an array, then a
NullPointerException is thrown at run time, but only after all parts of the array access
expression have been evaluated and only if these evaluations completed normally. Thus, the program:

class Test3 {
    public static void main(String[] args) {
        int index = 1;
        try {
            nada()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=\" + index);
        }
        }
    static int[] nada() { return null; }
}

prints:
java.lang.NullPointerException, index=2

because the embedded assignment of 2 to index occurs before the check for a null array
reference expression. As a related example, the program:

class Test4 {
    public static void main(String[] args) {
        int[] a = null;
        try {
            int i = a[vamoose()];
            System.out.println(i);
        }
    }
}
{ catch (Exception e) {
    System.out.println(e);
}
}

static int vamoose() throws Exception {
    throw new Exception("Twenty-three skidoo!");
}

always prints:

java.lang.Exception: Twenty-three skidoo!

A NullPointerException never occurs, because the index expression must be completely evaluated before any further part of the array access occurs, and that includes the check as to whether the value of the array reference expression is null.

15.11 Field Access Expressions

A field access expression may access a field of an object or array, a reference to which is the value of either an expression or the special keyword super.

FieldAccess:

Primary . Identifier
super . Identifier
TypeName . super . Identifier

The meaning of a field access expression is determined using the same rules as for qualified names (§6.5.6.2), but limited by the fact that an expression cannot denote a package, class type, or interface type.

It is also possible to refer to a field of the current instance or current class by using a simple name (§6.5.6.1).

15.11.1 Field Access Using a Primary

The type of the Primary must be a reference type $\tau$, or a compile-time error occurs. The meaning of the field access expression is determined as follows:

- If the identifier names several accessible (§6.6) member fields in type $\tau$, then the field access is ambiguous and a compile-time error occurs.
- If the identifier does not name an accessible member field in type $\tau$, then the field access is undefined and a compile-time error occurs.
• Otherwise, the identifier names a single accessible member field in type $\tau$, and the type of the field access expression is the type of the member field after capture conversion (§5.1.10).

At run time, the result of the field access expression is computed as follows: (assuming that the program is correct with respect to definite assignment analysis, that is, every blank final variable is definitely assigned before access)

• If the field is static:
  – The Primary expression is evaluated, and the result is discarded. If evaluation of the Primary expression completes abruptly, the field access expression completes abruptly for the same reason.
  – If the field is a non-blank final field, then the result is the value of the specified class variable in the class or interface that is the type of the Primary expression.
  – If the field is not final, or is a blank final and the field access occurs in a class variable initializer (§8.3.2) or static initializer (§8.7), then the result is a variable, namely, the specified class variable in the class that is the type of the Primary expression.

• If the field is not static:
  – The Primary expression is evaluated. If evaluation of the Primary expression completes abruptly, the field access expression completes abruptly for the same reason.
  – If the value of the Primary is null, then a NullPointerException is thrown.
  – If the field is a non-blank final, then the result is the value of the named member field in type $\tau$ found in the object referenced by the value of the Primary.
  – If the field is not final, or is a blank final and the field access occurs in an instance variable initializer (§8.3.2), instance initializer (§8.6), or constructor (§8.8), then the result is a variable, namely the named member field in type $\tau$ found in the object referenced by the value of the Primary.

Note that only the type of the Primary expression, not the class of the actual object referred to at run time, is used in determining which field to use.

**Example 15.11.1-1. Static Binding for Field Access**

```java
class S           { int x = 0; }
class T extends S { int x = 1; }
class Test1 {
```
15.11 Field Access Expressions

```
public static void main(String[] args) {
    T t = new T();
    System.out.println("t.x=" + t.x + when("t", t));
    S s = new S();
    System.out.println("s.x=" + s.x + when("s", s));
    s = t;
    System.out.println("s.x=" + s.x + when("s", s));
}
static String when(String name, Object t) {
    return " when " + name + " holds a " + t.getClass() + " at run time.";
}
```

This program produces the output:

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=0 when s holds a class T at run time.
```

The last line shows that, indeed, the field that is accessed does not depend on the run-time class of the referenced object; even if \( s \) holds a reference to an object of class \( T \), the expression \( s.x \) refers to the \( x \) field of class \( S \), because the type of the expression \( s \) is \( S \). Objects of class \( T \) contain two fields named \( x \), one for class \( T \) and one for its superclass \( S \).

This lack of dynamic lookup for field accesses allows programs to be run efficiently with straightforward implementations. The power of late binding and overriding is available, but only when instance methods are used. Consider the same example using instance methods to access the fields:

```
class S { int x = 0; int z() { return x; } }
class T extends S { int x = 1; int z() { return x; } }
class Test2 {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
        s = t;
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a " + t.getClass() + " at run time.";
    }
}
```

Now the output is:

```
t.z()=1 when t holds a class T at run time.
s.z()=0 when s holds a class S at run time.
s.z()=1 when s holds a class T at run time.
```
The last line shows that, indeed, the method that is accessed does depend on the run-time class of the referenced object; when \( s \) holds a reference to an object of class \( T \), the expression \( s.z() \) refers to the \( z \) method of class \( T \), despite the fact that the type of the expression \( s \) is \( S \). Method \( z \) of class \( T \) overrides method \( z \) of class \( S \).

**Example 15.11.1-2. Receiver Variable Is Irrelevant For static Field Access**

The following program demonstrates that a null reference may be used to access a class (static) variable without causing an exception:

```java
class Test3 {
    static String mountain = "Chocorua";
    static Test3 favorite() {
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }
}
```

It compiles, executes, and prints:

```
Mount Chocorua
```

Even though the result of \( \text{favorite()} \) is \( \text{null} \), a \( \text{NullPointerException} \) is not thrown. That "Mount " is printed demonstrates that the Primary expression is indeed fully evaluated at run time, despite the fact that only its type, not its value, is used to determine which field to access (because the field \( \text{mountain} \) is static).

### 15.11.2 Accessing Superclass Members using super

The form \( \text{super} . \text{Identifier} \) refers to the field named \( \text{Identifier} \) of the current object, but with the current object viewed as an instance of the superclass of the current class.

The form \( T . \text{super} . \text{Identifier} \) refers to the field named \( \text{Identifier} \) of the lexically enclosing instance corresponding to \( T \), but with that instance viewed as an instance of the superclass of \( T \).

The forms using the keyword \( \text{super} \) are valid only in an instance method, instance initializer, or constructor of a class, or in the initializer of an instance variable of a class. If they appear anywhere else, a compile-time error occurs.

These are exactly the same situations in which the keyword \( \text{this} \) may be used in a class declaration (§15.8.3).
It is a compile-time error if the forms using the keyword `super` appear in the declaration of class `Object`, since `Object` has no superclass.

Suppose that a field access expression `super.f` appears within class `C`, and the immediate superclass of `C` is class `S`. If `f` in `S` is accessible from class `C` (§6.6), then `super.f` is treated as if it had been the expression `this.f` in the body of class `S`. Otherwise, a compile-time error occurs.

Thus, `super.f` can access the field `f` that is accessible in class `S`, even if that field is hidden by a declaration of a field `f` in class `C`.

Suppose that a field access expression `T.super.f` appears within class `C`, and the immediate superclass of the class denoted by `T` is a class whose fully qualified name is `S`. If `f` in `S` is accessible from `C`, then `T.super.f` is treated as if it had been the expression `this.f` in the body of class `S`. Otherwise, a compile-time error occurs.

Thus, `T.super.f` can access the field `f` that is accessible in class `S`, even if that field is hidden by a declaration of a field `f` in class `T`.

It is a compile-time error if the current class is not an inner class of class `T` or `T` itself.

Example 15.11.2-1. The `super` Expression

```java
interface I
    { int x = 0; }
class T1 implements I { int x = 1; }
class T2 extends T1 { int x = 2; }
class T3 extends T2 {
    int x = 3;
    void test() {
        System.out.println("x=\t\t" + x);
        System.out.println("super.x=\t\t" + super.x);
        System.out.println("((T2)this).x=\t\t" + ((T2)this).x);
        System.out.println("((T1)this).x=\t\t" + ((T1)this).x);
        System.out.println("((I)this).x=\t\t" + ((I)this).x);
    }
}
class Test {
    public static void main(String[] args) {
        new T3().test();
    }
}
```

This program produces the output:

```
x=              3
super.x=        2
((T2)this).x=   2
((T1)this).x=   1
((I)this).x=    0
```
Within class $T_3$, the expression $\text{super}.x$ has the same effect as $((T_2)\text{this}).x$ when $x$ has package access. Note that $\text{super}.x$ is not specified in terms of a cast, due to difficulties around access to protected members of the superclass.

15.12 Method Invocation Expressions

A method invocation expression is used to invoke a class or instance method.

$\text{MethodInvocation:}$

\begin{align*}
\text{MethodName} & \ (\text{[ArgumentList]} \ ) \\
\text{TypeName} \ . \ \text{[TypeArguments]} \ \text{Identifier} & \ (\text{[ArgumentList]} \ ) \\
\text{ExpressionName} \ . \ \text{[TypeArguments]} \ \text{Identifier} & \ (\text{[ArgumentList]} \ ) \\
\text{Primary} \ . \ \text{[TypeArguments]} \ \text{Identifier} & \ (\text{[ArgumentList]} \ ) \\
\text{super} \ . \ \text{[TypeArguments]} \ \text{Identifier} & \ (\text{[ArgumentList]} \ ) \\
\text{TypeName} \ . \ \text{super} \ . \ \text{[TypeArguments]} \ \text{Identifier} & \ (\text{[ArgumentList]} \ ) 
\end{align*}

$\text{ArgumentList:}$

\begin{align*}
\text{Expression} & \ (, \ \text{Expression}) 
\end{align*}

Resolving a method name at compile time is more complicated than resolving a field name because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field because of the possibility of instance method overriding.

Determining the method that will be invoked by a method invocation expression involves several steps. The following three sections describe the compile-time processing of a method invocation. The determination of the type of the method invocation expression is specified in §15.12.3.

The exception types that a method invocation expression can throw are specified in §11.2.1.

It is a compile-time error if the name to the left of the rightmost "." that occurs before the ( in a MethodInvocation cannot be classified as a TypeName or an ExpressionName (§6.5.2).

If TypeArguments is present to the left of Identifier, then it is a compile-time error if any of the type arguments are wildcards (§4.5.1).

A method invocation expression is a poly expression if all of the following are true:

- The invocation appears in an assignment context or an invocation context (§5.2, §5.3).
• If the invocation is qualified (that is, any form of MethodInvocation except for the first), then the invocation elides TypeArguments to the left of the Identifier.

• The method to be invoked, as determined by the following subsections, is generic (§8.4.4) and has a return type that mentions at least one of the method’s type parameters.

Otherwise, the method invocation expression is a standalone expression.

15.12.1 Compile-Time Step 1: Determine Class or Interface to Search

The first step in processing a method invocation at compile time is to figure out the name of the method to be invoked and which class or interface to search for definitions of methods of that name.

The name of the method is specified by the MethodName or Identifier which immediately precedes the left parenthesis of the MethodInvocation.

For the class or interface to search, there are six cases to consider, depending on the form that precedes the left parenthesis of the MethodInvocation:

• If the form is MethodName, that is, just an Identifier, then:

  If the Identifier appears in the scope of a method declaration with that name (§6.3, §6.4.1), then:

  – If there is an enclosing type declaration of which that method is a member, let \( \tau \) be the innermost such type declaration. The class or interface to search is \( \tau \).

    This search policy is called the "comb rule". It effectively looks for methods in a nested class’s superclass hierarchy before looking for methods in an enclosing class and its superclass hierarchy. See §6.5.7.1 for an example.

  – Otherwise, the method declaration may be in scope due to one or more single-static-import or static-import-on-demand declarations. There is no class or interface to search, as the method to be invoked is determined later (§15.12.2.1).

• If the form is TypeName . [TypeArguments] Identifier, then the type to search is the type denoted by TypeName.

• If the form is ExpressionName . [TypeArguments] Identifier, then the class or interface to search is the declared type \( \tau \) of the variable denoted by ExpressionName if \( \tau \) is a class or interface type, or the upper bound of \( \tau \) if \( \tau \) is a type variable.
• If the form is `Primary . [TypeArguments] Identifier`, then let \( \tau \) be the type of the `Primary` expression. The class or interface to search is \( \tau \) if \( \tau \) is a class or interface type, or the upper bound of \( \tau \) if \( \tau \) is a type variable.

It is a compile-time error if \( \tau \) is not a reference type.

• If the form is `super . [TypeArguments] Identifier`, then the class to search is the superclass of the class whose declaration contains the method invocation.

Let \( \tau \) be the type declaration immediately enclosing the method invocation. It is a compile-time error if \( \tau \) is the class `Object` or \( \tau \) is an interface.

• If the form is `TypeName . super . [TypeArguments] Identifier`, then:
  – It is a compile-time error if `TypeName` denotes neither a class nor an interface.
  – If `TypeName` denote a class, \( c \), then the class to search is the superclass of \( c \).

    It is a compile-time error if \( c \) is not a lexically enclosing type declaration of the current class, or if \( c \) is the class `Object`.

    Let \( \tau \) be the type declaration immediately enclosing the method invocation. It is a compile-time error if \( \tau \) is the class `Object`.

    – Otherwise, `TypeName` denotes the interface to be searched, \( I \).

    Let \( \tau \) be the type declaration immediately enclosing the method invocation. It is a compile-time error if \( I \) is not a direct superinterface of \( \tau \), or if there exists some other direct superclass or direct superinterface of \( \tau \), \( J \), such that \( J \) is a subtype of \( I \).

The `TypeName . super` syntax is overloaded: traditionally, the `TypeName` refers to a lexically enclosing type declaration which is a class, and the target is the superclass of this class, as if the invocation were an unqualified `super` in the lexically enclosing type declaration.

class Superclass {
    void foo() { System.out.println("Hi"); }
}

class Subclass1 extends Superclass {
    void foo() { throw new UnsupportedOperationException(); }
}

Runnable tweak = new Runnable() {
    void run() {
        Subclass1.super.foo(); // Gets the 'println' behavior
    }
};
To support invocation of default methods in superinterfaces, the *TypeName* may also refer to a direct superinterface of the current class or interface, and the target is that superinterface.

```java
interface Superinterface {
    default void foo() { System.out.println("Hi"); }
}

class Subclass2 implements Superinterface {
    void foo() { throw new UnsupportedOperationException(); }

    void tweak() {
        Superinterface.super.foo(); // Gets the 'println' behavior
    }
}
```

No syntax supports a combination of these forms, that is, invoking a superinterface method of a lexically enclosing type declaration which is a class, as if the invocation were of the form *InterfaceName.super* in the lexically enclosing type declaration.

```java
class Subclass3 implements Superinterface {
    void foo() { throw new UnsupportedOperationException(); }

    Runnable tweak = new Runnable() {
        void run() {
            Subclass3.Superinterface.super.foo(); // Illegal
        }
    }
}
```

A workaround is to introduce a private method in the lexically enclosing type declaration, that performs the interface super call.

### 15.12.2 Compile-Time Step 2: Determine Method Signature

The second step searches the type determined in the previous step for member methods. This step uses the name of the method and the argument expressions to locate methods that are both accessible and applicable, that is, declarations that can be correctly invoked on the given arguments.

There may be more than one such method, in which case the most specific one is chosen. The descriptor (signature plus return type) of the most specific method is the one used at run time to perform the method dispatch.

Certain argument expressions that contain implicitly typed lambda expressions (§15.27.1) or inexact method references (§15.13.1) are ignored by the applicability tests, because their meaning cannot be determined until the invocation's target type is selected. On the other hand, it is only argument expressions - not the invocation's
target type - that influence the applicability tests, even if the method invocation expression is a poly expression.

The process of determining applicability begins by determining the potentially applicable methods (§15.12.2.1). Then, to ensure compatibility with the Java programming language prior to Java SE 5.0, the process continues in three phases:

1. The first phase performs overload resolution without permitting boxing or unboxing conversion, or the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the second phase.

   This guarantees that any calls that were valid in the Java programming language before Java SE 5.0 are not considered ambiguous as the result of the introduction of variable arity methods, implicit boxing and/or unboxing. However, the declaration of a variable arity method (§8.4.1) can change the method chosen for a given method invocation expression, because a variable arity method is treated as a fixed arity method in the first phase. For example, declaring \texttt{m(Object...)} in a class which already declares \texttt{m(Object)} causes \texttt{m(Object)} to no longer be chosen for some invocation expressions (such as \texttt{m(null)}), as \texttt{m(Object[])} is more specific.

2. The second phase performs overload resolution while allowing boxing and unboxing, but still precludes the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the third phase.

   This ensures that a method is never chosen through variable arity method invocation if it is applicable through fixed arity method invocation.

3. The third phase allows overloading to be combined with variable arity methods, boxing, and unboxing.

   A method is applicable if it is applicable by one of strict invocation (the first phase, §15.12.2.2), loose invocation (the second phase, §15.12.2.3), or variable arity invocation (the third phase, §15.12.2.4). Deciding whether a method is applicable will, in the case of generic methods (§8.4.4), require an analysis of the type arguments. Type arguments may be passed explicitly or implicitly; if they are passed implicitly, then bounds of the type arguments must be inferred from the argument expressions (§18 (Type Inference)).

   If several applicable methods have been identified during one of the three phases of applicability testing, then the most specific one is chosen, as specified in section §15.12.2.5.

   To check for applicability, the types of an invocation's arguments cannot, in general, be inputs to the analysis. This is because:
• The arguments to a method invocation may be poly expressions.
• Poly expressions cannot be typed in the absence of a target type.
• Overload resolution has to be completed before the arguments’ target types will be known.

Instead, the input to the applicability check is a list of the arguments themselves. The arguments can be checked for compatibility with potential target types, even if the ultimate types of the arguments are unknown.

Note that overload resolution is independent of a target type. This is for two reasons:

• First, it makes the user model more accessible and less error-prone. The meaning of a method name (i.e., the declaration corresponding to the name) is too fundamental to the meaning of a program to depend on subtle contextual hints. (In contrast, other poly expressions may have different behavior depending on a target type; but the variation in behavior is always limited and essentially equivalent, while no such guarantees can be made about the behavior of an arbitrary set of methods that share a name and arity.)
• Second, it allows other properties - such as whether or not the method is a poly expression (§15.12) or how to categorize a conditional expression (§15.25) - to depend on the meaning of the method name, even before a target type is known.

Example 15.12.2-1. Method Applicability

class Doubler {
    static int two() { return two(1); }
    private static int two(int i) { return 2*i; }
}
class Test extends Doubler {
    static long two(long j) { return j+j; }
}

public static void main(String[] args) {
    System.out.println(two(3));
    System.out.println(Doubler.two(3)); // compile-time error
}

For the method invocation two(1) within class Doubler, there are two accessible methods named two, but only the second one is applicable, and so that is the one invoked at run time.

For the method invocation two(3) within class Test, there are two applicable methods, but only the one in class Test is accessible, and so that is the one to be invoked at run time (the argument 3 is converted to type long).

For the method invocation Doubler.two(3), the class Doubler, not class Test, is searched for methods named two; the only applicable method is not accessible, and so this method invocation causes a compile-time error.

Another example is:
class ColoredPoint {
    int x, y;
    byte color;
    void setColor(byte color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37);  // compile-time error
    }
}

Here, a compile-time error occurs for the second invocation of `setColor`, because no applicable method can be found at compile time. The type of the literal 37 is int, and int cannot be converted to byte by invocation conversion. Assignment conversion, which is used in the initialization of the variable color, performs an implicit conversion of the constant from type int to byte, which is permitted because the value 37 is small enough to be represented in type byte; but such a conversion is not allowed for invocation conversion.

If the method `setColor` had, however, been declared to take an int instead of a byte, then both method invocations would be correct; the first invocation would be allowed because invocation conversion does permit a widening conversion from byte to int. However, a narrowing cast would then be required in the body of `setColor`:

```java
void setColor(int color) { this.color = (byte)color; }
```

Here is an example of overloading ambiguity. Consider the program:

```java
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static void test(ColoredPoint p, Point q) {
        System.out.println("(ColoredPoint, Point)");  
    }
    static void test(Point p, ColoredPoint q) {
        System.out.println("(Point, ColoredPoint)");  
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        test(cp, cp);  // compile-time error
    }
}
```

This example produces an error at compile time. The problem is that there are two declarations of `test` that are applicable and accessible, and neither is more specific than the other. Therefore, the method invocation is ambiguous.

If a third definition of `test` were added:
then it would be more specific than the other two, and the method invocation would no longer be ambiguous.

**Example 15.12.2-2. Return Type Not Considered During Method Selection**

```java
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static int test(ColoredPoint p) {
        return p.color;
    }
    static String test(Point p) {
        return "Point";
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        String s = test(cp);  // compile-time error
    }
}
```

Here, the most specific declaration of method `test` is the one taking a parameter of type `ColoredPoint`. Because the result type of the method is `int`, a compile-time error occurs because an `int` cannot be converted to a `String` by assignment conversion. This example shows that the result types of methods do not participate in resolving overloaded methods, so that the second `test` method, which returns a `String`, is not chosen, even though it has a result type that would allow the example program to compile without error.

**Example 15.12.2-3. Choosing The Most Specific Method**

The most specific method is chosen at compile time; its descriptor determines what method is actually executed at run time. If a new method is added to a class, then source code that was compiled with the old definition of the class might not use the new method, even if a recompilation would cause this method to be chosen.

So, for example, consider two compilation units, one for class `Point`:

```java
package points;
public class Point {
    public int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return toString(""); }
    public String toString(String s) {
        return "(" + x + "," + y + s + ")";
    }
}
```

and one for class `ColoredPoint`:

```java
package points;
public class ColoredPoint { static void test(ColoredPoint p, ColoredPoint q) {
    System.out.println("(ColoredPoint, ColoredPoint)");
}
```


package points;
public class ColoredPoint extends Point {
    public static final int
        RED = 0, GREEN = 1, BLUE = 2;
    public static String[] COLORS =
        { "red", "green", "blue" };

    public byte color;
    public ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = (byte)color;
    }

    /** Copy all relevant fields of the argument into
     * this ColoredPoint object. */
    public void adopt(Point p) { x = p.x; y = p.y; }

    public String toString() {
        String s ="," + COLORS[color];
        return super.toString(s);
    }
}

Now consider a third compilation unit that uses ColoredPoint:

import points.*;
class Test {
    public static void main(String[] args) {
        ColoredPoint cp =
            new ColoredPoint(6, 6, ColoredPoint.RED);
        ColoredPoint cp2 =
            new ColoredPoint(3, 3, ColoredPoint.GREEN);
        cp.adopt(cp2);
        System.out.println("cp: " + cp);
    }
}

The output is:

    cp: (3,3,red)

The programmer who coded class Test has expected to see the word green, because the actual argument, a ColoredPoint, has a color field, and color would seem to be a "relevant field". (Of course, the documentation for the package points ought to have been much more precise!)

Notice, by the way, that the most specific method (indeed, the only applicable method) for the method invocation of adopt has a signature that indicates a method of one parameter, and the parameter is of type Point. This signature becomes part of the binary representation of class Test produced by the Java compiler and is used by the method invocation at run time.
Suppose the programmer reported this software error and the maintainer of the `points` package decided, after due deliberation, to correct it by adding a method to class `ColoredPoint`:

```java
public void adopt(ColoredPoint p) {
    adopt((Point)p);
    color = p.color;
}
```

If the programmer then runs the old binary file for `Test` with the new binary file for `ColoredPoint`, the output is still:

```
 cp: (3,3,red)
```

because the old binary file for `Test` still has the descriptor "one parameter, whose type is `Point`; void" associated with the method call `cp.adopt(cp2)`. If the source code for `Test` is recompiled, the Java compiler will then discover that there are now two applicable `adopt` methods, and that the signature for the more specific one is "one parameter, whose type is `ColoredPoint`; void"; running the program will then produce the desired output:

```
 cp: (3,3,green)
```

With forethought about such problems, the maintainer of the `points` package could fix the `ColoredPoint` class to work with both newly compiled and old code, by adding defensive code to the old `adopt` method for the sake of old code that still invokes it on `ColoredPoint` arguments:

```java
public void adopt(Point p) {
    if (p instanceof ColoredPoint)
        color = ((ColoredPoint)p).color;
    x = p.x; y = p.y;
}
```

Ideally, source code should be recompiled whenever code that it depends on is changed. However, in an environment where different classes are maintained by different organizations, this is not always feasible. Defensive programming with careful attention to the problems of class evolution can make upgraded code much more robust. See §13 (`Binary Compatibility`) for a detailed discussion of binary compatibility and type evolution.

### 15.12.2.1 Identify Potentially Applicable Methods

The class or interface determined by compile-time step 1 (§15.12.1) is searched for all member methods that are potentially applicable to this method invocation; members inherited from superclasses and superinterfaces are included in this search.

In addition, if the form of the method invocation expression is `MethodName` - that is, a single `Identifier` - then the search for potentially applicable methods also examines all member methods that are imported by single-static-import
declarations and static-import-on-demand declarations of the compilation unit where the method invocation occurs (§7.5.3, §7.5.4) and that are not shadowed at the point where the method invocation appears.

A member method is potentially applicable to a method invocation if and only if all of the following are true:

- The name of the member is identical to the name of the method in the method invocation.
- The member is accessible (§6.6) to the class or interface in which the method invocation appears.

Whether a member method is accessible at a method invocation depends on the access modifier (public, protected, no modifier (package access), or private) in the member’s declaration, and on the inheritance of the member by the class or interface determined by compile-time step 1, and on where the method invocation appears.

- If the member is a fixed arity method with arity $n$, the arity of the method invocation is equal to $n$, and for all $i$ ($1 \leq i \leq n$), the $i$th argument of the method invocation is potentially compatible, as defined below, with the type of the $i$th parameter of the method.

- If the member is a variable arity method with arity $n$, then for all $i$ ($1 \leq i \leq n-1$), the $i$th argument of the method invocation is potentially compatible with the type of the $i$th parameter of the method; and, where the $n$th parameter of the method has type $T[]$, one of the following is true:
  - The arity of the method invocation is equal to $n-1$.
  - The arity of the method invocation is equal to $n$, and the $n$th argument of the method invocation is potentially compatible with either $T$ or $T[]$.
  - The arity of the method invocation is $m$, where $m > n$, and for all $i$ ($n \leq i \leq m$), the $i$th argument of the method invocation is potentially compatible with $T$.

- If the method invocation includes explicit type arguments, and the member is a generic method, then the number of type arguments is equal to the number of type parameters of the method.

This clause implies that a non-generic method may be potentially applicable to an invocation that supplies explicit type arguments. Indeed, it may turn out to be applicable. In such a case, the type arguments will simply be ignored.

This rule stems from issues of compatibility and principles of substitutability. Since interfaces or superclasses may be generified independently of their subtypes, we may override a generic method with a non-generic one. However, the overriding (non-generic) method must be applicable to calls to the generic method, including calls that
explicitly pass type arguments. Otherwise the subtype would not be substitutable for its generified supertype.

If the search does not yield at least one method that is potentially applicable, then a compile-time error occurs.

An expression is *potentially compatible* with a target type according to the following rules:

- A lambda expression (§15.27) is potentially compatible with a functional interface type \( \tau \) (§9.8) if all of the following are true:
  - The arity of the function type of \( \tau \) (§9.9) is the same as the arity of the lambda expression.
  - If the function type of \( \tau \) has a void return, then the lambda body is either a statement expression (§14.8) or a void-compatible block (§15.27.2).
  - If the function type of \( \tau \) has a (non-void) return type, then the lambda body is either an expression or a value-compatible block (§15.27.2).

- A method reference expression (§15.13) is potentially compatible with a functional interface type \( \tau \) if, where the arity of the function type of \( \tau \) is \( n \), there exists at least one potentially applicable method when the method reference expression targets the function type with arity \( n \) (§15.13.1), and one of the following is true:
  - The method reference expression has the form `ReferenceType :: [TypeArguments] Identifier` and at least one potentially applicable method is either (i) static and supports arity \( n \), or (ii) not static and supports arity \( n-1 \).
  - The method reference expression has some other form and at least one potentially applicable method is not static.

- A lambda expression or a method reference expression is potentially compatible with a type variable if the type variable is a type parameter of the candidate method.

- A parenthesized expression (§15.8.5) is potentially compatible with a type if its contained expression is potentially compatible with that type.

- A conditional expression (§15.25) is potentially compatible with a type if each of its second and third operand expressions are potentially compatible with that type.

- A class instance creation expression, a method invocation expression, or an expression of a standalone form (§15.2) is potentially compatible with any type.
The definition of potential applicability goes beyond a basic arity check to also take into account the presence and "shape" of functional interface target types. In some cases involving type argument inference, a lambda expression appearing as a method invocation argument cannot be properly typed until after overload resolution. These rules allow the form of the lambda expression to still be taken into account, discarding obviously incorrect target types that might otherwise cause ambiguity errors.

15.12.2.2 Phase 1: Identify Matching Arity Methods Applicable by Strict Invocation

An argument expression is considered pertinent to applicability for a potentially applicable method \( m \) unless it has one of the following forms:

- An implicitly typed lambda expression (§15.27.1).
- If \( m \) is a generic method and the method invocation does not provide explicit type arguments, an explicitly typed lambda expression or an exact method reference expression for which the corresponding target type (as derived from the signature of \( m \)) is a type parameter of \( m \).
- An explicitly typed lambda expression whose body is an expression that is not pertinent to applicability.
- An explicitly typed lambda expression whose body is a block, where at least one result expression is not pertinent to applicability.
- A parenthesized expression (§15.8.5) whose contained expression is not pertinent to applicability.
- A conditional expression (§15.25) whose second or third operand is not pertinent to applicability.

Let \( m \) be a potentially applicable method (§15.12.2.1) with arity \( n \) and formal parameter types \( F_1 \ldots F_n \), and let \( e_1, \ldots, e_n \) be the actual argument expressions of the method invocation. Then:

- If \( m \) is a generic method and the method invocation does not provide explicit type arguments, then the applicability of the method is inferred as specified in §18.5.1.
- If \( m \) is a generic method and the method invocation provides explicit type arguments, then let \( R_1 \ldots R_p \) \( (p \geq 1) \) be the type parameters of \( m \), let \( B_l \) be the declared bound of \( R_l \) \( (1 \leq l \leq p) \), and let \( U_1, \ldots, U_p \) be the explicit type arguments given in the method invocation. Then \( m \) is applicable by strict invocation if both of the following are true:

---

541
- For $1 \leq i \leq n$, if $e_i$ is pertinent to applicability then $e_i$ is compatible in a strict invocation context with $F_i[R_1:=U_1, \ldots, R_p:=U_p]$ (§5.3).

- For $1 \leq l \leq p$, $U_1 <: B_1[R_1:=U_1, \ldots, R_p:=U_p]$.

• If $m$ is not a generic method, then $m$ is applicable by strict invocation if, for $1 \leq i \leq n$, either $e_i$ is compatible in a strict invocation context with $F_i$ (§5.3) or $e_i$ is not pertinent to applicability.

If no method applicable by strict invocation is found, the search for applicable methods continues with phase 2 (§15.12.2.3).

Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by strict invocation.

The meaning of an implicitly typed lambda expression or an inexact method reference expression is sufficiently vague prior to resolving a target type that arguments containing these expressions are not considered pertinent to applicability; they are simply ignored (except for their expected arity) until overload resolution is finished.

15.12.2.3 Phase 2: Identify Matching Arity Methods Applicable by Loose Invocation

Let $m$ be a potentially applicable method (§15.12.2.1) with arity $n$ and formal parameter types $F_1, \ldots, F_n$, and let $e_1, \ldots, e_n$ be the actual argument expressions of the method invocation. Then:

• If $m$ is a generic method and the method invocation does not provide explicit type arguments, then the applicability of the method is inferred as specified in §18.5.1.

• If $m$ is a generic method and the method invocation provides explicit type arguments, then let $R_1 \ldots R_p$ ($p \geq 1$) be the type parameters of $m$, let $B_1$ be the declared bound of $R_1$ ($1 \leq l \leq p$), and let $U_1 \ldots U_p$ be the explicit type arguments given in the method invocation. Then $m$ is applicable by loose invocation if both of the following are true:

  – For $1 \leq i \leq n$, if $e_i$ is pertinent to applicability (§15.12.2.2) then $e_i$ is compatible in a loose invocation context with $F_i[R_1:=U_1, \ldots, R_p:=U_p]$ (§5.3).

  – For $1 \leq l \leq p$, $U_1 <: B_1[R_1:=U_1, \ldots, R_p:=U_p]$.

• If $m$ is not a generic method, then $m$ is applicable by loose invocation if, for $1 \leq i \leq n$, either $e_i$ is compatible in a loose invocation context with $F_i$ (§5.3) or $e_i$ is not pertinent to applicability.

If no method applicable by loose invocation is found, the search for applicable methods continues with phase 3 (§15.12.2.4).
Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by loose invocation.

15.12.2.4 Phase 3: Identify Methods Applicable by Variable Arity Invocation

Where a variable arity method has formal parameter types $F_1, ..., F_{n-1}, F_n$, let the $i$th variable arity parameter type of the method be defined as follows:

- For $i \leq n-1$, the $i$th variable arity parameter type is $F_i$.
- For $i \geq n$, the $i$th variable arity parameter type is $F_n$.

Let $m$ be a potentially applicable method (§15.12.2.1) with variable arity, let $T_1, ..., T_k$ be the first $k$ variable arity parameter types of $m$, and let $e_1, ..., e_k$ be the actual argument expressions of the method invocation. Then:

- If $m$ is a generic method and the method invocation does not provide explicit type arguments, then the applicability of the method is inferred as specified in §18.5.1.
- If $m$ is a generic method and the method invocation provides explicit type arguments, then let $R_1, ..., R_p$ ($p \geq 1$) be the type parameters of $m$, let $B_1$ be the declared bound of $R_l$ ($1 \leq l \leq p$), and let $U_1, ..., U_p$ be the explicit type arguments given in the method invocation. Then $m$ is applicable by variable arity invocation if:
  - For $1 \leq i \leq k$, if $e_i$ is pertinent to applicability (§15.12.2.2) then $e_i$ is compatible in a loose invocation context with $T_i[R_1:=U_1, ..., R_p:=U_p]$ (§5.3).
  - For $1 \leq l \leq p$, $U_l <: B_1[R_1:=U_1, ..., R_p:=U_p]$.
- If $m$ is not a generic method, then $m$ is applicable by variable arity invocation if, for $1 \leq i \leq k$, either $e_i$ is compatible in a loose invocation context with $T_i$ (§5.3) or $e_i$ is not pertinent to applicability.

If no method applicable by variable arity invocation is found, then a compile-time error occurs.

Otherwise, the most specific method (§15.12.2.5) is chosen among the methods applicable by variable arity invocation.

15.12.2.5 Choosing the Most Specific Method

If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the runtime method dispatch. The Java programming language uses the rule that the most specific method is chosen.
The informal intuition is that one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time error. In cases such as an explicitly typed lambda expression argument (§15.27.1) or a variable arity invocation (§15.12.2.4), some flexibility is allowed to adapt one signature to the other.

One applicable method \( m_1 \) is more specific than another applicable method \( m_2 \), for an invocation with argument expressions \( e_1, \ldots, e_k \), if any of the following are true:

- \( m_2 \) is generic, and \( m_1 \) is inferred to be more specific than \( m_2 \) for argument expressions \( e_1, \ldots, e_k \) by §18.5.4.
- \( m_2 \) is not generic, and \( m_1 \) and \( m_2 \) are applicable by strict or loose invocation, and where \( m_1 \) has formal parameter types \( S_1, \ldots, S_n \) and \( m_2 \) has formal parameter types \( T_1, \ldots, T_n \), the type \( S_1 \) is more specific than \( T_1 \) for argument \( e_1 \) for all \( i \) (\( 1 \leq i \leq n, n = k \)).
- \( m_2 \) is not generic, and \( m_1 \) and \( m_2 \) are applicable by variable arity invocation, and where the first \( k \) variable arity parameter types of \( m_1 \) are \( S_1, \ldots, S_k \) and the first \( k \) variable arity parameter types of \( m_2 \) are \( T_1, \ldots, T_k \), the type \( S_1 \) is more specific than \( T_1 \) for argument \( e_1 \) for all \( i \) (\( 1 \leq i \leq k \)). Additionally, if \( m_2 \) has \( k+1 \) parameters, then the \( k+1 \)'th variable arity parameter type of \( m_1 \) is a subtype of the \( k+1 \)'th variable arity parameter type of \( m_2 \).

The above conditions are the only circumstances under which one method may be more specific than another.

A type \( S \) is more specific than a type \( T \) for any expression if \( S <: T \) (§4.10).

A functional interface type \( S \) is more specific than a functional interface type \( T \) for an expression \( e \) if all of the following are true:

- The interface of \( S \) is neither a superinterface nor a subinterface of the interface of \( T \).

If \( S \) or \( T \) is an intersection type, it is not the case that any interface of \( S \) is a superinterface or a subinterface of any interface of \( T \). (The "interfaces of" an intersection type refers here to the set of interfaces that appear as (possibly parameterized) interface types in the intersection.)

- Let \( MT_S \) be the function type of the capture of \( S \), and let \( MT_T \) be the function type of \( T \). \( MT_S \) and \( MT_T \) must have the same type parameters (if any) (§8.4.4).

- Let \( P_1, \ldots, P_n \) be the formal parameter types of \( MT_S \), adapted to the type parameters of \( MT_T \). Let \( P_1', \ldots, P_n' \) be the formal parameter types of the function type of \( S \) (without capture), adapted to the type parameters of \( MT_T \). Let \( Q_1, \ldots, Q_n \) be the formal parameter types of \( MT_T \). Then, for all \( i \) (\( 1 \leq i \leq n \)), \( Q_i <: P_i \) and \( Q_i = P_i' \).
Generally, this rule asserts that the formal parameter types derived from $S$ and $T$ are the same. But in the case in which $S$ is a wildcard-parameterized type, the check is more complex in order to allow capture variables to occur in formal parameter types: first, each formal parameter type of $T$ must be a subtype of the corresponding formal parameter type of the capture of $S$; second, after mapping the wildcards to their bounds (§9.9), the formal parameter types of the resulting function types are the same.

- Let $R_S$ be the return type of $MT_S$, adapted to the type parameters of $MT_T$, and let $R_T$ be the return type of $MT_T$. One of the following must be true:
  
  - $e$ is an explicitly typed lambda expression (§15.27.1), and one of the following is true:
    - $R_T$ is `void`.
    - $R_S <: R_T$.
    - $R_S$ and $R_T$ are functional interface types, and there is at least one result expression, and $R_S$ is more specific than $R_T$ for each result expression of $e$.
    
    The result expression of a lambda expression with a block body is defined in §15.27.2; the result expression of a lambda expression with an expression body is simply the body itself.
    - $R_S$ is a primitive type, and $R_T$ is a reference type, and there is at least one result expression, and each result expression of $e$ is a standalone expression (§15.2) of a primitive type.
    - $R_S$ is a reference type, and $R_T$ is a primitive type, and there is at least one result expression, and each result expression of $e$ is either a standalone expression of a reference type or a poly expression.
  
  - $e$ is an exact method reference expression (§15.13.1), and one of the following is true:
    - $R_T$ is `void`.
    - $R_S <: R_T$.
    - $R_S$ is a primitive type, $R_T$ is a reference type, and the compile-time declaration for the method reference has a return type which is a primitive type.
    - $R_S$ is a reference type, $R_T$ is a primitive type, and the compile-time declaration for the method reference has a return type which is a reference type.
  
  - $e$ is a parenthesized expression, and one of these conditions applies recursively to the contained expression.
– $e$ is a conditional expression, and, for each of the second and third operands, one of these conditions applies recursively.

A method $m_1$ is strictly more specific than another method $m_2$ if and only if $m_1$ is more specific than $m_2$ and $m_2$ is not more specific than $m_1$.

A method is said to be maximally specific for a method invocation if it is accessible and applicable and there is no other method that is accessible and applicable that is strictly more specific.

If there is exactly one maximally specific method, then that method is in fact the most specific method; it is necessarily more specific than any other accessible method that is applicable. It is then subjected to some further compile-time checks as specified in §15.12.3.

It is possible that no method is the most specific, because there are two or more methods that are maximally specific. In this case:

- If all the maximally specific methods have override-equivalent signatures (§8.4.2), and exactly one of the maximally specific methods is concrete (that is, neither abstract nor default), then it is the most specific method.

- Otherwise, if all the maximally specific methods have override-equivalent signatures, and all the maximally specific methods are abstract or default, and the declarations of these methods have the same erased parameter types, and at least one maximally specific method is preferred according to the rules below, then the most specific method is chosen arbitrarily among the subset of the maximally specific methods that are preferred. The most specific method is then considered to be abstract.

A maximally specific method is preferred if it has:

– a signature that is a subsignature of every maximally specific method's signature; and

– a return type $R$ (possibly void), where either $R$ is the same as every maximally specific method's return type, or $R$ is a reference type and is a subtype of every maximally specific method's return type (after adapting for any type parameters (§8.4.4) if the two methods have the same signature).

If no preferred method exists according to the above rules, then a maximally specific method is preferred if it:

– has a signature that is a subsignature of every maximally specific method's signature; and

– is return-type-substitutable (§8.4.5) for every maximally specific method.
The thrown exception types of the most specific method are derived from the `throws` clauses of the maximally specific methods, as follows:

1. If the most specific method is generic, the `throws` clauses are first adapted to the type parameters of the most specific method (§8.4.4).

   If the most specific method is not generic but at least one maximally specific method is generic, the `throws` clauses are first erased.

2. Then, the thrown exception types include every type \( E \) which satisfies the following constraints:
   - \( E \) is mentioned in one of the `throws` clauses.
   - For each `throws` clause, \( E \) is a subtype of some type named in that clause.

   These rules for deriving a single method type from a group of overloaded methods are also used to identify the function type of a functional interface (§9.9).

- Otherwise, the method invocation is *ambiguous*, and a compile-time error occurs.

### 15.12.2.6 Method Invocation Type

The *invocation type* of a most specific accessible and applicable method is a method type (§8.2) which expresses the target types of the invocation arguments, the result (return type or `void`) of the invocation, and the exception types of the invocation. It is determined as follows:

- If the chosen method is generic and the method invocation does not provide explicit type arguments, the invocation type is inferred as specified in §18.5.2.

  In this case, if the method invocation expression is a poly expression, then its compatibility with a target type is as determined by §18.5.2.1.

  Testing for compatibility with a target type may occur multiple times before making a final determination of the method invocation expression's target type and invocation type. For example, an enclosing method invocation expression may require testing the deeper method invocation expression for compatibility with different methods' formal parameter types.

- If the chosen method is generic and the method invocation provides explicit type arguments, let \( p_i \) be the type parameters of the method and let \( t_i \) be the explicit type arguments provided for the method invocation (\( 1 \leq i \leq p \)). Then:

  - If unchecked conversion was necessary for the method to be applicable, then the invocation type's parameter types are obtained by applying the substitution \([p_1 := t_1, \ldots, p_p := t_p]\) to the parameter types of the method's type, and the
Method Invocation Expressions

invocation type's return type and thrown types are given by the erasure of the return type and thrown types of the method's type.

- If unchecked conversion was not necessary for the method to be applicable, then the invocation type is obtained by applying the substitution \( P_1 := T_1, ..., P_p := T_p \) to the method's type.

- If the chosen method is not generic, then:

  - If unchecked conversion was necessary for the method to be applicable, the parameter types of the invocation type are the parameter types of the method's type, and the return type and thrown types are given by the erasures of the return type and thrown types of the method's type.

  - Otherwise, if the chosen method is the `getClass` method of the class `Object` (§4.3.2), the invocation type is the same as the method's type, except that the return type is `Class<T extends |T>`, where \( T \) is the type that was searched, as determined by §15.12.1, and |\( T \)| denotes the erasure of \( T \) (§4.6).

- Otherwise, the invocation type is the same as the method's type.

15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate?

If there is a most specific method declaration for a method invocation, it is called the compile-time declaration for the method invocation.

It is a compile-time error if an argument to a method invocation is not compatible with its target type, as derived from the invocation type of the compile-time declaration.

If the compile-time declaration is applicable by variable arity invocation, then where the last formal parameter type of the invocation type of the method is \( F_n[] \), it is a compile-time error if the type which is the erasure of \( F_n \) is not accessible (§6.6) at the point of invocation.

If the compile-time declaration is `void`, then the method invocation must be a top level expression (that is, the `Expression` in an expression statement or in the `ForInit` or `ForUpdate` part of a `for` statement), or a compile-time error occurs. Such a method invocation produces no value and so must be used only in a situation where a value is not needed.

In addition, whether the compile-time declaration is appropriate may depend on the form of the method invocation expression before the left parenthesis, as follows:

- If the form is `MethodName` - that is, just an `Identifier` - and the compile-time declaration is an instance method, then:
It is a compile-time error if the method invocation occurs in a static context (§8.1.3).

Otherwise, let \( C \) be the immediately enclosing class of which the compile-time declaration is a member. If the method invocation is not directly enclosed by \( C \) or an inner class of \( C \), then a compile-time error occurs.

- If the form is \( \text{TypeName} . \ [\text{TypeArguments}] \text{Identifier} \), then the compile-time declaration must be \textit{static}, or a compile-time error occurs.

- If the form is \( \text{ExpressionName} . \ [\text{TypeArguments}] \text{Identifier} \) or \( \text{Primary} . \ [\text{TypeArguments}] \text{Identifier} \), then the compile-time declaration must not be a \textit{static} method declared in an interface, or a compile-time error occurs.

- If the form is \( \text{super} . \ [\text{TypeArguments}] \text{Identifier} \), then:
  - It is a compile-time error if the compile-time declaration is \textit{abstract}.
  - It is a compile-time error if the method invocation occurs in a static context.

- If the form is \( \text{TypeName} . \text{super} . \ [\text{TypeArguments}] \text{Identifier} \), then:
  - It is a compile-time error if the compile-time declaration is \textit{abstract}.
  - It is a compile-time error if the method invocation occurs in a static context.
  - If \( \text{TypeName} \) denotes a class \( C \), then if the method invocation is not directly enclosed by \( C \) or an inner class of \( C \), a compile-time error occurs.

  - If \( \text{TypeName} \) denotes an interface, let \( \tau \) be the type declaration immediately enclosing the method invocation. A compile-time error occurs if there exists a method, distinct from the compile-time declaration, that overrides (§9.4.1) the compile-time declaration from a direct superclass or direct superinterface of \( \tau \).

In the case that a superinterface overrides a method declared in a grandparent interface, this rule prevents the child interface from “skipping” the override by simply adding the grandparent to its list of direct superinterfaces. The appropriate way to access functionality of a grandparent is through the direct superinterface, and only if that interface chooses to expose the desired behavior. (Alternately, the programmer is free to define an additional superinterface that exposes the desired behavior with a \textit{super} method invocation.)

The \textit{compile-time parameter types} and \textit{compile-time result} are determined as follows:

- If the compile-time declaration for the method invocation is \textit{not} a signature polymorphic method, then:
The compile-time parameter types are the types of the formal parameters of the compile-time declaration.

The compile-time result is the result of the invocation type of the compile-time declaration (§15.12.2.6).

- If the compile-time declaration for the method invocation is a signature polymorphic method, then:
  
  - The compile-time parameter types are the types of the actual argument expressions. An argument expression which is the null literal `null` (§3.10.7) is treated as having the type `Void`.
  
  - The compile-time result is determined as follows:
    - If the signature polymorphic method is either `void` or has a return type other than `Object`, the compile-time result is the result of the invocation type of the compile-time declaration (§15.12.2.6).
    - Otherwise, if the method invocation expression is an expression statement, the compile-time result is `void`.
    - Otherwise, if the method invocation expression is the operand of a cast expression (§15.16), the compile-time result is the erasure of the type of the cast expression (§4.6).
    - Otherwise, the compile-time result is the signature polymorphic method's return type, `Object`.

A method is signature polymorphic if all of the following are true:

- It is declared in the `java.lang.invoke.MethodHandle` class or the `java.lang.invoke.VarHandle` class.
- It has a single variable arity parameter (§8.4.1) whose declared type is `Object[]`.
- It is native.

The following compile-time information is then associated with the method invocation for use at run time:

- The name of the method.
- The qualifying type of the method invocation (§13.1).
- The number of parameters and the compile-time parameter types, in order.
- The compile-time result.
- The invocation mode, computed as follows:
If the compile-time declaration has the static modifier, then the invocation mode is static.

– Otherwise, if the part of the method invocation before the left parenthesis is of the form super . Identifier or of the form TypeName . super . Identifier, then the invocation mode is super.

– Otherwise, if the qualifying type of the method invocation is an interface, then the invocation mode is interface.

– Otherwise, the invocation mode is virtual.

If the result of the invocation type of the compile-time declaration is not void, then the type of the method invocation expression is obtained by applying capture conversion (§5.1.10) to the return type of the invocation type of the compile-time declaration.

15.12.4 Run-Time Evaluation of Method Invocation

At run time, method invocation requires five steps. First, a target reference may be computed. Second, the argument expressions are evaluated. Third, the accessibility of the method to be invoked is checked. Fourth, the actual code for the method to be executed is located. Fifth, a new activation frame is created, synchronization is performed if necessary, and control is transferred to the method code.

15.12.4.1 Compute Target Reference (If Necessary)

There are six cases to consider, depending on the form of the method invocation:

• If the form is MethodName - that is, just an Identifier - then:
  – If the invocation mode is static, then there is no target reference.
  – Otherwise, let \( \tau \) be the enclosing type declaration of which the method is a member, and let \( n \) be an integer such that \( \tau \) is the \( n \)'th lexically enclosing type declaration of the class whose declaration immediately contains the method invocation. The target reference is the \( n \)'th lexically enclosing instance of this. It is a compile-time error if the \( n \)'th lexically enclosing instance of this does not exist.

• If the form is TypeName . [TypeArguments] Identifier, then there is no target reference.
• If form is ExpressionName . [TypeArguments] Identifier, then:
  – If the invocation mode is static, then there is no target reference. The ExpressionName is evaluated, but the result is then discarded.
  – Otherwise, the target reference is the value denoted by ExpressionName.

• If the form is Primary . [TypeArguments] Identifier involved, then:
  – If the invocation mode is static, then there is no target reference. The Primary expression is evaluated, but the result is then discarded.
  – Otherwise, the Primary expression is evaluated and the result is used as the target reference.

In either case, if the evaluation of the Primary expression completes abruptly, then no part of any argument expression appears to have been evaluated, and the method invocation completes abruptly for the same reason.

• If the form is super . [TypeArguments] Identifier, then the target reference is the value of this.

• If the form is TypeName . super . [TypeArguments] Identifier, then if TypeName denotes a class, the target reference is the value of TypeName . this; otherwise, the target reference is the value of this.

Example 15.12.4.1-1. Target References and static Methods

When a target reference is computed and then discarded because the invocation mode is static, the reference is not examined to see whether it is null:

```java
class Test1 {
    static void mountain() {
        System.out.println("Monadnock");
    }
    static Test1 favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        favorite().mountain();
    }
}
```

which prints:

```
Mount Monadnock
```

Here favorite() returns null, yet no NullPointerException is thrown.
Example 15.12.4.1-2. Evaluation Order During Method Invocation

As part of an instance method invocation (§15.12), there is an expression that denotes the object to be invoked. This expression appears to be fully evaluated before any part of any argument expression to the method invocation is evaluated.

So, for example, in:

```java
class Test2 {
    public static void main(String[] args) {
        String s = "one";
        if (s.startsWith(s = "two"))
            System.out.println("oops");
    }
}
```

the occurrence of `s` before `s.startsWith` is evaluated first, before the argument expression `s = "two"`. Therefore, a reference to the string "one" is remembered as the target reference before the local variable `s` is changed to refer to the string "two". As a result, the `startsWith` method is invoked for target object "one" with argument "two", so the result of the invocation is `false`, as the string "one" does not start with "two". It follows that the test program does not print "oops".

15.12.4.2 Evaluate Arguments

The process of evaluating the argument list differs, depending on whether the method being invoked is a fixed arity method or a variable arity method (§8.4.1).

If the method being invoked is a variable arity method `m`, it necessarily has `n > 0` formal parameters. The final formal parameter of `m` necessarily has type `T[]` for some `T`, and `m` is necessarily being invoked with `k ≥ 0` actual argument expressions.

If `m` is being invoked with `k ≠ n` actual argument expressions, or, if `m` is being invoked with `k = n` actual argument expressions and the type of the `k`th argument expression is not assignment compatible with `T[]`, then the argument list `(e₁, ..., eₙ₋₁, eₙ, ..., eₖ)` is evaluated as if it were written as `(e₁, ..., eₙ₋₁, new |T[]| { eₙ, ..., eₖ })`, where `|T[]|` denotes the erasure (§4.6) of `T[]`.

The preceding paragraph is crafted to handle the interaction of parameterized types and array types that occurs in a Java Virtual Machine with erased generics. Namely, if the element type `T` of the variable array parameter is non-reifiable, e.g. `List<String>`, then special care must be taken with the array creation expression (§15.10) because the created array’s element type must be reifiable. By erasing the array type of the final expression in the argument list, we are guaranteed to obtain a reifiable element type. Then, since the array creation expression appears in an invocation context (§5.3), an unchecked conversion is possible from the array type with reifiable element type to an array type with non-reifiable element type, specifically that of the variable arity parameter. A Java compiler is required to give a compile-time unchecked warning at this conversion. Oracle's reference
implementation of a Java compiler identifies this unchecked warning as a more informative
unchecked generic array creation.

The argument expressions (possibly rewritten as described above) are now
evaluated to yield argument values. Each argument value corresponds to exactly
one of the method's \( n \) formal parameters.

The argument expressions, if any, are evaluated in order, from left to right. If the
evaluation of any argument expression completes abruptly, then no part of any
argument expression to its right appears to have been evaluated, and the method
invocation completes abruptly for the same reason. The result of evaluating the
\( j \)'th argument expression is the \( j \)'th argument value, for \( 1 \leq j \leq n \). Evaluation then
continues, using the argument values, as described below.

15.12.4.3 Check Accessibility of Type and Method

In this section:
- Let \( D \) be the class containing the method invocation.
- Let \( T \) be the qualifying type of the method invocation (§13.1).
- Let \( m \) be the name of the method as determined at compile time (§15.12.3).

An implementation of the Java programming language must ensure, as part of
linkage, that the type \( T \) is accessible:
- If \( T \) is in the same package as \( D \), then \( T \) is accessible.
- If \( T \) is in a different package than \( D \), and their packages are in the same module,
  and \( T \) is public or protected, then \( T \) is accessible.
- If \( T \) is in a different package than \( D \), and their packages are in different modules,
  and \( T \)'s module exports \( T \)'s package to \( D \)'s module, and \( T \) is public or protected,
  then \( T \) is accessible.

If \( T \) is protected, it is necessarily a nested type, so at compile time, its accessibility is
affected by the accessibility of types enclosing its declaration. However, during linkage, its
accessibility is not affected by the accessibility of types enclosing its declaration. Moreover,
during linkage, a protected \( T \) is as accessible as a public \( T \). These discrepancies between
access control at compile time (§6.6) and access control at run time are due to limitations
in the Java Virtual Machine.

The implementation must also ensure, during linkage, that the method \( m \) can still
be found in \( T \) or a supertype of \( T \). If \( m \) cannot be found, then a NoSuchMethodError
(which is a subclass of IncompatibleClassChangeError) occurs. If \( m \) can be
found, then let $C$ be the type that declares $m$. The implementation must ensure, during linkage, that the declaration of $m$ in $C$ is accessible to $D$:

- If $m$ is public, then $m$ is accessible.
- If $m$ is protected, then $m$ is accessible iff (i) either $D$ is in the same package as $C$, or $D$ is a subtype of $C$ or $C$ itself; and (ii) if $m$ is a protected instance method, then $T$ must be a subtype of $D$ or $D$ itself.

This is the only place where $T$ is involved in checks for $m$, because a protected instance method may only be invoked via a qualifying type that aligns with the invoker's type.

- If $m$ has package access, then $m$ is accessible iff $D$ is in the same package as $C$.
- If $m$ is private, then $m$ is accessible iff $D$ is $C$, or $D$ encloses $C$, or $C$ encloses $D$, or $C$ and $D$ are both enclosed by a third type.

If either $T$ or $m$ is not accessible, then an IllegalAccessError occurs (§12.3).

If the invocation mode is interface, then the implementation must check that the target reference type still implements the specified interface. If the target reference type does not still implement the interface, then an IncompatibleClassChangeError occurs.

15.12.4.4 Locate Method to Invoke

As in the previous section (§15.12.4.3):

- Let $T$ be the qualifying type of the method invocation (§13.1).
- Let $m$ be the method found in $T$ or a supertype of $T$. (Note that $m$ was merely the name of the method in the previous section; here it is the actual declaration.)
- Let $C$ be the class or interface that declares $m$.

The strategy for locating a method to invoke depends on the invocation mode:

- If the invocation mode is static, no target reference is needed and overriding is not allowed. Method $m$ of class or interface $C$ is the one to be invoked.
- Otherwise, an instance method is to be invoked and there is a target reference. If the target reference is null, a NullPointerException is thrown at this point. Otherwise, the target reference is said to refer to a target object and will be used as the value of the keyword this in the invoked method. The other three possibilities for the invocation mode are then considered:

  –
If the invocation mode is `super`, overriding is not allowed. Method \( m \) of class or interface \( C \) is the one to be invoked. If \( m \) is `abstract`, an `AbstractMethodError` is thrown.

Otherwise, if the invocation mode is `virtual`, and \( T \) and \( m \) jointly indicate a signature polymorphic method (§15.12.3), then the target object is an instance of `java.lang.invoke.MethodHandle` or `java.lang.invoke.VarHandle`. The target object encapsulates state which is matched against the information associated with the method invocation at compile time. Details of this matching are given in *The Java Virtual Machine Specification, Java SE 11 Edition* and the Java SE Platform API. If matching succeeds, then either the method referenced by the `java.lang.invoke.MethodHandle` instance is directly and immediately invoked, or the variable represented by the `java.lang.invoke.VarHandle` instance is directly and immediately accessed, *and in either case the procedure in §15.12.4.5 is not executed*. If matching fails, then a `java.lang.invoke.WrongMethodTypeException` is thrown.

Otherwise, the invocation mode is `interface` or `virtual`.

If the method \( m \) of class or interface \( C \) is `private`, then it is the method to be invoked.

Otherwise, overriding may occur. A *dynamic method lookup*, specified below, is used to locate the method to invoke. The lookup procedure starts from class \( R \), the actual run-time class of the target object.

Note that for invocation mode `interface`, \( R \) necessarily implements \( T \); for invocation mode `virtual`, \( R \) is necessarily either \( T \) or a subclass of \( T \). If the target object is an array, then \( R \) is a "class" representing an array type.

The procedure for dynamic method lookup is as follows. Let \( S \) be the class to search, beginning with \( R \). Then:

1. If class \( S \) contains a declaration for a method that overrides method \( m \) of class or interface \( C \) from \( R \) (§8.4.8.1), then that overriding method is the method to be invoked, and the procedure terminates.

2. Otherwise, if \( S \) has a superclass, then steps 1 and 2 of this lookup procedure are performed recursively using the direct superclass of \( S \) in place of \( S \); the method to be invoked, if any, is the result of the recursive invocation of this lookup procedure.

3. If no method is found by the previous two steps, the superinterfaces of \( S \) are searched for a suitable method.
A set of candidate methods is considered with the following properties: (i) each method is declared in a (direct or indirect) superinterface of \( R \); (ii) each method has the name and descriptor required by the method invocation; (iii) each method is non-static and non-private; (iv) for each method, where the method's declaring interface is \( I \), there is no other method satisfying (i) through (iii) that is declared in a subinterface of \( I \).

If this set contains a default method, one such method is the method to be invoked. Otherwise, an abstract method in the set is selected as the method to be invoked.

Dynamic method lookup may cause the following errors to occur:

- If the method to be invoked is abstract, an AbstractMethodError is thrown.
- If the method to be invoked is default, and more than one default method appears in the set of candidates in step 3 above, an IncompatibleClassChangeError is thrown.
- If the invocation mode is interface and the method to be invoked is neither public nor private, an IllegalAccessException is thrown.

The above procedure (if it terminates without error) will find a non-abstract, accessible method to invoke, provided that all classes and interfaces in the program have been consistently compiled. However, if this is not the case, then various errors may occur, as specified above; additional details about the behavior of the Java Virtual Machine under these circumstances are given by *The Java Virtual Machine Specification, Java SE 11 Edition*.

The dynamic lookup process, while described here explicitly, will often be implemented implicitly, for example as a side-effect of the construction and use of per-class method dispatch tables, or the construction of other per-class structures used for efficient dispatch.

### Example 15.12.4.4-1. Overriding and Method Invocation

```java
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
            clear();
    }
    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }
}
```
class ColoredPoint extends Point {
    int color;
    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }
}

Here, the subclass ColoredPoint extends the clear abstraction defined by its superclass Point. It does so by overriding the clear method with its own method, which invokes the clear method of its superclass, using the form super.clear().

This method is then invoked whenever the target object for an invocation of clear is a ColoredPoint. Even the method move in Point invokes the clear method of class ColoredPoint when the class of this is ColoredPoint, as shown by the output of this test program:

class Test1 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20): ");
        p.move(20, 20);

        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20): ");
        cp.move(20, 20);

        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:");
        p.move(20, 20);
    }
}

which is:

    p.move(20,20):
      Point clear
    cp.move(20,20):
      ColoredPoint clear
      Point clear
    p.move(20,20), p colored:
      ColoredPoint clear
      Point clear

Overriding is sometimes called "late-bound self-reference"; in this example it means that the reference to clear in the body of Point.move (which is really syntactic shorthand for this.clear) invokes a method chosen "late" (at run time, based on the run-time class of the object referenced by this) rather than a method chosen "early" (at compile time, based only on the type of this). This provides the programmer a powerful way of extending abstractions and is a key idea in object-oriented programming.
Example 15.12.4-2. Method Invocation Using `super`

An overridden instance method of a superclass may be accessed by using the keyword `super` to access the members of the immediate superclass, bypassing any overriding declaration in the class that contains the method invocation.

When accessing an instance variable, `super` means the same as a cast of `this` (§15.11.2), but this equivalence does not hold true for method invocation. This is demonstrated by the example:

```java
class T1 {
    String s() { return "1"; }
}
class T2 extends T1 {
    String s() { return "2"; }
}
class T3 extends T2 {
    String s() { return "3"; }
    void test() {
        System.out.println("s()=\t\t" + s());
        System.out.println("super.s()=\t\t" + super.s());
        System.out.println("((T2)this).s()=\t\t" + ((T2)this).s());
        System.out.println("((T1)this).s()=\t\t" + ((T1)this).s());
    }
}
class Test2 {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}
```

which produces the output:

```
s()= 3
super.s()= 2
((T2)this).s()= 3
((T1)this).s()= 3
```

The casts to types `T1` and `T2` do not change the method that is invoked, because the instance method to be invoked is chosen according to the run-time class of the object referred to by `this`. A cast does not change the class of an object; it only checks that the class is compatible with the specified type.

15.12.4.5 Create Frame, Synchronize, Transfer Control

A method `m` in some class `s` has been identified as the one to be invoked.

Now a new activation frame is created, containing the target reference (if any) and the argument values (if any), as well as enough space for the local variables and
stack for the method to be invoked and any other bookkeeping information that may be required by the implementation (stack pointer, program counter, reference to previous activation frame, and the like). If there is not sufficient memory available to create such an activation frame, a `StackOverflowError` is thrown.

The newly created activation frame becomes the current activation frame. The effect of this is to assign the argument values to corresponding freshly created parameter variables of the method, and to make the target reference available as this, if there is a target reference. Before each argument value is assigned to its corresponding parameter variable, it is subjected to invocation conversion (§5.3), which includes any required value set conversion (§5.1.13).

If the erasure (§4.6) of the type of the method being invoked differs in its signature from the erasure of the type of the compile-time declaration for the method invocation (§15.12.3), then if any of the argument values is an object which is not an instance of a subclass or subinterface of the erasure of the corresponding formal parameter type in the compile-time declaration for the method invocation, then a `ClassCastException` is thrown.

If the method $m$ is a native method but the necessary native, implementation-dependent binary code has not been loaded or otherwise cannot be dynamically linked, then an `UnsatisfiedLinkError` is thrown.

If the method $m$ is not synchronized, control is transferred to the body of the method $m$ to be invoked.

If the method $m$ is synchronized, then an object must be locked before the transfer of control. No further progress can be made until the current thread can obtain the lock. If there is a target reference, then the target object must be locked; otherwise the `Class` object for class $S$, the class of the method $m$, must be locked. Control is then transferred to the body of the method $m$ to be invoked. The object is automatically unlocked when execution of the body of the method has completed, whether normally or abruptly. The locking and unlocking behavior is exactly as if the body of the method were embedded in a synchronized statement (§14.19).

**Example 15.12.4.5-1. Invoked Method Signature Has Different Erasure Than Compile-Time Method Signature**

Consider the declarations:

```java
abstract class C<T> {
    abstract T id(T x);
}
class D extends C<String> {
    String id(String x) { return x; }
}
```
Now, given an invocation:

```java
C c = new D();
c.id(new Object());  // fails with a ClassCastException
```

The erasure of the actual method being invoked, `D.id()`, differs in its signature from that of the compile-time method declaration, `C.id()`. The former takes an argument of type `String` while the latter takes an argument of type `Object`. The invocation fails with a `ClassCastException` before the body of the method is executed.

Such situations can only arise if the program gives rise to a compile-time unchecked warning (§4.8, §5.1.6, §5.1.9, §8.4.1, §8.4.8.3, §15.13.2, §15.12.4.2, §15.27.3).

Implementations can enforce these semantics by creating *bridge methods*. In the above example, the following bridge method would be created in class `D`:

```java
Object id(Object x) { return id((String) x); }
```

This is the method that would actually be invoked by the Java Virtual Machine in response to the call `c.id(new Object())` shown above, and it will execute the cast and fail, as required.

## 15.13 Method Reference Expressions

A method reference expression is used to refer to the invocation of a method without actually performing the invocation. Certain forms of method reference expression also allow class instance creation (§15.9) or array creation (§15.10) to be treated as if it were a method invocation.

```
MethodReference:
  ExpressionName :: [TypeArguments] Identifier
  Primary :: [TypeArguments] Identifier
  ReferenceType :: [TypeArguments] Identifier
  super :: [TypeArguments] Identifier
  TypeName . super :: [TypeArguments] Identifier
  ClassType :: [TypeArguments] new
  ArrayType :: new
```

If `TypeArguments` is present to the right of `::`, then it is a compile-time error if any of the type arguments are wildcards (§4.5.1).

If a method reference expression has the form `ExpressionName :: [TypeArguments] Identifier` or `Primary :: [TypeArguments] Identifier`, it is a
compile-time error if the type of the _ExpressionName_ or _Primary_ is not a reference type.

If a method reference expression has the form `super :: [TypeArguments] Identifier`, let \( \tau \) be the type declaration immediately enclosing the method reference expression. It is a compile-time error if \( \tau \) is the class `Object` or \( \tau \) is an interface.

If a method reference expression has the form `_TypeName` . `super :: [TypeArguments] Identifier`, then:

- If `_TypeName` denotes a class, \( C \), then it is a compile-time error if \( C \) is not a lexically enclosing class of the current class, or if \( C \) is the class `Object`.
- If `_TypeName` denotes an interface, \( I \), then let \( \tau \) be the type declaration immediately enclosing the method reference expression. It is a compile-time error if \( I \) is not a direct superinterface of \( \tau \), or if there exists some other direct superclass or direct superinterface of \( \tau, J \), such that \( J \) is a subtype of \( I \).
- If `_TypeName` denotes a type variable, then a compile-time error occurs.

If a method reference expression has the form `super :: [TypeArguments] Identifier` or `_TypeName` . `super :: [TypeArguments] Identifier`, it is a compile-time error if the expression occurs in a static context.

If a method reference expression has the form `ClassType :: [TypeArguments] new`, then:

- `ClassType` must denote a class that is accessible (§6.6), non-abstract, and not an enum type, or a compile-time error occurs.
- If `ClassType` denotes a parameterized type (§4.5), then it is a compile-time error if any of its type arguments are wildcards.
- If `ClassType` denotes a raw type (§4.8), then it is a compile-time error if `TypeArguments` is present after the ::.

If a method reference expression has the form `ArrayType :: new`, then `ArrayType` must denote a type that is reifiable (§4.7), or a compile-time error occurs.

The target reference of an instance method (§15.12.4.1) may be provided by the method reference expression using an _ExpressionName_, a _Primary_, or `super`, or it may be provided later when the method is invoked. The immediately enclosing instance of a new inner class instance (§15.9.2) is provided by a lexically enclosing instance of `this` (§8.1.3).

When more than one member method of a type has the same name, or when a class has more than one constructor, the appropriate method or constructor is selected
based on the functional interface type targeted by the method reference expression, as specified in §15.13.1.

If a method or constructor is generic, the appropriate type arguments may either be inferred or provided explicitly. Similarly, the type arguments of a generic type mentioned by the method reference expression may be provided explicitly or inferred.

Method reference expressions are always poly expressions (§15.2).

It is a compile-time error if a method reference expression occurs in a program in someplace other than an assignment context (§5.2), an invocation context (§5.3), or a casting context (§5.5).

Evaluation of a method reference expression produces an instance of a functional interface type (§9.8). This does not cause the execution of the corresponding method; instead, the execution may occur at a later time when an appropriate method of the functional interface is invoked.

Here are some method reference expressions, first with no target reference and then with a target reference:

```java
String::length // instance method
System::currentTimeMillis // static method
List<String>::size // explicit type arguments for generic type
List::size // inferred type arguments for generic type
int[]::clone
T::tvarMember
System.out::println
"abc"::length
foo[x]::bar
(test ? list.replaceAll(String::trim) : list) :: iterator
super::toString
```

Here are some more method reference expressions:

```java
String::valueOf // overload resolution needed
Arrays::sort // type arguments inferred from context
Arrays::<String>sort // explicit type arguments
```

Here are some method reference expressions that represent a deferred creation of an object or an array:
Method Reference Expressions

ArrayList<String>::new     // constructor for parameterized type
ArrayList::new             // inferred type arguments
Foo::<Integer>new          // explicit type arguments
Bar<String>::<Integer>new  // generic class, generic constructor
Outer.Inner::new           // inner class constructor
int[]::new                 // array creation

It is not possible to specify a particular signature to be matched, for example, Arrays::sort(int[]). Instead, the functional interface provides argument types that are used as input to the overload resolution algorithm (§15.12.2). This should satisfy the vast majority of use cases; when the rare need arises for more precise control, a lambda expression can be used.

The use of type argument syntax in the class name before a delimiter (List<String>::size) raises the parsing problem of distinguishing between < as a type argument bracket and < as a less-than operator. In theory, this is no worse than allowing type arguments in cast expressions; however, the difference is that the cast case only comes up when a ( token is encountered; with the addition of method reference expressions, the start of every expression is potentially a parameterized type.

15.13.1 Compile-Time Declaration of a Method Reference

The compile-time declaration of a method reference expression is the method to which the expression refers. In special cases, the compile-time declaration does not actually exist, but is a notional method that represents a class instance creation or an array creation. The choice of compile-time declaration depends on a function type targeted by the expression, just as the compile-time declaration of a method invocation depends on the invocation's arguments (§15.12.3).

The search for a compile-time declaration mirrors the process for method invocations in §15.12.1 and §15.12.2, as follows:

- First, a type to search is determined:
  - If the method reference expression has the form ExpressionName :: [TypeArguments] Identifier or Primary :: [TypeArguments] Identifier, the type to search is the type of the expression preceding the :: token.
  - If the method reference expression has the form ReferenceType :: [TypeArguments] Identifier, the type to search is the result of capture conversion (§5.1.10) applied to ReferenceType.
  - If the method reference expression has the form super :: [TypeArguments] Identifier, the type to search is the superclass type of the class whose declaration contains the method reference.
- If the method reference expression has the form `TypeName . super :: [TypeArguments] Identifier`, then if `TypeName` denotes a class, the type to search is the superclass type of the named class; otherwise, `TypeName` denotes an interface, and the corresponding superinterface type of the class or interface whose declaration contains the method reference is the type to search.

- For the two other forms (involving `:: new`), the referenced method is notional and there is no type to search.

- Second, given a targeted function type with `n` parameters, a set of potentially applicable methods is identified:

  - If the method reference expression has the form `ReferenceType :: [TypeArguments] Identifier`, then the potentially applicable methods are:
    - the member methods of the type to search that would be potentially applicable (§15.12.2.1) for a method invocation which names `Identifier`, has arity `n`, has type arguments `TypeArguments`, and appears in the same class as the method reference expression; plus
    - the member methods of the type to search that would be potentially applicable for a method invocation which names `Identifier`, has arity `n-1`, has type arguments `TypeArguments`, and appears in the same class as the method reference expression.

    Two different arities, `n` and `n-1`, are considered, to account for the possibility that this form refers to either a static method or an instance method.

  - If the method reference expression has the form `ClassType :: [TypeArguments] new`, then the potentially applicable methods are a set of notional methods corresponding to the constructors of `ClassType`.

    If `ClassType` is a raw type, but is not a non-static member type of a raw type, the candidate notional member methods are those specified in §15.9.3 for a class instance creation expression that uses `$<>` to elide the type arguments to a class. Otherwise, the candidate notional member methods are the constructors of `ClassType`, treated as if they were methods with return type `ClassType`.

    Among these candidates, the potentially applicable methods are the notional methods that would be potentially applicable for a method invocation which has arity `n`, has type arguments `TypeArguments`, and appears in the same class as the method reference expression.

  - If the method reference expression has the form `ArrayType :: new`, a single notional method is considered. The method has a single parameter of type `int`, returns the `ArrayType`, and has no `throws` clause. If `n = 1`, this is the only
potentially applicable method; otherwise, there are no potentially applicable methods.

- For all other forms, the potentially applicable methods are the member methods of the type to search that would be potentially applicable for a method invocation which names Identifier, has arity $n$, has type argument TypeArguments, and appears in the same class as the method reference expression.

- Finally, if there are no potentially applicable methods, then there is no compile-time declaration.

Otherwise, given a targeted function type with parameter types $P_1, ..., P_n$ and a set of potentially applicable methods, the compile-time declaration is selected as follows:

- If the method reference expression has the form ReferenceType :: [TypeArguments] Identifier, then two searches for a most specific applicable method are performed. Each search is as specified in §15.12.2.2 through §15.12.2.5, with the clarifications below. Each search produces a set of applicable methods and, possibly, designates a most specific method of the set. In the case of an error as specified in §15.12.2.4, the set of applicable methods is empty. In the case of an error as specified in §15.12.2.5, there is no most specific method.

In the first search, the method reference is treated as if it were an invocation with argument expressions of types $P_1, ..., P_n$. Type arguments, if any, are given by the method reference expression.

In the second search, if $P_1, ..., P_n$ is not empty and $P_1$ is a subtype of ReferenceType, then the method reference expression is treated as if it were a method invocation expression with argument expressions of types $P_2, ..., P_n$. If ReferenceType is a raw type, and there exists a parameterization of this type, $G<...>$, that is a supertype of $P_1$, the type to search is the result of capture conversion (§5.1.10) applied to $G<...>$; otherwise, the type to search is the same as the type of the first search. Type arguments, if any, are given by the method reference expression.

If the first search produces a most specific method that is static, and the set of applicable methods produced by the second search contains no non-static methods, then the compile-time declaration is the most specified method of the first search.

Otherwise, if the set of applicable methods produced by the first search contains no static methods, and the second search produces a most specific
method that is non-static, then the compile-time declaration is the most specific method of the second search.

Otherwise, there is no compile-time declaration.

- For all other forms of method reference expression, one search for a most specific applicable method is performed. The search is as specified in §15.12.2.2 through §15.12.2.5, with the clarifications below.

The method reference is treated as if it were an invocation with argument expressions of types $P_1, \ldots, P_n$; the type arguments, if any, are given by the method reference expression.

If the search results in an error as specified in §15.12.2.2 through §15.12.2.5, or if the most specific applicable method is static, there is no compile-time declaration.

Otherwise, the compile-time declaration is the most specific applicable method.

It is a compile-time error if a method reference expression has the form `ReferenceType :: [TypeArguments] Identifier`, and the compile-time declaration is static, and `ReferenceType` is not a simple or qualified name (§6.2).

It is a compile-time error if the method reference expression has the form `super :: [TypeArguments] Identifier` or `TypeName . super :: [TypeArguments] Identifier`, and the compile-time declaration is abstract.

It is a compile-time error if the method reference expression has the form `TypeName . super :: [TypeArguments] Identifier`, and `TypeName` denotes an interface, and there exists a method, distinct from the compile-time declaration, that overrides (§8.4.8, §9.4.1) the compile-time declaration from a direct superclass or direct superinterface of the type whose declaration immediately encloses the method reference expression.

It is a compile-time error if the method reference expression is of the form `ClassType :: [TypeArguments] new` and a compile-time error would occur when determining an enclosing instance for `ClassType` as specified in §15.9.2 (treating the method reference expression as if it were an unqualified class instance creation expression).

A method reference expression of the form `ReferenceType :: [TypeArguments] Identifier` can be interpreted in different ways. If `Identifier` refers to an instance method, then the implicit lambda expression has an extra parameter compared to if `Identifier` refers to a static method. It is possible for `ReferenceType` to have both kinds of applicable methods, so the search algorithm described above identifies them separately, since there are different parameter types for each case.
An example of ambiguity is:

```java
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
        // Error: instance method size()
        // or static method size(Object)?
    }
}
```

This ambiguity cannot be resolved by providing an applicable instance method which is more specific than an applicable `static` method:

```java
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    int size(C arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
        // Error: instance method size()
        // or static method size(Object)?
    }
}
```

The search is smart enough to ignore ambiguities in which all the applicable methods (from both searches) are instance methods:

```java
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    int size(Object arg) { return 0; }
    int size(C arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
        // OK: reference is to instance method size()
    }
}
```

For convenience, when the name of a generic type is used to refer to an instance method (where the receiver becomes the first parameter), the target type is used to determine the type arguments. This facilitates usage like `Pair::first` in place of `Pair<String,Integer>::first`. Similarly, a method reference like `Pair::new` is
For some method reference expressions, there is only one possible compile-time declaration with only one possible invocation type (§15.12.2.6), regardless of the targeted function type. Such method reference expressions are said to be exact. A method reference expression that is not exact is said to be inexact.

A method reference expression ending with Identifier is exact if it satisfies all of the following:

- If the method reference expression has the form ReferenceType :: [TypeArguments] Identifier, then ReferenceType does not denote a raw type.
- The type to search has exactly one member method with the name Identifier that is accessible to the class or interface in which the method reference expression appears.
- This method is not variable arity (§8.4.1).
- If this method is generic (§8.4.4), then the method reference expression provides TypeArguments.

A method reference expression of the form ClassType :: [TypeArguments] new is exact if it satisfies all of the following:

- The type denoted by ClassType is not raw, or is a non-static member type of a raw type.
- The type denoted by ClassType has exactly one constructor that is accessible to the class or interface in which the method reference expression appears.
- This constructor is not variable arity.
- If this constructor is generic, then the method reference expression provides TypeArguments.

A method reference expression of the form ArrayType :: new is always exact.

### 15.13.2 Type of a Method Reference

A method reference expression is compatible in an assignment context, invocation context, or casting context with a target type $\tau$ if $\tau$ is a functional interface type (§9.8) and the expression is congruent with the function type of the ground target type derived from $\tau$.

The ground target type is derived from $\tau$ as follows:
• If \( T \) is a wildcard-parameterized functional interface type, then the ground target type is the non-wildcard parameterization (§9.9) of \( T \).

• Otherwise, the ground target type is \( T \).

A method reference expression is congruent with a function type if both of the following are true:

• The function type identifies a single compile-time declaration corresponding to the reference.

• One of the following is true:
  
  – The result of the function type is \( \text{void} \).
  
  – The result of the function type is \( R \), and the result of applying capture conversion (§5.1.10) to the return type of the invocation type (§15.12.2.6) of the chosen compile-time declaration is \( R' \) (where \( R \) is the target type that may be used to infer \( R' \)), and neither \( R \) nor \( R' \) is \( \text{void} \), and \( R' \) is compatible with \( R \) in an assignment context.

If unchecked conversion was necessary for the compile-time declaration to be applicable, and this conversion would cause an unchecked warning in an invocation context, then a compile-time unchecked warning occurs, unless suppressed by @SuppressWarnings (§9.6.4.5).

If unchecked conversion was necessary for the return type \( R' \), described above, to be compatible with the function type's return type, \( R \), and this conversion would cause an unchecked warning in an assignment context, then a compile-time unchecked warning occurs, unless suppressed by @SuppressWarnings.

If a method reference expression is compatible with a target type \( T \), then the type of the expression, \( U \), is the ground target type derived from \( T \).

It is a compile-time error if any class or interface mentioned by either \( U \) or the function type of \( U \) is not accessible (§6.6) from the class or interface in which the method reference expression appears.

For each non-static member method \( m \) of \( U \), if the function type of \( U \) has a subsignature of the signature of \( m \), then a notional method whose method type is the function type of \( U \) is said to override \( m \), and any compile-time error or unchecked warning specified in §8.4.8.3 may occur.

For each checked exception type \( X \) listed in the throws clause of the invocation type of the compile-time declaration, \( X \) or a superclass of \( X \) must be mentioned in the throws clause of the function type of \( U \), or a compile-time error occurs.
The key idea driving the compatibility definition is that a method reference is compatible if and only if the equivalent lambda expression \((x, y, z) \rightarrow \text{exp.<T1, T2>method(x, y, z)}\) is compatible. (This is informal, and there are issues that make it difficult or impossible to formally define the semantics in terms of such a rewrite.)

These compatibility rules provide a convenient facility for converting from one functional interface to another:

```java
Task t = () -> System.out.println("hi");
Runnable r = t::invoke;
```

The implementation may be optimized so that when a lambda-derived object is passed around and converted to various types, this does not result in many levels of adaptation logic around the core lambda body.

Unlike a lambda expression, a method reference can be congruent with a generic function type (that is, a function type that has type parameters). This is because the lambda expression would need to be able to declare type parameters, and no syntax supports this; while for a method reference, no such declaration is necessary. For example, the following program is legal:

```java
interface ListFactory {
  <T> List<T> make();
}
ListFactory lf = ArrayList::new;
List<String> ls = lf.make();
List<Number> ln = lf.make();
```

### 15.13.3 Run-Time Evaluation of Method References

At run time, evaluation of a method reference expression is similar to evaluation of a class instance creation expression, insofar as normal completion produces a reference to an object. Evaluation of a method reference expression is distinct from invocation of the method itself.

First, if the method reference expression begins with an `ExpressionName` or a `Primary`, this subexpression is evaluated. If the subexpression evaluates to `null`, a `NullPointerException` is raised, and the method reference expression completes abruptly. If the subexpression completes abruptly, the method reference expression completes abruptly for the same reason.

Next, either a new instance of a class with the properties below is allocated and initialized, or an existing instance of a class with the properties below is referenced. If a new instance is to be created, but there is insufficient space to allocate the object, evaluation of the method reference expression completes abruptly by throwing an `OutOfMemoryError`. 
The value of a method reference expression is a reference to an instance of a class with the following properties:

- The class implements the targeted functional interface type and, if the target type is an intersection type, every other interface type mentioned in the intersection.
- Where the method reference expression has type $U$, for each non-static member method $m$ of $U$:

  If the function type of $U$ has a subsignature of the signature of $m$, then the class declares an *invocation method* that overrides $m$. The invocation method's body invokes the referenced method, creates a class instance, or creates an array, as described below. If the invocation method's result is not `void`, then the body returns the result of the method invocation or object creation, after any necessary assignment conversions (§5.2).

  If the erasure of the type of a method being overridden differs in its signature from the erasure of the function type of $U$, then before the method invocation or object creation, an invocation method's body checks that each argument value is an instance of a subclass or subinterface of the erasure of the corresponding parameter type in the function type of $U$, if not, a `ClassCastException` is thrown.

  - The class overrides no other methods of the functional interface type or other interface types mentioned above, although it may override methods of the `Object` class.

The body of an invocation method depends on the form of the method reference expression, as follows:

- If the form is `ExpressionName :: [TypeArguments] Identifier` or `Primary :: [TypeArguments] Identifier`, then the body of the invocation method has the effect of a method invocation expression for a compile-time declaration which is the compile-time declaration of the method reference expression. Run-time evaluation of the method invocation expression is as specified in §15.12.4.3, §15.12.4.4, and §15.12.4.5, where:

  - The invocation mode is derived from the compile-time declaration as specified in §15.12.3.

  - The target reference is the value of `ExpressionName` or `Primary`, as determined when the method reference expression was evaluated.

  - The arguments to the method invocation expression are the formal parameters of the invocation method.

- If the form is `ReferenceType :: [TypeArguments] Identifier`, the body of the invocation method similarly has the effect of a method invocation expression for
a compile-time declaration which is the compile-time declaration of the method reference expression. Run-time evaluation of the method invocation expression is as specified in §15.12.4.3, §15.12.4.4, and §15.12.4.5, where:

- The invocation mode is derived from the compile-time declaration as specified in §15.12.3.

- If the compile-time declaration is an instance method, then the target reference is the first formal parameter of the invocation method. Otherwise, there is no target reference.

- If the compile-time declaration is an instance method, then the arguments to the method invocation expression (if any) are the second and subsequent formal parameters of the invocation method. Otherwise, the arguments to the method invocation expression are the formal parameters of the invocation method.

• If the form is `super :: [TypeArguments] Identifier` or `TypeName . super :: [TypeArguments] Identifier`, the body of the invocation method has the effect of a method invocation expression for a compile-time declaration which is the compile-time declaration of the method reference expression. Run-time evaluation of the method invocation expression is as specified in §15.12.4.3, §15.12.4.4, and §15.12.4.5, where:

  - The invocation mode is `super`.

  - If the method reference expression begins with a `TypeName` that names a class, the target reference is the value of `TypeName . this` at the point at which the method reference is evaluated. Otherwise, the target reference is the value of `this` at the point at which the method reference is evaluated.

  - The arguments to the method invocation expression are the formal parameters of the invocation method.

• If the form is `ClassType :: [TypeArguments] new`, the body of the invocation method has the effect of a class instance creation expression of the form `new [TypeArguments] ClassType(A₁, ..., Aₙ)`, where the arguments `A₁, ..., Aₙ` are the formal parameters of the invocation method, and where:

  - The enclosing instance for the new object, if any, is derived from the site of the method reference expression, as specified in §15.9.2.

  - The constructor to invoke is the constructor that corresponds to the compile-time declaration of the method reference (§15.13.1).

• If the form is `Type[k] :: new (k ≥ 1)`, then the body of the invocation method has the same effect as an array creation expression of the form `new Type [ size ]`
\[ k-1 \text{, where } size \text{ is the invocation method's single parameter. (The notation } [ ]^k \text{ indicates a sequence of } k \text{ bracket pairs.)} \]

If the body of the invocation method has the effect of a method invocation expression, then the compile-time parameter types and the compile-time result of the method invocation are determined as specified in §15.12.3. For the purpose of determining the compile-time result, the method invocation expression is an expression statement if the invocation method's result is void, and the Expression of a return statement if the invocation method's result is non-void.

The timing of method reference expression evaluation is more complex than that of lambda expressions (§15.27.4). When a method reference expression has an expression (rather than a type) preceding the :: separator, that subexpression is evaluated immediately. The result of evaluation is stored until the method of the corresponding functional interface type is invoked; at that point, the result is used as the target reference for the invocation. This means the expression preceding the :: separator is evaluated only when the program encounters the method reference expression, and is not re-evaluated on subsequent invocations on the functional interface type.

It is interesting to contrast the treatment of null here with its treatment during method invocation. When a method invocation expression is evaluated, it is possible for the Primary that qualifies the invocation to evaluate to null but for no NullPointerException to be raised. This occurs when the invoked method is static (despite the syntax of the invocation suggesting an instance method). Since the applicable method for a method reference expression qualified by a Primary is prohibited from being static (§15.13.1), the evaluation of the method reference expression is simpler - a null Primary always raises a NullPointerException.

15.14 Postfix Expressions

Postfix expressions include uses of the postfix ++ and -- operators. Names are not considered to be primary expressions (§15.8), but are handled separately in the grammar to avoid certain ambiguities. They become interchangeable only here, at the level of precedence of postfix expressions.

\[
\text{PostfixExpression:} \\
\quad \text{Primary} \\
\quad \text{ExpressionName} \\
\quad \text{PostIncrementExpression} \\
\quad \text{PostDecrementExpression}
\]
15.14.1 Expression Names

The rules for evaluating expression names are given in §6.5.6.

15.14.2 Postfix Increment Operator ++

A postfix expression followed by a ++ operator is a postfix increment expression.

\[
\text{PostIncrementExpression:} \quad \text{PostfixExpression} \quad ++
\]

The result of the postfix expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the postfix increment expression is the type of the variable. The result of the postfix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value \(1\) is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value \(1\) and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the postfix increment expression is the value of the variable before the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared final cannot be incremented because when an access of such a final variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix increment operator.

15.14.3 Postfix Decrement Operator --

A postfix expression followed by a -- operator is a postfix decrement expression.

\[
\text{PostDecrementExpression:} \quad \text{PostfixExpression} \quad --
\]
The result of the postfix expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the postfix decrement expression is the type of the variable. The result of the postfix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value 1 is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the postfix decrement expression is the value of the variable before the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared final cannot be decremented because when an access of such a final variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix decrement operator.

15.15 Unary Operators

The operators +, -, ++, --, ~, !, and the cast operator (§15.16) are called the unary operators.

\[
\text{UnaryExpression:} \\
\quad \text{PreIncrementExpression} \\
\quad \text{PreDecrementExpression} \\
\quad + \text{UnaryExpression} \\
\quad - \text{UnaryExpression} \\
\quad \text{UnaryExpressionNotPlusMinus} \\
\]

\[
\text{PreIncrementExpression:} \\
\quad ++ \text{UnaryExpression} \\
\]

\[
\text{PreDecrementExpression:} \\
\quad -- \text{UnaryExpression} \\
\]
**UnaryExpressionNotPlusMinus:**

PostfixExpression

~ UnaryExpression

! UnaryExpression

CastExpression

The following production from §15.16 is shown here for convenience:

CastExpression:

( PrimitiveType ) UnaryExpression

( ReferenceType [AdditionalBound] ) UnaryExpressionNotPlusMinus

( ReferenceType [AdditionalBound] ) LambdaExpression

Expressions with unary operators group right-to-left, so that \(-\neg x\) means the same as \(- (\neg x)\).

This portion of the grammar contains some tricks to avoid two potential syntactic ambiguities.

The first potential ambiguity would arise in expressions such as \((p) + q\), which looks, to a C or C++ programmer, as though it could be either a cast to type \(p\) of a unary + operating on \(q\), or a binary addition of two quantities \(p\) and \(q\). In C and C++, the parser handles this problem by performing a limited amount of semantic analysis as it parses, so that it knows whether \(p\) is the name of a type or the name of a variable.

Java takes a different approach. The result of the + operator must be numeric, and all type names involved in casts on numeric values are known keywords. Thus, if \(p\) is a keyword naming a primitive type, then \((p) + q\) can make sense only as a cast of a unary expression. However, if \(p\) is not a keyword naming a primitive type, then \((p) + q\) can make sense only as a binary arithmetic operation. Similar remarks apply to the \(-\) operator. The grammar shown above splits CastExpression into two cases to make this distinction. The nonterminal UnaryExpression includes all unary operators, but the nonterminal UnaryExpressionNotPlusMinus excludes uses of all unary operators that could also be binary operators, which in Java are + and -.

The second potential ambiguity is that the expression \((p) ++\) could, to a C or C++ programmer, appear to be either a postfix increment of a parenthesized expression or the beginning of a cast, for example, in \((p) ++ q\). As before, parsers for C and C++ know whether \(p\) is the name of a type or the name of a variable. But a parser using only one-token lookahead and no semantic analysis during the parse would not be able to tell, when ++ is the lookahead token, whether \((p)\) should be considered a Primary expression or left alone for later consideration as part of a CastExpression.

In Java, the result of the ++ operator must be numeric, and all type names involved in casts on numeric values are known keywords. Thus, if \(p\) is a keyword naming a primitive type, then \((p) ++\) can make sense only as a cast of a prefix increment expression, and there had better be an operand such as \(q\) following the ++. However, if \(p\) is not a keyword naming a primitive type, then \((p) ++\) can make sense only as a postfix increment of \(p\). Similar remarks apply to the -- operator. The nonterminal UnaryExpressionNotPlusMinus therefore also excludes uses of the prefix operators ++ and --.
15.15.1 Prefix Increment Operator ++

A unary expression preceded by a ++ operator is a prefix increment expression.

The result of the unary expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the prefix increment expression is the type of the variable. The result of the prefix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value 1 is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the prefix increment expression is the value of the variable after the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared final cannot be incremented because when an access of such a final variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix increment operator.

15.15.2 Prefix Decrement Operator --

A unary expression preceded by a -- operator is a prefix decrement expression.

The result of the unary expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the prefix decrement expression is the type of the variable. The result of the prefix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value 1 is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of
the variable before it is stored. The value of the prefix decrement expression is the value of the variable after the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, format conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared final cannot be decremented because when an access of such a final variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix decrement operator.

15.15.3 Unary Plus Operator +

The type of the operand expression of the unary + operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary plus expression is the promoted type of the operand. The result of the unary plus expression is not a variable, but a value, even if the result of the operand expression is a variable.

At run time, the value of the unary plus expression is the promoted value of the operand.

15.15.4 Unary Minus Operator –

The type of the operand expression of the unary – operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand.

The type of the unary minus expression is the promoted type of the operand.

Note that unary numeric promotion performs value set conversion (§5.1.13). Whatever value set the promoted operand value is drawn from, the unary negation operation is carried out and the result is drawn from that same value set. That result is then subject to further value set conversion.

At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand.

For integer values, negation is the same as subtraction from zero. The Java programming language uses two's-complement representation for integers, and the range of two's-complement values is not symmetric, so negation of the maximum negative int or long results in that same maximum negative number. Overflow
occurs in this case, but no exception is thrown. For all integer values \( x \), \(-x\) equals \((-x)+1\).

For floating-point values, negation is not the same as subtraction from zero, because if \( x \) is \(+0.0\), then \( 0.0-x \) is \(+0.0\), but \(-x\) is \(-0.0\). Unary minus merely inverts the sign of a floating-point number. Special cases of interest:

- If the operand is NaN, the result is NaN. (Recall that NaN has no sign (§4.2.3).)
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

15.15.5 Bitwise Complement Operator ~

The type of the operand expression of the unary ~ operator must be a type that is convertible (§5.1.8) to a primitive integral type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary bitwise complement expression is the promoted type of the operand.

At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand. In all cases, \(~x\) equals \((-x)-1\).

15.15.6 Logical Complement Operator !

The type of the operand expression of the unary ! operator must be boolean or Boolean, or a compile-time error occurs.

The type of the unary logical complement expression is boolean.

At run time, the operand is subject to unboxing conversion (§5.1.8) if necessary. The value of the unary logical complement expression is true if the (possibly converted) operand value is false, and false if the (possibly converted) operand value is true.

15.16 Cast Expressions

A cast expression converts, at run time, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is boolean; or checks, at run time, that a reference value refers to an object either whose class is compatible with a specified reference type or list of reference types, or which embodies a value of a primitive type.
**CastExpression:**

( `PrimitiveType` ) `UnaryExpression`

( `ReferenceType` {`AdditionalBound`} ) `UnaryExpressionNotPlusMinus`

( `ReferenceType` {`AdditionalBound`} ) `LambdaExpression`

The following production from §4.4 is shown here for convenience:

`AdditionalBound:`

`&` `InterfaceType`

The parentheses and the type or list of types they contain are sometimes called the cast operator.

If the cast operator contains a list of types, that is, a `ReferenceType` followed by one or more `AdditionalBound` terms, then all of the following must be true, or a compile-time error occurs:

- `ReferenceType` must denote a class or interface type.
- The erasures (§4.6) of all the listed types must be pairwise different.
- No two listed types may be subtypes of different parameterizations of the same generic interface.

The target type for the casting context (§5.5) introduced by the cast expression is either the `PrimitiveType` or the `ReferenceType` (if not followed by `AdditionalBound` terms) appearing in the cast operator, or the intersection type denoted by the `ReferenceType` and `AdditionalBound` terms appearing in the cast operator.

The type of a cast expression is the result of applying capture conversion (§5.1.10) to this target type.

Casts can be used to explicitly "tag" a lambda expression or a method reference expression with a particular target type. To provide an appropriate degree of flexibility, the target type may be a list of types denoting an intersection type, provided the intersection induces a functional interface (§9.8).

The result of a cast expression is not a variable, but a value, even if the result of evaluating the operand expression is a variable.

A cast operator has no effect on the choice of value set (§4.2.3) for a value of type `float` or type `double`. Consequently, a cast to type `float` within an expression that is not FP-strict (§15.4) does not necessarily cause its value to be converted to an element of the float value set, and a cast to type `double` within an expression that is not FP-strict does not necessarily cause its value to be converted to an element of the double value set.
If the compile-time type of the operand cannot be converted by casting conversion (§5.5) to the target type specified by the cast operator, then a compile-time error occurs.

Otherwise, at run time, the operand value is converted (if necessary) by casting conversion to the target type specified by the cast operator.

A ClassCastException is thrown if a cast is found at run time to be impermissible.

Some casts result in an error at compile time. Some casts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its superclass; such a cast should require no special action at run time. Finally, some casts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time. See §5.5 for details.

15.17 Multiplicative Operators

The operators *, /, and % are called the multiplicative operators.

\[
\text{MultiplicativeExpression:} \\
\text{UnaryExpression} \\
\text{MultiplicativeExpression} * \text{UnaryExpression} \\
\text{MultiplicativeExpression} / \text{UnaryExpression} \\
\text{MultiplicativeExpression} % \text{UnaryExpression}
\]

The multiplicative operators have the same precedence and are syntactically left-associative (they group left-to-right).

The type of each of the operands of a multiplicative operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Binary numeric promotion is performed on the operands (§5.6.2).

Note that binary numeric promotion performs value set conversion (§5.1.13) and may perform unboxing conversion (§5.1.8).

The type of a multiplicative expression is the promoted type of its operands.

If the promoted type is int or long, then integer arithmetic is performed.

If the promoted type is float or double, then floating-point arithmetic is performed.
15.17.1 Multiplication Operator *

The binary * operator performs multiplication, producing the product of its operands.

Multiplication is a commutative operation if the operand expressions have no side effects.

Integer multiplication is associative when the operands are all of the same type.
Floating-point multiplication is not associative.

If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values.

The result of a floating-point multiplication is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.
- In the remaining cases, where neither an infinity nor NaN is involved, the exact mathematical product is computed. A floating-point value set is then chosen:
  - If the multiplication expression is FP-strict (§15.4):
    ‣ If the type of the multiplication expression is float, then the float value set must be chosen.
    ‣ If the type of the multiplication expression is double, then the double value set must be chosen.
  - If the multiplication expression is not FP-strict:
    ‣ If the type of the multiplication expression is float, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
    ‣ If the type of the multiplication expression is double, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.
Next, a value must be chosen from the chosen value set to represent the product. If the magnitude of the product is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the product is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, or loss of information may occur, evaluation of a multiplication operator * never throws a run-time exception.

### 15.17.2 Division Operator /

The binary / operator performs division, producing the quotient of its operands. The left-hand operand is the *dividend* and the right-hand operand is the *divisor*.

Integer division rounds toward 0. That is, the quotient produced for operands \( n \) and \( d \) that are integers after binary numeric promotion (§5.6.2) is an integer value \( q \) whose magnitude is as large as possible while satisfying \( |d \cdot q| \leq |n| \). Moreover, \( q \) is positive when \( |n| \geq |d| \) and \( n \) and \( d \) have the same sign, but \( q \) is negative when \( |n| \geq |d| \) and \( n \) and \( d \) have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is \(-1\), then integer overflow occurs and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an *ArithmeticException* is thrown.

The result of a floating-point division is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.
- Division of a finite value by an infinity results in a signed zero. The sign is determined by the rule stated above.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero. The sign is determined by the rule stated above.
• Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule stated above.

• In the remaining cases, where neither an infinity nor NaN is involved, the exact mathematical quotient is computed. A floating-point value set is then chosen:
  – If the division expression is FP-strict (§15.4):
    › If the type of the division expression is float, then the float value set must be chosen.
    › If the type of the division expression is double, then the double value set must be chosen.
  – If the division expression is not FP-strict:
    › If the type of the division expression is float, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
    › If the type of the division expression is double, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the quotient.

If the magnitude of the quotient is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the quotient is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, division by zero, or loss of information may occur, evaluation of a floating-point division operator / never throws a runtime exception.

15.17.3 Remainder Operator %

The binary % operator is said to yield the remainder of its operands from an implied division; the left-hand operand is the dividend and the right-hand operand is the divisor.

In C and C++, the remainder operator accepts only integral operands, but in the Java programming language, it also accepts floating-point operands.

The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that \((a/b) \times b + (a \% b)\) is equal to \(a\).
This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is $-1$ (the remainder is 0).

It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

If the value of the divisor for an integer remainder operator is 0, then an `ArithmeticException` is thrown.

**Example 15.17.3-1. Integer Remainder Operator**

```java
class Test1 {
    public static void main(String[] args) {
        int a = 5%3; // 2
        int b = 5/3; // 1
        System.out.println("5%3 produces " + a +
                           " (note that 5/3 produces " + b + ")");

        int c = 5%(-3); // 2
        int d = 5/(-3); // -1
        System.out.println("5%(-3) produces " + c +
                            " (note that 5/(-3) produces " + d + ")");

        int e = (-5)%3; // -2
        int f = (-5)/3; // -1
        System.out.println("(-5)%3 produces " + e +
                            " (note that (-5)/3 produces " + f + ")");

        int g = (-5)%(-3); // -2
        int h = (-5)/(-3); // 1
        System.out.println("(-5)%(-3) produces " + g +
                            " (note that (-5)/(-3) produces " + h + ")");
    }
}
```

This program produces the output:

```
5%3 produces 2 (note that 5/3 produces 1)
5%(-3) produces 2 (note that 5/(-3) produces -1)
(-5)%3 produces -2 (note that (-5)/3 produces -1)
(-5)%(-3) produces -2 (note that (-5)/(-3) produces 1)
```

The result of a floating-point remainder operation as computed by the `%` operator is *not* the same as that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java programming language
defines % on floating-point operations to behave in a manner analogous to that of the integer remainder operator; this may be compared with the C library function fmod. The IEEE 754 remainder operation may be computed by the library routine Math.IEEEremainder.

The result of a floating-point remainder operation is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder \( r \) from the division of a dividend \( n \) by a divisor \( d \) is defined by the mathematical relation \( r = n - (d \cdot q) \) where \( q \) is an integer that is negative only if \( n/d \) is negative and positive only if \( n/d \) is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of \( n \) and \( d \).

Evaluation of a floating-point remainder operator % never throws a run-time exception, even if the right-hand operand is zero. Overflow, underflow, or loss of precision cannot occur.

**Example 15.17.3-2. Floating-Point Remainder Operator**

```java
class Test2 {
    public static void main(String[] args) {
        double a = 5.0%3.0;  // 2.0
        System.out.println("5.0%3.0 produces " + a);

        double b = 5.0%(-3.0);  // 2.0
        System.out.println("5.0%(-3.0) produces " + b);

        double c = (-5.0)%3.0;  // -2.0
        System.out.println("(-5.0)%3.0 produces " + c);

        double d = (-5.0)%(-3.0);  // -2.0
        System.out.println("(-5.0)%(-3.0) produces " + d);
    }
}
```

This program produces the output:
Additive Operators EXPRESSIONS

5.0%3.0 produces 2.0
5.0%(-3.0) produces 2.0
(-5.0)%3.0 produces -2.0
(-5.0)%(-3.0) produces -2.0

15.18 Additive Operators

The operators + and – are called the additive operators.

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression – MultiplicativeExpression

The additive operators have the same precedence and are syntactically left-associative (they group left-to-right).

If the type of either operand of a + operator is String, then the operation is string concatenation.

Otherwise, the type of each of the operands of the + operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

In every case, the type of each of the operands of the binary – operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

15.18.1 String Concatenation Operator +

If only one operand expression is of type String, then string conversion (§5.1.11) is performed on the other operand to produce a string at run time.

The result of string concatenation is a reference to a String object that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

The String object is newly created (§12.5) unless the expression is a constant expression (§15.28).

An implementation may choose to perform conversion and concatenation in one step to avoid creating and then discarding an intermediate String object. To increase the performance of repeated string concatenation, a Java compiler may use the StringBuffer class or a similar technique to reduce the number of intermediate String objects that are created by evaluation of an expression.
For primitive types, an implementation may also optimize away the creation of a wrapper object by converting directly from a primitive type to a string.

Example 15.18.1-1. String Concatenation

The example expression:

"The square root of 2 is " + Math.sqrt(2)

produces the result:

"The square root of 2 is 1.4142135623730952"

The + operator is syntactically left-associative, no matter whether it is determined by type analysis to represent string concatenation or numeric addition. In some cases care is required to get the desired result. For example, the expression:

a + b + c

is always regarded as meaning:

(a + b) + c

Therefore the result of the expression:

1 + 2 + " fiddlers"

is:

"3 fiddlers"

but the result of:

"fiddlers " + 1 + 2

is:

"fiddlers 12"

Example 15.18.1-2. String Concatenation and Conditionals

In this jocular little example:

class Bottles {
    static void printSong(Object stuff, int n) {
        String plural = (n == 1) ? "" : "s";
        loop: while (true) {
            System.out.println(n + " bottle" + plural
                + " of " + stuff + " on the wall,");
        }
    }
}
System.out.println(n + " bottle" + plural + " of " + stuff + ";");
System.out.println("You take one down " + "and pass it around:");
--n;
plural = (n == 1) ? "" : "s";
if (n == 0)
    break loop;
System.out.println(n + " bottle" + plural + " of " + stuff + " on the wall!");
System.out.println();
}
System.out.println("No bottles of " + stuff + " on the wall!");

public static void main(String[] args) {
    printSong("slime", 3);
}

the method printSong will print a version of a children's song. Popular values for stuff include "pop" and "beer"; the most popular value for n is 100. Here is the output that results from running the program:

3 bottles of slime on the wall,
3 bottles of slime;
You take one down and pass it around:
2 bottles of slime on the wall!

2 bottles of slime on the wall,
2 bottles of slime;
You take one down and pass it around:
1 bottle of slime on the wall!

1 bottle of slime on the wall,
1 bottle of slime;
You take one down and pass it around:
No bottles of slime on the wall!

In the code, note the careful conditional generation of the singular "bottle" when appropriate rather than the plural "bottles"; note also how the string concatenation operator was used to break the long constant string:

"You take one down and pass it around:"

into two pieces to avoid an inconveniently long line in the source code.
15.18.2 Additive Operators (+ and −) for Numeric Types

The binary + operator performs addition when applied to two operands of numeric type, producing the sum of the operands.

The binary − operator performs subtraction, producing the difference of two numeric operands.

Binary numeric promotion is performed on the operands (§5.6.2).

Note that binary numeric promotion performs value set conversion (§5.1.13) and may perform unboxing conversion (§5.1.8).

The type of an additive expression on numeric operands is the promoted type of its operands.

If this promoted type is int or long, then integer arithmetic is performed.

If this promoted type is float or double, then floating-point arithmetic is performed.

Addition is a commutative operation if the operand expressions have no side effects.

Integer addition is associative when the operands are all of the same type.

Floating-point addition is not associative.

If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format. If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.

The result of a floating-point addition is determined using the following rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two zeros of opposite sign is positive zero.
- The sum of two zeros of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
• The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.

• In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the exact mathematical sum is computed. A floating-point value set is then chosen:
  – If the addition expression is FP-strict (§15.4):
    › If the type of the addition expression is float, then the float value set must be chosen.
    › If the type of the addition expression is double, then the double value set must be chosen.
  – If the addition expression is not FP-strict:
    › If the type of the addition expression is float, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
    › If the type of the addition expression is double, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the sum.

If the magnitude of the sum is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the sum is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

The binary \(-\) operator performs subtraction when applied to two operands of numeric type, producing the difference of its operands; the left-hand operand is the minuend and the right-hand operand is the subtrahend.

For both integer and floating-point subtraction, it is always the case that \(a-b\) produces the same result as \(a+(-b)\).

Note that, for integer values, subtraction from zero is the same as negation. However, for floating-point operands, subtraction from zero is not the same as negation, because if \(x\) is +0.0, then 0.0-\(x\) is +0.0, but -\(x\) is -0.0.

Despite the fact that overflow, underflow, or loss of information may occur, evaluation of a numeric additive operator never throws a run-time exception.
15.19 Shift Operators

The operators << (left shift), >> (signed right shift), and >>> (unsigned right shift) are called the shift operators. The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

\[
\text{ShiftExpression:} \\
\hspace{1em} \text{AdditiveExpression} \\
\hspace{2em} \text{ShiftExpression} \hspace{0.5em}<<\hspace{0.5em} \text{AdditiveExpression} \\
\hspace{2em} \text{ShiftExpression} \hspace{0.5em}>>\hspace{0.5em} \text{AdditiveExpression} \\
\hspace{2em} \text{ShiftExpression} \hspace{0.5em}>>\hspace{0.5em} \text{AdditiveExpression}
\]

The shift operators are syntactically left-associative (they group left-to-right).

Unary numeric promotion (§5.6.1) is performed on each operand separately. (Binary numeric promotion (§5.6.2) is not performed on the operands.)

It is a compile-time error if the type of each of the operands of a shift operator, after unary numeric promotion, is not a primitive integral type.

The type of the shift expression is the promoted type of the left-hand operand.

If the promoted type of the left-hand operand is \textit{int}, then only the five lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator \& (§15.22.1) with the mask value \texttt{0x1f} (\texttt{0b11111}). The shift distance actually used is therefore always in the range 0 to 31, inclusive.

If the promoted type of the left-hand operand is \textit{long}, then only the six lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator \& (§15.22.1) with the mask value \texttt{0x3f} (\texttt{0b111111}). The shift distance actually used is therefore always in the range 0 to 63, inclusive.

At run time, shift operations are performed on the two's-complement integer representation of the value of the left operand.

The value of \( n << s \) is \( n \) left-shifted \( s \) bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power \( s \).

The value of \( n >> s \) is \( n \) right-shifted \( s \) bit positions with sign-extension. The resulting value is \textit{floor}(\( n / 2^s \)). For non-negative values of \( n \), this is equivalent to truncating integer division, as computed by the integer division operator \( / \), by two to the power \( s \).
The value of \( n \gg\gg s \) is \( n \) right-shifted \( s \) bit positions with zero-extension, where:

- If \( n \) is positive, then the result is the same as that of \( n \gg s \).
- If \( n \) is negative and the type of the left-hand operand is \textit{int}, then the result is equal to that of the expression \( (n \gg s) + (2 << \neg s) \).
- If \( n \) is negative and the type of the left-hand operand is \textit{long}, then the result is equal to that of the expression \( (n \gg s) + (2L <\!< \neg s) \).

The added term \((2 <\!< \neg s)\) or \((2L <\!< \neg s)\) cancels out the propagated sign bit.

Note that, because of the implicit masking of the right-hand operand of a shift operator, \( \neg s \) as a shift distance is equivalent to \( 31-\neg s \) when shifting an \textit{int} value and to \( 63-\neg s \) when shifting a \textit{long} value.

### 15.20 Relational Operators

The numerical comparison operators \(<, >, \leq, \text{ and } \geq\), and the \texttt{instanceof} operator, are called the \textit{relational operators}.

\[
\text{RelationalExpression:} \\
\quad \text{ShiftExpression} \\
\quad \text{RelationalExpression} < \text{ShiftExpression} \\
\quad \text{RelationalExpression} > \text{ShiftExpression} \\
\quad \text{RelationalExpression} \leq \text{ShiftExpression} \\
\quad \text{RelationalExpression} \geq \text{ShiftExpression} \\
\quad \text{RelationalExpression} \text{ instanceof ReferenceType}
\]

The relational operators are syntactically left-associative (they group left-to-right).

However, this fact is not useful. For example, \( a<b<c \) parses as \( (a<b)<c \), which is always a compile-time error, because the type of \( a<b \) is always \textit{boolean} and \(<\) is not an operator on \textit{boolean} values.

The type of a relational expression is always \textit{boolean}.

### 15.20.1 Numerical Comparison Operators \(<, \leq, >, \text{ and } \geq\)

The type of each of the operands of a numerical comparison operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.
Binary numeric promotion is performed on the operands (§5.6.2).

Note that binary numeric promotion performs value set conversion (§5.1.13) and may perform unboxing conversion (§5.1.8).

If the promoted type of the operands is `int` or `long`, then signed integer comparison is performed.

If the promoted type is `float` or `double`, then floating-point comparison is performed.

Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

The result of a floating-point comparison, as determined by the specification of the IEEE 754 standard, is:

- If either operand is NaN, then the result is false.
- All values other than NaN are ordered, with negative infinity less than all finite values, and positive infinity greater than all finite values.
- Positive zero and negative zero are considered equal.

For example, `-0.0<0.0` is false, but `-0.0<=0.0` is true.

Note, however, that the methods `Math.min` and `Math.max` treat negative zero as being strictly smaller than positive zero.

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the `<` operator is `true` if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `<=` operator is `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>` operator is `true` if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>=` operator is `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is `false`.
15.20.2 Type Comparison Operator instanceof

The type of the RelationalExpression operand of the instanceof operator must be a reference type or the null type, or a compile-time error occurs.

It is a compile-time error if the ReferenceType mentioned after the instanceof operator does not denote a reference type that is reifiable (§4.7).

If a cast of the RelationalExpression to the ReferenceType would be rejected as a compile-time error (§15.16), then the instanceof relational expression likewise produces a compile-time error. In such a situation, the result of the instanceof expression could never be true.

At run time, the result of the instanceof operator is true if the value of the RelationalExpression is not null and the reference could be cast to the ReferenceType without raising a ClassCastException. Otherwise the result is false.

Example 15.20.2-1. The instanceof Operator

class Point { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        Element e = new Element();
        if (e instanceof Point) { // compile-time error
            System.out.println("I get your point!");
            p = (Point)e; // compile-time error
        }
    }
}

This program results in two compile-time errors. The cast (Point)e is incorrect because no instance of Element or any of its possible subclasses (none are shown here) could possibly be an instance of any subclass of Point. The instanceof expression is incorrect for exactly the same reason. If, on the other hand, the class Point were a subclass of Element (an admittedly strange notion in this example):

class Point extends Element { int x, y; }

then the cast would be possible, though it would require a run-time check, and the instanceof expression would then be sensible and valid. The cast (Point)e would never raise an exception because it would not be executed if the value of e could not correctly be cast to type Point.
15.21 Equality Operators

The operators == (equal to) and != (not equal to) are called the equality operators.

EqualityExpression:
   RelationalExpression
   EqualityExpression == RelationalExpression
   EqualityExpression != RelationalExpression

The equality operators are syntactically left-associative (they group left-to-right).

   However, this fact is essentially never useful. For example, a==b==c parses as (a==b)==c. The result type of a==b is always boolean, and c must therefore be of type boolean or a compile-time error occurs. Thus, a==b==c does not test to see whether a, b, and c are all equal.

The equality operators are commutative if the operand expressions have no side effects.

The equality operators are analogous to the relational operators except for their lower precedence. Thus, a<b==c<d is true whenever a<b and c<d have the same truth value.

The equality operators may be used to compare two operands that are convertible (§5.1.8) to numeric type, or two operands of type boolean or Boolean, or two operands that are each of either reference type or the null type. All other cases result in a compile-time error.

The type of an equality expression is always boolean.

In all cases, a!=b produces the same result as !(a==b).

15.21.1 Numerical Equality Operators == and !=

If the operands of an equality operator are both of numeric type, or one is of numeric type and the other is convertible (§5.1.8) to numeric type, binary numeric promotion is performed on the operands (§5.6.2).

   Note that binary numeric promotion performs value set conversion (§5.1.13) and may perform unboxing conversion (§5.1.8).

If the promoted type of the operands is int or long, then an integer equality test is performed.
If the promoted type is `float` or `double`, then a floating-point equality test is performed.

Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

Floating-point equality testing is performed in accordance with the rules of the IEEE 754 standard:

- If either operand is NaN, then the result of `==` is `false` but the result of `!=` is `true`. Indeed, the test `x!=x` is `true` if and only if the value of `x` is NaN.

  The methods `Float.isNaN` and `Double.isNaN` may also be used to test whether a value is NaN.

- Positive zero and negative zero are considered equal.

  For example, `-0.0==0.0` is `true`.

- Otherwise, two distinct floating-point values are considered unequal by the equality operators.

  In particular, there is one value representing positive infinity and one value representing negative infinity; each compares equal only to itself, and each compares unequal to all other values.

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the `==` operator is `true` if the value of the left-hand operand is equal to the value of the right-hand operand; otherwise, the result is `false`.

- The value produced by the `!=` operator is `true` if the value of the left-hand operand is not equal to the value of the right-hand operand; otherwise, the result is `false`.

### 15.21.2 Boolean Equality Operators `==` and `!=`

If the operands of an equality operator are both of type `boolean`, or if one operand is of type `boolean` and the other is of type `Boolean`, then the operation is boolean equality.

The boolean equality operators are associative.

If one of the operands is of type `Boolean`, it is subjected to unboxing conversion (§5.1.8).
The result of \(==\) is `true` if the operands (after any required unboxing conversion) are both `true` or both `false`; otherwise, the result is `false`.

The result of `!=` is `false` if the operands are both `true` or both `false`; otherwise, the result is `true`.

Thus `!=` behaves the same as \(^\text{§15.22.2}\) when applied to `boolean` operands.

### 15.21.3 Reference Equality Operators \(==\) and `!=`

If the operands of an equality operator are both of either reference type or the null type, then the operation is object equality.

It is a compile-time error if it is impossible to convert the type of either operand to the type of the other by a casting conversion (\(^\text{§5.5}\)). The run-time values of the two operands would necessarily be unequal (ignoring the case where both values are `null`).

At run time, the result of \(==\) is `true` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `false`.

The result of `!=` is `false` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `true`.

While \(==\) may be used to compare references of type `String`, such an equality test determines whether or not the two operands refer to the same `String` object. The result is `false` if the operands are distinct `String` objects, even if they contain the same sequence of characters (\(^\text{§3.10.5}\)). The contents of two strings `s` and `t` can be tested for equality by the method invocation `s.equals(t)`.

### 15.22 Bitwise and Logical Operators

The `bitwise operators` and `logical operators` include the AND operator `&`, exclusive OR operator `^`, and inclusive OR operator `|`.

```
AndExpression:
  EqualityExpression
  AndExpression & EqualityExpression

ExclusiveOrExpression:
  AndExpression
  ExclusiveOrExpression ^ AndExpression
```
**InclusiveOrExpression:**

```plaintext
InclusiveOrExpression:  
  ExclusiveOrExpression  
  InclusiveOrExpression | ExclusiveOrExpression
```

These operators have different precedence, with \& having the highest precedence and | the lowest precedence.

Each of these operators is syntactically left-associative (each groups left-to-right).

Each operator is commutative if the operand expressions have no side effects.

Each operator is associative.

The bitwise and logical operators may be used to compare two operands of numeric type or two operands of type `boolean`. All other cases result in a compile-time error.

### 15.22.1 Integer Bitwise Operators `&`, `^`, and `|`

When both operands of an operator & , ^, or | are of a type that is convertible (§5.1.8) to a primitive integral type, binary numeric promotion is first performed on the operands (§5.6.2).

The type of the bitwise operator expression is the promoted type of the operands.

For & , the result value is the bitwise AND of the operand values.

For ^ , the result value is the bitwise exclusive OR of the operand values.

For | , the result value is the bitwise inclusive OR of the operand values.

For example, the result of the expression:

```
0xff00 & 0xf0f0
```

is:

```
0xf000
```

The result of the expression:

```
0xff00 ^ 0xf0f0
```

is:

```
0x0ff0
```

The result of the expression:
0xff00 | 0xf0f0

is:

0xffff

15.22.2 Boolean Logical Operators &, ^, and |

When both operands of a &, ^, or | operator are of type boolean or Boolean, then the type of the bitwise operator expression is boolean. In all cases, the operands are subject to unboxing conversion (§5.1.8) as necessary.

For &, the result value is true if both operand values are true; otherwise, the result is false.

For ^, the result value is true if the operand values are different; otherwise, the result is false.

For |, the result value is false if both operand values are false; otherwise, the result is true.

15.23 Conditional-And Operator &&

The conditional-and operator && is like & (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is true.

\[
\text{ConditionalAndExpression:} \\
\quad \text{InclusiveOrExpression} \\
\quad \text{ConditionalAndExpression} \quad \&\& \quad \text{InclusiveOrExpression}
\]

The conditional-and operator is syntactically left-associative (it groups left-to-right).

The conditional-and operator is fully associative with respect to both side effects and result value. That is, for any expressions \(a\), \(b\), and \(c\), evaluation of the expression \(((a) \&\& (b)) \&\& (c)\) produces the same result, with the same side effects occurring in the same order, as evaluation of the expression \((a) \&\& ((b) \&\& (c))\).

Each operand of the conditional-and operator must be of type boolean or Boolean, or a compile-time error occurs.

The type of a conditional-and expression is always boolean.
At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8).

If the resulting value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not evaluated.

If the value of the left-hand operand is `true`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8). The resulting value becomes the value of the conditional-and expression.

Thus, `&&` computes the same result as `&` on `boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

### 15.24 Conditional-Or Operator `||`

The conditional-or operator `||` operator is like `|` (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is `false`.

```
ConditionalOrExpression:
  ConditionalAndExpression
  ConditionalOrExpression  ||  ConditionalAndExpression
```

The conditional-or operator is syntactically left-associative (it groups left-to-right).

The conditional-or operator is fully associative with respect to both side effects and result value. That is, for any expressions `a`, `b`, and `c`, evaluation of the expression `((a) || (b)) || (c)` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(a) || ((b) || (c))`.

Each operand of the conditional-or operator must be of type `boolean` or `Boolean`, or a compile-time error occurs.

The type of a conditional-or expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8).

If the resulting value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not evaluated.

If the value of the left-hand operand is `false`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8). The resulting value becomes the value of the conditional-or expression.
Thus, \( |\ | \) computes the same result as \( | \) on boolean or Boolean operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

### 15.25 Conditional Operator \( ? : \)

The conditional operator \( ? : \) uses the boolean value of one expression to decide which of two other expressions should be evaluated.

\[
\text{ConditionalExpression:} \\
\text{ConditionalOrExpression} \\
\text{ConditionalOrExpression} ? \text{Expression} : \text{ConditionalExpression} \\
\text{ConditionalOrExpression} ? \text{Expression} : \text{LambdaExpression}
\]

The conditional operator is syntactically right-associative (it groups right-to-left). Thus, \( a?b:c?d:e?f:g \) means the same as \( a?b:(c?d:(e?f:g)) \).

The conditional operator has three operand expressions. \( ? \) appears between the first and second expressions, and \( : \) appears between the second and third expressions.

The first expression must be of type boolean or Boolean, or a compile-time error occurs.

It is a compile-time error for either the second or the third operand expression to be an invocation of a void method.

In fact, by the grammar of expression statements (§14.8), it is not permitted for a conditional expression to appear in any context where an invocation of a void method could appear.

There are three kinds of conditional expressions, classified according to the second and third operand expressions: boolean conditional expressions, numeric conditional expressions, and reference conditional expressions. The classification rules are as follows:

- If both the second and the third operand expressions are boolean expressions, the conditional expression is a boolean conditional expression.

For the purpose of classifying a conditional, the following expressions are boolean expressions:

- An expression of a standalone form (§15.2) that has type boolean or Boolean.
- A parenthesized boolean expression (§15.8.5).
- A class instance creation expression (§15.9) for class Boolean.

- A method invocation expression (§15.12) for which the chosen most specific method (§15.12.2.5) has return type boolean or Boolean.

  Note that, for a generic method, this is the type before instantiating the method's type arguments.

- A boolean conditional expression.

  • If both the second and the third operand expressions are numeric expressions, the conditional expression is a numeric conditional expression.

  For the purpose of classifying a conditional, the following expressions are numeric expressions:

  - An expression of a standalone form (§15.2) with a type that is convertible to a numeric type (§4.2, §5.1.8).

  - A parenthesized numeric expression (§15.8.5).

  - A class instance creation expression (§15.9) for a class that is convertible to a numeric type.

  - A method invocation expression (§15.12) for which the chosen most specific method (§15.12.2.5) has a return type that is convertible to a numeric type.

    Note that, for a generic method, this is the type before instantiating the method's type arguments.

  - A numeric conditional expression.

  • Otherwise, the conditional expression is a reference conditional expression.

The process for determining the type of a conditional expression depends on the kind of conditional expression, as outlined in the following sections.

The following tables summarize the rules above by giving the type of a conditional expression for all possible types of its second and third operands. bnp(..) means to apply binary numeric promotion. The form "τ | bnp(..)" is used where one operand is a constant expression of type int and may be representable in type τ, where binary numeric promotion is used if the operand is not representable in type τ. The operand type Object means any reference type other than the null type and the eight wrapper classes Boolean, Byte, Short, Character, Integer, Long, Float, Double.
### Table 15.25-A. Conditional expression type (Primitive 3rd operand, Part I)

<table>
<thead>
<tr>
<th>3rd</th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
<td>short</td>
<td>bnp(byte,char)</td>
<td>byte</td>
</tr>
<tr>
<td>Byte</td>
<td>byte</td>
<td>short</td>
<td>bnp(Byte,char)</td>
<td>byte</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>short</td>
<td>bnp(short,char)</td>
<td>short</td>
</tr>
<tr>
<td>Short</td>
<td>short</td>
<td>short</td>
<td>bnp(Short,char)</td>
<td>short</td>
</tr>
<tr>
<td>char</td>
<td>bnp(char,byte)</td>
<td>bnp(char,short)</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>Character</td>
<td>bnp(Character,byte)</td>
<td>bnp(Character,short)</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>int</td>
<td>byte</td>
<td>bnp(int,byte)</td>
<td>short</td>
<td>bnp(int,short)</td>
</tr>
<tr>
<td>Integer</td>
<td>bnp(Integer,byte)</td>
<td>bnp(Integer,short)</td>
<td>bnp(Integer,char)</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>bnp(long,byte)</td>
<td>bnp(long,short)</td>
<td>bnp(long,char)</td>
<td>bnp(long,int)</td>
</tr>
<tr>
<td>Long</td>
<td>bnp(Long,byte)</td>
<td>bnp(Long,short)</td>
<td>bnp(Long,char)</td>
<td>bnp(Long,int)</td>
</tr>
<tr>
<td>float</td>
<td>bnp(float,byte)</td>
<td>bnp(float,short)</td>
<td>bnp(float,char)</td>
<td>bnp(float,int)</td>
</tr>
<tr>
<td>Float</td>
<td>bnp(Float,byte)</td>
<td>bnp(Float,short)</td>
<td>bnp(Float,char)</td>
<td>bnp(Float,int)</td>
</tr>
<tr>
<td>double</td>
<td>bnp(double,byte)</td>
<td>bnp(double,short)</td>
<td>bnp(double,char)</td>
<td>bnp(double,int)</td>
</tr>
<tr>
<td>Double</td>
<td>bnp(Double,byte)</td>
<td>bnp(Double,short)</td>
<td>bnp(Double,char)</td>
<td>bnp(Double,int)</td>
</tr>
<tr>
<td>boolean</td>
<td>lub(Boolean,Byte)</td>
<td>lub(Boolean,Short)</td>
<td>lub(Boolean,Character)</td>
<td>lub(Boolean,Integer)</td>
</tr>
<tr>
<td>Boolean</td>
<td>lub(Boolean,Byte)</td>
<td>lub(Boolean,Short)</td>
<td>lub(Boolean,Character)</td>
<td>lub(Boolean,Integer)</td>
</tr>
<tr>
<td>null</td>
<td>lub(null,Byte)</td>
<td>lub(null,Short)</td>
<td>lub(null,Character)</td>
<td>lub(null,Integer)</td>
</tr>
<tr>
<td>Object</td>
<td>lub(Object,Byte)</td>
<td>lub(Object,Short)</td>
<td>lub(Object,Character)</td>
<td>lub(Object,Integer)</td>
</tr>
</tbody>
</table>
### Table 15.25-B. Conditional expression type (Primitive 3rd operand, Part II)

<table>
<thead>
<tr>
<th>3rd →</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd ↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>bnp(byte,long)</td>
<td>bnp(byte,float)</td>
<td>bnp(byte,double)</td>
<td>lub(Byte,Boolean)</td>
</tr>
<tr>
<td>Byte</td>
<td>bnp(Byte,long)</td>
<td>bnp(Byte,float)</td>
<td>bnp(Byte,double)</td>
<td>lub(Byte,Boolean)</td>
</tr>
<tr>
<td>short</td>
<td>bnp(short,long)</td>
<td>bnp(short,float)</td>
<td>bnp(short,double)</td>
<td>lub(Short,Boolean)</td>
</tr>
<tr>
<td>Short</td>
<td>bnp(Short,long)</td>
<td>bnp(Short,float)</td>
<td>bnp(Short,double)</td>
<td>lub(Short,Boolean)</td>
</tr>
<tr>
<td>char</td>
<td>bnp(char,long)</td>
<td>bnp(char,float)</td>
<td>bnp(char,double)</td>
<td>lub(Character,Boolean)</td>
</tr>
<tr>
<td>Character</td>
<td>bnp(Character,long)</td>
<td>bnp(Character,float)</td>
<td>bnp(Character,double)</td>
<td>lub(Character,Boolean)</td>
</tr>
<tr>
<td>int</td>
<td>bnp(int,long)</td>
<td>bnp(int,float)</td>
<td>bnp(int,double)</td>
<td>lub(Integer,Boolean)</td>
</tr>
<tr>
<td>Integer</td>
<td>bnp(Integer,long)</td>
<td>bnp(Integer,float)</td>
<td>bnp(Integer,double)</td>
<td>lub(Integer,Boolean)</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>bnp(long,float)</td>
<td>bnp(long,double)</td>
<td>lub(Long,Boolean)</td>
</tr>
<tr>
<td>Long</td>
<td>long</td>
<td>bnp(Long,float)</td>
<td>bnp(Long,double)</td>
<td>lub(Long,Boolean)</td>
</tr>
<tr>
<td>float</td>
<td>bnp(float,long)</td>
<td>float</td>
<td>bnp(float,double)</td>
<td>lub(Float,Boolean)</td>
</tr>
<tr>
<td>Float</td>
<td>bnp(Float,long)</td>
<td>float</td>
<td>bnp(Float,double)</td>
<td>lub(Float,Boolean)</td>
</tr>
<tr>
<td>double</td>
<td>bnp(double,long)</td>
<td>bnp(double,float)</td>
<td>double</td>
<td>lub(Double,Boolean)</td>
</tr>
<tr>
<td>Double</td>
<td>bnp(Double,long)</td>
<td>bnp(Double,float)</td>
<td>double</td>
<td>lub(Double,Boolean)</td>
</tr>
<tr>
<td>boolean</td>
<td>lub(Boolean,long)</td>
<td>lub(Boolean,float)</td>
<td>lub(Boolean,double)</td>
<td>boolean</td>
</tr>
<tr>
<td>Boolean</td>
<td>lub(Boolean,long)</td>
<td>lub(Boolean,float)</td>
<td>lub(Boolean,double)</td>
<td>boolean</td>
</tr>
<tr>
<td>null</td>
<td>lub(null,long)</td>
<td>lub(null,Float)</td>
<td>lub(null,Double)</td>
<td>lub(null,Boolean)</td>
</tr>
<tr>
<td>Object</td>
<td>lub(Object,long)</td>
<td>lub(Object,Float)</td>
<td>lub(Object,Double)</td>
<td>lub(Object,Boolean)</td>
</tr>
</tbody>
</table>
### Table 15.25-C. Conditional expression type (Reference 3rd operand, Part I)

<table>
<thead>
<tr>
<th>3rd →</th>
<th>Byte</th>
<th>Short</th>
<th>Character</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte</td>
<td>short</td>
<td>bnp(byte,Character)</td>
<td>bnp(byte,Integer)</td>
</tr>
<tr>
<td></td>
<td>Byte</td>
<td>short</td>
<td>bnp(Byte,Character)</td>
<td>bnp(Byte,Integer)</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>short</td>
<td>bnp(short,Character)</td>
<td>bnp(short,Integer)</td>
</tr>
<tr>
<td></td>
<td>Short</td>
<td>Short</td>
<td>bnp(Short,Character)</td>
<td>bnp(Short,Integer)</td>
</tr>
<tr>
<td>char</td>
<td>bnp(char,Byte)</td>
<td>bnp(char,Short)</td>
<td>char</td>
<td>bnp(char,Integer)</td>
</tr>
<tr>
<td>Character</td>
<td>bnp(Character,Byte)</td>
<td>bnp(Character,Short)</td>
<td>Character</td>
<td>bnp(Character,Integer)</td>
</tr>
<tr>
<td>int</td>
<td>byte</td>
<td>bnp(int,Byte)</td>
<td>short</td>
<td>bnp(int,Short)</td>
</tr>
<tr>
<td>Integer</td>
<td>bnp(Integer,Byte)</td>
<td>bnp(Integer,Short)</td>
<td>bnp(Integer,Character)</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>bnp(long,Byte)</td>
<td>bnp(long,Short)</td>
<td>bnp(long,Character)</td>
<td>bnp(long,Integer)</td>
</tr>
<tr>
<td>Long</td>
<td>bnp(Long,Byte)</td>
<td>bnp(Long,Short)</td>
<td>bnp(Long,Character)</td>
<td>bnp(Long,Integer)</td>
</tr>
<tr>
<td>float</td>
<td>bnp(float,Byte)</td>
<td>bnp(float,Short)</td>
<td>bnp(float,Character)</td>
<td>bnp(float,Integer)</td>
</tr>
<tr>
<td>Float</td>
<td>bnp(Float,Byte)</td>
<td>bnp(Float,Short)</td>
<td>bnp(Float,Character)</td>
<td>bnp(Float,Integer)</td>
</tr>
<tr>
<td>double</td>
<td>bnp(double,Byte)</td>
<td>bnp(double,Short)</td>
<td>bnp(double,Character)</td>
<td>bnp(double,Integer)</td>
</tr>
<tr>
<td>Double</td>
<td>bnp(Double,Byte)</td>
<td>bnp(Double,Short)</td>
<td>bnp(Double,Character)</td>
<td>bnp(Double,Integer)</td>
</tr>
<tr>
<td>boolean</td>
<td>lub(Boolean,Byte)</td>
<td>lub(Boolean,Short)</td>
<td>lub(Boolean,Character)</td>
<td>lub(Boolean,Integer)</td>
</tr>
<tr>
<td>Boolean</td>
<td>lub(Boolean,Byte)</td>
<td>lub(Boolean,Short)</td>
<td>lub(Boolean,Character)</td>
<td>lub(Boolean,Integer)</td>
</tr>
<tr>
<td>null</td>
<td>Byte</td>
<td>Short</td>
<td>Character</td>
<td>Integer</td>
</tr>
<tr>
<td>Object</td>
<td>lub(Object,Byte)</td>
<td>lub(Object,Short)</td>
<td>lub(Object,Character)</td>
<td>lub(Object,Integer)</td>
</tr>
</tbody>
</table>
Table 15.25-D. Conditional expression type (Reference 3rd operand, Part II)

<table>
<thead>
<tr>
<th>3rd →</th>
<th>Long</th>
<th>Float</th>
<th>Double</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd ↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>bnp(byte,Long)</td>
<td>bnp(byte,Float)</td>
<td>bnp(byte,Double)</td>
<td>lub(Byte,Boolean)</td>
</tr>
<tr>
<td>Byte</td>
<td>bnp(Byte,Long)</td>
<td>bnp(Byte,Float)</td>
<td>bnp(Byte,Double)</td>
<td>lub(Byte,Boolean)</td>
</tr>
<tr>
<td>short</td>
<td>bnp(short,Long)</td>
<td>bnp(short,Float)</td>
<td>bnp(short,Double)</td>
<td>lub(Short,Boolean)</td>
</tr>
<tr>
<td>Short</td>
<td>bnp(Short,Long)</td>
<td>bnp(Short,Float)</td>
<td>bnp(Short,Double)</td>
<td>lub(Short,Boolean)</td>
</tr>
<tr>
<td>char</td>
<td>bnp(char,Long)</td>
<td>bnp(char,Float)</td>
<td>bnp(char,Double)</td>
<td>lub(Character,Boolean)</td>
</tr>
<tr>
<td>Character</td>
<td>bnp(Character,Long)</td>
<td>bnp(Character,Float)</td>
<td>bnp(Character,Double)</td>
<td>lub(Character,Boolean)</td>
</tr>
<tr>
<td>int</td>
<td>bnp(int,Long)</td>
<td>bnp(int,Float)</td>
<td>bnp(int,Double)</td>
<td>lub(Integer,Boolean)</td>
</tr>
<tr>
<td>Integer</td>
<td>bnp(Integer,Long)</td>
<td>bnp(Integer,Float)</td>
<td>bnp(Integer,Double)</td>
<td>lub(Integer,Boolean)</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>bnp(long,Float)</td>
<td>bnp(long,Double)</td>
<td>lub(Long,Boolean)</td>
</tr>
<tr>
<td>Long</td>
<td>Long</td>
<td>bnp(Long,Float)</td>
<td>bnp(Long,Double)</td>
<td>lub(Long,Boolean)</td>
</tr>
<tr>
<td>float</td>
<td>bnp(float,Long)</td>
<td>float</td>
<td>bnp(float,Double)</td>
<td>lub(Float,Boolean)</td>
</tr>
<tr>
<td>Float</td>
<td>bnp(Float,Long)</td>
<td>Float</td>
<td>bnp(Float,Double)</td>
<td>lub(Float,Boolean)</td>
</tr>
<tr>
<td>double</td>
<td>bnp(double,Long)</td>
<td>bnp(double,Float)</td>
<td>double</td>
<td>lub(Double,Boolean)</td>
</tr>
<tr>
<td>Double</td>
<td>bnp(Double,Long)</td>
<td>bnp(Double,Float)</td>
<td>Double</td>
<td>lub(Double,Boolean)</td>
</tr>
<tr>
<td>boolean</td>
<td>lub(Boolean,Long)</td>
<td>lub(Boolean,Float)</td>
<td>lub(Boolean,Double)</td>
<td>boolean</td>
</tr>
<tr>
<td>Boolean</td>
<td>lub(Boolean,Long)</td>
<td>lub(Boolean,Float)</td>
<td>lub(Boolean,Double)</td>
<td>Boolean</td>
</tr>
<tr>
<td>null</td>
<td>Long</td>
<td>Float</td>
<td>Double</td>
<td>Boolean</td>
</tr>
<tr>
<td>Object</td>
<td>lub(Object,Long)</td>
<td>lub(Object,Float)</td>
<td>lub(Object,Double)</td>
<td>lub(Object,Boolean)</td>
</tr>
</tbody>
</table>
At run time, the first operand expression of the conditional expression is evaluated first. If necessary, unboxing conversion is performed on the result.

The resulting boolean value is then used to choose either the second or the third operand expression:

- If the value of the first operand is true, then the second operand expression is chosen.
- If the value of the first operand is false, then the third operand expression is chosen.
The chosen operand expression is then evaluated and the resulting value is converted to the type of the conditional expression as determined by the rules stated below.

This conversion may include boxing or unboxing conversion (§5.1.7, §5.1.8).

The operand expression not chosen is not evaluated for that particular evaluation of the conditional expression.

15.25.1 Boolean Conditional Expressions

Boolean conditional expressions are standalone expressions (§15.2).

The type of a boolean conditional expression is determined as follows:

- If the second and third operands are both of type `Boolean`, the conditional expression has type `Boolean`.
- Otherwise, the conditional expression has type `boolean`.

15.25.2 Numeric Conditional Expressions

Numeric conditional expressions are standalone expressions (§15.2).

The type of a numeric conditional expression is determined as follows:

- If the second and third operands have the same type, then that is the type of the conditional expression.
- If one of the second and third operands is of primitive type $\tau$, and the type of the other is the result of applying boxing conversion (§5.1.7) to $\tau$, then the type of the conditional expression is $\tau$.
- If one of the operands is of type `byte` or `Byte` and the other is of type `short` or `Short`, then the type of the conditional expression is `short`.
- If one of the operands is of type $\tau$ where $\tau$ is `byte`, `short`, or `char`, and the other operand is a constant expression (§15.28) of type `int` whose value is representable in type $\tau$, then the type of the conditional expression is $\tau$.
- If one of the operands is of type $\tau$, where $\tau$ is `Byte`, `Short`, or `Character`, and the other operand is a constant expression of type `int` whose value is representable in the type $\psi$ which is the result of applying unboxing conversion to $\tau$, then the type of the conditional expression is $\psi$. 
• Otherwise, binary numeric promotion (§5.6.2) is applied to the operand types, and the type of the conditional expression is the promoted type of the second and third operands.

Note that binary numeric promotion performs value set conversion (§5.1.13) and may perform unboxing conversion (§5.1.8).

15.25.3 Reference Conditional Expressions

A reference conditional expression is a poly expression if it appears in an assignment context or an invocation context (§5.2, §5.3). Otherwise, it is a standalone expression.

Where a poly reference conditional expression appears in a context of a particular kind with target type \( T \), its second and third operand expressions similarly appear in a context of the same kind with target type \( T \).

A poly reference conditional expression is compatible with a target type \( T \) if its second and third operand expressions are compatible with \( T \).

The type of a poly reference conditional expression is the same as its target type.

The type of a standalone reference conditional expression is determined as follows:

• If the second and third operands have the same type (which may be the null type), then that is the type of the conditional expression.

• If the type of one of the second and third operands is the null type, and the type of the other operand is a reference type, then the type of the conditional expression is that reference type.

• Otherwise, the second and third operands are of types \( S_1 \) and \( S_2 \) respectively. Let \( T_1 \) be the type that results from applying boxing conversion to \( S_1 \), and let \( T_2 \) be the type that results from applying boxing conversion to \( S_2 \). The type of the conditional expression is the result of applying capture conversion (§5.1.10) to \( \text{lub}(T_1, T_2) \).

Because reference conditional expressions can be poly expressions, they can "pass down" context to their operands. This allows lambda expressions and method reference expressions to appear as operands:

```
return ... ? (x -> x) : (x -> -x);
```

It also allows use of extra information to improve type checking of generic method invocations. Prior to Java SE 8, this assignment was well-typed:

```
List<String> ls = Arrays.asList();
```
but this was not:

\[
\text{List}\langle\text{String}\rangle\ \text{ls} = \ldots \ ? \ \text{Arrays.asList()} : \ \text{Arrays.asList("a","b")};
\]

The rules above allow both assignments to be considered well-typed.

Note that a reference conditional expression does not have to contain a poly expression as an operand in order to be a poly expression. It is a poly expression simply by virtue of the context in which it appears. For example, in the following code, the conditional expression is a poly expression, and each operand is considered to be in an assignment context targeting `Class<? super Integer>`:

\[
\text{Class}<? \ \text{super Integer}>\ \text{choose}(\text{boolean } b, \\
\quad \text{Class}<\text{Integer}> \ c1, \\
\quad \text{Class}<\text{Number}> \ c2) \ { \\
\quad \quad \text{return } b \ ? \ c1 \ : \ c2; \\
\}
\]

If the conditional expression was not a poly expression, then a compile-time error would occur, as its type would be `lub(Class<\text{Integer}>, Class<\text{Number}>)` which is incompatible with the return type of `choose`.

## 15.26 Assignment Operators

There are 12 assignment operators; all are syntactically right-associative (they group right-to-left). Thus, `a=b=c` means `a=(b=c)`, which assigns the value of `c` to `b` and then assigns the value of `b` to `a`.

\[
\begin{align*}
\text{AssignmentExpression} & : \quad \text{ConditionalExpression} \quad \text{Assignment} \\
\text{Assignment} & : \quad \text{LeftHandSide} \ \text{AssignmentOperator} \ \text{Expression} \\
\text{LeftHandSide} & : \quad \text{ExpressionName} \quad \text{FieldAccess} \quad \text{ArrayAccess} \\
\text{AssignmentOperator} & : \quad \text{(one of)} \quad = \quad *= \quad /= \quad %= \quad += \quad -= \quad <<= \quad >>= \quad >>>= \quad &= \quad ^= \quad |=
\end{align*}
\]
The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs.

This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access (§15.11) or an array access (§15.10.3).

The type of the assignment expression is the type of the variable after capture conversion (§5.1.10).

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. The result of an assignment expression is not itself a variable.

A variable that is declared final cannot be assigned to (unless it is definitely unassigned (§16 (Definite Assignment))), because when an access of such a final variable is used as an expression, the result is a value, not a variable, and so it cannot be used as the first operand of an assignment operator.

15.26.1 Simple Assignment Operator =

If the type of the right-hand operand is not assignment compatible with the type of the variable (§5.2), then a compile-time error occurs.

Otherwise, at run time, the expression is evaluated in one of three ways.

If the left-hand operand expression is a field access expression $e.f$ (§15.11), possibly enclosed in one or more pairs of parentheses, then:

- First, the expression $e$ is evaluated. If evaluation of $e$ completes abruptly, the assignment expression completes abruptly for the same reason.

- Next, the right hand operand is evaluated. If evaluation of the right hand expression completes abruptly, the assignment expression completes abruptly for the same reason.

- Then, if the field denoted by $e.f$ is not static and the result of the evaluation of $e$ above is null, then a NullPointerException is thrown.

- Otherwise, the variable denoted by $e.f$ is assigned the value of the right hand operand as computed above.

If the left-hand operand is an array access expression (§15.10.3), possibly enclosed in one or more pairs of parentheses, then:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the
assignment expression completes abruptly for the same reason; the index
subexpression (of the left-hand operand array access expression) and the right-
hand operand are not evaluated and no assignment occurs.

- Otherwise, the index subexpression of the left-hand operand array access
expression is evaluated. If this evaluation completes abruptly, then the
assignment expression completes abruptly for the same reason and the right-hand
operand is not evaluated and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes
abruptly, then the assignment expression completes abruptly for the same reason
and no assignment occurs.

- Otherwise, if the value of the array reference subexpression is null, then no
assignment occurs and a NullPointerException is thrown.

- Otherwise, the value of the array reference subexpression indeed refers to an
array. If the value of the index subexpression is less than zero, or greater
than or equal to the length of the array, then no assignment occurs and an
ArrayIndexOutOfBoundsException is thrown.

- Otherwise, the value of the index subexpression is used to select a component of
the array referred to by the value of the array reference subexpression.

This component is a variable; call its type \( SC \). Also, let \( TC \) be the type of the left-
hand operand of the assignment operator as determined at compile time. Then
there are two possibilities:

- If \( TC \) is a primitive type, then \( SC \) is necessarily the same as \( TC \).

  The value of the right-hand operand is converted to the type of the selected
array component, is subjected to value set conversion (§5.1.13) to the
appropriate standard value set (not an extended-exponent value set), and the
result of the conversion is stored into the array component.

- If \( TC \) is a reference type, then \( SC \) may not be the same as \( TC \), but rather a type
that extends or implements \( TC \).

  Let \( RC \) be the class of the object referred to by the value of the right-hand
operand at run time.

  A Java compiler may be able to prove at compile time that the array component
will be of type \( TC \) exactly (for example, \( TC \) might be final). But if a Java
compiler cannot prove at compile time that the array component will be of
type \( TC \) exactly, then a check must be performed at run time to ensure that the
class \( RC \) is assignment compatible (§5.2) with the actual type \( SC \) of the array
component.
This check is similar to a narrowing cast (§5.5, §15.16), except that if the check fails, an `ArrayStoreException` is thrown rather than a `ClassCastException`.

If class `RC` is not assignable to type `SC`, then no assignment occurs and an `ArrayStoreException` is thrown.

Otherwise, the reference value of the right-hand operand is stored into the selected array component.

Otherwise, three steps are required:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

- Otherwise, the value of the right-hand operand is converted to the type of the left-hand variable, is subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

Example 15.26.1-1. Simple Assignment To An Array Component

```java
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow          extends RuntimeException { }
class RightHandSideThrow  extends RuntimeException { }

class IllustrateSimpleArrayAssignment {
    static Object[] objects = { new Object(), new Object() };
    static Thread[] threads = { new Thread(), new Thread() };

    static Object[] arrayThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() {
        throw new IndexThrow();
    }
    static Thread rightThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }
}
static void testFour(Object[] x, int j, Object y) {
    String sx = x == null ? "null" : name(x[0]) + "s";
    String sy = name(y);
    System.out.println();
    try {
        System.out.print(sx + "[throw]=throw => ");
        x[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[throw]=" + sy + " => ");
        x[indexThrow()] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=throw => ");
        x[j] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=" + sy + " => ");
        x[j] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

public static void main(String[] args) {
    try {
        System.out.print("throw[throw]=throw => ");
        arrayThrow()[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]=Thread => ");
        arrayThrow()[indexThrow()] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=throw => ");
        arrayThrow()[1] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=Thread => ");
        arrayThrow()[1] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

testFour(null, 1, new StringBuffer());
testFour(null, 9, new Thread());
testFour(objects, 1, new StringBuffer());
testFour(objects, 1, new Thread());
testFour(objects, 9, new StringBuffer());
This program produces the output:

testFour(objects, 9, new Thread());
testFour(threads, 1, new StringBuffer());
testFour(threads, 1, new Thread());
testFour(threads, 9, new StringBuffer());
testFour(threads, 9, new Thread());
}
throw[throw]=throw ⇒ ArrayReferenceThrow
toString[throw]=Thread ⇒ ArrayReferenceThrow
throw[1]=throw ⇒ ArrayReferenceThrow
throw[1]=Thread ⇒ ArrayReferenceThrow
null[throw]=throw ⇒ IndexThrow
null[throw]=StringBuffer ⇒ IndexThrow
null[1]=throw ⇒ RightHandSideThrow
null[1]=StringBuffer ⇒ NullPointerException
null[throw]=throw ⇒ IndexThrow
null[throw]=Thread ⇒ IndexThrow
null[9]=throw ⇒ RightHandSideThrow
null[9]=Thread ⇒ NullPointerException
Objects[throw]=throw ⇒ IndexThrow
Objects[throw]=StringBuffer ⇒ IndexThrow
Objects[1]=throw ⇒ RightHandSideThrow
Objects[1]=StringBuffer ⇒ Okay!
Objects[throw]=throw ⇒ IndexThrow
Objects[throw]=Thread ⇒ IndexThrow
Objects[1]=throw ⇒ RightHandSideThrow
Objects[1]=Thread ⇒ Okay!
Objects[throw]=throw ⇒ IndexThrow
Objects[throw]=StringBuffer ⇒ IndexThrow
Objects[9]=throw ⇒ RightHandSideThrow
Objects[9]=StringBuffer ⇒ ArrayIndexOutOfBoundsException
Objects[throw]=throw ⇒ IndexThrow
Objects[throw]=Thread ⇒ IndexThrow
Objects[9]=throw ⇒ RightHandSideThrow
Objects[9]=Thread ⇒ ArrayIndexOutOfBoundsException
Threads[throw]=throw ⇒ IndexThrow
Threads[throw]=StringBuffer ⇒ IndexThrow
Threads[1]=throw ⇒ RightHandSideThrow
Threads[1]=StringBuffer ⇒ ArrayStoreException
Threads[throw]=throw ⇒ IndexThrow
Threads[throw]=Thread ⇒ IndexThrow
Threads[1]=throw ⇒ RightHandSideThrow
Threads[1]=Thread ⇒ Okay!
Threads[throw]=throw ⇒ IndexThrow
Threads[throw]=StringBuffer ⇒ IndexThrow
Threads[9]=throw ⇒ RightHandSideThrow
Threads[9]=StringBuffer ⇒ ArrayIndexOutOfBoundsException
Threads[throw]=throw ⇒ IndexThrow
Threads[throw]=Thread ⇒ IndexThrow
Threads[9]=throw ⇒ RightHandSideThrow
Threads[9]=Thread ⇒ ArrayIndexOutOfBoundsException
The most interesting case of the lot is thirteenth from the end:

Threads[1]=StringBuffer => ArrayStoreException

which indicates that the attempt to store a reference to a StringBuffer into an array whose components are of type Thread throws an ArrayStoreException. The code is type-correct at compile time: the assignment has a left-hand side of type Object[] and a right-hand side of type Object. At run time, the first actual argument to method testFour is a reference to an instance of "array of Thread" and the third actual argument is a reference to an instance of class StringBuffer.

15.26.2 Compound Assignment Operators

A compound assignment expression of the form \( E_1 \ op= E_2 \) is equivalent to \( E_1 = (T) ((E_1) \ op (E_2)) \), where \( T \) is the type of \( E_1 \), except that \( E_1 \) is evaluated only once.

For example, the following code is correct:

```java
short x = 3;
x += 4.6;
```

and results in \( x \) having the value 7 because it is equivalent to:

```java
short x = 3;
x = (short)(x + 4.6);
```

At run time, the expression is evaluated in one of two ways.

If the left-hand operand expression is not an array access expression, then:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, the value of the left-hand operand is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the saved value of the left-hand variable and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the result of the binary operation is converted to the type of the left-hand variable, subjected to value set conversion (§5.1.13) to the appropriate
standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

If the left-hand operand expression is an array access expression (§15.10.3), then:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.

- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, if the value of the array reference subexpression is `null`, then no assignment occurs and a `NullPointerException` is thrown.

- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the `length` of the array, then no assignment occurs and an `ArrayIndexOutOfBoundsException` is thrown.

- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. The value of this component is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

  For a simple assignment operator, the evaluation of the right-hand operand occurs before the checks of the array reference subexpression and the index subexpression, but for a compound assignment operator, the evaluation of the right-hand operand occurs after these checks.

- Otherwise, consider the array component selected in the previous step, whose value was saved. This component is a variable; call its type $S$. Also, let $T$ be the type of the left-hand operand of the assignment operator as determined at compile time.

  - If $T$ is a primitive type, then $S$ is necessarily the same as $T$.

  The saved value of the array component and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator.
If this operation completes abruptly (the only possibility is an integer division by zero - see §15.17.2), then the assignment expression completes abruptly for the same reason and no assignment occurs.

Otherwise, the result of the binary operation is converted to the type of the selected array component, subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the array component.

- If \( T \) is a reference type, then it must be \texttt{String}. Because class \texttt{String} is a \texttt{final} class, \( S \) must also be \texttt{String}.

Therefore the run-time check that is sometimes required for the simple assignment operator is never required for a compound assignment operator.

The saved value of the array component and the value of the right-hand operand are used to perform the binary operation (string concatenation) indicated by the compound assignment operator (which is necessarily \(+\)). If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

Otherwise, the \texttt{String} result of the binary operation is stored into the array component.

**Example 15.26.2-1. Compound Assignment To An Array Component**

```java
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow     extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }

class IllustrateCompoundArrayAssignment {
    static String[] strings = { "Simon", "Garfunkel" };  
    static double[] doubles = { Math.E, Math.PI };  

    static String[] stringsThrow() {
        throw new ArrayReferenceThrow();
    }
    static double[] doublesThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() {
        throw new IndexThrow();
    }
    static String stringThrow() {
        throw new RightHandSideThrow();
    }
    static double doubleThrow() {
        throw new RightHandSideThrow();
    }
}
```
static String name(Object q) {
    String sq = q.getClass().getName();
    int k = sq.lastIndexOf('.');
    return (k < 0) ? sq : sq.substring(k + 1);
}

static void testEight(String[] x, double[] z, int j) {
    String sx = (x == null) ? "null" : "Strings";
    String sz = (z == null) ? "null" : "doubles";
    System.out.println();
    try {
        System.out.print(sx + "[\]" += throw => ");
        x[indexThrow()] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[\]" += throw => ");
        z[indexThrow()] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[\]" += "heh" => ");
        x[indexThrow()] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[\]" += 12345 => ");
        z[indexThrow()] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[\"]" += throw => ");
        x[j] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[\"]" += throw => ");
        z[j] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[\"]" += "heh" => ");
        x[j] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[\"]" += 12345 => ");
        z[j] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

public static void main(String[] args) {
This program produces the output:
The most interesting cases of the lot are eleventh and twelfth from the end:

Strings[9] += throw => ArrayIndexOutOfBoundsException
Strings[9] += "heh" => ArrayIndexOutOfBoundsException
Strings[9] += 12345 => ArrayIndexOutOfBoundsException

They are the cases where a right-hand side that throws an exception actually gets to throw the exception; moreover, they are the only such cases in the lot. This demonstrates that
the evaluation of the right-hand operand indeed occurs after the checks for a null array reference value and an out-of-bounds index value.


```java
class Test {
    public static void main(String[] args) {
        int k = 1;
        int[] a = { 1 };
        k += (k = 4) * (k + 2);
        a[0] += (a[0] = 4) * (a[0] + 2);
        System.out.println("k==" + k + " and a[0]==" + a[0]);
    }
}
```

This program produces the output:

```
k==25 and a[0]==25
```

The value 1 of k is saved by the compound assignment operator `+=` before its right-hand operand `(k = 4) * (k + 2)` is evaluated. Evaluation of this right-hand operand then assigns 4 to k, calculates the value 6 for `k + 2`, and then multiplies 4 by 6 to get 24. This is added to the saved value 1 to get 25, which is then stored into k by the `+=` operator. An identical analysis applies to the case that uses `a[0]`.

In short, the statements:

```java
k += (k = 4) * (k + 2);
```

```java
a[0] += (a[0] = 4) * (a[0] + 2);
```

behave in exactly the same manner as the statements:

```java
k = k + (k = 4) * (k + 2);
```

```java
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```

15.27 Lambda Expressions

A lambda expression is like a method: it provides a list of formal parameters and a body - an expression or block - expressed in terms of those parameters.

```
LambdaExpression:
   LambdaParameters -> LambdaBody
```

Lambda expressions are always poly expressions (§15.2).
It is a compile-time error if a lambda expression occurs in a program in someplace other than an assignment context (§5.2), an invocation context (§5.3), or a casting context (§5.5).

Evaluation of a lambda expression produces an instance of a functional interface (§9.8). Lambda expression evaluation does not cause the execution of the expression’s body; instead, this may occur at a later time when an appropriate method of the functional interface is invoked.

Here are some examples of lambda expressions:


This syntax has the advantage of minimizing bracket noise around simple lambda expressions, which is especially beneficial when a lambda expression is an argument to a method, or when the body is another lambda expression. It also clearly distinguishes between its expression and statement forms, which avoids ambiguities or over-reliance on ‘;’ tokens. When some extra bracketing is needed to visually distinguish either the full lambda expression or its body expression, parentheses are naturally supported (just as in other cases in which operator precedence is unclear).
The syntax has some parsing challenges. The Java programming language has always required arbitrary lookahead to distinguish between types and expressions after a `(` token: what follows may be a cast or a parenthesized expression. This was made worse when generics reused the binary operators `<` and `>` in types. Lambda expressions introduce a new possibility: the tokens following `(` may describe a type, an expression, or a lambda parameter list. Some tokens immediately indicate a parameter list (annotations, `final`); in other cases there are certain patterns that must be interpreted as parameter lists (two names in a row, a ``, `not nested inside of `<` and `>`); and sometimes, the decision cannot be made until a `->` is encountered after a `)`. The simplest way to think of how this might be efficiently parsed is with a state machine: each state represents a subset of possible interpretations (type, expression, or parameters), and when the machine transitions to a state in which the set is a singleton, the parser knows which case it is. This does not map very elegantly to a fixed-lookahead grammar, however.

There is no special nullary form: a lambda expression with zero arguments is expressed as `()` -> ... . The obvious special-case syntax, `-> ...`, does not work because it introduces an ambiguity between argument lists and casts: `(x) -> ...`.

Lambda expressions cannot declare type parameters. While it would make sense semantically to do so, the natural syntax (preceding the parameter list with a type parameter list) introduces messy ambiguities. For example, consider:

```java
foo( (x) < y , z > (w) -> v )
```

This could be an invocation of `foo` with one argument (a generic lambda cast to type `x`), or it could be an invocation of `foo` with two arguments, both the results of comparisons, the second comparing `z` with a lambda expression. (Strictly speaking, a lambda expression is meaningless as an operand to the relational operator `>`, but that is a tenuous assumption on which to build the grammar.)

There is a precedent for ambiguity resolution involving casts, which essentially prohibits the use of `–` and `+` following a non-primitive cast (§15.15), but to extend that approach to generic lambdas would involve invasive changes to the grammar.

### 15.27.1 Lambda Parameters

The formal parameters of a lambda expression, if any, are specified by either a parenthesized list of comma-separated parameter specifiers or a parenthesized list of comma-separated identifiers. In a list of parameter specifiers, each parameter specifier consists of optional modifiers, then a type (or `var`), then an identifier that specifies the name of the parameter. In a list of identifiers, each identifier specifies the name of the parameter.

If a lambda expression has no formal parameters, then an empty pair of parentheses appears before the `->` and the lambda body.
If a lambda expression has exactly one formal parameter, and the parameter is specified by an identifier instead of a parameter specifier, then the parentheses around the identifier may be elided.

LambdaParameters:
   ( [LambdaParameterList] )
   Identifier

LambdaParameterList:
   LambdaParameter , LambdaParameter
   Identifier , Identifier

LambdaParameter:
   {VariableModifier} LambdaParameterType VariableDeclaratorId
   VariableArityParameter

LambdaParameterType:
   UnannType
   var

The following productions from §8.4.1, §8.3, and §4.3 are shown here for convenience:

VariableArityParameter:
   {VariableModifier} UnannType {Annotation} . . . Identifier

VariableModifier:
   Annotation
   final

VariableDeclaratorId:
   Identifier [Dims]

Dims:
   {Annotation} [ ] {{Annotation} [ ]}

A formal parameter of a lambda expression may be declared final, or annotated, only if specified by a parameter specifier. If a formal parameter is specified by an identifier instead, then the formal parameter is not final and has no annotations.

A formal parameter of a lambda expression may be a variable arity parameter, indicated by an ellipsis following the type in a parameter specifier. At most one variable arity parameter is permitted for a lambda expression. It is a compile-time error if a variable arity parameter appears anywhere in the list of parameter specifiers except the last position.
Each formal parameter of a lambda expression has either an inferred type or a declared type:

- If a formal parameter is specified either by a parameter specifier that uses `var`, or by an identifier instead of a parameter specifier, then the formal parameter has an inferred type. The type is inferred from the functional interface type targeted by the lambda expression (§15.27.3).

- If a formal parameter is specified by a parameter specifier that does not use `var`, then the formal parameter has a declared type. The declared type is determined as follows:
  - If the formal parameter is not a variable arity parameter, then the declared type is denoted by `UnannType` if no bracket pairs appear in `UnannType` and `VariableDeclaratorId`, and specified by §10.2 otherwise.
  - If the formal parameter is a variable arity parameter, then the declared type is an array type specified by §10.2.

No distinction is made between the following lambda parameter lists:

```
(int... x) -> BODY
(int[] x) -> BODY
```

Either can be used, whether the functional interface's abstract method is fixed arity or variable arity. (This is consistent with the rules for method overriding.) Since lambda expressions are never directly invoked, using `int...` for the formal parameter where the functional interface uses `int[]` can have no impact on the surrounding program. In a lambda body, a variable arity parameter is treated just like an array-typed parameter.

A lambda expression where all the formal parameters have declared types is said to be explicitly typed. A lambda expression where all the formal parameters have inferred types is said to be implicitly typed. A lambda expression with no formal parameters is explicitly typed.

If a lambda expression is implicitly typed, then its lambda body is interpreted according to the context in which it appears. Specifically, the types of expressions in the body, and the checked exceptions thrown by the body, and the type correctness of code in the body all depend on the types inferred for the formal parameters. This implies that inference of formal parameter types must occur "before" attempting to type-check the lambda body.

It is a compile-time error if a lambda expression declares a formal parameter with a declared type and a formal parameter with an inferred type.

This rule prevents a mix of inferred and declared types in the formal parameters, such as `(x, int y) -> BODY` or `(var x, int y) -> BODY`. Note that if all the formal parameters
Lambda Expressions

have inferred types, the grammar prevents a mix of identifiers and var parameter specifiers, such as \((x, \text{var } y) \rightarrow \text{BODY}\) or \((\text{var } x, y) \rightarrow \text{BODY}\).

The rules for annotation modifiers on a formal parameter declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if final appears more than once as a modifier for a formal parameter declaration.

It is a compile-time error if the LambdaParameterType of a formal parameter is var and the VariableDeclaratorId of the same formal parameter has one or more bracket pairs.

The scope and shadowing of a formal parameter declaration is specified in §6.3 and §6.4.

It is a compile-time error for a lambda expression to declare two formal parameters with the same name. (That is, their declarations mention the same Identifier.)

In Java SE 8, the use of _ as the name of a lambda parameter was forbidden, and its use discouraged as the name for other kinds of variable (§4.12.3). As of Java SE 9, _ is a keyword (§3.9) so it cannot be used as a variable name in any context.

It is a compile-time error if a formal parameter that is declared final is assigned to within the body of the lambda expression.

When the lambda expression is invoked (via a method invocation expression (§15.12)), the values of the actual argument expressions initialize newly created parameter variables, each of the declared or inferred type, before execution of the lambda body. The Identifier that appears in the LambdaParameter or directly in the LambdaParameterList or LambdaParameters may be used as a simple name in the lambda body to refer to the formal parameter.

A lambda expression's formal parameter of type float always contains an element of the float value set (§4.2.3); similarly, a lambda expression's formal parameter of type double always contains an element of the double value set. It is not permitted for a lambda expression's formal parameter of type float to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a lambda expression's formal parameter of type double to contain an element of the double-extended-exponent value set that is not also an element of the double value set.
15.27.2 Lambda Body

A lambda body is either a single expression or a block (§14.2). Like a method body, a lambda body describes code that will be executed whenever an invocation occurs.

\[ \textit{LambdaBody:} \]
\[ \text{Expression} \]
\[ \text{Block} \]

Unlike code appearing in anonymous class declarations, the meaning of names and the \texttt{this} and \texttt{super} keywords appearing in a lambda body, along with the accessibility of referenced declarations, are the same as in the surrounding context (except that lambda parameters introduce new names).

The transparency of \texttt{this} (both explicit and implicit) in the body of a lambda expression - that is, treating it the same as in the surrounding context - allows more flexibility for implementations, and prevents the meaning of unqualified names in the body from being dependent on overload resolution.

Practically speaking, it is unusual for a lambda expression to need to talk about itself (either to call itself recursively or to invoke its other methods), while it is more common to want to use names to refer to things in the enclosing class that would otherwise be shadowed (\texttt{this}, \texttt{toString()}). If it is necessary for a lambda expression to refer to itself (as if via \texttt{this}), a method reference or an anonymous inner class should be used instead.

A block lambda body is \textit{void-compatible} if every return statement in the block has the form \texttt{return ;}.

A block lambda body is \textit{value-compatible} if it cannot complete normally (§14.21) and every return statement in the block has the form \texttt{return Expression ;}.

It is a compile-time error if a block lambda body is neither void-compatible nor value-compatible.

In a value-compatible block lambda body, the \textit{result expressions} are any expressions that may produce an invocation's value. Specifically, for each statement of the form \texttt{return Expression ;} contained by the body, the \texttt{Expression} is a result expression.

The following lambda bodies are void-compatible:

\[
() \rightarrow ()
\]
\[
() \rightarrow () \rightarrow \text{System.out.println("done"); }
\]

These are value-compatible:
Lambda Expressions

15.27

(\text{expression}) \rightarrow \{ \text{return \text{"done\"}; } \}

(\text{expression}) \rightarrow \{ \text{if (...} \text{return } 1; \text{else return } 0; \}

These are both:

(\text{expression}) \rightarrow \{ \text{throw new \text{RuntimeException\(); } \}

(\text{expression}) \rightarrow \{ \text{while (true); } \}

This is neither:

(\text{expression}) \rightarrow \{ \text{if (...} \text{return \text{"done\"}; System.out.println("done"); } \}

The handling of void/value-compatible and the meaning of names in the body jointly serve to minimize the dependency on a particular target type in the given context, which is useful both for implementations and for programmer comprehension. While expressions can be assigned different types during overload resolution depending on the target type, the meaning of unqualified names and the basic structure of the lambda body do not change.

Note that the void/value-compatible definition is not a strictly structural property: "can complete normally" depends on the values of constant expressions, and these may include names that reference constant variables.

Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared \text{final} or be effectively final (§4.12.4), or a compile-time error occurs where the use is attempted.

Any local variable used but not declared in a lambda body must be definitely assigned (§16 (\text{Definite Assignment})) before the lambda body, or a compile-time error occurs.

Similar rules on variable use apply in the body of an inner class (§8.1.3). The restriction to effectively final variables prohibits access to dynamically-changing local variables, whose capture would likely introduce concurrency problems. Compared to the \text{final} restriction, it reduces the clerical burden on programmers.

The restriction to effectively final variables includes standard loop variables, but not enhanced-\text{for} loop variables, which are treated as distinct for each iteration of the loop (§14.14.2).

The following lambda bodies demonstrate use of effectively final variables.

\begin{verbatim}
void m1(int x) {
    int y = 1;
    foo(\text{\rightarrow } x+y);
    // Legal: x and y are both effectively final.
}

void m2(int x) {
    int y;
\end{verbatim}
y = 1;
foo(() -> x+y);
// Legal: x and y are both effectively final.
}

void m3(int x) {
    int y;
    if (...) y = 1;
    foo(() -> x+y);
    // Illegal: y is effectively final, but not definitely assigned.
}

void m4(int x) {
    int y;
    if (...) y = 1; else y = 2;
    foo(() -> x+y);
    // Legal: x and y are both effectively final.
}

void m5(int x) {
    int y;
    if (...) y = 1;
    y = 2;
    foo(() -> x+y);
    // Illegal: y is not effectively final.
}

void m6(int x) {
    foo(() -> x+1);
    x++;
    // Illegal: x is not effectively final.
}

void m7(int x) {
    foo(() -> x=1);
    // Illegal: x is not effectively final.
}

void m8() {
    int y;
    foo(() -> y=1);
    // Illegal: y is not definitely assigned before the lambda.
}

void m9(String[] arr) {
    for (String s : arr) {
        foo(() -> s);
        // Legal: s is effectively final
        // (it is a new variable on each iteration)
    }
}

void m10(String[] arr) {

for (int i = 0; i < arr.length; i++) {
    foo(() -> arr[i]);
    // Illegal: i is not effectively final
    // (it is not final, and is incremented)
} }

15.27.3 Type of a Lambda Expression

A lambda expression is compatible in an assignment context, invocation context, or casting context with a target type \( \tau \) if \( \tau \) is a functional interface type (§9.8) and the expression is congruent with the function type of the ground target type derived from \( \tau \).

The ground target type is derived from \( \tau \) as follows:

- If \( \tau \) is a wildcard-parameterized functional interface type and the lambda expression is explicitly typed, then the ground target type is inferred as described in §18.5.3.
- If \( \tau \) is a wildcard-parameterized functional interface type and the lambda expression is implicitly typed, then the ground target type is the non-wildcard parameterization (§9.9) of \( \tau \).
- Otherwise, the ground target type is \( \tau \).

A lambda expression is congruent with a function type if all of the following are true:

- The function type has no type parameters.
- The number of lambda parameters is the same as the number of parameter types of the function type.
- If the lambda expression is explicitly typed, its formal parameter types are the same as the parameter types of the function type.
- If the lambda parameters are assumed to have the same types as the function type's parameter types, then:
  - If the function type's result is \texttt{void}, the lambda body is either a statement expression (§14.8) or a \texttt{void}-compatible block.
  - If the function type's result is a (non-\texttt{void}) type \( R \), then either (i) the lambda body is an expression that is compatible with \( R \) in an assignment context, or (ii) the lambda body is a value-compatible block, and each result expression (§15.27.2) is compatible with \( R \) in an assignment context.
If a lambda expression is compatible with a target type $T$, then the type of the expression, $U$, is the ground target type derived from $T$.

It is a compile-time error if any class or interface mentioned by either $U$ or the function type of $U$ is not accessible (§6.6) from the class or interface in which the lambda expression appears.

For each non-static member method $m$ of $U$, if the function type of $U$ has a subsignature of the signature of $m$, then a notional method whose method type is the function type of $U$ is deemed to override $m$, and any compile-time error or unchecked warning specified in §8.4.8.3 may occur.

A checked exception that can be thrown in the body of the lambda expression may cause a compile-time error, as specified in §11.2.3.

The parameter types of explicitly typed lambdas are required to exactly match those of the function type. While it would be possible to be more flexible - allow boxing or contravariance, for example - this kind of generality seems unnecessary, and is inconsistent with the way overriding works in class declarations. A programmer ought to know exactly what function type is being targeted when writing a lambda expression, so he should thus know exactly what signature must be overridden. (In contrast, this is not the case for method references, and so more flexibility is allowed when they are used.) In addition, more flexibility with parameter types would add to the complexity of type inference and overload resolution.

Note that while boxing is not allowed in a strict invocation context, boxing of lambda result expressions is always allowed - that is, the result expression appears in an assignment context, regardless of the context enclosing the lambda expression. However, if an explicitly typed lambda expression is an argument to an overloaded method, a method signature that avoids boxing or unboxing the lambda result is preferred by the most specific check (§15.12.2.5).

If the body of a lambda is a statement expression (that is, an expression that would be allowed to stand alone as a statement), it is compatible with a void-producing function type; any result is simply discarded. So, for example, both of the following are legal:

```java
// Predicate has a boolean result
java.util.function.Predicate<String> p = s -> list.add(s);
// Consumer has a void result
java.util.function.Consumer<String> c = s -> list.add(s);
```

Generally speaking, a lambda of the form $(\cdot) \rightarrow expr$, where $expr$ is a statement expression, is interpreted as either $(\cdot) \rightarrow \{ \text{return } expr; \}$ or $(\cdot) \rightarrow \{ expr; \}$, depending on the target type.
15.27.4 Run-Time Evaluation of Lambda Expressions

At run time, evaluation of a lambda expression is similar to evaluation of a class instance creation expression, insofar as normal completion produces a reference to an object. Evaluation of a lambda expression is distinct from execution of the lambda body.

Either a new instance of a class with the properties below is allocated and initialized, or an existing instance of a class with the properties below is referenced. If a new instance is to be created, but there is insufficient space to allocate the object, evaluation of the lambda expression completes abruptly by throwing an OutOfMemoryError.

This implies that the identity of the result of evaluating a lambda expression (or, of serializing and deserializing a lambda expression) is unpredictable, and therefore identity-sensitive operations (such as reference equality (§15.21.3), object locking (§14.19), and the System.identityHashCode method) may produce different results in different implementations of the Java programming language, or even upon different lambda expression evaluations in the same implementation.

The value of a lambda expression is a reference to an instance of a class with the following properties:

- The class implements the targeted functional interface type and, if the target type is an intersection type, every other interface type mentioned in the intersection.

- Where the lambda expression has type $U$, for each non-static member method $m$ of $U$:

  If the function type of $U$ has a subsignature of the signature of $m$, then the class declares a method that overrides $m$. The method's body has the effect of evaluating the lambda body, if it is an expression, or of executing the lambda body, if it is a block; if a result is expected, it is returned from the method.

  If the erasure of the type of a method being overridden differs in its signature from the erasure of the function type of $U$, then before evaluating or executing the lambda body, the method's body checks that each argument value is an instance of a subclass or subinterface of the erasure of the corresponding parameter type in the function type of $U$; if not, a ClassCastException is thrown.

- The class overrides no other methods of the targeted functional interface type or other interface types mentioned above, although it may override methods of the Object class.

These rules are meant to offer flexibility to implementations of the Java programming language, in that:
• A new object need not be allocated on every evaluation.

• Objects produced by different lambda expressions need not belong to different classes (if the bodies are identical, for example).

• Every object produced by evaluation need not belong to the same class (captured local variables might be inlined, for example).

• If an "existing instance" is available, it need not have been created at a previous lambda evaluation (it might have been allocated during the enclosing class's initialization, for example).

If the targeted functional interface type is a subtype of `java.io.Serializable`, the resulting object will automatically be an instance of a serializable class. Making an object derived from a lambda expression serializable can have extra run time overhead and security implications, so lambda-derived objects are not required to be serializable "by default".

### 15.28 Constant Expressions

A constant expression is an expression denoting a value of primitive type or a `String` that does not complete abruptly and is composed using only the following:

- Literals of primitive type and literals of type `String` (§3.10.1, §3.10.2, §3.10.3, §3.10.4, §3.10.5)
- Casts to primitive types and casts to type `String` (§15.16)
- The unary operators `+`, `-`, `~`, and `!` (but not `++` or `--`) (§15.15.3, §15.15.4, §15.15.5, §15.15.6)
- The multiplicative operators `*`, `/`, and `%` (§15.17)
- The additive operators `+` and `-` (§15.18)
- The shift operators `<<`, `>>`, and `>>>` (§15.19)
- The relational operators `<`, `<=`, `>`, and `>=` (but not `instanceof`) (§15.20)
- The equality operators `==` and `!=` (§15.21)
- The bitwise and logical operators `&`, `^`, and `|` (§15.22)
- The conditional-and operator `&&` and the conditional-or operator `||` (§15.23, §15.24)
- The ternary conditional operator `?:` (§15.25)
• Parenthesized expressions (§15.8.5) whose contained expression is a constant expression.

• Simple names (§6.5.6.1) that refer to constant variables (§4.12.4).

• Qualified names (§6.5.6.2) of the form `TypeName . Identifier` that refer to constant variables (§4.12.4).

Constant expressions of type `String` are always "interned" so as to share unique instances, using the method `String.intern`.

A constant expression is always treated as FP-strict (§15.4), even if it occurs in a context where a non-constant expression would not be considered to be FP-strict.

Constant expressions are used as case labels in switch statements (§14.11) and have a special significance in assignment contexts (§5.2) and the initialization of a class or interface (§12.4.2). They may also govern the ability of a `while`, `do`, or `for` statement to complete normally (§14.21), and the type of a conditional operator `? :` with numeric operands.

**Example 15.28-1. Constant Expressions**

```java
true
(short)(1*2*3*4*5*6)
Integer.MAX_VALUE / 2
2.0 * Math.PI
"The integer " + Long.MAX_VALUE + " is mighty big."
```
EACH local variable (§14.4) and every blank final field (§4.12.4, §8.3.1.2) must have a \textit{definitely assigned} value when any access of its value occurs.

An access to its value consists of the simple name of the variable (or, for a field, the simple name of the field qualified by \texttt{this}) occurring anywhere in an expression except as the left-hand operand of the simple assignment operator = (§15.26.1).

For every access of a local variable or blank final field \( x \), \( x \) must be definitely assigned before the access, or a compile-time error occurs.

Similarly, every blank final variable must be assigned at most once; it must be \textit{definitely unassigned} when an assignment to it occurs.

Such an assignment is defined to occur if and only if either the simple name of the variable (or, for a field, its simple name qualified by \texttt{this}) occurs on the left hand side of an assignment operator.

For every assignment to a blank final variable, the variable must be definitely unassigned before the assignment, or a compile-time error occurs.

The remainder of this chapter is devoted to a precise explanation of the words "definitely assigned before" and "definitely unassigned before".

The idea behind definite assignment is that an assignment to the local variable or blank final field must occur on every possible execution path to the access. Similarly, the idea behind definite unassignment is that no other assignment to the blank final variable is permitted to occur on any possible execution path to an assignment.

The analysis takes into account the structure of statements and expressions; it also provides a special treatment of the expression operators \(!, \&\&, |\|, \text{ and } ?, :\), and of boolean-valued constant expressions.
Except for the special treatment of the conditional boolean operators $\&\&$, $|\|$, and $?$ : and of boolean-valued constant expressions, the values of expressions are not taken into account in the flow analysis.

**Example 16-1. Definite Assignment Considers Structure of Statements and Expressions**

A Java compiler recognizes that $k$ is definitely assigned before its access (as an argument of a method invocation) in the code:

```java
int k;
if (v > 0 && (k = System.in.read()) >= 0)
   System.out.println(k);
```

because the access occurs only if the value of the expression:

$$v > 0 \&\& (k = \text{System.in.read()}) \geq 0$$

is true, and the value can be true only if the assignment to $k$ is executed (more properly, evaluated).

Similarly, a Java compiler will recognize that in the code:

```java
int k;
while (true) {
   k = n;
   if (k >= 5) break;
   n = 6;
}
System.out.println(k);
```

the variable $k$ is definitely assigned by the while statement because the condition expression true never has the value false, so only the break statement can cause the while statement to complete normally, and $k$ is definitely assigned before the break statement.

On the other hand, the code:

```java
int k;
while (n < 4) {
   k = n;
   if (k >= 5) break;
   n = 6;
}
System.out.println(k);  /* k is not "definitely assigned"
   before this statement */
```
must be rejected by a Java compiler, because in this case the while statement is not guaranteed to execute its body as far as the rules of definite assignment are concerned.

Example 16-2. Definite Assignment Does Not Consider Values of Expressions

A Java compiler must produce a compile-time error for the code:

```java
{  
  int k;
  int n = 5;
  if (n > 2)
    k = 3;
  System.out.println(k);  /* k is not "definitely assigned"
      before this statement */
}
```

even though the value of \( n \) is known at compile time, and in principle it can be known at compile time that the assignment to \( k \) will always be executed (more properly, evaluated). A Java compiler must operate according to the rules laid out in this section. The rules recognize only constant expressions; in this example, the expression \( n > 2 \) is not a constant expression as defined in §15.28.

As another example, a Java compiler will accept the code:

```java
void flow(boolean flag) {
  int k;
  if (flag)
    k = 3;
  else
    k = 4;
  System.out.println(k);
}
```

as far as definite assignment of \( k \) is concerned, because the rules outlined in this section allow it to tell that \( k \) is assigned no matter whether the flag is true or false. But the rules do not accept the variation:

```java
void flow(boolean flag) {
  int k;
  if (flag)
    k = 3;
  if (!flag)
    k = 4;
  System.out.println(k);  /* k is not "definitely assigned"
      before this statement */
}
```

and so compiling this program must cause a compile-time error to occur.
Example 16-3. Definite Unassignment

A Java compiler will accept the code:

```java
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    else {
        k = 4;
        System.out.println(k);
    }
}
```

as far as definite unassignment of `k` is concerned, because the rules outlined in this section allow it to tell that `k` is assigned at most once (indeed, exactly once) no matter whether the `flag` is true or false. But the rules do not accept the variation:

```java
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    if (!flag) {
        k = 4;
        System.out.println(k);  /* k is not "definitely unassigned"
before this statement */
    }
}
```

and so compiling this program must cause a compile-time error to occur.

In order to precisely specify all the cases of definite assignment, the rules in this section define several technical terms:

- whether a variable is *definitely assigned before* a statement or expression;
- whether a variable is *definitely unassigned before* a statement or expression;
- whether a variable is *definitely assigned after* a statement or expression; and
- whether a variable is *definitely unassigned after* a statement or expression.

For boolean-valued expressions, the last two are refined into four cases:

- whether a variable is *definitely assigned after* the expression *when true*;
- whether a variable is *definitely unassigned after* the expression *when true*;
DEFINITE ASSIGNMENT

• whether a variable is *definitely assigned after* the expression *when false*; and
• whether a variable is *definitely unassigned after* the expression *when false*.

Here, *when true* and *when false* refer to the value of the expression.

For example, the local variable \( k \) is definitely assigned a value after evaluation of the expression:

\[
a \&\& ((k=m) > 5)
\]

when the expression is *true* but not when the expression is *false* (because if \( a \) is false, then the assignment to \( k \) is not necessarily executed (more properly, evaluated)).

The phrase "\( V \) is definitely assigned after \( X \)" (where \( V \) is a local variable and \( X \) is a statement or expression) means "\( V \) is definitely assigned after \( X \) if \( X \) completes normally". If \( X \) completes abruptly, the assignment need not have occurred, and the rules stated here take this into account.

A peculiar consequence of this definition is that "\( V \) is definitely assigned after \( \text{break} \);" is always true! Because a \( \text{break} \) statement never completes normally, it is vacuously true that \( V \) has been assigned a value if the \( \text{break} \) statement completes normally.

The statement "\( V \) is definitely unassigned after \( X \)" (where \( V \) is a variable and \( X \) is a statement or expression) means "\( V \) is definitely unassigned after \( X \) if \( X \) completes normally".

An even more peculiar consequence of this definition is that "\( V \) is definitely unassigned after \( \text{break} \);" is always true! Because a \( \text{break} \) statement never completes normally, it is vacuously true that \( V \) has not been assigned a value if the \( \text{break} \) statement completes normally. (For that matter, it is also vacuously true that the moon is made of green cheese if the \( \text{break} \) statement completes normally.)

In all, there are four possibilities for a variable \( V \) after a statement or expression has been executed:

• \( V \) is definitely assigned and is not definitely unassigned.
  (The flow analysis rules prove that an assignment to \( V \) has occurred.)
• \( V \) is definitely unassigned and is not definitely assigned.
  (The flow analysis rules prove that an assignment to \( V \) has not occurred.)
• \( V \) is not definitely assigned and is not definitely unassigned.
  (The rules cannot prove whether or not an assignment to \( V \) has occurred.)
• \( V \) is definitely assigned and is definitely unassigned.
DEFINITE ASSIGNMENT

(It is impossible for the statement or expression to complete normally.)

To shorten the rules, the customary abbreviation "iff" is used to mean "if and only if". We also use an abbreviation convention: if a rule contains one or more occurrences of "[un]assigned" then it stands for two rules, one with every occurrence of "[un]assigned" replaced by "definitely assigned" and one with every occurrence of "[un]assigned" replaced by "definitely unassigned".

For example:

• $v$ is [un]assigned after an empty statement iff it is [un]assigned before the empty statement.

should be understood to stand for two rules:

• $v$ is definitely assigned after an empty statement iff it is definitely assigned before the empty statement.

• $v$ is definitely unassigned after an empty statement iff it is definitely unassigned before the empty statement.

Throughout the rest of this chapter, we will, unless explicitly stated otherwise, write $v$ to represent a local variable or blank final field which is in scope (§6.3). Likewise, we will use $a$, $b$, $c$, and $e$ to represent expressions, and $s$ and $t$ to represent statements. We will use the phrase "$a$ is $v$" to mean that $a$ is either the simple name of the variable $v$, or $v$'s simple name qualified by $this$ (ignoring parentheses). We will use the phrase "$a$ is not $v$" to mean the negation of "$a$ is $v$".

The definite unassignment analysis of loop statements raises a special problem. Consider the statement while ($e$) $s$. In order to determine whether $v$ is definitely unassigned within some subexpression of $e$, we need to determine whether $v$ is definitely unassigned before $e$. One might argue, by analogy with the rule for definite assignment (§16.2.10), that $v$ is definitely unassigned before $e$ iff it is definitely unassigned before the while statement. However, such a rule is inadequate for our purposes. If $e$ evaluates to true, the statement $s$ will be executed. Later, if $v$ is assigned by $s$, then in the following iteration(s) $v$ will have already been assigned when $e$ is evaluated. Under the rule suggested above, it would be possible to assign $v$ multiple times, which is exactly what we have sought to avoid by introducing these rules.

A revised rule would be: "$v$ is definitely unassigned before $e$ iff it is definitely unassigned before the while statement and definitely unassigned after $s$". However, when we formulate the rule for $s$, we find: "$v$ is definitely unassigned before $s$ iff it is definitely unassigned after $e$ when true". This leads to a circularity. In effect, $v$ is definitely unassigned before the loop condition $e$ only if it is unassigned after the loop as a whole!

We break this vicious circle using a hypothetical analysis of the loop condition and body. For example, if we assume that $v$ is definitely unassigned before $e$ (regardless of whether $v$
really is definitely unassigned before \( e \), and can then prove that \( V \) was definitely unassigned after \( e \) then we know that \( e \) does not assign \( V \). This is stated more formally as:

Assuming \( V \) is definitely unassigned before \( e \), \( V \) is definitely unassigned after \( e \).

Variations on the above analysis are used to define well founded definite unassignment rules for all loop statements in the Java programming language.

### 16.1 Definite Assignment and Expressions

#### 16.1.1 Boolean Constant Expressions

- \( V \) is [un]assigned after any constant expression (§15.28) whose value is \texttt{true} when false.
- \( V \) is [un]assigned after any constant expression whose value is \texttt{false} when true.
- \( V \) is [un]assigned after any constant expression whose value is \texttt{true} when true iff \( V \) is [un]assigned before the constant expression.
- \( V \) is [un]assigned after any constant expression whose value is \texttt{false} when false iff \( V \) is [un]assigned before the constant expression.
- \( V \) is [un]assigned after a boolean-valued constant expression \( e \) iff \( V \) is [un]assigned after \( e \) when true and \( V \) is [un]assigned after \( e \) when false.

This is equivalent to saying that \( V \) is [un]assigned after \( e \) iff \( V \) is [un]assigned before \( e \).

Because a constant expression whose value is \texttt{true} never has the value \texttt{false}, and a constant expression whose value is \texttt{false} never has the value \texttt{true}, the first two rules are vacuously satisfied. They are helpful in analyzing expressions involving the operators && (§16.1.2), || (§16.1.3), ! (§16.1.4), and ? : (§16.1.5).

#### 16.1.2 Conditional-And Operator &&

- \( V \) is [un]assigned after \( a \&\& b \) (§15.23) when true iff \( V \) is [un]assigned after \( b \) when true.
- \( V \) is [un]assigned after \( a \&\& b \) when false iff \( V \) is [un]assigned after \( a \) when false and \( V \) is [un]assigned after \( b \) when false.
- \( V \) is [un]assigned before \( a \) iff \( V \) is [un]assigned before \( a \&\& b \).
- \( V \) is [un]assigned before \( b \) iff \( V \) is [un]assigned after \( a \) when true.
16.1.3 Conditional-Or Operator \texttt{||}

- \(V\) is [un]assigned after \(a \texttt{||} b\) when \(V\) is [un]assigned after \(a\) when true and \(V\) is [un]assigned after \(b\) when false.

- \(V\) is [un]assigned after \(a \texttt{||} b\) when false iff \(V\) is [un]assigned after \(b\) when false.

- \(V\) is [un]assigned before \(a\) iff \(V\) is [un]assigned before \(a \texttt{||} b\).

- \(V\) is [un]assigned before \(b\) iff \(V\) is [un]assigned after \(a\) when false.

- \(V\) is [un]assigned after \(a \texttt{||} b\) iff \(V\) is [un]assigned after \(a \texttt{||} b\) when true and \(V\) is [un]assigned after \(a \texttt{||} b\) when false.

16.1.4 Logical Complement Operator \texttt{!}

- \(V\) is [un]assigned after \(!a\) (§15.15.6) when true iff \(V\) is [un]assigned after \(a\) when false.

- \(V\) is [un]assigned after \(!a\) when false iff \(V\) is [un]assigned after \(a\) when true.

- \(V\) is [un]assigned before \(a\) iff \(V\) is [un]assigned before \(!a\).

- \(V\) is [un]assigned after \(!a\) iff \(V\) is [un]assigned after \(!a\) when true and \(V\) is [un]assigned after \(!a\) when false.

This is equivalent to saying that \(V\) is [un]assigned after \(!a\) iff \(V\) is [un]assigned after \(a\).

16.1.5 Conditional Operator \texttt{? :}

Suppose that \(b\) and \(c\) are boolean-valued expressions.

- \(V\) is [un]assigned after \(a \texttt{? b : c}\) (§15.25) when true iff \(V\) is [un]assigned after \(b\) when true and \(V\) is [un]assigned after \(c\) when true.

- \(V\) is [un]assigned after \(a \texttt{? b : c}\) when false iff \(V\) is [un]assigned after \(b\) when false and \(V\) is [un]assigned after \(c\) when false.

- \(V\) is [un]assigned before \(a\) iff \(V\) is [un]assigned before \(a \texttt{? b : c}\).

- \(V\) is [un]assigned before \(b\) iff \(V\) is [un]assigned after \(a\) when true.

- \(V\) is [un]assigned before \(c\) iff \(V\) is [un]assigned after \(a\) when false.
• \( V \) is [un]assigned after \( a \ ? \ b : c \) iff \( V \) is [un]assigned after \( a \ ? \ b : c \) when true and \( V \) is [un]assigned after \( a \ ? \ b : c \) when false.

### 16.1.6 Conditional Operator \(? :\)

Suppose that \( b \) and \( c \) are expressions that are not boolean-valued.

• \( V \) is [un]assigned after \( a \ ? \ b : c \) (§15.25) iff \( V \) is [un]assigned after \( b \) and \( V \) is [un]assigned after \( c \).

• \( V \) is [un]assigned before \( a \) iff \( V \) is [un]assigned before \( a \ ? \ b : c \).

• \( V \) is [un]assigned before \( b \) iff \( V \) is [un]assigned after \( a \) when true.

• \( V \) is [un]assigned before \( c \) iff \( V \) is [un]assigned after \( a \) when false.

### 16.1.7 Other Expressions of Type boolean

Suppose that \( e \) is an expression of type boolean and is not a boolean constant expression, logical complement expression \( !a \), conditional-and expression \( a \ &\& \ b \), conditional-or expression \( a \ || \ b \), or conditional expression \( a \ ? \ b : c \).

• \( V \) is [un]assigned after \( e \) when true iff \( V \) is [un]assigned after \( e \).

• \( V \) is [un]assigned after \( e \) when false iff \( V \) is [un]assigned after \( e \).

### 16.1.8 Assignment Expressions

Consider an assignment expression \( a = b \), \( a += b \), \( a -= b \), \( a *= b \), \( a /= b \), \( a %= b \), \( a <<= b \), \( a >>= b \), \( a >>>= b \), \( a &= b \), \( a |= b \), \( a \ &= b \), or \( a ^= b \) (§15.26).

• \( V \) is definitely assigned after the assignment expression iff either:
  
  – \( a \) is \( V \), or

  – \( V \) is definitely assigned after \( b \).

• \( V \) is definitely unassigned after the assignment expression iff \( a \) is not \( V \) and \( V \) is definitely unassigned after \( b \).

• \( V \) is [un]assigned before \( a \) iff \( V \) is [un]assigned before the assignment expression.

• \( V \) is [un]assigned before \( b \) iff \( V \) is [un]assigned after \( a \).

Note that if \( a \) is \( V \) and \( V \) is not definitely assigned before a compound assignment such as \( a \ &= b \), then a compile-time error will necessarily occur. The first rule for definite assignment stated above includes the disjunct "\( a \) is \( V \)" even for compound assignment expressions, not just simple assignments, so that \( V \) will be considered to have been definitely assigned at
later points in the code. Including the disjunct "a is V" does not affect the binary decision as to whether a program is acceptable or will result in a compile-time error, but it affects how many different points in the code may be regarded as erroneous, and so in practice it can improve the quality of error reporting. A similar remark applies to the inclusion of the conjunct "a is not V" in the first rule for definite unassignment stated above.

16.1.9 Operators ++ and --

• V is definitely assigned after ++a (§15.15.1), --a (§15.15.2), a++ (§15.14.2), or a-- (§15.14.3) iff either a is V or V is definitely assigned after the operand expression.

• V is definitely unassigned after ++a, --a, a++, or a-- iff a is not V and V is definitely unassigned after the operand expression.

• V is [un]assigned before a iff V is [un]assigned before ++a, --a, a++, or a--.

16.1.10 Other Expressions

If an expression is not a boolean constant expression, and is not a preincrement expression ++a, predecrement expression --a, postincrement expression a++, postdecrement expression a--, logical complement expression !a, conditional-and expression a && b, conditional-or expression a || b, conditional expression a ? b : c, assignment expression, or lambda expression, then the following rules apply:

• If the expression has no subexpressions, V is [un]assigned after the expression iff V is [un]assigned before the expression.

This case applies to literals, names, this (both qualified and unqualified), unqualified class instance creation expressions with no arguments, array creation expressions with initializers that contain no expressions, superclass field access expressions, unqualified and type-qualified method invocation expressions with no arguments, superclass method invocation expressions with no arguments, and superclass and type-qualified method reference expressions.

• If the expression has subexpressions, V is [un]assigned after the expression iff V is [un]assigned after its rightmost immediate subexpression.

There is a piece of subtle reasoning behind the assertion that a variable V can be known to be definitely unassigned after a method invocation expression. Taken by itself, at face value and without qualification, such an assertion is not always true, because an invoked method can perform assignments. But it must be remembered that, for the purposes of the Java programming language, the concept of definite unassignment is applied only to blank final variables. If V is a blank final local variable, then only the method to which its declaration belongs can perform assignments to V. If V is a blank final field, then only a constructor or an initializer for the class containing the declaration for V can perform assignments to V; no method can perform assignments to V. Finally, explicit constructor invocations (§8.8.7.1) are handled specially (§16.9); although they are syntactically similar
to expression statements containing method invocations, they are not expression statements and therefore the rules of this section do not apply to explicit constructor invocations.

If an expression is a lambda expression, then the following rules apply:

- \( V \) is [un]assigned after the expression iff \( V \) is [un]assigned before the expression.
- \( V \) is definitely assigned before the expression or block that is the lambda body (§15.27.2) iff \( V \) is definitely assigned before the lambda expression.

No rule allows \( V \) to be definitely unassigned before a lambda body. This is by design: a variable that was definitely unassigned before the lambda body may end up being assigned to later on, so we cannot conclude that the variable will be unassigned when the body is executed.

For any immediate subexpression \( y \) of an expression \( x \), where \( x \) is not a lambda expression, \( V \) is [un]assigned before \( y \) iff one of the following is true:

- \( y \) is the leftmost immediate subexpression of \( x \) and \( V \) is [un]assigned before \( x \).
- \( y \) is the right-hand operand of a binary operator and \( V \) is [un]assigned after the left-hand operand.
- \( x \) is an array access, \( y \) is the subexpression within the brackets, and \( V \) is [un]assigned after the subexpression before the brackets.
- \( x \) is a primary method invocation expression, \( y \) is the first argument expression in the method invocation expression, and \( V \) is [un]assigned after the primary expression that computes the target object.
- \( x \) is a method invocation expression or a class instance creation expression; \( y \) is an argument expression, but not the first; and \( V \) is [un]assigned after the argument expression to the left of \( y \).
- \( x \) is a qualified class instance creation expression, \( y \) is the first argument expression in the class instance creation expression, and \( V \) is [un]assigned after the primary expression that computes the qualifying object.
- \( x \) is an array creation expression; \( y \) is a dimension expression, but not the first; and \( V \) is [un]assigned after the dimension expression to the left of \( y \).
- \( x \) is an array creation expression initialized via an array initializer; \( y \) is the array initializer in \( x \); and \( V \) is [un]assigned after the dimension expression to the left of \( y \).
16.2 Definite Assignment and Statements

16.2.1 Empty Statements

• $v$ is [un]assigned after an empty statement (§14.6) iff it is [un]assigned before the empty statement.

16.2.2 Blocks

• A blank final member field $v$ is definitely assigned (and moreover is not definitely unassigned) before the block (§14.2) that is the body of any method in the scope of $v$ and before the declaration of any class declared within the scope of $v$.

• A local variable $v$ is definitely unassigned (and moreover is not definitely assigned) before the block that is the body of the constructor, method, instance initializer or static initializer that declares $v$.

• Let $c$ be a class declared within the scope of $v$. Then $v$ is definitely assigned before the block that is the body of any constructor, method, instance initializer, or static initializer declared in $c$ iff $v$ is definitely assigned before the declaration of $c$.

Note that there are no rules that would allow us to conclude that $v$ is definitely unassigned before the block that is the body of any constructor, method, instance initializer, or static initializer declared in $c$. We can informally conclude that $v$ is not definitely unassigned before the block that is the body of any constructor, method, instance initializer, or static initializer declared in $c$, but there is no need for such a rule to be stated explicitly.

• $v$ is [un]assigned after an empty block iff $v$ is [un]assigned before the empty block.

• $v$ is [un]assigned after a non-empty block iff $v$ is [un]assigned after the last statement in the block.

• $v$ is [un]assigned before the first statement of the block iff $v$ is [un]assigned before the block.

• $v$ is [un]assigned before any other statement $s$ of the block iff $v$ is [un]assigned after the statement immediately preceding $s$ in the block.

We say that $v$ is definitely unassigned everywhere in a block $B$ iff:

• $v$ is definitely unassigned before $B$. 
• \( V \) is definitely assigned after \( e \) in every assignment expression \( V = e, V += e, V -= e, V *= e, V /= e, V %= e, V <<= e, V >>= e, V >>>= e, V &= e, V |= e, \) or \( V ^= e \) that occurs in \( B \).

• \( V \) is definitely assigned before every expression \( ++V, --V, V++ \), or \( V-- \) that occurs in \( B \).

These conditions are counterintuitive and require some explanation. Consider a simple assignment \( V = e \). If \( V \) is definitely assigned after \( e \), then either:

- The assignment occurs in dead code, and \( V \) is vacuously definitely assigned. In this case, the assignment will not actually take place, and we can assume that \( V \) is not being assigned by the assignment expression. Or:

- \( V \) was already assigned by an earlier expression prior to \( e \). In this case the current assignment will cause a compile-time error.

So, we can conclude that if the conditions are met by a program that causes no compile time error, then any assignments to \( V \) in \( B \) will not actually take place at run time.

### 16.2.3 Local Class Declaration Statements

• \( V \) is [un]assigned after a local class declaration statement (§14.3) iff \( V \) is [un]assigned before the local class declaration statement.

### 16.2.4 Local Variable Declaration Statements

• \( V \) is [un]assigned after a local variable declaration statement (§14.4) that contains no variable initializers iff \( V \) is [un]assigned before the local variable declaration statement.

• \( V \) is definitely assigned after a local variable declaration statement that contains at least one variable initializer iff either \( V \) is definitely assigned after the last variable initializer in the local variable declaration statement or the last variable initializer in the declaration is in the declarator that declares \( V \).

• \( V \) is definitely unassigned after a local variable declaration statement that contains at least one variable initializer iff \( V \) is definitely unassigned after the last variable initializer in the local variable declaration statement and the last variable initializer in the declaration is not in the declarator that declares \( V \).

• \( V \) is [un]assigned before the first variable initializer in a local variable declaration statement iff \( V \) is [un]assigned before the local variable declaration statement.

• \( V \) is definitely assigned before any variable initializer \( e \) other than the first one in the local variable declaration statement iff either \( V \) is definitely assigned after
the variable initializer to the left of \( e \) or the initializer expression to the left of \( e \)
is in the declarator that declares \( v \).

- \( v \) is definitely unassigned before any variable initializer \( e \) other than the first one
in the local variable declaration statement iff \( v \) is definitely unassigned after the
variable initializer to the left of \( e \) and the initializer expression to the left of \( e \) is
not in the declarator that declares \( v \).

### 16.2.5 Labeled Statements

- \( v \) is [un]assigned after a labeled statement \( L : S \) (where \( L \) is a label) (§14.7) iff \( v \)
is [un]assigned after \( S \) and \( v \) is [un]assigned before every \texttt{break} statement that
may exit the labeled statement \( L : S \).

- \( v \) is [un]assigned before \( S \) iff \( v \) is [un]assigned before \( L : S \).

### 16.2.6 Expression Statements

- \( v \) is [un]assigned after an expression statement \( e ; \) (§14.8) iff it is [un]assigned
after \( e \).

- \( v \) is [un]assigned before \( e \) iff it is [un]assigned before \( e ; \).

### 16.2.7 if Statements

The following rules apply to a statement \( \texttt{if } (e) S \) (§14.9.1):

- \( v \) is [un]assigned after \( \texttt{if } (e) S \) iff \( v \) is [un]assigned after \( S \) and \( v \) is [un]assigned
after \( e \) when false.

- \( v \) is [un]assigned before \( e \) iff \( v \) is [un]assigned before \( \texttt{if } (e) S \).

- \( v \) is [un]assigned before \( S \) iff \( v \) is [un]assigned after \( e \) when true.

The following rules apply to a statement \( \texttt{if } (e) S \texttt{ else } T \) (§14.9.2):

- \( v \) is [un]assigned after \( \texttt{if } (e) S \texttt{ else } T \) iff \( v \) is [un]assigned after \( S \) and \( v \) is
[un]assigned after \( T \).

- \( v \) is [un]assigned before \( e \) iff \( v \) is [un]assigned before \( \texttt{if } (e) S \texttt{ else } T \).

- \( v \) is [un]assigned before \( S \) iff \( v \) is [un]assigned after \( e \) when true.

- \( v \) is [un]assigned before \( T \) iff \( v \) is [un]assigned after \( e \) when false.
16.2.8 **assert Statements**

The following rules apply both to a statement `assert e_1` and to a statement `assert e_1 : e_2` (§14.10):

- \( V \) is [un]assigned before \( e_1 \) iff \( V \) is [un]assigned before the `assert` statement.
- \( V \) is definitely assigned after the `assert` statement iff \( V \) is definitely assigned before the `assert` statement.
- \( V \) is definitely unassigned after the `assert` statement iff \( V \) is definitely unassigned before the `assert` statement and \( V \) is definitely unassigned after \( e_1 \) when true.

The following rule applies to a statement `assert e_1 : e_2`:

- \( V \) is [un]assigned before \( e_2 \) iff \( V \) is [un]assigned after \( e_1 \) when false.

16.2.9 **switch Statements**

- \( V \) is [un]assigned after a `switch` statement (§14.11) iff all of the following are true:
  - Either there is a default label in the switch block or \( V \) is [un]assigned after the switch expression.
  - Either there are no switch labels in the switch block that do not begin a block-statement-group (that is, there are no switch labels immediately before the ")" that ends the switch block) or \( V \) is [un]assigned after the switch expression.
  - Either the switch block contains no block-statement-groups or \( V \) is [un]assigned after the last block-statement of the last block-statement-group.
  - \( V \) is [un]assigned before every `break` statement that may exit the `switch` statement.

- \( V \) is [un]assigned before the switch expression iff \( V \) is [un]assigned before the `switch` statement.

If a switch block contains at least one block-statement-group, then the following rules also apply:

- \( V \) is [un]assigned before the first block-statement of the first block-statement-group in the switch block iff \( V \) is [un]assigned after the switch expression.

- \( V \) is [un]assigned before the first block-statement of any block-statement-group other than the first iff \( V \) is [un]assigned after the switch expression and \( V \) is [un]assigned after the preceding block-statement.
16.2.10 while Statements

- \( v \) is [un]assigned after while \((e)\) \( S (§14.12)\) iff \( v \) is [un]assigned after \( e \) when false and \( v \) is [un]assigned before every break statement for which the while statement is the break target.

- \( v \) is definitely assigned before \( e \) iff \( v \) is definitely assigned before the while statement.

- \( v \) is definitely unassigned before \( e \) iff all of the following are true:
  - \( v \) is definitely unassigned before the while statement.
  - Assuming \( v \) is definitely unassigned before \( e \), \( v \) is definitely unassigned after \( S \).
  - Assuming \( v \) is definitely unassigned before \( e \), \( v \) is definitely unassigned before every continue statement for which the while statement is the continue target.

- \( v \) is [un]assigned before \( S \) iff \( v \) is [un]assigned after \( e \) when true.

16.2.11 do Statements

- \( v \) is [un]assigned after do \( S \) while \((e)\); \((§14.13)\) iff \( v \) is [un]assigned after \( e \) when false and \( v \) is [un]assigned before every break statement for which the do statement is the break target.

- \( v \) is definitely assigned before \( S \) iff \( v \) is definitely assigned before the do statement.

- \( v \) is definitely unassigned before \( S \) iff all of the following are true:
  - \( v \) is definitely unassigned before the do statement.
  - Assuming \( v \) is definitely unassigned before \( S \), \( v \) is definitely unassigned after \( e \) when true.

- \( v \) is [un]assigned before \( e \) iff \( v \) is [un]assigned after \( S \) and \( v \) is [un]assigned before every continue statement for which the do statement is the continue target.

16.2.12 for Statements

The rules herein cover the basic for statement \((§14.14.1)\). Since the enhanced for statement \((§14.14.2)\) is defined by translation to a basic for statement, no special rules need to be provided for it.

- \( v \) is [un]assigned after a for statement iff both of the following are true:
Either a condition expression is not present or $v$ is [un]assigned after the condition expression when false.

$V$ is [un]assigned before every break statement for which the for statement is the break target.

- $V$ is [un]assigned before the initialization part of the for statement iff $V$ is [un]assigned before the for statement.
- $V$ is definitely assigned before the condition part of the for statement iff $V$ is definitely assigned after the initialization part of the for statement.
- $V$ is definitely unassigned before the condition part of the for statement iff both of the following are true:
  - $V$ is definitely unassigned after the initialization part of the for statement.
  - Assuming $V$ is definitely unassigned before the condition part of the for statement, $V$ is definitely unassigned after the incrementation part of the for statement.

- $V$ is [un]assigned before the contained statement iff either of the following is true:
  - A condition expression is present and $V$ is [un]assigned after the condition expression when true.
  - No condition expression is present and $V$ is [un]assigned before the condition part of the for statement.
- $V$ is [un]assigned before the incrementation part of the for statement iff $V$ is [un]assigned after the contained statement and $V$ is [un]assigned before every continue statement for which the for statement is the continue target.

16.2.12.1 Initialization Part of for Statement

- If the initialization part of the for statement is a local variable declaration statement, the rules of §16.2.4 apply.
- Otherwise, if the initialization part is empty, then $V$ is [un]assigned after the initialization part iff $V$ is [un]assigned before the initialization part.
- Otherwise, three rules apply:
  - $V$ is [un]assigned after the initialization part iff $V$ is [un]assigned after the last expression statement in the initialization part.
– \( v \) is [un]assigned before the first expression statement in the initialization part iff \( v \) is [un]assigned before the initialization part.

– \( v \) is [un]assigned before an expression statement \( s \) other than the first in the initialization part iff \( v \) is [un]assigned after the expression statement immediately preceding \( s \).

### 16.2.12.2 Incrementation Part of \texttt{for} Statement

- If the incrementation part of the \texttt{for} statement is empty, then \( v \) is [un]assigned after the incrementation part iff \( v \) is [un]assigned before the incrementation part.

- Otherwise, three rules apply:
  – \( v \) is [un]assigned after the incrementation part iff \( v \) is [un]assigned after the last expression statement in the incrementation part.
  – \( v \) is [un]assigned before the first expression statement in the incrementation part iff \( v \) is [un]assigned before the incrementation part.
  – \( v \) is [un]assigned before an expression statement \( s \) other than the first in the incrementation part iff \( v \) is [un]assigned after the expression statement immediately preceding \( s \).

### 16.2.13 break, continue, return, and throw Statements

- By convention, we say that \( v \) is [un]assigned after any \texttt{break}, \texttt{continue}, \texttt{return}, or \texttt{throw} statement (§14.15, §14.16, §14.17, §14.18).

The notion that a variable is "[un]assigned after" a statement or expression really means "is [un]assigned after the statement or expression completes normally". Because a \texttt{break}, \texttt{continue}, \texttt{return}, or \texttt{throw} statement never completes normally, it vacuously satisfies this notion.

- In a \texttt{return} statement with an expression \( e \) or a \texttt{throw} statement with an expression \( e \), \( v \) is [un]assigned before \( e \) iff \( v \) is [un]assigned before the \texttt{return} or \texttt{throw} statement.

### 16.2.14 synchronized Statements

- \( v \) is [un]assigned after \texttt{synchronized} (\( e \)) \( S \) (§14.19) iff \( v \) is [un]assigned after \( S \).

- \( v \) is [un]assigned before \( e \) iff \( v \) is [un]assigned before the statement \texttt{synchronized} (\( e \)) \( S \).

- \( v \) is [un]assigned before \( S \) iff \( v \) is [un]assigned after \( e \).
16.2.15 try Statements

These rules apply to every try statement (§14.20), whether or not it has a finally block:

- \( V \) is [un]assigned before the try block iff \( V \) is [un]assigned before the try statement.
- \( V \) is definitely assigned before a catch block iff \( V \) is definitely assigned before the try block.
- \( V \) is definitely unassigned before a catch block iff all of the following are true:
  - \( V \) is definitely unassigned after the try block.
  - \( V \) is definitely unassigned before every return statement that belongs to the try block.
  - \( V \) is definitely unassigned after \( e \) in every statement of the form throw \( e \) that belongs to the try block.
  - \( V \) is definitely unassigned after every assert statement that occurs in the try block.
  - \( V \) is definitely unassigned before every break statement that belongs to the try block and whose break target contains (or is) the try statement.
  - \( V \) is definitely unassigned before every continue statement that belongs to the try block and whose continue target contains the try statement.

If a try statement does not have a finally block, then this rule also applies:

- \( V \) is [un]assigned after the try statement iff \( V \) is [un]assigned after the try block and \( V \) is [un]assigned after every catch block in the try statement.

If a try statement does have a finally block, then these rules also apply:

- \( V \) is definitely assigned after the try statement iff at least one of the following is true:
  - \( V \) is definitely assigned after the try block and \( V \) is definitely assigned after every catch block in the try statement.
  - \( V \) is definitely assigned after the finally block.
- \( V \) is definitely unassigned after the try statement iff \( V \) is definitely unassigned after the finally block.
- \( V \) is definitely assigned before the finally block iff \( V \) is definitely assigned before the try statement.
• $V$ is definitely unassigned before the `finally` block iff all of the following are true:
  – $V$ is definitely unassigned after the `try` block.
  – $V$ is definitely unassigned before every `return` statement that belongs to the `try` block.
  – $V$ is definitely unassigned after $e$ in every statement of the form `throw e` that belongs to the `try` block.
  – $V$ is definitely unassigned after every `assert` statement that occurs in the `try` block.
  – $V$ is definitely unassigned before every `break` statement that belongs to the `try` block and whose break target contains (or is) the `try` statement.
  – $V$ is definitely unassigned before every `continue` statement that belongs to the `try` block and whose continue target contains the `try` statement.
  – $V$ is definitely unassigned after every `catch` block of the `try` statement.

16.3 Definite Assignment and Parameters

• A formal parameter $V$ of a method or constructor (§8.4.1, §8.8.1) is definitely assigned (and moreover is not definitely unassigned) before the body of the method or constructor.

• An exception parameter $V$ of a `catch` clause (§14.20) is definitely assigned (and moreover is not definitely unassigned) before the body of the `catch` clause.

16.4 Definite Assignment and Array Initializers

• $V$ is [un]assigned after an empty array initializer (§10.6) iff $V$ is [un]assigned before the empty array initializer.

• $V$ is [un]assigned after a non-empty array initializer iff $V$ is [un]assigned after the last variable initializer in the array initializer.

• $V$ is [un]assigned before the first variable initializer of the array initializer iff $V$ is [un]assigned before the array initializer.
• \( v \) is [un]assigned before any other variable initializer \( e \) of the array initializer iff \( v \) is [un]assigned after the variable initializer to the left of \( e \) in the array initializer.

### 16.5 Definite Assignment and Enum Constants

The rules determining when a variable is definitely assigned or definitely unassigned before an enum constant (§8.9.1) are given in §16.8.

This is because an enum constant is essentially a static final field (§8.3.1.1, §8.3.1.2) that is initialized with a class instance creation expression (§15.9).

• \( v \) is definitely assigned before the declaration of a class body of an enum constant with no arguments that is declared within the scope of \( v \) iff \( v \) is definitely assigned before the enum constant.

• \( v \) is definitely assigned before the declaration of a class body of an enum constant with arguments that is declared within the scope of \( v \) iff \( v \) is definitely assigned after the last argument expression of the enum constant.

The definite assignment/unassignment status of any construct within the class body of an enum constant is governed by the usual rules for classes.

• \( v \) is [un]assigned before the first argument to an enum constant iff it is [un]assigned before the enum constant.

• \( v \) is [un]assigned before \( y \) (an argument of an enum constant, but not the first) iff \( v \) is [un]assigned after the argument to the left of \( y \).

### 16.6 Definite Assignment and Anonymous Classes

• \( v \) is definitely assigned before an anonymous class declaration (§15.9.5) that is declared within the scope of \( v \) iff \( v \) is definitely assigned after the class instance creation expression that declares the anonymous class.

It should be clear that if an anonymous class is implicitly defined by an enum constant, the rules of §16.5 apply.
16.7 Definite Assignment and Member Types

Let \( C \) be a class, and let \( V \) be a blank final member field of \( C \). Then:

- \( V \) is definitely assigned (and moreover, not definitely unassigned) before the declaration of any member type (§8.5, §9.5) of \( C \).

Let \( C \) be a class declared within the scope of \( V \). Then:

- \( V \) is definitely assigned before a member type declaration of \( C \) iff \( V \) is definitely assigned before the declaration of \( C \).

16.8 Definite Assignment and Static Initializers

Let \( C \) be a class declared within the scope of \( V \). Then:

- \( V \) is definitely assigned before an enum constant (§8.9.1) or static variable initializer (§8.3.2) of \( C \) iff \( V \) is definitely assigned before the declaration of \( C \).

Note that there are no rules that would allow us to conclude that \( V \) is definitely unassigned before a static variable initializer or enum constant. We can informally conclude that \( V \) is not definitely unassigned before any static variable initializer of \( C \), but there is no need for such a rule to be stated explicitly.

Let \( C \) be a class, and let \( V \) be a blank static final member field of \( C \), declared in \( C \). Then:

- \( V \) is definitely unassigned (and moreover is not definitely assigned) before the leftmost enum constant, static initializer (§8.7), or static variable initializer of \( C \).
- \( V \) is [un]assigned before an enum constant, static initializer, or static variable initializer of \( C \) other than the leftmost iff \( V \) is [un]assigned after the preceding enum constant, static initializer, or static variable initializer of \( C \).

Let \( C \) be a class, and let \( V \) be a blank static final member field of \( C \), declared in a superclass of \( C \). Then:

- \( V \) is definitely assigned (and moreover is not definitely unassigned) before every enum constant of \( C \).
- \( V \) is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a static initializer of \( C \).
- \( V \) is definitely assigned (and moreover is not definitely unassigned) before every static variable initializer of \( C \).
16.9 Definite Assignment, Constructors, and Instance Initializers

Let \( c \) be a class declared within the scope of \( v \). Then:

- \( v \) is definitely assigned before an instance variable initializer (§8.3.2) of \( c \) iff \( v \) is definitely assigned before the declaration of \( c \).

Note that there are no rules that would allow us to conclude that \( v \) is definitely unassigned before an instance variable initializer. We can informally conclude that \( v \) is not definitely unassigned before any instance variable initializer of \( c \), but there is no need for such a rule to be stated explicitly.

Let \( c \) be a class, and let \( v \) be a blank final non-static member field of \( c \), declared in \( c \). Then:

- \( v \) is definitely unassigned (and moreover is not definitely assigned) before the leftmost instance initializer (§8.6) or instance variable initializer of \( c \).

- \( v \) is [un]assigned before an instance initializer or instance variable initializer of \( c \) other than the leftmost iff \( v \) is [un]assigned after the preceding instance initializer or instance variable initializer of \( c \).

The following rules hold within the constructors (§8.8.7) of class \( c \):

- \( v \) is definitely assigned (and moreover is not definitely unassigned) after an alternate constructor invocation (§8.8.7.1).

- \( v \) is definitely unassigned (and moreover is not definitely assigned) before an explicit or implicit superclass constructor invocation (§8.8.7.1).

- If \( c \) has no instance initializers or instance variable initializers, then \( v \) is not definitely assigned (and moreover is definitely unassigned) after an explicit or implicit superclass constructor invocation.

- If \( c \) has at least one instance initializer or instance variable initializer then \( v \) is [un]assigned after an explicit or implicit superclass constructor invocation iff \( v \) is [un]assigned after the rightmost instance initializer or instance variable initializer of \( c \).

Let \( c \) be a class, and let \( v \) be a blank final member field of \( c \), declared in a superclass of \( c \). Then:

- \( v \) is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a constructor or instance initializer of \( c \).
• $v$ is definitely assigned (and moreover is not definitely unassigned) before every instance variable initializer of $c$. 
While most of the discussion in the preceding chapters is concerned only with the behavior of code as executed a single statement or expression at a time, that is, by a single thread, the Java Virtual Machine can support many threads of execution at once. These threads independently execute code that operates on values and objects residing in a shared main memory. Threads may be supported by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors.

Threads are represented by the Thread class. The only way for a user to create a thread is to create an object of this class; each thread is associated with such an object. A thread will start when the start() method is invoked on the corresponding Thread object.

The behavior of threads, particularly when not correctly synchronized, can be confusing and counterintuitive. This chapter describes the semantics of multithreaded programs; it includes rules for which values may be seen by a read of shared memory that is updated by multiple threads. As the specification is similar to the memory models for different hardware architectures, these semantics are known as the Java programming language memory model. When no confusion can arise, we will simply refer to these rules as "the memory model".

These semantics do not prescribe how a multithreaded program should be executed. Rather, they describe the behaviors that multithreaded programs are allowed to exhibit. Any execution strategy that generates only allowed behaviors is an acceptable execution strategy.
17.1 Synchronization

The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is *synchronization*, which is implemented using *monitors*. Each object in Java is associated with a monitor, which a thread can *lock* or *unlock*. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread *t* may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

The *synchronized* statement (§14.19) computes a reference to an object; it then attempts to perform a lock action on that object’s monitor and does not proceed further until the lock action has successfully completed. After the lock action has been performed, the body of the *synchronized* statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

A *synchronized* method (§8.4.3.6) automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as *this* during execution of the body of the method). If the method is *static*, it locks the monitor associated with the *Class* object that represents the class in which the method is defined. If execution of the method’s body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

The Java programming language neither prevents nor requires detection of deadlock conditions. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that do not deadlock, if necessary.

Other mechanisms, such as reads and writes of *volatile* variables and the use of classes in the *java.util.concurrent* package, provide alternative ways of synchronization.

17.2 Wait Sets and Notification

Every object, in addition to having an associated monitor, has an associated *wait set*. A wait set is a set of threads.
When an object is first created, its wait set is empty. Elementary actions that add threads to and remove threads from wait sets are atomic. Wait sets are manipulated solely through the methods `Object.wait`, `Object.notify`, and `Object.notifyAll`.

Wait set manipulations can also be affected by the interruption status of a thread, and by the `Thread` class's methods dealing with interruption. Additionally, the `Thread` class's methods for sleeping and joining other threads have properties derived from those of wait and notification actions.

### 17.2.1 Wait

Wait actions occur upon invocation of `wait()`, or the timed forms `wait(long millisecs)` and `wait(long millisecs, int nanosecs)`.

A call of `wait(long millisecs)` with a parameter of zero, or a call of `wait(long millisecs, int nanosecs)` with two zero parameters, is equivalent to an invocation of `wait()`.

A thread returns normally from a wait if it returns without throwing an `InterruptedException`.

Let thread `t` be the thread executing the `wait` method on object `m`, and let `n` be the number of lock actions by `t` on `m` that have not been matched by unlock actions. One of the following actions occurs:

- If `n` is zero (i.e., thread `t` does not already possess the lock for target `m`), then an `IllegalMonitorStateException` is thrown.
- If this is a timed wait and the `nanosecs` argument is not in the range of `0-999999` or the `millisecs` argument is negative, then an `IllegalArgumentException` is thrown.
- If thread `t` is interrupted, then an `InterruptedException` is thrown and `t`'s interruption status is set to false.
- Otherwise, the following sequence occurs:
  1. Thread `t` is added to the wait set of object `m`, and performs `n` unlock actions on `m`.
  2. Thread `t` does not execute any further instructions until it has been removed from `m`'s wait set. The thread may be removed from the wait set due to any one of the following actions, and will resume sometime afterward.
17.2 Wait Sets and Notification

Thread and Locks

A notify action being performed on $m$ in which $t$ is selected for removal from the wait set.

A notifyAll action being performed on $m$.

An interrupt action being performed on $t$.

If this is a timed wait, an internal action removing $t$ from $m$'s wait set that occurs after at least $\text{millisecs}$ milliseconds plus $\text{nanosecs}$ nanoseconds elapse since the beginning of this wait action.

An internal action by the implementation. Implementations are permitted, although not encouraged, to perform "spurious wake-ups", that is, to remove threads from wait sets and thus enable resumption without explicit instructions to do so.

Notice that this provision necessitates the Java coding practice of using `wait` only within loops that terminate only when some logical condition that the thread is waiting for holds.

Each thread must determine an order over the events that could cause it to be removed from a wait set. That order does not have to be consistent with other orderings, but the thread must behave as though those events occurred in that order.

For example, if a thread $t$ is in the wait set for $m$, and then both an interrupt of $t$ and a notification of $m$ occur, there must be an order over these events. If the interrupt is deemed to have occurred first, then $t$ will eventually return from `wait` by throwing `InterruptedException`, and some other thread in the wait set for $m$ (if any exist at the time of the notification) must receive the notification. If the notification is deemed to have occurred first, then $t$ will eventually return normally from `wait` with an interrupt still pending.

3. Thread $t$ performs $n$ lock actions on $m$.

4. If thread $t$ was removed from $m$'s wait set in step 2 due to an interrupt, then $t$'s interruption status is set to false and the `wait` method throws `InterruptedException`.

17.2.2 Notification

Notification actions occur upon invocation of methods `notify` and `notifyAll`.

Let thread $t$ be the thread executing either of these methods on object $m$, and let $n$ be the number of lock actions by $t$ on $m$ that have not been matched by unlock actions. One of the following actions occurs:
• If \( n \) is zero, then an `IllegalMonitorStateException` is thrown. This is the case where thread \( t \) does not already possess the lock for target \( m \).

• If \( n \) is greater than zero and this is a `notify` action, then if \( m \)'s wait set is not empty, a thread \( u \) that is a member of \( m \)'s current wait set is selected and removed from the wait set.

There is no guarantee about which thread in the wait set is selected. This removal from the wait set enables \( u \)'s resumption in a wait action. Notice, however, that \( u \)'s lock actions upon resumption cannot succeed until some time after \( t \) fully unlocks the monitor for \( m \).

• If \( n \) is greater than zero and this is a `notifyAll` action, then all threads are removed from \( m \)'s wait set, and thus resume.

Notice, however, that only one of them at a time will lock the monitor required during the resumption of wait.

### 17.2.3 Interruptions

Interruption actions occur upon invocation of `Thread.interrupt`, as well as methods defined to invoke it in turn, such as `ThreadGroup.interrupt`.

Let \( t \) be the thread invoking \( u \).`interrupt`, for some thread \( u \), where \( t \) and \( u \) may be the same. This action causes \( u \)'s interruption status to be set to true.

Additionally, if there exists some object \( m \) whose wait set contains \( u \), then \( u \) is removed from \( m \)'s wait set. This enables \( u \) to resume in a wait action, in which case this wait will, after re-locking \( m \)'s monitor, throw `InterruptedException`.

Invocations of `Thread.isInterrupted` can determine a thread's interruption status. The static method `Thread.interrupted` may be invoked by a thread to observe and clear its own interruption status.

### 17.2.4 Interactions of Waits, Notification, and Interruption

The above specifications allow us to determine several properties having to do with the interaction of waits, notification, and interruption.

If a thread is both notified and interrupted while waiting, it may either:

• return normally from `wait`, while still having a pending interrupt (in other words, a call to `Thread.interrupted` would return true)

• return from `wait` by throwing an `InterruptedException`
The thread may not reset its interrupt status and return normally from the call to \texttt{wait}.

Similarly, notifications cannot be lost due to interrupts. Assume that a set $s$ of threads is in the wait set of an object $m$, and another thread performs a \texttt{notify} on $m$. Then either:

- at least one thread in $s$ must return normally from \texttt{wait}, or
- all of the threads in $s$ must exit \texttt{wait} by throwing \texttt{InterruptedException}

Note that if a thread is both interrupted and woken via \texttt{notify}, and that thread returns from \texttt{wait} by throwing an \texttt{InterruptedException}, then some other thread in the wait set must be notified.

### 17.3 Sleep and Yield

\texttt{Thread.sleep} causes the currently executing thread to sleep (temporarily cease execution) for the specified duration, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors, and resumption of execution will depend on scheduling and the availability of processors on which to execute the thread.

It is important to note that neither \texttt{Thread.sleep} nor \texttt{Thread.yield} have any synchronization semantics. In particular, the compiler does not have to flush writes cached in registers out to shared memory before a call to \texttt{Thread.sleep} or \texttt{Thread.yield}, nor does the compiler have to reload values cached in registers after a call to \texttt{Thread.sleep} or \texttt{Thread.yield}.

For example, in the following (broken) code fragment, assume that \texttt{this.done} is a non-volatile boolean field:

```java
while (!this.done)
    Thread.sleep(1000);
```

The compiler is free to read the field \texttt{this.done} just once, and reuse the cached value in each execution of the loop. This would mean that the loop would never terminate, even if another thread changed the value of \texttt{this.done}. 

17.4 Memory Model

A memory model describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. The Java programming language memory model works by examining each read in an execution trace and checking that the write observed by that read is valid according to certain rules.

The memory model describes possible behaviors of a program. An implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model.

This provides a great deal of freedom for the implementor to perform a myriad of code transformations, including the reordering of actions and removal of unnecessary synchronization.

Example 17.4-1. Incorrectly Synchronized Programs May Exhibit Surprising Behavior

The semantics of the Java programming language allow compilers and microprocessors to perform optimizations that can interact with incorrectly synchronized code in ways that can produce behaviors that seem paradoxical. Here are some examples of how incorrectly synchronized programs may exhibit surprising behaviors.

Consider, for example, the example program traces shown in Table 17.4-A. This program uses local variables r1 and r2 and shared variables A and B. Initially, A == B == 0.

Table 17.4-A. Surprising results caused by statement reordering - original code

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: r2 = A;</td>
<td>3: r1 = B;</td>
</tr>
<tr>
<td>2: B = 1;</td>
<td>4: A = 2;</td>
</tr>
</tbody>
</table>

It may appear that the result r2 == 2 and r1 == 1 is impossible. Intuitively, either instruction 1 or instruction 3 should come first in an execution. If instruction 1 comes first, it should not be able to see the write at instruction 4. If instruction 3 comes first, it should not be able to see the write at instruction 2.

If some execution exhibited this behavior, then we would know that instruction 4 came before instruction 1, which came before instruction 2, which came before instruction 3, which came before instruction 4. This is, on the face of it, absurd.

However, compilers are allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation. If instruction 1 is reordered with instruction 2, as shown in the trace in Table 17.4-B, then it is easy to see how the result r2 == 2 and r1 == 1 might occur.
To some programmers, this behavior may seem "broken". However, it should be noted that this code is improperly synchronized:

- there is a write in one thread,
- a read of the same variable by another thread,
- and the write and read are not ordered by synchronization.

This situation is an example of a data race (§17.4.5). When code contains a data race, counterintuitive results are often possible.

Several mechanisms can produce the reordering in Table 17.4-B. A Just-In-Time compiler in a Java Virtual Machine implementation may rearrange code, or the processor. In addition, the memory hierarchy of the architecture on which a Java Virtual Machine implementation is run may make it appear as if code is being reordered. In this chapter, we shall refer to anything that can reorder code as a compiler.

Another example of surprising results can be seen in Table 17.4-C. Initially, \( p == q \) and \( p.x == 0 \). This program is also incorrectly synchronized; it writes to shared memory without enforcing any ordering between those writes.

One common compiler optimization involves having the value read for \( r2 \) reused for \( r5 \); they are both reads of \( r1.x \) with no intervening write. This situation is shown in Table 17.4-D.
Table 17.4-D. Surprising results caused by forward substitution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 = p; )</td>
<td>( r_6 = p; )</td>
</tr>
<tr>
<td>( r_2 = r_1.x; )</td>
<td>( r_6.x = 3; )</td>
</tr>
<tr>
<td>( r_3 = q; )</td>
<td></td>
</tr>
<tr>
<td>( r_4 = r_3.x; )</td>
<td></td>
</tr>
<tr>
<td>( r_5 = r_2; )</td>
<td></td>
</tr>
</tbody>
</table>

Now consider the case where the assignment to \( r_6.x \) in Thread 2 happens between the first read of \( r_1.x \) and the read of \( r_3.x \) in Thread 1. If the compiler decides to reuse the value of \( r_2 \) for the \( r_5 \), then \( r_2 \) and \( r_5 \) will have the value 0, and \( r_4 \) will have the value 3. From the perspective of the programmer, the value stored at \( p.x \) has changed from 0 to 3 and then changed back.

The memory model determines what values can be read at every point in the program. The actions of each thread in isolation must behave as governed by the semantics of that thread, with the exception that the values seen by each read are determined by the memory model. When we refer to this, we say that the program obeys *intra-thread semantics*. Intra-thread semantics are the semantics for single-threaded programs, and allow the complete prediction of the behavior of a thread based on the values seen by read actions within the thread. To determine if the actions of thread \( t \) in an execution are legal, we simply evaluate the implementation of thread \( t \) as it would be performed in a single-threaded context, as defined in the rest of this specification.

Each time the evaluation of thread \( t \) generates an inter-thread action, it must match the inter-thread action \( a \) of \( t \) that comes next in program order. If \( a \) is a read, then further evaluation of \( t \) uses the value seen by \( a \) as determined by the memory model.

This section provides the specification of the Java programming language memory model except for issues dealing with *final* fields, which are described in §17.5.

The memory model specified herein is not fundamentally based in the object-oriented nature of the Java programming language. For conciseness and simplicity in our examples, we often exhibit code fragments without class or method definitions, or explicit dereferencing. Most examples consist of two or more threads containing statements with access to local variables, shared global variables, or instance fields of an object. We typically use variables names such as \( r_1 \) or \( r_2 \) to indicate variables local to a method or thread. Such variables are not accessible by other threads.
17.4 Memory Model

17.4.1 Shared Variables

Memory that can be shared between threads is called *shared memory* or *heap memory*.

All instance fields, *static* fields, and array elements are stored in heap memory. In this chapter, we use the term *variable* to refer to both fields and array elements.

Local variables (§14.4), formal method parameters (§8.4.1), and exception handler parameters (§14.20) are never shared between threads and are unaffected by the memory model.

Two accesses to (reads of or writes to) the same variable are said to be *conflicting* if at least one of the accesses is a write.

17.4.2 Actions

An *inter-thread action* is an action performed by one thread that can be detected or directly influenced by another thread. There are several kinds of inter-thread action that a program may perform:

- *Read* (normal, or non-volatile). Reading a variable.
- *Write* (normal, or non-volatile). Writing a variable.
- *Synchronization actions*, which are:
  - *Volatile read*. A volatile read of a variable.
  - *Volatile write*. A volatile write of a variable.
  - *Lock*. Locking a monitor
  - *Unlock*. Unlocking a monitor.
  - The (synthetic) first and last action of a thread.
  - Actions that start a thread or detect that a thread has terminated (§17.4.4).
- *External Actions*. An external action is an action that may be observable outside of an execution, and has a result based on an environment external to the execution.
- *Thread divergence actions* (§17.4.9). A thread divergence action is only performed by a thread that is in an infinite loop in which no memory, synchronization, or external actions are performed. If a thread performs a thread divergence action, it will be followed by an infinite number of thread divergence actions.
Thread divergence actions are introduced to model how a thread may cause all other threads to stall and fail to make progress.

This specification is only concerned with inter-thread actions. We do not need to concern ourselves with intra-thread actions (e.g., adding two local variables and storing the result in a third local variable). As previously mentioned, all threads need to obey the correct intra-thread semantics for Java programs. We will usually refer to inter-thread actions more succinctly as simply actions.

An action $a$ is described by a tuple $< t, k, v, u >$, comprising:

- $t$ - the thread performing the action
- $k$ - the kind of action
- $v$ - the variable or monitor involved in the action.
  
  For lock actions, $v$ is the monitor being locked; for unlock actions, $v$ is the monitor being unlocked.
  
  If the action is a (volatile or non-volatile) read, $v$ is the variable being read.
  
  If the action is a (volatile or non-volatile) write, $v$ is the variable being written.
- $u$ - an arbitrary unique identifier for the action

An external action tuple contains an additional component, which contains the results of the external action as perceived by the thread performing the action. This may be information as to the success or failure of the action, and any values read by the action.

Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple. These parameters are set up by other actions within the thread and can be determined by examining the intra-thread semantics. They are not explicitly discussed in the memory model.

In non-terminating executions, not all external actions are observable. Non-terminating executions and observable actions are discussed in §17.4.9.

### 17.4.3 Programs and Program Order

Among all the inter-thread actions performed by each thread $t$, the **program order** of $t$ is a total order that reflects the order in which these actions would be performed according to the intra-thread semantics of $t$. 
A set of actions is *sequentially consistent* if all actions occur in a total order (the execution order) that is consistent with program order, and furthermore, each read \( r \) of a variable \( v \) sees the value written by the write \( w \) to \( v \) such that:

- \( w \) comes before \( r \) in the execution order, and
- there is no other write \( w' \) such that \( w \) comes before \( w' \) and \( w' \) comes before \( r \) in the execution order.

Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. Within a sequentially consistent execution, there is a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and each individual action is atomic and is immediately visible to every thread.

If a program has no data races, then all executions of the program will appear to be sequentially consistent.

Sequential consistency and/or freedom from data races still allows errors arising from groups of operations that need to be perceived atomically and are not.

If we were to use sequential consistency as our memory model, many of the compiler and processor optimizations that we have discussed would be illegal. For example, in the trace in Table 17.4-C, as soon as the write of 3 to \( p.x \) occurred, subsequent reads of that location would be required to see that value.

### 17.4.4 Synchronization Order

Every execution has a *synchronization order*. A synchronization order is a total order over all of the synchronization actions of an execution. For each thread \( t \), the synchronization order of the synchronization actions (§17.4.2) in \( t \) is consistent with the program order (§17.4.3) of \( t \).

Synchronization actions induce the *synchronized-with* relation on actions, defined as follows:

- An unlock action on monitor \( m \) synchronizes-with all subsequent lock actions on \( m \) (where "subsequent" is defined according to the synchronization order).
- A write to a volatile variable \( v \) (§8.3.1.4) synchronizes-with all subsequent reads of \( v \) by any thread (where "subsequent" is defined according to the synchronization order).
- An action that starts a thread synchronizes-with the first action in the thread it starts.
• The write of the default value (zero, false, or null) to each variable synchronizes-with the first action in every thread.

   Although it may seem a little strange to write a default value to a variable before the object containing the variable is allocated, conceptually every object is created at the start of the program with its default initialized values.

• The final action in a thread \( T_1 \) synchronizes-with any action in another thread \( T_2 \) that detects that \( T_1 \) has terminated.

   \( T_2 \) may accomplish this by calling \( T_1.\text{isAlive()} \) or \( T_1.\text{join()} \).

• If thread \( T_1 \) interrupts thread \( T_2 \), the interrupt by \( T_1 \) synchronizes-with any point where any other thread (including \( T_2 \)) determines that \( T_2 \) has been interrupted (by having an InterruptedException thrown or by invoking Thread.interrupted or Thread.isInterrupted).

The source of a synchronizes-with edge is called a release, and the destination is called an acquire.

### 17.4.5 Happens-before Order

Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second.

If we have two actions \( x \) and \( y \), we write \( \text{hb}(x, y) \) to indicate that \( x \) happens-before \( y \).

• If \( x \) and \( y \) are actions of the same thread and \( x \) comes before \( y \) in program order, then \( \text{hb}(x, y) \).

• There is a happens-before edge from the end of a constructor of an object to the start of a finalizer (§12.6) for that object.

• If an action \( x \) synchronizes-with a following action \( y \), then we also have \( \text{hb}(x, y) \).

• If \( \text{hb}(x, y) \) and \( \text{hb}(y, z) \), then \( \text{hb}(x, z) \).

The wait methods of class Object (§17.2.1) have lock and unlock actions associated with them; their happens-before relationships are defined by these associated actions.

It should be noted that the presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal.
For example, the write of a default value to every field of an object constructed by a thread need not happen before the beginning of that thread, as long as no read ever observes that fact.

More specifically, if two actions share a \textit{happens-before} relationship, they do not necessarily have to appear to have happened in that order to any code with which they do not share a \textit{happens-before} relationship. Writes in one thread that are in a data race with reads in another thread may, for example, appear to occur out of order to those reads.

The \textit{happens-before} relation defines when data races take place.

A set of synchronization edges, $S$, is \textit{sufficient} if it is the minimal set such that the transitive closure of $S$ with the program order determines all of the \textit{happens-before} edges in the execution. This set is unique.

It follows from the above definitions that:

- An unlock on a monitor \textit{happens-before} every subsequent lock on that monitor.
- A write to a \texttt{volatile} field (§8.3.1.4) \textit{happens-before} every subsequent read of that field.
- A call to \texttt{start()} on a thread \textit{happens-before} any actions in the started thread.
- All actions in a thread \textit{happen-before} any other thread successfully returns from a \texttt{join()} on that thread.
- The default initialization of any object \textit{happens-before} any other actions (other than default-writes) of a program.

When a program contains two conflicting accesses (§17.4.1) that are not ordered by a happens-before relationship, it is said to contain a \textit{data race}.

The semantics of operations other than inter-thread actions, such as reads of array lengths (§10.7), executions of checked casts (§5.5, §15.16), and invocations of virtual methods (§15.12), are not directly affected by data races.

Therefore, a data race cannot cause incorrect behavior such as returning the wrong length for an array.

A program is \textit{correctly synchronized} if and only if all sequentially consistent executions are free of data races.

If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent (§17.4.3).
This is an extremely strong guarantee for programmers. Programmers do not need to reason about reorderings to determine that their code contains data races. Therefore they do not need to reason about reorderings when determining whether their code is correctly synchronized. Once the determination that the code is correctly synchronized is made, the programmer does not need to worry that reorderings will affect his or her code.

A program must be correctly synchronized to avoid the kinds of counterintuitive behaviors that can be observed when code is reordered. The use of correct synchronization does not ensure that the overall behavior of a program is correct. However, its use does allow a programmer to reason about the possible behaviors of a program in a simple way; the behavior of a correctly synchronized program is much less dependent on possible reorderings. Without correct synchronization, very strange, confusing and counterintuitive behaviors are possible.

We say that a read \( r \) of a variable \( v \) is allowed to observe a write \( w \) to \( v \) if, in the \textit{happens-before} partial order of the execution trace:

- \( r \) is not ordered before \( w \) (i.e., it is not the case that \( hb(r, w) \)), and
- there is no intervening write \( w' \) to \( v \) (i.e. no write \( w' \) to \( v \) such that \( hb(w, w') \) and \( hb(w', r) \)).

Informally, a read \( r \) is allowed to see the result of a write \( w \) if there is no \textit{happens-before} ordering to prevent that read.

A set of actions \( A \) is \textit{happens-before consistent} if for all reads \( r \) in \( A \), where \( W(r) \) is the write action seen by \( r \), it is not the case that either \( hb(r, W(r)) \) or that there exists a write \( w \) in \( A \) such that \( w.v = r.v \) and \( hb(W(r), w) \) and \( hb(w, r) \).

In a \textit{happens-before consistent} set of actions, each read sees a write that it is allowed to see by the \textit{happens-before} ordering.

\begin{example}
For the trace in Table 17.4.5-A, initially \( A == B == 0 \). The trace can observe \( r2 == 0 \) and \( r1 == 0 \) and still be \textit{happens-before consistent}, since there are execution orders that allow each read to see the appropriate write.
\end{example}

\begin{table}
\begin{center}
\begin{tabular}{|c|c|}
\hline
Thread 1 & Thread 2 \\
\hline
\( B = 1; \) & \( A = 2; \) \\
\( r2 = A; \) & \( r1 = B; \) \\
\hline
\end{tabular}
\end{center}
\end{table}

Since there is no synchronization, each read can see either the write of the initial value or the write by the other thread. An execution order that displays this behavior is:
Another execution order that is happens-before consistent is:

1: r2 = A; // sees write of A = 2
3: r1 = B; // sees write of B = 1
2: B = 1;
4: A = 2;

In this execution, the reads see writes that occur later in the execution order. This may seem counterintuitive, but is allowed by happens-before consistency. Allowing reads to see later writes can sometimes produce unacceptable behaviors.

### 17.4.6 Executions

An execution $E$ is described by a tuple $< P, A, po, so, W, V, sw, hb >$, comprising:

- $P$ - a program
- $A$ - a set of actions
- $po$ - program order, which for each thread $t$, is a total order over all actions performed by $t$ in $A$
- $so$ - synchronization order, which is a total order over all synchronization actions in $A$
- $W$ - a write-seen function, which for each read $r$ in $A$, gives $W(r)$, the write action seen by $r$ in $E$.
- $V$ - a value-written function, which for each write $w$ in $A$, gives $V(w)$, the value written by $w$ in $E$.
- $sw$ - synchronizes-with, a partial order over synchronization actions
- $hb$ - happens-before, a partial order over actions

Note that the synchronizes-with and happens-before elements are uniquely determined by the other components of an execution and the rules for well-formed executions (§17.4.7).

An execution is **happens-before consistent** if its set of actions is **happens-before consistent** (§17.4.5).
17.4.7 Well-Formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, po, so, W, V, sw, hb \rangle$ is well formed if the following are true:

1. Each read sees a write to the same variable in the execution.

   All reads and writes of volatile variables are volatile actions. For all reads $r$ in $A$, we have $W(r) \in A$ and $W(r).v = r.v$. The variable $r.v$ is volatile if and only if $r$ is a volatile read, and the variable $w.v$ is volatile if and only if $w$ is a volatile write.

2. The happens-before order is a partial order.

   The happens-before order is given by the transitive closure of synchronizes-with edges and program order. It must be a valid partial order: reflexive, transitive and antisymmetric.

3. The execution obeys intra-thread consistency.

   For each thread $t$, the actions performed by $t$ in $A$ are the same as would be generated by that thread in program-order in isolation, with each write $w$ writing the value $V(w)$, given that each read $r$ sees the value $V(W(r))$. Values seen by each read are determined by the memory model. The program order given must reflect the program order in which the actions would be performed according to the intra-thread semantics of $P$.

4. The execution is happens-before consistent (§17.4.6).

5. The execution obeys synchronization-order consistency.

   For all volatile reads $r$ in $A$, it is not the case that either $so(r, W(r))$ or that there exists a write $w$ in $A$ such that $w.v = r.v$ and $so(W(r), w)$ and $so(w, r)$.

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of $f$ to $d$. For all $x$ in $d$, $f|_d(x) = f(x)$, and for all $x$ not in $d$, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order $p$ to the elements in $d$. For all $x,y$ in $d$, $p(x,y)$ if and only if $p|_d(x,y)$. If either $x$ or $y$ are not in $d$, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = \langle P, A, po, so, W, V, sw, hb \rangle$ is validated by committing actions from $A$. If all of the actions in $A$ can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.
Starting with the empty set as $C_0$, we perform a sequence of steps where we take actions from the set of actions $A$ and add them to a set of committed actions $C_i$ to get a new set of committed actions $C_{i+1}$. To demonstrate that this is reasonable, for each $C_i$ we need to demonstrate an execution $E$ containing $C_i$ that meets certain conditions.

Formally, an execution $E$ satisfies the causality requirements of the Java programming language memory model if and only if there exist:

- Sets of actions $C_0, C_1, \ldots$ such that:
  - $C_0$ is the empty set
  - $C_i$ is a proper subset of $C_{i+1}$
  - $A = \cup (C_0, C_1, \ldots)$
  - If $A$ is finite, then the sequence $C_0, C_1, \ldots$ will be finite, ending in a set $C_n = A$.
  - If $A$ is infinite, then the sequence $C_0, C_1, \ldots$ may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to $A$.

- Well-formed executions $E_1, \ldots$, where $E_i = < P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i >$.

Given these sets of actions $C_0, \ldots$ and executions $E_1, \ldots$, every action in $C_i$ must be one of the actions in $E_i$. All actions in $C_i$ must share the same relative happens-before order and synchronization order in both $E_i$ and $E$. Formally:

1. $C_i$ is a subset of $A_i$
2. $hb_{| C_i} = hb_{| C_i}$
3. $so_{| C_i} = so_{| C_i}$

The values written by the writes in $C_i$ must be the same in both $E_i$ and $E$. Only the reads in $C_{i-1}$ need to see the same writes in $E_i$ as in $E$. Formally:

4. $V_{| C_i} = V_{| C_i}$
5. $W_{| C_{i-1}} = W_{| C_{i-1}}$

All reads in $E_i$ that are not in $C_{i-1}$ must see writes that happen-before them. Each read $r$ in $C_i - C_{i-1}$ must see writes in $C_{i-1}$ in both $E_i$ and $E$, but may see a different write in $E_i$ from the one it sees in $E$. Formally:

6. For any read $r$ in $A_i - C_{i-1}$, we have $hb_{i}(W_{i}(r), r)$
7. For any read $r$ in $(C_i - C_{i-1})$, we have $W_{i}(r)$ in $C_{i-1}$ and $W(r)$ in $C_{i-1}$
Given a set of sufficient synchronizes-with edges for $E_i$, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all $E_j$, where $j \geq i$. Formally:

8. Let $ssw_i$ be the $sw_i$ edges that are also in the transitive reduction of $hb_i$ but not in $po$. We call $ssw_i$ the sufficient synchronizes-with edges for $E_i$. If $ssw_i(x, y)$ and $hb_i(y, z)$ and $z$ in $C_i$, then $sw_j(x, y)$ for all $j \geq i$.

If an action $y$ is committed, all external actions that happen-before $y$ are also committed.

9. If $y$ is in $C_i$, $x$ is an external action and $hb_j(x, y)$, then $x$ in $C_i$.

Example 17.4.8-1. Happens-before Consistency Is Not Sufficient

Happens-before consistency is a necessary, but not sufficient, set of constraints. Merely enforcing happens-before consistency would allow for unacceptable behaviors - those that violate the requirements we have established for programs. For example, happens-before consistency allows values to appear "out of thin air". This can be seen by a detailed examination of the trace in Table 17.4.8-A.

Table 17.4.8-A. Happens-before consistency is not sufficient

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r1 = x;$</td>
<td>$r2 = y;$</td>
</tr>
<tr>
<td>if $(r1 != 0)$ $y = 1;$</td>
<td>if $(r2 != 0)$ $x = 1;$</td>
</tr>
</tbody>
</table>

The code shown in Table 17.4.8-A is correctly synchronized. This may seem surprising, since it does not perform any synchronization actions. Remember, however, that a program is correctly synchronized if, when it is executed in a sequentially consistent manner, there are no data races. If this code is executed in a sequentially consistent way, each action will occur in program order, and neither of the writes will occur. Since no writes occur, there can be no data races: the program is correctly synchronized.

Since this program is correctly synchronized, the only behaviors we can allow are sequentially consistent behaviors. However, there is an execution of this program that is happens-before consistent, but not sequentially consistent:

$r1 = x;$  // sees write of $x = 1$
$y = 1$;
$r2 = y;$  // sees write of $y = 1$
$x = 1$;

This result is happens-before consistent: there is no happens-before relationship that prevents it from occurring. However, it is clearly not acceptable: there is no sequentially consistent execution that would result in this behavior. The fact that we allow a read to see a write that comes later in the execution order can sometimes thus result in unacceptable behaviors.
Although allowing reads to see writes that come later in the execution order is sometimes undesirable, it is also sometimes necessary. As we saw above, the trace in Table 17.4.5-A requires some reads to see writes that occur later in the execution order. Since the reads come first in each thread, the very first action in the execution order must be a read. If that read cannot see a write that occurs later, then it cannot see any value other than the initial value for the variable it reads. This is clearly not reflective of all behaviors.

We refer to the issue of when reads can see future writes as causality, because of issues that arise in cases like the one found in Table 17.4.8-A. In that case, the reads cause the writes to occur, and the writes cause the reads to occur. There is no "first cause" for the actions. Our memory model therefore needs a consistent way of determining which reads can see writes early.

Examples such as the one found in Table 17.4.8-A demonstrate that the specification must be careful when stating whether a read can see a write that occurs later in the execution (bearing in mind that if a read sees a write that occurs later in the execution, it represents the fact that the write is actually performed early).

The memory model takes as input a given execution, and a program, and determines whether that execution is a legal execution of the program. It does this by gradually building a set of "committed" actions that reflect which actions were executed by the program. Usually, the next action to be committed will reflect the next action that can be performed by a sequentially consistent execution. However, to reflect reads that need to see later writes, we allow some actions to be committed earlier than other actions that happen-before them.

Obviously, some actions may be committed early and some may not. If, for example, one of the writes in Table 17.4.8-A were committed before the read of that variable, the read could see the write, and the "out-of-thin-air" result could occur. Informally, we allow an action to be committed early if we know that the action can occur without assuming some data race occurs. In Table 17.4.8-A, we cannot perform either write early, because the writes cannot occur unless the reads see the result of a data race.

### 17.4.9 Observable Behavior and Nonterminating Executions

For programs that always terminate in some bounded finite period of time, their behavior can be understood (informally) simply in terms of their allowable executions. For programs that can fail to terminate in a bounded amount of time, more subtle issues arise.

The observable behavior of a program is defined by the finite sets of external actions that the program may perform. A program that, for example, simply prints "Hello" forever is described by a set of behaviors that for any non-negative integer \( i \), includes the behavior of printing "Hello" \( i \) times.

Termination is not explicitly modeled as a behavior, but a program can easily be extended to generate an additional external action \( \text{executionTermination} \) that occurs when all threads have terminated.
We also define a special *hang* action. If behavior is described by a set of external actions including a *hang* action, it indicates a behavior where after the external actions are observed, the program can run for an unbounded amount of time without performing any additional external actions or terminating. Programs can hang if all threads are blocked or if the program can perform an unbounded number of actions without performing any external actions.

A thread can be blocked in a variety of circumstances, such as when it is attempting to acquire a lock or perform an external action (such as a read) that depends on external data.

An execution may result in a thread being blocked indefinitely and the execution's not terminating. In such cases, the actions generated by the blocked thread must consist of all actions generated by that thread up to and including the action that caused the thread to be blocked, and no actions that would be generated by the thread after that action.

To reason about observable behaviors, we need to talk about sets of observable actions.

If $O$ is a set of observable actions for an execution $E$, then set $O$ must be a subset of $E$'s actions, $A$, and must contain only a finite number of actions, even if $A$ contains an infinite number of actions. Furthermore, if an action $y$ is in $O$, and either $hb(x, y)$ or $so(x, y)$, then $x$ is in $O$.

Note that a set of observable actions are not restricted to external actions. Rather, only external actions that are in a set of observable actions are deemed to be observable external actions.

A behavior $B$ is an allowable behavior of a program $P$ if and only if $B$ is a finite set of external actions and either:

- There exists an execution $E$ of $P$, and a set $O$ of observable actions for $E$, and $B$ is the set of external actions in $O$ (If any threads in $E$ end in a blocked state and $O$ contains all actions in $E$, then $B$ may also contain a *hang* action); or

- There exists a set $O$ of actions such that $B$ consists of a *hang* action plus all the external actions in $O$ and for all $k \geq |O|$, there exists an execution $E$ of $P$ with actions $A$, and there exists a set of actions $O'$ such that:
  
  - Both $O$ and $O'$ are subsets of $A$ that fulfill the requirements for sets of observable actions.
  
  - $O \subseteq O' \subseteq A$

  - $|O'| \geq k$
– \( O' \) - \( O \) contains no external actions

Note that a behavior \( B \) does not describe the order in which the external actions in \( B \) are observed, but other (internal) constraints on how the external actions are generated and performed may impose such constraints.

### 17.5 final Field Semantics

Fields declared final are initialized once, but never changed under normal circumstances. The detailed semantics of final fields are somewhat different from those of normal fields. In particular, compilers have a great deal of freedom to move reads of final fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, compilers are allowed to keep the value of a final field cached in a register and not reload it from memory in situations where a non-final field would have to be reloaded.

final fields also allow programmers to implement thread-safe immutable objects without synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads. This can provide safety guarantees against misuse of an immutable class by incorrect or malicious code. final fields must be used correctly to provide a guarantee of immutability.

An object is considered to be completely initialized when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields.

The usage model for final fields is a simple one: Set the final fields for an object in that object's constructor; and do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished. If this is followed, then when the object is seen by another thread, that thread will always see the correctly constructed version of that object's final fields. It will also see versions of any object or array referenced by those final fields that are at least as up-to-date as the final fields are.

**Example 17.5-1. final Fields In The Java Memory Model**

The program below illustrates how final fields compare to normal fields.

```java
class FinalFieldExample {
    final int x;
    int y;
}
```
static FinalFieldExample f;

public FinalFieldExample() {
    x = 3;
    y = 4;
}

static void writer() {
    f = new FinalFieldExample();
}

static void reader() {
    if (f != null) {
        int i = f.x;  // guaranteed to see 3
        int j = f.y;  // could see 0
    }
}

The class FinalFieldExample has a final int field x and a non-final int field y. One thread might execute the method writer and another might execute the method reader.

Because the writer method writes f after the object's constructor finishes, the reader method will be guaranteed to see the properly initialized value for f.x: it will read the value 3. However, f.y is not final; the reader method is therefore not guaranteed to see the value 4 for it.

Example 17.5-2. final Fields For Security

final fields are designed to allow for necessary security guarantees. Consider the following program. One thread (which we shall refer to as thread 1) executes:

Global.s = "/tmp/usr".substring(4);

while another thread (thread 2) executes

String myS = Global.s;
if (myS.equals("/tmp")) System.out.println(myS);

String objects are intended to be immutable and string operations do not perform synchronization. While the String implementation does not have any data races, other code could have data races involving the use of String objects, and the memory model makes weak guarantees for programs that have data races. In particular, if the fields of the String class were not final, then it would be possible (although unlikely) that thread 2 could initially see the default value of 0 for the offset of the string object, allowing it to compare as equal to "/tmp". A later operation on the String object might see the correct offset of 4, so that the String object is perceived as being "/usr". Many security features of the Java programming language depend upon String objects being perceived as truly immutable, even if malicious code is using data races to pass String references between threads.
17.5.1 Semantics of final Fields

Let \( o \) be an object, and \( c \) be a constructor for \( o \) in which a final field \( f \) is written. A freeze action on final field \( f \) of \( o \) takes place when \( c \) exits, either normally or abruptly.

Note that if one constructor invokes another constructor, and the invoked constructor sets a final field, the freeze for the final field takes place at the end of the invoked constructor.

For each execution, the behavior of reads is influenced by two additional partial orders, the dereference chain \( \text{dereferences()} \) and the memory chain \( \text{mc()} \), which are considered to be part of the execution (and thus, fixed for any particular execution). These partial orders must satisfy the following constraints (which need not have a unique solution):

- Dereference Chain: If an action \( a \) is a read or write of a field or element of an object \( o \) by a thread \( t \) that did not initialize \( o \), then there must exist some read \( r \) by thread \( t \) that sees the address of \( o \) such that \( r \text{ dereferences}(r, a) \).

- Memory Chain: There are several constraints on the memory chain ordering:
  - If \( r \) is a read that sees a write \( w \), then it must be the case that \( \text{mc}(w, r) \).
  - If \( r \) and \( a \) are actions such that \( \text{dereferences}(r, a) \), then it must be the case that \( \text{mc}(r, a) \).
  - If \( w \) is a write of the address of an object \( o \) by a thread \( t \) that did not initialize \( o \), then there must exist some read \( r \) by thread \( t \) that sees the address of \( o \) such that \( \text{mc}(r, w) \).

Given a write \( w \), a freeze \( f \), an action \( a \) (that is not a read of a final field), a read \( r_1 \) of the final field frozen by \( f \), and a read \( r_2 \) such that \( \text{hb}(w, f), \text{hb}(f, a), \text{mc}(a, r_1) \), and \( \text{dereferences}(r_1, r_2) \), then when determining which values can be seen by \( r_2 \), we consider \( \text{hb}(w, r_2) \). (This happens-before ordering does not transitively close with other happens-before orderings.)

Note that the dereferences order is reflexive, and \( r_1 \) can be the same as \( r_2 \).

For reads of final fields, the only writes that are deemed to come before the read of the final field are the ones derived through the final field semantics.

17.5.2 Reading final Fields During Construction

A read of a final field of an object within the thread that constructs that object is ordered with respect to the initialization of that field within the constructor by the
usual *happens-before* rules. If the read occurs after the field is set in the constructor, it sees the value the `final` field is assigned, otherwise it sees the default value.

### 17.5.3 Subsequent Modification of `final` Fields

In some cases, such as deserialization, the system will need to change the `final` fields of an object after construction. `final` fields can be changed via reflection and other implementation-dependent means. The only pattern in which this has reasonable semantics is one in which an object is constructed and then the `final` fields of the object are updated. The object should not be made visible to other threads, nor should the `final` fields be read, until all updates to the `final` fields of the object are complete. Freezes of a `final` field occur both at the end of the constructor in which the `final` field is set, and immediately after each modification of a `final` field via reflection or other special mechanism.

Even then, there are a number of complications. If a `final` field is initialized to a constant expression (§15.28) in the field declaration, changes to the `final` field may not be observed, since uses of that `final` field are replaced at compile time with the value of the constant expression.

Another problem is that the specification allows aggressive optimization of `final` fields. Within a thread, it is permissible to reorder reads of a `final` field with those modifications of a `final` field that do not take place in the constructor.

**Example 17.5.3-1. Aggressive Optimization of `final` Fields**

```java
class A {
    final int x;
    A() {
        x = 1;
    }

    int f() {
        return d(this, this);
    }

    int d(A a1, A a2) {
        int i = a1.x;
        g(a1);
        int j = a2.x;
        return j - i;
    }

    static void g(A a) {
        // uses reflection to change a.x to 2
    }
}
```
In the $d$ method, the compiler is allowed to reorder the reads of $x$ and the call to $g$ freely. Thus, new $A().f()$ could return $-1$, $0$, or $1$.

An implementation may provide a way to execute a block of code in a final-field-safe context. If an object is constructed within a final-field-safe context, the reads of a final field of that object will not be reordered with modifications of that final field that occur within that final-field-safe context.

A final-field-safe context has additional protections. If a thread has seen an incorrectly published reference to an object that allows the thread to see the default value of a final field, and then, within a final-field-safe context, reads a properly published reference to the object, it will be guaranteed to see the correct value of the final field. In the formalism, code executed within a final-field-safe context is treated as a separate thread (for the purposes of final field semantics only).

In an implementation, a compiler should not move an access to a final field into or out of a final-field-safe context (although it can be moved around the execution of such a context, so long as the object is not constructed within that context).

One place where use of a final-field-safe context would be appropriate is in an executor or thread pool. By executing each Runnable in a separate final-field-safe context, the executor could guarantee that incorrect access by one Runnable to an object $o$ will not remove final field guarantees for other Runnables handled by the same executor.

### 17.5.4 Write-Protected Fields

Normally, a field that is final and static may not be modified. However, System.in, System.out, and System.err are static final fields that, for legacy reasons, must be allowed to be changed by the methods System.setIn, System.setOut, and System.setErr. We refer to these fields as being write-protected to distinguish them from ordinary final fields.

The compiler needs to treat these fields differently from other final fields. For example, a read of an ordinary final field is "immune" to synchronization: the barrier involved in a lock or volatile read does not have to affect what value is read from a final field. Since the value of write-protected fields may be seen to change, synchronization events should have an effect on them. Therefore, the semantics dictate that these fields be treated as normal fields that cannot be changed by user code, unless that user code is in the System class.
17.6 Word Tearing

One consideration for implementations of the Java Virtual Machine is that every field and array element is considered distinct; updates to one field or element must not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array separately must not interfere or interact and do not need synchronization to ensure sequential consistency.

Some processors do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as word tearing, and on processors that cannot easily update a single byte in isolation some other approach will be required.

Example 17.6-1. Detection of Word Tearing

The following program is a test case to detect word tearing:

```java
public class WordTearing extends Thread {
    static final int LENGTH = 8;
    static final int ITERS = 1000000;
    static byte[] counts = new byte[LENGTH];
    static Thread[] threads = new Thread[LENGTH];

    final int id;
    WordTearing(int i) {
        id = i;
    }

    public void run() {
        byte v = 0;
        for (int i = 0; i < ITERS; i++) {
            byte v2 = counts[id];
            if (v != v2) {
                System.err.println("Word-Tearing found: "+
                        "counts[" + id + "] = " + v2 +
                        ", should be " + v);
                return;
            }
            v++;
            counts[id] = v;
        }
    }
}

public static void main(String[] args) {
    for (int i = 0; i < LENGTH; ++i)
        (threads[i] = new WordTearing(i)).start();
}
```
This makes the point that bytes must not be overwritten by writes to adjacent bytes.

17.7 Non-Atomic Treatment of double and long

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid possible complications.
A variety of compile-time analyses require reasoning about types that are not yet known. Principal among these are generic method applicability testing (§18.5.1) and generic method invocation type inference (§18.5.2). In general, we refer to the process of reasoning about unknown types as type inference.

At a high level, type inference can be decomposed into three processes:

- **Reduction** takes a compatibility assertion about an expression or type, called a constraint formula, and reduces it to a set of bounds on inference variables. Often, a constraint formula reduces to other constraint formulas, which must be recursively reduced. A procedure is followed to identify these additional constraint formulas and, ultimately, to express via a bound set the conditions under which the choices for inferred types would render each constraint formula true.

- **Incorporation** maintains a set of inference variable bounds, ensuring that these are consistent as new bounds are added. Because the bounds on one variable can sometimes impact the possible choices for another variable, this process propagates bounds between such interdependent variables.

- **Resolution** examines the bounds on an inference variable and determines an instantiation that is compatible with those bounds. It also decides the order in which interdependent inference variables are to be resolved.

These processes interact closely: reduction can trigger incorporation; incorporation may lead to further reduction; and resolution may cause further incorporation.

- §18.1 more precisely defines the concepts used as intermediate results and the notation used to express them.
- §18.2 describes reduction in detail.
- §18.3 describes incorporation in detail.
§18.4 describes resolution in detail.

§18.5 defines how these inference tools are used to solve certain compile-time analysis problems.

In comparison to the Java SE 7 Edition of *The Java® Language Specification*, important changes to inference include:

- Adding support for lambda expressions and method references as method invocation arguments.
- Generalizing to define inference in terms of poly expressions, which may not have well-defined types until *after* inference is complete. This has the notable effect of improving inference for nested generic method and diamond constructor invocations.
- Describing how inference is used to handle wildcard-parameterized functional interface target types and most specific method analysis.
- Clarifying the distinction between invocation applicability testing (which involves only the invocation arguments) and invocation type inference (which incorporates a target type).
- Delaying resolution of all inference variables, even those with lower bounds, until invocation type inference, in order to get better results.
- Improving inference behavior for interdependent (or self-dependent) variables.
- Eliminating bugs and potential sources of confusion. This revision more carefully and precisely handles the distinction between specific conversion contexts and subtyping, and describes reduction by paralleling the corresponding non-inference relations. Where there are intentional departures from the non-inference relations, these are explicitly identified as such.
- Laying a foundation for future evolution: enhancements to or new applications of inference will be easier to integrate into the specification.

### 18.1 Concepts and Notation

This section defines *inference variables*, *constraint formulas*, and *bounds*, as the terms will be used throughout this chapter. It also presents notation.

#### 18.1.1 Inference Variables

*Inference variables* are *meta-variables* for types - that is, they are special names that allow abstract reasoning about types. To distinguish them from *type variables*, inference variables are represented with Greek letters, principally $\alpha$.

The term "type" is used loosely in this chapter to include type-like syntax that contains inference variables. The term *proper type* excludes such "types"
that mention inference variables. Assertions that involve inference variables are assertions about every proper type that can be produced by replacing each inference variable with a proper type.

### 18.1.2 Constraint Formulas

*Constraint formulas* are assertions of compatibility or subtyping that may involve inference variables. The formulas may take one of the following forms:

- \(<\text{Expression} \rightarrow \tau>\): An expression is compatible in a loose invocation context with type \(\tau\) (§5.3).
- \(<\alpha \rightarrow \tau>\): A type \(\alpha\) is compatible in a loose invocation context with type \(\tau\) (§5.3).
- \(<\alpha < \tau>\): A reference type \(\alpha\) is a subtype of a reference type \(\tau\) (§4.10).
- \(<\alpha <= \tau>\): A type argument \(\alpha\) is contained by a type argument \(\tau\) (§4.5.1).
- \(<\alpha = \tau>\): A type \(\alpha\) is the same as a type \(\tau\) (§4.3.4), or a type argument \(\alpha\) is the same as type argument \(\tau\).
- \(<\text{LambdaExpression} \rightarrow \text{throws} \tau>\): The checked exceptions thrown by the body of the LambdaExpression are declared by the throws clause of the function type derived from \(\tau\).
- \(<\text{MethodReference} \rightarrow \text{throws} \tau>\): The checked exceptions thrown by the referenced method are declared by the throws clause of the function type derived from \(\tau\).

Examples of constraint formulas:

- From \(\text{Collections.singleton("hi")}\), we have the constraint formula \(<\text{"hi"} \rightarrow \alpha>\). Through reduction, this will become the constraint formula: \(<\text{String} < \alpha>\).
- From \(\text{Arrays.asList(1, 2.0)}\), we have the constraint formulas \(<1 \rightarrow \alpha>\) and \(<2.0 \rightarrow \alpha>\). Through reduction, these will become the constraint formulas \(<\text{int} \rightarrow \alpha>\) and \(<\text{double} \rightarrow \alpha>\), and then \(<\text{Integer} < \alpha>\) and \(<\text{Double} < \alpha>\).
- From the target type of the constructor invocation \(\text{List<Thread> lt = new ArrayList<>()}\), we have the constraint formula \(<\text{List<\alpha>} \rightarrow \text{List<Thread>}>\). Through reduction, this will become the constraint formula \(<\alpha <= \text{Thread}>\), and then \(<\alpha = \text{Thread}>\).

### 18.1.3 Bounds

During the inference process, a set of *bounds* on inference variables is maintained. A bound has one of the following forms:

- \(\alpha = \tau\), where at least one of \(\alpha\) or \(\tau\) is an inference variable: \(\alpha\) is the same as \(\tau\).
• $S <: T$, where at least one of $S$ or $T$ is an inference variable: $S$ is a subtype of $T$.

• $false$: No valid choice of inference variables exists.

• $G<\alpha_1, ..., \alpha_n> = capture(G<A_1, ..., A_n>)$: The variables $\alpha_1, ..., \alpha_n$ represent the result of capture conversion (§5.1.10) applied to $G<A_1, ..., A_n>$ (where $A_1, ..., A_n$ may be types or wildcards and may mention inference variables).

• $throws \alpha$: The inference variable $\alpha$ appears in a throws clause.

A bound is satisfied by an inference variable substitution if, after applying the substitution, the assertion is true. The bound $false$ can never be satisfied.

Some bounds relate an inference variable to a proper type. Let $T$ be a proper type. Given a bound of the form $\alpha = T$ or $T = \alpha$, we say $T$ is an instantiation of $\alpha$. Similarly, given a bound of the form $\alpha <: T$, we say $T$ is a proper upper bound of $\alpha$, and given a bound of the form $T <: \alpha$, we say $T$ is a proper lower bound of $\alpha$.

Other bounds relate two inference variables, or an inference variable to a type that contains inference variables. Such bounds, of the form $S = T$ or $S <: T$, are called dependencies.

A bound of the form $G<\alpha_1, ..., \alpha_n> = capture(G<A_1, ..., A_n>)$ indicates that $\alpha_1, ..., \alpha_n$ are placeholders for the results of capture conversion. This is necessary because capture conversion can only be performed on a proper type, and the inference variables in $A_1, ..., A_n$ may not yet be resolved.

A bound of the form $throws \alpha$ is purely informational: it directs resolution to optimize the instantiation of $\alpha$ so that, if possible, it is not a checked exception type.

An important intermediate result of inference is a bound set. It is sometimes convenient to refer to an empty bound set with the symbol $true$; this is merely out of convenience, and the two are interchangeable.

Examples of bound sets:

• $\{ \alpha = String \}$ contains a single bound, instantiating $\alpha$ as $String$.

• $\{ Integer <: \alpha, Double <: \alpha, Object \}$ describes two proper lower bounds and one proper upper bound for $\alpha$.

• $\{ \alpha <: Iterable<?>, \beta <: Object, \alpha <: List<\beta> \}$ describes a proper upper bound for each of $\alpha$ and $\beta$, along with a dependency between them.

• $\{ \}$ contains no bounds nor dependencies, and can be referred to as $true$.

• $\{ false \}$ expresses the fact that no satisfactory instantiation exists.
When inference begins, a bound set is typically generated from a list of type parameter declarations $P_1, ..., P_p$ and associated inference variables $\alpha_1, ..., \alpha_p$. Such a bound set is generated as follows. For each $l$ ($1 \leq l \leq p$):

- If $P_l$ has no TypeBound, the bound $\alpha_l <: \text{Object}$ appears in the set.
- Otherwise, for each type $\tau$ delimited by $\&$ in the TypeBound, the bound $\alpha_l <: \tau[P_1:=\alpha_1, ..., P_p:=\alpha_p]$ appears in the set; if this results in no proper upper bounds for $\alpha_l$ (only dependencies), then the bound $\alpha_l <: \text{Object}$ also appears in the set.

### 18.2 Reduction

Reduction is the process by which a set of constraint formulas (§18.1.2) is simplified to produce a bound set (§18.1.3).

Each constraint formula is considered in turn. The rules in this section specify how the formula is reduced to one or both of:

- A bound or bound set, which is to be incorporated with the "current" bound set. Initially, the current bound set is empty.
- Further constraint formulas, which are to be reduced recursively.

Reduction completes when no further constraint formulas remain to be reduced.

The results of a reduction step are always soundness-preserving: if an inference variable instantiation satisfies the reduced constraints and bounds, it will also satisfy the original constraint. On the other hand, reduction is not completeness-preserving: there may exist inference variable instantiations that satisfy the original constraint but do not satisfy a reduced constraint or bound. This is due to inherent limitations of the algorithm, along with a desire to avoid undue complexity. One effect is that there are expressions for which type argument inference fails to find a solution, but that can be well-typed if the programmer explicitly inserts appropriate types.

### 18.2.1 Expression Compatibility Constraints

A constraint formula of the form $\langle \text{Expression} \rightarrow \tau \rangle$ is reduced as follows:

- If $\tau$ is a proper type, the constraint reduces to $\text{true}$ if the expression is compatible in a loose invocation context with $\tau$ (§5.3), and $\text{false}$ otherwise.
- Otherwise, if the expression is a standalone expression (§15.2) of type $s$, the constraint reduces to $\langle s \rightarrow \tau \rangle$.
- Otherwise, the expression is a poly expression (§15.2). The result depends on the form of the expression:
If the expression is a parenthesized expression of the form `(Expression')`, the constraint reduces to `<Expression' → T>`.

If the expression is a class instance creation expression or a method invocation expression, the constraint reduces to the bound set $B_3$ which would be used to determine the expression's compatibility with target type $T$, as defined in §18.5.2.1. (For a class instance creation expression, the corresponding "method" used for inference is defined in §15.9.3.)

This bound set may contain new inference variables, as well as dependencies between these new variables and the inference variables in $T$.

If the expression is a conditional expression of the form $e_1 ? e_2 : e_3$, the constraint reduces to two constraint formulas, $<e_2 → T>$ and $<e_3 → T>$.

If the expression is a lambda expression or a method reference expression, the result is specified below.

By treating nested generic method invocations as poly expressions, we improve the behavior of inference for nested invocations. For example, the following is illegal in Java SE 7 but legal in Java SE 8:

```java
ProcessBuilder b = new ProcessBuilder(Collections.emptyList());  // ProcessBuilder's constructor expects a List<String>
```

When both the outer and the nested invocation require inference, the problem is more difficult. For example:

```java
List<String> ls = new ArrayList<>(Collections.emptyList());
```

Our approach is to "lift" the bounds inferred for the nested invocation (simply `α <: Object` in the case of `emptyList`) into the outer inference process (in this case, trying to infer $β$ where the constructor is for type `ArrayList<β>`). We also infer dependencies between the nested inference variables and the outer inference variables (the constraint $<\text{List}<α> → \text{Collection}<β> >$ would reduce to the dependency $α = β$). In this way, resolution of the inference variables in the nested invocation can wait until additional information can be inferred from the outer invocation (based on the assignment target, $β = String$).

A constraint formula of the form `<LambdaExpression → T>`, where $T$ mentions at least one inference variable, is reduced as follows:

- If $T$ is not a functional interface type (§9.8), the constraint reduces to `false`.
- Otherwise, let $T'$ be the ground target type derived from $T$, as specified in §15.27.3. If §18.5.3 is used to derive a functional interface type which is
parameterized, then the test that $F^{<A_1', ..., A_m'>}$ is a subtype of $F^{<A_1, ..., A_m>}$ is not performed (instead, it is asserted with a constraint formula below). Let the target function type for the lambda expression be the function type of $T'$. Then:

– If no valid function type can be found, the constraint reduces to \textit{false}.

– Otherwise, the congruence of \textit{LambdaExpression} with the target function type is asserted as follows:

  > If the number of lambda parameters differs from the number of parameter types of the function type, the constraint reduces to \textit{false}.

  > If the lambda expression is implicitly typed and one or more of the function type’s parameter types is not a proper type, the constraint reduces to \textit{false}.

  This condition never arises in practice, due to the handling of implicitly typed lambda expressions in §18.5.1 and the substitution applied to the target type in §18.5.2.2.

  > If the function type’s result is \textit{void} and the lambda body is neither a statement expression nor a \textit{void}-compatible block, the constraint reduces to \textit{false}.

  > If the function type’s result is not \textit{void} and the lambda body is a block that is not value-compatible, the constraint reduces to \textit{false}.

  > Otherwise, the constraint reduces to all of the following constraint formulas:

    » If the lambda parameters have explicitly declared types $F_1, ..., F_n$ and the function type has parameter types $G_1, ..., G_n$, then (i) for all $i$ ($1 \leq i \leq n$), $F_i = G_i$, and (ii) $\langle T' \prec: T \rangle$.

    » If the function type’s return type is a (non-\textit{void}) type $R$, assume the lambda’s parameter types are the same as the function type’s parameter types. Then:

      • If $R$ is a proper type, and if the lambda body or some result expression in the lambda body is not compatible in an assignment context with $R$, then \textit{false}.

      • Otherwise, if $R$ is not a proper type, then where the lambda body has the form \textit{Expression}, the constraint $<\langle Expression \rightarrow R \rangle$; or where the lambda body is a block with result expressions $e_1, ..., e_m$, for all $i$ ($1 \leq i \leq m$), $\langle e_i \rightarrow R \rangle$.

The key piece of information to derive from a compatibility constraint involving a lambda expression is the set of bounds on inference variables appearing in the target function
Reduction

The type's return type. This is crucial, because functional interfaces are often generic, and many methods operating on these types are generic, too.

In the simplest case, a lambda expression may simply provide a lower bound for an inference variable:

```java
<T> List<T> makeThree(Factory<T> factory) { ... }
String s = makeThree(() -> "abc").get(2);
```

In more complex cases, a result expression may be a poly expression - perhaps even another lambda expression - and so the inference variable might be passed through multiple constraint formulas with different target types before a bound is produced.

Most of the work described in this section precedes assertions about the result expressions; its purpose is to derive the lambda expression's function type, and to check for expressions that are clearly disqualified from compatibility.

We do not attempt to produce bounds on inference variables that appear in the target function type's throws clause. This is because exception containment is not part of compatibility (§15.27.3) - in particular, it must not influence method applicability (§18.5.1). However, we do get bounds on these variables later, because invocation type inference (§18.5.2.2) produces exception containment constraint formulas (§18.2.5).

Note that if the target type is an inference variable, or if the target type's parameter types contain inference variables, we produce false. During invocation type inference (§18.5.2.2), extra substitutions are performed in order to instantiate these inference variables, thus avoiding this scenario. (In other words, reduction will, in practice, never be "invoked" with a target type of one of these forms.)

Finally, note that the result expressions of a lambda expression are required by §15.27.3 to be compatible in an assignment context with the target type's return type, \( R \). If \( R \) is a proper type, such as Byte derived from Function<\( \alpha \), Byte>, then assignability is easy enough to test, and reduction does so above. If \( R \) is not a proper type, such as \( \alpha \) derived from Function<String, \( \alpha \)>, then we make the simplifying assumption above that loose invocation compatibility will be sufficient. The difference between assignment compatibility and loose invocation compatibility is that only assignment allows narrowing of constant expressions, such as Byte \( b = 100; \). Consequently, our simplifying assumption is not completeness-preserving: given target return type \( \alpha \) and an integer literal result expression 100, it is conceivable that \( \alpha \) could be instantiated to Byte, but reduction will not in fact produce such a bound.

A constraint formula of the form \(<\text{MethodReference} \rightarrow \tau>\), where \( \tau \) mentions at least one inference variable, is reduced as follows:

- If \( \tau \) is not a functional interface type, or if \( \tau \) is a functional interface type that does not have a function type (§9.9), the constraint reduces to false.
- Otherwise, if there does not exist a potentially applicable method for the method reference when targeting \( \tau \), the constraint reduces to false.
• Otherwise, if the method reference is exact (§15.13.1), then let $P_1, ..., P_n$ be the parameter types of the function type of $T$, and let $F_1, ..., F_k$ be the parameter types of the potentially applicable method. The constraint reduces to a new set of constraints, as follows:

  – In the special case where $n = k+1$, the parameter of type $P_1$ is to act as the target reference of the invocation. The method reference expression necessarily has the form $\text{ReferenceType :: [TypeArguments] Identifier}$. The constraint reduces to $\langle P_1 <: \text{ReferenceType}\rangle$ and, for all $i \ (2 \leq i \leq n)$, $\langle P_i \to F_{i-1}\rangle$.

    In all other cases, $n = k$, and the constraint reduces to, for all $i \ (1 \leq i \leq n)$, $\langle P_i \to F_i\rangle$.

  – If the function type's result is not $\text{void}$, let $R$ be its return type. Then, if the result of the potentially applicable compile-time declaration is $\text{void}$, the constraint reduces to $\text{false}$. Otherwise, the constraint reduces to $\langle R' \to R\rangle$, where $R'$ is the result of applying capture conversion (§5.1.10) to the return type of the potentially applicable compile-time declaration.

• Otherwise, the method reference is inexact, and:

  – If one or more of the function type's parameter types is not a proper type, the constraint reduces to $\text{false}$.

This condition never arises in practice, due to the handling of inexact method references in §18.5.1 and the substitution applied to the target type in §18.5.2.2.

  – Otherwise, a search for a compile-time declaration is performed, as specified in §15.13.1. If there is no compile-time declaration for the method reference, the constraint reduces to $\text{false}$. Otherwise, there is a compile-time declaration, and: (let $R$ be the result of the function type)

    » If $R$ is $\text{void}$, the constraint reduces to $\text{true}$.

    » Otherwise, if the method reference expression elides $\text{TypeArguments}$, and the compile-time declaration is a generic method, and the return type of the compile-time declaration mentions at least one of the method's type parameters, then:

      » If $R$ mentions one of the type parameters of the function type, the constraint reduces to $\text{false}$.

In this case, a constraint in terms of $R$ might lead an inference variable to be bound by an out-of-scope type variable. Since instantiating an inference variable with an out-of-scope type variable is nonsensical, we prefer to avoid the situation by giving up immediately whenever the possibility arises. This simplification is not completeness-preserving.
If \( R \) does not mention one of the type parameters of the function type, then the constraint reduces to the bound set \( B_3 \) which would be used to determine the method reference's compatibility when targeting the return type of the function type, as defined in §18.5.2.1. \( B_3 \) may contain new inference variables, as well as dependencies between these new variables and the inference variables in \( T \).

The strategy used to determine a return type for a generic referenced method follows the pattern used earlier in this section for generic method invocations. This may involve "lifting" bounds into the outer context and inferring dependencies between the two sets of inference variables.

Otherwise, let \( R' \) be the result of applying capture conversion (§5.1.10) to the return type of the invocation type (§15.12.2.6) of the compile-time declaration. If \( R' \) is \texttt{void}, the constraint reduces to \texttt{false}; otherwise, the constraint reduces to \( <R' \rightarrow R> \).

### 18.2.2 Type Compatibility Constraints

A constraint formula of the form \( <s \rightarrow t> \) is reduced as follows:

- If \( s \) and \( t \) are proper types, the constraint reduces to \textit{true} if \( s \) is compatible in a loose invocation context with \( t \) (§5.3), and \textit{false} otherwise.

- Otherwise, if \( s \) is a primitive type, let \( s' \) be the result of applying boxing conversion (§5.1.7) to \( s \). Then the constraint reduces to \( <s' \rightarrow t> \).

- Otherwise, if \( t \) is a primitive type, let \( t' \) be the result of applying boxing conversion (§5.1.7) to \( t \). Then the constraint reduces to \( <s = t'> \).

- Otherwise, if \( t \) is a parameterized type of the form \( \mathcal{G}\langle T_1, ..., T_n \rangle \), and there exists no type of the form \( \mathcal{G}\langle...\rangle \) that is a supertype of \( s \), but the raw type \( \mathcal{G} \) is a supertype of \( s \), then the constraint reduces to \textit{true}.

- Otherwise, if \( t \) is an array type of the form \( \mathcal{G}\langle T_1, ..., T_n \rangle [\cdot]^k \), and there exists no type of the form \( \mathcal{G}\langle...\rangle[\cdot]^k \) that is a supertype of \( s \), but the raw type \( \mathcal{G}[\cdot]^k \) is a supertype of \( s \), then the constraint reduces to \textit{true}. (The notation \( [\cdot]^k \) indicates an array type of \( k \) dimensions.)

- Otherwise, the constraint reduces to \( <s <: t> \).

The fourth and fifth cases are implicit uses of unchecked conversion (§5.1.9). These, along with any use of unchecked conversion in the first case, may result in compile-time unchecked warnings, and may influence a method's invocation type (§15.12.2.6).
Boxing $T$ to $T'$ is not completeness-preserving; for example, if $T$ were `long`, $S$ might be instantiated to `Integer`, which is not a subtype of `Long` but could be unboxed and then widened to `long`. We avoid this problem in most cases by giving special treatment to inference-variable return types that we know are already constrained to be certain boxed primitive types; see §18.5.2.1.

Similarly, the treatment of unchecked conversion sacrifices completeness in cases in which $T$ is not a parameterized type (for example, if $T$ is an inference variable). It is not usually clear in such situations whether the unchecked conversion is necessary or not. Since unchecked conversions introduce unchecked warnings, inference prefers to avoid them unless it is clearly necessary.

### 18.2.3 Subtyping Constraints

A constraint formula of the form $\langle S <: T \rangle$ is reduced as follows:

- If $S$ and $T$ are proper types, the constraint reduces to `true` if $S$ is a subtype of $T$ (§4.10), and `false` otherwise.
- Otherwise, if $S$ is the null type, the constraint reduces to `true`.
- Otherwise, if $T$ is the null type, the constraint reduces to `false`.
- Otherwise, if $S$ is an inference variable, $\alpha$, the constraint reduces to the bound $\alpha <: T$.
- Otherwise, if $T$ is an inference variable, $\alpha$, the constraint reduces to the bound $S <: \alpha$.
- Otherwise, the constraint is reduced according to the form of $T$:
  - If $T$ is a parameterized class or interface type, or an inner class type of a parameterized class or interface type (directly or indirectly), let $A_1, ..., A_n$ be the type arguments of $T$. Among the supertypes of $S$, a corresponding class or interface type is identified, with type arguments $B_1, ..., B_n$. If no such type exists, the constraint reduces to `false`. Otherwise, the constraint reduces to the following new constraints: for all $i \ (1 \leq i \leq n)$, $\langle B_i <= A_i \rangle$.
  - If $T$ is any other class or interface type, then the constraint reduces to `true` if $T$ is among the supertypes of $S$, and `false` otherwise.
  - If $T$ is an array type, $T'[]$, then among the supertypes of $S$ that are array types, a most specific type is identified, $S'[]$ (this may be $S$ itself). If no such array type exists, the constraint reduces to `false`. Otherwise:
    - If neither $S'$ nor $T'$ is a primitive type, the constraint reduces to $\langle S' <: T' \rangle$.
    - Otherwise, the constraint reduces to `true` if $S'$ and $T'$ are the same primitive type, and `false` otherwise.
18.2 Reduction

- If $T$ is a type variable, there are three cases:
  - If $S$ is an intersection type of which $T$ is an element, the constraint reduces to $true$.
  - Otherwise, if $T$ has a lower bound, $B$, the constraint reduces to $S <: B$.
  - Otherwise, the constraint reduces to $false$.

- If $T$ is an intersection type, $I_1 \& ... \& I_n$, the constraint reduces to the following new constraints: for all $i$ ($1 \leq i \leq n$), $S <: I_i$.

A constraint formula of the form $<S <= T>$, where $S$ and $T$ are type arguments (§4.5.1), is reduced as follows:

- If $T$ is a type:
  - If $S$ is a type, the constraint reduces to $S = T$.
  - If $S$ is a wildcard, the constraint reduces to $false$.

- If $T$ is a wildcard of the form $?$, the constraint reduces to $true$.

- If $T$ is a wildcard of the form $? \text{ extends } T'$:
  - If $S$ is a type, the constraint reduces to $S <: T'$.
  - If $S$ is a wildcard of the form $?$, the constraint reduces to $\text{Object } <: T'$.
  - If $S$ is a wildcard of the form $? \text{ extends } S'$, the constraint reduces to $S' <: T'$.
  - If $S$ is a wildcard of the form $? \text{ super } S'$, the constraint reduces to $\text{Object } = T'$.

- If $T$ is a wildcard of the form $? \text{ super } T'$:
  - If $S$ is a type, the constraint reduces to $T' <: S$.
  - If $S$ is a wildcard of the form $? \text{ super } S'$, the constraint reduces to $T' <: S'$.
  - Otherwise, the constraint reduces to $false$.

18.2.4 Type Equality Constraints

A constraint formula of the form $<S = T>$, where $S$ and $T$ are types, is reduced as follows:

- If $S$ and $T$ are proper types, the constraint reduces to $true$ if $S$ is the same as $T$ (§4.3.4), and $false$ otherwise.
- Otherwise, if $S$ or $T$ is the null type, the constraint reduces to $false$. 
• Otherwise, if \( S \) is an inference variable, \( \alpha \), and \( T \) is not a primitive type, the constraint reduces to the bound \( \alpha = T \).

• Otherwise, if \( T \) is an inference variable, \( \alpha \), and \( S \) is not a primitive type, the constraint reduces to the bound \( S = \alpha \).

• Otherwise, if \( S \) and \( T \) are class or interface types with the same erasure, where \( S \) has type arguments \( B_1, ..., B_n \) and \( T \) has type arguments \( A_1, ..., A_n \), the constraint reduces to the following new constraints: for all \( i \ (1 \leq i \leq n) \), \( \langle B_i = A_i \rangle \).

• Otherwise, if \( S \) and \( T \) are array types, \( S'[] \) and \( T'[] \), the constraint reduces to \( \langle S' = T' \rangle \).

• Otherwise, if \( S \) and \( T \) are intersection types, a correspondence between the elements of \( S \) and the elements of \( T \) is established. An element of \( S \), \( S_i \), corresponds to an element of \( T \), \( T_j \), if \( S_i \) and \( T_j \) are either the same type, or both parameterizations of the same generic class or interface, or both array types.

If each element of \( S \) corresponds to exactly one element of \( T \), and vice versa, then the constraint reduces to the following new constraints: for each element \( S_i \) of \( S \) and the corresponding element \( T_j \) of \( T \), \( \langle S_i = T_j \rangle \). If not, the constraint reduces to \text{false}.

This rule does not accommodate inference variables appearing directly as elements of an intersection type (rather than nested in a parameterized type). Due to the restrictions on type parameter declarations (§4.4), such intersection types do not arise in practice.

• Otherwise, the constraint reduces to \text{false}.

A constraint formula of the form \( \langle S = T \rangle \), where \( S \) and \( T \) are type arguments (§4.5.1), is reduced as follows:

• If \( S \) and \( T \) are types, the constraint is reduced as described above.

• If \( S \) has the form \( ? \) and \( T \) has the form \( ? \), the constraint reduces to \text{true}.

• If \( S \) has the form \( ? \) and \( T \) has the form \( ? \ extends \ T' \), the constraint reduces to \( \langle \text{Object} = T' \rangle \).

• If \( S \) has the form \( ? \ extends \ S' \) and \( T \) has the form \( ? \), the constraint reduces to \( \langle S' = \text{Object} \rangle \).

• If \( S \) has the form \( ? \ extends \ S' \) and \( T \) has the form \( ? \ extends \ T' \), the constraint reduces to \( \langle S' = T' \rangle \).

• If \( S \) has the form \( ? \ super \ S' \) and \( T \) has the form \( ? \ super \ T' \), the constraint reduces to \( \langle S' = T' \rangle \).

• Otherwise, the constraint reduces to \text{false}.
18.2.5 Checked Exception Constraints

A constraint formula of the form $\langle \text{LambdaExpression} \rightarrow_{\text{throws}} T \rangle$ is reduced as follows:

- If $T$ is not a functional interface type ($\S$9.8), the constraint reduces to $false$.
- Otherwise, let the target function type for the lambda expression be determined as specified in $\S$15.27.3. If no valid function type can be found, the constraint reduces to $false$.
- Otherwise, if the lambda expression is implicitly typed, and one or more of the function type’s parameter types is not a proper type, the constraint reduces to $false$.

This condition never arises in practice, due to the substitution applied to the target type in $\S$18.5.2.2.

- Otherwise, if the function type’s return type is neither $\text{void}$ nor a proper type, the constraint reduces to $false$.

This condition never arises in practice, due to the substitution applied to the target type in $\S$18.5.2.2.

- Otherwise, let $E_1, \ldots, E_n$ be the types in the function type’s $\text{throws}$ clause that are not proper types. If the lambda expression is implicitly typed, let its parameter types be the function type’s parameter types. If the lambda body is a poly expression or a block containing a poly result expression, let the targeted return type be the function type’s return type. Let $x_1, \ldots, x_m$ be the checked exception types that the lambda body can throw ($\S$11.2). Then there are two cases:

  - If $n = 0$ (the function type’s $\text{throws}$ clause consists only of proper types), then if there exists some $i$ ($1 \leq i \leq m$) such that $x_i$ is not a subtype of any proper type in the $\text{throws}$ clause, the constraint reduces to $false$; otherwise, the constraint reduces to $true$.

  - If $n > 0$, the constraint reduces to a set of subtyping constraints: for all $i$ ($1 \leq i \leq m$), if $x_i$ is not a subtype of any proper type in the $\text{throws}$ clause, then the constraints include, for all $j$ ($1 \leq j \leq n$), $\langle x_i <: E_j \rangle$. In addition, for all $j$ ($1 \leq j \leq n$), the constraint reduces to the bound $\text{throws } E_j$.

A constraint formula of the form $\langle \text{MethodReference} \rightarrow_{\text{throws}} T \rangle$ is reduced as follows:

- If $T$ is not a functional interface type, or if $T$ is a functional interface type but does not have a function type ($\S$9.9), the constraint reduces to $false$. 
• Otherwise, let the target function type for the method reference expression be the function type of \( \tau \). If the method reference is inexact (§15.13.1) and one or more of the function type's parameter types is not a proper type, the constraint reduces to \( \text{false} \).

• Otherwise, if the method reference is inexact and the function type's result is neither \( \text{void} \) nor a proper type, the constraint reduces to \( \text{false} \).

• Otherwise, let \( E_1, \ldots, E_n \) be the types in the function type's \( \text{throws} \) clause that are \( \text{not} \) proper types. Let \( X_1, \ldots, X_m \) be the checked exceptions in the \( \text{throws} \) clause of the invocation type of the method reference's compile-time declaration (§15.13.2) (as derived from the function type's parameter types and return type). Then there are two cases:
  
  – If \( n = 0 \) (the function type's \( \text{throws} \) clause consists only of proper types), then if there exists some \( i \) \((1 \leq i \leq m)\) such that \( X_i \) is not a subtype of any proper type in the \( \text{throws} \) clause, the constraint reduces to \( \text{false} \); otherwise, the constraint reduces to \( \text{true} \).
  
  – If \( n > 0 \), the constraint reduces to a set of subtyping constraints: for all \( i \) \((1 \leq i \leq m)\), if \( X_i \) is not a subtype of any proper type in the \( \text{throws} \) clause, then the constraints include, for all \( j \) \((1 \leq j \leq n)\), \( \langle X_i \triangleleft E_j \rangle \). In addition, for all \( j \) \((1 \leq j \leq n)\), the constraint reduces to the bound \( \text{throws} E_j \).

Constraints on checked exceptions are handled separately from constraints on return types, because return type compatibility influences applicability of methods (§18.5.1), while exceptions only influence the invocation type after overload resolution is complete (§18.5.2). This could be simplified by including exception compatibility in the definition of lambda expression compatibility (§15.27.3), but this would lead to possibly surprising cases in which exceptions that can be thrown by an explicitly typed lambda body change overload resolution.

The exceptions thrown by a lambda body cannot be determined until (i) the parameter types of the lambda are known, and (ii) the target type of result expressions in the body is known. (The second requirement is to account for generic method invocations in which, for example, the same type parameter appears in the return type and the \( \text{throws} \) clause.) Hence, we require both of these, as derived from the target type \( \tau \), to be proper types.

One consequence is that lambda expressions returned from \( \text{other} \) lambda expressions cannot generate constraints from their thrown exceptions. These constraints can only be generated from top-level lambda expressions.

Note that the handling of the case in which more than one inference variable appears in a function type's \( \text{throws} \) clause is not completeness-preserving. Either variable may, on its own, satisfy the constraint that each checked exception be declared, but we cannot be sure which one is intended. So, for predictability, we constrain them both.
18.3 Incorporation

As bound sets are generated and grown during inference, it is possible that new bounds can be inferred based on the assertions of the original bounds. The process of incorporation identifies these new bounds and adds them to the bound set.

Incorporation can happen in two scenarios. One scenario is that the bound set contains complementary pairs of bounds; this implies new constraint formulas, as specified in §18.3.1. The other scenario is that the bound set contains a bound involving capture conversion; this implies new bounds and may imply new constraint formulas, as specified in §18.3.2. In both scenarios, any new constraint formulas are reduced, and any new bounds are added to the bound set. This may trigger further incorporation; ultimately, the set will reach a fixed point and no further bounds can be inferred.

If incorporation of a bound set has reached a fixed point, and the set does not contain the bound \textit{false}, then the bound set has the following properties:

- For each combination of a proper lower bound $L$ and a proper upper bound $U$ of an inference variable, $L <: U$.
- If every inference variable mentioned by a bound has an instantiation, the bound is satisfied by the corresponding substitution.
- Given a dependency $\alpha = \beta$, every bound of $\alpha$ matches a bound of $\beta$, and vice versa.
- Given a dependency $\alpha <: \beta$, every lower bound of $\alpha$ is a lower bound of $\beta$, and every upper bound of $\beta$ is an upper bound of $\alpha$.

The assertion that incorporation reaches a fixed point oversimplifies the matter slightly. Building on the work of Kennedy and Pierce, \textit{On Decidability of Nominal Subtyping with Variance}, this property can be proven by making the argument that the set of types that may appear in the bound set is finite. The argument relies on two assumptions:

- New capture variables are not generated when reducing subtyping constraints (§18.2.3).
- Expansive inheritance paths are not pursued.

This specification does not currently guarantee these properties (it is imprecise about the handling of wildcards when reducing subtyping constraints, and does not detect expansive inheritance paths), but may do so in a future version. (This is not a new problem: the Java subtyping algorithm is also at risk of non-termination.)
18.3.1 Complementary Pairs of Bounds

(In this section, \(s\) and \(t\) are inference variables or types, and \(u\) is a proper type. For conciseness, a bound of the form \(\alpha = t\) may also match a bound of the form \(t = \alpha\).)

When a bound set contains a pair of bounds that match one of the following rules, a new constraint formula is implied:

- \(\alpha = s\) and \(\alpha = t\) imply \(<s = t>\)
- \(\alpha = s\) and \(\alpha <: t\) imply \(<s <: t>\)
- \(\alpha = s\) and \(t <: \alpha\) imply \(<t <: s>\)
- \(s <: \alpha\) and \(\alpha <: t\) imply \(<s <: t>\)
- \(\alpha = u\) and \(s = t\) imply \(<s [\alpha := u] = t [\alpha := u]>\)
- \(\alpha = u\) and \(s <: t\) imply \(<s [\alpha := u] <: t [\alpha := u]>\)

When a bound set contains a pair of bounds \(\alpha <: s\) and \(\alpha <: t\), and there exists a supertype of \(s\) of the form \(g<s_1, ..., s_n>\) and a supertype of \(t\) of the form \(g<t_1, ..., t_n>\) (for some generic class or interface, \(g\)), then for all \(i\) (\(1 \leq i \leq n\)), if \(s_i\) and \(t_i\) are types (not wildcards), the constraint formula \(<s_i = t_i>\) is implied.

18.3.2 Bounds Involving Capture Conversion

When a bound set contains a bound of the form \(g<\alpha_1, ..., \alpha_n> = \text{capture}(g<A_1, ..., A_n>)\), new bounds are implied and new constraint formulas may be implied, as follows.

Let \(p_1, ..., p_n\) represent the type parameters of \(g\) and let \(b_1, ..., b_n\) represent the bounds of these type parameters. Let \(\theta\) represent the substitution \([p_1 := \alpha_1, ..., p_n := \alpha_n]\). Let \(r\) be a type that is not an inference variable (but is not necessarily a proper type).

A set of bounds on \(\alpha_1, ..., \alpha_n\) is implied, generated from the declared bounds of \(p_1, ..., p_n\) as specified in §18.1.3.

In addition, for all \(i\) (\(1 \leq i \leq n\)):

- If \(A_i\) is not a wildcard, then the bound \(\alpha_i = A_i\) is implied.
- If \(A_i\) is a wildcard of the form \(?\):
  - \(\alpha_i = r\) implies the bound \(false\)
  - \(\alpha_i <: r\) implies the constraint formula \(<b_i \theta <: r>\)
  - \(r <: \alpha_i\) implies the bound \(false\)
• If $A_i$ is a wildcard of the form $\_{}$ extends $T$:
  – $\alpha_i = R$ implies the bound $false$
  – If $B_i$ is Object, then $\alpha_i <: R$ implies the constraint formula $\langle T <: R \rangle$
  – If $T$ is Object, then $\alpha_i <: R$ implies the constraint formula $\langle B_i \theta <: R \rangle$
  – $R <: \alpha_i$ implies the bound $false$

• If $A_i$ is a wildcard of the form $\_{}$ super $T$:
  – $\alpha_i = R$ implies the bound $false$
  – $\alpha_i <: R$ implies the constraint formula $\langle B_i \theta <: R \rangle$
  – $R <: \alpha_i$ implies the constraint formula $\langle R <: T \rangle$

18.4 Resolution

Given a bound set that does not contain the bound $false$, a subset of the inference variables mentioned by the bound set may be resolved. This means that a satisfactory instantiation may be added to the set for each inference variable, until all the requested variables have instantiations.

Dependencies in the bound set may require that the variables be resolved in a particular order, or that additional variables be resolved. Dependencies are specified as follows:

• Given a bound of one of the following forms, where $T$ is either an inference variable $\beta$ or a type that mentions $\beta$:
  – $\alpha = T$
  – $\alpha <: T$
  – $T = \alpha$
  – $T <: \alpha$

If $\alpha$ appears on the left-hand side of another bound of the form $G<..., \alpha, ...> = \text{capture}(G<...>)$, then $\beta$ depends on the resolution of $\alpha$. Otherwise, $\alpha$ depends on the resolution of $\beta$.

• An inference variable $\alpha$ appearing on the left-hand side of a bound of the form $G<..., \alpha, ...> = \text{capture}(G<...>)$ depends on the resolution of every other inference variable mentioned in this bound (on both sides of the $=$ sign).
• An inference variable $\alpha$ depends on the resolution of an inference variable $\beta$ if there exists an inference variable $\gamma$ such that $\alpha$ depends on the resolution of $\gamma$ and $\gamma$ depends on the resolution of $\beta$.

• An inference variable $\alpha$ depends on the resolution of itself.

Given a set of inference variables to resolve, let $V$ be the union of this set and all variables upon which the resolution of at least one variable in this set depends.

If every variable in $V$ has an instantiation, then resolution succeeds and this procedure terminates.

Otherwise, let $\{\alpha_1, \ldots, \alpha_n\}$ be a non-empty subset of uninstantiated variables in $V$ such that (i) for all $i$ ($1 \leq i \leq n$), if $\alpha_i$ depends on the resolution of a variable $\beta$, then either $\beta$ has an instantiation or there is some $j$ such that $\beta = \alpha_j$; and (ii) there exists no non-empty proper subset of $\{\alpha_1, \ldots, \alpha_n\}$ with this property. Resolution proceeds by generating an instantiation for each of $\alpha_1, \ldots, \alpha_n$ based on the bounds in the bound set:

• If the bound set does not contain a bound of the form $G<\ldots, \alpha_i, \ldots> = \text{capture}(G<\ldots>)$ for all $i$ ($1 \leq i \leq n$), then a candidate instantiation $T_i$ is defined for each $\alpha_i$:
  – If $\alpha_i$ has one or more proper lower bounds, $L_1, \ldots, L_k$, then $T_i = \text{lub}(L_1, \ldots, L_k)$ (§4.10.4).
  – Otherwise, if the bound set contains throws $\alpha_i$, and each proper upper bound of $\alpha_i$ is a supertype of $\text{RuntimeException}$, then $T_i = \text{RuntimeException}$.
  – Otherwise, where $\alpha_i$ has proper upper bounds $U_1, \ldots, U_k$, $T_i = \text{glb}(U_1, \ldots, U_k)$ (§5.1.10).

The bounds $\alpha_1 = T_1, \ldots, \alpha_n = T_n$ are incorporated with the current bound set.

If the result does not contain the bound false, then the result becomes the new bound set, and resolution proceeds by selecting a new set of variables to instantiate (if necessary), as described above.

Otherwise, the result contains the bound false, so a second attempt is made to instantiate $\{\alpha_1, \ldots, \alpha_n\}$ by performing the step below:

• If the bound set contains a bound of the form $G<\ldots, \alpha_i, \ldots> = \text{capture}(G<\ldots>)$ for some $i$ ($1 \leq i \leq n$), or;

If the bound set produced in the step above contains the bound false; then let $Y_1, \ldots, Y_n$ be fresh type variables whose bounds are as follows:
– For all \(i (1 \leq i \leq n)\), if \(\alpha_i\) has one or more proper lower bounds \(L_1, ..., L_k\), then let the lower bound of \(Y_i\) be \(\text{lub}(L_1, ..., L_k)\); if not, then \(Y_i\) has no lower bound.

– For all \(i (1 \leq i \leq n)\), where \(\alpha_i\) has upper bounds \(U_1, ..., U_k\), let the upper bound of \(Y_i\) be \(\text{glb}(U_1 \theta, ..., U_k \theta)\), where \(\theta\) is the substitution \([\alpha_1:=Y_1, ..., \alpha_n:=Y_n]\).

If the type variables \(Y_1, ..., Y_n\) do not have well-formed bounds (that is, a lower bound is not a subtype of an upper bound, or an intersection type is inconsistent), then resolution fails.

Otherwise, for all \(i (1 \leq i \leq n)\), all bounds of the form \(G<..., \alpha_i, ...> = \text{capture}(G<...>)\) are removed from the current bound set, and the bounds \(\alpha_1 = Y_1, ..., \alpha_n = Y_n\) are incorporated.

If the result does not contain the bound \(\text{false}\), then the result becomes the new bound set, and resolution proceeds by selecting a new set of variables to instantiate (if necessary), as described above.

Otherwise, the result contains the bound \(\text{false}\), and resolution fails.

The first method of instantiating an inference variable derives the instantiation from that variable's bounds. Sometimes, however, complex dependencies mean that the result is not within the variable's bounds. In that case, a different method of instantiation is performed, analogous to capture conversion (§5.1.10): fresh type variables are introduced, with bounds derived from the bounds of the inference variables. Note that the lower bounds of these "capture" variables are computed using only proper types: this is important in order to avoid attempts to perform typing computations on uninstantiated type variables.

### 18.5 Uses of Inference

Using the inference processes defined above, the following analyses are performed at compile time.

#### 18.5.1 Invocation Applicability Inference

Given a method invocation that provides no explicit type arguments, the process to determine whether a potentially applicable generic method \(m\) is applicable is as follows:

- Where \(P_1, ..., P_p (p \geq 1)\) are the type parameters of \(m\), let \(\alpha_1, ..., \alpha_p\) be inference variables, and let \(\theta\) be the substitution \([P_1:=\alpha_1, ..., P_p:=\alpha_p]\).

- An initial bound set, \(B_0\), is generated from the declared bounds of \(P_1, ..., P_p\), as described in §18.1.3.
• For all \( i \leq i \leq p \), if \( F_i \) appears in the throws clause of \( m \), then the bound throws \( \alpha_i \) is implied. These bounds, if any, are incorporated with \( B_0 \) to produce a new bound set, \( B_1 \).

• A set of constraint formulas, \( C \), is generated as follows.

Let \( F_1, ..., F_n \) be the formal parameter types of \( m \), and let \( e_1, ..., e_k \) be the actual argument expressions of the invocation. Then:

– To test for **applicability by strict invocation**:

  If \( k \neq n \), or if there exists an \( i \leq i \leq n \) such that \( e_i \) is pertinent to applicability (§15.12.2.2) and either (i) \( e_i \) is a standalone expression of a primitive type but \( F_i \) is a reference type, or (ii) \( F_i \) is a primitive type but \( e_i \) is not a standalone expression of a primitive type; then the method is not applicable and there is no need to proceed with inference.

  Otherwise, \( C \) includes, for all \( i \leq i \leq k \) where \( e_i \) is pertinent to applicability, \( \langle e_i \rightarrow F_i \theta \rangle \).

– To test for **applicability by loose invocation**:

  If \( k \neq n \), the method is not applicable and there is no need to proceed with inference.

  Otherwise, \( C \) includes, for all \( i \leq i \leq k \) where \( e_i \) is pertinent to applicability, \( \langle e_i \rightarrow F_i \theta \rangle \).

– To test for **applicability by variable arity invocation**:

  Let \( F'_1, ..., F'_k \) be the first \( k \) variable arity parameter types of \( m \) (§15.12.2.4). \( C \) includes, for all \( i \leq i \leq k \) where \( e_i \) is pertinent to applicability, \( \langle e_i \rightarrow F'_i \theta \rangle \).

• \( C \) is reduced (§18.2) and the resulting bounds are incorporated with \( B_1 \) to produce a new bound set, \( B_2 \).

• Finally, the method \( m \) is applicable if \( B_2 \) does not contain the bound \( false \) and resolution of all the inference variables in \( B_2 \) succeeds (§18.4).

Consider the following method invocation and assignment:

```java
List<Number> ln = Arrays.asList(1, 2.0);
```

A most specific applicable method for the invocation must be identified as described in §15.12. The only potentially applicable method (§15.12.2.1) is declared as follows:

```java
public static <T> List<T> asList(T... a)
```
Trivially (because of its arity), this method is neither applicable by strict invocation (§15.12.2.2) nor applicable by loose invocation (§15.12.2.3). But since there are no other candidates, in a third phase the method is checked for applicability by variable arity invocation.

The initial bound set, $B$, is a trivial upper bound for a single inference variable, $\alpha$:

$\{ \alpha <: \text{Object} \}$

The initial constraint formula set is as follows:

$\{ \langle 1 \rightarrow \alpha, \langle 2.0 \rightarrow \alpha \rangle \}$

These are reduced to a new bound set, $B_1$:

$\{ \alpha <: \text{Object}, \text{Integer} <: \alpha, \text{Double} <: \alpha \}$

Then, to test whether the method is applicable, we attempt to resolve these bounds. We succeed, producing the rather complex instantiation

$\alpha = \text{Number} & \text{Comparable}<? \text{ extends Number} & \text{Comparable}<?>>$

We have thus demonstrated that the method is applicable; since no other candidates exist, it is the most specific applicable method. Still, the type of the method invocation, and its compatibility with the target type in the assignment, is not determined until further inference can occur, as described in the next section.

### 18.5.2 Invocation Type Inference

Given a method invocation expression that provides no explicit type arguments, and a corresponding most specific applicable generic method $m$, the process to infer the invocation type (§15.12.2.6) of the chosen method may require resolving additional constraints, both to assert compatibility with a target type and to assert validity of the method invocation's argument expressions.

It is important to note that multiple "rounds" of inference are involved in finding the type of a method invocation. This is necessary, for example, to allow a target type to influence the type of the invocation without allowing it to influence the choice of an applicable method. The first round (§18.5.1) produces a bound set and tests that a resolution exists, but does not commit to that resolution. Subsequent rounds reduce additional constraints until a final resolution step determines the "real" type of the expression.

#### 18.5.2.1 Poly Method Invocation Compatibility

If the method invocation expression is a poly expression (§15.12), its compatibility with a target type $\tau$ is determined as follows.
If the method invocation expression appears in a strict invocation context and \( T \) is a primitive type, the expression is not compatible with \( T \).

Otherwise:

- Let \( B_2 \) be the bound set produced by reduction in order to demonstrate that \( m \) is applicable in §18.5.1.

  (While it was necessary in §18.5.1 to demonstrate that the inference variables in \( B_2 \) could be resolved, in order to establish applicability, the instantiations produced by this resolution step are not considered part of \( B_2 \).)

- Let \( B_3 \) be the bound set derived from \( B_2 \) as follows.

  Let \( R \) be the return type of \( m \), and let \( \theta \) be the substitution \([P_1:=\alpha_1, \ldots, P_p:=\alpha_p]\) defined in §18.5.1 to replace the type parameters of \( m \) with inference variables, and let \( \tau \) be the invocation's target type. Then:

  - If unchecked conversion was necessary for the method to be applicable during constraint set reduction in §18.5.1, the constraint formula \( \langle R \rangle \rightarrow \tau \) is reduced and incorporated with \( B_2 \).

  - Otherwise, if \( R \theta \) is a parameterized type, \( G<A_1, \ldots, A_n> \), and one of \( A_1, \ldots, A_n \) is a wildcard, then, for fresh inference variables \( \beta_1, \ldots, \beta_n \), the constraint formula \( \langle G<\beta_1, \ldots, \beta_n> \rightarrow \tau \rangle \) is reduced and incorporated, along with the bound \( G<\beta_1, \ldots, \beta_n> = \text{capture}(G<A_1, \ldots, A_n>) \), with \( B_2 \).

  - Otherwise, if \( R \theta \) is an inference variable \( \alpha \), and one of the following is true:

    - \( \tau \) is a reference type, but is not a wildcard-parameterized type, and either (i) \( B_2 \) contains a bound of one of the forms \( \alpha = S \) or \( S <: \alpha \), where \( S \) is a wildcard-parameterized type, or (ii) \( B_2 \) contains two bounds of the forms \( S_1 <: \alpha \) and \( S_2 <: \alpha \), where \( S_1 \) and \( S_2 \) have supertypes that are two different parameterizations of the same generic class or interface.

    - \( \tau \) is a parameterization of a generic class or interface, \( G \), and \( B_2 \) contains a bound of one of the forms \( \alpha = S \) or \( S <: \alpha \), where there exists no type of the form \( G<\ldots> \) that is a supertype of \( S \), but the raw type \( |G<\ldots>| \) is a supertype of \( S \).

    - \( \tau \) is a primitive type, and one of the primitive wrapper classes mentioned in §5.1.7 is an instantiation, upper bound, or lower bound for \( \alpha \) in \( B_2 \).

    then \( \alpha \) is resolved in \( B_2 \), and where the capture of the resulting instantiation of \( \alpha \) is \( U \), the constraint formula \( \langle U \rightarrow \tau \rangle \) is reduced and incorporated with \( B_2 \).

- Otherwise, the constraint formula \( \langle R \theta \rightarrow \tau \rangle \) is reduced and incorporated with \( B_2 \).
- The method invocation expression is compatible with $T$ if $B_3$ does not contain the bound `false` and resolution of all the inference variables in $B_3$ succeeds (§18.4).

Consider the example from the previous section:

```java
List<Number> ln = Arrays.asList(1, 2.0);
```

The most specific applicable method was identified as:

```java
public static <T> List<T> asList(T... a)
```

In order to complete type-checking of the method invocation, we must determine whether it is compatible with its target type, `List<Number>`.

The bound set used to demonstrate applicability in the previous section, $B_2$, was:

```java
{ α <: Object, Integer <: α, Double <: α }
```

The new constraint formula set is as follows:

```java
{ List<α> → List<Number> }
```

This compatibility constraint produces an equality bound for $α$, which is included in the new bound set, $B_3$:

```java
{ α <: Object, Integer <: α, Double <: α, α = Number }
```

These bounds are trivially resolved:

```java
α = Number
```

Finally, we perform a substitution on the declared return type of `asList` to determine that the method invocation has type `List<Number>`; clearly, this is compatible with the target type.

This inference strategy is different than the Java SE 7 Edition of *The Java® Language Specification*, which would have instantiated $α$ based on its lower bounds (before even considering the invocation's target type), as we did in the previous section. This would result in a type error, since the resulting type is not a subtype of `List<Number>`.

Under various special circumstances, based on the bounds appearing in $B_2$, we eagerly resolve an inference variable that appears as the return type of the invocation. This is to avoid unfortunate situations in which the usual constraint, $<R \theta \rightarrow T$, is not completeness-preserving. It is, unfortunately, possible that by eagerly resolving the variable, we are unable to make use of bounds that would be inferred later. It is also possible that, in some cases, bounds that will later be inferred from the invocation arguments (such as implicitly typed lambda expressions) would have caused a different outcome if they had been present in $B_2$. 
Despite these limitations, the strategy allows for reasonable outcomes in typical use cases, and is backwards compatible with the algorithm in the Java SE 7 Edition of *The Java® Language Specification*.

### 18.5.2.2 Additional Argument Constraints

The invocation type for the chosen method is determined after considering additional constraints that may be implied by the argument expressions of the method invocation expression, as follows:

- If the method invocation expression is a poly expression, let \( B_3 \) be the bound set generated in §18.5.2.1 to demonstrate compatibility with the actual target type of the method invocation.

If the method invocation expression is not a poly expression, let \( B_3 \) be the same as the bound set produced by reduction in order to demonstrate that \( m \) is applicable in §18.5.1.

(While it was necessary in §18.5.1 and §18.5.2.1 to demonstrate that the inference variables in the bound set could be resolved, the instantiations produced by these resolution steps are *not* considered part of \( B_3 \).)

- A set of constraint formulas, \( C \), is generated as follows.

Let \( e_1, ..., e_k \) be the actual argument expressions of the method invocation expression.

If \( m \) is applicable by strict or loose invocation, let \( F_1, ..., F_k \) be the formal parameter types of \( m \); if \( m \) is applicable by variable arity invocation, let \( F_1, ..., F_k \) the first \( k \) variable arity parameter types of \( m \) (§15.12.2.4).

Let \( \theta \) be the substitution \([P_1:=\alpha_1, ..., P_p:=\alpha_p]\) defined in §18.5.1 to replace the type parameters of \( m \) with inference variables.

Then, for all \( i \) (1 ≤ \( i \) ≤ \( k \)):

- If \( e_i \) is not pertinent to applicability, \( C \) contains \( \langle e_i \rightarrow F_i \theta \rangle \).

- Additional constraints may be included, depending on the form of \( e_i \):

  - If \( e_i \) is a *LambdaExpression*, \( C \) contains \( \langle \text{LambdaExpression} \rightarrow \text{throws} F_i \theta \rangle \), and the lambda body is searched for additional constraints:

    - For a block lambda body, the search is applied recursively to each result expression.

    - For a poly class instance creation expression or a poly method invocation expression, \( C \) contains all the constraint formulas that would appear in the
set $C$ generated by §18.5.2 when inferring the poly expression's invocation type.

» For a parenthesized expression, the search is applied recursively to the contained expression.

» For a conditional expression, the search is applied recursively to the second and third operands.

» For a lambda expression, the search is applied recursively to the lambda body.

› If $e_i$ is a MethodReference, $C$ contains $\langle\text{MethodReference} \rightarrow \text{throws } F_\theta\rangle$.

› If $e_i$ is a poly class instance creation expression or a poly method invocation expression, $C$ contains all the constraint formulas that would appear in the set $C$ generated by §18.5.2 when inferring the poly expression's invocation type.

› If $e_i$ is a parenthesized expression, these rules are applied recursively to the contained expression.

› If $e_i$ is a conditional expression, these rules are applied recursively to the second and third operands.

• While $C$ is not empty, the following process is repeated, starting with the bound set $B_3$ and accumulating new bounds into a "current" bound set, ultimately producing a new bound set, $B_4$:

1. A subset of constraints is selected in $C$, satisfying the property that, for each constraint, no input variable can influence an output variable of another constraint in $C$. The terms input variable and output variable are defined below. An inference variable $\alpha$ can influence an inference variable $\beta$ if $\alpha$ depends on the resolution of $\beta$ (§18.4), or vice versa; or if there exists a third inference variable $\gamma$ such that $\alpha$ can influence $\gamma$ and $\gamma$ can influence $\beta$.

If this subset is empty, then there is a cycle (or cycles) in the graph of dependencies between constraints. In this case, the constraints in $C$ that participate in a dependency cycle (or cycles) and do not depend on any constraints outside of the cycle (or cycles) are considered. A single constraint is selected from these considered constraints, as follows:

– If any of the considered constraints have the form $\langle\text{Expression} \rightarrow t\rangle$, then the selected constraint is the considered constraint of this form that contains the expression to the left (§3.5) of the expression of every other considered constraint of this form.
– If no considered constraint has the form \(<Expression \rightarrow T>\), then the selected constraint is the considered constraint that contains the expression to the left of the expression of every other considered constraint.

2. The selected constraint(s) are removed from \(C\).

3. The input variables \(\alpha_1, \ldots, \alpha_m\) of all the selected constraint(s) are resolved.

4. Where \(T_1, \ldots, T_m\) are the instantiations of \(\alpha_1, \ldots, \alpha_m\), the substitution \([\alpha_1 := T_1, \ldots, \alpha_m := T_m]\) is applied to every constraint.

5. The constraint(s) resulting from substitution are reduced and incorporated with the current bound set.

• Finally, if \(B_4\) does not contain the bound \(false\), the inference variables in \(B_4\) are resolved.

If resolution succeeds with instantiations \(T_1, \ldots, T_p\) for inference variables \(\alpha_1, \ldots, \alpha_p\), let \(\theta'\) be the substitution \([P_1 := T_1, \ldots, P_p := T_p]\). Then:

– If unchecked conversion was necessary for the method to be applicable during constraint set reduction in §18.5.1, then the parameter types of the invocation type of \(m\) are obtained by applying \(\theta'\) to the parameter types of \(m\)'s type, and the return type and thrown types of the invocation type of \(m\) are given by the erasure of the return type and thrown types of \(m\)'s type.

– If unchecked conversion was not necessary for the method to be applicable, then the invocation type of \(m\) is obtained by applying \(\theta'\) to the type of \(m\).

If \(B_4\) contains the bound \(false\), or if resolution fails, then a compile-time error occurs.

The process of reducing additional argument constraints may require carefully ordering constraint formulas of the forms \(<Expression \rightarrow T>, <LambdaExpression \rightarrow \text{throws } T>, \text{and } <MethodReference \rightarrow \text{throws } T>\). To facilitate this ordering, the input variables of these constraints are defined as follows:

• For \(<\text{LambdaExpression} \rightarrow T>\):

  – If \(T\) is an inference variable, it is the (only) input variable.

  – If \(T\) is a functional interface type, and a function type can be derived from \(T\) (§15.27.3), then the input variables include (i) if the lambda expression is implicitly typed, the inference variables mentioned by the function type's parameter types; and (ii) if the function type's return type, \(R\), is not \(\text{void}\), then for each result expression \(e\) in the lambda body (or for the body itself if it is an expression), the input variables of \(<e \rightarrow R>\).
- Otherwise, there are no input variables.

• For `<LambdaExpression → throws T>`:
  - If $T$ is an inference variable, it is the (only) input variable.
  - If $T$ is a functional interface type, and a function type can be derived, as described in §15.27.3, the input variables include (i) if the lambda expression is implicitly typed, the inference variables mentioned by the function type’s parameter types; and (ii) the inference variables mentioned by the function type’s return type.
  - Otherwise, there are no input variables.

• For `<MethodReference → T>`:
  - If $T$ is an inference variable, it is the (only) input variable.
  - If $T$ is a functional interface type with a function type, and if the method reference is inexact (§15.13.1), the input variables are the inference variables mentioned by the function type’s parameter types.
  - Otherwise, there are no input variables.

• For `<MethodReference → throws T>`:
  - If $T$ is an inference variable, it is the (only) input variable.
  - If $T$ is a functional interface type with a function type, and if the method reference is inexact (§15.13.1), the input variables are the inference variables mentioned by the function type’s parameter types and the function type’s return type.
  - Otherwise, there are no input variables.

• For `<Expression → T>`, if `Expression` is a parenthesized expression:
  Where the contained expression of `Expression` is `Expression'`, the input variables are the input variables of `<Expression' → T>.

• For `<ConditionalExpression → T>`:
  Where the conditional expression has the form $e_1 \ ? \ e_2 : e_3$, the input variables are the input variables of `<e_2 → T>` and `<e_3 → T>`.

• For all other constraint formulas, there are no input variables.

The output variables of these constraints are all inference variables mentioned by the type on the right-hand side of the constraint, $T$, that are not input variables.
18.5.3 Functional Interface Parameterization Inference

Where a lambda expression with explicit parameter types \( P_1, ..., P_n \) targets a functional interface type \( F<A_1, ..., A_m> \) with at least one wildcard type argument, then a parameterization of \( F \) may be derived as the ground target type of the lambda expression as follows.

Let \( Q_1, ..., Q_k \) be the parameter types of the function type of the type \( F<\alpha_1, ..., \alpha_m> \), where \( \alpha_1, ..., \alpha_m \) are fresh inference variables.

If \( n \neq k \), no valid parameterization exists. Otherwise, a set of constraint formulas is formed with, for all \( i \) \((1 \leq i \leq n)\), \( P_i = Q_i \). This constraint formula set is reduced to form the bound set \( B \).

If \( B \) contains the bound \( false \), no valid parameterization exists. Otherwise, a new parameterization of the functional interface type, \( F<A'_1, ..., A'_m> \), is constructed as follows, for \( 1 \leq i \leq m \):

- If \( B \) contains an instantiation (§18.1.3) for \( \alpha_i \), \( T \), then \( A'_i = T \).
- Otherwise, \( A'_i = A_i \).

If \( F<A'_1, ..., A'_m> \) is not a well-formed type (that is, the type arguments are not within their bounds), or if \( F<A'_1, ..., A'_m> \) is not a subtype of \( F<A_1, ..., A_m> \), no valid parameterization exists. Otherwise, the inferred parameterization is either \( F<A'_1, ..., A'_m> \), if all the type arguments are types, or the non-wildcard parameterization (§9.9) of \( F<A'_1, ..., A'_m> \), if one or more type arguments are still wildcards.

In order to determine the function type of a wildcard-parameterized functional interface, we have to "instantiate" the wildcard type arguments with specific types. The "default" approach is to simply replace the wildcards with their bounds, as described in §9.8, but this produces spurious errors in cases where a lambda expression has explicit parameter types that do not correspond to the wildcard bounds. For example:

\[
\text{Predicate<? super Integer> p = (Number n) -> n.equals(23);}
\]

The lambda expression is a \text{Predicate<Number>}, which is a subtype of \text{Predicate<? super Integer>} but not \text{Predicate<Integer>}. The analysis in this section is used to infer that \text{Number} is an appropriate choice for the type argument to \text{Predicate}.

That said, the analysis here, while described in terms of general type inference, is intentionally quite simple. The only constraints are equality constraints, which means that reduction amounts to simple pattern matching. A more powerful strategy might also infer constraints from the body of the lambda expression. But, given possible interactions with inference for surrounding and/or nested generic method invocations, this would introduce a lot of extra complexity.
18.5.4 More Specific Method Inference

When testing that one applicable method is *more specific* than another (§15.12.2.5), where the second method is generic, it is necessary to test whether some instantiation of the second method's type parameters can be inferred to make the first method more specific than the second.

Let $m_1$ be the first method and $m_2$ be the second method. Where $m_2$ has type parameters $P_1, \ldots, P_p$, let $\alpha_1, \ldots, \alpha_p$ be inference variables, and let $\theta$ be the substitution $\left[ P_1 := \alpha_1, \ldots, P_p := \alpha_p \right]$.

Let $e_1, \ldots, e_k$ be the argument expressions of the corresponding invocation. Then:

- If $m_1$ and $m_2$ are applicable by strict or loose invocation (§15.12.2.2, §15.12.2.3), then let $S_1, \ldots, S_k$ be the formal parameter types of $m_1$, and let $T_1, \ldots, T_k$ be the result of $\theta$ applied to the formal parameter types of $m_2$.

- If $m_1$ and $m_2$ are applicable by variable arity invocation (§15.12.2.4), then let $S_1, \ldots, S_k$ be the first $k$ variable arity parameter types of $m_1$, and let $T_1, \ldots, T_k$ be the result of $\theta$ applied to the first $k$ variable arity parameter types of $m_2$.

Note that no substitution is applied to $S_1, \ldots, S_k$; even if $m_1$ is generic, the type parameters of $m_1$ are treated as type variables, not inference variables.

The process to determine if $m_1$ is more specific than $m_2$ is as follows:

- First, an initial bound set, $B$, is generated from the declared bounds of $P_1, \ldots, P_p$, as specified in §18.1.3.

- Second, for all $i (1 \leq i \leq k)$, a set of constraint formulas or bounds is generated.

If $T_i$ is a proper type, the result is *true* if $S_i$ is more specific than $T_i$ for $e_i$ (§15.12.2.5), and *false* otherwise. (Note that $S_i$ is always a proper type.)

Otherwise, if $S_i$ and $T_i$ are not both functional interface types, the constraint formula $\langle S_i <: T_i \rangle$ is generated.

Otherwise, if the interface of $S_i$ is a superinterface or a subinterface of the interface of $T_i$ (or, where $S_i$ or $T_i$ is an intersection type, some interface of $S_i$ is a superinterface or a subinterface of some interface of $T_i$), the constraint formula $\langle S_i <: T_i \rangle$ is generated.

Otherwise, let $MT_S$ be the function type of the capture of $S_i$, let $MT_S'$ be the function type of $S_i$ (without capture), and let $MT_T$ be the function type of $T_i$. If $MT_S$ and $MT_T$ have a different number of formal parameters or type parameters, or if $MT_S$ and $MT_S'$ do not have the same type parameters (§8.4.4), the result is *false*. 


Otherwise, the following constraint formulas or bounds are generated from the type parameters, formal parameter types, and return types of \( MT_S \) and \( MT_T \):

- Let \( A_1, \ldots, A_n \) be the type parameters of \( MT_S \), and let \( B_1, \ldots, B_n \) be the type parameters of \( MT_T \).

  Let \( \theta' \) be the substitution \([B_1:=A_1, \ldots, B_n:=A_n]\). Then, for all \( j (1 \leq j \leq n) \):

  > If the bound of \( A_j \) mentions one of \( A_1, \ldots, A_n \), and the bound of \( B_j \) is not a proper type, \( false \).

  > Otherwise, where \( X \) is the bound of \( A_j \) and \( Y \) is the bound of \( B_j \), \( X = Y \theta' \).

  If the bound \( A_j \) mentions one of \( A_1, \ldots, A_n \), and the bound of \( B_j \) is not a proper type, then producing an equality constraint would raise the possibility of an inference variable being bounded by an out-of-scope type variable. Since instantiating an inference variable with an out-of-scope type variable is nonsensical, we prefer to avoid the situation by giving up immediately whenever the possibility arises. This simplification is not completeness-preserving. (The same comment applies to the treatment of formal parameter types and return types below.)

- Let \( U_1, \ldots, U_k \) be the formal parameter types of \( MT_S \), and let \( V_1, \ldots, V_k \) be the formal parameter types of \( MT_T \). Then, for all \( j (1 \leq j \leq k) \):

  > If \( U_j \) mentions one of \( A_1, \ldots, A_n \), and \( V_j \) is not a proper type, \( false \).

  > Otherwise, \( V_j \theta' <: U_j \), and, where \( U_1', \ldots, U_k' \) are the formal parameter types of \( MT_S' \), and \( A_1', \ldots, A_n' \) are the type parameters of \( MT_S' \), \( V_j[B_1:=A_1', \ldots, B_n:=A_n'] = U_j' \).

- Let \( R_S \) be the return type of \( MT_S \), and let \( R_T \) be the return type of \( MT_T \). Then:

  > If \( R_S \) mentions one of \( A_1, \ldots, A_n \), and \( R_T \) is not a proper type, \( false \).

  > Otherwise, if \( e_i \) is an explicitly typed lambda expression:

    » If \( R_T \) is void, \( true \).

    » Otherwise, if \( R_S \) and \( R_T \) are functional interface types, and \( e_i \) has at least one result expression, then for each result expression in \( e_i \), this entire second step is repeated to infer constraints under which \( R_S \) is more specific than \( R_T \theta' \) for the given result expression.

    » Otherwise, if \( R_S \) is a primitive type and \( R_T \) is not, and \( e_i \) has at least one result expression, and each result expression of \( e_i \) is a standalone expression \((\S 15.2)\) of a primitive type, \( true \).

    » Otherwise, if \( R_T \) is a primitive type and \( R_S \) is not, and \( e_i \) has at least one result expression, and each result expression of \( e_i \) is either a standalone expression of a reference type or a poly expression, \( true \).
» Otherwise, \(<\mathit{R}_S <:\mathit{R}_T \theta'>\).

» Otherwise, if \(e_i\) is an exact method reference:

» If \(\mathit{R}_T\) is \texttt{void}, \texttt{true}.

» Otherwise, if \(\mathit{R}_S\) is a primitive type and \(\mathit{R}_T\) is not, and the compile-time declaration for \(e_i\) has a primitive return type, \texttt{true}.

» Otherwise if \(\mathit{R}_T\) is a primitive type and \(\mathit{R}_S\) is not, and the compile-time declaration for \(e_i\) has a reference return type, \texttt{true}.

» Otherwise, \(<\mathit{R}_S <:\mathit{R}_T \theta'>\).

» Otherwise, if \(e_i\) is a parenthesized expression, these rules for constraints derived from \(\mathit{R}_S\) and \(\mathit{R}_T\) are applied recursively for the contained expression.

» Otherwise, if \(e_i\) is a conditional expression, these rules for constraints derived from \(\mathit{R}_S\) and \(\mathit{R}_T\) are applied recursively for each of the second and third operands.

» Otherwise, \texttt{false}.

- Third, if \(m_2\) is applicable by variable arity invocation and has \(k+1\) parameters, then where \(\mathit{S}_{k+1}\) is the \(k+1\)'th variable arity parameter type of \(m_1\) and \(\mathit{T}_{k+1}\) is the result of \(\theta\) applied to the \(k+1\)'th variable arity parameter type of \(m_2\), the constraint \(<\mathit{S}_{k+1} <:\mathit{T}_{k+1}\>\) is generated.

- Fourth, the generated bounds and constraint formulas are reduced and incorporated with \(B\) to produce a bound set \(B'\).

If \(B'\) does not contain the bound \texttt{false}, and resolution of all the inference variables in \(B'\) succeeds, then \(m_1\) is more specific than \(m_2\).

Otherwise, \(m_1\) is not more specific than \(m_2\).
This chapter repeats the syntactic grammar given in Chapters 4, 6-10, 14, and 15, as well as key parts of the lexical grammar from Chapter 3, using the notation from §2.4.

Productions from §3 (Lexical Structure)

**Identifier:**

*IdentifierChars* but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

**IdentifierChars:**

*JavaLetter* { *JavaLetterOrDigit* }

**JavaLetter:**

any Unicode character that is a "Java letter"

**JavaLetterOrDigit:**

any Unicode character that is a "Java letter-or-digit"

**TypeIdentifier:**

*Identifier* but not *var*

**Literal:**

*IntegerLiteral*  
*FloatingPointLiteral*  
*BooleanLiteral*  
*CharacterLiteral*  
*StringLiteral*  
*NullLiteral*
Productions from §4 (Types, Values, and Variables)

Type:
  PrimitiveType
  ReferenceType

PrimitiveType:
  {Annotation} NumericType
  {Annotation} boolean

NumericType:
  IntegralType
  FloatingPointType

IntegralType:
  (one of)
  byte short int long char

FloatingPointType:
  (one of)
  float double

ReferenceType:
  ClassOrInterfaceType
  TypeVariable
  ArrayType

ClassOrInterfaceType:
  ClassType
  InterfaceType

ClassType:
  {Annotation} TypeIdentifier [TypeArguments]
  PackageName . {Annotation} TypeIdentifier [TypeArguments]
  ClassOrInterfaceType . {Annotation} TypeIdentifier [TypeArguments]

InterfaceType:
  ClassType

TypeVariable:
  {Annotation} TypeIdentifier
ArrayType:
  
  PrimitiveType Dims
  ClassOrInterfaceType Dims
  TypeVariable Dims

Dims:
  {Annotation} [] {{Annotation} []}

TypeParameter:
  {TypeParameterModifier} TypeIdentifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  extends TypeVariable
  extends ClassOrInterfaceType {AdditionalBound}

AdditionalBound:
  & InterfaceType

TypeArguments:
  < TypeArgumentList >

TypeArgumentList:
  TypeArgument {, TypeArgument}

TypeArgument:
  ReferenceType
  Wildcard

Wildcard:
  {Annotation} ? [WildcardBounds]

WildcardBounds:
  extends ReferenceType
  super ReferenceType
Productions from §6 (Names)

ModuleName:
  Identifier
  ModuleName . Identifier

PackageName:
  Identifier
  PackageName . Identifier

TypeName:
  TypeIdentifier
  PackageOrTypeName . TypeIdentifier

ExpressionName:
  Identifier
  AmbiguousName . Identifier

MethodName:
  Identifier

PackageOrTypeName:
  Identifier
  PackageOrTypeName . Identifier

AmbiguousName:
  Identifier
  AmbiguousName . Identifier
Productions from §7 (Packages and Modules)

CompilationUnit:
    OrdinaryCompilationUnit
    ModularCompilationUnit

OrdinaryCompilationUnit:
    [PackageDeclaration] [ImportDeclaration] [TypeDeclaration]

ModularCompilationUnit:
    [ImportDeclaration] ModuleDeclaration

PackageDeclaration:
    [PackageModifier] package Identifier { . Identifier} ;

PackageModifier:
    Annotation

ImportDeclaration:
    SingleTypeImportDeclaration
    TypeImportOnDemandDeclaration
    SingleStaticImportDeclaration
    StaticImportOnDemandDeclaration

SingleTypeImportDeclaration:
    import TypeName ;

TypeImportOnDemandDeclaration:
    import PackageOrTypeName . * ;

SingleStaticImportDeclaration:
    import static TypeName . Identifier ;

StaticImportOnDemandDeclaration:
    import static TypeName . * ;

TypeDeclaration:
    ClassDeclaration
    InterfaceDeclaration
    ;
ModuleDeclaration:
   {Annotation} [open] module Identifier {. Identifier}
   { {ModuleDirective} }

ModuleDirective:
   requires {RequiresModifier} ModuleName ;
   exports PackageName [to ModuleName {, ModuleName}] ;
   opens PackageName [to ModuleName {, ModuleName}] ;
   uses TypeName ;
   provides TypeName with TypeName {, TypeName} ;

RequiresModifier:
   (one of)
   transitive static
Productions from §8 (Classes)

ClassDeclaration:
   NormalClassDeclaration
   EnumDeclaration

NormalClassDeclaration:
   { ClassModifier } class TypeIdentifier [TypeParameters]
   [Superclass] [Superinterfaces] ClassBody

ClassModifier:
   (one of)
   Annotation public protected private
   abstract static final strictfp

TypeParameters:
   < TypeParameterList >

TypeParameterList:
   TypeParameter { , TypeParameter}

Superclass:
   extends ClassType

Superinterfaces:
   implements InterfaceTypeList

InterfaceTypeList:
   InterfaceType { , InterfaceType}

ClassBody:
   { {ClassBodyDeclaration} }

ClassBodyDeclaration:
   ClassMemberDeclaration
   InstanceInitializer
   StaticInitializer
   ConstructorDeclaration
ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;

FieldDeclaration:
  {FieldModifier} UnannType VariableDeclaratorList ;

FieldModifier:
  (one of)
  Annotation public protected private
  static final transient volatile

VariableDeclaratorList:
  VariableDeclarator [, VariableDeclarator]

VariableDeclarator:
  VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
  Identifier [Dims]

VariableInitializer:
  Expression
  ArrayInitializer
SYNTAX

UnannType:
  UnannPrimitiveType
  UnannReferenceType

UnannPrimitiveType:
  NumericType
    boolean

UnannReferenceType:
  UnannClassOrInterfaceType
  UnannTypeVariable
  UnannArrayType

UnannClassOrInterfaceType:
  UnannClassType
  UnannInterfaceType

UnannClassType:
  TypeIdentifier [TypeArguments]
  PackageName . {Annotation} TypeIdentifier [TypeArguments]
  UnannClassOrInterfaceType . {Annotation} TypeIdentifier
    [TypeArguments]

UnannInterfaceType:
  UnannClassType

UnannTypeVariable:
  TypeIdentifier

UnannArrayType:
  UnannPrimitiveType Dims
  UnannClassOrInterfaceType Dims
  UnannTypeVariable Dims
MethodDeclaration:
   {MethodModifier} MethodHeader MethodBody

MethodModifier:
   (one of)
   Annotation public protected private
   abstract static final synchronized native strictfp

MethodHeader:
   Result MethodDeclarator [Throws]
   TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
   UnannType
   void

MethodDeclarator:
   Identifier ( [ReceiverParameter ,] [FormalParameterList] ) [Dims]

ReceiverParameter:
   {Annotation} UnannType [Identifier .] this

FormalParameterList:
   FormalParameter [, FormalParameter]

FormalParameter:
   {VariableModifier} UnannType VariableDeclaratorId
   VariableArityParameter

VariableArityParameter:
   {VariableModifier} UnannType {Annotation} . . . Identifier

VariableModifier:
   Annotation
   final

Throws:
   throws ExceptionTypeList

ExceptionTypeList:
   ExceptionType [, ExceptionType]
ExceptionType:
  ClassType
  TypeVariable

MethodBody:
  Block
  ;

InstanceInitializer:
  Block

StaticInitializer:
  static Block

ConstructorDeclaration:
  {ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

ConstructorModifier:
  (one of)
  Annotation public protected private

ConstructorDeclarator:
  [TypeParameters] SimpleTypeName
    ( [ReceiverParameter , ] [FormalParameterList] )

SimpleTypeName:
  TypeIdentifier

ConstructorBody:
  { [ExplicitConstructorInvocation] [BlockStatements] }

ExplicitConstructorInvocation:
  [TypeArguments] this ( [ArgumentList] ) ;
  [TypeArguments] super ( [ArgumentList] ) ;
  ExpressionName . [TypeArguments] super ( [ArgumentList] ) ;
  Primary . [TypeArguments] super ( [ArgumentList] ) ;
EnumDeclaration:
   {ClassModifier} enum TypeIdentifier [Superinterfaces] EnumBody

EnumBody:
   { [EnumConstantList] [,] [EnumBodyDeclarations] }

EnumConstantList:
   EnumConstant [, EnumConstant]

EnumConstant:
   {EnumConstantModifier} Identifier [ ( [ArgumentList] ) ] [ClassBody]

EnumConstantModifier:
   Annotation

EnumBodyDeclarations:
   ; {ClassBodyDeclaration}
Productions from §9 (Interfaces)

InterfaceDeclaration:
   NormalInterfaceDeclaration
   AnnotationTypeDeclaration

NormalInterfaceDeclaration:
   {InterfaceModifier} interface TypeIdentifier [TypeParameters]
   [ExtendsInterfaces] InterfaceBody

InterfaceModifier:
   (one of)
   Annotation public protected private
   abstract static strictfp

ExtendsInterfaces:
   extends InterfaceTypeList

InterfaceBody:
   { {InterfaceMemberDeclaration} }

InterfaceMemberDeclaration:
   ConstantDeclaration
   InterfaceMethodDeclaration
   ClassDeclaration
   InterfaceDeclaration
   ;

ConstantDeclaration:
   {ConstantModifier} UnannType VariableDeclaratorList ;

ConstantModifier:
   (one of)
   Annotation public
   static final

InterfaceMethodDeclaration:
   {InterfaceMethodModifier} MethodHeader MethodBody
**SYNTAX**

**InterfaceMethodModifier:**
(one of)

*Annotation* public private
abstract default static strictfp

**AnnotationTypeDeclaration:**

{**InterfaceModifier**} @ interface **TypeIdentifier** AnnotationTypeBody

**AnnotationTypeBody:**

{ **AnnotationTypeMemberDeclaration** }

**AnnotationTypeMemberDeclaration:**

AnnotationTypeElementDeclaration
ConstantDeclaration
ClassDeclaration
InterfaceDeclaration
;

**AnnotationTypeElementDeclaration:**

{ **AnnotationTypeElementModifier** } UnannType Identifier () [Dims]
[DefaultValue] ;

**AnnotationTypeElementModifier:**

( one of )
*Annotation* public
abstract

**DefaultValue:**

default *ElementValue*
Annotation:
   NormalAnnotation
   MarkerAnnotation
   SingleElementAnnotation

NormalAnnotation:
   @ TypeName ([ElementValuePairList])

ElementValuePairList:
   ElementValuePair {, ElementValuePair}

ElementValuePair:
   Identifier = ElementValue

ElementValue:
   ConditionalExpression
   ElementValueArrayInitializer
   Annotation

ElementValueArrayInitializer:
   { [ElementValueList], ] }

ElementValueList:
   ElementValue {, ElementValue}

MarkerAnnotation:
   @ TypeName

SingleElementAnnotation:
   @ TypeName ( ElementValue )
Productions from §10 (Arrays)

ArrayInitializer:
   { [VariableInitializerList] [, ] }

VariableInitializerList:
   VariableInitializer { , VariableInitializer}
Productions from §14 (Blocks and Statements)

Block:

\{
    [BlockStatements]
\}

BlockStatements:

    BlockStatement {BlockStatement}

BlockStatement:

    LocalVariableDeclarationStatement
    ClassDeclaration
    Statement

LocalVariableDeclarationStatement:

    LocalVariableDeclaration

LocalVariableDeclaration:

    \{VariableModifier\} LocalVariableType VariableDeclaratorList

LocalVariableType:

    UnannType
    var

Statement:

    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement

StatementNoShortIf:

    StatementWithoutTrailingSubstatement
    LabeledStatementNoShortIf
    IfThenElseStatementNoShortIf
    WhileStatementNoShortIf
    ForStatementNoShortIf
StatementWithoutTrailingSubstatement:
  Block
  EmptyStatement
  ExpressionStatement
  AssertStatement
  SwitchStatement
  DoStatement
  BreakStatement
  ContinueStatement
  ReturnStatement
  SynchronizedStatement
  ThrowStatement
  TryStatement

EmptyStatement:
  ;

LabeledStatement:
  Identifier : Statement

LabeledStatementNoShortIf:
  Identifier : StatementNoShortIf

ExpressionStatement:
  StatementExpression ;

StatementExpression:
  Assignment
  PreIncrementExpression
  PreDecrementExpression
  PostIncrementExpression
  PostDecrementExpression
  MethodInvocation
  ClassInstanceCreationExpression
**SYNTAX**

IfThenStatement:
    if ( Expression ) Statement

IfThenElseStatement:
    if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
    if ( Expression ) StatementNoShortIf else StatementNoShortIf

AssertStatement:
    assert Expression ;
    assert Expression : Expression ;

SwitchStatement:
    switch ( Expression ) SwitchBlock

SwitchBlock:
    { {SwitchBlockStatementGroup} {SwitchLabel} }

SwitchBlockStatementGroup:
    SwitchLabels BlockStatements

SwitchLabels:
    SwitchLabel {SwitchLabel}

SwitchLabel:
    case ConstantExpression :
    case EnumConstantName :
    default :

EnumConstantName:
    Identifier

WhileStatement:
    while ( Expression ) Statement

WhileStatementNoShortIf:
    while ( Expression ) StatementNoShortIf

DoStatement:
    do Statement while ( Expression ) ;
ForStatement:
  BasicForStatement
  EnhancedForStatement

ForStatementNoShortIf:
  BasicForStatementNoShortIf
  EnhancedForStatementNoShortIf

BasicForStatement:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement

BasicForStatementNoShortIf:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) StatementNoShortIf

ForInit:
  StatementExpressionList
  LocalVariableDeclaration

ForUpdate:
  StatementExpressionList

StatementExpressionList:
  StatementExpression { , StatementExpression}

EnhancedForStatement:
  for ( {VariableModifier} LocalVariableType VariableDeclaratorId
       : Expression )
    Statement

EnhancedForStatementNoShortIf:
  for ( {VariableModifier} LocalVariableType VariableDeclaratorId
         : Expression )
    StatementNoShortIf

BreakStatement:
  break [Identifier] ;

ContinueStatement:
  continue [Identifier] ;

ReturnStatement:
  return [Expression] ;
SYNTAX

ThrowStatement:
    throw Expression ;

SynchronizedStatement:
    synchronized ( Expression ) Block

TryStatement:
    try Block Catches
    try Block [Catches] Finally
    TryWithResourcesStatement

Catches:
    CatchClause {CatchClause}

CatchClause:
    catch ( CatchFormalParameter ) Block

CatchFormalParameter:
    {VariableModifier} CatchType VariableDeclaratorId

CatchType:
    UnannClassType { | ClassType}

Finally:
    finally Block

TryWithResourcesStatement:
    try ResourceSpecification Block [Catches] [Finally]

ResourceSpecification:
    ( ResourceList [ ; ] )

ResourceList:
    Resource { ; Resource}

Resource:
    {VariableModifier} LocalVariableType Identifier = Expression
    VariableAccess
Productions from §15 (Expressions)

Primary:
  PrimaryNoNewArray
  ArrayCreationExpression

PrimaryNoNewArray:
  Literal
  ClassLiteral
  this
  This
  this ( Expression )
  ClassInstanceCreationExpression
  FieldAccess
  ArrayAccess
  MethodInvocation
  MethodReference

ClassLiteral:
  TypeName { [ ] } . class
  NumericType { [ ] } . class
  boolean { [ ] } . class
  void . class

ClassInstanceCreationExpression:
  UnqualifiedClassInstanceCreationExpression
  ExpressionName . UnqualifiedClassInstanceCreationExpression
  Primary . UnqualifiedClassInstanceCreationExpression

UnqualifiedClassInstanceCreationExpression:
  new TypeArguments
    ClassOrInterfaceTypeToInstantiate ( [ArgumentList] ) [ClassBody]

ClassOrInterfaceTypeToInstantiate:
  [Annotation] Identifier { . [Annotation] Identifier }
    [TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
  TypeArguments
  <>
FieldAccess:

Primary . Identifier
super . Identifier
TypeName . super . Identifier

ArrayAccess:

ExpressionName [ Expression ]
PrimaryNoNewArray [ Expression ]

MethodInvocation:

MethodName ( [ArgumentList] )
TypeName . [TypeArguments] Identifier ( [ArgumentList] )
ExpressionName . [TypeArguments] Identifier ( [ArgumentList] )
Primary . [TypeArguments] Identifier ( [ArgumentList] )
super . [TypeArguments] Identifier ( [ArgumentList] )
TypeName . super . [TypeArguments] Identifier ( [ArgumentList] )

ArgumentList:

Expression , Expression

MethodReference:

ExpressionName :: [TypeArguments] Identifier
Primary :: [TypeArguments] Identifier
ReferenceType :: [TypeArguments] Identifier
super :: [TypeArguments] Identifier
TypeName . super :: [TypeArguments] Identifier
ClassType :: [TypeArguments] new
ArrayType :: new

ArrayCreationExpression:

new PrimitiveType DimExprs [ Dims ]
new ClassOrInterfaceType DimExprs [ Dims ]
new PrimitiveType Dims ArrayInitializer
new ClassOrInterfaceType Dims ArrayInitializer

DimExprs:

DimExpr { DimExpr }

DimExpr:

{ Annotation } [ Expression ]
Expression:
   LambdaExpression
   AssignmentExpression

LambdaExpression:
   LambdaParameters -> LambdaBody

LambdaParameters:
   ( [LambdaParameterList] )
   Identifier

LambdaParameterList:
   LambdaParameter {, LambdaParameter}
   Identifier {, Identifier}

LambdaParameter:
   [VariableModifier] LambdaParameterType VariableDeclaratorId
   VariableArityParameter

LambdaParameterType:
   UnannType
   var

LambdaBody:
   Expression
   Block
AssignmentExpression:
  ConditionalExpression
  Assignment

Assignment:
  LeftHandSide AssignmentOperator Expression

LeftHandSide:
  ExpressionName
  FieldAccess
  ArrayAccess

AssignmentOperator:
  (one of)
  =  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=

ConditionalExpression:
  ConditionalOrExpression
  ConditionalOrExpression ? Expression : ConditionalExpression
  ConditionalOrExpression ? Expression : LambdaExpression

ConditionalOrExpression:
  ConditionalAndExpression
  ConditionalOrExpression || ConditionalAndExpression

ConditionalAndExpression:
  InclusiveOrExpression
  ConditionalAndExpression && InclusiveOrExpression

InclusiveOrExpression:
  ExclusiveOrExpression
  InclusiveOrExpression | ExclusiveOrExpression

ExclusiveOrExpression:
  AndExpression
  ExclusiveOrExpression ^ AndExpression

AndExpression:
  EqualityExpression
  AndExpression & EqualityExpression
EqualityExpression:
    RelationalExpression
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression

RelationalExpression:
    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression
    RelationalExpression instanceof ReferenceType

ShiftExpression:
    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression

MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression
    + UnaryExpression
    - UnaryExpression
    UnaryExpressionNotPlusMinus

PreIncrementExpression:
    ++ UnaryExpression

PreDecrementExpression:
    -- UnaryExpression
UnaryExpressionNotPlusMinus:
  PostfixExpression
  ~ UnaryExpression
  ! UnaryExpression
  CastExpression

PostfixExpression:
  Primary
  ExpressionName
  PostIncrementExpression
  PostDecrementExpression

PostIncrementExpression:
  PostfixExpression ++

PostDecrementExpression:
  PostfixExpression --

CastExpression:
  ( PrimitiveType ) UnaryExpression
  ( ReferenceType {AdditionalBound} ) UnaryExpressionNotPlusMinus
  ( ReferenceType {AdditionalBound} ) LambdaExpression

ConstantExpression:
  Expression
ORACLE AMERICA, INC. IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY.

Specification: JSR-384 Java SE 11 (18.9) ("Specification")
Version: 11
Status: Final Release
Specification Lead: Oracle America, Inc. ("Specification Lead")
Release: September 2018

Copyright © 1997, 2018, Oracle America, Inc.
All rights reserved.

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Specification Lead hereby grants you a fully-paid, non-exclusive, nontransferable, worldwide, limited license (without the right to sublicense), under Specification Lead's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Specification Lead also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include
any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Specification Lead or Specification Lead's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle America, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Specification Lead's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.


a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b. With respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Specification Lead that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.
c. Also with respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Specification Lead that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Specification Lead's source code or binary code materials nor, except with an appropriate and separate license from Specification Lead, includes any of Specification Lead's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.oracle", “com.sun” or their equivalents in any subsequent naming convention adopted by Oracle America, Inc. through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Specification Lead which corresponds to the Specification and that was available either (i) from Specification Lead 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Specification Lead if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL
SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY
DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE,
PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL,
INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND
REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF
OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR
OTHERWISE USING THE SPECIFICATION, EVEN IF SPECIFICATION
LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Specification Lead and its licensors
from any claims arising or resulting from: (i) your use of the Specification; (ii) the
use or distribution of your Java application, applet and/or implementation; and/or
(iii) any claims that later versions or releases of any Specification furnished to you
are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the
U.S. Government or by a U.S. Government prime contractor or subcontractor
(at any tier), then the Government's rights in the Software and accompanying
documentation shall be only as set forth in this license; this is in accordance
with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD)
acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Specification Lead with any comments or suggestions concerning
the Specification ("Feedback"), you hereby: (i) agree that such Feedback
is provided on a non-proprietary and nonconfidential basis, and (ii) grant
Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up,
irrevocable license, with the right to sublicense through multiple levels of
sublicensees, to incorporate, disclose, and use without limitation the Feedback for
any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and
controlling U.S. federal law. The U.N. Convention for the International Sale of
Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export
or import regulations in other countries. Licensee agrees to comply strictly with all
such laws and regulations and acknowledges that it has the responsibility to obtain
LIMITED LICENSE GRANT

such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.