

Java Platform, Standard Edition

Java Language Updates



Release 27
G54225-01
September 2026

ORACLE®

Copyright © 2017, 2026, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience	i
Related Documents	i
Conventions	i

1 Java Language Changes Summary

2 Java Language Changes by Release

Java Language Updates for Java SE 27	1
Java Language Updates for Java SE 26	1
Java Language Changes for Java SE 25	1
Java Language Changes for Java SE 24	2
Java Language Changes for Java SE 23	3
Java Language Changes for Java SE 22	5
Java Language Changes for Java SE 21	6
Java Language Changes for Java SE 20	7
Java Language Changes for Java SE 19	8
Java Language Changes for Java SE 18	9
Java Language Changes for Java SE 17	10
Java Language Changes for Java SE 16	11
Java Language Changes for Java SE 15	12
Java Language Changes for Java SE 14	12
Java Language Changes for Java SE 13	13
Java Language Changes for Java SE 12	13
Java Language Changes for Java SE 11	13
Java Language Changes for Java SE 10	14
Java Language Changes for Java SE 9	14
More Concise try-with-resources Statements	14
@SafeVarargs Annotation Allowed on Private Instance Methods	15
Diamond Syntax and Anonymous Inner Classes	15
Underscore Character Not Legal Name	15

3	Preview Features	
4	Module Import Declarations	
5	Flexible Constructor Bodies	
	Early Construction Context	1
6	Compact Source Files and Instance main Methods	
	Creating Compact Source Files	2
	The java.lang.IO Class for Basic Console I/O	4
	Automatic Import of the java.base Module	5
	Running Compact Source Files with the java Launcher and Instance Main Methods	5
	Growing a Program	6
7	Sealed Classes	
8	Primitive Types in Patterns, instanceof, and switch	
9	Pattern Matching	
	Pattern Matching with instanceof	2
	Scope of Pattern Variables and instanceof	4
	Pattern Matching with switch	5
	When Clauses	6
	Pattern Label Dominance	8
	Dominance and Unconditionally Exact Conversions	9
	Scope of Pattern Variables and switch	10
	Null case Labels	12
	Type Patterns	13
	Type Patterns with Reference Types	13
	Type Patterns with Primitive Types	14
	Type Patterns with Parameterized Types	15
	Record Patterns	17
	Generic Record Patterns	18
	Using var in Record Patterns	18

Primitive Types in Record Patterns	19
Nested Record Patterns	20

10 Record Classes

The Canonical Constructor of a Record Class	2
Alternative Record Constructors	3
Explicit Declaration of Record Class Members	3
Features of Record Classes	5
Record Classes and Sealed Classes and Interfaces	6
Local Record Classes	6
Static Members of Inner Classes	7
Differences Between the Serialization of Records and Ordinary Objects	8
Record Serialization Principles	8
How Record Serialization Works	9
APIs Related to Record Classes	12

11 Unnamed Variables and Patterns

12 Switch Expressions and Statements

Arrow Cases	1
Colon Cases and the the yield Statement	2
Qualified enum Constants as case Constants	4
Primitive Values in switch Expressions and Statements	5
Exhaustiveness of switch	6
Completion and switch Expressions	9

13 Text Blocks

14 Local Variable Type Inference

15 Safe Casting with instanceof and switch

Conversion Safety and switch	2
------------------------------	---

Preface

This guide describes the updated language features in Java SE 9 and subsequent releases.

Audience

This document is for Java developers.

Related Documents

See [JDK 27 Documentation](#).

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Java Language Changes Summary

The following tables summarize new Java language features since Java SE 9.

New Language Features

The following table lists the new features in the Java language since Java SE 9.

Table 1-1 Java Language Changes Summary

Release	New Language Features
9	Java Platform Module System (JSR 376): see Project Jigsaw on OpenJDK Milling Project Coin (JEP 213): see Java Language Changes for Java SE 9 Small Enhancements to the Java Programming Language (JSR 334): see Java Language Changes for Java SE 9
10	Local Variable Type Inference (JEP 286)
11	Local-Variable Syntax for Lambda Parameters (JEP 323): see Local Variable Type Inference
14	Switch Expressions and Statements (JEP 361)
15	Text Blocks (JEP 378)
16	Record Classes (JEP 395) Pattern Matching with instanceof (JEP 394)
17	Sealed Classes (JEP 409)
21	Record Patterns (JEP 440) Pattern Matching with switch (JEP 441)
22	Unnamed Variables and Patterns (JEP 456)
25	Module Import Declarations (JEP 511) Compact Source Files and Instance main Methods (JEP 512) Flexible Constructor Bodies (JEP 513)

Permanent and Preview Language Features

The following table lists, for each release since Java SE 9, which language features became permanent and which language features were in preview. See [Preview Features](#) for information about preview features and how they can become permanent.

Table 1-2 Java Language Changes Summary

Release	Permanent Features	Preview Features
27		Primitive Types in Patterns, instanceof, and switch (JEP 532)
26		Primitive Types in Patterns, instanceof, and switch (JEP 530)

Table 1-2 (Cont.) Java Language Changes Summary

Release	Permanent Features	Preview Features
25	Module Import Declarations (JEP 511) Compact Source Files and Instance main Methods (JEP 512) Flexible Constructor Bodies (JEP 513)	Primitive Types in Patterns, instanceof, and switch (JEP 507)
24		Module Import Statements (JEP 494) Primitive Types in Patterns, instanceof, and switch (JEP 488): see: • When Clauses • Type Patterns with Primitive Types • Primitive Types in Record Patterns • Primitive Values in switch Expressions and Statements • Safe Casting with instanceof and switch Flexible Constructor Bodies (JEP 482) Simple Source Files and Instance Main Methods (JEP 495)
23		Module Import Declarations (JEP 476) Primitive Types in Patterns, instanceof, and switch (JEP 455): see: Implicitly Declared Classes and Instance Main Methods (JEP 477) Flexible Constructor Bodies (JEP 482)
22	Unnamed Variables and Patterns (JEP 456)	Statements Before super(...) (JEP 447) String Templates (JEP 459) Implicitly Declared Classes and Instance Main Methods (JEP 463)
21	Record Patterns (JEP 440) Pattern Matching with switch (JEP 441)	String Templates (JEP 430) Unnamed Patterns and Variables (JEP 443) Unnamed Classes and Instance Main Methods (JEP 445)
20		Pattern Matching for switch Expressions and Statements (JEP 433) Record Patterns (JEP 432)
19		Pattern Matching for switch Expressions and Statements (JEP 427) Record Patterns (JEP 405)
18		Pattern Matching for switch Expressions and Statements (JEP 420)
17	Sealed Classes (JEP 409)	Pattern Matching for switch Expressions and Statements (JEP 406)
16	Record Classes (JEP 395) Pattern Matching with instanceof (JEP 394)	Sealed Classes (JEP 397)
15	Text Blocks (JEP 378)	Sealed Classes (JEP 36) Record Classes (JEP 384) Pattern Matching for instanceof (JEP 375)

Table 1-2 (Cont.) Java Language Changes Summary

Release	Permanent Features	Preview Features
14	Switch Expressions and Statements (JEP 361)	Pattern Matching for the instanceof Operator (JEP 305) Records (JEP 359) Text Blocks (JEP 368)
13		Text Blocks (JEP 355), see Programmer's Guide to Text Blocks Switch Expressions (JEP 354)
12		Switch Expressions (JEP 325)
11	Local-Variable Syntax for Lambda Parameters (JEP 323): see Local Variable Type Inference	
10	Local Variable Type Inference (JEP 286)	
9	Java Platform Module System (JSR 376): see Project Jigsaw on OpenJDK Milling Project Coin (JEP 213): see Java Language Changes for Java SE 9 Small Enhancements to the Java Programming Language (JSR 334): see Java Language Changes for Java SE 9	

Evolution of Language Features

The following tables list the changes made to the Java language since Java SE 9. The first column specifies the feature while the subsequent columns specify a Java SE release. An icon specifies whether the feature was made permanent, withdrawn for the specified release, or available as a preview feature:

- : Permanent feature
- : The feature was withdrawn
- , , , : First, second, third, or fourth preview, respectively

See [Preview Features](#) for information about preview features and how they can become permanent or be withdrawn.

Table 1-3 Java Language Changes in Java SE 27 to Java SE 25

Feature	27	26	25
Primitive Types in Patterns, instanceof, and switch	 JEP 532	 JEP 530	 JEP 507
Module Import Declarations			 JEP 511
Flexible Constructor Bodies			 JEP 513

Table 1-3 (Cont.) Java Language Changes in Java SE 27 to Java SE 25

Feature	27	26	25
Compact Source Files and Instance main Methods			 JEP 512

Table 1-4 Java Language Changes from Java SE 24 to Java SE 17

Feature	24	23	22	21	20	19	18	17
Module Import Declarations	 JEP 494	 JEP 476						
Primitive Types in Patterns, instanceof, and switch	 JEP 488	 JEP 455						
Flexible Constructor Bodies	 JEP 492	 JEP 482	 JEP 447					
Simple Source Files and Instance Main Methods	 JEP 495	 JEP 477	 JEP 463	 JEP 445				
Unnamed Variables and Patterns			 JEP 456	 JEP 443				
String Templates		  JEP 459	 JEP 430					
Record Patterns				 JEP 440	 JEP 432	 JEP 405		
Pattern Matching with switch				 JEP 441	 JEP 433	 JEP 427	 JEP 420	 JEP 406
Sealed Classes								 JEP 409

Table 1-5 Java Language Changes from Java SE 16 to Java SE 9

Feature	16	15	14	13	12	11	10	9
Sealed Classes	 JEP 397	 JEP 360						
Record Classes	 JEP 395	 JEP 384	 JEP 359					
Pattern Matching with instanceof	 JEP 394	 JEP 375	 JEP 305					

Table 1-5 (Cont.) Java Language Changes from Java SE 16 to Java SE 9

Feature	16	15	14	13	12	11	10	9
Text Blocks		 JEP 378	 JEP 368	 JEP 355				
Switch Expressions and Statements			 JEP 361	 JEP 354	 JEP 325			
Local-Variable Syntax for Lambda Parameters: see Local Variable Type Inference						 JEP 323		
Local Variable Type Inference							 JEP 286	
Java Platform Module System: see Project Jigsaw on OpenJDK								 JSR 376
Milling Project Coin: see Java Language Changes for Java SE 9								 JEP 213
Small Enhancements to the Java Programming Language: see Java Language Changes for Java SE 9								 JSR 334

2

Java Language Changes by Release

This section describes in detail the updated language features in Java SE 9 and subsequent releases.

Java Language Updates for Java SE 27

Feature	Description	JEP
Primitive Types in Patterns, instanceof, and switch	First previewed in Java SE 23 and previewed again in Java SE 24, Java SE 25, and Java SE 26, this feature is re-previewed for this release without any significant changes.	JEP 532: Primitive Types in Patterns, instanceof, and switch (Fifth Preview)

Java Language Updates for Java SE 26

Feature	Description	JEP
Primitive Types in Patterns, instanceof, and switch	First previewed in Java SE 23 and previewed again in Java SE 24 and Java SE 25, this feature is re-previewed for this release. In this release: - The definition of unconditional exactness has been enhanced; see Safe Casting with instanceof and switch - Tighter dominance checks have been applied in switch constructs; see Dominance and Unconditionally Exact Conversions	JEP 530: Primitive Types in Patterns, instanceof, and switch (Fourth Preview)

Java Language Changes for Java SE 25

Feature	Description	JEP
Primitive Types in Patterns, instanceof, and switch	First previewed in Java SE 23 and previewed again in Java SE 24, this feature is re-previewed for this release without any significant changes.	JEP 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)

Feature	Description	JEP
Module Import Statements	First previewed in Java SE 23, and previewed again in Java SE 24, this feature is permanent in this release without any significant changes.	JEP 511: Module Import Declarations
Compact Source Files and Instance Main Methods	First previewed in Java SE 21 as <i>JEP 445: Unnamed Classes and Instance Main Methods (Preview)</i> and previewed again in Java SE 22, 23, and 24, this feature is permanent in this release with a revised title. In this release: - The new <code>IO</code> class for basic console I/O is now in the <code>java.lang</code> package rather than the <code>java.io</code> package. Thus, it is implicitly imported by every source file. - The static methods of the <code>IO</code> class are no longer implicitly imported into compact source files. Thus invocations of these methods must name the class, for example, <code>IO.println("Hello, world!")</code>, unless the methods are explicitly imported. - The implementation of the <code>IO</code> class is now based upon <code>System.out</code> and <code>System.in</code> rather than the <code>java.io.Console</code> class.	JEP 512: Compact Source Files and Instance Main Methods
Flexible Constructor Bodies	First previewed in Java SE 22 as <i>JEP 447: Statements before <code>super(...)</code> (Preview)</i> , previewed again in Java SE 23 and Java SE 24, this feature is permanent in this release without any significant changes.	JEP 513: Flexible Constructor Bodies

Java Language Changes for Java SE 24

Feature	Description	JEP
• When Clauses • Type Patterns with Primitive Types • Primitive Types in Record Patterns • Primitive Values in switch Expressions and Statements • Safe Casting with instanceof and switch	First previewed in Java SE 23, this feature is re-previewed for this release. It is unchanged between Java SE 23 and this release.	JEP 488: Primitive Types in Patterns, instanceof, and switch (Second Preview)
Flexible Constructor Bodies	First previewed in Java SE 22 as <i>JEP 447: Statements before super(...) (Preview)</i> and previewed again in Java SE 23 as <i>JEP 482: Flexible Constructor Bodies (Second Preview)</i> . This feature is re-previewed for this release without any significant changes.	JEP 492: Flexible Constructor Bodies (Third Preview)
Module Import Statements	First previewed in Java SE 23, this feature is re-previewed for this release. In this release: • Type-import-on-demand declarations shadow module import declarations. • Modules may declare a transitive dependence on the <code>java.base</code> module. • The <code>java.se</code> module transitively requires the <code>java.base</code> module. Consequently, importing the <code>java.se</code> module imports the entire Java SE API.	JEP 494: Module Import Declarations (Second Preview)
Simple Source Files and Instance Main Methods	First previewed in Java SE 21 as <i>JEP 445: Unnamed Classes and Instance Main Methods (Preview)</i> and previewed again in Java SE 23. This feature is re-previewed for this release with new terminology and a revised title but otherwise unchanged.	JEP 495: Simple Source Files and Instance Main Methods (Fourth Preview)

Java Language Changes for Java SE 23

Feature	Description	JEP
• When Clauses • Type Patterns with Primitive Types • Primitive Types in Record Patterns • Primitive Values in switch Expressions and Statements • Safe casting with instanceof and switch	Introduced as a preview feature for this release. In this release: • Primitive type patterns are allowed in all pattern contexts. • The instanceof operator and switch expressions and statements work with all primitive types.	JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)
Module Import Declarations	Introduced as a preview feature for this release. In this release, you can succinctly import all of the packages exported by a module with a module import declaration.	JEP 476: Module Import Declarations (Preview)
Implicitly Declared Classes and Instance Main Methods	First previewed in Java SE 21 as <i>JEP 445: Unnamed Classes and Instance Main Methods (Preview)</i>, this feature is re-previewed for this release. In this release: • Implicitly declared classes automatically import the static methods from the new class <code>java.io.IO</code> for simple textual I/O with the console. • Implicitly declared classes automatically import, on demand, all of the public top-level classes and interfaces of the packages exported by the <code>java.base</code> module.	JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
Flexible Constructor Bodies	First previewed in Java SE 22 as <i>JEP 447: Statements before super(...) (Preview)</i>, this feature is re-previewed for this release with a new title. In this release, a constructor body may initialize fields in the same class before explicitly invoking a constructor. This enables a constructor in a subclass to ensure that a constructor in a superclass never runs code that sees the default value of a field in the subclass (for example, <code>0</code>, <code>false</code>, or <code>null</code>). This can occur when, due to overriding, the superclass constructor invokes a method in the subclass that uses the field.	JEP 482: Flexible Constructor Bodies (Second Preview)

Note

String Templates were first previewed in JDK 21 (JEP 430) and re-previewed in JDK 22 (JEP 459). String Templates were intended to re-preview again in JDK 23 (JEP 465). However, after feedback and extensive discussion, we concluded that the feature is unsuitable in its current form. There is no consensus on what a better design will be; therefore, we have withdrawn the feature for now, and JDK 23 will not include it.

See [March 2024 Archives by thread](#) and [Update on String Templates \(JEP 459\)](#) from the Project Amber `amber-spec-experts` mailing list for further discussion.

Java Language Changes for Java SE 22

Feature	Description	JEP
Statements Before super(...)	Introduced as a preview feature for this release. In constructors in the Java programming language, you may add statements that don't reference the instance being created before an explicit constructor invocation.	JEP 447: Statements before super(...) (Preview)
Unnamed Variables and Patterns	First previewed in Java SE 21 as <i>Unnamed Patterns and Variables</i> , this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 22 without enabling preview features. This feature remains unchanged since Java SE 21.	JEP 456: Unnamed Variables & Patterns
String Templates	Preview feature from Java SE 21 re-previewed for this release. Except for a technical change in the types of template expressions, which is described in String Templates (Second Preview) in <i>The Java Language Specification: Java SE 22 Edition</i> , this feature remains unchanged since Java SE 21.	JEP 459: String Templates (Second Preview)

Feature	Description	JEP
Implicitly Declared Classes and Instance Main Methods	First previewed in Java SE 21 as <i>JEP 445: Unnamed Classes and Instance Main Methods (Preview)</i>, this feature is re- previewed for this release as <i>JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)</i>. In this release: • A source file without an enclosing class declaration is said to implicitly declare a class with a name chosen by the host system. • The procedure for selecting a main method is simplified. If there is a candidate main method with a <code>String[]</code> parameter then we invoke that method; otherwise we invoke a candidate main method with no parameters.	JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)

Java Language Changes for Java SE 21

Feature	Description	JEP
Record Patterns	First previewed in Java SE 19, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 21 without enabling preview features. In this release, support for record patterns appearing in the header of an enhanced for statement has been removed.	JEP 440: Record Patterns
Pattern Matching for switch Expressions and Statements	First previewed in Java SE 17, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 21 without enabling preview features. In this release: • Parenthesized patterns have been removed. • Qualified enum constants as case constants in switch expressions and statements are allowed.	JEP 441: Pattern Matching for switch

Feature	Description	JEP
String Templates	Introduced as a preview feature for this release. String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and template processors to produce specialized results.	JEP 430: String Templates (Preview)
Unnamed Patterns and Variables	Introduced as a preview feature for this release. Unnamed patterns match a record component without stating the component's name or type. Unnamed variables are variables that can be initialized but not used. You denote both with the underscore character (_).	JEP 443: Unnamed Patterns and Variables (Preview)
Unnamed Classes and Instance Main Methods	Introduced as a preview feature for this release. Unnamed classes and instance main methods enable students to write streamlined declarations for single-class programs and then seamlessly expand their programs later to use more advanced features as their skills grow.	JEP 445: Unnamed Classes and Instance Main Methods (Preview)

Java Language Changes for Java SE 20

Feature	Description	JEP
Pattern Matching for switch Expressions and Statements	Preview feature from Java SE 17 re-previewed for this release. In this release: - An exhaustive switch (that is, a switch expression or a pattern switch statement) over an enum class throws a <code>MatchException</code> instead of an <code>IncompatibleClassChangeError</code> if no switch label applies at run time. - The grammar for switch labels is simpler. - The compiler can infer the type of the type arguments for generic record patterns in all constructs that accept patterns: switch statements and expressions, instanceof expressions, and enhanced for statements.	JEP 433: Pattern Matching for switch (Fourth Preview)
Record Patterns	Preview feature from Java SE 19 re-previewed for this release. In this release: - The compiler can infer the type of the type arguments for generic record patterns. - Record patterns can appear in an enhanced for statement. - Named record patterns are no longer supported.	JEP 432: Record Patterns (Second Preview)

Java Language Changes for Java SE 19

Feature	Description	JEP
Pattern Matching for switch Expressions and Statements	Preview feature from Java SE 17 re-previewed for this release. In this release: • The syntax of a guarded pattern label consists of a pattern and a when clause. • If a selector expression evaluates to null and the switch block does not have null case label, then a <code>NullPointerException</code> is thrown, even if a pattern label can match the type of the null value. • If a switch expression or statement is exhaustive at compile time but <i>not</i> at run time, then a <code>MatchException</code> is thrown.	JEP 427: Pattern Matching for switch (Third Preview)
Record Patterns	Introduced as a preview feature for this release. A record pattern consists of a type, a record component pattern list used to match against the corresponding record components, and an optional identifier. You can nest record patterns and type patterns to enable a powerful, declarative, and composable form of data navigation and processing.	JEP 405: Record Patterns (Preview)

Java Language Changes for Java SE 18

Feature	Description	JEP
Pattern Matching for switch Expressions and Statements	Preview feature from Java SE 17 re-previewed for this release. In this release: • Dominance checking forces a constant label to appear before a guarded pattern labels, which must appear before a non-guarded type pattern label; see the section "Pattern Label Dominance" in Pattern Matching for switch Expressions and Statements. • Exhaustiveness checking has been expanded to take into account generic sealed classes and to check switch expressions; see the section "Type Coverage in switch Expressions and Statements" in Pattern Matching for switch Expressions and Statements and the section "Exhaustiveness of switch Statements" in Switch Expressions.	JEP 420: Pattern Matching for switch (Second Preview)

Java Language Changes for Java SE 17

Feature	Description	JEP
Sealed Classes	First previewed in Java SE 15, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 17 without enabling preview features. A sealed class or interface restricts which classes or interfaces can extend or implement it.	JEP 409: Sealed Classes
Pattern Matching for switch Expressions and Statements	Introduced as a preview feature for this release. Pattern matching for switch expressions and statements allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely.	JEP 406: Pattern Matching for switch (Preview)

Java Language Changes for Java SE 16

Feature	Description	JEP
Sealed Classes	Preview feature from Java SE 15 re-previewed for this release. It has been enhanced with several refinements, including more strict checking of narrowing reference conversions with respect to sealed type hierarchies. A sealed class or interface restricts which classes or interfaces can extend or implement it.	JEP 397: Sealed Classes (Second Preview)
Record Classes	First previewed in Java SE 14, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 16 without enabling preview features. In this release, inner classes may declare members that are either explicitly or implicitly static. This includes record class members, which are implicitly static. A record is a class that acts as transparent carrier for immutable data.	JEP 395: Records
Pattern Matching for instanceof	First previewed in Java SE 14, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 16 without enabling preview features. In this release, pattern variables are no longer implicitly final, and it's a compile-time error if a pattern <code>instanceof</code> expression compares an expression of type <i>S</i> with a pattern of type <i>T</i>, where <i>S</i> is a subtype of <i>T</i>. Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely.	JEP 394: Pattern Matching for instanceof

Java Language Changes for Java SE 15

Feature	Description	JEP
Sealed Classes	Introduced as a preview feature for this release. A sealed class or interface restricts which classes or interfaces can extend or implement it.	JEP 360: Sealed Classes (Preview)
Record Classes	Preview feature from Java SE 14 re-previewed for this release. It has been enhanced with support for local records. A record is a class that acts as transparent carrier for immutable data.	JEP 384: Records (Second Preview)
Pattern Matching for instanceof	Preview feature from Java SE 14 re-previewed for this release. It is unchanged between Java SE 14 and this release. Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely.	JEP 375: Pattern Matching for instanceof (Second Preview)
Text Blocks See also Programmer's Guide to Text Blocks	First previewed in Java SE 13, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 15 without enabling preview features. A text block is a multiline string literal that avoids the need for most escape sequences, automatically formats the string in a predictable way, and gives the developer control over the format when desired.	JEP 378: Text Blocks

Java Language Changes for Java SE 14

Feature	Description	JEP
Pattern Matching for the instanceof Operator	Introduced as a preview feature for this release. Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely.	JEP 305: Pattern Matching for instanceof (Preview)
Records	Introduced as a preview feature for this release. Records provide a compact syntax for declaring classes which are transparent holders for shallowly immutable data.	JEP 359: Records (Preview)
Switch Expressions	First previewed in Java SE 12, this feature is permanent in this release. This means that it can be used in any program compiled for Java SE 14 without needing to enable preview features. This feature extends <code>switch</code> so it can be used as either a statement or an expression, and so that both forms can use either traditional <code>case ... : labels</code> (with fall through) or new <code>case ... -> labels</code> (with no fall through), with a further new statement for yielding a value from a <code>switch</code> expression.	JEP 361: Switch Expressions

Feature	Description	JEP
Text Blocks See also Programmer's Guide to Text Blocks	Preview feature from Java SE 13 re-previewed for this release. It has been enhanced with support for more escape sequences. A text block is a multiline string literal that avoids the need for most escape sequences, automatically formats the string in a predictable way, and gives the developer control over the format when desired.	JEP 368: Text Blocks (Second Preview)

Java Language Changes for Java SE 13

Feature	Description	JEP
Text Blocks, see Programmer's Guide to Text Blocks	Introduced as a preview feature for this release. A text block is a multi-line string literal that avoids the need for most escape sequences, automatically formats the string in a predictable way, and gives the developer control over format when desired.	JEP 355: Text Blocks (Preview)
Switch Expressions	Preview feature from Java SE 12 re-previewed for this release. It has been enhanced with one change: To specify the value of a switch expression, use the new yield statement instead of the break statement. This feature extends switch so it can be used as either a statement or an expression, and so that both forms can use either traditional case ... : labels (with fall through) or new case ... -> labels (with no fall through), with a further new statement for yielding a value from a switch expression. .	JEP 354: Switch Expressions (Second Preview)

Java Language Changes for Java SE 12

Feature	Description	JEP
Switch Expressions	Introduced as a preview feature for this release. This feature extends the switch statement so that it can be used as either a statement or an expression, and that both forms can use either a "traditional" or "simplified" scoping and control flow behavior.	JEP 325: Switch Expressions (Preview)

Java Language Changes for Java SE 11

Feature	Description	JEP
Local Variable Type Inference See also Local Variable Type Inference: Style Guidelines	Introduced in Java SE 10. In this release, it has been enhanced with support for allowing var to be used when declaring the formal parameters of implicitly typed lambda expressions. Local-Variable Type Inference extends type inference to declarations of local variables with initializers.	• JEP 286: Local-Variable Type Inference • JEP 323: Local-Variable Syntax for Lambda Parameters

Java Language Changes for Java SE 10

Feature	Description	JEP
Local Variable Type Inference	Introduced in this release.	JEP 286: Local-Variable Type Inference
See also Local Variable Type Inference: Style Guidelines	Local-Variable Type Inference extends type inference to declarations of local variables with initializers.	

Java Language Changes for Java SE 9

Feature	Description	JEP
Java Platform module system, see Project Jigsaw on OpenJDK.	Introduced in this release. The Java Platform module system introduces a new kind of Java programming component, the module, which is a named, self-describing collection of code and data. Its code is organized as a set of packages containing types, that is, Java classes and interfaces; its data includes resources and other kinds of static information. Modules can either export or encapsulate packages, and they express dependencies on other modules explicitly.	Java Platform Module System (JSR 376) JEP 261: Module System JEP 200: The Modular JDK JEP 220: Modular Run-Time Images JEP 260: Encapsulate Most Internal APIs
Small language enhancements (Project Coin): More Concise try-with-resources Statements @SafeVarargs Annotation Allowed on Private Instance Methods Diamond Syntax and Anonymous Inner Classes Underscore Character Not Legal Name Support for Private Interface Methods	Introduced in Java SE 7 as Project Coin. It has been enhanced with a few amendments.	JEP 213: Milling Project Coin JSR 334: Small Enhancements to the Java Programming Language

More Concise try-with-resources Statements

If you already have a resource as a `final` or `effectively final` variable, you can use that variable in a `try-with-resources` statement without declaring a new variable. An "effectively final" variable is one whose value is never changed after it is initialized.

For example, you declared these two resources:

```
// A final resource
final Resource resource1 = new Resource("resource1");
// An effectively final resource
Resource resource2 = new Resource("resource2");
```

In Java SE 7 or 8, you would declare new variables, like this:

```
try (Resource r1 = resource1;  
    Resource r2 = resource2) {  
    ...  
}
```

In Java SE 9, you don't need to declare r1 and r2:

```
// New and improved try-with-resources statement in Java SE 9  
try (resource1;  
    resource2) {  
    ...  
}
```

There is a more complete description of [the try-with-resources statement](#) in The Java Tutorials (Java SE 8 and earlier).

@SafeVarargs Annotation Allowed on Private Instance Methods

The @SafeVarargs annotation is allowed on private instance methods. It can be applied only to methods that cannot be overridden. These include static methods, final instance methods, and, new in Java SE 9, private instance methods.

Diamond Syntax and Anonymous Inner Classes

You can use diamond syntax in conjunction with anonymous inner classes. Types that can be written in a Java program, such as int or String, are called denotable types. The compiler-internal types that cannot be written in a Java program are called non-denotable types.

Non-denotable types can occur as the result of the inference used by the diamond operator. Because the inferred type using diamond with an anonymous class constructor could be outside of the set of types supported by the signature attribute in class files, using the diamond with anonymous classes was not allowed in Java SE 7.

Underscore Character Not Legal Name

If you use the underscore character ("_") as an identifier, your source code can no longer be compiled.

Support for Private Interface Methods

Private interface methods are supported. This support allows nonabstract methods of an interface to share code between them.

3

Preview Features

A preview feature is a new feature whose design, specification, and implementation are complete, but which is not permanent, which means that the feature may exist in a different form or not at all in future JDK releases.

Introducing a feature as a preview feature in a mainline JDK release enables the largest developer audience possible to try the feature out in the real world and provide feedback. In addition, tool vendors are encouraged to build support for the feature before Java developers use it in production. Developer feedback helps determine whether the feature has any design mistakes, which includes hard technical errors (such as a flaw in the type system), soft usability problems (such as a surprising interaction with an older feature), or poor architectural choices (such as one that forecloses on directions for future features). Through this feedback, the feature's strengths and weaknesses are evaluated to determine if the feature has a long-term role in the Java SE Platform, and if so, whether it needs refinement. Consequently, the feature may be granted final and permanent status (with or without refinements), or undergo a further preview period (with or without refinements), or else be removed.

Every preview feature is described by a JDK Enhancement Proposal (JEP) that defines its scope and sketches its design. For example, [JEP 325](#) describes the JDK 12 preview feature for switch expressions. For background information about the role and lifecycle of preview features, see [JEP 12](#).

Using Preview Features

To use preview language features in your programs, you must explicitly enable them in the compiler and the runtime system. If not, you'll receive an error message that states that your code is using a preview feature and preview features are disabled by default.

To compile source code with `javac` that uses preview features from JDK release *n*, use `javac` from JDK release *n* with the `--enable-preview` command-line option in conjunction with either the `--release n` or `-source n` command-line option.

For example, suppose you have an application named `MyApp.java` that uses the JDK 12 preview language feature switch expressions. Compile this with JDK 12 as follows:

```
javac --enable-preview --release 12 MyApp.java
```

Note

When you compile an application that uses preview features, you'll receive a warning message similar to the following:

Note: MyApp.java uses preview language features.
Note: Recompile with -Xlint:preview for details

Remember that preview features are subject to change and are intended to provoke feedback.

To run an application that uses preview features from JDK release n , use `java` from JDK release n with the `--enable-preview` option. To continue the previous example, to run `MyApp`, run `java` from JDK 12 as follows:

```
java --enable-preview MyApp
```

Note

Code that uses preview features from an older release of the Java SE Platform will not necessarily compile or run on a newer release.

The tools `jshell` and `javadoc` also support the `--enable-preview` command-line option.

Sending Feedback

You can provide feedback on preview features, or anything else about the Java SE Platform, as follows:

- If you find any bugs, then submit them at [Java Bug Database](#).
- If you want to provide substantive feedback on the usability of a preview feature, then post it on the OpenJDK mailing list where the feature is being discussed. To find the mailing list of a particular feature, see the feature's JEP page and look for the label *Discussion*. For example, on the page [JEP 325: Switch Expressions \(Preview\)](#), you'll find "*Discussion* amber dash dev at openjdk dot java dot net" near the top of the page.
- If you are working on an open source project, then see [Quality Outreach](#) on the OpenJDK Wiki.

4

Module Import Declarations

You can import all packages exported by a module with a single declaration.

For more information about module import declarations, see [JEP 511](#)

Consider the following example, which imports four classes:

```
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FruitMap {
    public static void main(String[] args) {
        String[] fruits = new String[] { "apple", "berry", "citrus" };
        Map<String, String> m = Stream
            .of(fruits)
            .collect(Collectors.toMap(
                s -> s.toUpperCase().substring(0,1),
                Function.identity()));
        m.forEach((k, v) ->
            System.out.println(k + " " + v));
    }
}
```

You can replace the four single-type-import declarations in this example with type-import-on-demand declarations. However, you still need three of them:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
```

Because the module `java.base` exports the packages `java.util`, `java.util.function` and `java.util.stream`, you can replace these three declarations with one module import declaration:

```
import module java.base;
```

A *module import declaration* has the following form:

```
import module M;
```

It imports, on demand, all of the public top-level class and interfaces in the following:

- The packages exported by the module *M* to the current module.
- The packages exported by the modules that are read by the current module due to reading the module *M*. This enables a program to use the API of a module, which might refer to

classes and interfaces from other modules, without having to import all those other modules.

For example, the module import declaration `import module java.sql` has the same effect as `import java.sql.*` plus on-demand imports for the indirect exports of the `java.sql` module, which include the packages `java.logging` and `java.xml`.

Ambiguous Imports

It's possible to import classes with the same simple name from different packages with module import declarations. However, this can lead to compile-time errors. The following example uses both `java.awt.List` and `java.util.List`:

```
import java.awt.Frame;
import java.awt.Label;
import java.awt.List;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Arrays;

public class FruitApp {

    FruitApp(java.util.List<String> fruits) {
        Frame f = new Frame();
        Label l = new Label("Fruits");
        List lst = new List();
        fruits.forEach(i -> lst.add(i));
        l.setBounds(20, 40, 80, 30);
        lst.setBounds(20, 70, 80, 80);
        f.add(l);
        f.add(lst);
        f.setSize(200,200);
        f.setTitle("Fruit");
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public static void main(String args[]) {
        String[] fruits = new String[] { "apple", "berry", "citrus" };
        FruitApp f = new FruitApp(Arrays.asList(fruits));
    }
}
```

Suppose you replace the single-type-import declarations with these module import declarations:

```
import module java.desktop;
import module java.base;
```

You'll get these compile-time errors:

```
FruitApp.java:8: error: reference to Label is ambiguous
    Label l = new Label("Fruits");
    ^
    both interface java.lang.classfile.Label in java.lang.classfile and class
    java.awt.Label in java.awt match
FruitApp.java:8: error: reference to Label is ambiguous
    Label l = new Label("Fruits");
    ^
    both interface java.lang.classfile.Label in java.lang.classfile and class
    java.awt.Label in java.awt match
FruitApp.java:9: error: reference to List is ambiguous
    List lst = new List();
    ^
    both interface java.util.List in java.util and class java.awt.List in
    java.awt match
FruitApp.java:9: error: reference to List is ambiguous
    List lst = new List();
    ^
    both interface java.util.List in java.util and class java.awt.List in
    java.awt match
```

The use of the simple name `Label` is ambiguous. It's contained in `java.lang.classfile` and `java.awt`, which the modules `java.base` and `java.desktop` export, respectively. This issue applies similarly to the simple name `List`.

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration *shadows* the declaration of the enclosing scope.

To resolve these ambiguities, use single-type-import declarations to specify the canonical names of the simple names used in your code. These declarations *shadow* the `Label` and `List` classes imported by the `import` module declarations.

```
import module java.desktop;
import module java.base;
import java.awt.Label;
import java.awt.List;
```

A declaration shadows another declaration when the shadowing declaration has the same name as another declaration in an enclosing scope. You can refer to a declaration that shadows by its simple name but not the declaration that it shadows. As demonstrated in this example, shadowing can also occur with single-type-import declarations.

Type-import-on-demand declarations can also shadow `import` module declarations. For example, you can reduce the number of import statements in the `FruitApp` example by using a type-import-on-demand declarations:

```
import module java.desktop;
import module java.base;
import java.awt.*;
```

Implicitly Declared Classes and the `java.base` Module

Every implicitly declared class (see [Creating Compact Source Files](#)) automatically imports, on demand, all public top-level classes and interfaces in all packages exported by the `java.base` module. It's as if the module import declaration `import module java.base` appears at the beginning of every implicitly declared class instead of the type-import-on-demand declaration `import java.lang.*` at the beginning of every ordinary class. For example, you can simplify the `FruitMap` example with an implicitly declared class as follows:

```
void main() {
    String[] fruits = new String[] { "apple", "berry", "citrus" };
    Map<String, String> m = Stream
        .of(fruits)
        .collect(Collectors.toMap(
            s -> s.toUpperCase().substring(0,1),
            Function.identity()));
    m.forEach((k, v) ->
        System.out.println(k + " " + v));
}
```

Importing the `java.se` Module and Declaring a Transitive Dependence on the `java.base` Module

You can import the `java.se` module to import the entire standard Java API, including all the packages contained in the `java.base` module. Consequently, you can reduce the number of import statements even further in the `FruitApp` example:

```
import module java.se;
import java.awt.*;
```

With this updated list of import statements, compile the `FruitApp` with the `--add-modules java.se` option. The `java.se` module is not part of the default set of root modules for the unnamed module.

Prior to JDK 24, the `java.se` module didn't contain a transitive dependence on the `java.base` module. No module could have a transitive dependence on `java.base` because every module has an implicit dependence on it. However, because module import declarations derive a set of packages to be imported, you can now require the `java.base` module transitively.

5

Flexible Constructor Bodies

In a constructor's body, you may add statements that don't reference the instance being created before an explicit constructor invocation.

For more information about flexible constructor bodies, see [JEP 513](#).

You can use this feature to prepare arguments for a superclass constructor by performing nontrivial computations or to validate arguments you want to pass to a superclass constructor. The following example validates whether the argument value is positive before passing it to the superclass constructor:

```
public class PositiveBigInteger extends BigInteger {  
  
    public PositiveBigInteger(long value) {  
        if (value <= 0)  
            throw new IllegalArgumentException("non-positive value");  
        super(Long.toString(value));  
    }  
}
```

The *prologue* of the constructor's body consists of the statements that appear before the `super(...)` invocation. The *epilogue* of the constructor's body consists of the statements that follow the `super(...)` invocation.

Early Construction Context

The *early construction context* of a constructor consists of the arguments to an explicit constructor invocation, such as `super(...)`, and any statements before it. The *late construction context* consists of the statements after the explicit or implicit `super(...)` call. In the late construction context, the current class can safely access this.

In the previous example, the early construction context of `PositiveBigInteger` consists of the argument `Long.toString(value)` and the `if`-statement that checks whether `value` is positive.

Code in an early construction context may not access the instance under construction. This means you can't have the following in an early construction context:

- **Any unqualified `this` expression:** Note that you don't need to use the `this` keyword to access the instance under construction. For example:

```
class A {  
    int i;  
    A() {  
        // Error: Cannot reference this before supertype constructor has  
        been  
        //      called  
        this.i++;  
  
        // Error: cannot reference i before supertype constructor has been
```

```

        //      called
        i++;

        // Error: cannot reference this before supertype constructor has
been
        //      called
        this.hashCode();

        // Error: cannot reference hashCode() before supertype constructor
has
        //      been called
        hashCode();

        // Error: cannot reference this before supertype constructor has
been
        //      called
        System.out.print(this);
        super();
    }
}

```

- **Any field access, method invocation, or method reference qualified by super:** Again, note that you don't need to use the super keyword to access the superclass of the instance under construction:

```

class D {
    int j;
}

class E extends D {
    E() {
        // Error: cannot reference super before supertype constructor has
been
        //      called
        super.j++;

        // Error: cannot reference j before supertype constructor has been
        //      called
        j++;
        super();
    }
}

```

Initializing Fields Before the super(...) Invocation

As mentioned previously, you can't read any of the fields of the current instance, whether declared in the same class as the constructor or in a superclass, until after the explicit constructor invocation. However, you can initialize fields of the current instance with the assignment operator before the `super(...)` invocation.

Consider the following example, which consists of two classes: `Super` and `Sub`, which extends `Super` and overrides `Super::overriddenMethod`:

```

class Super {
    Super() { overriddenMethod(); }
    void overriddenMethod() { System.out.println("hello"); }
}

```

```
}

class Sub extends Super {

    final int x;

    Sub(int x) {
        // The Super constructor is implicitly invoked,
        // which calls overriddenMethod(), before initializing
        // the field x in Sub.
        this.x = x;
    }

    @Override
    void overriddenMethod() { System.out.println(x); }

    public static void main(String[] args) {
        Sub myApp = new Sub(42);
        myApp.overriddenMethod();
    }
}
```

The example prints the following output:

```
0
42
```

When the example invokes the constructor for Sub, it implicitly invokes the constructor for Super before assigning a value to the field x in Sub. As a result, when the example invokes Sub:overriddenMethod in the constructor for Sub, it prints the uninitialized value of x, which is 0.

You can initialize the field x in Sub and then invoke super () afterward:

```
class BetterSub extends Super {

    final int x;

    BetterSub(int x) {
        // Initialize the int x field in BetterSub before
        // invoking the Super constructor with super().
        this.x = x;
        super();
    }

    @Override
    void overriddenMethod() { System.out.println(x); }

    public static void main(String[] args) {
        BetterSub myApp = new BetterSub(42);
        myApp.overriddenMethod();
    }
}
```

This example prints the following output:

```
42
42
```

Nested Classes

A nested class is a member of its enclosing class, which means you can't access a nested class from its enclosing class's early construction context. For example:

```
class B {
    class C { }

    B() {
        // Error: cannot reference this before supertype constructor has been
        //      called
        new C();
        super();
    }
}
```

However, a nested class's enclosing class is *not* one of its members, which means you can access its enclosing class from its early construction context. In the following example, both accessing F's member variable `f` and method `hello()` in the early construction context of its nested class G is permitted:

```
class F {
    int f;

    void hello() {
        System.out.println("Hello!");
    }

    class G {
        G() {
            F.this.f++;
            hello();
            super();
        }
    }
}
```

Records

Record constructors may not invoke `super(...)`. However, noncanonical constructors must involve a canonical constructor by invoking `this(...)`. Statements may appear before `this(...)`.

Remember that a canonical constructor is a constructor whose signature is the same as the record's component list. It initializes all the component fields of the record class. Alternative or noncanonical record constructors have argument lists that don't match the record's type parameters.

In the following example, the record `RectanglePair` contains a noncanonical constructor, `RectanglePair(Pair<Float> corner)`. Because it's a noncanonical constructor, it must invoke a canonical constructor with `this(...)`. It contains several statements before `this(...)` that retrieve both values from the `Pair<Float>` parameter and validate that these values aren't negative:

```
record Pair<T extends Number>(T x, T y) { }

record RectanglePair(float length, float width) {
    public RectanglePair(Pair<Float> corner) {
        float x = corner.x().floatValue();
        float y = corner.y().floatValue();
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException("non-positive value");
        }
        this(corner.x().floatValue(), corner.y().floatValue());
    }
}
```

See [Alternative Record Constructors](#) in [Record Classes](#) for more information.

Note

In JDK 27, if you have enabled preview features, then like a value class, all of a record's constructor code occurs in the early construction context. Attempts to invoke instance methods or otherwise use `this` results in a compiler error. The following example compiles if preview features are **not** enabled, but doesn't compile if they are enabled:

```
record Bag(String contents) {
    public Bag {
        if (contents == null) {
            throw new IllegalArgumentException("null argument for " +

                // Compiler error with --enable-preview: reference to
                // this may only appear after an explicit constructor
                // invocation
                this.toString());
        }
    }
}
```

6

Compact Source Files and Instance main Methods

Compact source files and instance main methods enable students to write their first programs without needing to understand the full set of language features designed for large programs.

For background information about compact source files and instance main methods, see [JEP 512](#).

The Java programming language excels in developing large, complex applications developed and maintained over many years by large teams. It has rich features for data hiding, reuse, access control, namespace management, and modularity which allow components to be cleanly composed while being developed and maintained independently. The composition of large components is called *programming-in-the-large*.

However, the Java programming language is also intended to be a first language and offers many constructs that are useful for *programming-in-the-small* (everything that is internal to a component). When programmers first start out, they do not write large programs in a team — they write small programs by themselves. At this stage, there is no need for the programming-in-the-large concepts of classes, packages, and modules.

When teaching programming, instructors start with the basic programming-in-the-small concepts of variables, control flow, and subroutines. There is no need for the programming-in-the-large concepts of classes, packages, and modules. Students who are learning to program have no need for encapsulation and namespaces which are useful later to separately evolve components written by different people.

Compact source files and instance main methods enhance the Java programming language's support for programming in the small as follows:

- Enable students and developers to create *compact source files*, which are source files with fields and methods that aren't enclosed in a class declaration. A compact source file *implicitly declares* a class whose members are the unenclosed fields and methods. See [8.1.8 Implicitly Declared Classes](#) in the [Java Language Specification](#) and [Creating Compact Source Files](#).
- Automatically import public top-level classes and interfaces of the packages exported by the `java.base` module. See [Automatic Import of the java.base Module](#).
- Enhance the protocol by which Java programs are launched by allowing instance main methods that are not static, need not be public, and need not have a `String[]` parameter. See [Running Compact Source Files with the java Launcher and Instance Main Methods](#).

In addition, the `java.lang.IO` class provides basic line-oriented I/O methods, which are simpler alternatives to the `System.out.println` method. See [The java.lang.IO Class for Basic Console I/O](#).

Consider the classic HelloWorld program that is often used as the first program for Java students:

```
public class HelloWorld {  
    public static void main(String[] args) {
```

```
        System.out.println("Hello, World!");  
    }  
}
```

In this first program:

- The class declaration and the mandatory public access modifier are programming-in-the-large constructs. They are useful when encapsulating a code unit with a well-defined interface to external components, but rather pointless in this little example.
- The `String[] args` parameter also exists to interface the code with an external component, in this case the operating system's shell. It is mysterious and unhelpful here, especially since it is not used in simple programs like `HelloWorld`.
- The static modifier is part of Java's class-and-object model. For the novice, static is not just mysterious but also harmful. To add more methods or fields that `main` can call and use, the student must either declare them all as `static` (propagating an idiom which is neither a common nor a good habit) or else confront the difference between static and instance members and learn how to instantiate an object.

The new programmer encounters these programming-in-the-large constructs before they learn about variables and control flow and before they can appreciate the utility of programming-in-the-large constructs for keeping a large program well organized. Educators often offer the admonition, "Don't worry about that. You'll understand it later." This is unsatisfying to them and to their students. It leaves students with the enduring impression that the Java language is overly complicated.

Compact source files and instance `main` methods reduce the complexity of writing simple programs such as `HelloWorld` by enabling programmers to write programs without using access modifiers, static modifiers, or the `String[]` parameter. Far from being a separate dialect, students can now use the Java language to write streamlined declarations for single-class programs and then later seamlessly expand their beginning programs to include more advanced features as their skills grow. Java veterans might also find that compact source files and instance `main` methods are useful features when writing simple Java programs that do not require the programming-in-the-large features of the Java language. The introduction of programming-in-the-large constructs can be postponed by instructors until they are needed.

Topics

- [Creating Compact Source Files](#)
- [The `java.lang.IO` Class for Basic Console I/O](#)
- [Automatic Import of the `java.base` Module](#)
- [Running Compact Source Files with the `java` Launcher and Instance `Main` Methods](#)
- [Growing a Program](#)

Creating Compact Source Files

In the Java language, every class resides in a package and every package resides in a module. These namespacing and encapsulation constructs apply to all code. However, small programs that don't need them can omit them.

A program that doesn't need class namespaces can omit the package statement, making its classes implicit members of the unnamed package. Classes in the unnamed package cannot be referenced explicitly by classes in named packages. A program that doesn't need to encapsulate its packages can omit the module declaration, making its packages implicit

members of the unnamed module. Packages in the unnamed module cannot be referenced explicitly by packages in named modules.

Before classes serve their main purpose as templates for the construction of objects, they serve as namespaces for methods and fields. We should not require students to confront the concept of classes:

- Before they are comfortable with the basic building blocks of variables, control flow, and subroutines,
- Before they embark on learning object orientation, and
- When they are still writing simple, single-file programs.

Even though every method resides in a class, we can stop requiring explicit class declarations for code that doesn't need it — just as we don't require explicit package or module declarations for code that don't need them.

When the Java compiler encounters a source file containing a method not enclosed in a class declaration, it considers that method, any similar methods, and any unenclosed fields and classes in the file to form the body of an *implicitly declared* top-level class. A source file containing an implicitly declared class is called a *compact source file*.

An implicitly declared class of a compact source file:

- Is a `final` top level class in the unnamed package.
- Extends `java.lang.Object` and does not implement any interfaces.
- Has a default constructor with no parameters, and no other constructors.
- Has, as its members, the fields and methods in the compact source file.
- Must have a launchable `main` method, which can be an *instance main method*. See [Running Compact Source Files with the java Launcher and Instance Main Methods](#).

An implicitly declared class can't be referenced by name, so there can be no method references to its static methods. However, the `this` keyword can still be used, as well as method references to instance methods.

The code of a compact source file can't refer to its implicitly declared class by name, so instances of an implicitly declared class can't be constructed directly. Such a class is useful only as a standalone program or as an entry point to a program. Therefore, an implicitly declared class must have a `main` method that can be launched as described in [Running Compact Source Files with the java Launcher and Instance Main Methods](#). This requirement is enforced by the Java compiler.

An implicitly declared class resides in the unnamed package, and the unnamed package resides in the unnamed module. While there can only be one unnamed package (barring multiple class loaders) and only one unnamed module, there can be multiple implicitly declared class in the unnamed module. Every implicitly declared class contains a `main` method and represents a program. Consequently, multiple implicitly declared classes in an unnamed package represent multiple programs.

An implicitly declared class is similar to an explicitly declared class. Its members can have the same modifiers (such as `private` and `static`) and the modifiers have the same defaults (such as package access and instance membership). One key difference is that while an implicitly declared class has a default zero-parameter constructor, it can have no other constructor.

With these changes, we can now write the HelloWorld program as a compact source file:

```
void main() {  
    System.out.println("Hello, World!");  
}
```

It has one launchable main method, namely `void main()`, which is an instance main method. Because top-level members are interpreted as members of the implicitly declared class, we can also write the program as:

```
String greeting() { return "Hello, World!"; }  
  
void main() {  
    System.out.println(greeting());  
}
```

Or, by using a field, we can write the program as:

```
String greeting = "Hello, World!";  
  
void main() {  
    System.out.println(greeting);  
}
```

You can launch a compact source file named `HelloWorld.java` with the `java` command-line tool as follows:

```
java HelloWorld.java
```

The Java compiler compiles that file to the launchable class file `HelloWorld.class`. In this case, the compiler chooses `HelloWorld` for the class name as an implementation detail. However, that name still can't be used directly in Java source code. See [Running Compact Source Files with the java Launcher and Instance Main Methods](#).

At this time, the `javadoc` tool can't generate API documentation for an implicitly declared class because implicitly declared classes don't define an API that is accessible from other classes. However, fields and methods of an implicitly declared class can generate API documentation.

The java.lang.IO Class for Basic Console I/O

The [java.lang.IO](#) class provides basic line-oriented I/O methods, which are simpler alternatives to the `System.out.println` method:

- [println\(Object\)](#): Writes a string representation of the specified object to the system console, and then flushes that console.
- [print\(Object\)](#): Writes a string representation of the specified object to the system console, terminates the line, and then flushes that console.
- [readln\(String\)](#): Writes a prompt as if calling `print`, and then reads a single line of text from the system console.

Consequently, you can further simplify the `HelloWorld` program as:

```
String greeting = "Hello, World!";

void main() {
    IO.println(greeting);
}
```

Although the `IO` class is in the `java.lang` package, which means that it's implicitly imported by every source file (and not just compact source files), the static methods of the `IO` class aren't implicitly imported. To invoke these methods, you must name the class, for example, `IO.println("Hello, world!")`, unless you explicitly import them.

Automatic Import of the `java.base` Module

Every compact source file imports, on demand, all public top-level classes and interfaces in all packages exported by the `java.base` module. It is as if the module import declaration `import module java.base` appears at the beginning of every compact source file. See [Module Import Declarations](#) for more information.

The following example is a compact source file that requires no import declarations for `Map`, `Stream`, `Collectors`, or `Function` as they are contained in packages exported by the `java.base` module.

```
void main() {
    String[] fruits = new String[] { "apple", "berry", "citrus" };
    Map<String, String> m = Stream
        .of(fruits)
        .collect(Collectors.toMap(
            s -> s.toUpperCase().substring(0,1),
            Function.identity()));
    m.forEach((k, v) -> println(k + " " + v));
}
```

Running Compact Source Files with the `java` Launcher and Instance Main Methods

You can run a single-file, source-code program, including one that's a compact source file, with the `java` command-line tool without first explicitly compiling it.

For example, if `HelloWorld.java` is a compact source file, you can run it directly at the command line:

```
java HelloWorld.java
```

Alternatively, like any other Java source file, you can explicitly compile a compact source file with the `javac` command-line tool and then run the compiled compact source file with the `java` command-line tool:

```
javac HelloWorld.java
java HelloWorld
```

In both cases, the `java` tool finds and invokes a `main` method as described by the following protocol:

- If the compact source file contains an accessible `main` method with a single `String[]` parameter and a `void` result, then this method is chosen.
Otherwise, if the compact source file contains an accessible `main` method with no parameters and a `void` result, then this method is chosen.
Otherwise, the `java` tool reports an error and terminates.
- If the method chosen was a static `main` method, then it's invoked along with any arguments from the command line passed as elements of an array, as appropriate.
Otherwise, an instance method was chosen. In this case, an instance of the compact source file's implicitly declared class must be created first. This class instance is the result of invoking the default constructor of the implicitly declared class. The `main` method is then invoked on the resulting instance with any arguments from the command line passed as elements of an array, as appropriate.

Any `main` method that can be invoked under this protocol is called a *launchable main method*. For example, consider again the compact source file `HelloWorld.java`:

```
void main() {
    IO.println("Hello, World!");
}
```

It has one launchable `main` method, namely `void main()`, which is an instance `main` method.

The following compact source file example contains an instance `main` method with a `String[]` parameter:

```
void main(String[] args) {
    IO.println("Hello, " + args[0]);
}
```

Growing a Program

By omitting the concepts and constructs it doesn't need, a `HelloWorld` program written as a compact source file is more focused on what the program actually does. Even so, all of the members of the implicitly declared class in a compact source file continue to be interpreted just as they are in an ordinary class.

Concepts and constructs can easily be added to a compact source file as needed by the program. To evolve the implicitly declared class in a compact source file into an ordinary class, all we need to do is wrap its declaration, excluding `import` statements, inside an explicit class declaration.

7

Sealed Classes

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

For background information about sealed classes and interfaces, see [JEP 409](#).

One of the primary purposes of inheritance is code reuse: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug) them yourself.

However, what if you want to model the various possibilities that exist in a domain by defining its entities and determining how these entities should relate to each other? For example, you're working on a graphics library. You want to determine how your library should handle common geometric primitives like circles and squares. You've created a `Shape` class that these geometric primitives can extend. However, you're not interested in allowing any arbitrary class to extend `Shape`; you don't want clients of your library declaring any further primitives. By sealing a class, you can specify which classes are permitted to extend it and prevent any other arbitrary class from doing so.

Declaring Sealed Classes

To seal a class, add the sealed modifier to its declaration. Then, after any extends and implements clauses, add the permits clause. This clause specifies the classes that may extend the sealed class.

For example, the following declaration of `Shape` specifies three permitted subclasses, `Circle`, `Square`, and `Rectangle`:

Figure 7-1 `Shape.java`

```
public sealed class Shape
    permits Circle, Square, Rectangle {
}
```

Define the following three permitted subclasses, `Circle`, `Square`, and `Rectangle`, in the same module or in the same package as the sealed class:

Figure 7-2 `Circle.java`

```
public final class Circle extends Shape {
    public float radius;
}
```

Figure 7-3 Square.java

Square is a *non-sealed class*. This type of class is explained in [Constraints on Permitted Subclasses](#).

```
public non-sealed class Square extends Shape {  
    public double side;  
}
```

Figure 7-4 Rectangle.java

```
public sealed class Rectangle extends Shape permits FilledRectangle {  
    public double length, width;  
}
```

Rectangle has a further subclass, FilledRectangle:

Figure 7-5 FilledRectangle.java

```
public final class FilledRectangle extends Rectangle {  
    public int red, green, blue;  
}
```

Alternatively, you can define permitted subclasses in the same file as the sealed class. If you do so, then you can omit the `permits` clause:

```
package com.example.geometry;  
  
public sealed class Figure  
    // The permits clause has been omitted  
    // as its permitted classes have been  
    // defined in the same file.  
{ }  
  
final class Circle extends Figure {  
    float radius;  
}  
non-sealed class Square extends Figure {  
    float side;  
}  
sealed class Rectangle extends Figure {  
    float length, width;  
}  
final class FilledRectangle extends Rectangle {  
    int red, green, blue;  
}
```

Constraints on Permitted Subclasses

Permitted subclasses have the following constraints:

- They must be accessible by the sealed class at compile time.

For example, to compile `Shape.java`, the compiler must be able to access all of the permitted classes of `Shape`: `Circle.java`, `Square.java`, and `Rectangle.java`. In addition, because `Rectangle` is a sealed class, the compiler also needs access to `FilledRectangle.java`.

- They must directly extend the sealed class.
- They must have exactly one of the following modifiers to describe how it continues the sealing initiated by its superclass:
 - `final`: Cannot be extended further
 - `sealed`: Can only be extended by its permitted subclasses
 - `non-sealed`: Can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this

For example, the permitted subclasses of `Shape` demonstrate each of these three modifiers: `Circle` is `final` while `Rectangle` is `sealed` and `Square` is `non-sealed`.

- They must be in the same module as the sealed class (if the sealed class is in a named module) or in the same package (if the sealed class is in the unnamed module, as in the `Shape.java` example).

For example, in the following declaration of `com.example.graphics.Shape`, its permitted subclasses are all in different packages. This example will compile only if `Shape` and all of its permitted subclasses are in the same named module.

```
package com.example.graphics;

public sealed class Shape
    permits com.example.polar.Circle,
           com.example.quad.Rectangle,
           com.example.quad.simple.Square { }
```

Declaring Sealed Interfaces

Like sealed classes, to seal an interface, add the `sealed` modifier to its declaration. Then, after any `extends` clause, add the `permits` clause, which specifies the classes that can implement the sealed interface and the interfaces that can extend the sealed interface.

The following example declares a sealed interface named `Expr`. Only the classes `ConstantExpr`, `PlusExpr`, `TimesExpr`, and `NegExpr` may implement it:

```
package com.example.expressions;

public class TestExpressions {
    public static void main(String[] args) {
        // (6 + 7) * -8
        System.out.println(
            new TimesExpr(
                new PlusExpr(new ConstantExpr(6), new ConstantExpr(7)),
                new NegExpr(new ConstantExpr(8))
            ).eval());
    }
}

sealed interface Expr
```

```

        permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {
        public int eval();
    }

    final class ConstantExpr implements Expr {
        int i;
        ConstantExpr(int i) { this.i = i; }
        public int eval() { return i; }
    }

    final class PlusExpr implements Expr {
        Expr a, b;
        PlusExpr(Expr a, Expr b) { this.a = a; this.b = b; }
        public int eval() { return a.eval() + b.eval(); }
    }

    final class TimesExpr implements Expr {
        Expr a, b;
        TimesExpr(Expr a, Expr b) { this.a = a; this.b = b; }
        public int eval() { return a.eval() * b.eval(); }
    }

    final class NegExpr implements Expr {
        Expr e;
        NegExpr(Expr e) { this.e = e; }
        public int eval() { return -e.eval(); }
    }

```

Record Classes as Permitted Subclasses

You can name a record class in the permits clause of a sealed class or interface. See [Record Classes](#) for more information.

Record classes are implicitly final, so you can implement the previous example with record classes instead of ordinary classes:

```

package com.example.records.expressions;

public class TestExpressions {
    public static void main(String[] args) {
        // (6 + 7) * -8
        System.out.println(
            new TimesExpr(
                new PlusExpr(new ConstantExpr(6), new ConstantExpr(7)),
                new NegExpr(new ConstantExpr(8))
            ).eval());
    }
}

sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {
    public int eval();
}

record ConstantExpr(int i) implements Expr {
    public int eval() { return i(); }
}

```

```
}

record PlusExpr(Expr a, Expr b) implements Expr {
    public int eval() { return a.eval() + b.eval(); }
}

record TimesExpr(Expr a, Expr b) implements Expr {
    public int eval() { return a.eval() * b.eval(); }
}

record NegExpr(Expr e) implements Expr {
    public int eval() { return -e.eval(); }
}
```

Narrowing Reference Conversion and Disjoint Types

Narrowing reference conversion is one of the conversions used in type checking cast expressions. It enables an expression of a reference type *S* to be treated as an expression of a different reference type *T*, where *S* is not a subtype of *T*. 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, 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.

Consider the following example:

```
public interface Polygon { }
public class Rectangle implements Polygon { }

public void work(Rectangle r) {
    Polygon p = (Polygon) r;
}
```

The cast expression `Polygon p = (Polygon) r` is allowed because it's possible that the `Rectangle` value `r` could be of type `Polygon`; `Rectangle` is a subtype of `Polygon`. However, consider this example:

```
public interface Polygon { }
public class Triangle { }

public void work(Triangle t) {
    Polygon p = (Polygon) t;
}
```

Even though the class `Triangle` and the interface `Polygon` are unrelated, the cast expression `Polygon p = (Polygon) t` is also allowed because at run time these types could be related. A developer could declare the following class:

```
class MeshElement extends Triangle implements Polygon { }
```

However, there are cases where the compiler can deduce that there are no values (other than the null reference) shared between two types; these types are considered *disjoint*. For example:

```
public interface Polygon { }
public final class UtahTeapot { }

public void work(UtahTeapot u) {
    Polygon p = (Polygon) u; // Error: The cast can never succeed as
                           // UtahTeapot and Polygon are disjoint
}
```

Because the class `UtahTeapot` is `final`, it's impossible for a class to be a descendant of both `Polygon` and `UtahTeapot`. Therefore, `Polygon` and `UtahTeapot` are disjoint, and the cast statement `Polygon p = (Polygon) u` isn't allowed.

The compiler has been enhanced to navigate any sealed hierarchy to check if your cast statements are allowed. For example:

```
public sealed interface Shape permits Polygon { }
public non-sealed interface Polygon extends Shape { }
public final class UtahTeapot { }
public class Ring { }

public void work(Shape s) {
    UtahTeapot u = (UtahTeapot) s; // Error
    Ring r = (Ring) s;             // Permitted
}
```

The first cast statement `UtahTeapot u = (UtahTeapot) s` isn't allowed; a `Shape` can only be a `Polygon` because `Shape` is sealed. However, as `Polygon` is non-sealed, it can be extended. However, no potential subtype of `Polygon` can extend `UtahTeapot` as `UtahTeapot` is `final`. Therefore, it's impossible for a `Shape` to be a `UtahTeapot`.

In contrast, the second cast statement `Ring r = (Ring) s` is allowed; it's possible for a `Shape` to be a `Ring` because `Ring` is not a `final` class.

APIs Related to Sealed Classes and Interfaces

The class `java.lang.Class` has two new methods related to sealed classes and interfaces:

- `java.lang.constant.ClassDesc[] permittedSubclasses()`: Returns an array containing `java.lang.constant.ClassDesc` objects representing all the permitted subclasses of the class if it is sealed; returns an empty array if the class is not sealed
- `boolean isSealed()`: Returns true if the given class or interface is sealed; returns false otherwise

8

Primitive Types in Patterns, instanceof, and switch

You can use primitive types in all pattern contexts, and the `instanceof` operator and `switch` statements and expressions work with all primitive types. See the following sections for more information:

- [Type Patterns with Primitive Types](#)
- [Primitive Types in Record Patterns](#)
- [Primitive Values in switch Expressions and Statements](#)
- [Safe Casting with instanceof and switch](#): You can use the `instanceof` operator to test whether a cast is safe between any two types, and not only between two reference types.
- [When Clauses](#): You can use guards that restrict values of primitive types that come from a top-level pattern. A *guard* is a boolean expression in a `when` clause that follows a pattern label.

Note

Using primitive types in patterns, `instanceof`, and `switch` is a preview feature. This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [Preview Language and VM Features](#).

See [JEP 532: Primitive Types in Patterns, instanceof, and switch \(Fifth Preview\)](#) for additional information.

9

Pattern Matching

Pattern matching involves testing whether an object has a particular structure, then extracting data from that object if there's a match. You can already do this with Java. However, pattern matching introduces new language enhancements that enable you to conditionally extract data from objects with code that's more concise and robust.

A *pattern* describes a test that can be performed on a value. Patterns appear as operands of statements and expressions, which provide the values to be tested. For example, consider this expression:

```
s instanceof Rectangle r
```

The pattern `Rectangle r` is an operand of the `instanceof` expression. It's testing if the argument `s` has the type given in the pattern, which is `Rectangle`. Sometimes, the argument that a pattern tests is called the *target*.

① Note

When the operand to the right of `instanceof` is a pattern, like the previous example, then `instanceof` is the *pattern match operator*.

When the operand to the right of `instanceof` is a type, then `instanceof` is the *type comparison operator*. The following example uses `instanceof` as the type comparison operator:

```
s instanceof Rectangle
```

A pattern can declare zero or more *pattern variables*. For example, the pattern `Rectangle r` declares only one, `r`.

The process of testing a value against a pattern is called *pattern matching*. If a value successfully matches a pattern, then the pattern variables are initialized with data from the target. In this example, if `s` is a `Rectangle`, then `s` is converted to a `Rectangle` and then assigned to `r`.

Patterns can also appear in the case labels of a switch statement or expression. For example:

```
public static double getArea(Shape s) throws IllegalArgumentException {
    switch (s) {
        case Rectangle r:
            return r.length() * r.width();
        case Circle c:
            return c.radius() * c.radius() * Math.PI;
        default:
            throw new IllegalArgumentException("Unrecognized shape");
    }
}
```

```
    }
}
```

A *type pattern* consists of a type along with a single pattern variable. In this example, `Rectangle r` is a type pattern.

A *record pattern* consists of a record type and a (possibly empty) record pattern list. For example, consider this record declaration and expression:

```
record Point(double x, double y) {}
// ...

obj instanceof Point(double a, double b)
```

The record pattern `Point(double x, double y)` tests whether the target `obj` is a `Point(double, double)`. If so, it extracts the `x` and `y` values from `obj` directly and assigns them to the pattern variables `a` and `b`, respectively.

Topics

- [Pattern Matching with instanceof](#)
- [Pattern Matching with switch](#)
- [Type Patterns](#)
- [Record Patterns](#)

Pattern Matching with instanceof

Pattern matching with `instanceof` involves testing whether a target's type matches the type in the pattern. If so, the target is converted to the type in the pattern, and then the pattern variables are automatically initialized with data from the target.

For background information about pattern matching for the `instanceof` operator, see [JEP 394](#).

Consider the following code that calculates the perimeter of certain shapes:

```
public interface Shape {
    public static double getPerimeter(Shape s) throws
        IllegalArgumentException {
        if (s instanceof Rectangle) {
            Rectangle r = (Rectangle) s;
            return 2 * r.length() + 2 * r.width();
        } else if (s instanceof Circle) {
            Circle c = (Circle) s;
            return 2 * c.radius() * Math.PI;
        } else {
            throw new IllegalArgumentException("Unrecognized shape");
        }
    }
}

public class Rectangle implements Shape {
    final double length;
    final double width;
    public Rectangle(double length, double width) {
```

```

        this.length = length;
        this.width = width;
    }
    double length() { return length; }
    double width() { return width; }
}

public class Circle implements Shape {
    final double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    double radius() { return radius; }
}

```

The method `getPerimeter` performs the following:

1. A test to determine the type of the `Shape` object
2. A conversion, casting the `Shape` object to `Rectangle` or `Circle`, depending on the result of the `instanceof` operator
3. A destructuring, extracting either the length and width or the radius from the `Shape` object

Pattern matching enables you to remove the conversion step by changing the second operand of the `instanceof` operator with a type pattern, making your code shorter and easier to read:

```

public static double getPerimeter(Shape shape) throws
IllegalArgumentException {
    if (s instanceof Rectangle r) {
        return 2 * r.length() + 2 * r.width();
    } else if (s instanceof Circle c) {
        return 2 * c.radius() * Math.PI;
    } else {
        throw new IllegalArgumentException("Unrecognized shape");
    }
}

```

① Note

Removing this conversion step also makes your code safer. Testing an object's type with `instanceof` and then assigning that object to a new variable with a cast can introduce coding errors in your application. You might change the type of one of the objects (either the tested object or the new variable) and accidentally forget to change the type of the other object. See [Safe Casting with instanceof and switch](#) for more information.

This example uses two patterns:

- `Rectangle r`
- `Circle c`

The pattern `Rectangle r` is an operand of the `instanceof` expression. It's testing if its target `s` has the type given in the pattern, which is `Rectangle`. Similarly, the expression `s instanceof Circle c` tests if `s` has the type `Circle`.

If a value successfully matches a pattern, then the pattern variables are initialized with data from the target. In this example, if the target `s` is a `Rectangle`, then `s` is converted to a `Rectangle` and then assigned to `r`. Similarly, if `s` is a `Circle`, then it's converted to a `Circle` and then assigned to `c`,

Scope of Pattern Variables and instanceof

The scope of a pattern variable is the places where the program can reach only if the `instanceof` operator is true:

```
public static double getPerimeter(Shape shape) throws
IllegalArgumentException {
    if (shape instanceof Rectangle s) {
        // You can use the pattern variable s (of type Rectangle) here.
    } else if (shape instanceof Circle s) {
        // You can use the pattern variable s of type Circle here
        // but not the pattern variable s of type Rectangle.
    } else {
        // You cannot use either pattern variable here.
    }
}
```

The scope of a pattern variable can extend beyond the statement that introduced it:

```
public static boolean bigEnoughRect(Shape s) {
    if (!(s instanceof Rectangle r)) {
        // You cannot use the pattern variable r here because
        // the predicate s instanceof Rectangle is false.
        return false;
    }
    // You can use r here.
    return r.length() > 5;
}
```

You can use a pattern variable in the expression of an `if` statement:

```
if (shape instanceof Rectangle r && r.length() > 5) {
    // ...
}
```

Because the conditional-AND operator (`&&`) is short-circuiting, the program can reach the `r.length() > 5` expression only if the `instanceof` operator is true.

Conversely, you can't pattern match with the `instanceof` operator in this situation:

```
if (shape instanceof Rectangle r || r.length() > 0) { // error
    // ...
}
```

The program can reach the `r.length() || 5` if the `instanceof` is false; thus, you cannot use the pattern variable `r` here.

See [Scope of Pattern Variables and switch](#) for more examples of where you can use a pattern variable.

Pattern Matching with switch

A switch expression or statement transfers control to one of several statements or expressions, depending on the value of its selector expression. The selector expression can be any reference or primitive type.

Also, case labels can have patterns. Consequently, a switch expression or statement can test whether its selector expression matches a pattern, which offers more flexibility and expressiveness compared to testing whether its selector expression is exactly equal to a constant.

For background information about pattern matching for switch expressions and statements, see [JEP 441](#).

Consider the following code that calculates the perimeter of certain shapes from the section [Pattern Matching with instanceof](#):

```
interface Shape { }
record Rectangle(double length, double width) implements Shape { }
record Circle(double radius) implements Shape { }
// ...

public static double getPerimeter(Shape s) throws IllegalArgumentException {
    if (s instanceof Rectangle r) {
        return 2 * r.length() + 2 * r.width();
    } else if (s instanceof Circle c) {
        return 2 * c.radius() * Math.PI;
    } else {
        throw new IllegalArgumentException("Unrecognized shape");
    }
}
```

You can rewrite this code to use a pattern switch expression as follows:

```
public static double getPerimeter(Shape s) throws IllegalArgumentException {
    return switch (s) {
        case Rectangle r ->
            2 * r.length() + 2 * r.width();
        case Circle c ->
            2 * c.radius() * Math.PI;
        default ->
            throw new IllegalArgumentException("Unrecognized shape");
    };
}
```

The following example uses a switch statement instead of a switch expression:

```
public static double getPerimeter(Shape s) throws IllegalArgumentException {
    switch (s) {
        case Rectangle r:
            return 2 * r.length() + 2 * r.width();
        case Circle c:
            return 2 * c.radius() * Math.PI;
    }
}
```

```

        return 2 * c.radius() * Math.PI;
    default:
        throw new IllegalArgumentException("Unrecognized shape");
    }
}

```

Topics

- [When Clauses](#)
- [Pattern Label Dominance](#)
- [Scope of Pattern Variables and switch](#)
- [Null case Labels](#)

When Clauses

You can add a Boolean expression right after a pattern label with a when clause. This is called a *guarded pattern label*. The Boolean expression in the when clause is called a *guard*. A value matches a guarded pattern label if it matches the pattern and, if so, the guard also evaluates to true. Consider the following example:

```

static void test(Object obj) {
    switch (obj) {
        case String s:
            if (s.length() == 1) {
                System.out.println("Short: " + s);
            } else {
                System.out.println(s);
            }
            break;
        default:
            System.out.println("Not a string");
    }
}

```

You can move the Boolean expression `s.length() == 1` right after the case label with a when clause:

```

static void test(Object obj) {
    switch (obj) {
        case String s when s.length() == 1 -> System.out.println("Short: " +
s);
        case String s -> System.out.println(s);
        default -> System.out.println("Not a
string");
    }
}

```

The first pattern label (which is a guarded pattern label) matches if `obj` is both a `String` and of length 1. The second pattern label matches if `obj` is a `String` of a different length.

A guarded pattern label has the form `p when e` where `p` is a pattern and `e` is a Boolean expression. The scope of any pattern variable declared in `p` includes `e`.

The following example demonstrates the use of a when clause with a record class:

```
record R(int x) { }  
// ...  
static void testSwitch(R r) {  
    switch(r) {  
        case R(int x) when x >= 5 -> System.out.println(x + " => 5");  
        default -> System.out.println(r.x + " < 5");  
    }  
}
```

The guard case `R(int x) when x >= 5` contains a type pattern `int x` nested in a record pattern. See [Record Patterns](#).

The following example also demonstrates when clauses. It uses a primitive type, `double`, for its switch expression's selector expression:

```
String doubleToRating(double rating) {  
    return switch(rating) {  
        case 0d -> "0 stars";  
        case double d when d > 0d && d < 2.5d  
            -> d + " is not good";  
        case double d when d >= 2.5f && d < 5d  
            -> d + " is better";  
        case 5d -> "5 stars";  
        default -> "Invalid rating";  
    };  
}
```

The guard case `double d when d > 0d && d < 2.5d` contains a type pattern `double d` that's *not* nested in a record pattern. This is called a top-level pattern.

Note

Guards restricting values of *primitive types* coming from a top-level pattern is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [Preview Language and VM Features](#).
See [JEP 530: Primitive Types in Patterns, instanceof, and switch \(Fourth Preview\)](#) for additional information.

The following example uses a long for its selector expression:

```
void bigNumbers(long v) {  
    switch (v) {  
        case long x when x < 1_000_000L ->  
            System.out.println("Less than a million");  
        case long x when x < 1_000_000_000L ->  
            System.out.println("Less than a billion");  
        case long x when x < 1_000_000_000_000L ->
```

```

        System.out.println("Less than a trillion");
    case long x when x < 1_000_000_000_000_000L ->
        System.out.println("Less than a quadrillion");
    default -> System.out.println("At least a quadrillion");
    }
}

```

Pattern Label Dominance

It's possible that many pattern labels could match the value of the selector expression. To help predictability, the labels are tested in the order that they appear in the switch block. In addition, the compiler raises an error if a pattern label can never match because a preceding one will always match first. The following example results in a compile-time error:

```

static void error(Object obj) {
    switch(obj) {
        case CharSequence cs ->
            System.out.println("A sequence of length " + cs.length());
        // error: this case label is dominated by a preceding case label
        case String s ->
            System.out.println("A string: " + s);
        default -> { break; }
    }
}

```

The first pattern label `case CharSequence cs` *dominates* the second pattern label `case String s` because every value that matches the pattern `String s` also matches the pattern `CharSequence cs` but not the other way around. It's because `String` is a subtype of `CharSequence`.

A pattern label can dominate a constant label. These examples cause compile-time errors:

```

static void error2(Integer value) {
    switch(value) {
        case Integer i ->
            System.out.println("Integer: " + i);
        // Compile-time errors for both cases -1 and 1:
        // this case label is dominated by a preceding case label
        case -1, 1 ->
            System.out.println("The number 42");
        default -> { break; }
    }
}
// ...

enum Color { RED, GREEN, BLUE; }
// ...

static void error3(Color value) {
    switch(value) {
        case Color c ->
            System.out.println("Color: " + c);
        // error: this case label is dominated by a preceding case label
        case RED ->

```

```

        System.out.println("The color red");
    }
}

```

① Note

Guarded pattern labels don't dominate constant labels. For example:

```

static void testInteger(Integer value) {
    switch(value) {
        case Integer i when i > 0 ->
            System.out.println("Positive integer");
        case 1 ->
            System.out.println("Value is 1");
        case -1 ->
            System.out.println("Value is -1");
        case Integer i ->
            System.out.println("An integer");
    }
}

```

Although the value 1 matches both the guarded pattern label `case Integer i when i > 0` and the constant label `case 1`, the guarded pattern label doesn't dominate the constant label. Guarded patterns aren't checked for dominance because they're generally undecidable. Consequently, you should order your case labels so that constant labels appear first, followed by guarded pattern labels, and then followed by nonguarded pattern labels:

```

static void testIntegerBetter(Integer value) {
    switch(value) {
        case 1 ->
            System.out.println("Value is 1");
        case -1 ->
            System.out.println("Value is -1");
        case Integer i when i > 0 ->
            System.out.println("Positive integer");
        case Integer i ->
            System.out.println("An integer");
    }
}

```

Dominance and Unconditionally Exact Conversions

Whether a conversion between primitive types is unconditionally exact (see [Conversion Safety and switch](#)) affects how patterns can dominate other patterns.

In the following example, the compiler generates an error for the case label `case byte b`:

```

float f = 100.01f;

switch (f) {

```

```

case int i ->
    System.out.println(i + " as an int");
// error: this case label is dominated by a preceding case label
case byte b ->
    System.out.println(b + " as a byte");
default ->
    System.out.println(f + " as a float");
}

```

The case label `case int i` dominates the case label `case byte b`. Every value that matches the pattern `int i` unconditionally matches every value that matches the pattern `int i` because converting a value from `byte` to `int` is a type-based unconditionally exact conversion.

case labels can dominate other case labels that are constants. In the following example, the compiler generates an error for the case label `case 100`:

```

int i = 100;

switch (i) {
    case float f    -> {}
    // error: this case label is dominated by a preceding case label
    case 100        -> {}
    default         -> {}
}

```

The case label `case float i` dominates the case label `case 100`. The constant `100` would unconditionally match the pattern `float f` because converting `int 100` to a `float` is a value-based unconditionally exact conversion.

A case label's type pattern is an *unconditional pattern* if it matches all possible selector values of the switch statement or expression. Such a pattern dominates all following case labels. In the following example, the case label `case int _` will dominate all other case labels that would follow it:

```

int i = 100;

switch (i) {
    case int _      -> {} // unconditional pattern
    // error: this case label is dominated by a preceding case label
    case float _    -> {}
}

```

Scope of Pattern Variables and switch

In a switch expression or statement, the scope of a pattern variable declared in a case label includes the following:

- The `when` clause of the case label:

```

static void test(Object obj) {
    switch (obj) {
        case Character c when c.charValue() == 7:
            System.out.println("Ding!");
            break;
    }
}

```

```

        default:
            break;
    }
}

```

The scope of pattern variable `c` includes the when clause of the case label that contains the declaration of `c`.

- The expression, block, or throw statement that appears to the right of the arrow of the case label:

```

static void test(Object obj) {
    switch (obj) {
        case Character c -> {
            if (c.charValue() == 7) {
                System.out.println("Ding!");
            }
            System.out.println("Character, value " + c.charValue());
        }
        case Integer i ->
            System.out.println("Integer: " + i);
        default -> {
            break;
        }
    }
}

```

The scope of pattern variable `c` includes the block to the right of case `Character c ->`. The scope of pattern variable `i` includes the `println` statement to the right of case `Integer i ->`.

- The switch-labeled statement group of a case label:

```

static void test(Object obj) {
    switch (obj) {
        case Character c:
            if (c.charValue() == 7) {
                System.out.print("Ding ");
            }
            if (c.charValue() == 9) {
                System.out.print("Tab ");
            }
            System.out.println("character, value " + c.charValue());
        default:
            // You cannot use the pattern variable c here:
            break;
    }
}

```

The scope of pattern variable `c` includes the case `Character c` statement group: the two `if` statements and the `println` statement that follows them. The scope doesn't include the default statement group even though the switch statement can execute the case `Character c` statement group, fall through the default label, and then execute the default statement group.

Note

You will get a compile-time error if it's possible to fall through a case label that declares a pattern variable. The following example doesn't compile:

```
static void test(Object obj) {
    switch (obj) {
        case Character c:
            if (c.charValue() == 7) {
                System.out.print("Ding ");
            }
            if (c.charValue() == 9) {
                System.out.print("Tab ");
            }
            System.out.println("character");
        case Integer i: // Compile-time error
            System.out.println("An integer " + i);
        default:
            System.out.println("Neither character nor integer");
    }
}
```

If this code were allowed, and the value of the selector expression, `obj`, were a `Character`, then the switch statement can execute the case `Character c` statement group and then fall through the case `Integer i` label, where the pattern variable `i` would have not been initialized.

See [Scope of Pattern Variables and instanceof](#) for more examples of where you can use a pattern variable.

Null case Labels

switch expressions and switch statements used to throw a `NullPointerException` if the value of the selector expression is null. Currently, to add more flexibility, a null case label is available:

```
static void test(Object obj) {
    switch (obj) {
        case null      -> System.out.println("null!");
        case String s -> System.out.println("String");
        default        -> System.out.println("Something else");
    }
}
```

This example prints `null!` when `obj` is null instead of throwing a `NullPointerException`.

You may not combine a null case label with anything but a default case label. The following generates a compiler error:

```
static void testStringOrNull(Object obj) {
    switch (obj) {
        // error: invalid case label combination
        case null, String s -> System.out.println("String: " + s);
    }
}
```

```

        default          -> System.out.println("Something else");
    }
}

```

However, the following compiles:

```

static void testStringOrNull(Object obj) {
    switch (obj) {
        case String s      -> System.out.println("String: " + s);
        case null, default -> System.out.println("null or not a string");
    }
}

```

If a selector expression evaluates to null and the switch block does not have null case label, then a `NullPointerException` is thrown as normal. Consider the following switch statement:

```

String s = null;
switch (s) {
    case Object obj -> System.out.println("This doesn't match null");
    // No null label; NullPointerException is thrown
    //     if s is null
}

```

Although the pattern label `case Object obj` matches objects of type `String`, this example throws a `NullPointerException`. The selector expression evaluates to null, and the switch expression doesn't contain a null case label.

Type Patterns

A *type pattern* consists of a type along with a single pattern variable. It's used to test whether a value is an instance of the type appearing in the pattern.

Topics

- [Type Patterns with Reference Types](#)
- [Type Patterns with Primitive Types](#)
- [Type Patterns with Parameterized Types](#)

Type Patterns with Reference Types

The pattern variable of a type pattern can be any reference type.

The following switch statement matches the selector expression `obj` with type patterns that involve a class type, an enum type, a record type, and an `int[]` array type:

```

record Point(int x, int y) { }
enum Color { RED, GREEN, BLUE; }
//...

static void typeTester(Object obj) {
    switch (obj) {
        case null          -> System.out.println("null");
    }
}

```

```
        case String s -> System.out.println("String");
        case Color c  -> System.out.println("Color with " + c.values().length
+ " values");
        case Point p  -> System.out.println("Record class: " + p.toString());
        case int[] ia -> System.out.println("Array of int values of length" +
ia.length);
        default       -> System.out.println("Something else");
    }
}
```

Type Patterns with Primitive Types

You can specify a primitive type instead of a reference type in a type pattern.

Note

Primitive types in type patterns is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [Preview Language and VM Features](#). See [JEP 530: Primitive Types in Patterns, instanceof, and switch \(Fourth Preview\)](#) for additional information.

The following example uses type patterns to test whether a value is a float, double, or int:

```
float v = 1000.01f;

if (v instanceof float f) System.out.println("float value: " + f);
if (v instanceof double d) System.out.println("double value: " + d);
if (v instanceof int i) System.out.println("int value: " + i);
```

It prints output similar to the following:

```
float value: 1000.01
double value: 1000.010009765625
```

See the section [Decimal ↔ Binary Conversion Issues](#) in the Double JavaDoc API documentation for an explanation why the converted double value is different than the original float value in this example.

Tip

Using pattern matching with primitives is particularly useful for ensuring type safety when casting values. See [Safe Casting with instanceof and switch](#) for more information.

The case labels in the following switch expression use type patterns that involve float:

```
String floatToRating(float rating) {
    return switch(rating) {
        case 0f      -> "0 stars";
        case 2.5f    -> "Average";
        case 5f      -> "Best";
        case float f -> "Invalid rating: " + f;
    };
}
```

Type Patterns with Parameterized Types

As described in [Safe Casting with instanceof and switch](#), you can use the instanceof operator to test if the right operand's type can be cast to left operand's type. This still applies if the right operand is a type pattern that contains one or more parameterized types. However, because of type erasure (see [Type Erasure](#) in The Java Tutorials), there's no parameterized type information at run time. Consequently, the runtime can't fully validate some casts between types that contain parameterized types. The compiler generates warnings or errors for these situations. However, the compiler will generate an error if it can't fully validate casts between instanceof operands that contain parameterized types.

Consider the following example that tries to cast a List to a List<String>:

```
List myList = null;
List<String> stringList = (List<String>) myList;
```

The compiler generates this warning with the -Xlint:unchecked option:

```
warning: [unchecked] unchecked cast
List<String> stringList = (List<String>) myList;
                             ^
   required: List<String>
   found:    List
```

Consider the following example where the instanceof operator tests whether myList, a List, has the type List<String>:

```
List myList = null;

if (myList instanceof List<String> sl) {
    System.out.println("myList is a List<String>, size " + sl.size());
}
```

The compiler generates an error for the second instanceof expression:

```
error: List cannot be safely cast to List<String>
if (myList instanceof List<String> sl) {
    ^
```

The following contains additional examples of instanceof expressions containing type patterns with parameterized types.

```
public class Box<T> {
    private T t;
    public Box() { }
    public Box(T t) { this.t = t; }
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

public class Shoebox<T> extends Box<T> {
    public Shoebox(T t) { super(t); }
}
// ...

Shoebox<String> sb = new Shoebox<>("a pair of new shoes");

if (sb instanceof Box<String> s) {
    System.out.println("Box<String> contains: " + s.get());
}

if (sb instanceof Shoebox<String> s) {
    System.out.println("Shoebox<String> contains: " + s.get());
}

if (sb instanceof Shoebox<?> s) {
    System.out.println("Shoebox<?> contains: " + s.get());
}

if (sb instanceof Shoebox<Object> s) {
    // error: incompatible types: Shoebox<String> cannot be
    // converted to Shoebox<Object>
    System.out.println("Shoebox<Object> contains: " + s.get());
}
```

The following expressions return true. The type parameters on both sides of the instanceof pattern matching operator are the same:

- sb instanceof Box<String> s
- sb instanceof Shoebox<String> s

The expression sb instanceof Shoebox<?> s returns true. The wildcard type (?) matches all types.

However, the compiler can't fully validate at runtime whether Shoebox<Object> can be cast to Shoebox<String> even though String is a subtype of Object. This parameterized type information won't exist at run time because of type erasure. Consequently, the compiler generates an error for the following expression:

```
sb instanceof Shoebox<Object> s
```

Note

The previous example uses `instanceof` as the pattern matching operator. If you use `instanceof` as the type comparison operator, the same issues about parameterized types and type erasure still apply. For example, the compiler generates an error for this expression:

```
sb instanceof Shoebox<Object>
```

Record Patterns

You can use a record pattern to test whether a value is an instance of a record class type (see [Record Classes](#)) and, if it is, to recursively perform pattern matching on its component values.

For background information about record patterns, see [JEP 440](#).

The following example tests whether `obj` is an instance of the `Point` record with the record pattern `Point(double x, double y)`:

```
record Point(double x, double y) {}  
// ...  
  
static void printAngleFromXAxis(Object obj) {  
    if (obj instanceof Point(double x, double y)) {  
        System.out.println(Math.toDegrees(Math.atan2(y, x)));  
    }  
}
```

In addition, this example extracts the `x` and `y` values from `obj` directly, automatically calling the `Point` record's accessor methods.

A *record pattern* consists of a type and a (possibly empty) record pattern list. In this example, the type is `Point` and the pattern list is `(double x, double y)`.

Note

The null value does not match any record pattern.

The following example is the same as the previous one except it uses a type pattern instead of a record pattern:

```
static void printAngleFromXAxisTypePattern(Object obj) {  
    if (obj instanceof Point p) {  
        System.out.println(Math.toDegrees(Math.atan2(p.y(), p.x())));  
    }  
}
```

Topics

- [Generic Record Patterns](#)
- [Using var in Record Patterns](#)
- [Primitive Types in Record Patterns](#)
- [Nested Record Patterns](#)

Generic Record Patterns

If a record class is generic, then you can explicitly specify the type arguments in a record pattern. For example:

```
record Box<T>(T t) { }  
// ...  
  
static void printBoxContents(Box<String> bo) {  
    if (bo instanceof Box<String>(String s)) {  
        System.out.println("Box contains: " + s);  
    }  
}
```

You can test whether a value is an instance of a parameterized record type provided that the value could be cast to the record type in the pattern without requiring an unchecked conversion. The following example doesn't compile:

```
static void uncheckedConversion(Box bo) {  
    // error: Box cannot be safely cast to Box<String>  
    if (bo instanceof Box<String>(var s)) {  
        System.out.println("String " + s);  
    }  
}
```

Using var in Record Patterns

You can use var in the record pattern's component list. The compiler can infer the type of the type arguments for record patterns in all constructs that accept patterns: switch statements, switch expressions, and instanceof expressions.

In the following example, the compiler infers that the pattern variables x and y are of type double:

```
static void printAngleFromXAxis(Object obj) {  
    if (obj instanceof Point(var x, var y)) {  
        System.out.println(Math.toDegrees(Math.atan2(y, x)));  
    }  
}
```

The following example is equivalent to `printBoxContents`. The compiler infers its type argument and pattern variable: `Box(var s)` is inferred as `Box<String>(String s)`

```
static void printBoxContentsAgain(Box<String> bo) {  
    if (bo instanceof Box(var s)) {  
        System.out.println("Box contains: " + s);  
    }  
}
```

In the following example, the compiler infers `MyPair(var s, var i)` as `MyPair<String, Integer>(String s, Integer i)`:

```
record MyPair<T, U>(T x, U y) { }  
// ...  
  
static void recordInference(MyPair<String, Integer> p) {  
    switch (p) {  
        case MyPair(var s, var i) ->  
            System.out.println(s + ", #" + i);  
    }  
}
```

Primitive Types in Record Patterns

You can perform pattern matching on the types of a record's components, even if these types are primitive.

① Note

Support for pattern matching on primitive types of a record's components is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [JEP 530: Primitive Types in Patterns, instanceof, and switch \(Fourth Preview\)](#) for additional information.

In the following example, the record `Cup` contains one component of type `double`. The example creates a `Cup` that contains a `float` and then tests whether the `Cup` contains a `double`, `float`, or `int` value with the `instanceof` operator:

```
record Cup(double d) { }  
//...  
  
Cup myCup = new Cup(34.567f);  
  
if (myCup instanceof Cup(double d)) System.out.println("Cup contains double:  
" + d);  
if (myCup instanceof Cup(float f)) System.out.println("Cup contains float: "  
+ f);  
if (myCup instanceof Cup(int i)) System.out.println("Cup contains int: " + i);
```

It prints output similar to the following:

```
Cup contains double: 34.56700134277344
Cup contains float: 34.567
```

Similarly, as described in [Type Patterns with Primitive Types](#), a value of a record's component matches the corresponding type pattern if it's safe to cast the value to the type in the type pattern. A cast is safe if there's no loss of information or exception thrown during the cast or conversion. For example, casting a float to a double or float is safe; casting a float to an int isn't. (See the section [Decimal → Binary Conversion Issues](#) in the Double JavaDoc API documentation for an explanation why the converted double value is different than the original float value in this example.)

Nested Record Patterns

You can nest a record pattern inside another record pattern:

```
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record ColoredRectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
// ...

static void printXCoordOfUpperLeftPointWithPatterns(ColoredRectangle r) {
    if (r instanceof ColoredRectangle(
        ColoredPoint(Point(var x, var y), var upperLeftColor),
        var lowerRightCorner)) {
        System.out.println("Upper-left corner: " + x);
    }
}
```

You can do the same for parameterized records. The compiler infers the types of the record pattern's type arguments and pattern variables. In the following example, the compiler infers `Box(Box(var s))` as `Box<Box<String>>(Box(String s))`.

```
static void nestedBox(Box<Box<String>> bo) {
    // Box(Box(var s)) is inferred to be Box<Box<String>>(Box(var s))
    if (bo instanceof Box(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

10

Record Classes

Record classes, which are a special kind of class, help to model plain data aggregates with less ceremony than normal classes.

For background information about record classes, see [JEP 395](#).

A record declaration specifies in a header a description of its contents; the appropriate accessors, constructor, equals, hashCode, and toString methods are created automatically. A record's fields are final because the class is intended to serve as a simple "data carrier".

For example, here is a record class with two fields:

```
record Rectangle(double length, double width) { }
```

This concise declaration of a rectangle is equivalent to the following normal class:

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width() { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```

A record class declaration consists of a name; optional type parameters (generic record declarations are supported); a header, which lists the "components" of the record; and a body.

A record class declares the following members automatically:

- For each component in the header, the following two members:
 - A private final field with the same name and declared type as the record component. This field is sometimes referred to as a *component field*.

- A public accessor method with the same name and type of the component; in the `Rectangle` record class example, these methods are `Rectangle::length()` and `Rectangle::width()`.
- A *canonical constructor* whose signature is the same as the header. This constructor assigns each argument from the new expression that instantiates the record class to the corresponding component field.
- Implementations of the `equals` and `hashCode` methods, which specify that two record classes are equal if they are of the same type and contain equal component values.
- An implementation of the `toString` method that includes the string representation of all the record class's components, with their names.

As record classes are just special kinds of classes, you create a record object (an instance of a record class) with the `new` keyword, for example:

```
Rectangle r = new Rectangle(4,5);
```

To access a record's component fields, call its accessor methods:

```
System.out.println("Length: " + r.length() + ", width: " + r.width());
```

This example prints the following output:

```
Length: 4.0, width: 5.0
```

Topics

- [The Canonical Constructor of a Record Class](#)
- [Alternative Record Constructors](#)
- [Explicit Declaration of Record Class Members](#)
- [Features of Record Classes](#)
- [Record Classes and Sealed Classes and Interfaces](#)
- [Local Record Classes](#)
- [Static Members of Inner Classes](#)
- [Differences Between the Serialization of Records and Ordinary Objects](#)
- [APIs Related to Record Classes](#)

The Canonical Constructor of a Record Class

The following example explicitly declares the canonical constructor for the `Rectangle` record class. It verifies that `length` and `width` are greater than zero. If not, it throws an `IllegalArgumentException`:

```
record Rectangle(double length, double width) {  
    public Rectangle(double length, double width) {  
        if (length <= 0 || width <= 0) {  
            throw new java.lang.IllegalArgumentException(  
                String.format("Invalid dimensions: %f, %f", length, width));  
        }  
    }  
}
```

```

        this.length = length;
        this.width = width;
    }
}

```

Repeating the record class's components in the signature of the canonical constructor can be tiresome and error-prone. To avoid this, you can declare a *compact constructor* whose signature is implicit (derived from the components automatically).

For example, the following compact constructor declaration validates length and width in the same way as in the previous example:

```

record Rectangle(double length, double width) {
    public Rectangle {
        if (length <= 0 || width <= 0) {
            throw new java.lang.IllegalArgumentException(
                String.format("Invalid dimensions: %f, %f", length, width));
        }
    }
}

```

This succinct form of constructor declaration is only available in a record class. Note that the statements `this.length = length;` and `this.width = width;` which appear in the canonical constructor do not appear in the compact constructor. At the end of a compact constructor, its implicit formal parameters are assigned to the record class's private fields corresponding to its components.

Alternative Record Constructors

You can define alternative, noncanonical constructors whose argument list doesn't match the record's type parameters. However, these constructors must invoke the record's canonical constructor. In the following example, the constructor for the record `RectanglePair` contains one parameter, a `Pair<Float>`. It calls its implicitly defined canonical constructor to initialize its fields `length` and `width`:

```

record Pair<T extends Number>(T x, T y) { }

record RectanglePair(double length, double width) {
    public RectanglePair(Pair<Double> corner) {
        this(corner.x().doubleValue(), corner.y().doubleValue());
    }
}

```

Explicit Declaration of Record Class Members

You can explicitly declare any of the members derived from the header, such as the public accessor methods that correspond to the record class's components, for example:

```

record Rectangle(double length, double width) {

    // Public accessor method
    public double length() {
        System.out.println("Length is " + length);
    }
}

```

```
        return length;
    }
}
```

If you implement your own accessor methods, then ensure that they have the same characteristics as implicitly derived accessors (for example, they're declared `public` and have the same return type as the corresponding record class component). Similarly, if you implement your own versions of the `equals`, `hashCode`, and `toString` methods, then ensure that they have the same characteristics and behavior as those in the `java.lang.Record` class, which is the common superclass of all record classes.

You can declare static fields, static initializers, and static methods in a record class, and they behave as they would in a normal class, for example:

```
record Rectangle(double length, double width) {

    // Static field
    static double goldenRatio;

    // Static initializer
    static {
        goldenRatio = (1 + Math.sqrt(5)) / 2;
    }

    // Static method
    public static Rectangle createGoldenRectangle(double width) {
        return new Rectangle(width, width * goldenRatio);
    }
}
```

You cannot declare instance variables (non-static fields) or instance initializers in a record class.

For example, the following record class declaration doesn't compile:

```
record Rectangle(double length, double width) {

    // Field declarations must be static:
    BiFunction<Double, Double, Double> diagonal;

    // Instance initializers are not allowed in records:
    {
        diagonal = (x, y) -> Math.sqrt(x*x + y*y);
    }
}
```

You can declare instance methods in a record class, independent of whether you implement your own accessor methods. You can also declare nested classes and interfaces in a record class, including nested record classes (which are implicitly static). For example:

```
record Rectangle(double length, double width) {

    // Nested record class
    record RotationAngle(double angle) {
        public RotationAngle {
```

```

        angle = Math.toRadians(angle);
    }
}

// Public instance method
public Rectangle getRotatedRectangleBoundingBox(double angle) {
    RotationAngle ra = new RotationAngle(angle);
    double x = Math.abs(length * Math.cos(ra.angle())) +
               Math.abs(width * Math.sin(ra.angle()));
    double y = Math.abs(length * Math.sin(ra.angle())) +
               Math.abs(width * Math.cos(ra.angle()));
    return new Rectangle(x, y);
}
}

```

You cannot declare native methods in a record class.

Features of Record Classes

A record class is implicitly `final`, so you cannot explicitly extend a record class. However, beyond these restrictions, record classes behave like normal classes:

- You can create a generic record class, for example:

```
record Triangle<C extends Coordinate> (C top, C left, C right) { }
```

- You can declare a record class that implements one or more interfaces, for example:

```
record Customer(...) implements Billable { }
```

- You can annotate a record class and its individual components, for example:

```

import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface GreaterThanZero { }

record Rectangle(
    @GreaterThanZero double length,
    @GreaterThanZero double width) { }

```

If you annotate a record component, then the annotation may be propagated to members and constructors of the record class. This propagation is determined by the contexts in which the annotation interface is applicable. In the previous example, the `@Target(ElementType.FIELD)` meta-annotation means that the `@GreaterThanZero` annotation is propagated to the field corresponding to the record component. Consequently, this record class declaration would be equivalent to the following normal class declaration:

```

public final class Rectangle {
    private final @GreaterThanZero double length;
    private final @GreaterThanZero double width;
}

```

```

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width() { return this.width; }
}

```

Record Classes and Sealed Classes and Interfaces

Record classes work well with sealed classes and interfaces. See [Record Classes as Permitted Subclasses](#) for an example.

Local Record Classes

A local record class is similar to a local class; it's a record class defined in the body of a method.

In the following example, a merchant is modeled with a record class, `Merchant`. A sale made by a merchant is also modeled with a record class, `Sale`. Both `Merchant` and `Sale` are top-level record classes. The aggregation of a merchant and their total monthly sales is modeled with a *local* record class, `MonthlySales`, which is declared inside the `findTopMerchants` method. This local record class improves the readability of the stream operations that follow:

```

import java.time.*;
import java.util.*;
import java.util.stream.*;

record Merchant(String name) { }

record Sale(Merchant merchant, LocalDate date, double value) { }

public class MerchantExample {

    List<Merchant> findTopMerchants(
        List<Sale> sales, List<Merchant> merchants, int year, Month month) {

        // Local record class
        record MerchantSales(Merchant merchant, double sales) {}

        return merchants.stream()
            .map(merchant -> new MerchantSales(
                merchant, this.computeSales(sales, merchant, year, month)))
            .sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))
            .map(MerchantSales::merchant)
            .collect(Collectors.toList());
    }

    double computeSales(List<Sale> sales, Merchant mt, int yr, Month mo) {
        return sales.stream()
            .filter(s -> s.merchant().name().equals(mt.name()) &&
                s.date().getYear() == yr &&

```

```

        s.date().getMonth() == mo)
        .mapToDouble(s -> s.value())
        .sum();
    }

    public static void main(String[] args) {

        Merchant sneha = new Merchant("Sneha");
        Merchant raj = new Merchant("Raj");
        Merchant florence = new Merchant("Florence");
        Merchant leo = new Merchant("Leo");

        List<Merchant> merchantList = List.of(sneha, raj, florence, leo);

        List<Sale> salesList = List.of(
            new Sale(sneha,      LocalDate.of(2020, Month.NOVEMBER, 13),
11034.20),
            new Sale(raj,       LocalDate.of(2020, Month.NOVEMBER, 20),
8234.23),
            new Sale(florence, LocalDate.of(2020, Month.NOVEMBER, 19),
10003.67),
            // ...
            new Sale(leo,       LocalDate.of(2020, Month.NOVEMBER, 4),
9645.34));

        MerchantExample app = new MerchantExample();

        List<Merchant> topMerchants =
            app.findTopMerchants(salesList, merchantList, 2020,
Month.NOVEMBER);
        System.out.println("Top merchants: ");
        topMerchants.stream().forEach(m -> System.out.println(m.name()));
    }
}

```

Like nested record classes, local record classes are implicitly static, which means that their own methods can't access any variables of the enclosing method, unlike local classes, which are never static.

Static Members of Inner Classes

Prior to Java SE 16, you could not declare an explicitly or implicitly static member in an inner class unless that member is a constant variable. This means that an inner class cannot declare a record class member because nested record classes are implicitly static.

In Java SE 16 and later, an inner class may declare members that are either explicitly or implicitly static, which includes record class members. The following example demonstrates this:

```

public class ContactList {

    record Contact(String name, String number) { }

    public static void main(String[] args) {

```

```
class Task implements Runnable {  
  
    // Record class member, implicitly static,  
    // declared in an inner class  
    Contact c;  
  
    public Task(Contact contact) {  
        c = contact;  
    }  
    public void run() {  
        System.out.println(c.name + ", " + c.number);  
    }  
}  
  
List<Contact> contacts = List.of(  
    new Contact("Sneha", "555-1234"),  
    new Contact("Raj", "555-2345"));  
contacts.stream()  
    .forEach(cont -> new Thread(new Task(cont)).start());  
}
```

Differences Between the Serialization of Records and Ordinary Objects

You can serialize and deserialize instances of record classes, but you can't customize the process by providing `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, or `readExternal` methods. The components of a record class govern serialization, while the canonical constructor of a record class governs deserialization.

Records are serialized differently than ordinary serializable objects. The serialized form of a record object is a sequence of values derived from the record components. When a record object is deserialized, its component values are reconstructed from this sequence of values. Afterward, a record object is created by invoking the record's canonical constructor with the component values as arguments. In contrast, when an ordinary object is deserialized, it's possible that it can be created without one of its constructors being invoked. Consequently, serializable records leverage the guarantees provided by record classes to offer a simpler and more secure serialization model.

Topics

- [Record Serialization Principles](#)
- [How Record Serialization Works](#)

Record Serialization Principles

Two principles exist with regards to record serialization:

- Serialization of a record object is based only on its state components.
- Deserialization of a record object uses only the canonical constructor.

A consequence of the first point is that customization of the serialized form of a record object is not possible — the serialized form is based on the state components and the state

components only. As a result of this restriction, programmers more easily understand the serialized form of a record: it consists of the state components of the record.

The second point relates to the mechanics of the deserialization process. Suppose deserialization is reading the bytes of an object for a normal class (not a record class). Deserialization would create a new object by invoking the no-args constructor of a superclass, then use reflection to set the object's fields to values deserialized from the stream. This is insecure because the normal class has no opportunity to validate the values coming from the stream. The result may be an "impossible" object that could never be created by an ordinary Java program using constructors. With records, deserialization works differently. Deserialization creates a new record object by invoking a record class's canonical constructor, passing values deserialized from the stream as arguments to the canonical constructor. This is secure because it means the record class can validate the values before assigning them to fields, just like when an ordinary Java program creates a record object via `new`. "Impossible" objects are impossible. This is achievable because the record components, the canonical constructor, and the serialized form are all known and consistent.

How Record Serialization Works

To demonstrate the differences between the serialization of ordinary objects and records, let's create two classes, `RangeClass` and `RangeRecord`, that model a range of integers. `RangeClass` is an ordinary class while `RangeRecord` is a record. They have a low-end value, `lo`, and a high-end value, `hi`. They also implement `Serializable` because we want to be able to serialize instances of them.

```
public class RangeClass implements Serializable {
    private static final long serialVersionUID = -3305276997530613807L;
    private final int lo;
    private final int hi;
    public RangeClass(int lo, int hi) {
        this.lo = lo;
        this.hi = hi;
    }
    public int lo() { return lo; }
    public int hi() { return hi; }
    @Override public boolean equals(Object other) {
        if (other instanceof RangeClass that
            && this.lo == that.lo && this.hi == that.hi) {
            return true;
        }
        return false;
    }
    @Override public int hashCode() {
        return Objects.hash(lo, hi);
    }
    @Override public String toString() {
        return String.format("%s[lo=%d, hi=%d]", getClass().getName(), lo, hi);
    }
}
```

Note the verbose boilerplate code for the `equals`, `hashCode`, and `toString` methods.

The following example is the equivalent record counterpart of `RangeClass`. You make a record class serializable in the same way as a normal class, by implementing `Serializable`.

```
public record RangeRecord (int lo, int hi) implements Serializable { }
```

Note that you don't have to add any additional boilerplate to `RangeRecord` to make it serializable. Specifically, you don't need to add a `serialVersionUID` field because the `serialVersionUID` of a record class is 0L unless explicitly declared, and the requirement for matching the `serialVersionUID` value is waived for record classes.

Rarely, for migration compatibility between normal classes and record classes, a `serialVersionUID` may be declared. See the section [5.6.2 Compatible Changes](#) in [Java Object Serialization Specification](#) for more information.

The following examples serialize a `RangeClass` object and `RangeRecord` object, both with the same high-end and low-end values.

```
import java.io.*;

public class Serialize {
    public static void main(String... args) throws Exception {
        try (var fos = new FileOutputStream("serial.data");
             var oos = new ObjectOutputStream(fos)) {
            oos.writeObject(new RangeClass(100, 1));
            oos.writeObject(new RangeRecord(100, 1));
        }
    }
}

import java.io.*;

public class Deserialize {
    public static void main(String... args) throws Exception {
        try (var fis = new FileInputStream("serial.data");
             var ois = new ObjectInputStream(fis)) {
            System.out.println(ois.readObject());
            System.out.println(ois.readObject());
        }
    }
}
```

Running these two examples print the following output:

```
RangeClass[lo=100, hi=1]
```

```
RangeRecord[lo=100, hi=1]
```

Oops! Did you manage to spot the mistake? The low-end value is higher than the high-end value. This shouldn't be allowed. An integer range implementation should have this invariant: the low end of the range can be no higher than the high end.

Let's modify `RangeClass` and `RangeRecord` as follows to add this invariant:

```
public class RangeClass implements ... {
    // ...
```

```

    public RangeClass(int lo, int hi) {
        if (lo > hi)
            throw new IllegalArgumentException(String.format("%d, %d", lo,
hi));
        this.lo = lo;
        this.hi = hi;
    }
    // ..
}

public record RangeRecord (int lo, int hi) implements ... {
    public RangeRecord {
        if (lo > hi)
            throw new IllegalArgumentException(String.format("%d, %d", lo,
hi));
    }
}

```

Note that `RangeRecord` uses the compact version of the canonical constructor declaration, which enables you to omit the boilerplate assignments.

With the updated `RangeClass` and `RangeRecord` examples, the `Deserialize` example prints output similar to the following:

```

RangeClass[lo=100, hi=1]
Exception in thread "main" java.io.InvalidObjectException: 100, 1
    at java.base/
java.io.ObjectInputStream.readRecord(ObjectInputStream.java:2296)
    at java.base/
java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2183)
    at java.base/
java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1685)
    at java.base/
java.io.ObjectInputStream.readObject(ObjectInputStream.java:499)
    at java.base/
java.io.ObjectInputStream.readObject(ObjectInputStream.java:457)
    at Deserialize.main(Deserialize.java:9)
Caused by: java.lang.IllegalArgumentException: 100, 1
    at RangeRecord.<init>(RangeRecord.java:6)
    at java.base/
java.io.ObjectInputStream.readRecord(ObjectInputStream.java:2294)
    ... 5 more

```

The example attempts to deserialize the stream objects in the `serial.data` file even though `RangeClass` and `RangeRecord` have been created with a low-end value of 100 and a high-end value of 1.

The example demonstrates that a normal class and a record class are deserialized differently:

- A `RangeClass` object was deserialized even though the newly created object violates the constructor invariant. This may seem counterintuitive at first, but as described earlier, deserialization of an object whose class is a normal class (not a record class) creates the object by invoking the no-args constructor of the (first non-serializable) superclass, which in this case is `java.lang.Object`. Of course, it would not be possible for the example `Serialize` to generate such a byte stream for a `RangeClass` object because the example must use the two-arg constructor with its invariant checking. However, remember

deserialization operates on just a stream of bytes, and these bytes can, in some cases, come from almost anywhere.

- However, the `RangeRecord` stream object failed to deserialize because its stream field values for the low end and high end violate the invariant check in the constructor. This is nice, and actually what we want: deserialization proceeds through the canonical constructor.

The fact that a serializable class can have a new object created without one of its constructors being invoked is often overlooked, even by experienced developers. An object created by invoking a distant no-args constructor can lead to unexpected behavior at run time, since invariant checks in the deserialized class's constructor are not performed. However, deserialization of a record object cannot be exploited to create an "impossible" object.

See the section [1.13 Serialization of Records](#) in [Java Object Serialization Specification](#) for more information.

APIs Related to Record Classes

The abstract class `java.lang.Record` is the common superclass of all record classes.

You might get a compiler error if your source file imports a class named `Record` from a package other than `java.lang`. A Java source file automatically imports all the types in the `java.lang` package through an implicit `import java.lang.*;` statement. This includes the `java.lang.Record` class, regardless of whether preview features are enabled or disabled.

Consider the following class declaration of `com.myapp.Record`:

```
package com.myapp;

public class Record {
    public String greeting;
    public Record(String greeting) {
        this.greeting = greeting;
    }
}
```

The following example, `org.example.MyappPackageExample`, imports `com.myapp.Record` with a wildcard but doesn't compile:

```
package org.example;
import com.myapp.*;

public class MyappPackageExample {
    public static void main(String[] args) {
        Record r = new Record("Hello world!");
    }
}
```

The compiler generates an error message similar to the following:

```
./org/example/MyappPackageExample.java:6: error: reference to Record is
ambiguous
    Record r = new Record("Hello world!");
    ^
    both class com.myapp.Record in com.myapp and class java.lang.Record in
```

```
java.lang match
```

```
./org/example/MyappPackageExample.java:6: error: reference to Record is  
ambiguous
```

```
    Record r = new Record("Hello world!");  
                  ^
```

```
both class com.myapp.Record in com.myapp and class java.lang.Record in  
java.lang match
```

Both `Record` in the `com.myapp` package and `Record` in the `java.lang` package are imported with a wildcard. Consequently, neither class takes precedence, and the compiler generates an error when it encounters the use of the simple name `Record`.

To enable this example to compile, change the `import` statement so that it imports the fully qualified name of `Record`:

```
import com.myapp.Record;
```

Note

The introduction of classes in the `java.lang` package is rare but necessary from time to time, such as `Enum` in Java SE 5, `Module` in Java SE 9, and `Record` in Java SE 14.

The class `java.lang.Class` has two methods related to record classes:

- `RecordComponent[] getRecordComponents()`: Returns an array of `java.lang.reflect.RecordComponent` objects, which correspond to the record class's components.
- `boolean isRecord()`: Similar to `isEnum()` except that it returns `true` if the class was declared as a record class.

Unnamed Variables and Patterns

Unnamed variables are variables that can be initialized but not used. *Unnamed patterns* can appear in a pattern list of a record pattern, and always match the corresponding record component. You can use them instead of a type pattern. They remove the burden of having to write a type and name of a pattern variable that's not needed in subsequent code. You denote both with the underscore character (`_`).

For background information about unnamed variables and patterns, see [JEP 456](#).

Unnamed Variables

You can use the underscore keyword (`_`) as the name of a local variable, exception, or lambda parameter in a declaration when the value of the declaration isn't needed. This is called an *unnamed variable*, which represents a variable that's being declared but it has no usable name.

Unnamed variables are useful when the side effect of a statement is more important than its result.

Consider the following example that iterates through the elements of the array `orderIDs` with a `for` loop. The side effect of this `for` loop is that it calculates the number of elements in `orderIDs` without ever using the loop variable `id`:

```
int[] orderIDs = {34, 45, 23, 27, 15};
int total = 0;
for (int id : orderIDs) {
    total++;
}
System.out.println("Total: " + total);
```

You can use an unnamed variable to omit or *elide* the unused variable `id`:

```
int[] orderIDs = {34, 45, 23, 27, 15};
int total = 0;
for (int _ : orderIDs) {
    total++;
}
System.out.println("Total: " + total);
```

The following table describes where you can declare an unnamed variable:

Table 11-1 Valid Unnamed Variable Declarations

Declaration Type	Example with Unnamed Variable
A local variable declaration statement in a block	<pre> record Caller(String phoneNumber) { } static List everyFifthCaller(Queue<Caller> q, int prizes) { var winners = new ArrayList<Caller>(); try { while (prizes > 0) { Caller _ = q.remove(); Caller _ = q.remove(); Caller _ = q.remove(); Caller _ = q.remove(); winners.add(q.remove()); prizes--; } } catch (NoSuchElementException _) { // Do nothing } return winners; } </pre> Note that you don't have to assign the value returned by <code>Queue::remove</code> to a variable, named or unnamed. You might want to do so to signify that a lesser-known API returns a value that your application doesn't use.
A resource specification of a try-with-resources statement	<pre> static void doesFileExist(String path) { try (var _ = new FileReader(path)) { // Do nothing } catch (IOException e) { e.printStackTrace(); } } </pre>
The header of a basic for statement	<pre> Function<String,Integer> sideEffect = s -> { System.out.println(s); return 0; }; for (int i = 0, _ = sideEffect.apply("Starting for-loop"); i < 10; i++) { System.out.println(i); } </pre>

Table 11-1 (Cont.) Valid Unnamed Variable Declarations

Declaration Type	Example with Unnamed Variable
The header of an enhanced for loop	<pre>static void stringLength(String s) { int len = 0; for (char _ : s.toCharArray()) { len++; } System.out.println("Length of " + s + ": " + len); }</pre>
An exception parameter of a catch block	<pre>static void validateNumber(String s) { try { int i = Integer.parseInt(s); System.out.println(i + " is valid"); } catch (NumberFormatException _) { System.out.println(s + " isn't valid"); } }</pre>
A formal parameter of a lambda expression	<pre>record Point(double x, double y) {} record UniqueRectangle(String id, Point upperLeft, Point lowerRight) {} static Map getIDs(List<UniqueRectangle> r) { return r.stream() .collect(Collectors.toMap(UniqueRectangle::id, _ -> "NODATA")); }</pre>

Unnamed Patterns

Consider the following example that calculates the distance between two instances of `ColoredPoint`:

```
record Point(double x, double y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}

double getDistance(Object obj1, Object obj2) {
    if (obj1 instanceof ColoredPoint(Point p1, Color c1) &&
        obj2 instanceof ColoredPoint(Point p2, Color c2)) {
        return java.lang.Math.sqrt(
```

```

        java.lang.Math.pow(p2.x - p1.x, 2) +
        java.lang.Math.pow(p2.y - p1.y, 2));
    } else {
        return -1;
    }
}

```

The example doesn't use the `Color` component of the `ColoredPoint` record. To simplify the code and improve readability, you can elide the type patterns `Color c1` and `Color c2` with the unnamed pattern (`_`):

```

double getDistance(Object obj1, Object obj2) {
    if (obj1 instanceof ColoredPoint(Point p1, _) &&
        obj2 instanceof ColoredPoint(Point p2, _)) {
        return java.lang.Math.sqrt(
            java.lang.Math.pow(p2.x - p1.x, 2) +
            java.lang.Math.pow(p2.y - p1.y, 2));
    } else {
        return -1;
    }
}

```

Alternatively, you can keep the type pattern's type and elide just its name:

```

    if (obj1 instanceof ColoredPoint(Point p1, Color _) &&
        obj2 instanceof ColoredPoint(Point p2, Color _))

```

No value is bound to the unnamed pattern variable. Consequently, the highlighted statement in the following example is invalid:

```

    if (obj1 instanceof ColoredPoint(Point p1, Color _) &&
        obj2 instanceof ColoredPoint(Point p2, Color _)) {
        // Compiler error: the underscore keyword '_' is only allowed to
        // declare unnamed patterns, local variables, exception
parameters or
        // lambda parameters
        System.out.println("Color: " + _);
        // ...
    }

```

Also, you can't use an unnamed pattern as a top-level pattern:

```

// Compiler error: the underscore keyword '_' is only allowed to
// declare unnamed patterns, local variables, exception parameters or
// lambda parameters
if (obj1 instanceof _) {
    // ...
}

```

You can use unnamed patterns in switch expressions and statements:

```

sealed interface Employee permits Salaried, Freelancer, Intern { }
record Salaried(String name, long salary) implements Employee { }

```

```
record Freelancer(String name) implements Employee { }
record Intern(String name) implements Employee { }

void printSalary(Employee b) {
    switch (b) {
        case Salaried r    -> System.out.println("Salary: " + r.salary());
        case Freelancer _  -> System.out.println("Other");
        case Intern _      -> System.out.println("Other");
    }
}
```

You may use multiple patterns in a case label provided that they don't declare any pattern variables. For example, you can rewrite the previous switch statement as follows:

```
switch (b) {
    case Salaried r          -> System.out.println("Salary: " +
r.salary());
    case Freelancer _, Intern _ -> System.out.println("Other");
}
```

12

Switch Expressions and Statements

You can use the `switch` keyword as either a statement or an expression. Like all expressions, switch expressions evaluate to a single value and can be used in statements. Switch expressions may contain "case L ->" labels that eliminate the need for break statements to prevent fall through. You can use a `yield` statement to specify the value of a switch expression.

For background information about the design of switch expressions, see [JEP 361](#).

Topics

- [Arrow Cases](#)
- [Colon Cases and the the `yield` Statement](#)
- [Qualified enum Constants as case Constants](#)
- [Primitive Values in switch Expressions and Statements](#)
- [Exhaustiveness of switch](#)
- [Completion and switch Expressions](#)

Arrow Cases

Consider the following switch statement that prints the number of letters of a day of the week:

```
public enum Day { SUNDAY, MONDAY, TUESDAY,
                 WEDNESDAY, THURSDAY, FRIDAY, SATURDAY; }
// ...

int numLetters = 0;
Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
}
System.out.println(numLetters);
```

It would be better if you could "return" the length of the day's name instead of storing it in the variable `numLetters`; you can do this with a `switch` expression. Furthermore, it would be better if you didn't need `break` statements to prevent fall through; they are laborious to write and easy to forget. You can do this with an *arrow case*. The following is a `switch` expression that uses arrow cases to print the number of letters of a day of the week:

```
Day day = Day.WEDNESDAY;
System.out.println(
    switch (day) {
        case MONDAY, FRIDAY, SUNDAY -> 6;
        case TUESDAY                 -> 7;
        case THURSDAY, SATURDAY      -> 8;
        case WEDNESDAY               -> 9;
    }
);
```

An arrow case has the following form:

```
case label_1, label_2, ..., label_n -> expression; | throw-statement; | block
```

When the Java runtime matches any of the labels to the left of the arrow, it runs the code to the right of the arrow and does not fall through; it does not run any other code in the `switch` expression (or statement). If the code to the right of the arrow is an expression, then the value of that expression is the value of the `switch` expression.

You can use arrow cases in `switch` statements. The following is like the first example, except it uses arrow cases instead of colon cases:

```
int numLetters = 0;
Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> numLetters = 6;
    case TUESDAY                 -> numLetters = 7;
    case THURSDAY, SATURDAY      -> numLetters = 8;
    case WEDNESDAY               -> numLetters = 9;
};
System.out.println(numLetters);
```

An arrow case along with its code to its right is called a *switch-labeled rule*.

Colon Cases and the the yield Statement

A *colon case* is a case label in the form `case L:`. You can use colon cases in `switch` expressions. A colon case along with its code to the right is called a *switch-labeled statement group*:

```
Day day = Day.WEDNESDAY;
int numLetters = switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        yield 6;
};
```

```
        case TUESDAY:
            System.out.println(7);
            yield 7;
        case THURSDAY:
        case SATURDAY:
            System.out.println(8);
            yield 8;
        case WEDNESDAY:
            System.out.println(9);
            yield 9;
    };
    System.out.println(numLetters);
```

The previous example uses `yield` statements. They take one argument, which is the value that the colon case produces in a switch expression.

The `yield` statement makes it easier for you to differentiate between switch statements and switch expressions. A switch statement, but not a switch expression, can be the target of a `break` statement. Conversely, a switch expression, but not a switch statement, can be the target of a `yield` statement.

① Note

It's recommended that you use arrow cases. It's easy to forget to insert `break` or `yield` statements when using colon cases; if you do, you might introduce unintentional fall through in your code.

For arrow cases, to specify multiple statements or code that are not expressions or throw statements, enclose them in a block. Specify the value that the arrow case produces with the `yield` statement:

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> {
        System.out.println(6);
        yield 6;
    }
    case TUESDAY -> {
        System.out.println(7);
        yield 7;
    }
    case THURSDAY, SATURDAY -> {
        System.out.println(8);
        yield 8;
    }
    case WEDNESDAY -> {
        System.out.println(9);
        yield 9;
    }
};
```

Qualified enum Constants as case Constants

You can use qualified enum constants as case constants in switch expressions and statements.

Consider the following switch expression whose selector expression is an enum type:

```
public enum Standard { SPADE, HEART, DIAMOND, CLUB }
// ...

static void determineSuitStandardDeck(Standard d) {
    switch (d) {
        case SPADE    -> System.out.println("Spades");
        case HEART    -> System.out.println("Hearts");
        case DIAMOND  -> System.out.println("Diamonds");
        default       -> System.out.println("Clubs");
    }
}
```

In the following example, the type of the selector expression is an interface that's been implemented by two enum types. Because the type of the selector expression isn't an enum type, this switch expression uses guarded patterns instead:

```
sealed interface CardClassification permits Standard, Tarot {}
public enum Standard implements CardClassification
    { SPADE, HEART, DIAMOND, CLUB }
public enum Tarot implements CardClassification
    { SPADE, HEART, DIAMOND, CLUB, TRUMP, EXCUSE }
// ...

static void determineSuit(CardClassification c) {
    switch (c) {
        case Standard s when s == Standard.SPADE    ->
System.out.println("Spades");
        case Standard s when s == Standard.HEART    ->
System.out.println("Hearts");
        case Standard s when s == Standard.DIAMOND  ->
System.out.println("Diamonds");
        case Standard s                             ->
System.out.println("Clubs");
        case Tarot t when t == Tarot.SPADE          ->
System.out.println("Spades or Piques");
        case Tarot t when t == Tarot.HEART          ->
System.out.println("Hearts or C\u0015ur");
        case Tarot t when t == Tarot.DIAMOND        ->
System.out.println("Diamonds or Carreaux");
        case Tarot t when t == Tarot.CLUB           ->
System.out.println("Clubs or Trefles");
        case Tarot t when t == Tarot.TRUMP          ->
System.out.println("Trumps or Atouts");
        case Tarot t                                 ->
System.out.println("The Fool or L'Excuse");
    }
}
```

However, switch expressions and statements allow qualified enum constants, so you could rewrite this example as follows:

```
static void determineSuitQualifiedNames(CardClassification c) {
    switch (c) {
        case Standard.SPADE -> System.out.println("Spades");
        case Standard.HEART -> System.out.println("Hearts");
        case Standard.DIAMOND -> System.out.println("Diamonds");
        case Standard.CLUB -> System.out.println("Clubs");
        case Tarot.SPADE -> System.out.println("Spades or Piques");
        case Tarot.HEART -> System.out.println("Hearts or C\u0003ur");
        case Tarot.DIAMOND -> System.out.println("Diamonds or Carreaux");
        case Tarot.CLUB -> System.out.println("Clubs or Trefles");
        case Tarot.TRUMP -> System.out.println("Trumps or Atouts");
        case Tarot.EXCUSE -> System.out.println("The Fool or L'Excuse");
    }
}
```

Therefore, you can use an enum constant when the type of the selector expression is not an enum type provided that the enum constant's name is qualified and its value is assignment-compatible with the type of the selector expression.

Primitive Values in switch Expressions and Statements

In addition to the primitive types char, byte, short, and int, a switch expression or statement's selector expression can be of type long, float, double, and boolean, as well as the corresponding boxed types.

Note

This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [Preview Language and VM Features](#).
See [JEP 530: Primitive Types in Patterns, instanceof, and switch \(Fourth Preview\)](#) for additional information.

If a selector expression is of type long, float, double, and boolean, then its case labels must have the same type as the selector expression or its corresponding boxed type. For example:

```
void whichFloat(float v) {
    switch (v) {
        case 0f ->
            System.out.println("Zero");
        case float x when x > 0f && x <= 10f ->
            System.out.println(x + " is between 1 and 10");
        case float x ->
            System.out.println(x + " is larger than 10");
    }
}
```

If you change the case label `0f` to `0`, you would get the following compile-time error:

```
error: constant label of type int is not compatible with switch selector type float
```

You can't use two floating-point literals as case labels that are *representationally equivalent*. (See the JavaDoc API documentation for the [Double](#) class for more information about representation equivalence.) The following example generates a compiler error. The value `0.999999999f` is representationally equivalent to `1.0f`:

```
void duplicateLabels(float v) {
    switch (v) {
        case 1.0f -> System.out.println("One");
        // error: duplicate case label
        case 0.999999999f -> System.out.println("Almost one");
        default -> System.out.println("Another float value");
    }
}
```

Switching on boolean values is a useful alternative to the ternary conditional operator (`?:`) because a boolean switch expression or statement can contain statements as well as expressions. For example, the following code uses a boolean switch expression to perform some logging when false:

```
long startProcessing(OrderStatus.NEW, switch (user.isLoggedIn()) {
    case true -> user.id();
    case false -> { log("Unrecognized user"); yield -1L; }
});
```

Exhaustiveness of switch

The cases of a switch expression or statement must be *exhaustive*, which means that for all possible values, there must be a matching case label. Thus, a switch expression or statement normally require a default clause. However, for an enum switch expression that covers all known constants, the compiler inserts an implicit default clause, like the examples at the beginning of this section that print the number of letters in name of a day of the week.

The cases of a switch statement must be exhaustive if it uses pattern or null labels. See [Pattern Matching with switch](#) and [Null case Labels](#) for more information.

The following switch expression is not exhaustive and generates a compile-time error. The type coverage of its labels consist of the subtypes of `String` and `Integer`. However, it doesn't include the type of the selector expression, `Object`:

```
static int coverage(Object obj) {
    return switch (obj) {
        case String s -> s.length();
        case Integer i -> i;
    };
}
```

However, the type coverage of the case label default is all types, so the following example compiles:

```
static int coverage(Object obj) {
    return switch (obj) {
        case String s -> s.length();
        case Integer i -> i;
        default      -> 0;
    };
}
```

Consequently, for a switch expression or statement to be exhaustive, the type coverage of its labels must include the type of the selector expression.

The compiler takes into account whether the type of a selector expression is a sealed class. The following switch expression compiles. It doesn't need a default case label because its type coverage is the classes A, B, and C, which are the only permitted subclasses of S, the type of the selector expression:

```
sealed interface S permits A, B, C { }
final class A implements S { }
final class B implements S { }
record C(int i) implements S { } // Implicitly final
//...

static int testSealedCoverage(S s) {
    return switch (s) {
        case A a -> 1;
        case B b -> 2;
        case C c -> 3;
    };
}
```

The compiler can also determine the type coverage of a switch expression or statement if the type of its selector expression is a generic sealed class. The following example compiles. The only permitted subclasses of interface I are classes A and B. However, because the selector expression is of type I<Integer>, the switch block requires only class B in its type coverage to be exhaustive:

```
sealed interface I<T> permits A, B {}
final class A<X> implements I<String> {}
final class B<Y> implements I<Y> {}

static int testGenericSealedExhaustive(I<Integer> i) {
    return switch (i) {
        // Exhaustive as no A case possible!
        case B<Integer> bi -> 42;
    };
}
```

The type of a switch expression or statement's selector expression can also be a generic record. As always, a switch expression or statement must be exhaustive. The following example doesn't compile. No match for a Pair exists that contains two values, both of type A:

```
record Pair<T>(T x, T y) {}
class A {}
class B extends A {}

static void notExhaustive(Pair<A> p) {
    switch (p) {
        // error: the switch statement does not cover all possible input
        values
        case Pair<A>(A a, B b) -> System.out.println("Pair<A>(A a, B b)");
        case Pair<A>(B b, A a) -> System.out.println("Pair<A>(B b, A a)");
    }
}
```

The following example compiles. Interface I is sealed. Types C and D cover all possible instances:

```
record Pair<T>(T x, T y) {}
sealed interface I permits C, D {}
record C(String s) implements I {}
record D(String s) implements I {}
// ...

static void exhaustiveSwitch(Pair<I> p) {
    switch (p) {
        case Pair<I>(I i, C c) -> System.out.println("C = " + c.s());
        case Pair<I>(I i, D d) -> System.out.println("D = " + d.s());
    }
}
```

If a switch expression or statement is exhaustive at compile time but *not* at run time, then a `MatchException` is thrown. This can happen when a class that contains an exhaustive switch expression or statement has been compiled, but a sealed hierarchy that is used in the analysis of the switch expression or statement has been subsequently changed and recompiled. Such changes are *migration incompatible* and may lead to a `MatchException` being thrown when running the switch statement or expression. Consequently, you need to recompile the class containing the switch expression or statement.

Consider the following class `PrintA` and sealed interface `OnlyAB`:

```
class PrintA {
    public static void main(String[] args) {
        System.out.println(switch (OnlyAB.getAValue()) {
            case A a -> 1;
            case B b -> 2;
        });
    }
}

sealed interface OnlyAB permits A, B {
    static OnlyAB getAValue() {
```

```

        return new A();
    }
}
final class A implements OnlyAB {}
final class B implements OnlyAB {}

```

The switch expression in the class PrintA is exhaustive and this example compiles. When you run PrintA, it prints the value 1. However, suppose you edit OnlyAB as follows and compile this interface and *not* PrintA:

```

sealed interface OnlyAB permits A, B, C {
    static OnlyAB getAValue() {
        return new A();
    }
}
final class A implements OnlyAB {}
final class B implements OnlyAB {}
final class C implements OnlyAB {}

```

When you run PrintA, it throws a MatchException:

```

Exception in thread "main" java.lang.MatchException
    at PrintA.main(PrintA.java:3)

```

Completion and switch Expressions

A switch expression must either complete normally with a value or complete abruptly by throwing an exception

A. For example, the following code doesn't compile because the switch labeled rule doesn't contain a yield statement:

```

int i = switch (day) {
    case MONDAY -> {
        System.out.println("Monday");
        // error: block doesn't contain a yield statement
    }
    default -> 1;
};

```

The following example doesn't compile because the switch labeled statement group doesn't contain a yield statement:

```

i = switch (day) {
    case MONDAY, TUESDAY, WEDNESDAY:
        yield 0;
    default:
        System.out.println("Second half of the week");
        // error: group doesn't contain a yield statement
};

```

Because a switch expression must evaluate to a single value (or throw an exception), you can't jump through a switch expression with a break, yield, return, or continue statement, like in the following example:

```
z:
    for (int i = 0; i < MAX_VALUE; ++i) {
        int k = switch (e) {
            case 0:
                yield 1;
            case 1:
                yield 2;
            default:
                continue z;
                // error: illegal jump through a switch expression
        };
    }
// ...
}
```

13

Text Blocks

JEP 378 adds the language feature *text blocks* to Java SE 15 and later. While the JEP explains the feature in great detail, it's not always apparent how the feature can and should be used. This guide assembles practical usage advice for text blocks, along with some style guidelines.

Introduction

A text block's *principalis munus* is to provide clarity by way of minimizing the Java syntax required to render a string that spans multiple lines.

In earlier releases of the JDK, embedding multiline code snippets required a tangled mess of explicit line terminators, string concatenations, and delimiters. Text blocks eliminate most of these obstructions, allowing you to embed code snippets and text sequences more or less as-is.

A text block is an alternative form of Java string representation that can be used anywhere a traditional double-quoted string literal can be used. For example:

```
// Using a literal string
String dqName = "Pat Q. Smith";

// Using a text block
String tbName = """
    Pat Q. Smith""";
```

The object produced from a text block is a `java.lang.String` with the same characteristics as a traditional double-quoted string. This includes object representation and interning. Continuing with `dqName` and `tbName` from the previous examples:

```
// Both dqName and tbName are strings of equal value
dqName.equals(tbName)    // true

// Both dqName and tbName intern to the same string
dqName == tbName         // true
```

Text blocks can be used anywhere a string literal can be used. For example, text blocks may be intermixed with string literals in a string concatenation expression:

```
String str = "The old";
String tb = """
    the new""";
String together = str + " and " + tb + ".";
```

Text blocks may be used as a method argument:

```
System.out.println("""
    This is the first line
```

```

    This is the second line
    This is the third line
    """);

```

String methods may be applied to a text block:

```

""
John Q. Smith"".substring(8).equals("Smith")    // true

```

A text block can be used in place of a string literal to improve the readability and clarity of the code. This primarily occurs when a string literal is used to represent a multiline string. In this case, there is considerable clutter from quotation marks, newline escapes, and concatenation operators:

```

// ORIGINAL
String message = "'The time has come,' the Walrus said,\n" +
    "'To talk of many things:\n" +
    "Of shoes -- and ships -- and sealing-wax --\n" +
    "Of cabbages -- and kings --\n" +
    "And why the sea is boiling hot --\n" +
    "And whether pigs have wings.\n";

```

Using text blocks removes much of the clutter:

```

// BETTER
String message = ""
    'The time has come,' the Walrus said,
    'To talk of many things:
    Of shoes -- and ships -- and sealing-wax --
    Of cabbages -- and kings --
    And why the sea is boiling hot --
    And whether pigs have wings.'
    "";

```

Text Block Syntax

A text block begins with three double-quote characters followed by a line terminator. You can't put a text block on a single line, nor can the contents of the text block follow the three opening double-quotes without an intervening line terminator. The reason for this is that text blocks are primarily designed to support multiline strings, and requiring the initial line terminator simplifies the indentation handling rules. See [Incidental White Space](#).

```

// ERROR
String name = ""Pat Q. Smith"";

```

```

// ERROR
String name = ""red
                green
                blue
                "";

```

```

// OK
String name = ""
    red

```

```
green
blue
"";
```

This last example is equivalent to the following string literal:

```
String name = "red\n" +
              "green\n" +
              "blue\n";
```

Here's an example of a snippet of Java code within a text block:

```
String source = ""
    String message = "Hello, World!";
    System.out.println(message);
    "";
```

Note that there is no need to escape the embedded double quotes. The equivalent string literal would be:

```
String source = "String message = \"Hello, World!\";\n" +
                "System.out.println(message);\n";
```

That Final New Line

Note that this example is equivalent to "red\ngreen\nblue\n":

```
String name = ""
    red
    green
    blue
    "";
```

What if you want to represent a multiline string without that final \n?

```
String name = ""
    red
    green
    blue"";
```

This text block is equivalent to is "red\ngreen\nblue". Thus, placing the closing delimiter on the last visible line effectively drops the last \n.

Alternatively, you can use the <line-terminator> escape sequence to suppresses the inclusion of an implicit new line character:

```
String name = ""
    red
    green
    blue\
    "";
```

See [The \<line-terminator> and \s Escape Sequences](#).

Incidental White Space

Ideally, a text block would be indented to match the indentation of the surrounding code. For example:

```
void writeHTML() {
    String html = ""
        <html>
            <body>
                <p>Hello World.</p>
            </body>
        </html>
        "";
    writeOutput(html);
}
```

However, this raises the question of how spaces used for indentation affect the contents of the string. A naive interpretation would include all of this whitespace in the text block. The consequence would be that reindenting the code would affect the contents of the text block. This is quite likely to be an error.

To avoid this problem, a text block differentiates *incidental* white space from *essential* white space. The Java compiler automatically strips away the incidental white space. The indentation to the left of <html> and </html> is considered incidental because these lines are indented the least. Thus, they effectively determine the left margin of the text in the text block. However, the indentation of <body> relative to <html> is *not* considered to be incidental white space. Presumably, this relative indentation is intended to be part of the string's contents.

The following example uses the middle dot character (·) to visualize the incidental white space, with essential white space shown as actual white space.

```
void writeHTML() {
    String html = ""
    .....<html>
    .....    <body>
    .....        <p>Hello World.</p>
    .....    </body>
    .....</html>
    ....."";
    writeOutput(html);
}
```

After the incidental white space is stripped, the resulting contents of the text block are as follows:

```
<html>
    <body>
        <p>Hello World.</p>
    </body>
</html>
```

The algorithm for determining incidental white space is described in JEP 378 in detail. Nevertheless, the net effect is quite simple. The entire contents of the text block is shifted to the left until the line with the least leading white space has no leading white space.

To preserve some white space and not have it be considered incidental white space, simply shift the content lines of the text block to the right, while keeping the closing triple-quote delimiter at the indentation appropriate for the surrounding code. For example:

```
void writeHTML() {
    String html = ""
    .....    <html>
    .....        <body>
    .....            <p>Hello World.</p>
    .....        </body>
    .....    </html>
    ....."";
    writeOutput(html);
}
```

results in the following:

```
<html>
    <body>
        <p>Hello World.</p>
    </body>
</html>
```

A text block can *opt out* of incidental white space stripping by positioning the closing delimiter in the first character position of a source line:

```
void writeHTML() {
    String html = ""
        <html>
        <body>
            <p>Hello World.</p>
        </body>
    </html>
"";
    writeOutput(html);
}
```

The result is that there is no incidental white space that is stripped, and the string includes leading white space on each line:

```
<html>
    <body>
        <p>Hello World.</p>
    </body>
</html>
```

This technique for controlling the amount of indentation that is preserved only works if the last line of the text block ends with a line terminator. If the last line does *not* end with a line

terminator, you need to use `String::indent` to control the indentation explicitly. In the following example,

```
String colors = ""  
    red  
    green  
    blue"";
```

all of the indentation is treated as incidental and is stripped away:

```
red  
green  
blue
```

To include some indentation in the string's contents, invoke the `indent` method on the text block:

```
String colors = ""  
    red  
    green  
    blue"".indent(4);
```

This results in:

```
    red  
    green  
    blue
```

Trailing White Space

Trailing white space on each line in a text block is also considered incidental and is stripped away by the Java compiler. This is done so that the contents of the text block are always visually discernible. If this were not done, a text editor that automatically strips trailing white space could invisibly change the contents of a text block.

If you need to have trailing white space in a text block, then you can use one of the following strategies:

```
// character substitution  
String r = ""  
    trailing$$$  
    white space  
    """.replace('$', ' ');
```

```
// character fence  
String s = ""  
    trailing |  
    white space|  
    """.replace("|\\n", "\\n");
```

```
// octal escape sequence for space  
String t = ""
```

```

        trailing\040\040\040
        white space
        """,

// \s escape sequence
String t = \s\s\s
        trailing\040\040\040
        white space
        """,

```

① Note

\u0020 cannot be used because Unicode escapes are translated early during source file reading, prior to lexical analysis. By contrast, character and string escapes such as \040 are processed after lexical analysis has divided the source file into tokens and has identified string literals and text blocks.

Detecting Potential Issues with White Space

In the preceding examples, all indentation consisted of space characters. However, sometimes people use TAB \t characters. Unfortunately it is not possible for the Java compiler to know how tab characters are displayed in different editors. Therefore, the rule is that each individual white space character is treated equally. A single space character is treated the same as a single tab character, even though the latter might result in white space equivalent up to eight spaces when displayed on some particular system.

It follows that mixing white space characters can have inconsistent and unintended effects. Consider the following example, in which some lines are indented with spaces and some with tabs, which are visualized with the symbol for horizontal tabulation (†):

```

        String colors = ""
        .....red
†   †   †   †   †   green
        .....blue""";

```

In this case, stripping of incidental indentation would be uneven because the second line only has five white space characters and the others have twenty. The result would look something like this:

```

                red
green
                blue

```

It is possible to detect issues related to incidental white space by turning on text block lint detection with a Java compiler lint flag, `-Xlint:text-blocks`. If lint detection is on, then this example will generate a warning, "inconsistent white space indentation".

This lint flag also enables another warning, "trailing white space will be removed", which will be emitted if there is trailing white space on any line within a text block. If you need to preserve trailing white space, use one of the escaping or replacement techniques described in the section [Trailing White Space](#).

Normalization of Line Terminators

One of the complications of a multiline string literal is that the line terminator (\n, \r, or \r\n) used in the source file varies from platform to platform. Editors on different platforms may

invisibly change line terminators. Or, if a source file is edited on different platforms, a text block might contain a mixture of different line terminators. This is likely to produce confusing and inconsistent results.

To avoid these problems, the Java compiler normalizes all line terminators in a text block to be `\n`, regardless of what line terminators actually appear in the source file. The following text block, where the symbol for line feed (`␣`) and the symbol for carriage return (`↵`) represent `\n` and `\r`, respectively:

```
String colors = ""
    red␣
    green↵
    blue␣↵
    "";
```

is equivalent to this string literal:

```
String colors = "red\ngreen\nblue\n";
```

If the platform line terminator is required, then `String::replaceAll("\n", System.lineSeparator())` can be used.

Translation of Escape Sequences

As with string literals, text blocks recognize the escape sequences, `\b`, `\f`, `\n`, `\t`, `\r`, `\s`, `\"`, `\'`, `\\`, `<line-terminator>`, and octal escapes. Unlike string literals, escapes sequences are often not required. Under most circumstances, the actual characters `\n`, `\t`, `\"`, and `\'` can be used instead of escape sequences. The following text block, where `␣` and `↵` represent `\t` and `\n`:

```
String s = ""
    Color␣    Shape↵
    Red␣ ␣    Circle↵
    Green␣    Square↵
    Blue␣␣    Triangle↵
    "";
```

results in:

```
Color␣    Shape↵
Red␣ ␣    Circle↵
Green␣    Square↵
Blue␣␣    Triangle↵
```

Escaping is required when three or more double quotes occur consecutively.

```
String code = ""
    String source = \"""
        String message = "Hello, World!";
        System.out.println(message);
        \"""
    "";
```

Escape translation occurs as the last step of processing by the Java compiler, so you can bypass the line terminator normalization and whitespace stripping steps by using explicit escape sequences. For example:

```
String s = ""
    red  \040
```

```
green\040
blue \040
""";
```

would guarantee that all lines are of equal length because the `\040` does not get translated to a space until after stripping of trailing white space. The middle dot character (`·`) is used to show trailing space. The result is:

```
red...
green·
blue..
```

Note

As noted previously, the Unicode escape sequence `\u0020` *cannot* be used as a substitute for `\040`.

The `<line-terminator>` and `ls` Escape Sequences

The `<line-terminator>` escape sequence explicitly suppresses the inclusion of an implicit new line character.

For example, it is common practice to split very long string literals into concatenations of smaller substrings and then hard wrapping the resulting string expression onto multiple lines.

```
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
                 "elit, sed do eiusmod tempor incididunt ut labore " +
                 "et dolore magna aliqua.";
```

With the `<line-terminator>` escape sequence, this could be expressed as:

```
String text = ""
             Lorem ipsum dolor sit amet, consectetur adipiscing \
             elit, sed do eiusmod tempor incididunt ut labore \
             et dolore magna aliqua.\
             """;
```

The `\s` escape sequence simply translates to space (`\040`, ASCII character 32, white space.) Because escape sequences don't get translated until after incident space stripping, `\s` can act as fence to prevent the stripping of trailing white space. Using `\s` at the end of each line in the following example, guarantees that each line is exactly six characters long:

```
String colors = ""
               red \s
               green\s
               blue \s
               """;
```

Style Guidelines for Text Blocks

G1. You should use a text block when it improves the clarity of the code, particularly with multiline strings.

```
// ORIGINAL
String message = "'The time has come,' the Walrus said,\n" +
    "'To talk of many things:\n" +
    "Of shoes -- and ships -- and sealing-wax --\n" +
    "Of cabbages -- and kings --\n" +
    "And why the sea is boiling hot --\n" +
    "And whether pigs have wings.'\n";

// BETTER
String message = """
    'The time has come,' the Walrus said,
    'To talk of many things:
    Of shoes -- and ships -- and sealing-wax --
    Of cabbages -- and kings --
    And why the sea is boiling hot --
    And whether pigs have wings.'
    """;
```

G2. If a string fits on a single line, without concatenation and escaped newlines, you should probably continue to use a string literal.

```
// ORIGINAL - is a text block helpful here?
String name = """
    Pat Q. Smith""";

// BETTER - a string literal works fine
String name = "Pat Q. Smith";
```

G3. Use embedded escape sequences when they maintain readability.

```
var data = """
    Name | Address | City
    Bob Smith | 123 Anytown St\nApt 100 | Vancouver
    Jon Brown | 1000 Golden Place\nSuite 5 | Santa Ana
    """;
```

G4. For most multiline strings, place the opening delimiter at the right end of the previous line, and place the closing delimiter on its own line, at the left margin of the text block.

```
String string = """
    red
    green
    blue
    """;
```

G5. Avoid aligning the opening and closing delimiters and the text block's left margin. This requires reindentation of the text block if the variable name or modifiers are changed.

```
// ORIGINAL
String string = ""
    red
    green
    blue
    "";

// ORIGINAL - after variable declaration changes
static String rgbNames = ""
    red
    green
    blue
    "";

// BETTER
String string = ""
    red
    green
    blue
    "";

// BETTER - after variable declaration changes
static String rgbNames = ""
    red
    green
    blue
    ";
```

G6. Avoid in-line text blocks within complex expressions, as doing so can distort readability. Consider refactoring to a local variable or to a static final field.

```
// ORIGINAL
String poem = new String(Files.readAllBytes(Paths.get("jabberwocky.txt")));
String middleVerses = Pattern.compile("\\n\\n")
    .splitAsStream(poem)
    .match(verse -> !""
        'Twas brillig, and the slithy toves
        Did gyre and gimble in the wabe;
        All mimsy were the borogoves,
        And the mome raths outgrabe.
        "").equals(verse))
    .collect(Collectors.joining("\\n\\n"));

// BETTER
String firstLastVerse = ""
    'Twas brillig, and the slithy toves
    Did gyre and gimble in the wabe;
    All mimsy were the borogoves,
    And the mome raths outgrabe.
    "";
String poem = new String(Files.readAllBytes(Paths.get("jabberwocky.txt"));
```

```
String middleVerses = Pattern.compile("\\n\\n")
    .splitAsStream(poem)
    .match(verse -> !firstLastVerse.equals(verse))
    .collect(Collectors.joining("\\n\\n"));
```

G7. Either use only spaces or only tabs for the indentation of a text block. Mixing white space will lead to a result with irregular indentation.

```
// ORIGINAL
String colors = ""
.....red
    green
.....blue"";    // result: ".....red\\ngreen\\n.....blue"

// PROBABLY WHAT WAS INTENDED
String colors = ""
.....red
.....green
.....blue"";    // result: "red\\ngreen\\nblue"
```

G8. When a text block contains sequences of three or more double quotes, escape the first double quote of every run of three double quotes.

```
// ORIGINAL
String code = ""
String source = "\\\"\\\"\\\"
    String message = "Hello, World!";
    System.out.println(message);
    "\\\"\\\"\\\";
    """;

// BETTER
String code = ""
String source = ""
    String message = "Hello, World!";
    System.out.println(message);
    "";
    """;
```

G9. Most text blocks should be indented to align with neighboring Java code.

```
// ORIGINAL - odd indentation
void printPoem() {
    String poem = ""
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
""";
    System.out.print(poem);
}

// BETTER
void printPoem() {
```

```

        String poem = ""
        'Twas brillig, and the slithy toves
        Did gyre and gimble in the wabe;
        All mimsy were the borogoves,
        And the mome raths outgrabe.
        "";
        System.out.print(poem);
    }

```

G10. It is recommended to fully left justify a wide string in order to avoid horizontal scrolling or line wrapping.

// ORIGINAL

```

class Outer {
    class Inner {
        void printPoetry() {
            String lilacs = ""
                Over the breast of the spring, the land, amid cities,
                Amid lanes and through old woods, where lately the violets
            peep'd from the ground, spotting the gray debris,
                Amid the grass in the fields each side of the lanes, passing
            the endless grass,
                Passing the yellow-spear'd wheat, every grain from its shroud
            in the dark-brown fields uprisen,
                Passing the apple-tree blows of white and pink in the
            orchards,
                Carrying a corpse to where it shall rest in the grave,
                Night and day journeys a coffin.
            "";
            System.out.println(lilacs);
        }
    }
}

```

// BETTER

```

class Outer {
    class Inner {
        void printPoetry() {
            String lilacs = ""
            Over the breast of the spring, the land, amid cities,
            Amid lanes and through old woods, where lately the violets peep'd from the
            ground, spotting the gray debris,
            Amid the grass in the fields each side of the lanes, passing the endless
            grass,
            Passing the yellow-spear'd wheat, every grain from its shroud in the dark-
            brown fields uprisen,
            Passing the apple-tree blows of white and pink in the orchards,
            Carrying a corpse to where it shall rest in the grave,
            Night and day journeys a coffin.
            "";
            System.out.println(lilacs);
        }
    }
}

```

```
    }
}
```

G11. Similarly, it is also reasonable to fully left justify a text block when a high line count causes the closing delimiter is likely to vertically scroll out of view. This allows the reader to track indentation with the left margin when the closing delimiter is out of view.

```
// ORIGINAL
```

```
String validWords = ""
    aa
    aah
    aahed
    aahing
    aahs
    aal
    aalii
    aaliis
...
    zythum
    zythums
    zyzzyva
    zyzzyvas
    zzz
    zzzs
    "";
```

```
// BETTER
```

```
String validWords = ""
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
...
zythum
zythums
zyzzyva
zyzzyvas
zzz
zzzs
"";
```

G12. The `\<line-terminator>` escape sequence should be used when a text block's final new line needs to be excluded. This better frames the text block and allows the closing delimiter to manage indentation.

```
// ORIGINAL
```

```
String name = ""
```

```
    red
    green
    blue"".indent(4);

// BETTER
```

```
String name = ""
    red
    green
    blue\
    "",
```

String Methods Related to Text Blocks

Several new methods are included in the `String` class as part of the text blocks feature.

- `String formatted(Object... args)`: This method is equivalent to `String.format(this, args)`. The advantage is that, as an instance method, it can be chained off the end of a text block:

```
String output = ""
    Name: %s
    Phone: %s
    Address: %s
    Salary: $%.2f
    "".formatted(name, phone, address, salary);
```

- `String stripIndent()`: The `stripIndent` method removes incidental white space from a multiline string, using the same algorithm used by the Java compiler. This is useful if you have a program that reads text as input data and you want to strip indentation in the same manner as is done for text blocks.
- `String translateEscapes()`: The `translateEscapes` method performs the translation of escape sequences (`\b`, `\f`, `\n`, `\t`, `\r`, `\s`, `\"`, `\'`, `\\`, `<line-terminator>`, and octal escapes) and is used by the Java compiler to process text blocks and string literals. This is useful if you have a program that reads text as input data and you want to perform escape sequence processing.

Note

Unicode escapes (`\uNNNN`) are *not* processed.

References

"The Walrus and the Carpenter"

Lewis Carroll, *Through the Looking-Glass and What Alice Found There*, 1872.

"Jabberwocky"

Lewis Carroll, *Mischmasch*, 1855.

"When Lilacs Last in the Dooryard Bloom'd"

Walt Whitman, *Sequel to Drum-Taps*, 1865.

Local Variable Type Inference

In JDK 10 and later, you can declare local variables with non-null initializers with the `var` identifier, which can help you write code that's easier to read.

Consider the following example, which seems redundant and is hard to read:

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

You can rewrite this example by declaring the local variables with the `var` identifier. The type of the variables are inferred from the context:

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

`var` is a reserved type name, not a keyword, which means that existing code that uses `var` as a variable, method, or package name is not affected. However, code that uses `var` as a class or interface name is affected and the class or interface needs to be renamed.

`var` can be used for the following types of variables:

- Local variable declarations with initializers:

```
var list = new ArrayList<String>();    // infers ArrayList<String>
var stream = list.stream();           // infers Stream<String>
var path = Paths.get(fileName);       // infers Path
var bytes = Files.readAllBytes(path); // infers bytes[]
```

- Enhanced for-loop indexes:

```
List<String> myList = Arrays.asList("a", "b", "c");
for (var element : myList) {...} // infers String
```

- Index variables declared in traditional for loops:

```
for (var counter = 0; counter < 10; counter++) {...} // infers int
```

- try-with-resources variable:

```
try (var input =
    new FileInputStream("validation.txt")) {...} // infers
FileInputStream
```

- Formal parameter declarations of implicitly typed lambda expressions: A lambda expression whose formal parameters have inferred types is *implicitly typed*:

```
BiFunction<Integer, Integer, Integer> = (a, b) -> a + b;
```

In JDK 11 and later, you can declare each formal parameter of an implicitly typed lambda expression with the `var` identifier:

```
(var a, var b) -> a + b;
```

As a result, the syntax of a formal parameter declaration in an implicitly typed lambda expression is consistent with the syntax of a local variable declaration; applying the `var` identifier to each formal parameter in an implicitly typed lambda expression has the same effect as not using `var` at all.

You cannot mix inferred formal parameters and `var`-declared formal parameters in implicitly typed lambda expressions nor can you mix `var`-declared formal parameters and manifest types in explicitly typed lambda expressions. The following examples are not permitted:

```
(var x, y) -> x.process(y)      // Cannot mix var and inferred formal
parameters                      // in implicitly typed lambda expressions
(var x, int y) -> x.process(y) // Cannot mix var and manifest types
// in explicitly typed lambda expressions
```

Local Variable Type Inference Style Guidelines

Local variable declarations can make code more readable by eliminating redundant information. However, it can also make code less readable by omitting useful information. Consequently, use this feature with judgment; no strict rule exists about when it should and shouldn't be used.

Local variable declarations don't exist in isolation; the surrounding code can affect or even overwhelm the effects of `var` declarations. [Local Variable Type Inference: Style Guidelines](#) examines the impact that surrounding code has on `var` declarations, explains tradeoffs between explicit and implicit type declarations, and provides guidelines for the effective use of `var` declarations.

Safe Casting with instanceof and switch

You can use the `instanceof` operator to check at run time whether a cast from one type to another is *safe*. A cast or conversion is safe if there's no loss of information or exception thrown during the cast or conversion. You can use the `instanceof` operator to test whether a cast is safe between two reference types or two primitive types. In addition, conversion safety affects how `switch` statements and expressions behave.

Note

The ability to use the `instanceof` operator to test whether a cast is safe between any two types, and not only between two reference types, is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See [Preview Language and VM Features](#). See [JEP 530: Primitive Types in Patterns, instanceof, and switch \(Fourth Preview\)](#) for additional information.

The following example uses `instanceof` as the type comparison operator to test whether the `Shape s` is a `Circle`. If it is, then it's safe to cast `Shape s` to a `Circle`, which means that there's definitely a radius that you can print:

```
public interface Shape { }
record Rectangle(double length, double width) implements Shape { }
record Circle(double radius) implements Shape { }
//...

Shape s = new Rectangle(4,3);
if (s instanceof Circle) {
    Circle c = (Circle)s;
    System.out.println("Radius: " + c.radius());
}
```

The Java runtime throws a `ClassCastException` if it can't cast a reference type to another reference type. If you remove the `s instanceof Circle` expression from the example, the Java runtime throws an exception when it tries to cast the `Shape`, which is a `Rectangle`, to a `Circle`:

```
Shape s = new Rectangle(4,3);

// ClassCastException: class Rectangle cannot be cast
// to class Circle
Circle c = (Circle)s;

System.out.println("Radius: " + c.radius());
```

When it comes to casting between primitive types, the Java runtime won't ever throw a `ClassCastException`. It will convert the value (if the types are compatible), which might result in the loss of information about the value's overall magnitude as well as its precision and range. The following example casts an `int` to a `byte`:

```
int i = 2345323;
byte b = (byte)i;
System.out.println(b);
```

It prints the following output:

```
107
```

An improperly or unintendedly converted value, such as from 2345323 to 107, or more subtly, from 2.05f to 2, can silently propagate through a program, causing potentially elusive bugs.

As the primitive types `int` and `byte` differ by their ranges, you could test that before casting the value:

```
if (i >= -128 && i <= 127) {
    System.out.println((byte)i);
}
```

However, a clearer and more concise way to test whether a cast between two primitive type is safe is to use `instanceof` as the type comparison operator:

```
if (i instanceof byte) {
    System.out.println((byte)i);
}
```

Note that this example still has to cast the value `i` to a `byte`. The `instanceof` type comparison operator only checks if `i` is a `byte`. It doesn't perform the actual cast.

This also works with `instanceof` as the pattern match operator, which has the added benefit of performing the cast for you if the test is successful:

```
if (i instanceof byte b) {
    System.out.println(b);
}
```

Conversion Safety and switch

Consider the following example:

```
void safeConversion(float f) {
    switch (f) {
        case double d -> System.out.println("As double: " + d);
    }
}
```

The case label `case double d` converts the `float` value of the `switch` statement's selector expression to a `double` value. This conversion is *safe*, which means that during the conversion,

there's no possibility that information may be lost about the value's overall magnitude as well as its precision and range. This switch statement works for all float values.

Consider the following example, which doesn't compile.

```
void unsafeConversion(float f) {  
    // error: the switch statement does not cover all  
    // possible input values  
    switch (f) {  
        case int i -> System.out.println("As int: " + i);  
    }  
}
```

The case label `case int i` converts the float value of the switch statement's selector expression to an int value. This conversion is not always safe, which means that there's a possibility that information may be lost about the value's overall magnitude as well as its precision and range. If this switch statement were permitted, then an improperly or unintentionally converted value, such as from `100.01f` to `100`, can propagate through a program, causing subtle bugs. To help prevent this from happening, the compiler generates an error.

In the `safeConversion` example, the conversion from a float value to a double value is an *unconditionally exact conversion*, which means that it's known at compile time that the conversion will not lose information at run time. Whether a conversion between primitive types is unconditionally exact affects the exhaustiveness of switch and how of patterns dominate other patterns. See [Exhaustiveness of switch](#) and [Dominance and Unconditionally Exact Conversions](#) for more information.

Note

Converting a float value to a double value is an example of a *type-based unconditionally exact conversion*, which is a conversion where there will be no loss of information regardless of the value of the expression to be converted and the target type. There's another kind of unconditionally exact conversion called a *value-based unconditionally exact conversion*, which is a conversion where there can be information loss in general because of the type of the expression to be converted and the target type, but the value to be converted is a constant expression. In such cases, the compiler is able to test whether the conversion results in no loss of information. For example, converting an int value of 42 to a byte value is unconditionally exact. The compiler can find case labels that dominate other case labels that are constants because of this kind of conversion. See [Dominance and Unconditionally Exact Conversions](#) for an example. See [5.7.1. Exactness of Testing Conversions](#) in *The Java Language Specification* for more information.