Java Platform, Standard Edition

Java Language Updates

Release 21
F79543-01
September 2023

ORACLE

Java Platform, Standard Edition Java Language Updates, Release 21

F79543-01

Contents

Preface

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

Audience

This document is for Java developers.

Documentation Accessibility

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.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See JDK 21 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.
italic	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

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

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
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)

Feature	Description	JEP
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 `MatchException` instead of an `IncompatibleClassChangeError` 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)

Feature	Description	JEP
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 `NullPointerException` 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 *not* at run time, then a `MatchException` 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

Feature	Description	JEP
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

Feature	Description	JEP
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 `instanceof` expression compares an expression of type *`S`* with a pattern of type *`T`*, where *`S`* is a subtype of *`T`*. 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)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)
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 375: Pattern Matching for instanceof (Second 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 `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 361: Switch Expressions (Standard)

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

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 programing 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

Feature	Description	JEP
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.

2
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.

3
String Templates

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. An *embedded expression* is a Java expression except it has additional syntax to differentiate it from the literal text in the string template. A *template processor* combines the literal text in the template with the values of the embedded expressions to produce a result.

See the `StringTemplate` interface in the Java SE API specification for more information.

> **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.
> For background information about string templates, see JEP 430.

Basic Usage of String Templates

The following example declares a template expression that uses the template processor `STR` and contains one embedded expression, `name`:

```
String name = "Duke";
String info = STR."My name is \{name}";
System.out.println(info);
```

It prints the following:

```
My name is Duke
```

The template processor `STR` is one of the template processors that's included in the JDK. It automatically performs *string interpolation* by replacing each embedded expression in the template with its value, converted to a string. The JDK includes two other template processors:

- The FMT Template Processor: It's like the `STR` template processor except that it accepts format specifiers as defined in `java.util.Formatter` and locale information in a similar way as in `printf` method invocations.
- The RAW Template Processor: It doesn't automatically process the string template like the `STR` template processors. You can use it to help you create your own template processors. Note that you can also implement the `StringTemplate.Processor` interface to create a template processor. See Creating a Template Processor.

A dot (`.`) follows the template processor, which makes template expressions look similar to accessing an object's field or calling an object's method.

A string template follows the dot character, which is a string that contains one or more embedded expressions. An embedded expression is a Java expression that's surrounded by a backslash and opening brace (`\{`) and a closing brace `}`. See Embedded Expressions in String Templates for additional examples.

Embedded Expressions in String Templates

You can use any Java expression as an embedded expression in a string template.

You can use strings and characters as embedded expressions.

```
String user = "Duke";
char option = 'b';
String choice  = STR."\{user} has chosen option \{option}";
System.out.println(choice);
```

This example prints the following:

```
Duke has chosen option b
```

Embedded expressions can perform arithmetic operations.

```
double x = 10.5, y = 20.6;
String p = STR."\{x} * \{y} = \{x * y}";
System.out.println(p);
```

This example prints the following:

```
10.5 * 20.6 = 216.3
```

As with any Java expression, embedded expressions are evaluated from left to right. They can also contain prefix and postfix operators.

```
int index = 0;
String data = STR."\{index++}, \{index++}, \{++index}, \{index++}, \
{index}";
System.out.println(data);
```

This example prints the following:

```
0, 1, 3, 3, 4
```

Embedded expressions can invoke methods and access fields.

```
String time = STR."Today is \{java.time.LocalDate.now()}";
System.out.println(time);
String canLang = STR."The language code of \{
    Locale.CANADA_FRENCH} is \{
```

```
    Locale.CANADA_FRENCH.getLanguage()}";
System.out.println(canLang);
```

This example prints output similar to the following:

```
Today is 2023-07-11
The language code of fr_CA is fr
```

Note that you can insert line breaks in an embedded expression and not introduce newlines in the result like in the previous example. In addition, you don't have to escape quotation marks in an embedded expression.

```
Path filePath = Paths.get("Stemp.java");
String msg = STR."The file \{filePath} \{
    // The Files class is in the package java.nio.file
    Files.exists(filePath) ? "does" : "does not"} exist";
System.out.println(msg);

String currentTime = STR."The time is \{
    DateTimeFormatter
        .ofPattern("HH:mm:ss")
        .format(LocalTime.now())
} right now";
System.out.println(currentTime);
```

This example prints output similar to the following:

```
The file Stemp.java does exist
The time is 11:32:38 right now
```

You can embed a template expression in a string template.

```
String[] a = { "X", "Y", "Z" };
String letters = STR."\{a[0]}, \{STR."\{a[1]}, \{a[2]}"}";
System.out.println(letters);
```

This example prints the following:

```
X, Y, Z
```

To make this example clearer, you can substitute the embedded template expression with a variable:

```
String temp     = STR."\{a[1]}, \{a[2]}";
String letters2 = STR."\{a[0]}, \{temp}";
System.out.println(letters2);
```

Multiline String Templates

When you specify the template for a string template, you can use a text block instead of a regular string. See Text Blocks for more information. This enables you to use HTML documents and JSON data more easily with string templates as demonstrated in the following examples.

```
String title = "My Web Page";
String text = "Hello, world";
String webpage = STR."""
    <html>
      <head>
        <title>\{title}</title>
      </head>
      <body>
        <p>\{text}</p>
      </body>
    </html>
    """;
System.out.println(webpage);
```

This example prints the following:

```
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <p>Hello, world</p>
  </body>
</html>
```

The following example creates JSON data.

```
String customerName    = "Java Duke";
String phone           = "555-123-4567";
String address         = "1 Maple Drive, Anytown";
String json = STR."""
{
    "name":    "\{customerName}",
    "phone":   "\{phone}",
    "address": "\{address}"
}
""";
```

It prints the following:

```
{
    "name":    "Java Duke",
    "phone":   "555-123-4567",
```

```
    "address": "1 Maple Drive, Anytown"
}
```

You can use a text block to work with tabular data more clearly.

```
record Rectangle(String name, double width, double height) {
    double area() {
        return width * height;
    }
}

Rectangle[] zone = new Rectangle[] {
    new Rectangle("First",  17.8, 31.4),
    new Rectangle("Second",  9.6, 12.2),
};

String table = STR."""
    Description\tWidth\tHeight\tArea
    \{zone[0].name}\t\t\{zone[0].width}\t\{zone[0].height}\t\{zone[0].area()}
    \{zone[1].name}\t\t\{zone[1].width}\t\{zone[1].height}\t\{zone[1].area()}
    Total \{zone[0].area() + zone[1].area()}
    """;

System.out.println(table);
```

This example prints the following:

```
Description     Width   Height  Area
First           17.8    31.4    558.92
Second          9.6     12.2    117.11999999999999
Total 676.04
```

The FMT Template Processor

The `FMT` template processor is like the STR template processor except that it can use format specifiers that appear to the left of an embedded expression. These format specifiers are the same as those defined in the class `java.util.Formatter`.

The following example is just like the tabular data example in Multiline String Templates but better formatted.

```
String formattedTable = FormatProcessor.FMT."""
    Description     Width    Height     Area
    %-12s\{zone[0].name}  %7.2f\{zone[0].width}  %7.2f\{zone[0].height}
%7.2f\{zone[0].area()}
    %-12s\{zone[1].name}  %7.2f\{zone[1].width}  %7.2f\{zone[1].height}
%7.2f\{zone[1].area()}
    \{" ".repeat(28)} Total %7.2f\{zone[0].area() + zone[1].area()}
    """;

System.out.println(formattedTable);
```

It prints the following:

```
Description     Width    Height     Area
First           17.80    31.40      558.92
Second           9.60    12.20      117.12
                                Total  676.04
```

The RAW Template Processor

The `RAW` template processor defers the processing of the template to a later time. Consequently, you can retrieve the template's string literals and embedded expression results before processing it.

To retrieve a template's string literals and embedded expression results, call `StringTemplate::fragments` and `StringTemplate::values`, respectively. To process a string template, call `StringTemplate.process(StringTemplate.Processor)` or `StringTemplate.Processor.process(StringTemplate)`. You can also call the method `StringTemplate::interpolate`, which returns the same result as the `STR` template processor.

The following example demonstrates these methods.

```
int v = 10, w = 20;
StringTemplate rawST = StringTemplate.RAW."\{v} plus \{w} equals \{v +
w}";
java.util.List<String> fragments = rawST.fragments();
java.util.List<Object> values = rawST.values();

System.out.println(rawST.toString());

fragments.stream().forEach(f -> System.out.print("[" + f + "]"));
System.out.println();

values.stream().forEach(val -> System.out.print("[" + val + "]"));
System.out.println();

System.out.println(rawST.process(STR));
System.out.println(STR.process(rawST));
System.out.println(rawST.interpolate());
```

It prints the following:

```
StringTemplate{ fragments = [ "", " plus ", " equals ", "" ], values =
[10, 20, 30] }
[][ plus ][ equals ][]
[10][20][30]
10 plus 20 equals 30
10 plus 20 equals 30
10 plus 20 equals 30
```

The method `StringTemplate::fragments` returns a list of *fragment literals*, which are the character sequences preceding each of the embedded expressions, plus the character sequence following the last embedded expression. In this example, the first and last fragment literals are zero-length strings because an embedded expression appears at the beginning and end of the template.

The method `StringTemplate::values` returns a list of embedded expression results. These values are computed every time a string template is evaluated. However, fragment literals are constant across all evaluations of a template expression. The following example demonstrates this.

```
int t = 20;
for (int u = 0; u < 3; u++) {
    StringTemplate stLoop = StringTemplate.RAW."\{t} plus \{u} equals \{t +
u}";
    System.out.println("Fragments: " + stLoop.fragments());
    System.out.println("Values:    " + stLoop.values());
}
```

It prints the following:

```
Fragments: [,  plus ,  equals , ]
Values:    [20, 0, 20]
Fragments: [,  plus ,  equals , ]
Values:    [20, 1, 21]
Fragments: [,  plus ,  equals , ]
Values:    [20, 2, 22]
```

Creating a Template Processor

By implementing the `StringTemplate.Processor` interface, you can create your own template processor, which can return objects of any type, not just `String`, and throw check exceptions if processing fails.

Creating a Template Processor that Returns JSON Objects

The following is an example of a template processor that returns JSON objects.

> **Note:**
>
> These examples use the classes `Json`, `JsonException`, `JsonObject`, and `JsonReader` from the `jakarta.json` package, which contains the Jakarta JSON Processing API. It also indirectly uses the `org.eclipse.parsson.JsonProviderImpl` class from Eclipse Parsson, which is an implementation of Jakarta JSON Processing Specification. Obtain libraries for these APIs from Eclipse GlassFish.

```
var JSON = StringTemplate.Processor.of(
    (StringTemplate stJSON) -> {
```

```
        try (JsonReader jsonReader = Json.createReader(new
StringReader(
            stJSON.interpolate()))) {
            return jsonReader.readObject();
        }
    }
);

String accountType = "user";
String userName = "Duke";
String pw = "my_password";

JsonObject newAccount = JSON."""
{
    "account":    "\{accountType}",
    "user":       "\{userName}",
    "password":   "\{pw}"
}
""";

System.out.println(newAccount);

userName = "Duke\", \"account\": \"administrator";

newAccount = JSON."""
{
    "account":    "\{accountType}",
    "user":       "\{userName}",
    "password":   "\{pw}"
}
""";

System.out.println(newAccount);
```

It prints the following:

```
{"account":"user","user":"Duke","password":"my_password"}
{"account":"administrator","user":"Duke","password":"my_password"}
```

There's a problem with this example: It is susceptible to one type of JSON injection attack. This type of attack involves inserting malicious data containing quotation marks in a JSON string, changing the JSON string itself. In this example, it changes the user name to `"Duke\", \"account\": \"administrator"`. The resulting JSON string becomes the following:

```
{
    "account":    "user",
    "user":       "Duke",
    "account":    "administrator",
    "password":   "my_password"
}
```

If the JSON parser encounters entries with the same name, it takes the last one. Consequently, the user Duke now has administrator privileges.

The following example addresses this kind of JSON injection attack. It throws an exception if the template contains any strings with a quotation mark. It also throws an exception if any of its values aren't numbers, or Boolean values.

```
StringTemplate.Processor<JsonObject, JsonException> JSON_VALIDATE =
    (StringTemplate stJVAL) -> {
    String[] invalidStrings = new String[] { "\"", "'" };
    List<Object> filtered = new ArrayList<>();
    for (Object value : stJVAL.values()) {
        if (value instanceof String str) {
            if (Arrays.stream(invalidStrings).anyMatch(str::contains)) {
                throw new JsonException("Injection vulnerability");
            }
            filtered.add(str);
        } else if (value instanceof Number ||
                   value instanceof Boolean) {
            filtered.add(value);
        } else {
            throw new JsonException("Invalid value type");
        }
    }

    String jsonSource =
        StringTemplate.interpolate(stJVAL.fragments(), filtered);

    try (JsonReader jsonReader = Json.createReader(new StringReader(
        jsonSource))) {
        return jsonReader.readObject();
    }
};

String accountType = "user";
String userName = "Duke";
String pw = "my_password";

try {
    JsonObject newAccount = JSON_VALIDATE."""
    {
        "account":    "\{accountType}",
        "user":       "\{userName}",
        "password":   "\{pw}"
    }
    """;

    System.out.println(newAccount);

    userName = "Duke\", \"account\": \"administrator";

    newAccount = JSON_VALIDATE."""
    {
        "account":    "\{accountType}",
        "user":       "\{userName}",
```

```
        "password":   "\{pw}"
    }
    """;

    System.out.println(newAccount);

} catch (JsonException ex) {
    System.out.println(ex.getMessage());
}
```

It prints the following:

```
{"account":"user","user":"Duke","password":"my_password"}
Injection vulnerability
```

Creating a Template Processor that Safely Composes and Runs Database Queries

You can create a template processor that returns a prepared statement.

Consider the following example that retrieves information about a specific coffee supplier (identified by SUP_NAME) from the table SUPPLIERS. See the subsection SUPPLIERS Table from the JDBC tutorial in *The Java Tutorials (Java SE 8 and earlier)* for more information about this database table, including how to create and populate it.

```
  public static void getSupplierInfo(Connection con, String supName)
throws SQLException {
    String query = "SELECT * from SUPPLIERS s WHERE s.SUP_NAME = '" +
supName + "'";
      try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);
        System.out.println("ID   " +
                           "Name                        " +
                           "Street               " +
                           "City          State  Zip");
        while (rs.next()) {
          int supplierID = rs.getInt("SUP_ID");
          String supplierName = rs.getString("SUP_NAME");
          String street = rs.getString("STREET");
          String city = rs.getString("CITY");
          String state = rs.getString("STATE");
          String zip = rs.getString("ZIP");
          String supRow = FormatProcessor.FMT."%-4s\{supplierID} %-26s\
{
            supplierName} %-20s\{street} %-13s\{city} %-6s\{state} \
{zip}";
          System.out.println(supRow);
        }
      } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

> **Note:**
>
> You can add this method to `SuppliersTable.java`.

Suppose you call this method as follows:

```
SuppliersTable.getSupplierInfoST(myConnection, "Superior Coffee");
```

The method prints the following:

```
ID   Name                      Street              City          State  Zip
49   Superior Coffee           1 Party Place       Mendocino     CA
95460
```

There's a problem with this example: It is susceptible to SQL injection attacks. Suppose you call the method as follows:

```
SuppliersTable.getSupplierInfo(
  myConnection, "Superior Coffee' OR s.SUP_NAME <> 'Superior Coffee");
```

The SQL query becomes the following:

```
SELECT * from SUPPLIERS s WHERE
  s.SUP_NAME = 'Superior Coffee' OR
  s.SUP_NAME <> 'Superior Coffee'
```

The `Statement` object treats the "invalid" supplier name as part of an SQL statement. As a result, the method prints all entries in the SUPPLIERS table, potentially exposing confidential information.

Prepared statements help prevent SQL injection attacks. They always treat client-supplied data as content of a parameter and never as a part of an SQL statement. The following example creates an SQL query string from a string template, creates a JDBC `PreparedStatement` from the query string, and then sets its parameters to the embedded expressions' values.

```
  record QueryBuilder(Connection conn)
    implements StringTemplate.Processor<PreparedStatement, SQLException> {

      public PreparedStatement process(StringTemplate st) throws
SQLException {
        // 1. Replace StringTemplate placeholders with PreparedStatement
placeholders
        String query = String.join("?", st.fragments());

        // 2. Create the PreparedStatement on the connection
        PreparedStatement ps = conn.prepareStatement(query);

        // 3. Set parameters of the PreparedStatement
        int index = 1;
        for (Object value : st.values()) {
```

```
            switch (value) {
                case Integer i -> ps.setInt(index++, i);
                case Float f   -> ps.setFloat(index++, f);
                case Double d  -> ps.setDouble(index++, d);
                case Boolean b -> ps.setBoolean(index++, b);
                default        -> ps.setString(index++,
String.valueOf(value));
            }
        }
        return ps;
    }
  }
```

The following example is like the `getSupplierInfo` example except that it uses the `QueryBuilder` template processor:

```
  public static void getSupplierInfoST(Connection con, String supName)
throws SQLException {
      var DB = new QueryBuilder(con);
      try (PreparedStatement ps = DB."SELECT * from SUPPLIERS s WHERE
s.SUP_NAME = \{supName}") {
        ResultSet rs = ps.executeQuery();
        System.out.println("ID   " +
                           "Name                        " +
                           "Street               " +
                           "City          State  Zip");
        while (rs.next()) {
          int supplierID = rs.getInt("SUP_ID");
          String supplierName = rs.getString("SUP_NAME");
          String street = rs.getString("STREET");
          String city = rs.getString("CITY");
          String state = rs.getString("STATE");
          String zip = rs.getString("ZIP");
          String supRow = FormatProcessor.FMT."%-4s\{supplierID} %-26s\
{
            supplierName} %-20s\{street} %-13s\{city} %-6s\{state} \
{zip}";
          System.out.println(supRow);
        }
      } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

Creating a Template Processor that Simplifies the Use of Resource Bundles

The following template processor, `LocalizationProcessor`, maps a string to a corresponding property in a resource bundle. When using this template processor, in your resource bundles, the property names are the string templates in your

applications, where embedded expressions are substituted with underscores (_) and spaces with periods (.).

```
record LocalizationProcessor(Locale locale)
    implements StringTemplate.Processor<String, RuntimeException> {

    public String process(StringTemplate st) {
        ResourceBundle resource = ResourceBundle.getBundle("resources",
locale);
        String stencil = String.join("_", st.fragments());
        String msgFormat = resource.getString(stencil.replace(' ', '.'));
        return MessageFormat.format(msgFormat, st.values().toArray());
    }
}
```

Suppose the following are your resource bundles:

Figure 3-1 resources_en_US.properties

```
# resources_en_US.properties file
_.chose.option._=\
    {0} chose option {1}
```

Figure 3-2 resources_fr_CA.properties

```
# resources_fr_CA.properties file
_.chose.option._=\
    {0} a choisi l''option {1}
```

The following example uses `LocalizationProcessor` and these two resource bundles:

```
var userLocale = new Locale("en", "US");
var LOCALIZE = new LocalizationProcessor(userLocale);

String user = "Duke", option = "b";
System.out.println(LOCALIZE."\{user} chose option \{option}");

userLocale = new Locale("fr", "CA");
LOCALIZE = new LocalizationProcessor(userLocale);
System.out.println(LOCALIZE."\{user} chose option \{option}");
```

It prints the following:

```
Duke chose option b
Duke a choisi l'option b
```

4

Unnamed Classes and Instance Main Methods

The addition of *instance main methods* and *unnamed classes* to the Java language enables 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.

Two preview features, instance main methods and unnamed classes, are added to the Java language. This is an evolutionary step in the language that enables students to begin writing small programs without needing to understand the full set of language features designed for large programs. Far from being a separate dialect, students can now use the Java language to write streamlined declarations for single-class programs and later seamlessly expand their beginning programs to use more advanced features as their skills grow. Java veterans might also find *instance main methods* and *unnamed classes* features useful when writing simple Java programs that do not require programming-in-the-large scaffolding of the Java language.

> **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.
>
> For background information about *instance main methods* and *unnamed classes*, see JEP 445.

The Java language enables development teams to create, develop, and maintain large and complex applications over many years. It is a multiparadigm language with 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. With these features, components can expose well-defined interfaces for interaction with other components and also hide internal implementation details to permit the independent evolution of each. The object-oriented paradigm itself is designed for plugging together pieces that interact through well-defined protocols and also to abstract away implementation details. This composition of large components is called "programming in the large."

The Java language also offers many constructs useful for "programming in the small" in which everything is internal to a component. In recent years, it has enhanced its programming-in-the-large capabilities with modules and its programming-in-the-small capabilities with data-oriented programming.

The Java language is also intended to be a first programming language. When programmers first start out they do not write large programs, as part of a development team. They write small programs, alone. They have no need for encapsulation and namespaces which are useful to separately evolve components written by different people. When teaching

programming, instructors start with the basic programming-in-the-small concepts of variables, control flow, and subroutines. At that stage there is no need for the programming-in-the-large concepts of classes, packages, and modules.

Consider the classic `Hello, World!` 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 never used.
- The `static` modifier is part of Java's class-and-object model. For the novice, `static` is not just mysterious but harmful: To add more methods or fields that `main` can call and use, the student must either declare them all as `static` (thereby propagating an idiom which is neither common nor a good habit) or else confront the difference between static and instance members and learn how to instantiate an object.

Novice programmers encounter these programming-in-the-large constructs before they have learned 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 their students. It leaves students with the enduring impression that Java is overly complicated.

The preview language features, instance main methods and unnamed classes, reduce the complexity of writing simple programs such as `Hello, World!` by enabling programmers to write programs without using access modifiers, `static` modifiers, or the `String[]` parameter. The introduction of programming-in-the-large constructs can be postponed by instructors until they are needed.

Flexible Launch Protocol

New programmers want to write and run computer programs, but the current Java Language Specification focuses students on defining the core Java unit of the class and the basic compilation unit, namely a source file comprised of a package declaration, followed by import declarations and one or more class declarations.

The actions of choosing the class containing the `main` method, assembling its dependencies in the form of a module path or a class path (or both), loading the class, initializing it, and invoking the `main` method with its arguments constitute the launch protocol. In the JDK, the launch protocol is implemented by the launcher as the `java` executable.

By allowing *instance* `main` methods, the new preview language features enhance the Java launch protocol and provide greater flexibility in the declaration of a program's entry point as follows:

- Allows the `main` method of a launched class to have `public`, `protected`, or default (such as package) access.
- If a launched class does not contain a `static main` method with a `String[]` parameter, but does contain a `static main` method without parameters, the launch protocol invokes that method.
- If a launched class does not contain a `static main` method, but has a non-`private` zero-parameter constructor (such as, `public`, `protected`, or package access) and a non-`private` *instance* `main` method, the launch protocol constructs an instance of the class. If the class has an *instance* `main` method with a `String[]` parameter, the launch protocol invokes that method; otherwise, it invokes the *instance* `main` method with no parameters.

By using *instance* `main` methods, we can simplify the `Hello, World!` program presented in **Unnamed Classes and Instance Main Methods** to:

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

Selecting a main Method

This is a change of behavior when launching a class.

The launch protocol chooses to invoke the first of the following methods:

- A `static void main(String[] args)` method of non-private access (such as, `public`, `protected` or package) declared in the launched class
- A `static void main()` method of non-private access declared in the launched class
- A `void main(String[] args)` instance method of non-private access declared in the launched class or inherited from a superclass
- A `void main()` instance method of non-private access declared in the launched class or inherited from a superclass

> **Note:**
>
> If the launched class declares an *instance* `main` method, that method will be invoked rather than an inherited "traditional" `public static void main(String[] args)` declared in a superclass. Therefore, if the launched class inherits a "traditional" `main` method but another method (such as, an *instance* `main`) is selected, the JVM will issue a warning to the standard error at runtime.

If the selected `main` is an instance method and is a member of an inner class, the program will fail to launch.

Unnamed Classes

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 do not need them can omit them.

A program that does not 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 does not 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 does not need it — just as we do not require explicit package or module declarations for code that does not need them.

When the Java compiler encounters a source file containing a method that is not enclosed in a class declaration, it implicitly considers such methods (as well as any unenclosed fields and any classes declared in the file) to be members of an unnamed top-level class.

An *unnamed class* is always a member of the unnamed package. It is also `final` and cannot implement any interface or extend any class other than `Object`. An *unnamed class* cannot 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.

No code can refer to an *unnamed class* by name, so instances of an *unnamed class* cannot be constructed directly. Such a class is useful only as a standalone program or as an entry point to a program. Therefore, an *unnamed class* must have a `main` method that can be launched as described above. This requirement is enforced by the Java compiler.

An *unnamed 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 *unnamed classes* in the unnamed module. Every *unnamed class* contains a `main` method and represents a program. Multiple such classes in the unnamed package represent multiple programs.

An *unnamed class* is almost exactly like 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). The class can have static initializers as well as instance initializers. One key difference is that while

an *unnamed class* has a default zero-parameter constructor, it can have no other constructor.

With these changes, we can now write `Hello, World!` as:

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

Because top-level members are interpreted as members of the *unnamed 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);
}
```

A source file named `HelloWorld.java` that contains an *unnamed class* can be launched with the source-code launcher 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 cannot be used directly in Java source code.

At this time, the `javadoc` tool will fail when asked to generate API documentation for a Java file with an *unnamed class* because *unnamed classes* do not define an API that is accessible from other classes. This behavior may change in a future release.

The `Class.isSynthetic` method returns `true` for an *unnamed class*.

Growing a Program

A `Hello, World!` program written as an *unnamed class* is more focused on what the program actually does by omitting concepts and constructs it does not need. Even so, all members are interpreted just as they are in an ordinary class.

Concepts and constructs can be added to an *unnamed class* as as needed by the program. An *unnamed class* can easily be evolved later into an ordinary class, by wrapping its declaration (excluding import statements) inside an explicit class declaration.

Eliminating the `main` method altogether might seem like a natural next step, but it would work against the goal of gracefully evolving a student's first Java program into a larger one as well

as impose some non-obvious restrictions (see see the section Alternatives in JEP 445). Dropping the `void` modifier would similarly create a distinct Java dialect.

5
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 5-1 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 5-2 Circle.java

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

Figure 5-3 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 5-4 Rectangle.java

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

`Rectangle` has a further subclass, `FilledRectangle`:

Figure 5-5 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

6
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.

Topics

Pattern Matching for the instanceof Operator

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.

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");
        }
    }
```

> **Note:**
>
> Removing this conversion step also makes your code safer. Testing an object's type with the `instanceof`, 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.

A *pattern* is a combination of a test, which is called a *predicate*; a *target*; and a set of local variables, which are called *pattern variables*. The `getPerimeter` example contains two patterns, `s instanceof Rectangle r` and `s instanceof Circle c`:

- The predicate is a Boolean-valued function with one argument. In these two patterns, it's the `instanceof` operator, testing if the `s` argument is a `Rectangle` or a `Circle`.
- The target is the argument of the predicate. In these two patterns, it's the `s` value.

- The pattern variables are those that store data from the target only if the predicate returns `true`. In these two patterns, they're the variables `r` and `c`.

A *type pattern* consists of a predicate that specifies a type, along with a single pattern variable. In this example, the type patterns are `Rectangle r` and `Circle c`.

Scope of Pattern Variables

The scope of a pattern variable are 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 Variable Declarations for more examples of where you can use a pattern variable.

Pattern Matching for switch Expressions and Statements

A `switch` statement transfers control to one of several statements or expressions, depending on the value of its selector expression. In earlier releases, the selector expression must evaluate to a number, string or `enum` constant, and case labels must be constants. However, in this release, the selector expression can be any reference type or an `int` type but not a `long`, `float`, `double`, or `boolean` type, and `case` labels can have patterns. Consequently, a `switch` statement or expression 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 for the instanceof Operator:

```
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;
            default:          throw new
IllegalArgumentException("Unrecognized shape");
        }
    }
```

Topics

- Selector Expression Type
- When Clauses
- Qualified enum Constants as case Constants
- Pattern Label Dominance
- Type Coverage in switch Expressions and Statements
- Inference of Type Arguments in Record Patterns
- Scope of Pattern Variable Declarations
- Null case Labels

Selector Expression Type

The type of a selector expression can either be an integral primitive type or any reference type, such as in the previous examples. The following `switch` expression matches the selector expression `obj` with type patterns that involve a class type, an enum type, a record type, and an 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");
        }
    }
```

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 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 patten label matches if `obj` is a `String` of a different length.

A guarded patten 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*.

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\u0153ur");
            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\u0153ur");
            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.

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());
            case String s -> // error: this case label is dominated by
a preceding case label
                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);
            case -1, 1 -> // Compile-time errors for both cases -1 and
1:
                          // this case label is dominated by a
preceding case label
                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);
            case RED -> // error: this case label is dominated by a
preceding case label
                System.out.println("The color red");
        }
    }
```

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");
        }
    }
```

Type Coverage in switch Expressions and Statements

As described in Switch Expressions, the `switch` blocks of `switch` expressions and `switch` statements, which use pattern or `null` labels, must be exhaustive. This means that for all possible values, there must be a matching `switch` label. The following `switch` expression is not exhaustive and generates a compile-time error. Its type coverage consists of the subtypes of `String` or `Integer`, which doesn't include the type of the selector expression, `Object`:

```
static int coverage(Object obj) {
    return switch (obj) {          // Error - not exhaustive
        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;
    };
}
```

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 two classes `ME` and `Seal`:

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


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

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

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

When you run `ME`, it throws a `MatchException`:

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

Inference of Type Arguments in Record Patterns

The compiler can infer the type arguments for a generic record pattern in all constructs that accept patterns: `switch` statements, `instanceof` expressions, and enhanced `for` statements.

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);
        }
    }
```

See Record Patterns for more examples of inference of type arguments in record patterns.

Scope of Pattern Variable Declarations

As described in the section Pattern Matching for the instanceof Operator, 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.
        }
    }
```

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.

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.

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())));
        }
    }
```

Generic Records

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);
        }
    }
```

Type Inference

You can use `var` in the record pattern's component list. 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 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.

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);
        }
    }
```

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);
        }
    }
```

Unnamed Patterns and Variables

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. *Unnamed variables* are variables that can be initialized but not used. You denote both with the underscore character (_).

> **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.
> For background information about unnamed patterns and variables, see JEP 443.

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 omit or *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");
        }
```

Unnamed Variables

You can use the underscore keyword (_) not just as a pattern in a pattern list, but also 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'ss 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 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 6-1 Valid Unnamed Variable Declarations

Declaration Type	Example with Unnamed Variable

A local variable declaration statement in a block	`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;`
	`}`
	
	Note that you don't have to assign the value returned by `Queue::remove` 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.

Table 6-1 (Cont.) Valid Unnamed Variable Declarations

Declaration Type	Example with Unnamed Variable

A resource specification of a `try`-with-resources statement	`static void doesFileExist(String path) {`
	`  try (var _ = new FileReader(path)) {`
	`  // Do nothing`
	`  } catch (IOException e) {`
	`  e.printStackTrace();`
	`  }`
	`}`

The header of a basic `for` statement	`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);`
	`}`

The header of an enhanced `for` loop	`static void stringLength(String s) {`
	`  int len = 0;`
	`  for (char _ : s.toCharArray()) {`
	`  len++;`
	`  }`
	`  System.out.println("Length of " + s`
	`+ ": " + len);`
	`}`

An exception parameter of a `catch` block	`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");`
	`  }`
	`}`

Table 6-1 (Cont.) Valid Unnamed Variable Declarations

Declaration Type	Example with Unnamed Variable

A formal parameter of a lambda expression	`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"));`
	`}`

7
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
```

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.

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());
```

```
    }
}
```

Record Serialization

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. See Serializable Records for more information and an extended example. See also the section Serialization of Records in the *Java Object Serialization Specification*.

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 though 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.

8
Switch Expressions

Like all expressions, `switch` expressions evaluate to a single value and can be used in statements. They 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.

"case L ->" Labels

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;
        default:
            throw new IllegalStateException("Invalid day: " + day);
    }
    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 a new kind of `case` label. The following is a `switch`

expression that uses the new kind of `case` label 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;
            default -> throw new IllegalStateException("Invalid day: "
+ day);
        }
    );
```

The new kind of `case` label 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 the new kind of `case` label in `switch` statements. The following is like the first example, except it uses "`case L ->`" labels instead of "`case L:`" labels:

```
    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;
        default -> throw new IllegalStateException("Invalid day: " +
day);
    };
    System.out.println(numLetters);
```

A "`case L ->`" label along with its code to its right is called a switch labeled rule.

"case L:" Statements and the yield Statement

You can use "`case L:`" labels in `switch` expressions; a "`case L:`" label 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;
        default:
            throw new IllegalStateException("Invalid day: " + day);
    };
    System.out.println(numLetters);
```

The previous example uses `yield` statements. They take one argument, which is the value that the `case` label 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.

Note:

It's recommended that you use "`case L ->`" labels. It's easy to forget to insert `break` or `yield` statements when using "`case L:`" labels; if you do, you might introduce unintentional fall through in your code.

For "`case L ->`" labels, to specify multiple statements or code that are not expressions or `throw` statements, enclose them in a block. Specify the value that the `case` label 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;
        }
        default -> {
            throw new IllegalStateException("Invalid day: " +
day);
        }
    };
```

Exhaustiveness of switch Expressions

The cases of a `switch` expression must be *exhaustive*, which means that for all possible values, there must be a matching switch label. Thus, a `switch` expression normally requires a `default` clause. However, for an `enum switch` expression that covers all known constants, the compiler inserts an implicit `default` clause.

In addition, a `switch` expression must either complete normally with a value or complete abruptly by throwing an exception. 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
        };
    // ...
    }
```

Exhaustiveness of switch Statements

The cases of a `switch` statement must be exhaustive if it uses pattern or `null` labels, or if its selector expression isn't one of the legacy types (`char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String`, or an `enum` type).

The following example doesn't compile because the `switch` statement (which uses pattern labels) is not exhaustive:

```
    static void testSwitchStatementExhaustive(Object obj) {
        switch (obj) {  // error: the switch statement does not cover
                        //        all possible input values
            case String s:
                System.out.println(s);
                break;
            case Integer i:
                System.out.println("Integer");
                break;
        }
    }
```

You can make it exhaustive by adding a `default` clause:

```
static void testSwitchStatementExhaustive(Object obj) {
    switch (obj) {
        case String s:
            System.out.println(s);
            break;
        case Integer i:
            System.out.println("Integer");
            break;
        default:
            break;
    }
}
```

9
Text Blocks

See Programmer's Guide to Text Blocks for more information about this language feature. For background information about text blocks, see JEP 378.

10

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.