Java Platform, Standard Edition

Java Language Updates

Release 15

F31046-01

September 2020

ORACLE

Java Platform, Standard Edition Java Language Updates, Release 15

F31046-01

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.

Related Documents

See JDK 15 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 Updates for Java SE 15

Feature	Description	JEP
Sealed Classes	Introduced as a preview feature for this release. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.	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. Records are classes that act as transparent carriers 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 needing to enable 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)

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

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

Note:

This is a preview feature, which is a feature whose design, specification, and implementation are complete, but is not permanent, which means that the 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 Features.

For background information about sealed classes and interfaces, see JEP 360.

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.

Defining 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 3-1 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 3-2 Circle.java

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

Figure 3-3 Square.java

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

Figure 3-4 Rectangle.java

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

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

Figure 3-5 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 { }
  ```

Defining 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(); }
}
```

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

4

Pattern Matching for instanceof

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.

More specifically, JDK 15 extends the `instanceof` operator: you can specify a *binding variable*; if the result of the `instanceof` operator is `true`, then the object being tested is assigned to the binding variable.

Note:

This is a preview feature, which is a feature whose design, specification, and implementation are complete, but is not permanent, which means that the 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 Features.

For background information about pattern matching for the `instaceof` operator, see JEP 375.

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

```
public interface Shape { }

final 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; }
}

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

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 test pattern, making your code shorter and easier to read:

```
    public static double getPerimeter(Shape shape) throws
IllegalArgumentException {
        if (shape instanceof Rectangle s) {
            return 2 * s.length() + 2 * s.width();
        } else if (shape instanceof Circle s) {
            return 2 * s.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 predicate that can be applied to a target and a set of binding variables that are extracted from the target only if the predicate successfully matches it. The *predicate* is a Boolean-valued function of one argument; in this case, it's the `instanceof` operator testing whether the `Shape` argument is a `Rectangle` or a `Circle`. The *target* is the argument of the predicate, which is the `Shape` argument. The *binding variables* are those that store data from the target only if the predicate returns `true`, which is the variable `s`.

A *type test pattern* consists of a predicate that specifies a type, along with a single binding variable. In this example, the type test pattens are `Rectangle s` and `Circle s`.

Scope of Binding Variables

The scope of a binding 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 binding variable s (of type Rectangle)
here.
        } else if (shape instanceof Circle s) {
            // You can use the binding variable s of type Circle here
            // but not the binding variable s of type Rectangle.
        } else {
            // You cannot use either binding variable here.
        }
    }
```

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

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

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

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

Because the conditional-AND operator (`&&`) is short-circuiting, the program can reach the `s.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 s || s.length() > 0) { // error
            // ...
        }
```

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

5
Record Classes

Introduced as a preview feature in Java SE 14, record classes help to model plain data aggregates with less ceremony than normal classes. Java SE 15 extends the preview feature with additional capabilities such as local record classes.

Note:

This is a preview feature, which is a feature whose design, specification, and implementation are complete, but is not permanent, which means that the 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 Features.

For background information about record classes, see JEP 384.

A record class declares a sequence of fields, and then the appropriate accessors, constructors, `equals`, `hashCode`, and `toString` methods are created automatically. The 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, a header (which lists the fields of the class, known as its "components"), 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.
  - 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 `private` 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.

You create a record object (an instance of a record class) with the `new` keyword, for example:

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

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:

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

  If an annotation is applied to the record class's components, then it also applies to the numerous members in the record class that are declared automatically. In this example, the `@GreaterThanZero` annotation applies not only to the `length` and `width` components, but also to:

  – The parameters of the canonical constructor

  – The record class's components

  – The private fields `length` and `width`

  – The accessor methods `length()` and `width()`

  You can specify the members to which an annotation applies with the `@Target` meta-annotation. For example, if `@GreaterThanZero` is applicable to only field declarations:

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

  Then the previous 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 you can define 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.

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 in Java SE 15:

```
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 new 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.

6 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

Unlike `switch` statements, the cases of `switch` expressions must be *exhaustive*, which means that for all possible values, there must be a matching switch label. Thus, `switch` expressions normally require a `default` clause. However, for `enum switch` expressions that cover 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
        };
    // ...
    }
```

7 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.

8

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. Style Guidelines for Local Variable Type Inference in Java 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.