Java Platform, Standard Edition

Java Language Updates

Release 14

F23118-01

March 2020

Java Platform, Standard Edition Java Language Updates, Release 14

F23118-01

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 14 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 14

Java SE 14 introduces pattern matching for the `instanceof` operator; if the result of the `instanceof` operator is `true`, then the object being tested is automatically assigned to a variable that you previously declared. See Pattern Matching for the instanceof Operator. This release also introduces records, which are a new kind of type declaration that's ideal for "plain data carriers," classes that contain data not meant to be altered and only the most fundamental methods such as constructors and accessors. See Records.

Text blocks accept two more escape sequences (see Programmer's Guide to Text Blocks), and Switch Expressions is now a permanent language feature.

Java Language Changes for Java SE 13

Java SE 13 introduces text blocks, which are multiline string literals that don't require common escape sequences; see Programmer's Guide to Text Blocks. It also introduces one change to `switch` expressions: To specify their value, use the new `yield` statement instead of the `break` statement; see Switch Expressions in *Java Platform, Standard Edition Java Language Updates, Release 13*.

Java Language Changes for Java SE 12

Java SE 12 introduces `switch` expressions, plus a new kind of `case` label that prevents fall through. This is available as a preview feature. See Switch Expressions in *Java Platform, Standard Edition Java Language Updates, Release 12*.

Java Language Changes for Java SE 11

Java SE 11 lets you declare formal parameters of implicitly typed lambda expressions with the `var` identifier; see Local Variable Type Inference.

Java Language Changes for Java SE 10

Java SE 10 introduces support for inferring the type of local variables from the context, which makes code more readable and reduces the amount of required boilerplate code.

Java Language Changes for Java SE 9

The major change to Java Platform, Standard Edition (Java SE) 9 is the introduction of the Java Platform module system.

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, i.e., 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.

To learn more about the Java Platform module system, see Project Jigsaw on OpenJDK.

Apart from the new module system, a few changes have been made to the Java language; see More Concise try-with-resources Statements and Small Language Changes in Java SE 9.

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

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.

More specifically, JDK 14 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 JDK 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 305.

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.

4 Records

JDK 14 introduces records, which are a new kind of type declaration. Like an `enum`, a record is a restricted form of a class. It’s ideal for "plain data carriers," classes that contain data not meant to be altered and only the most fundamental methods such as constructors and accessors.

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 JDK releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Features.

For background information about records, see JEP 359.

Consider the following class definition:

```
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; }
}
```

It has the following characteristics:

- All of its members are declared final
- Its only methods consist of a constructor, `Rectangle(double length, double width)` and two accessors, `length()` and `width()`

You can represent this class with a record:

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

A record consists of a name (in this example, it's `Rectangle`) and a list of the record's components (which in this example are `float length` and `float width`).

A record acquires these members automatically:

- A private `final` field for each of its components
- A public read accessor method for each component with the same name and type of the component; in this example, these methods are `Rectangle::length()` and `Rectangle::width()`
- A public constructor whose signature is derived from the record components list. The constructor initializes each private field from the corresponding argument.
- Implementations of the `equals()` and `hashCode()` methods, which specify that two records are equal if they are of the same type and their corresponding record components are equal
- An implementation of the `toString()` method that includes the string representation of all the record's components, with their names

Compact Constructors

If you want your record's constructor to do more than initialize its private fields, you can define a custom constructor for the record. However, unlike a class constructor, a record constructor doesn't have a formal parameter list; this is called a *compact constructor*.

For example, the following record, `HelloWorld`, has one field, `message`. Its custom constructor calls `Objects.requireNonNull(message)`, which specifies that if the `message` field is initialized with a null value, then a `NullPointerException` is thrown. (Custom record constructors still initialize their record's private fields.)

```
record HelloWorld(String message) {
    public HelloWorld {
        java.util.Objects.requireNonNull(message);
    }
}
```

Restrictions on Records

The following are restrictions on the use of records:

- Records cannot extend any class
- Records cannot declare instance fields (other than the private `final` fields that correspond to the components of the record component list); any other declared fields must be `static`
- Records cannot be abstract; they are implicitly `final`
- The components of a record are implicitly final

Beyond these restrictions, records behave like regular classes:

- You can declare them inside a class; nested records are implicitly static
- You can create generic records
- Records can implement interfaces
- You instantiate records with the `new` keyword
- You can declare in a record's body static methods, static fields, static initializers, constructors, instance methods, and nested types
- You can annotate records and a record's individual components

APIs Related to Records

The class `java.lang.Class` has two new methods related to records:

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

5 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
        };
    // ...
    }
```

6

Text Blocks

See Programmer's Guide to Text Blocks for more information about this language feature.

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 JDK releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Features.

For background information about text blocks, see JEP 368.

7 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.

8 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).

9 Small Language Changes in Java SE 9

There are several small language changes in Java SE 9.

@SafeVarargs annotation is allowed on private instance methods.

The @SafeVarargs annotation 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.

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.

In Java SE 9, as long as the inferred type is denotable, you can use the diamond operator when you create an anonymous inner class.

The underscore character is not a legal name.

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

Private interface methods are supported.

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