Java Platform, Standard Edition

Java Language Updates

Release 11

E94884-01

September 2018

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.

Local-Variable Type Inference

In Java SE 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 Java SE 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. Note that 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
  ```

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. The rest of this guide describes those changes.

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).

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.

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.

Java Platform, Standard Edition Java Language Updates, Release 11
E94884-01

Describes the updated language features in Java SE 9 and subsequent releases.