Java Platform, Standard Edition

Java Language Updates

Release 11
E94884-05
January 2025

ORACLE

Java Platform, Standard Edition Java Language Updates, Release 11

E94884-05

Contents

Preface

This guide describes updates to the language in Java SE 9.

Audience

This document is for Java developers.

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

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

Related Documents

See JDK 11 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.

Convention	Meaning
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 Summary

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

New Language Features

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

Table 1-1 Java Language Changes Summary

Release	New Language Features
9	Java Platform Module System (JSR 376): see Project Jigsaw on OpenJDK Milling Project Coin (JEP 213): seeJava Language Changes for Java SE 9 Small Enhancements to the Java Programming Language (JSR 334): see Java Language Changes for Java SE 9
10	Local Variable Type Inference (JEP 286)
11	Local-Variable Syntax for Lambda Parameters (JEP 323): see Local Variable Type Inference

Evolution of Language Features

The following tables list the changes made to the Java language since Java SE 9. The first column specifies the feature while the subsequent columns specify a Java SE release. An icon specifies when the feature was made permanent.

Table 1-2 Java Language Changes from Java SE 11 to Java SE 9

Feature	11	10	9
Local-Variable Syntax for Lambda Parameters: see Local Variable Type Inference	P JEP 323		
Local Variable Type Inference		P JEP 286	
Java Platform Module System: see Project Jigsaw on OpenJDK			P JSR 376
Milling Project Coin: see Java Language Changes for Java SE 9			P JEP 213
Small Enhancements to the Java Programming Language: see Java Language Changes for Java SE 9			P JSR 334

2

Java Language Changes

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

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.

3
Local Variable Type Inference

In JDK 10 and later, you can declare local variables with non-null initializers with the `var` identifier, which can help you write code that's easier to read.

Consider the following example, which seems redundant and is hard to read:

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

You can rewrite this example by declaring the local variables with the `var` identifier. The type of the variables are inferred from the context:

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

`var` is a reserved type name, not a keyword, which means that existing code that uses `var` as a variable, method, or package name is not affected. However, code that uses `var` as a class or interface name is affected and the class or interface needs to be renamed.

`var` can be used for the following types of variables:

- Local variable declarations with initializers:

  ```
  var list = new ArrayList<String>();    // infers ArrayList<String>
  var stream = list.stream();            // infers Stream<String>
  var path = Paths.get(fileName);        // infers Path
  var bytes = Files.readAllBytes(path);  // infers bytes[]
  ```

- Enhanced `for`-loop indexes:

  ```
  List<String> myList = Arrays.asList("a", "b", "c");
  for (var element : myList) {...}  // infers String
  ```

- Index variables declared in traditional `for` loops:

  ```
  for (var counter = 0; counter < 10; counter++)  {...}   // infers int
  ```

- `try`-with-resources variable:

  ```
  try (var input =
       new FileInputStream("validation.txt")) {...}   // infers
  FileInputStream
  ```

- Formal parameter declarations of implicitly typed lambda expressions: A lambda expression whose formal parameters have inferred types is *implicitly typed*:

  ```
  BiFunction<Integer, Integer, Integer> = (a, b) -> a + b;
  ```

  In JDK 11 and later, you can declare each formal parameter of an implicitly typed lambda expression with the `var` identifier:

  ```
  (var a, var b) -> a + b;
  ```

  As a result, the syntax of a formal parameter declaration in an implicitly typed lambda expression is consistent with the syntax of a local variable declaration; applying the `var` identifier to each formal parameter in an implicitly typed lambda expression has the same effect as not using `var` at all.

  You cannot mix inferred formal parameters and `var`-declared formal parameters in implicitly typed lambda expressions nor can you mix `var`-declared formal parameters and manifest types in explicitly typed lambda expressions. The following examples are not permitted:

  ```
  (var x, y) -> x.process(y)      // Cannot mix var and inferred formal
  parameters
                                  // in implicitly typed lambda expressions
  (var x, int y) -> x.process(y)  // Cannot mix var and manifest types
  // in explicitly typed lambda expressions
  ```

Local Variable Type Inference Style Guidelines

Local variable declarations can make code more readable by eliminating redundant information. However, it can also make code less readable by omitting useful information. Consequently, use this feature with judgment; no strict rule exists about when it should and shouldn't be used.

Local variable declarations don't exist in isolation; the surrounding code can affect or even overwhelm the effects of `var` declarations. Local Variable Type Inference: Style Guidelines examines the impact that surrounding code has on `var` declarations, explains tradeoffs between explicit and implicit type declarations, and provides guidelines for the effective use of `var` declarations.