ADVANCED JAVA LANGUAGE FEATURES

Get started with lambda expressions in Java

Learn how to use lambda expressions and functional programming techniques in your Java programs

Before Java SE 8, anonymous classes were typically used to pass functionality to a method. This practice obfuscated source code, making it harder to understand. Java 8 eliminated this problem by introducing lambdas. This tutorial first introduces the lambda language feature, then provides a more detailed introduction to functional programming with lambda expressions along with target types. You'll also learn how lambdas interact with scopes, local variables, the `this` and `super` keywords, and Java exceptions.

Note that code examples in this tutorial are compatible with JDK 12.

Discovering types for yourself

I won't introduce any non-lambda language features in this tutorial that you haven't previously learned about, but I will demonstrate lambdas via types that I haven't previously discussed in this series. One example is the `java.lang.Math` class. I will introduce these types in future Java 101 tutorials. For now, I suggest reading the JDK 12 API documentation to learn more about them.

Get the code

Download the source code for example applications in this tutorial. *Created by Jeff Friesen for JavaWorld.*

Lambdas: A primer

A *lambda expression (lambda)* describes a block of code (an anonymous function) that can be passed to constructors or methods for subsequent execution. The constructor or method receives the lambda as an argument. Consider the following example:

```
() -> System.out.println("Hello")
```

This example identifies a lambda for outputting a message to the standard output stream. From left to right, `()` identifies the lambda's formal parameter list (there are no parameters in the example), `->` indicates that the expression is a lambda, and `System.out.println("Hello")` is the code to be executed.

Lambdas simplify the use of *functional interfaces*, which are annotated interfaces that each declare exactly one abstract method (although they can also declare any combination of default, static, and private methods). For example, the standard class library provides a `java.lang.Runnable` interface with a single abstract `void run()` method. This functional interface's declaration appears below:

```
@FunctionalInterface
public interface Runnable
{
   public abstract void run();
}
```

The class library annotates `Runnable` with `@FunctionalInterface`, which is an instance of the `java.lang.FunctionalInterface` annotation type. `FunctionalInterface` is used to annotate those interfaces that are to be used in lambda contexts.

A lambda doesn't have an explicit interface type. Instead, the compiler uses the surrounding context to infer which functional interface to instantiate when a lambda is specified--the lambda is *bound* to that interface. For example, suppose I specified the following code fragment, which passes the previous lambda as an argument to the `java.lang.Thread` class's `Thread(Runnable target)` constructor:

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

```
new Thread(() -> System.out.println("Hello"));
```

The compiler determines that the lambda is being passed to `Thread(Runnable r)` because this is the only constructor that satisfies the lambda: `Runnable` is a functional interface, the lambda's empty formal parameter list `()` matches `run()`'s empty parameter list, and the return types (`void`) also agree. The lambda is bound to `Runnable`.

Listing 1 presents the source code to a small application that lets you play with this example.

Listing 1. LambdaDemo.java (version 1)

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      new Thread(() -> System.out.println("Hello")).start();
   }
}
```

Compile Listing 1 (`javac LambdaDemo.java`) and run the application (`java LambdaDemo`). You should observe the following output:

```
Hello
```

Lambdas can greatly simplify the amount of source code that you must write, and can also make source code much easier to understand. For example, without lambdas, you would probably specify Listing 2's more verbose code, which is based on an instance of an anonymous class that implements `Runnable`.

Listing 2. LambdaDemo.java (version 2)

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         System.out.println("Hello");
                      }
                   };
      new Thread(r).start();
   }
}
```

After compiling this source code, run the application. You'll discover the same output as previously shown.

Lambdas and the Streams API

As well as simplifying source code, lambdas play an important role in Java's functionally-oriented Streams API. They describe units of functionality that are passed to various API methods.

Java lambdas in depth

To use lambdas effectively, you must understand the syntax of lambda expressions along with the notion of a target type. You also need to understand how lambdas interact with scopes, local variables, the `this` and `super` keywords, and exceptions. I'll cover all of these topics in the sections that follow.

How lambdas are implemented

Lambdas are implemented in terms of the Java virtual machine's `invokedynamic` instruction and the `java.lang.invoke` API. Watch the video Lambda: A Peek Under the Hood to learn about lambda architecture.

Lambda syntax

Every lambda conforms to the following syntax:

```
( formal-parameter-list ) -> { expression-or-statements }
```

The *`formal-parameter-list`* is a comma-separated list of formal parameters, which must match the parameters of a functional interface's single abstract method at runtime. If you omit their types, the compiler infers these types from the context in which the lambda is used. Consider the following examples:

```
(double a, double b) // types explicitly specified
(a, b) // types inferred by compiler
```

Lambdas and var

Starting with Java SE 11, you can replace a type name with `var`. For example, you could specify `(var a, var b)`.

You must specify parentheses for multiple or no formal parameters. However, you can omit the parentheses (although you don't have to) when specifying a single formal parameter. (This applies to the parameter name only--parentheses are required when the type is also specified.) Consider the following additional examples:

```
x // parentheses omitted due to single formal parameter
(double x) // parentheses required because type is also present
() // parentheses required when no formal parameters
(x, y) // parentheses required because of multiple formal parameters
```

The *`formal-parameter-list`* is followed by a `->` token, which is followed by *`expression-or-statements`*--an expression or a block of statements (either is known as the lambda's body). Unlike expression-based bodies, statement-based bodies must be placed between open ({) and close (}) brace characters:

```
(double radius) -> Math.PI * radius * radius
radius -> { return Math.PI * radius * radius; }
radius -> { System.out.println(radius); return Math.PI * radius * radius; }
```

The first example's expression-based lambda body doesn't have to be placed between braces. The second example converts the expression-based body to a statement-based body, in which `return` must be specified to return the expression's value. The final example demonstrates

multiple statements and cannot be expressed without the braces.

Lambda bodies and semicolons

Note the absence or presence of semicolons (**;**) in the previous examples. In each case, the lambda body isn't terminated with a semicolon because the lambda isn't a statement. However, within a statement-based lambda body, each statement must be terminated with a semicolon.

Listing 3 presents a simple application that demonstrates lambda syntax; note that this listing builds on the previous two code examples.

Listing 3. LambdaDemo.java (version 3)

```
@FunctionalInterface
interface BinaryCalculator
{
   double calculate(double value1, double value2);
}
@FunctionalInterface
interface UnaryCalculator
{
   double calculate(double value);
}
public class LambdaDemo
{
   public static void main(String[] args)
   {
      System.out.printf("18 + 36.5 = %f%n", calculate((double v1, double v2) ->
                        v1 + v2, 18, 36.5));
      System.out.printf("89 / 2.9 = %f%n", calculate((v1, v2) -> v1 / v2, 89,
                        2.9));
      System.out.printf("-89 = %f%n", calculate(v -> -v, 89));
      System.out.printf("18 * 18 = %f%n", calculate((double v) -> v * v, 18));
   }
   static double calculate(BinaryCalculator calc, double v1, double v2)
   {
      return calc.calculate(v1, v2);
   }
   static double calculate(UnaryCalculator calc, double v)
   {
      return calc.calculate(v);
   }
}
```

Listing 3 first introduces the `BinaryCalculator` and `UnaryCalculator` functional interfaces whose `calculate()` methods perform calculations on two input arguments or on a single input argument, respectively. This listing also introduces a `LambdaDemo` class whose `main()` method demonstrates these functional interfaces.

The functional interfaces are demonstrated in the `static double calculate(BinaryCalculator calc, double v1, double v2)` and `static double calculate(UnaryCalculator calc, double v)` methods. The lambdas pass code as data to these methods, which are received as `BinaryCalculator` or `UnaryCalculator` instances.

Compile Listing 3 and run the application. You should observe the following output:

```
18 + 36.5 = 54.500000
89 / 2.9 = 30.689655
-89 = -89.000000
18 * 18 = 324.000000
```

Target types

A lambda is associated with an implicit *target type*, which identifies the type of object to which a lambda is bound. The target type must be a functional interface that's inferred from the context, which limits lambdas to appearing in the following contexts:

- Variable declaration
- Assignment
- Return statement
- Array initializer
- Method or constructor arguments
- Lambda body
- Ternary conditional expression
- Cast expression

Listing 4 presents an application that demonstrates these target type contexts.

Listing 4. LambdaDemo.java (version 4)

```
import java.io.File;
import java.io.FileFilter;
import java.nio.file.Files;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.FileVisitor;
import java.nio.file.FileVisitResult;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.concurrent.Callable;
public class LambdaDemo
{
   public static void main(String[] args) throws Exception
   {
      // Target type #1: variable declaration
      Runnable r = () -> { System.out.println("running"); };
      r.run();
      // Target type #2: assignment
      r = () -> System.out.println("running");
      r.run();
      // Target type #3: return statement (in getFilter())
      File[] files = new File(".").listFiles(getFilter("txt"));
      for (int i = 0; i < files.length; i++)
         System.out.println(files[i]);
      // Target type #4: array initializer
      FileSystem fs = FileSystems.getDefault();
      final PathMatcher matchers[] =
      {
         (path) -> path.toString().endsWith("txt"),
         (path) -> path.toString().endsWith("java")
      };
      FileVisitor<Path> visitor;
      visitor = new SimpleFileVisitor<Path>()
                {
                   @Override
                   public FileVisitResult visitFile(Path file,
                                                    BasicFileAttributes attribs)
                   {
```

```
                    Path name = file.getFileName();
                    for (int i = 0; i < matchers.length; i++)
                    {
                       if (matchers[i].matches(name))
                          System.out.printf("Found matched file: '%s'.%n",
                                            file);
                    }
                    return FileVisitResult.CONTINUE;
                 }
              };
      Files.walkFileTree(Paths.get("."), visitor);
      // Target type #5: method or constructor arguments
      new Thread(() -> System.out.println("running")).start();
      // Target type #6: lambda body (a nested lambda)
      Callable<Runnable> callable = () -> () ->
         System.out.println("called");
      callable.call().run();
      // Target type #7: ternary conditional expression
      boolean ascendingSort = false;
      Comparator<String> cmp;
      cmp = (ascendingSort) ? (s1, s2) -> s1.compareTo(s2)
                            : (s1, s2) -> s2.compareTo(s1);
      List<String> cities = Arrays.asList("Washington", "London", "Rome",
                                          "Berlin", "Jerusalem", "Ottawa",
                                          "Sydney", "Moscow");
      Collections.sort(cities, cmp);
      for (int i = 0; i < cities.size(); i++)
         System.out.println(cities.get(i));
      // Target type #8: cast expression
      String user = AccessController.doPrivileged((PrivilegedAction<String>) ()
                                           -> System.getProperty("user.name"));
      System.out.println(user);
   }
   static FileFilter getFilter(String ext)
   {
      return (pathname) -> pathname.toString().endsWith(ext);
   }
}
```