JAVA 101: LEARN JAVA
By Jeff Friesen, JavaWorld
NOV 12, 2019 11:27 AM PST

About
A beginner's library for learning about essential Java programming concepts, syntax, APIs, and packages.

ADVANCED JAVA LANGUAGE FEATURES

Get started with method references in Java

Use method references to simplify functional programming in Java

Along with lambdas, Java SE 8 brought method references to the Java language. This tutorial offers a brief overview of method references in Java, then gets you started using them with Java code examples. By the end of the tutorial you will know how to use method references to refer to a class's static methods, bound and unbound non-static methods, and constructors, as well as how to use them to refer to instance methods in superclass and current class types. You'll also understand why many Java developers have adopted lambda expressions and method references as a cleaner, simpler alternative to anonymous classes.

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

Get the code

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

Method references: A primer

My previous Java 101 tutorial introduced lambda expressions, which are used to define anonymous methods that can then be treated as instances of a functional interface. Sometimes, a lambda expression does nothing more than call an existing method. For example, the following code fragment uses a lambda to invoke `System.out`'s `void println(s)` method on the lambda's single argument--`s`'s type is not yet known:

```
(s) -> System.out.println(s)
```

The lambda presents `(s)` as its formal parameter list and a code body whose `System.out.println(s)` expression prints `s`'s value to the standard output stream. It doesn't have an explicit interface type. Instead, the compiler infers from the surrounding context which functional interface to instantiate. For example, consider the following code fragment:

```
Consumer<String> consumer = (s) -> System.out.println(s);
```

The compiler analyzes the previous declaration and determines that the `java.util.function.Consumer` predefined functional interface's `void accept(T t)` method matches the lambda's formal parameter list (`(s)`). It also determines that `accept()`'s `void` return type matches `println()`'s `void` return type. The lambda is thus *bound* to `Consumer`.

More specifically, the lambda is bound to `Consumer<String>`. The compiler generates code so that an invocation of `Consumer<String>`'s `void accept(String s)` method results in the string argument passed to `s` being passed to `System.out`'s `void println(String s)` method. This invocation is shown below:

```
consumer.accept("Hello"); // Pass "Hello" to lambda body. Print Hello to standard output.
```

To save keystrokes, you can replace the lambda with a *method reference*, which is a compact reference to an existing method. For example, the following code fragment replaces `(String s) -> System.out.println(s)` with `System.out::println`, where `::` signifies that `System.out`'s `void println(String s)` method is being referenced:

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

```
Consumer<String> consumer2 = System.out::println; // The method reference is shorter.
consumer2.accept("Hello"); // Pass "Hello" to lambda body. Print Hello to standard output.
```

It isn't necessary to specify a formal parameter list for the previous method reference because the compiler can infer this list based on `Consumer<String>` This parameterized type's `java.lang.String` actual type argument replaces `T` in `void accept(T t)`, and is also the type of the single parameter in the lambda body's `System.out.println()` method call.

Method references in depth

A *method reference* is a syntactic shortcut for creating a lambda from an existing method. Instead of providing an implementation body, a method reference refers to an existing class's or object's method. As with a lambda, a method reference requires a target type.

You can use method references to refer to a class's static methods, bound and unbound non-static methods, and constructors. You can also use method references to refer to instance methods in superclass and current class types. I'll introduce you to each of these method reference categories and show how they're used in a small demo.

Learn more about method references
After reading this section, check out Method References in Java 8 (Toby Weston, February 2014) for more insight into method references in bound and unbound non-static method contexts.

References to static methods

A *static method reference* refers to a static method in a specific class. Its syntax is *`className`*`::`*`staticMethodName`*, where *`className`* identifies the class and *`staticMethodName`* identifies the static method. An example is `Integer::bitCount`. Listing 1 demonstrates a static method reference.

Listing 1. MRDemo.java (version 1)

```
import java.util.Arrays;
import java.util.function.Consumer;
public class MRDemo
{
   public static void main(String[] args)
   {
      int[] array = { 10, 2, 19, 5, 17 };
      Consumer<int[]> consumer = Arrays::sort;
      consumer.accept(array);
      for (int i = 0; i < array.length; i++)
         System.out.println(array[i]);
      System.out.println();
      int[] array2 = { 19, 5, 14, 3, 21, 4 };
      Consumer<int[]> consumer2 = (a) -> Arrays.sort(a);
      consumer2.accept(array2);
      for (int i = 0; i < array2.length; i++)
         System.out.println(array2[i]);
   }
}
```

Listing 1's `main()` method sorts a pair of integer arrays via the `java.util.Arrays` class's `static void sort(int[] a)` method, which appears in static method reference and equivalent lambda expression contexts. After sorting an array, a `for` loop prints the sorted array's contents to the standard output stream.

Before we can use a method reference or a lambda, it must be bound to a functional interface. I'm using the predefined `Consumer` functional interface, which meets the method reference/lambda requirements. The sort operation commences by passing the array to be sorted to `Consumer`'s `accept()` method.

Compile Listing 1 (`javac MRDemo.java`) and run the application (`java MRDemo`). You'll observe the following output:

```
2
5
10
17
19
3
4
5
14
19
21
```

References to bound non-static methods

A *bound non-static method reference* refers to a non-static method that's bound to a *receiver* object. Its syntax is *`objectName`*`::`*`instanceMethodName`*, where *`objectName`* identifies the receiver and *`instanceMethodName`* identifies the instance method. An example is `s::trim`. Listing 2 demonstrates a bound non-static method reference.

Listing 2. MRDemo.java (version 2)

```
import java.util.function.Supplier;
public class MRDemo
{
   public static void main(String[] args)
   {
      String s = "The quick brown fox jumped over the lazy dog";
      print(s::length);
      print(() -> s.length());
      print(new Supplier<Integer>()
      {
         @Override
         public Integer get()
         {
            return s.length(); // closes over s
         }
      });
   }
   public static void print(Supplier<Integer> supplier)
   {
      System.out.println(supplier.get());
   }
}
```

Listing 2's `main()` method assigns a string to `String` variable `s` and then invokes the `print()` class method with functionality to obtain this string's length as this method's argument. `print()` is invoked in method reference (`s::length` -- `length()` is bound to `s`), equivalent lambda, and equivalent anonymous class contexts.

I've defined `print()` to use the `java.util.function.Supplier` predefined functional interface, whose `get()` method returns a supplier of results. In this case, the `Supplier` instance passed to `print()` implements its `get()` method to return `s.length()`; `print()` outputs this length.

`s::length` introduces a closure that closes over `s`. You can see this more clearly in the lambda example. Because the lambda has no arguments, the value of `s` is only available from the enclosing scope. Therefore, the lambda body is a closure that closes over `s`. The anonymous class example makes this even clearer.

Compile Listing 2 and run the application. You'll observe the following output:

```
44
44
44
```

References to unbound non-static methods

An *unbound non-static method reference* refers to a non-static method that's not bound to a receiver object. Its syntax is *`className`*`::`*`instanceMethodName`*, where *`className`* identifies the class that declares the instance method and *`instanceMethodName`* identifies the instance method. An example is `String::toLowerCase`.

`String::toLowerCase` is an unbound non-static method reference that identifies the non-static `String toLowerCase()` method of the `String` class. However, because a non-static method still requires a receiver object (in this example a `String` object, which is used to invoke `toLowerCase()` via the method reference), the receiver object is created by the virtual machine. `toLowerCase()` will be invoked on this object. `String::toLowerCase` specifies a method that takes a single `String` argument, which is the receiver object, and returns a `String` result. `String::toLowerCase()` is equivalent to lambda `(String s) -> { return s.toLowerCase(); }`.

Listing 3 demonstrates this unbound non-static method reference.

Listing 3. MRDemo.java (version 3)

```
import java.util.function.Function;
public class MRDemo
{
   public static void main(String[] args)
   {
      print(String::toLowerCase, "STRING TO LOWERCASE");
      print(s -> s.toLowerCase(), "STRING TO LOWERCASE");
      print(new Function<String, String>()
      {
         @Override
         public String apply(String s) // receives argument in parameter s;
         {                             // doesn't need to close over s
            return s.toLowerCase();
         }
      }, "STRING TO LOWERCASE");
   }
   public static void print(Function<String, String> function, String s)
   {
      System.out.println(function.apply(s));
   }
}
```

Listing 3's `main()` method invokes the `print()` class method with functionality to convert a string to lowercase and the string to be converted as the method's arguments. `print()` is invoked in method reference (`String::toLowerCase`, where `toLowerCase()` isn't bound to a user-specified object) and equivalent lambda and anonymous class contexts.

I've defined `print()` to use the `java.util.function.Function` predefined functional interface, which represents a function that accepts one argument and produces a result. In this case, the `Function` instance passed to `print()` implements its `R apply(T t)` method to return `s.toLowerCase()`; `print()` outputs this string.

Although the `String` part of `String::toLowerCase` makes it look like a class is being referenced, only an instance of this class is referenced. The anonymous class example makes this more obvious. Note that in the anonymous class example the lambda receives an argument; it doesn't close over parameter `s` (i.e., it's not a closure).

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

```
string to lowercase
string to lowercase
string to lowercase
```

References to constructors

You can use a method reference to refer to a constructor without instantiating the named class. This kind of method reference is known as a *constructor reference*. Its syntax is *`className`*`::new`. *`className`* must support object creation; it cannot name an abstract class or interface. Keyword `new` names the referenced constructor. Here are some examples:

- `Character::new`: equivalent to lambda `(Character ch) -> new Character(ch)`
- `Long::new`: equivalent to lambda `(long value) -> new Long(value)` or `(String s) -> new Long(s)`
- `ArrayList<City>::new`: equivalent to lambda `() -> new ArrayList<City>()`
- `float[]::new`: equivalent to lambda `(int size) -> new float[size]`

The last constructor reference example specifies an array type instead of a class type, but the principle is the same. The example demonstrates an *array constructor reference* to the "constructor" of an array type.

To create a constructor reference, specify `new` without a constructor. When a class such as `java.lang.Long` declares multiple constructors, the compiler compares the functional interface's type against all of the constructors and chooses the best match. Listing 4 demonstrates a constructor reference.

Listing 4. MRDemo.java (version 4)

```
import java.util.function.Supplier;
public class MRDemo
{
   public static void main(String[] args)
   {
      Supplier<MRDemo> supplier = MRDemo::new;
      System.out.println(supplier.get());
   }
}
```

Listing 4's `MRDemo::new` constructor reference is equivalent to lambda `() -> new MRDemo()`. Expression `supplier.get()` executes this lambda, which invokes `MRDemo`'s default no-argument constructor and returns the `MRDemo` object, which is passed to `System.out.println()`. This method converts the object to a string, which it prints.

Now suppose you have a class with a no-argument constructor and a constructor that takes an argument, and you want to call the constructor that takes an argument. You can accomplish this task by choosing a different functional interface, such as the predefined `Function` interface shown in Listing 5.

Listing 5. MRDemo.java (version 5)

```
import java.util.function.Function;
public class MRDemo
{
   private String name;
   MRDemo()
   {
      name = "";
   }
   MRDemo(String name)
   {
      this.name = name;
      System.out.printf("MRDemo(String name) called with %s%n", name);
   }
   public static void main(String[] args)
   {
      Function<String, MRDemo> function = MRDemo::new;
      System.out.println(function.apply("some name"));
   }
}
```


IDG COMMUNICATIONS Copyright © 2020 IDG Communications, Inc.