An Introduction to Lambda Expressions in Java 8

Expanded Version 1.2

December 2014 (revised December 2016)

Contents

Many features have been added to the Java language and eco-system since its original 1.0 release. Two main features stand out because they have profoundly reshaped the language, changing the way that code is written.

1. The first was the addition of *generics*, added by JDK 5.
2. The second is the lambda expression.

lambda expressions (and their related features) significantly enhance Java because of two primary reasons.

1. they add new syntax elements that increase the expressive power of the language. They streamline the way that certain common constructs are implemented.
2. the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multi-core environments.

This applies to the handling of *for-each* style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the *default method*, which lets you define default behaviour for an interface method, and the *method reference*, which lets you refer to a method without executing it.

Beyond the benefits that lambda expressions bring to the language, there is another reason why they constitute an important addition to Java. Over the past few years, lambda expressions have become a major focus of computer language design. For example, they have been added to languages such as C# and C++. Their inclusion in JDK 8 helps Java remain a relevant programming language, rather than a dinosaur!

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions are reshaping Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

1 Introducing lambda expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression, itself. The second is the functional interface. Let's begin with a simple definition of each.

> A lambda expression is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class.

Lambda expressions are also commonly referred to as closures.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface `Runnable` is a functional interface because it defines only one method: `run()`. Therefore, `run()` defines the action of `Runnable`. Furthermore, a functional interface defines the target type of a lambda expression, and here is a key point:

> a lambda expression can be used only in a context in which its target type is specified.

A functional interface is sometimes referred to as a *SAM type — Single Abstract Method*.

It should be noted that a functional interface may specify any public method defined by `Object`, such as `equals()`, without affecting its "functional interface" status. The public `Object` methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

We will now consider both lambda expressions and functional interfaces.

2 Lambda expressions — the fundamentals

The lambda operator (or the arrow operator), is `->`. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The `->` can be *read* as "becomes" or "goes to".

Java defines two types of lambda bodies.

- One consists of a single expression, and
- the other type consists of a block of code.

We will begin with lambdas that define a single expression and discuss lambdas with block bodies later on.

At this point, it will be helpful to look a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value `123.45`. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from `Math.random()`, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2) == 0
```

This lambda expression returns `true` if the value of parameter `n` is even. Although it is possible to explicitly specify the type of a parameter, such as `n` in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

2.1 Function interfaces

A *functional interface* is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract. Although this was true prior to JDK 8, the situation has changed. Beginning with JDK 8, it is possible to specify default behaviour for a method declared in an interface. This is called a *default method*.

Today, an interface method is abstract only if it does not specify a default implementation. As non-default interface methods are implicitly abstract, there is no need to use the abstract modifier (although you can specify it, if you want to).

Here is an example of a functional interface:

```
interface MyNumber {
    double getValue();
}
```

In this case, the method `getValue()` is implicitly abstract, and it is the only method defined by `MyNumber`. Therefore, `MyNumber` is a functional interface, and its function is defined by `getValue()`.

A lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. A lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface

reference. Other target type contexts include variable initialisation, return statements, and method arguments, etc.

We now examine an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface `MyNumber` is declared:

```
// create a reference to a MyNumber instance.
MyNumber myNum;
```

Then a lambda expression is assigned to that interface reference:

```
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behaviour of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Therefore, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the `getValue()` method. Therefore, the following displays the value `123.45`:

```
System.out.println(myNum.getValue());
```

As the lambda expression assigned to `myNum` returns the value `123.45`, that is the value obtained when `getValue()` is called.

For a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two `int` parameters, then the lambda must specify two parameters whose type either is explicitly `int` or can be implicitly inferred as `int` by the context.

In general,

- the type and number of the lambda expression's parameters must be compatible with the method's parameters;
- the return types must be compatible; and
- any exceptions thrown by the lambda expression must be acceptable to the method.

2.2 Examples

Let us now examine some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing discussion.

```
package one;

//A functional interface.
public interface MyNumber {
  double getValue();
}
```

```
package one;

public class LambdaDemo {
  public static void main(String args[]) {
    MyNumber myNum; // declare an interface reference

    // Here, the lambda expression is simply a constant expression.
    // When it is assigned to myNum, a class instance is
    // constructed in which the lambda expression provides an override
    // of the getValue() method in MyNumber.
    myNum = () -> 123.45;

    // Call getValue(), which is overridden by the previously assigned
    // lambda expression.
    System.out.println("A fixed value: " + myNum.getValue());

    // Here, a more complex expression is used.
    myNum = () -> Math.random() * 100;

    // These call the lambda expression in the previous line.
    System.out.println("A random value: " + myNum.getValue());
    System.out.println("Another random value: " + myNum.getValue());

    // A lambda expression must be compatible with the method
    // defined by the functional interface. Therefore, this won't work:
    // myNum = () -> "123.03"; // Error!
  }
}
```

The output from the program is:

```
A fixed value: 123.45
A random value: 24.497044687345248
Another random value: 21.036350099244714
```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type `String` is not compatible with `double`, which is the return type required by `getValue()`.

The next example shows the use of a parameter with a lambda expression:

```
package two;

//Another functional interface.
public interface NumericTest {
  boolean test(int n);
}
```

```
package two;

public class LambdaDemo {
  public static void main(String args[]) {
    // A lambda expression that tests if a number is even.
    NumericTest isEven = (n) -> (n % 2) == 0;

    if (isEven.test(10))
      System.out.println("10 is even");
    if (!isEven.test(9))
      System.out.println("9 is not even");

    // Now, use a lambda expression that tests if a number
    // is non-negative.
    NumericTest isNonNeg = (n) -> n >= 0;

    if (isNonNeg.test(1))
      System.out.println("1 is non-negative");
    if (!isNonNeg.test(-1))
      System.out.println("-1 is negative");
  }
}
```

The output from the program is:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for *evenness*. It is shown again here:

```
(n) -> (n % 2) == 0
```

Notice that the type of `n` is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of `test()` as defined by the `NumericTest` interface, which is `int`. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid:

```
(int n) -> (n % 2) == 0
```

Here, `n` is explicitly specified as `int`. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it.

This program demonstrates another important point about lambda expressions:

> A functional interface reference can be used to execute any lambda expression that is compatible with it.

Notice that the program defines two different lambda expressions that are compatible with the `test()` method of the functional interface `NumericTest`.

The first, called `isEven`, determines if a value is even. The second, called `isNonNeg`, checks if a value is non-negative. In each case, the value of the parameter `n` is tested. As each lambda expression is compatible with `test()`, each can be executed through a `NumericTest` reference.

When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2) == 0
```

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
package three;

//Demonstrate a lambda expression that takes two parameters.

public interface NumericTest {
  boolean test(int n, int d);
}
```

```
package three;

public class LambdaDemo {
```

```
  public static void main(String args[]) {
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest isFactor = (n, d) -> (n % d) == 0;

    if (isFactor.test(10, 2))
      System.out.println("2 is a factor of 10");

    if (!isFactor.test(10, 3))
      System.out.println("3 is not a factor of 10");
  }
}
```

The output is shown here:

```
2 is a factor of 10
3 is not a factor of 10
```

In this program, the functional interface `NumericTest` defines the `test()` method:

```
boolean test(int n, int d);
```

In this version, `test()` specifies two parameters. Thus, for a lambda expression to be compatible with `test()`, the lambda expression must also specify two parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, `n` and `d`, are specified in the parameter list, separated by commas. This example can be generalised.

> Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesised list on the left side of the lambda operator.

If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n,d) -> (n % d) == 0
```

2.3 Block lambda expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas*. In an expression body, the code on the right side of the lambda operator must consist of a single expression. While expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body*. Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify if and switch statements, create nested blocks, etc. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Apart from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a `return` statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an `int` value:

```
package four;

//A block lambda that computes the factorial of an int value.

public interface NumericFunc {
  int func(int n);
}
```

```
package four;

public class LambdaDemo {
  public static void main(String args[]) {

    // This block lambda computes the factorial of an int value.
    NumericFunc factorial = (n) -> {
      int result = 1;

      for (int i = 1; i <= n; i++)
        result = i * result;

      return result;
    };
```

```
    System.out.println("The factorial of 3 is " + factorial.func(3));
    System.out.println("The factorial of 5 is " + factorial.func(5));
  }
}
```

The output from the program is:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

Notice that the block lambda declares a variable called `result`, uses a `for` loop, and has a `return` statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. When a `return` statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
package five;

//A block lambda that reverses the characters in a string.

public interface StringFunc {
  String func(String n);
}
```

```
package five;

public class LambdaDemo {
  public static void main(String args[]) {

    // This block lambda that reverses the charactrers in a string.
    StringFunc reverse = (str) -> {
      String result = "";
      int i;

      for (i = str.length() - 1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " + reverse.func("Lambda"));
    System.out.println("Expression reversed is "
        + reverse.func("Expression"));
  }
}
```

The output from the program is:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface `StringFunc` declares the `func()` method. This method takes a parameter of type `String` and has a return type of `String`. Thus, in the reverse lambda expression, the type of `str` is inferred to be `String`. Notice that the `charAt()` method is called on `str`. This is legal because of the inference that `str` is of type `String`.

2.4 Generic functional interfaces

A lambda expression, itself, cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some "generic" qualities.)

However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is determined, in part, by the type argument or arguments specified when a functional interface reference is declared. To understand the value of generic functional interfaces, consider the following example.

The two examples in the previous section used two different functional interfaces, one called `NumericFunc` and the other called `StringFunc`. However, both defined a method called `func()` that took one parameter and returned a result. In the first case, the type of the parameter and return type was `int`. In the second case, the parameter and return type was `String`. Therefore the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances.

```
package six;

//Use a generic functional interface with lambda expressions.

//A generic functional interface.
public interface SomeFunc<T> {
  T func(T t);
}
```

```
package six;

public class GenericFunctionalInterfaceDemo {
  public static void main(String args[]) {

    // Use a String-based version of SomeFunc.
    SomeFunc<String> reverse = (str) -> {
```

```
      String result = "";
      int i;

      for (i = str.length() - 1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " + reverse.func("Lambda"));
    System.out.println("Expression reversed is "
        + reverse.func("Expression"));

    // Now, use an Integer-based version of SomeFunc.
    SomeFunc<Integer> factorial = (n) -> {
      int result = 1;

      for (int i = 1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

The output from the program is:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factoral of 3 is 6
The factoral of 5 is 120
```

In the program, the generic functional interface `SomeFunc` is declared as shown here:

```
public interface SomeFunc<T> {
    T func(T t);
}
```

Here, `T` specifies both the return type and the parameter type of `func()`. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The `SomeFunc` interface is used to provide a reference to two different types of lambdas.

- The first uses type `String`.
- The second uses type `Integer`.

Thus, the same functional interface can be used to refer to the reverse lambda and the factorial lambda. Only the type argument passed to `SomeFunc` differs.

2.5 Passing lambda expressions as arguments

Remember that a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. Passing a lambda expression as an argument is a common use of lambdas and it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
package seven;

//Use lambda expressions as an argument to a method.

public interface StringFunc {
  String func(String n);
}
```

```
package seven;

public class LambdasAsArgumentsDemo {
  //
  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed a reference to
  // any instance of that interface, including the instance created
  // by a lambda expression.
  //
  // The second parameter specifies the string to operate on.
  //
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[]) {
    String inStr = "Lambdas add power to Java";
    String outStr;
```

```
    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) -> {
      String result = "";
      int i;

      for (i = 0; i < str.length(); i++)
        if (str.charAt(i) != ' ')
          result += str.charAt(i);

      return result;
    }, inStr);

    System.out.println("The string with spaces removed: " + outStr);

    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to a
    // synthetic instance of StringFunc.
    StringFunc reverse = (str) -> {
      String result = "";
      int i;

      for (i = str.length() - 1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    // Now, reverse can be passed as the first parameter to stringOp()
    // since it refers to a StringFunc object.
    System.out.println("The string reversed: " + stringOp(reverse, inStr));
  }
}
```

The output from the program is:

```
Here is input string: Lambdas add power to Java
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL
```

In this program, first notice the `stringOp()` method. It has two parameters. The first is of type `StringFunc`, which is a functional interface. Therefore this parameter can receive a reference to any instance of `StringFunc`, including one created by a lambda expression. The second argument of `stringOp()` is of type `String`, and this is the string operated on. Consider the first call to `stringOp()`:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface `StringFunc` is created and a reference to that object is passed to the first parameter of `stringOp()`. The lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter.

As the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique, especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to `stringOp()`. This lambda removes spaces from a string:

```
outStr = stringOp((str) -> {
  String result = "";
  int i;

  for (i = 0; i < str.length(); i++)
    if (str.charAt(i) != ' ')
      result += str.charAt(i);

  return result;
}, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case, however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to `reverse`, which is a reference to a `StringFunc` instance. Therefore, `reverse` can be used as an argument to the first parameter of `stringOp()`.

The program then calls `stringOp()`, passing in `reverse` and the string on which to operate. As the instance obtained by the evaluation of each lambda expression is an implementation of `StringFunc`, each can be used as the first parameter to `stringOp()`.

3 Lambda expressions and exceptions

A lambda expression can throw an exception. If it throws a *checked exception*, then that exception must be compatible with the exception(s) listed in the `throws` clause of the

abstract method in the functional interface.

Here is an example that illustrates this. It computes the average of an array of double values. If a zero-length array is passed it throws the custom exception `EmptyArrayException`. As the example shows, this exception is listed in the `throws` clause of `func()` declared inside the `DoubleNumericArrayFunc` functional interface.

```
package eight;

//Throw an exception from a lambda expression.

public interface DoubleNumericArrayFunc {
  double func(double[] n) throws EmptyArrayException;
}
```

```
package eight;

class EmptyArrayException extends Exception {
  private static final long serialVersionUID = 1L;

  EmptyArrayException() {
    super("Array Empty");
  }
}
```

```
package eight;

public class LambdaDemo {
  public static void main(String args[]) throws EmptyArrayException {
    double[] values = { 1.0, 2.0, 3.0, 4.0 };

    // This block lambda computes the average of an array of doubles.
    DoubleNumericArrayFunc average = (n) -> {
      double sum = 0;

      if (n.length == 0)
        throw new EmptyArrayException();

      for (int i = 0; i < n.length; i++)
        sum += n[i];

      return sum / n.length;
    };

    System.out.println("The average is " + average.func(values));

    // This causes an exception to be thrown.
```

```
    System.out.println("The average is " + average.func(new double[0]));
  }
}
```

The output from the program is:

```
Exception in thread "main" The average is 2.5
eight.EmptyArrayException: Array Empty
  at eight.LambdaDemo.lambda$0(LambdaDemo.java:12)
  at eight.LambdaDemo$$Lambda$1/140435067.func(Unknown Source)
  at eight.LambdaDemo.main(LambdaDemo.java:23)
```

The first call to `average.func()` returns the value `2.5`. The second call, which passes a zero-length array, causes an `EmptyArrayException` to be thrown. Remember, the inclusion of the `throws` clause in `func()` is necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with `func()`.

This example demonstrates another important point about lambda expressions, that the parameter specified by `func()` in the functional interface `DoubleNumericArrayFunc` is an array. However, the parameter to the lambda expression is simply `n`, rather than `n[]`. Remember, the type of a lambda expression parameter will be inferred from the target context. In this case, the target context is `double[]`, thus the type of `n` will be `double[]`. It is not necessary (or legal) to specify it as `n[]`. It would be legal to explicitly declare it as `double[] n`, but doing so gains nothing in this case.

4 Lambda expressions and variable capture

Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or static variable defined by its enclosing class. A lambda expression also has access to this (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an instance or static variable and call a method defined by its enclosing class.

However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. In this case, a lambda expression may only use local variables that are effectively `final`. An *effectively final* variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as `final`, although doing so would not be an error. (The `this` parameter of an enclosing scope is automatically effectively `final`, and lambda expressions do not have a `this` of their own.)

It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

The following program illustrates the difference between effectively final and mutable local variables:

```
package nine;

//An example of capturing a local variable from the enclosing scope.

public interface MyFunc {
  int func(int n);
}
```

```
package nine;

public class LambdaDemo {
  public static void main(String args[]) {
    // A local variable that can be captured.
    int num = 10;

    MyFunc myLambda = (n) -> {
      // This use of num is OK. It does not modify num.
      int v = num + n;

      // However, the following is illegal because it attempts
      // to modify the value of num.
      // num++;

      return v;
    };

    // The following line would also cause an error, because
    // it would remove the effectively final status from num.
    // num = 9;
  }
}
```

As the comments indicate, `num` is effectively `final` and can therefore, be used inside `myLambda`. However, if `num` were to be modified, either inside the lambda or outside of it, `num` would lose its effectively final status. This would cause an error, and the program would not compile.

It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just canâĂŹt use a local variable of its enclosing scope unless that variable is effectively final.

5 Method references

There is an important feature related to lambda expressions called a *method reference*. A method reference provides a way to refer to a method without executing it. It relates

to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

There are different types of method references. We will begin with method references to static methods.

5.1 Method references to static methods

To create a static method reference, use this general syntax:

```
ClassName::methodName
```

Notice that the class name is separated from the method name by a double colon. The `::` is a new separator that has been added to Java expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a static method reference:

```
package ten;

//Demonstrate a method reference for a static method.

//A functional interface for string operations.
public interface StringFunc {
  String func(String n);
}
```

```
package ten;

//This class defines a static method called strReverse().
public class MyStringOps {
  // A static method that reverses a string.
  public static String strReverse(String str) {
    String result = "";
    int i;

    for (i = str.length() - 1; i >= 0; i--)
      result += str.charAt(i);

    return result;
  }
}
```

```
package ten;
```

```
public class MethodRefDemo {
  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including a method reference.
  private static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[]) {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Here, a method reference to strReverse is passed to stringOp().
    outStr = stringOp(MyStringOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
    System.out.println("String reversed: " + outStr);
  }
}
```

The output is shown here:

```
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
```

In the program, pay special attention to this line:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Here, a reference to the static method `strReverse()`, declared inside `MyStringOps`, is passed as the first argument to `stringOp()`. This works because `strReverse` is compatible with the `StringFunc` functional interface. Thus, the expression `MyStringOps::strReverse` evaluates to a reference to an object in which `strReverse` provides the implementation of `func()` in `StringFunc`.

5.2 Method references to instance methods

To pass a reference to an instance method on a specific object, use this basic syntax:

```
objRef::methodName
```

As you can see, the syntax is similar to that used for a static method, except that an object reference is used instead of a class name. Here is the previous program rewritten to use an instance method reference:

```
package eleven;

//Demonstrate a method reference to an instance method

//A functional interface for string operations.
public interface StringFunc {
  String func(String n);
}
```

```
package eleven;

//Now, this class defines an instance method called strReverse().
public class MyStringOps {
  String strReverse(String str) {
    String result = "";
    int i;

    for (i = str.length() - 1; i >= 0; i--)
      result += str.charAt(i);

    return result;
  }
}
```

```
package eleven;

public class MethodRefDemo {
  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including method references.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[]) {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Create a MyStringOps object.
    MyStringOps strOps = new MyStringOps();

    // Now, a method reference to the instance method strReverse
    // is passed to stringOp().
    outStr = stringOp(strOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
```

```
    System.out.println("String reversed: " + outStr);
  }
}
```

The output is shown here:

```
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
```

In the program, notice that `strReverse()` is now an instance method of `MyStringOps`. Inside `main()`, an instance of `MyStringOps` called `strOps` is created. This instance is used to create the method reference to `strReverse` in the call to `stringOp`, as shown again, here:

```
outStr = stringOp(strOps::strReverse, inStr);
```

In this example, `strReverse()` is called on the `strOps` object.

It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class — not just a specified object. In this case, you will create a method reference as shown here:

```
ClassName::instanceMethodName
```

Here, the name of the class is used instead of a specific object, even though an instance method is specified. With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter specified by the method. Here is an example. It defines a method called `counter()` that counts the number of objects in an array that satisfy the condition defined by the `func()` method of the `MyFunc` functional interface. In this case, it counts instances of the `HighTemp` class.

```
package twelve;

//Use an instance method reference with different objects.

//A functional interface that takes two reference arguments
//and returns a boolean result.
public interface MyFunc<T> {
  boolean func(T v1, T v2);
}
```

```
package twelve;

//A class that stores the temperature high for a day.
public class HighTemp {
```

```
  private int hTemp;

  public HighTemp(int ht) {
    hTemp = ht;
  }

  // Return true if the invoking HighTemp object has the same
  // temperature as ht2.
  public boolean sameTemp(HighTemp ht2) {
    return hTemp == ht2.hTemp;
  }

  // Return true if the invoking HighTemp object has a temperature
  // that is less than ht2.
  public boolean lessThanTemp(HighTemp ht2) {
    return hTemp < ht2.hTemp;
  }
}
```

```
package twelve;

public class InstanceMethWithObjectRefDemo {

  // A method that returns the number of occurences
  // of an object for which some criteria, as specified by
  // the MyFunc parameter, is true.
  static <T> int counter(T[] vals, MyFunc<T> f, T v) {
    int count = 0;

    for (int i = 0; i < vals.length; i++)
      if (f.func(vals[i], v))
        count++;

    return count;
  }

  public static void main(String args[]) {
    int count;

    // Create an array of HighTemp objects.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
        new HighTemp(90), new HighTemp(89), new HighTemp(89),
        new HighTemp(91), new HighTemp(84), new HighTemp(83) };

    // Use counter() with arrays of the class HighTemp.
    // Notice that a reference to the instance method
    // sameTemp() is passed as the second argument.
    count = counter(weekDayHighs, HighTemp::sameTemp, new HighTemp(89));
```

```
    System.out.println(count + " days had a high of 89");

    // Now, create and use another array of HighTemp objects.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
        new HighTemp(24), new HighTemp(19), new HighTemp(18),
        new HighTemp(12), new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp, new HighTemp(12));
    System.out.println(count + " days had a high of 12");

    // Now, use lessThanTemp() to find days when temperature was less
    // that a specified value.
    count = counter(weekDayHighs, HighTemp::lessThanTemp, new HighTemp(89));
    System.out.println(count + " days had a high less than 89");

    count = counter(weekDayHighs2, HighTemp::lessThanTemp, new HighTemp(19));
    System.out.println(count + " days had a high of less than 19");
  }
}
```

The output is shown here:

```
3 days had a high of 89
2 days had a high of 12
3 days had a high less than 89
5 days had a high of less than 19
```

In the program, notice that `HighTemp` has two instance methods: `sameTemp()` and `lessThanTemp()`. The first returns `true` if two `HighTemp` objects contain the same temperature. The second returns `true` if the temperature of the invoking object is less than that of the passed object. Each method has a parameter of type `HighTemp` and each method returns a boolean result. Therefore, each is compatible with the `MyFunc` functional interface because the invoking object type can be mapped to the first parameter of `func()` and the argument mapped to `func()`'s second parameter. When the expression

```
HighTemp::sameTemp
```

is passed to the `counter()` method, an instance of the functional interface `MyFunc` is created in which the parameter type of the first parameter is that of the invoking object of the instance method, which is `HighTemp`. The type of the second parameter is also `HighTemp` because that is the type of the parameter to `sameTemp()`. The same is true for the `lessThanTemp()` method.

One other point: you can refer to the superclass version of a method by use of `super`, as shown here:

```
super::name
```

The name of the method is specified by `name`.

5.3 Method references with generics

You can use method references with generic classes and/or generic methods. For example, consider the following program:

```
package thirteen;

//Demonstrate a method reference to a generic method
//declared inside a non-generic class.

//A functional interface that operates on an array
//and a value, and returns an int result.
public interface MyFunc<T> {
  int func(T[] vals, T v);
}
```

```
package thirteen;

//This class defines a method called countMatching() that
//returns the number of items in an array that are equal
//to a specified value. Notice that countMatching()
//is generic, but MyArrayOps is not.
public class MyArrayOps {
  public static <T> int countMatching(T[] vals, T v) {
    int count = 0;

    for (int i = 0; i < vals.length; i++)
      if (vals[i] == v)
        count++;

    return count;
  }
}
```

```
package thirteen;

public class GenericMethodRefDemo {

  // This method has the MyFunc functional interface as the
  // type of its first parameter. The other two parameters
  // receive an array and a value, both of type T.
  public static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
    return f.func(vals, v);
  }

  public static void main(String args[]) {
    Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
```

```
    String[] strs = { "One", "Two", "Three", "Two" };
    int count;

    count = myOp(MyArrayOps::<Integer> countMatching, vals, 4);
    System.out.println("vals contains " + count + " 4s");

    count = myOp(MyArrayOps::<String> countMatching, strs, "Two");
    System.out.println("strs contains " + count + " Twos");
  }
}
```

The output is shown here:

```
vals contains 3 4s
strs contains 2 Twos
```

In the program, `MyArrayOps` is a non-generic class that contains a generic method called `countMatching()`. The method returns a count of the elements in an array that match a specified value. Notice how the generic type argument is specified. For example, its first call in `main()`, shown here:

```
count = myOp(MyArrayOps::<Integer> countMatching, vals, 4);
```

passes the type argument `Integer`. Notice that it occurs after the `::`. This syntax can be generalised:

> When a generic method is specified as a method reference, its type argument comes after the `::` and before the method name.

It is important to point out, however, that explicitly specifying the type argument is not required in this situation (and many others) because the type argument would have been automatically inferred. In cases in which a generic class is specified, the type argument follows the class name and precedes the `::`.

Although the preceding examples show the mechanics of using method references, they don't show their real benefits. One place method references can be quite useful is in conjunction with the Collections Framework (http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html). For example, we are going to use a method reference to help determine the largest element in a collection.

One way to find the largest element in a collection is to use the `max()` method defined by the `Collections` class. For the version of `max()` used here, you must pass a reference to the collection and an instance of an object that implements the `Comparator<T>` interface. This interface specifies how two objects are compared. It defines only one abstract method, called `compare()`, that takes two arguments, each the type of the objects being compared. It must return

- greater than zero if the first argument is greater than the second,
- zero if the two arguments are equal, and
- less than zero if the first object is less than the second.

In the past, to use `max()` with user-defined objects, an instance of `Comparator<T>` had to be obtained by first explicitly implementing it by a class, and then creating an instance of that class. This instance was then passed as the comparator to `max()`. With Java 8, it is possible to simply pass a reference to a comparison method to `max()` because doing so automatically implements the comparator.

The following simple example shows the process by creating an `ArrayList` of `MyClass` objects and then finding the one in the list that has the highest value (as defined by the comparison method).

```
package fourteen;

//Use a method reference to help find the maximum value in a collection.

public class MyClass {
  private int val;

  MyClass(int v) {
    val = v;
  }

  int getVal() {
    return val;
  }
}
```

```
package fourteen;

import java.util.ArrayList;
import java.util.Collections;

public class UseMethodRef {
  // A compare() method compatible with the one defined by Comparator<T>.
  static int compareMC(MyClass a, MyClass b) {
    return a.getVal() - b.getVal();
  }

  public static void main(String args[]) {
    ArrayList<MyClass> al = new ArrayList<MyClass>();

    al.add(new MyClass(1));
    al.add(new MyClass(4));
    al.add(new MyClass(2));
```

```
    al.add(new MyClass(9));
    al.add(new MyClass(3));
    al.add(new MyClass(7));

    // Find the maximum value in al using the compareMC() method.
    MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

    System.out.println("Maximum value is: " + maxValObj.getVal());
  }
}
```

The output is shown here:

```
Maximum value is: 9
```

In the program, notice that `MyClass` neither defines any comparison method of its own, nor does it implement `Comparator`. However, the maximum value of a list of `MyClass` items can still be obtained by calling `max()` because `UseMethodRef` defines the static method `compareMC()`, which is compatible with the `compare()` method defined by `Comparator`. Therefore, there is no need to explicitly implement and create an instance of `Comparator`.

5.4 Constructor references

Similar to the way that you can create references to methods, you can create references to constructors. Here is the general form of the syntax that you will use:

```
classname::new
```

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor. Here is a simple example:

```
package fifteen;

//Demonstrate a Constructor reference.

//MyFunc is a functional interface whose method returns
//a MyClass reference.
public interface MyFunc {
  MyClass func(int n);
}
```

```
package fifteen;
```

```
public class MyClass {
  private int val;

  // This constructor takes an argument.
  MyClass(int v) {
    val = v;
  }

  // This is the default constructor.
  MyClass() {
    val = 0;
  }

  // ...

  int getVal() {
    return val;
  };
}
```

```
package fifteen;

public class ConstructorRefDemo {
  public static void main(String args[]) {
    // Create a reference to the MyClass constructor.
    // Because func() in MyFunc takes an argument, new
    // refers to the parameterized constructor in MyClass,
    // not the default constructor.
    MyFunc myClassCons = MyClass::new;

    // Create an instance of MyClass via that constructor reference.
    MyClass mc = myClassCons.func(100);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal());
  }
}
```

The output is shown here:

```
val in mc is 100
```

In the program, notice that the `func()` method of `MyFunc` returns a reference of type `MyClass` and has an `int` parameter. Next, notice that `MyClass` defines two constructors.

- The first specifies a parameter of type `int`.

- The second is the *default*, parameterless constructor.

Now, examine the following line:

```
MyFunc myClassCons = MyClass::new;
```

Here, the expression `MyClass::new` creates a constructor reference to a `MyClass` constructor. In this case, because `MyFunc`'s `func()` method takes an `int` parameter, the constructor being referred to is `MyClass(int v)` because it is the one that matches. Also notice that the reference to this constructor is assigned to a `MyFunc` reference called `myClassCons`. After this statement executes, `myClassCons` can be used to create an instance of `MyClass`, as this line shows:

```
MyClass mc = myClassCons.func(100);
```

In essence, `myClassCons` has become another way to call `MyClass(int v)`.

Constructor references to generic classes are created in the same fashion. The only difference is that the type argument can be specified. This works the same as it does for using a generic class to create a method reference: simply specify the type argument after the class name. The following illustrates this by modifying the previous example so that `MyFunc` and `MyClass` are generic.

```
package sixteen;

//Demonstrate a constructor reference with a generic class.

//MyFunc is now a generic functional interface.
public interface MyFunc<T> {
  MyClass<T> func(T n);
}
```

```
package sixteen;

public class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) {
    val = v;
  }

  // This is the default constructor.
  MyClass() {
    val = null;
  }
```

```
  // ...

  T getVal() {
    return val;
  };
}
```

```
package sixteen;

public class ConstructorRefDemo {
  public static void main(String args[]) {
    // Create a reference to the MyClass<T> constructor.
    MyFunc<Integer> myClassCons = MyClass<Integer>::new;

    // Create an instance of MyClass<T> via that constructor reference.
    MyClass<Integer> mc = myClassCons.func(100);

    // Use the instance of MyClass<T> just created.
    System.out.println("val in mc is " + mc.getVal());
  }
}
```

This program produces the same output as the previous version.

The difference is that now both `MyFunc` and `MyClass` are generic. Thus, the sequence that creates a constructor reference can include a type argument (although one is not always needed), as shown here:

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Because the type argument Integer has already been specified when `myClassCons` is created, it can be used to create a `MyClass<Integer>` object, as the next line shows:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Although the preceding examples demonstrate the mechanics of using a constructor reference, no one would use a constructor reference as just shown because nothing is gained. Furthermore, having what amounts to two names for the same constructor creates a confusing situation (to say the least). However, to give you the flavour of a more practical usage, the following program uses a static method, called `myClassFactory()`, that is a factory for objects of any type of `MyFunc` objects. It can be used to create any type of object that has a constructor compatible with its first parameter.

```
package seventeen;
```

```
//Implement a simple class factory using a constructor reference.

public interface MyFunc<R, T> {
  R func(T n);
}
```

```
package seventeen;

//A simple generic class.
public class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) {
    val = v;
  }

  // The default constructor. This constructor
  // is NOT used by this program.
  MyClass() {
    val = null;
  }

  // ...

  T getVal() {
    return val;
  };
}
```

```
package seventeen;

// A simple, non-generic class.
public class MyClass2 {
  String str;

  // A constructor that takes an argument.
  MyClass2(String s) {
    str = s;
  }

  // The default constructor. This
  // constructor is NOT used by this program.
  MyClass2() {
    str = "";
  }
```

```
  // ...

  String getVal() {
    return str;
  };
}
```

```
package seventeen;

public class ConstructorRefDemo {
  // A factory method for class objects. The class must
  // have a constructor that takes one parameter of type T.
  // R specifies the type of object being created.
  static <R, T> R myClassFactory(MyFunc<R, T> cons, T v) {
    return cons.func(v);
  }

  public static void main(String args[]) {
    // Create a reference to a MyClass constructor.
    // In this case, new refers to the constructor that
    // takes an argument.
    MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

    // Create an instance of MyClass by use of the factory method.
    MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal());

    // Now, create a different class by use of myClassFactory().
    MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

    // Create an instance of MyClass2 by use of the factory method.
    MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");

    // Use the instance of MyClass just created.
    System.out.println("str in mc2 is " + mc2.getVal());
  }
}
```

The output is shown here:

```
val in mc is 100.1
str in mc2 is Lambda
```

As you can see, `myClassFactory()` is used to create objects of type `MyClass<Double>` and `MyClass2`. Although both classes differ, for example `MyClass` is generic and `MyClass2`

is not, both can be created by `myClassFactory()` because they both have constructors that are compatible with `func()` in `MyFunc`. This works because `myClassFactory()` is passed the constructor for the object that it builds. You might want to experiment with this program a bit, trying different classes that you create. Try creating instances of different types of `MyClass` objects. As you will see, `myClassFactory()` can create any type of object whose class has a constructor that is compatible with `func()` in `MyFunc`. Although this example is quite simple, it hints at the power that constructor references bring to Java.

There is a second form of the constructor reference syntax that is used for arrays. To create a constructor reference for an array, use this construct:

```
type[]::new
```

Here, `type` specifies the type of object being created. For example, assuming the form of `MyClass` as shown in the first constructor reference example (`ConstructorRefDemo`) and given the `MyArrayCreator` interface shown here:

```
package seventeen;

public interface MyArrayCreator<T> {
  T func(int n);
}
```

the following creates a two-element array of `MyClass` objects and gives each element an initial value:

```
MyArrayCreator<MyClass[]> myArrayCons = MyClass[]::new;
MyClass[] a = myArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);
```

Here, the call to `func(2)` causes a two-element array to be created. In general, a functional interface must contain a method that takes a single int parameter if it is to be used to refer to an array constructor.

6 Predefined functional interfaces

The examples we have considered so far have defined their own functional interfaces so that the fundamental concepts behind lambda expressions and functional interfaces could be clearly illustrated. However, in many cases, you won't need to define your own functional interface because Java 8 added a new package called `java.util.function` that provides several predefined ones:

Interface	Purpose
`UnaryOperator<T>`	Apply a unary operation to an object of type `T` and return the result, which is also of type `T`. Its method is called `apply()`
`BinaryOperator<T>`	Apply an operation to two objects of type `T` and return the result, which is also of type `T`. Its method is called `apply()`
`Consumer<T>`	Apply an operation on an object of type `T`. Its method is called `accept()`
`Supplier<T>`	Return an object of type `T`. Its method is called `get()`
`Function<T, R>`	Apply an operation to an object of type `T` and return the result as an object of type `R`. Its method is called `apply()`
`Predicate<T>`	Determine if an object of type `T` fulfils some constraint. Return a boolean value that indicates the outcome. Its method is called `test()`

We will examine some of these under the *streams* section of the notes.

The following program shows the `Function` interface in action by using it to rework the earlier example called `BlockLambdaDemo` that demonstrated block lambdas by implementing a factorial example. That example created its own functional interface called `NumericFunc`, but the built-in `Function` interface could have been used, as this version of the program illustrates:

```
package eighteen;

//Use the Function built-in functional interface.

//Import the Function interface.
import java.util.function.Function;

public class UseFunctionInterfaceDemo {
  public static void main(String args[]) {

    // This block lambda computes the factorial of an int value.
    // This time, Function is the functional interface.
    Function<Integer, Integer> factorial = (n) -> {
      int result = 1;
      for (int i = 1; i <= n; i++)
        result = i * result;
      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.apply(3));
    System.out.println("The factoral of 5 is " + factorial.apply(5));
  }
}
```

It produces the same output as the previous versions of the program.

To be continued...

Credits

These notes are based upon three main sources:

- *Java The Complete Reference, 9th Edition* by Herbert Schildt, Published by McGraw-Hill Osborne Media, 2014
- *Java 8: Lambdas*, by Ted Neward, Oracle Technology Network, 2014
- *Java 8 in Action — Lambdas, streams, and functional-style programming*, by Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft, Manning Press, 2014.
- ...