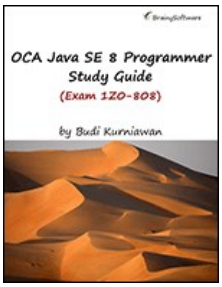

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 10: Lambda Expressions

Overview

The lambda expression is the most important new feature in Java SE 8. Long considered a missing feature in Java, it has made the Java language complete. At least for now. In this chapter you will learn what lambda expressions are and why they are a nice addition to the language. You will also be introduced to new technical terms such as single abstract method (SAM) and functional interface.

Before you get acquainted with lambda expressions, however, you should first learn about anonymous inner classes. This is because a lambda expression is basically a shorter way of doing the same thing with an anonymous inner class. Knowing what anonymous inner classes are helps you understand lambda expressions better. The first section of this chapter is therefore a brief discussion of anonymous inner classes.

10.1 Anonymous Inner Classes

An anonymous inner class is a nested class, which is a class that is defined within another class. An anonymous inner class does not have a name and is mainly used for writing an interface implementation that will only be used locally within a method. An anonymous inner class is used when there is no point in defining a separate class for the implementation because the implementation will only be used once.

For example, the **AnonymousInnerClassDemo1** class in Listing 10.1 creates an anonymous inner class that is an implementation of **Printable**.

Listing 10.1: Using an anonymous inner class

```
package app10;
interface Printable {
    void print(String message);
}

public class AnonymousInnerClassDemo1 {
    public static void main(String[] args) {

        Printable printer = new Printable() {
            public void print(String message) {
                System.out.println(message);
            }
        }; // this is a semicolon

        printer.print("Beach Music");
    }
}
```

The interesting thing here is that you create an anonymous inner class by using the **new** keyword followed by what looks like a class's constructor (in this case **Printable()**). However, note that **Printable** is an interface and does not have a constructor. **Printable()** is followed by the implementation of the **print** method. Also, note that after the closing brace, you use a semicolon to terminate the statement that instantiates the anonymous inner class.

In addition, you can also create an anonymous inner class by extending an abstract or concrete class, as demonstrated in the code in Listing 10.2.

Listing 10.2: Using an anonymous inner class with an abstract class

```
package app10;
abstract class Printable2 {
    void print(String message) {
    }
}

public class AnonymousInnerClassDemo2 {
    public static void main(String[] args) {
        Printable2 printer = new Printable2() {
            public void print(String message) {
                System.out.println(message);
            }
```

```
        }; // this is a semicolon

        printer.print("Beach Music");
    }
}
```

Now that you know what the anonymous inner class is, you are ready to learn about lambda expressions.

10.2 Why Lambda Expressions?

Also known as closures, lambda expressions can make certain constructs shorter and more readable, especially when you are dealing with inner classes.

Consider the following code snippet that defines an anonymous inner class out of the **java.lang.Runnable** interface and instantiates the class.

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running...");
    }
}
```

The code can be replaced with a lambda expression as short as this:

```
Runnable runnable = () -> System.out.println("Running...")
```

In other words, if you need to pass a **Runnable** to a **java.util.concurrent.Executor** like so

```
executor.execute(new Runnable() {
    @Override
    public void run() {
        System.out.println("Running...");
    }
});
```

you can use a lambda expression to produce code with the same effect:

```
executor.execute(() -> System.out.println("Running..."));
```

Short and sweet. And clearer and more expressive too.

10.3 Functional Interfaces

Before I explain the lambda expression further, I will introduce the functional interface. A functional interface is an interface that has exactly one abstract method that does not override a method in **java.lang.Object**. A functional interface is also called a single abstract method (SAM) interface. For example, **java.lang.Runnable** is a functional interface because it has only one abstract method, **run**. A functional interface may have any number of default and static methods and methods that override public methods in **java.lang.Object** and still qualifies as a functional interface. For example, the **Calculator** interface in Listing 10.3 is a functional interface with a single abstract method called **calculate**. It is a functional interface even though it has two default methods and another abstract method overriding the **toString** method from **java.lang.Object**.

Listing 10.3: A functional interface

```
package app10;
public interface Calculator {

    double calculate(int a, int b);

    public default int subtract(int a, int b) {
        return a - b;
    }

    public default int add(int a, int b) {
        return a * b;
    }

    @Override
    public String toString();
}
```

Examples of functional interfaces in the core Java library include **java.lang.Runnable, java.lang.AutoCloseable, java.lang.Comparable** and **java.util.Comparator**. In addition, the new package **java.util.function** contains dozens of functional interfaces and is discussed in the section "Predefined Functional Interfaces" later in this chapter.

Optionally, a functional interface can be annotated with **@FunctionalInterface**.

Why is the functional interface important? Because you can use a lambda expression to create the equivalent of an anonymous inner class from a functional interface. You cannot use an interface that is not a functional interface for this purpose.

So, let the fun begin.

10.4 Lambda Expression Syntax

The **Calculator** interface in Listing 10.3 has a **calculate** method that can be the basis of a lambda expression. The method allows you to define any mathematical operation that takes two integers and return a double. For instance, here are two lambda expressions based on **Calculator**.

```
Calculator addition = (int a, int b) -> (a + b);
System.out.println(addition.calculate(5, 20)); // prints 25.0

Calculator division = (int a, int b) -> (double) a / b;
System.out.println(division.calculate(5, 2)); // prints 2.5
```

The lambda expression is such an elegant design. How many more lines of code would you need to implement the same program without lambda expressions?

Now that you have acquainted yourself with the lambda expression, I will show you its formal syntax.

```
(parameter list) -> expression
```

or

```
(parameter list) -> {
    statements
}
```

The parameter list is the same as the list of parameters of the abstract method in the underlying functional interface. However, the type for each parameter is optional. In other words, both of these expressions are valid.

```
Calculator addition = (int a, int b) -> (a + b);

Calculator addition = (a, b) -> (a + b);
```

To summarize, a lambda expression is a shortcut to defining an implementation of a functional interface. A lambda expression is equivalent to an instance of a functional interface implementation. Since it is possible to pass objects as parameters to a method, it is too possible to pass lambda expressions as parameters to a method.

10.5 Predefined Functional Interfaces

The **java.util.function** package is a new package in JDK 8. It contains more than forty predefined functional interfaces that can make it easier for you to write lambda expressions. Some of the predefined functional interfaces are shown in Table 10.1.

Table 10.1: Core functional interfaces

Functional Interface	Description
Function	Models a function that can take one parameter and return a result. The result can be of a different type than the parameter.
BiFunction	Models a function that can take two parameters and return a result. The result can be of a different type than any of the parameters.
UnaryOperator	Represents an operation on a single operand that returns a result whose type is the same as the type of the operand. A **UnaryOperator** can be thought of as a **Function** whose return value is of the same type as the parameter. In fact, **UnaryOperator** is a subinterface of **Function**.
BiOperator	Represents an operation on two operands. The result and the operands must be of the same type.
Predicate	A Function that takes a parameter and returns **true** or **false** based on the value of the parameter.

Table 10.1: Core functional interfaces

Functional Interface	Description
Supplier	Represents a supplier of results.
Consumer	An operation that takes a parameter and returns no result.

Function, BiFunction and Other Variants

The **Function** interface is used to create a one-argument function that returns a result. It is a parameterized type and here is its definition.

```
public interface Function<T, R>
```

Here, T represents the type of the argument and R the type of the result.

Function has one abstract method, **apply**, whose signature is as follows.

```
R apply(T argument)
```

This is the method you need to override when using **Function**. For example, the class in Listing 10.4 defines a **Function** for converting miles to kilometers. The **Function** takes an Integer as an argument and returns a **Double**.

Listing 10.4: The FunctionDemo1 class

```
package app10.function;
import java.util.function.Function;

public class FunctionDemo1 {
    public static void main(String[] args) {
        Function<Integer, Double> milesToKms =
                (input) -> 1.6 * input;
        int miles = 3;
        double kms = milesToKms.apply(miles);
        System.out.printf("%d miles = %3.2f kilometers\n",
                miles, kms);
    }
}
```

If you run the **FunctionDemo1** class, you will see this on your console.

```
3 miles = 4.80 kilometers
```

A variant of **Function, BiFunction** takes two arguments and returns a result. Listing 10.5 shows an example of **BiFunction**. It uses **BiFunction** to create a function that calculates an area given a width and a length. Invoking the function is done by calling its **apply** method.

Listing 10.5: The BiFunctionDemo1 class

```
package app10.function;
import java.util.function.BiFunction;

public class BiFunctionDemo1 {
    public static void main(String[] args) {
        BiFunction<Float, Float, Float> area =
                (width, length) -> width * length;
        float width = 7.0F;
        float length = 10.0F;
        float result = area.apply(width, length);
        System.out.println(result);
    }
}
```

Running the **BiFunctionDemo1** class prints the following on the console.

```
70.0
```

In addition to **BiFunction**, there are also variants that are specializations of **Function**. For example, the **IntFunction** interface

always takes an **Integer** and requires only one parameterized type for the result type. Its **apply** method returns an **int**.

```
R apply(int input)
```

The **LongFunction** and **DoubleFunction** interfaces are similar to **IntFunction**, except they take a long and a double as an argument, respectively.

Then, there are variants that do not require parameterized arguments at all, because they have been designed for a specific argument type and a specific return type. For instance, the **IntToDoubleFunction** interface can be used to create a function that accepts an int and returns a double. Instead of **apply**, the interface offers an **applyAsDouble** method. An example of **IntToDoubleFunction** interface is given in Listing 10.6. It is a function that converts a temperature on the Celcius scale to Fahrenheit.

Listing 10.6: The IntToDoubleFunctionDemo1 class

```
package app10.function;
import java.util.function.IntToDoubleFunction;

public class IntToDoubleFunctionDemo1 {
    public static void main(String[] args) {
        IntToDoubleFunction celciusToFahrenheit =
                (input) -> 1.8 * input + 32;
        int celcius = 100;
        double fahrenheit =
                celciusToFahrenheit.applyAsDouble(celcius);
        System.out.println(celcius + "\u2103" + " = "
                + fahrenheit + "\u2109\n");
    }
}
```

This is the output of **IntToDoubleFunctionDemo1**.

```
100°C = 212.0°F
```

Similar to **IntToDoubleFunction** are **LongToDoubleFunction** and **LongToIntFunction**. I am sure you can guess what they do from their names.

The **UnaryOperator** interface is another specialization of **Function** whose operand type is the same as the return type. Its declaration is as follows.

public interfaceUnaryOperator<T> extends Function<T,T>**BinaryOperator** is a specialization of **BiFunction. BinaryOperator** represents an operation with two operands of the same type and returns a result that has the same type as the operands.

Predicate

A **Predicate** is a function that takes a parameter and returns **true** or **false** based on the value of the parameter. It has a single abstract method called **test**.

For example, the **PredicateDemo1** class in Listing 10.7 defines a **Predicate** that evaluates a string input and returns true if every character in the string is a number.

Listing 10.7: The PredicateDemo1 class

```
package app10.function;
import java.util.function.Predicate;

public class PredicateDemo1 {
    public static void main(String[] args) {
        Predicate<String> numbersOnly = (input) -> {
            for (int i = 0; i < input.length(); i++) {
                char c = input.charAt(i);
                if ("0123456789".indexOf(c) == -1) {
                    return false;
                }
            }
            return true;
        };
```

```
        System.out.println(numbersOnly.test("12345"));// true
        System.out.println(numbersOnly.test("100a")); // false
    }
}
```

Supplier

A **Supplier** takes no parameter and returns a value. Implementations must override its **get** abstract method and returns an instance of the interface's type parameter.

Listing 10.8 shows an example of **Supplier**. It defines a **Supplier** that returns a one-digit random number and uses a **for** loop to print five random numbers.

Listing 10.8: The SupplierDemo1 class

```
package app10.function;
import java.util.Random;
import java.util.function.Supplier;

public class SupplierDemo1 {
    public static void main(String[] args) {
        Supplier<Integer> oneDigitRandom = () -> {
            Random random = new Random();
            return random.nextInt(10);
        };
        for (int i = 0; i < 5; i++) {
            System.out.println(
                    oneDigitRandom.get());
        }
    }
}
```

There are also specialized variants of **Supplier**, such as **DoubleSupplier** (returns a **Double), IntSupplier** and **LongSupplier**.

Consumer

A **Consumer** is an operation that returns no result. It has one abstract method called **accept**.

Listing 10.9 shows an example of **Consumer** that takes a string and print it center-justified.

Listing 10.9: Consumer example

```
package app10.function;
import java.util.function.Consumer;
import java.util.function.Function;

public class ConsumerDemo1 {
    public static void main(String[] args) {
        Function>Integer, String> spacer = (count) -> {
            StringBuilder sb = new StringBuilder(count);
            for (int i = 0; i < count; i++) {
                sb.append(" ");
            }
            return sb.toString();
        };

        int lineLength = 60; // characters
        Consumer<String> printCentered =
                (input) -> {
                    int length = input.length();
                    String spaces = spacer.apply(
                            (lineLength - length) / 2);
                    System.out.println(spaces + input);
                };

        printCentered.accept("A lambda expression a day");
        printCentered.accept("makes you");
        printCentered.accept("look smarter");
    }
```

```
}
```

The example in Listing 10.9 features a **Consumer** that takes a **String** and prints it after prefixing it with a certain number of spaces. The maximum number of characters in each line is 60 and the spaces are obtained by calling a **Function** named spacer. The implementation of the Consumer's **accept** method is given by this Lambda expression.

```
(input) -> {
    int length = input.length();
    String spaces = spacer.apply(
            (lineLength - length) / 2);
    System.out.println(spaces + input);
}
```

The function **spacer** returns the specified number of spaces and is defined as

```
Function<Integer, String> spacer = (count) -> {
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append(" ");
    }
    return sb.toString();
};
```

The function employs a **for** loop that appends a space to a **StringBuilder** *count* number of times, where *count* is the parameter to the function. When the loop exits, it returns the **String** representation of the **StringBuilder**.

If you run the **ConsumerDemo1** class, you will see this on your console.

```
A lambda expression a day
        makes you
       look smarter
```

Self Test

1 Question ?

Consider this interface:

```
interface Computable {
    double compute(double a, double b);
}
```

Which of the following lambda expressions can be created out of the **Computable** interface?

A. Computable multiply = (x, y) -> x * y;

B. Computable div = (double x, double y) -> x / y;

C. Computable add = (int x, int y) -> x / y;

D. Computable add = (a, b) => x / y;

2 Question ?

Consider this interface:

```
interface Formatter {
    String format(double a, double b);
    String format(int a, int b);
}
```

Which of the following lambda expressions can be created out of the **Formatter** interface?

A. Formatter f1 = (x, y) -> { return x + "," + y };

B. Formatter f2 = (double x, double y) -> x + "" + y;

C. Formatter f3 = (int x, int y) -> x + y;

D. Formatter cannot be used to construct a lambda expression.

3 Question ?

Which of the following statements are true? (Choose all that apply)

A. A functional interface may only have one method

B. A functional interface may have more than one method

C. A functional interface may not have default methods

D. A functional interface can be used as the assignment target for a lambda expression

4 Question ?

Which of the following are predefined functional interfaces? (Choose all that apply)

A. Function

B. BiFunction

C. TriFunction

D. UnaryFunction

5 Question ?

Which of the following statements are true? (Choose all that apply)

A. A **Predicate** is a function that takes a parameter and returns true or false

B. **Predicate** is a predefined functional interface

C. The **Predicate** interface has a single abstract method called **predicate**

D. The **Predicate** interface has a single abstract method called **test**

6 Question ?

Consider this lambda expression that uses **Predicate:**

```
Predicate<String> cap = s -> s.charAt(0) == s.toUpperCase().charAt(0);
```

Which of the following expressions return true? (Choose all that apply)

A. cap.test("Abc")

B. cap.test("abc")

C. cap.test("ABC")

D. cap.test("aBC")

7 Question ?

Which of the following statements are true with regard to **Supplier?**

A. Supplier is a predefined functional interface

B. Supplier has an abstract method called supply

C. Supplier has an abstract method called test

D. Supplier takes two arguments

8 Question ?

Consider the following lambda expression:

```
Supplier<String> supplier = () -> "ABC";
```

Which of the following statements are correct? (Choose all that apply)

A. System.out.print(supplier.get()) prints "ABC"

B. System.out.print(supplier) also prints "ABC"

C. System.out.print(supplier.getClass().getName()) prints "java.lang.String"

D. System.out.print(supplier.getClass().getName()) prints "java.util.function.Supplier"

9 Question ?

Consider the following code:

```
package test;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class Tester {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        Consumer<String> consumer = (s) -> {
            if (s.length() % 2 == 0)
                list.add(s.toUpperCase());
        };
        consumer.accept("hello");
        consumer.accept("world!");
        for (String s : list) {
            System.out.print(s);
        }
    }
}
```

What is printed on the console?

A. WORLD!

B. helloworld!

C. HELLOWORLD!

D. an empty string

10 Question ?

Consider the following lambda expression that uses a **BiPredicate:**

```
BiPredicate<Double, Integer> pr = (x, y) -> x > y;
```

Which of the following statements is/are true?

A. pr must be called with two arguments.

B. Passing 0 and 0 to pr.test returns false

C. Passing 10 and 0 to pr.test returns true

D. Passing 10 and 0 to pr.test returns false

Answers

1 **A, B**.

A and B are correct. C is incorrect because the method **compute** expects two **double** arguments, not **int**s. D is incorrect because -> is the operator for lambda expressions, not =>.

2 **D**.

Formatter is not a functional interface because it has more than one abstract method. As such, it cannot be used as the assignment target for a lambda expression.

3 **B, D**.

A is incorrect because a functional interface may have multiple methods as long as only one of them is abstract.

B is correct because a functional interface may have more than one method as long as one of them is abstract.

C is incorrect because a functional interface may have default methods.

D is correct because a function interface can be used as the assignment target for a lambda expression.

4 **A, B**.

Function and **BiFunction** are predefined functional interfaces. **TriFunction** and **UnaryFunction** are not.

5 **A, B, D**.

The **Predicate** interface is one of the predefined functional interfaces. It has an abstract method named **test** that returns true or false depending on the given argument.

6 **A, C**.

The lambda expression tests if the first character of a **String** is capitalized. Therefore, A and C return true.

7 **A**.

Supplier is a predefined functional interface with an abstract method called **get**. The method takes no argument.

8 **A**.

A is correct because **Supplier.get()** returns the object being supplied.

B is incorrect and it will print the **Supplier** and not what is being supplied.

C and D are incorrect because **supplier** is a lambda expression and not an instance of **Supplier**. In fact, the statement prints a String having this format:

```
nameOfContainingClass$$Lambda$someId
```

9 **A**.

The Consumer accepts a String and adds it the uppercase of it to the List if the number of characters in the String is even.

10 **A**.

A is correct because a **BiPredicate** takes two arguments.

B, C and D cause compile errors as passing two ints to a method that takes a **Double** and an **Integer** is illegal.