CHAPTER 5

■ ■ ■

Lambda Expressions

QUESTIONS AND EXERCISES

1. What are lambda expressions and how are they related to functional interfaces?

Answer:

A lambda expression is an unnamed block of code (or an unnamed function) with a list of formal parameters and a body. It is an expression that represents an instance of a functional interface.

2. How does a lambda expression differ from an anonymous class? Can you always replace a lambda expression with an anonymous class and vice versa?

Answer:

Anonymous classes use bulky syntax, whereas lambda expressions use a very concise syntax. Like anonymous classes, lambda expressions cannot implement abstract class methods or more than one method of an interface. A lambda expression in Java is like a higher-order function in functional programming. Neither one can always replace other completely.

3. Are the following two lambda expressions different?

   a. `(int x, int y) -> { return x + y; }`

   b. `(int x, int y) -> x + y`

Answer:

No. These two lambda expressions are the same. One uses a block statement and the other an expression.

4. If someone shows you the following lambda expressions, explain the possible functions they may represent.

   a. `(int x, int y) -> x + y`

   b. `(x, y) -> x + y`

   c. `(String msg) -> { System.out.println(msg); }`

   d. `() -> {}`

   Answer:

   a) The function takes two `int` parameters and returns an `int`.

   b) It's an implicit lambda expression. Unless you have the target for this lambda expression, it is not possible to know the exact function it represents. For example, it might represent a function with two `int` formal parameters with `int` return type, a function with two String formal parameters with a `String` return type, etc. There are many other possibilities.

   c) The function takes a `String` formal parameter and returns `void`.

   d) The function takes no formal parameters and returns `void`.

5. What kind of function the following lambda expression may represent?

   ```
   x -> x;
   ```

   Answer:

   A function represented by this lambda expression is called an identity function. The function takes a parameter and returns the same. Given this lambda expression, you cannot tell the function signature unless you have a target for this lambda expression.

6. Will the following declaration of a `MathUtil` interface compile? Explain your answer.

   ```
   @FunctionalInterface
   public interface MathUtil {
       int factorial(int n);
       int abs(int n);
   }
   ```

   Answer:

   No. The `MathUtil` interface won't compile. A functional Interface is an interface that has exactly one abstract method. Compilation fails because the interface has been annotated with an `@FunctionalInterface` annotation and it declares two abstract methods. To make it compile, you will need to remove the `@FunctionalInterface` annotation.

7. Will the following statement compile? Explain your answer.

```
Object obj = x -> x + 1;
```

Answer:

The target of a lambda expression must be a functional interface. In this statement, the target is the `Object` type, which is not a functional interface. The statement will not compile.

8. Will the following statements compile? Explain your answer.

```
Function<Integer,Integer> f = x -> x + 1;
Object obj = f;
```

Answer:

Yes. In the first statement, an instance of a lambda expression is created and assigned to variable `f`. Since an instance of all interfaces is an `Object` in Java, the second statement is fine.

9. What will be the output when you run the following `Scope` class?

```
// Scope.java
package com.jdojo.lambda.exercises;

import java.util.function.Function;

public class Scope {
    private static long n = 100;
    private static Function<Long,Long> f = n -> n + 1;

    public static void main(String[] args) {
        System.out.println(n);
        System.out.println(f.apply(n));
        System.out.println(n);
    }
}
```

Answer:

```
100
101
100
```

This question has a trick. Intentionally, I named the static variable and the formal parameter of the lambda expression the same (`n`). However, they have nothing to do with each other. Mention of `n` inside the body of the lambda expression refers to the formal parameter `n`, not the static variable `n`. Inside the body of the lambda expression, both n (static variable and formal parameter) are in scope. However, the formal parameter takes precedence. If you want to test this, you can try the following declarations of the two static variables:

```
public class Scope {
    private static long n = 100;
    private static Function<Long, Long> f = m -> n++;
    // Other code goes here
}
```

This time, in `n++`, `n` refers to the static variable `n`. Now, every time you call `f.apply()`, the value of the static variable `n` will be increment by 1.

The first time, the value of the static variable `n` is printed, which is 100.

The second time, the value of the static variable `n` is passed to the `apply()` method, which performs a computation (`n + 1`) and returns 101. The second statement prints 101. Note that calling the `f.apply(n)` method does change the value of the static variable `n`. After the second statement is executed, the value of the static variable n is still 100.

The third statement, prints the current value of the static variable `n`, which is 100.

10. Why does the following method declaration not compile?

```
public static void test() {
    int n = 100;
    Function<Integer,Integer> f = n -> n + 1;
    System.out.println(f.apply(100));
}
```

Answer:

The compilation fails because the `test()` method defines two local variables with the same name `n` – one as an `int` in the first statement and another as the formal parameter of the lambda expression.

11. What will be the output when the following `Capture` class is run?

```
// Capture.java
package com.jdojo.lambda.exercises;

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

```
public class Capture {
    public static void main(String[] args) {
        test();
        test();
    }

    public static void test() {
        int n = 100;
        Function<Integer,Integer> f = x -> n + 1;
        System.out.println(f.apply(100));
    }
}
```

Answer:

```
101
102
```

12. Assume that there is a `Person` class, which contains four constructors. One of the constructors is a no-args constructor. Given a constructor reference, `Person::new`, can you tell which constructor of the `Person` it refers to?

Answer:

No. The target type, which is always a functional interface, determines the method's (or constructor's) details. Since `Person` class' constructor is overloaded, the compiler will choose the most specific constructor based on the target type.

13. Will the following declaration of the `FeelingLucky` interface compile? Notice that it has been annotated with `@FunctionalInterface`.

```
@FunctionalInterface
public interface FeelingLucky {
    void gamble();

    public static void hitJackpot() {
        System.out.println("You have won 80M dollars.");
    }
}
```

Answer:

Yes. `FeelingLucky` is a functional interface because it contains only one abstract method named `gamble()`. The `hitJackpot()` method is declared `static` and it does not count against the one abstract method rule of a functional interface. The code compiles fine.

14. Why does the following declaration of the `Mystery` interface not compile?

```
@FunctionalInterface
public interface Mystery {
    @Override
    String toString();
}
```

Answer:

The `Mystery` interface has been annotated with an `@FunctionalInterface` annotation, which requires it to have one and only one abstract method of its own. Having an interface override a public method of the `Object` class and declaring it abstract does not count against the functional interface rule of having an abstract method. In this case, the `Mystery` interface override the `toString()` public method of the `Object` class and makes it abstract. Thus the compilation fails with a message – no abstract method found.

15. What will be the output when the following `PredicateTest` class is run?

```
// PredicateTest.java
package com.jdojo.lambda.exercises;

import java.util.function.Predicate;

public class PredicateTest {
   public static void main(String[] args) {
       int[] nums = {1, 2, 3, 4, 5};
       filterThenPrint(nums, n -> n%2 == 0);
       filterThenPrint(nums, n -> n%2 == 1);
   }

   static void filterThenPrint(int[] nums, Predicate<Integer> p) {
       for(int x : nums) {
           if(p.test(x)) {
               System.out.println(x);
           }
       }
   }
}
```

Answer:

```
2
4
1
3
5
```

16. What will be the output when the following `SupplierTest` class is run? Explain your answer.

```
// SupplierTest.java
package com.jdojo.lambda.exercises;

import java.util.function.Supplier;
public class SupplierTest {
    public static void main(String[] args) {
        Supplier<Integer> supplier = () -> {
            int counter = 0;
            return ++counter;
        };

        System.out.println(supplier.get());
        System.out.println(supplier.get());
    }
}
```

Answer:

```
1
1
```

17. What will be the output when the following `ConsumerTest` class is run?

```
// ConsumerTest.java
package com.jdojo.lambda.exercises;

import java.util.function.Consumer;

public class ConsumerTest {
    public static void main(String[] args) {
         Consumer<String> c1 = System.out::println;
         Consumer<String> c2 = s -> {};

         consume(c1, "Hello");
         consume(c2, "Hello");
    }

    static <T> void consume(Consumer<T> consumer, T item) {
        consumer.accept(item);
    }
}
```

Answer:

```
Hello
```