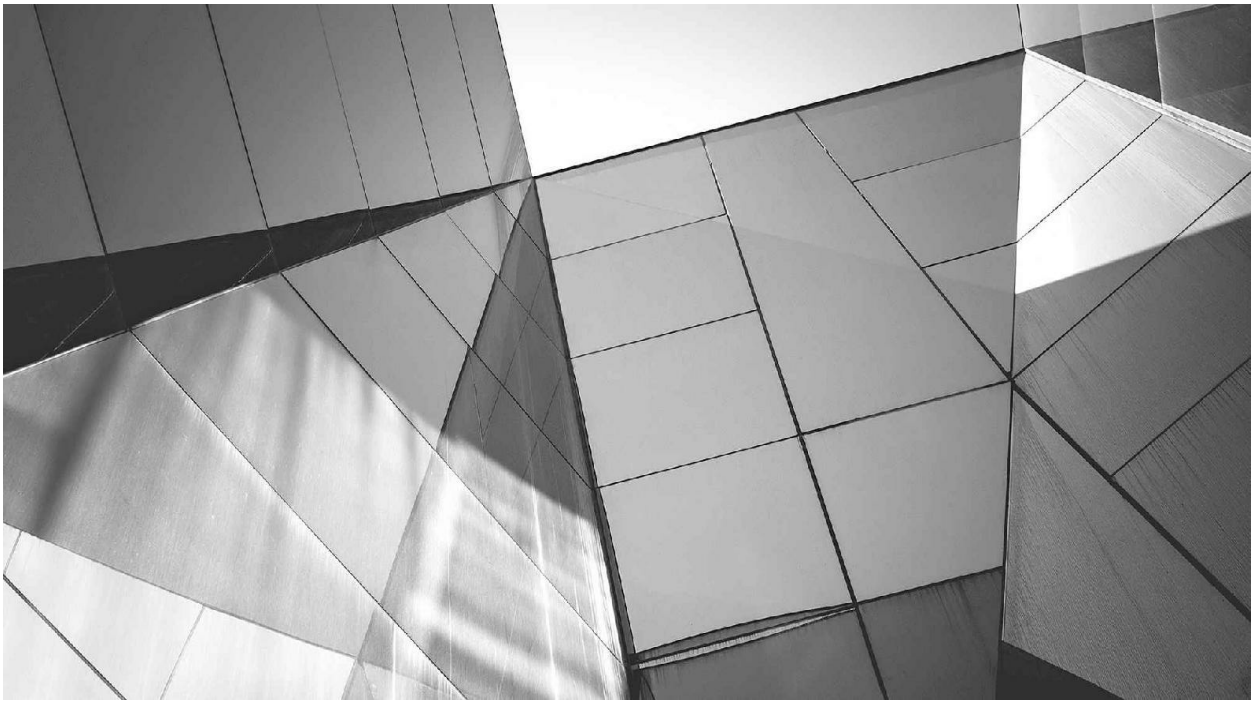


8

Lambda Expressions and Functional Interfaces

CERTIFICATION OBJECTIVES

- Create and Use Lambda Expressions
- Iterate Using forEach Methods of Streams and List
- Use Method References with Streams
- Use the Built-in Interfaces Included in the java.util.function Package such as Predicate, Consumer, Function, and Supplier
- Develop Code That Uses Primitive Versions of Functional Interfaces
- Develop Code That Uses Binary Versions of Functional Interfaces
- Develop Code That Uses the UnaryOperator Interface

✓ Two-Minute Drill

Q&A Self Test

One of the big new language features added in Java 8 is the lambda expression. We've already talked a bit about lambdas, introducing the basic syntax and the idea of functional interfaces. You've seen how to use a lambda expression to replace an inner class (e.g., a `Comparator`) and you've also seen how to pass a lambda expression—really, just a block of code—to methods that expect Comparators, like `Collections.sort()`. Those earlier tastes of lambda expressions probably left you with more questions than answers, and this chapter is where we dive into all those details. As with inner classes, you'll need to know the syntax of lambda expressions and

when you can use them like the back of your hand. You'll also need to get familiar with a variety of functional interfaces so you can recognize them easily. Do that and you'll sail through the lambda expressions part of the exam.

CERTIFICATION OBJECTIVE

Lambda Expression Syntax (OCP Objective 2.6)

2.6 Create and use Lambda expressions.

You already know (a little) how lambda expressions can make your code more concise. Concise code is good, but only if it makes sense. And, let's face it, lambda expression syntax can take some getting used to. Let's take another look at the syntax and talk about the varieties of lambda expression syntax you might expect to see in the real world and on the exam.

Imagine you've got a super simple interface, `DogQuerier`, with one abstract method, `test()` (remember that all interface methods are abstract unless they are declared default or static):

```
interface DogQuerier {
   public boolean test(Dog d);
}
```

We say that `DogQuerier` is a "functional interface": it's an interface with one abstract method.

Later, you use an inner class to define an instance of a class that implements this interface:

```
DogQuerier dq = new DogQuerier() {
   public boolean test(Dog d) { return d.getAge() > 9; }
};
```

Wait, scrap that—we can make the code much more concise by replacing that inner class with a lambda expression:

```
DogQuerier dq = (d) -> d.getAge() > 9;
```

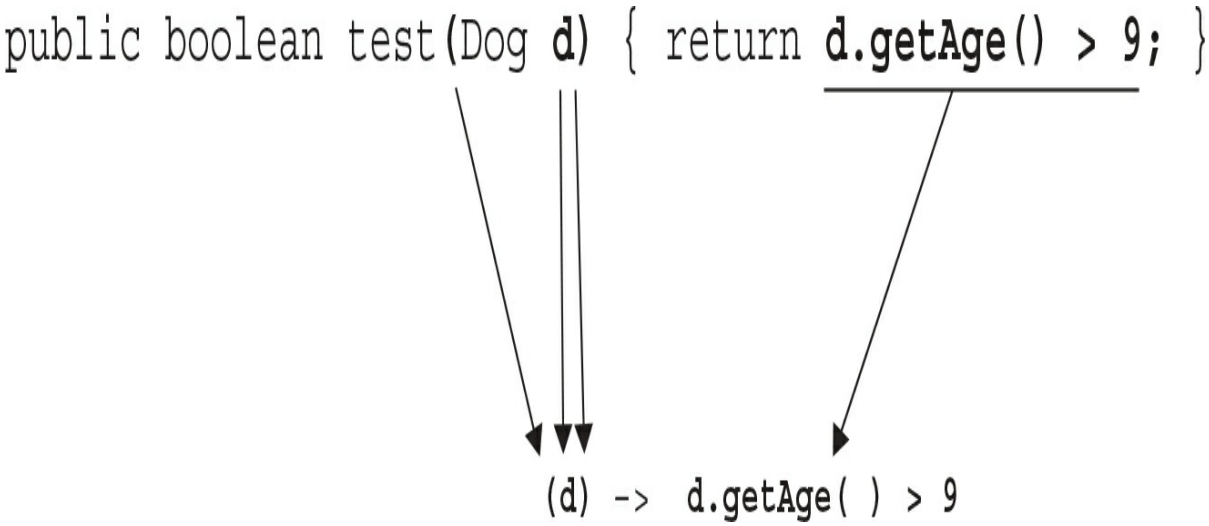
You use that lambda expression just like you'd use the instance of the inner class:

```
System.out.println("Is Boi older than 9? " + dq.test(boi));
```

(assuming boi is a `Dog` with a `getAge()` method, of course).

Think of a lambda expression as a shorthand way of writing an instance of a class that implements a functional interface. It looks a lot like a method (in fact, some call lambda expressions "anonymous methods"), but it's a bit more than that; it's more like an instance with everything but the method stripped away. The important part of the instance is the method (the rest can be inferred from the interface definition), so the lambda expression is the syntax of the instance that's been boiled down to the bare essentials.

To make the lambda expression, we copy the parameter of the `test()` method from the inner class, then write an arrow, and then copy the expression in the body of the test method, leaving out the `return` and the semicolon:



The type of the lambda expression is `DogQuerier`. That's the same as the type of the interface and the same as the type of the instance being created by

the inner class.

```
DogQuerier dq = (d) -> d.getAge() > 9;
```

And, of course, when we write the statement assigning the lambda to the instance variable, we end that statement with a semicolon. Now let's look at some variations. Because the `test()` method has only one parameter, it's perfectly legal to leave off the parentheses around the parameter and write the lambda like this instead:

```
DogQuerier dq = d -> d.getAge() > 9;
```

If you have more than one parameter, however, you *must* use the parentheses.

You might be wondering: how does the lambda "know" what `d`'s type is supposed to be? That can be inferred from the `DogQuerier` interface definition. However, there may be times when the type can't be inferred and you will need to write it in. And, in this example, you can supply the type if you want to, but if you do, you'll have to use the parentheses around the parameter:

```
DogQuerier dq = (Dog d) -> d.getAge() > 9;
```

What about the return value's type? That, too, can be inferred from the `DogQuerier` interface definition. And wait a sec, where'd that `return` go to anyway?

The rule is, if there is only one expression in the lambda, then the value of that expression gets returned by default, and you don't need a return. In fact, if you try to write

```
DogQuerier dq = d -> return d.getAge() > 9;  // does not compile
```

you'll get a compile-time error:

```
Syntax error on token "->", { expected after this token.
```

If you want to write `return`, then you'll have to write the lambda like this:

```
DogQuerier dq = d -> { return d.getAge() > 9; };
```

In other words, if the body of your lambda is anything more than an expression—that is, a statement or multiple statements—you'll need to use the curly braces. Here's an example of a lambda expression with multiple statements:

```
DogQuerier dq = d -> {
   System.out.println("Testing " + d.getName());
   return d.getAge() > 9;
};
```

To summarize, we can write the original `DogQuerier` instance as a lambda expression in the following ways, all of which are equivalent:

```
DogQuerier dq = (d) -> d.getAge() > 9;
DogQuerier dq = d -> d.getAge() > 9;
DogQuerier dq = (Dog d) -> d.getAge() > 9;
DogQuerier dq = d -> { return d.getAge() > 9; };
```

Here's the full code so you can test the `DogQuerier` lambda expression:

```
// Our trusty Dog class
public class Dog {
   private String name;
   private int weight;
   private int age;
   public Dog(String name, int weight, int age) {
```

```
        this.name = name;
        this.weight = weight;
        this.age = age;
    }
    public String getName() { return this.name; }
    public int getWeight() { return this.weight; }
    public int getAge() { return this.age; }
    public String toString() { return this.name; }
}
// Our functional interface to test dogs
interface DogQuerier {
    public boolean test(Dog d);
}

public class TestDogs {
    public static void main(String[] args) {
        Dog boi = new Dog("boi", 30, 6);
        Dog clover = new Dog("clover", 35, 12);
        // We don't need this inner class anymore; replace with a lambda
// DogQuerier dq = new DogQuerier() {
//     public boolean test(Dog d) { return d.getAge() > 9; }
// };
DogQuerier dq = d -> d.getAge() > 9;  // replaces the inner class
System.out.println("Is Boi older than 9? " + dq.test(boi));
System.out.println("Is Clover older than 9? " + dq.test(clover));
    }
}
```

The output is

```
Is Boi older than 9? false
Is Clover older than 9? true
```

exam Watch

Here are a few examples of invalid lambda expression syntax to watch out for:

```
Sheep s -> s.color.equals(color);
```

Needs parentheses around the argument.

```
(Sheep s) -> if (s.color.equals(color)) return true; else return false;
```

Needs { } around body of lambda and an ending ;

```
SheepQuerier testSheep = s, n -> s.age > n;
```

Needs parentheses around the arguments.

Passing Lambda Expressions to Methods

Lambda expressions are easy to pass to methods. It's a bit like passing a block of code to a method. To demonstrate, let's add a class to our `Dogs` example:

```
class DogsPlay {
   DogQuerier dogQuerier;
   public DogsPlay(DogQuerier dogQuerier) {
      this.dogQuerier = dogQuerier;
   }
   public boolean doQuery(Dog d) {
      return dogQuerier.test(d);
   }
}
```

The constructor for the `DogsPlay` class takes a `DogQuerier` instance. We can pass the `dq` instance we created with a lambda expression to `DogsPlay` like this:

```
DogsPlay dp = new DogsPlay(dq);
```

Or we can pass a lambda expression directly:

```
DogsPlay dp = new DogsPlay(d -> d.getAge() > 9);
```

When we call `dp.doQuery()` and pass in a dog, the `test()` method of the `DogQuerier` gets called:

```
System.out.println("Is Boi older than 9? " + dp.doQuery(boi));
```

And we see the output:

```
Is Boi older than 9? false
```

The lambda expression we're passing to `DogsPlay` is simple; it has one `Dog` parameter and simply tests that dog's age and returns true or false.

Accessing Variables from Lambda Expressions

What do you think happens if a lambda has a reference to another variable? While you can declare and use variables within the lambda expression, just like you would in the body of a method, the lambda is essentially just creating a nested block, so you can't use the same variable name as you've used in the enclosing scope. So this is fine:

```
int numCats = 3;
DogQuerier dqWithCats = d -> {
   int numBalls = 1;       // completely new variable local to lambda
   numBalls++;             // can modify numBalls
   System.out.println("Number of balls: " + numBalls);  // can access numBalls
   System.out.println("Number of cats: " + numCats);    // can access numCats
   return d.getAge() > 9;
};
```

But this is not:

```
int numCats = 3;
int numBalls = 1;          // now we have numBalls in enclosing scope
DogQuerier dqWithCats = d -> {
   int numBalls = 5;       // won't compile! Trying to redeclare numBalls
   System.out.println("Number of balls: " + numBalls);
   System.out.println("Number of cats: " + numCats);
   return d.getAge() > 9;
};
```

A lambda expression "captures" variables from the enclosing scope, so you can access those variables in the body of the lambda, but those variables must be final or effectively final. An effectively final variable is a variable or parameter whose value isn't changed after it is initialized. Let's see what happens if we try to modify the number of cats in the lambda expression:

```
int numCats = 3;
DogQuerier dqWithCats = d -> {
   int numBalls = 1;
   numBalls++;
   numCats++;          // Won't compile! Can't change numCats
   System.out.println("Number of balls: " + numBalls);
   System.out.println("Number of cats: " + numCats);
   return d.getAge() > 9;
};
```

If we try to change the value either in the lambda itself or elsewhere in the enclosing scope, we will get an error. We can *use* the value of `numCats`, but we can't *change* it. So this will work:

```
int numCats = 3;            // numCats is effectively final
DogQuerier dqWithCats = d -> {
   int numBalls = 1;
   numBalls++;
   System.out.println("Number of balls: " + numBalls);
   System.out.println("Number of cats: " + numCats);  // Okay to use numCats
   return d.getAge() > 9;
};
```

The value of the variable `numCats` is captured by the lambda so it can be used later when we invoke the lambda (by calling its `test()` method). Let's see what happens when we pass a lambda with captured values to the `DogsPlay` constructor:

```
System.out.println("--- use DogsPlay ---");
DogsPlay dp = new DogsPlay(dqWithCats);
System.out.println("Is Clover older than 9? " + dp.doQuery(clover));
```

This works fine; the captured value for `numCats` is sent along with the lambda to `DogsPlay` and used when the lambda is invoked later when we call `doQuery()`. Here is the output:

```
--- use DogsPlay ---
Number of balls: 2
Number of cats: 3
Is Clover older than 9? True
```

CERTIFICATION OBJECTIVE

Functional Interfaces (OCP Objectives 3.5, 4.1, 4.2, 4.3, and 4.4)

3.5 Iterate using forEach methods of Streams and List.
4.1 Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier.
4.2 Develop code that uses primitive versions of functional interfaces.
4.3 Develop code that uses binary versions of functional interfaces.
4.4 Develop code that uses the UnaryOperator interface.

Lambda expressions work by standing in for instances of classes that implement interfaces with one abstract method, so there is no confusion about which method the lambda is defining. As we've said before, we call these interfaces with one and only one abstract method "functional interfaces." There's nothing particularly special about them, except their relationship to lambda expressions. We can explicitly identify a functional interface using the `@FunctionalInterface` annotation, like this:

```
@FunctionalInterface
interface DogQuerier {
   public boolean test(Dog d);
}
```

This annotation is not required but can be helpful when you want to ensure you don't inadvertently add methods to a functional interface that will then break other code. If you use the `DogQuerier` type to define a lambda expression and then later add another abstract method to this interface, without the `@FunctionalInterface` annotation, you'll get a compiler error, "The target type of this expression must be a functional interface." Yikes!

By using the `@FunctionalInterface`, the compiler will warn you that adding an extra method won't work and you can avoid the error. Seeing that annotation, you'll be reminded that you created a functional interface for a reason, so you'll know not to change it. You may not see `@FunctionalInterface` used on the exam, but you'll definitely be expected to identify functional interfaces with or without this annotation.

Built-in Functional Interfaces

Functional interfaces turn out to be useful in all sorts of scenarios, some of which we'll get into here and some in the next chapter. You've already seen one useful functional interface: the `Comparator`. In addition, with Java 8 we get a big collection of functional interfaces in the `java.util.function` package, where you'll find `Predicate`, which you've also seen before, along with a variety of others. You will need to be familiar with them all.

Looking at the list, you might be wondering why you need all these different kinds of functional interfaces? Well, you probably won't ever need them all, but it's nice to have them available to use when you need a functional interface and don't want to define your own. As you'll see, there are many more uses for these interfaces than you might think, and having those interfaces already defined can save you time and code when you just want a quick lambda expression and don't want to bother with creating your own functional interface.

What Makes an Interface Functional?

We already said that a functional interface is an interface with one and only one abstract method. Let's delve into this just a little more because it can get a bit tricky.

Take a look at the Java 8 API documentation (https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) for `java.util.function.Predicate`, and you'll find it has five methods: `and()`, `isEqual()`, `negate()`, `or()`, and `test()`. Every method except `test()` is declared as static or default, so it's pretty easy to determine that `test()` is the single abstract method in this functional interface. And, indeed, if you look under "Abstract Methods" for this interface, you'll see one method there, `test()`. Clearly, `Predicate` is a functional interface.

Now, take a look at the documentation for `java.util.Comparator`. This has quite a few default and static methods, so click on "Abstract Methods" to narrow down your search for its one and only one abstract method. What do you see? Two methods: `compare()` and `equals()`. But we said this is a functional interface, so why are two abstract methods listed here? `equals()` is inherited from `Object`, and inherited public methods are not counted when you're determining whether an interface is a functional interface. So even though `equals()` is abstract in `Comparator`, because it's inherited, it doesn't count. Tricky! Be ready to spot this on the exam.

To sum up, here is the rule for functional interfaces: A functional interface is an interface that has one abstract method. Default methods don't count; static methods don't count; and methods inherited from Object don't count.

Oh, and just so you know, the single abstract method in a functional interface is called the "functional method."

Categories of Functional Interfaces

You'll find 43 functional interfaces in `java.util.function`, and they all fall into one of four categories: suppliers, consumers, predicates, or functions. Each functional interface has a single abstract method (the functional method) and sometimes static and default methods.

The basic types of functional interfaces are `Supplier`, `Consumer`, `Predicate`, and `Function`.

Suppliers Suppliers supply results. `java.util.function.Supplier`'s functional method, `get()`, never takes an argument, and it always returns something.

Consumers Consumers consume values. `java.util.function.Consumer`'s functional method, `accept()`, always takes an argument and never returns anything.

Predicates Predicates test things (and do logical operations). `java.util.function.Predicate`'s functional method, `test()`, takes a value, does a logical test, and returns true or false.

Functions You can think of functions as the most generic of the functional interfaces. `java.util.function.Function`'s functional method, `apply()`, takes an argument and returns a value.

All of the functional interfaces in `java.util.function` are variations of suppliers, consumers, predicates, and functions. For example, `BooleanSupplier` is a supplier whose functional method supplies a boolean value. An `IntConsumer` is a consumer whose functional method takes an `int` value. And so on. Most of the variations are either to give you a way to provide more arguments (e.g., a `BiConsumer` that takes two arguments instead of one) or avoid autoboxing (e.g., `IntConsumer` and `BooleanSupplier`), which makes them more efficient. We'll give a lot of examples in the next few sections, so you get a sense of how these work.

When you first start working with functional interfaces, they can seem rather abstract. That's because, on their own, they don't really mean anything. What on earth is the point of having a `Function` interface with a method that takes a value and returns a value? That seems like the most generic thing in the world and rather pointless.

Seeing these interfaces in use and using them yourself will help make them seem less abstract and more useful. One good way to see functional interfaces in use is to look at the Oracle API documentation for one of the interfaces and click on "USE" in the top navigation bar. This shows you how that functional interface is used in the rest of the JDK.

Working with Suppliers

Let’s begin with a super simple supplier:

```
Supplier<Integer> answerSupplier = () -> 42;
System.out.println("Answer to everything: " + answerSupplier.get());
```

And in the output we see

```
Answer to everything: 42
```

We’ve made a `Supplier` whose functional method, `get()` returns an `Integer` object, `42`.

Looking at the definition of `java.util.function.Supplier`, we can see that it’s defined like this:

```
@FunctionalInterface
public interface Supplier<T>
```

And its functional method is `get()`, which returns a value of type `T`. So here `T` is a type parameter indicating the type of what’s returned from the functional method. We like the method detail from the documentation page where it describes what the `get()` method returns: “A result.” About as generic as you can get, right? This is why functional interfaces can feel so abstract when you first start working with them.

Just to cement how the lambda expression is being used for `answerSupplier` in the code above, let’s take a look at how we might create a `Supplier` without using a lambda expression, so you can see one more time how we get to the lambda from, say, an inner class. Remember, `Supplier` is just an interface! So we can make an instance of a class implementing that interface the old-fashioned way, with an inner class, like this:

```
Supplier<Integer> answerSupplierInnerClass = new Supplier<Integer>() {
   public Integer get() {
      return 42;
   }
};
```

As you can see, using a lambda expression to stand in for this inner class is a whole lot shorter and easier. Hopefully by this point, you've got the hang of reading lambda expressions. From here on out, we'll primarily use lambda expressions when implementing functional interfaces. That said, remember you might see a full-blown implementation of a functional interface on the exam.

Okay, so a supplier that supplies `42` every time isn't that interesting; let's try another one:

```
Supplier<String> userSupplier = () -> {
   Map<String, String> env = System.getenv();  // get the system environment map
   return env.get("USER");                     // get the value with the key
                                               // "USER" from the map and
                                               // return it (Note: on Windows,
                                               // this key is "USERNAME")
};
System.out.println("User is: " + userSupplier.get());
```

Here we've got a supplier with multiple statements, so we're using a slightly longer form of the lambda expression with curly braces to define a block of code and including a `return` statement to return the `String` that this supplier expects. In this case the `String` that's returned is the system username.

The key thing to note about both of these suppliers is they take no arguments and return an object. That's what suppliers do.

IntSupplier Looking at `java.util.function`, we can see that there are some

variations on supplier, including `IntSupplier`, `DoubleSupplier`, and `LongSupplier`. As you can guess, these supply an `int`, a `double`, and a `long`, respectively. These are there primarily to avoid autoboxing in case you want a primitive, rather than an object, back from the supplier. The functional method names of each of these is different and is not `get()`, so you'll need to remember that for the exam. For example, the functional method of `IntSupplier` is `getAsInt()`, so we can use the following code to make a supplier that returns a new random `int` when you call the `getAsInt()` method:

```
Random random = new Random();
IntSupplier randomIntSupplier = () -> random.nextInt(50);
int myRandom = randomIntSupplier.getAsInt();
System.out.println("Random number: " + myRandom);
```

Notice that `IntSupplier` doesn't use a type parameter, because the functional method returns a primitive. This avoids the autoboxing to `Integer` that we get with `Supplier<Integer>` (as in our `answerSupplier` above), which saves just a tiny bit of computation time (woo hoo!).

What's the Point of Supplier? At this stage, you might be asking yourself, What is the point of a supplier? After all we could just as easily write `42`, or `random.nextInt(50)`, or put the block of code into a regular method, instead of creating a lambda expression and then calling its functional method.

Looking at how `Suppliers` are used in the JDK can give you a hint as to where and why they are useful. As an example we'll take a look at the `java.util.logging.Logger` class, which has been augmented with several methods that take `Supplier` as an argument. `Logger` is used to log messages for a system or application component, and you can set the logging level to determine what kinds of messages and how many messages to log. We can use the `log()` method to log a string if the log level is set, like this:

```
Logger logger = Logger.getLogger("Status Logger");
logger.setLevel(Level.SEVERE);

// Later...
String currentStatus = "Everything's okay";
logger.log(Level.INFO, currrentStatus);
```

In this example, we do *not* see the current status string logged because the log level is set to `SEVERE` and our call to the `log()` method says only log the message if the level is set to `INFO` or below (logging levels are ordered with `SEVERE` at the highest level, and `INFO` is a couple of levels below that). If we write this instead

```
String currentStatus = "Something's horribly wrong!";
logger.log(Level.SEVERE, currentStatus);
```

then we will see the message.

Now what if we need to do some expensive call to check the status of the system? Here we've hard-coded the status to a `String` but it's more likely that the status is actually determined by, say, making a network call to see if a system is up or down. If we go ahead and compute the status before sending a string to the `log()` method, then we've potentially wasted time checking the status if the log level is not set high enough that we'll actually log the status string. This is where `Supplier` comes in handy.

A new `log()` method in `Logger` takes a level and a `Supplier<String>` rather than a `String`. Let's see how we might use this. We'll write some code that will determine the status of a system by sending a network request to a host to check to see if it's up and running. Imagine this is part of a bigger program that does something useful with the data from the host (in this case, javaranch.com); perhaps it alerts you every time someone posts a new message.

We want to log the status—that is, whether the host is up—when things are fine, but only if the logging level is set to `INFO` or below. We don't want to bother checking the status and logging anything if the logging level is set

to SEVERE. In that case, we only want to log messages that indicate things are really bad!

And we want to log the status if things go wrong, pretty much no matter what the logging level is, so we'll log the status if the level is set to SEVERE or below (and since SEVERE is the highest level, then we'll *always* log the status if things go wrong).

Here's the code:

```
String host = "coderanch.com";
int port = 80;
// set up logging
Logger logger = Logger.getLogger("Status Logger");
logger.setLevel(Level.SEVERE);          // line 5

// in case we need to check the status
Supplier<String> status = () -> {
    int timeout = 1000;
    try (Socket socket = new Socket()) {
        socket.connect(new InetSocketAddress(host, port), timeout);
        return "up";
    } catch (IOException e) {
```

```
        return "down";                  // Error; can't reach the system!
    }
};

try {
    logger.log(Level.INFO, status);     // only calls the get() method of the
                                        // status Supplier if level is INFO
                                        // or below
   // do stuff with coderanch.com
   // ...
} catch (Exception e) {
   logger.log(Level.SEVERE, status);   // calls the get() method of the status
                                       // Supplier if level is SEVERE or below
}
```

The logger's `log()` method takes the status `Supplier` and checks the log level. Because we've set the log level to `SEVERE` on line 5, in the `try` block, where we call the `log()` method with `Level.INFO`, the `log()` method won't bother calling the `get()` method of the status `Supplier`, and so we avoid making that expensive network call. And, of course, in the `catch` block, we're passing `Level.SEVERE` with the status `Supplier`, so if we end up in this `catch` block, we'll always use the `Supplier`: the `log()` method will call the status `Supplier's get()` method, get the status (a `String`—look at the type parameter on the `Supplier`), and log it.

Imagine if you were to always check the status to get the string to pass to `log()` instead. In that case, you'd be checking the status unnecessarily in the case where everything's fine and the logging level is set to `SEVERE`.

So using a `Supplier` here avoids that expensive operation when it's unnecessary. We're passing a block of code (the `Supplier`) that gets executed only if a certain condition applies. We don't have to check that condition ourselves; `Logger` does it for us and then can call the `get()` method on the

`Supplier` only if it needs to. (You can test the log status yourself by adding a

```
throw new IOException();
```

in the `try` block if you want).

It's no different than if you created your own `Supplier` interface with a `get()` method that could be called whenever a value is needed, but having these built-in functional interfaces that other parts of the JDK can use in situations like this makes life just a bit easier for you. And using lambda expressions to eliminate one more step (actually instantiating a class that implements that interface) reduces your work, helps make your code more concise, and (once you're used to reading lambda expressions) easier to read.

Working with Consumers

You can think of consumers as the opposite of suppliers (like matter and antimatter, it helps keeps the universe balanced). Consumers accept one or more arguments and don't return anything. So this lambda expression

```
Consumer<String> redOrBlue = pill -> {
   if (pill.equals("red")) {
      System.out.println("Down the rabbit hole");
   } else if (pill.equals("blue")) {
      System.out.println("Stay in lala land");
   }
};
```

implements the `java.util.function.Consumer` interface, which is defined like this:

```
@FunctionalInterface
public interface Consumer<T>
```

The `Consumer`'s functional method is `accept()`, which takes an object of type

T and returns nothing, so we use the redOrBlue consumer like this:

```
redOrBlue.accept("red");
```

and see the output:

```
Down the rabbit hole
```

As with suppliers, there are variations on consumers in the java.util.function package. IntConsumer, DoubleConsumer, and LongConsumer do what you'd expect: their accept() methods take one primitive argument and avoid the autoboxing you get with Consumer.

In addition to these, there's ObjIntConsumer, ObjDoubleConsumer, and ObjLongConsumer, whose accept() methods take an object (type T) and an int, a double, or a long.

And finally we have BiConsumer, which is similar, except that its accept() method takes two objects (types T and U, meaning the two objects don't have to be of the same type). So the BiConsumer interface looks like this:

```
@FunctionalInterface
public interface BiConsumer<T,U>
```

with one function method, accept():

```
void accept(T t, U u)
```

Here's a good use for a BiConsumer:

```
Map<String, String> env = System.getenv();
BiConsumer<String, String> printEnv = (key, value) -> {
   System.out.println(key + ": " + value);
};
printEnv.accept("USER", env.get("USER"));
```

The printEnv BiConsumer is a lambda expression with two arguments that

we are using to display a key and value from a `Map`. To use the `printEnv` consumer, we call its `accept()` method, passing in two strings, and see the result displayed in the console.

ForEach Now is a good time to talk about the `forEach()` method that's been added to the Java 8 `Iterable` interface. Why? Because `forEach()` expects a consumer. You'll end up using `forEach()` a lot, as it's a handy way to iterate through collections.

Here's how you use `forEach()` with a `Consumer` to iterate through a `List` and display each item in the list:

```
List<String> dogNames = Arrays.asList("boi", "clover", "zooey");
Consumer<String> printName = name -> System.out.println(name);
dogNames.forEach(printName);    // pass the printName consumer to
                                // forEach()
```

We could, of course, combine the last two of these lines, like this:

```
dogNames.forEach(name -> System.out.println(name));
```

This consumer takes a `String` and just prints it.

The kind of consumer `forEach()` expects depends on the type of collection you're using. For example, when you iterate through `List`, you're accessing one object at a time, so the consumer you'll use is `Consumer` (that is, a consumer whose `accept()` method expects one argument, an object). And for `Map`, the consumer you'll use is a `BiConsumer` (that is, a consumer whose `accept()` method expects two arguments, both objects). Let's use the `BiConsumer` we created earlier, `printEnv`, with the `Map`'s `forEach()` method to display every key/value pair in the map:

```
Map<String, String> env = System.getenv();
BiConsumer<String, String> printEnv = (key, value) -> {
   System.out.println(key + ": " + value);
};
```

```
env.forEach(printEnv);   // the forEach() method of Map expects a BiConsumer
```

This displays every key/value pair in the `Map` that you get from `System.getenv()`.

Side Effects from Within Lambdas You already know that there are restrictions on modifying the value of variables from within lambda expressions. For instance, if you define a variable in the enclosing scope of a lambda expression, you can't modify that variable from within the lambda:

```
String username;
BiConsumer<String, String> findUsername = (key, value) -> {
   if (key.equals("USER")) username = value;  // compile error!
                                              // username must be
                                              // effectively final
};
env.forEach(findUsername);
```

This code will not compile because we're trying to change the value of `username` from within the lambda expression. Remember that variables declared outside a lambda expression must be final or effectively final to be used within a lambda expression.

However, we can cheat. Although we can't modify a variable from within a lambda expression, we *can* modify a field of an object. If we want to use `forEach()` to iterate through a collection of objects to, say, find a value, we can't return the value we find (because `forEach()` takes a consumer and the `accept()` method of a consumer is void), but we can change the field of an object from within the lambda to do effectively the same thing:

```
public class Consumers {
   public static void main(String[] args) {
      Map<String, String> env = System.getenv();
      User user = new User();
      BiConsumer<String, String> findUsername = (key, value) -> {
         if (key.equals("USER")) user.setUsername(value);
      };
      env.forEach(findUsername);
      System.out.println("Username from env: " + user.getUsername());
   }
}
class User {
   String username;
   public void setUsername(String username) {
      this.username = username;
   }
   public String getUsername() {
      return this.username;
   }
}
```

Note that this code will be less efficient than using a `for` loop iteration and breaking out of the iteration once the value is found. You might not ever need to do this, but now you know you can, just in case. In the next chapter on Streams, you'll learn a better way to extract and collect values that doesn't rely on side effects, as we're doing here.

andThen… andThen the murders began… Oh wait, no. This isn't a crime thriller; it's a Java book.

`andThen()` is actually a default method of the `Consumer` interface that you can use to chain consumers together. Let's imagine you have a `Dog` that can bark:

```
public class Dog {
    private String name;
    private int age;
    private int weight;

    public Dog(String name, int weight, int age) {
        this.name = name;
        this.weight = weight;
        this.age = age;
    }
    // getters and setters here...
    public String toString() { return this.name; }
    public void bark() { System.out.println("Woof!"); }
}
```

Now let's make some dogs and display them with `forEach()`:

```
public class ConsumerDogs {
   public static void main(String[] args) {
      List<Dog> dogs = new ArrayList<>();
      Dog boi = new Dog("boi", 30, 6);
      Dog clover = new Dog("clover", 35, 12);
      Dog zooey = new Dog("zooey", 45, 8);
      dogs.add(boi); dogs.add(clover); dogs.add(zooey);

      Consumer<Dog> displayName = d -> System.out.print(d + " ");
      dogs.forEach(displayName);     // line 10
   }
}
```

When you run this, you'll see

```
boi clover zooey
```

Now let's say we want to display the dog's name *andThen* we want to have the dog bark. Can we do it with consumers? Yes! Here's how:

```
dogs.forEach(displayName.andThen(d -> d.bark()));
```

Replace line 10 with this line and now you'll see

```
boi Woof!
clover Woof!
zooey Woof!
```

So how does this work? We're passing a "composed `Consumer`" to the `forEach()` method of the dogs `ArrayList`. Let's step through this.

First, note that the `forEach()` method is calling the `accept()` method of

the `Consumer` you're passing in behind the scenes. So for each dog `d` in the list, `forEach()` is essentially doing this:

```
displayName.accept(d)
```

When the `andThen()` method of the `Consumer` is called, it says, okay, now use that same dog object, `d`, that we just used in the first `Consumer` for the `accept()` method of the second `Consumer`. Note that we've written the second `Consumer` as an inline lambda (rather than as a separate declaration like we did for `displayName`)—but, of course, it works the same way. So the dog `d` whose name we just displayed is used in the second `Consumer`, and we call that dog's `bark()` method, which simply displays `Woof!` in the console. And so the result we see is the dog's name followed by `Woof!` for each of the dogs in the list.

You might think that you could write both lambdas inline, like this:

```
dogs.forEach((d -> System.out.print(d + " ")).andThen(d -> d.bark()));
```

But you can't. You'll get a compile error. You can, however, use named consumers for both:

```
Consumer<Dog> displayName = d -> System.out.print(d + " ");
Consumer<Dog> doBark = d -> d.bark();
dogs.forEach(displayName.andThen(doBark));
```

Most (but not all!) of the consumers in `java.util.function` have an `andThen()` method, and notice that the type of the consumer you pass to the `andThen()` method must match the type of the consumer used as the first operation. So you can chain a `Consumer` with a `Consumer` and a `BiConsumer` with a `BiConsumer`, but not a `Consumer` with a `BiConsumer`.

Working with Predicates

Remember earlier in this chapter we created a `DogQuerier` interface and used that interface to create an inner class and then a lambda expression.

Our interface looked like this:

```
interface DogQuerier {
   public boolean test(Dog d);
}
```

This interface is a functional interface, with a functional method `test()`, so although we could create an instance using an inner class:

```
DogQuerier dq = new DogQuerier() {
   public boolean test(Dog d) { return d.getAge() > 9; }
};
```

we realized we could create an instance much more concisely using a lambda expression like this:

```
DogQuerier dq = d -> d.getAge() > 9;  // replaces inner class!
```

Well, take a look at `java.util.function.Predicate`, and the interface might look familiar:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // default and static methods here.
}
```

`Predicate` is a functional interface, with a functional method, `test()`, which returns a `boolean`, just like our `DogQuerier`. That means we can use the built-in `Predicate` interface in place of the `DogQuerier` interface (and get rid of the `DogQuerier` interface definition completely). Let's do that:

```
Dog boi = new Dog("boi", 30, 6);
Dog clover = new Dog("clover", 35, 12);

Predicate<Dog> p = d -> d.getAge() > 9;
System.out.println("Is Boi older than 9? " + p.test(boi));
System.out.println("Is Clover older than 9? " + p.test(clover));
```

Boi is 6, and Clover is 12, so we see `false` and `true` for the results, just like we did before with `DogQuerier`.

Notice that one thing we must do to use `Predicate` in place of `DogQuerier` is add a type parameter to `Predicate`. Whereas we created `DogQuerier` to be specific to dogs, `Predicate` is a generic functional interface, so we have to provide a bit more information. The `T` argument defined in the `test()` method means we can pass any object to `Predicate`, so by adding a type parameter to `Predicate`, the predicate will know what type of argument to expect.

Let’s expand the example just a bit so we can experiment more with predicates.

```
public class Dog {
   private String name;
   private int age;
   private int weight;

   public Dog(String name, int weight, int age) {
      this.name = name;
      this.weight = weight;
      this.age = age;
   }
   // getters and setters here
   // add a better description of the dog:
   public String toString() {
      return this.name + " is " + this.age + " years old and weighs " +
this.weight + " pounds";
   }
   public void bark() { System.out.println("Woof!"); }
}


public class TestDogPredicates {
   public static void main(String[] args) {
      ArrayList<Dog> dogs = new ArrayList<>();
      Dog boi = new Dog("boi", 30, 6);
      Dog clover = new Dog("clover", 35, 12);
      Dog aiko = new Dog("aiko", 50, 10);
      Dog zooey = new Dog("zooey", 45, 8);
      Dog charis = new Dog("charis", 120, 7);
      dogs.add(boi); dogs.add(clover); dogs.add(aiko);
      dogs.add(zooey); dogs.add(charis);

      System.out.println("--- All dogs ---");
      dogs.forEach(d -> System.out.println(d));

      System.out.println("--- Dogs younger than 9 ---");
      printDogIf(dogs, (d) -> d.getAge() < 9);

      System.out.println("--- Dogs 9 or older ---");
      printDogIf(dogs, (d) -> d.getAge() >= 9);
   }

   public static void printDogIf(ArrayList<Dog> dogs, Predicate<Dog> p) {
      for (Dog d : dogs) {
         if (p.test(d)) {
            System.out.println(d);
         }
      }
   }
}
```

What we've done here is create a method `printDogIf()` that takes a list of dogs and a `Predicate`, and tests each dog in the list against the predicate to see if the dog should be displayed. We can then use this method to display all the dogs younger than 9 with one predicate and display all the dogs 9 or older with another.

Running this code, we get the following output:

```
--- All dogs ---
boi is 6 years old and weighs 30 pounds
clover is 12 years old and weighs 35 pounds
aiko is 10 years old and weighs 50 pounds
zooey is 8 years old and weighs 45 pounds
charis is 7 years old and weighs 120 pounds
--- Dogs younger than 9 ---
boi is 6 years old and weighs 30 pounds
zooey is 8 years old and weighs 45 pounds
charis is 7 years old and weighs 120 pounds
--- Dogs 9 or older ---
clover is 12 years old and weighs 35 pounds
aiko is 10 years old and weighs 50 pounds
```

Now let's check out how `Predicates` are used in the JDK. One example is the `removeIf()` method of `ArrayList`, which takes a `Predicate` and removes an item from the `ArrayList` if the predicate's `test()` method returns true for that item. So, we can remove all dogs whose names begin with "c" (sorry Charis and Clover!) like this:

```
Predicate<Dog> findCs = d -> d.getName().startsWith("c");
dogs.removeIf(findCs);
System.out.println("---- After removing dogs whose names begin with c ---");
dogs.forEach(d -> System.out.println(d));
```

And see that, indeed, Charis and Clover have been removed from the output:

```
---- After removing dogs whose names begin with c ---
boi is 6 years old and weighs 30 pounds
aiko is 10 years old and weighs 50 pounds
zooey is 8 years old and weighs 45 pounds
```

Predicate's Default and Static Methods If you look at the `Predicate` interface in the `java.util.function` package, you'll probably notice that `Predicate` has a few default methods and one static method. The default methods, `and()`, `or()`, and `negate()`, are there so you can chain predicates together, much like we did with consumers and the `andThen()` method. This can save you time creating new predicates that are logical combinations of predicates you already have. So, for instance, if we have a `Predicate` that tests to see if a dog's age is 6, we can easily test for a dog not being age 6 with the `negate()` method:

```
Predicate<Dog> age = d -> d.getAge() == 6;
System.out.println("Is boi NOT 6? " + age.negate().test(boi));
```

And we get the result false because Boi is, indeed, 6 years old:

```
Is boi NOT 6? false
```

The `or()` and `and()` methods both take other predicates, so to chain them together you need two predicates. Let's create predicates to see if a dog's name is "boi" and the dog's age is 6 and a third predicate that chains the first two together with `and()`:

```
Predicate<Dog> name = d -> d.getName().equals("boi");
Predicate<Dog> age = d -> d.getAge() == 6;
Predicate<Dog> nameAndAge = d -> name.and(age).test(d);
System.out.println("--- Test name and age of boi ---");
System.out.println("Is boi named 'boi' and age 6? " + nameAndAge.test(boi));
boi.setAge(7);
System.out.println("Is boi named 'boi' and age 6? " + nameAndAge.test(boi));
```

First, we test to see if Boi is named “boi” and is age 6, then we set Boi’s age to 7, and test again. We get the output:

```
--- Test name and age of boi ---
Is boi named 'boi' and age 6? true
Is boi named 'boi' and age 6? false
```

We can simplify the `nameAndAge` `Predicate` even further by writing

```
Predicate<Dog> nameAndAge = name.and(age);
```

This works! Remember, what this does is create a new `Predicate` that is the composition of two `Predicates`, `name` and `age`. So the result of calling the `and()` method on the `name` predicate with the argument `age` is a new `Predicate<Dog>` that ands the result of calling `name.test()` on a dog and `age.test()` on that same dog.

The syntax for chaining can take a bit getting used to. Try writing a few of your own predicates, combining them with `and()`, `or()`, and `negate()`, to get the hang of it.

The static method in `Predicate`, `isEqual()`, just gives you a way to test if one object equals another, using the same test as `equals()` uses when comparing two objects (that is, are they the same object?).

```
Predicate<Dog> p = Predicate.isEqual(zooey);
System.out.println("Is aiko the same object as zooey? " + p.test(aiko));
System.out.println("Is zooey the same object as zooey? " + p.test(zooey));
```

One thing to note about `isEqual()` is this method is defined only on the predicates that take objects as arguments.

Along with `Predicate`, you'll find `BiPredicate`, `DoublePredicate`, `IntPredicate`, and `LongPredicate` in `java.util.function`. You can probably guess what these do. Yep, `BiPredicate`'s `test()` method takes two arguments, whereas `DoublePredicate`, `IntPredicate`, and `LongPredicate` each take one argument of a primitive type (to avoid autoboxing).

Here's a quick `IntPredicate` to demonstrate:

```
IntPredicate universeAnswer = i -> i == 42;
System.out.println("Is the answer 42? " + universeAnswer.test(42));
```

`IntPredicate` is better than `Predicate<Integer>` for this example because the argument `i` doesn't have to be converted from `Integer` to `int` before it's tested. It's a similar idea with `DoublePredicate` and `LongPredicate`.

BiPredicate `BiPredicate` is just a variation on `Predicate` that allows you to pass in two objects for testing instead of one. Let's say we have an `ArrayList` of books and we want to create a set of predicates that will determine if we should buy a book based on its name, its price, or its name *and* price together. Here's how we might do that with `BiPredicate`:

```
List<Book> books = new ArrayList<>();
// fill the books list with books here...
BiPredicate<String, Double> javaBuy = (name, price) -> name.contains("Java");
BiPredicate<String, Double> priceBuy = (name, price) -> price < 55.00;
BiPredicate<String, Double> definitelyBuy = javaBuy.and(priceBuy);
books.forEach(book -> {
   if (definitelyBuy.test(book.getName(), book.getPrice())) {
      System.out.println("You should definitely buy " + book.getName()
                         + "(" + book.getPrice() + ")");
   }
});
```

If we load up our books `ArrayList` with the following books:

```
"Your Brain is Better with Java", 58.99
"OCP8 Java Certification Study Guide", 53.39
"Is Java Coffee or Programming?", 39.86
"While you were out Java happened", 12.99
```

then what we'll see in the output when we run this code is

```
You should definitely buy "OCP8 Java Certification Study Guide"(53.39)
You should definitely buy "Is Java Coffee or Programming?"(39.86)
You should definitely buy "While you were out Java happened"(12.99)
```

These are the books with Java in the title and a price less than $55.00.

Caveat Time Of course, you don't *need* `BiPredicate` lambda expressions to write this code; you could just write the test (does the name contain "Java" and is the price less than 55.00) in the `forEach` lambda expression. Likewise, we haven't exactly needed many of the lambda expressions we've written in

the simple examples throughout this chapter so far.

The main reason to use lambda expressions is if you'll end up using them in other ways too, and if having the code packaged up into lambdas will help make your code more concise and get you some code reuse (same reasons why you might use an inner class or even an external class). And, of course, if a method of a Java class, like the `Logger log()` example, requires a functional interface, like a `Supplier`, then that's a great reason to use a lambda expression (although, again, you don't *have* to—they are a convenience).

A big caveat here is that although we're showing lots of examples of building lambda expressions using the built-in functional interfaces so you get practice for the exam, once you're back in the real world, think about whether you really need a lambda expression before you write one.

Working with Functions

We saved the most abstract of the functional interfaces from `java.util.function` for last. Aren't you lucky? Of course, by now, we hope you're feeling pretty solid about using functional interfaces.

The purpose of the `apply()` functional method in the `Function` interface is to take one value and turn it into another. The two values don't have to be the same type, so in the `Function` interface definition, we have two different type parameters, `T` (the type of the argument to `apply()`) and `R` (the type of the return value):

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
}
```

We can create an instance of `Function` to turn an `Integer` into a `String`:

```
Function<Integer, String> answer = a -> {
   if (a == 42) return "forty-two";
   else return "No answer for you!";
};
System.out.println(answer.apply(42));
System.out.println(answer.apply(64));
```

If you run this code, you'll see the output:

```
forty-two
No answer for you!
```

A `BiFunction` is similar except the `apply()` method takes two arguments and returns a value:

```
BiFunction<String, String, String> firstLast =
    (first, last) -> first + " " + last;
System.out.println("First and Last name: " + firstLast.apply("Joe", "Smith"));
```

In this example, the `BiFunction` `apply()` method takes two `String`s and returns a `String`, but you could pass two arguments of different types and return a value of a third type. What we see, of course, is

```
First and Last name: Joe Smith
```

Functions in the JDK Let's use both a `Function` and a `BiFunction` in an example using the `Map` methods `computeIfAbsent()` and `replaceAll()`. These are two examples from the JDK where you'll find `Function` and `BiFunction` used.

```
public class Functions {
   public static void main(String[] args) {
      Map<String, String> aprilWinner = new TreeMap<>();
      aprilWinner.put("April 2017", "Bob");
      aprilWinner.put("April 2016", "Annette");
      aprilWinner.put("April 2015", "Lamar");

      System.out.println("--- List, before checking April 2014 ---");
      aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));

      // no key for April 2014, so John Doe gets added to the map
      aprilWinner.computeIfAbsent("April 2014", (k) -> "John Doe");

      // key April 2014 now has a value, so Jane won't be added
      aprilWinner.computeIfAbsent("April 2014", (k) -> "Jane Doe");

      System.out.println("--- List, after checking April 2014 ---");
      aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));

      // use a BiFunction to replace all values in the map with
      // uppercase values
      aprilWinner.replaceAll((key, oldValue) -> oldValue.toUpperCase());
      System.out.println("--- List, after replacing values with
uppercase ---");
```

```
        aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));

    }
}
```

The output is as follows:

```
--- List, before checking April 2014 ---
April 2015: Lamar
April 2016: Annette
April 2017: Bob
--- List, after checking April 2014 ---
April 2014: John Doe
April 2015: Lamar
April 2016: Annette
April 2017: Bob
--- List, after replacing values with uppercase ---
April 2014: JOHN DOE
April 2015: LAMAR
April 2016: ANNETTE
April 2017: BOB
```

We first use `computeIfAbsent()` to add a key and value to our `Map` of April winners if that key/value pair doesn't yet exist in the `Map`.

`computeIfAbsent()` takes a key and a `Function`. The `Function` provides a value to store in the `Map` for the key if a value for that key doesn't yet exist. The argument in the `Function` lambda expression is the key, so you could create a value based on the key, but in this example, we're keeping it super simple and just returning a `String`.

Then we use the `replaceAll()` method to replace every value in the `Map` with the uppercase version of the old value. So where we have stored `Bob`, we'll now be storing `BOB`.

`replaceAll()` takes a `BiFunction`. The lambda expression we pass to `replaceAll()` has two arguments, a key, and the current value in the `Map`, and returns a new value to store in the `Map` for that key. In this example, we're just returning the previous value in uppercase.

More Functions `Function` has a couple of default methods and a static method in addition to its functional method, `apply()`: `andThen()`, `compose()`, and `identity()`. `andThen()` is similar to the `Consumer`'s `andThen()` method, applying `Functions` in sequence. `compose()` is the same except it applies the `Functions` in reverse order.

And the static method in `Function` is `identity()`, which just returns its input argument:

```
Function<Integer, Integer> id = Function.identity();
System.out.println(id.apply(42));
```

Answer: 42. (Of course.)

What on earth is an identity function used for? Imagine a scenario where you have defined a method that takes a `Function` as an argument that changes a value in a data structure. But in some cases, you don't want that value to change. In those cases, pass the identity `Function` as an easy "do nothing" operation.

Along with `Function` and `BiFunction`, as you might expect, you'll also find `DoubleFunction` (`apply()` takes a `double` as an argument and returns an object), `IntFunction` (`apply()` takes an `int` as an argument and returns an object), and `LongFunction` (`apply()` takes a `long` as an argument and returns an object).

Ah, but wait, there's more! We also have `DoubleToIntFunction`, `DoubleToLongFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongToDoubleFunction`, `LongToIntFunction`, `ToDoubleFunction`, `ToIntFunction`, `ToLongFunction`, `ToDoubleBiFunction`, `ToIntBiFunction`, and last but definitely not least, `ToLongBiFunction`.

Oh, my goodness—the variations the JDK authors thought of when

creating the functional interfaces. (It's a wonder they didn't think of 100 more.) We are pretty sure you can make a good guess at what these do. The main trick with these is that the functional method is not `apply()`; it's a slight variation on `apply()`, so just keep that in the back of your mind.

For instance, how about `IntToDoubleFunction`? The functional method is `applyAsDouble()`, and yes, it takes an `int` and returns a `double`, avoiding all that inefficient autoboxing. How about `ToIntFunction`? The `applyAsInt()` method takes an object and returns an `int`.

Okay, we are sure you get the hang of it at this point, and we'll leave you to the Java docs to find out more if you're interested. For the exam, focus on `Function` and `BiFunction`, but be aware that these other variations exist and remember that there are variations in the functional method names that go with them.

Working with Operators

Finally (yes, really!), we have the operator variations on the functional interfaces. All the operators are, in fact, slightly modified versions of other functional interfaces. Let's pick the one operator you should be familiar with for the exam, `UnaryOperator`, to look at, and you can explore the rest on your own.

`UnaryOperator` extends the `Function` interface, so its functional method is also `apply()`. However, unlike `Function`, it requires that the type of the argument to `apply()` be the same as the type of the return value, so `UnaryOperator` is defined like this:

```
@FunctionalInterface
public interface UnaryOperator<T>
extends Function<T,T>
```

The `T, T` in the `Function` type parameters is what tips you off that the type of the argument and return value must be the same. That's why you only have to specify one type parameter for `UnaryOperator`:

```
UnaryOperator<Double> log2 = v -> Math.log(v) / Math.log(2);
System.out.println(log2.apply(8.0));
```

In this example, we're defining a `log2` `UnaryOperator` that computes the log base 2 of a value. Log base 2 of 8.0 is 3.0 because 2 * 2 * 2 (3 times) is 8.

Notice that you could, of course, use `Function<Double, Double>` instead…that's essentially the same thing. `UnaryOperator` saves a little typing and makes it a bit clearer that you're defining an operator that takes a value that is the same type as the return value…but that's about it. It's just a slightly restricted version of `Function`. And the same applies to the other operators in the package.

Time to pat yourself on the back, take a break, and eat some cookies, because you made it through all of the functional interfaces in `java.util.function`. As long as you have a good sense of how to use them to create lambda expressions and how to use the core interfaces that we've covered here, you'll be in solid shape.

CERTIFICATION OBJECTIVE

Method References (OCP Objective 3.8)

3.8 Use method references with Streams.

You already know that a lambda expression is a shorthand way of writing an instance of a class that implements a functional interface. Believe it or not, there are a few circumstances when you can make your code even *more* concise by writing a shorthand for the lambda expression (yes, it's a shorthand for the shorthand).

Sometimes, the only thing a lambda expression does is call another method, for instance:

```
List<String> trees = Arrays.asList("fir", "cedar", "pine");
trees.forEach(t -> System.out.println(t));
```

Here we're using a lambda expression to take a tree name, `t`, and pass it to `System.out.println()`. This code is already pretty short; can we shorten it even more? Yes! Apparently, the Java 8 authors like finding ways to avoid typing, so they invented the "method reference":

```
List<String> trees = Arrays.asList("fir", "cedar", "pine");
trees.forEach(System.out::println);
```

This method reference is a shorthand way of writing the lambda expression. But wait, where did the argument `t` go and don't we need that? We know that `forEach()` takes a `Consumer`, and we know what it's consuming is tree names, which are `String`s. And we know that `System.out.println()` takes a `String`. A lot can be inferred from this shorthand for the lambda expression. Here what's inferred is that we want to call the `println()` method of `System.out`, passing in the `String` object that we have at hand via the `forEach()`. Of course, keep in mind, we can't do anything fancy with how we print the tree name because we're not specifying anything but the method to call on the tree name argument that the `Consumer` (the method reference) is getting behind the scenes.

Kinds of Method References

The `System.out::println` method reference is an example of a "method reference," meaning we're calling a method, `println()`, of `System.out`. You can create method references for your own methods, too:

```
public class MethodRefs {
   public static void main(String[] args) {
      List<String> trees = Arrays.asList("fir", "cedar", "pine");

      trees.forEach(t -> System.out.println(t));  // print with lambda
      trees.forEach(System.out::println);         // print with a
                                                  // method reference

      trees.forEach(MethodRefs::printTreeStatic); // print with our own
                                                  // static method reference
   }

   public static void printTreeStatic(String t) {
      System.out.println("Tree name: " + t);
   }
}
```

For this code, we'll see this output:

```
fir
cedar
pine
fir
cedar
pine
Tree name: fir
Tree name: cedar
Tree name: pine
```

First, the tree names are printed using a lambda expression; then the tree names are printed using an *instance method reference* to `System.out.println()` and finally a longer string using a *static method reference* to our own static method, `printTreeStatic()`. A "static method reference" is a method reference to a static method. The method reference to `System.out.println()` is an "instance method reference"—that is, a reference to the `println()` instance method of `System.out`.

Writing `::` instead of `.` in the method reference takes a little getting used to; you'll find yourself typing a `.` instead of a `::` and then going back to fix it (at least we do!). What the method reference does here is make a `Consumer` that calls the method on the implicit argument that's getting passed to that lambda behind the scenes.

In addition to instance and static method references, there are a couple of other types of method references: method references for arbitrary objects and constructor method references. We'll see some more examples of method references in the next chapter when we talk about streams.

Write Your Own Functional Interface

Guess what, you already wrote your own functional interface. Remember `DogQuerier`? Yep, that's the one:

```
@FunctionalInterface
interface DogQuerier {
   public boolean test(Dog d);
}
```

Of course, `DogQuerier` is all about `Dogs`. You then saw that `DogQuerier` is really just a `Dog` version of `Predicate`, which works on any type of object.

You might want to write a functional interface that works on any type of object, too. Imagine, for instance, that you want an interface with a `test()` method that takes three objects and returns a `boolean`—a `TriPredicate` if you will. There is no `TriPredicate` in `java.util.function` (only `Predicate` and `BiPredicate`), so how about writing your own?

```
@FunctionalInterface
interface TriPredicate<T, U, V> {
   boolean test(T t, U u, V v);
}
```

We're specifying an interface with three type parameters: `T`, `U`, and `V`. The `test()` method takes three types of objects to test, and they can all be of differing types or all the same. Now let's write a lambda expression of this type and test it out:

```
public void triPredicate() {
   TriPredicate<String, Integer, Integer> theTest =
      (s, n, w) -> {
         if (s.equals("There is no spoon") && n > 2 && w < n) {
            return true;
         } else {
            return false;
         }
      };
      System.out.println("Pass the test? " +
                         theTest.test("Follow the White Rabbit", 2, 3));
      System.out.println("Pass the test? " +
                         theTest.test("There is no spoon", 101, 3));
}
```

We see this output:

```
Pass the test? false
Pass the test? true
```

The trick to writing your own generic functional interfaces is having a good handle on generics. As long as you understand what a functional interface is and how to use generics to specify parameter and return types, you're good to go.

Functional Interface Overview

As we've said, the `java.util.function` package has 43 different functional interfaces. You need to make sure you are familiar with the core interfaces: `Supplier`, `Consumer`, `Predicate`, and `Function`. In addition, you should understand the variations on these core interfaces—the primitive interfaces

designed to avoid autoboxing, and the `Bi`-versions that allow you to pass two parameters to the functional methods rather than one. You don't need to memorize all 43 interfaces, but you need to understand the patterns of the variations and their functional method names. We've listed the core interfaces, plus several variations, and their functional methods in Table 8-1. In addition to the functional methods, you should be familiar with the default and static methods in the interfaces, such as `andThen()`, `and()`, `negate()`, `or()`, and so on. We don't list those in Table 8-1, so look at the online documentation for details beyond what we've covered in this chapter.

TABLE 8-1 Functional Interfaces Covered on the Exam

Interface	Functional Method	Description
`Supplier<T>`	`T get()`	Suppliers supply results. The `Supplier`'s functional method `get()` takes no arguments and supplies a generic object result (`T`).
`Consumer<T>`	`void accept(T t)`	Consumers consume values. The `Consumer`'s functional method `accept()` takes a generic object argument (`T`) and returns no value.
`Predicate<T>`	`boolean test(T t)`	Predicates test things (and do logical operations). The `Predicate`'s functional method `test()` takes a generic object argument (`T`) and returns a `boolean`.
`Function<T,R>`	`R apply(T t)`	Functions are generic functional interfaces. The functional method `apply()` takes a generic object argument (`T`) and returns a generic object (`R`).
`IntSupplier`	`int getAsInt()`	Supplies `int` values (to avoid autoboxing). Note the difference in the functional method name (`getAsInt()` rather than `get()`, as for `Supplier`). This pattern is used across the various primitive versions of the functional interfaces.
`IntConsumer`	`void accept(int value)`	Consumes `int` values.
`IntPredicate`	`boolean test(int value)`	Tests `int` values.
`IntFunction<R>`	`R apply(int value)`	The `IntFunction`'s functional method `apply()` takes an `int` argument and returns a generic object result (`R`).
`BiConsumer<T,U>`	`void accept(T t, U u)`	Consumes two values.
`BiPredicate<T,U>`	`boolean test(T t, U u)`	The `BiPredicate`'s functional method `test()` takes two generic arguments (`T, U`).
`BiFunction<T,U,R>`	`R apply(T t, U u)`	The `BiFunction`'s functional method `apply()` takes two generic arguments (`T, U`) and returns a generic object result (`R`).
`UnaryOperator<T>`	`T apply(T t)`	The `UnaryOperator`'s functional method `apply()` takes one generic operator (`T`) and returns a value of the same type (`T`).

Note: You should review all the functional interfaces in `java.util.function` for variations on the functional interfaces covered in Table 8-1 and for details of the default and static methods of interfaces.

You know that functional interfaces are just interfaces with one abstract method—the functional method. The interfaces in Table 8-1 have been added to the JDK to make it easier for you to write code, but as you know, there's nothing particularly special about functional interfaces beyond their use in the JDK (as we saw with Logger).

There are a few functional interfaces already in the JDK, not listed in `java.util.function`. You've already seen one of these—Comparator—with its functional method, `compare()`. Another is `Comparable`, which is implemented by various types (like `String`, `LocalDateTime`, and so on) for sorting.

A third example is `Runnable`, which has a functional method, `run()`. So where you might have implemented a `Runnable` as an inner class before:

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Do this");
    }
};
```

you can now use a lambda expression instead:

```
Runnable r = () -> System.out.println("Do this");
```

CERTIFICATION SUMMARY

We began this chapter by looking at what lambda expressions are, how to write lambda expressions, and when to write them. Lambdas are syntax shorthands for writing a class to implement a functional interface and then instantiating that class. With lambdas, you focus on the method that's implementing the abstract method of the interface (remember, there's only one, because the functional interfaces have only one abstract method) and eliminate the other syntax, so you end up with more concise code.

The lambda syntax can take a little getting used to, but there aren't too many rules to remember: you use an `->` symbol with parameters on the left and a method body on the right. One trick is that if a lambda expression is just one expression, you can even eliminate the `{` and `}` that usually go with a method body, like this:

```
() -> 42
```

We talked about how you can think of lambdas as chunks of code that you pass around. We can pass an object to a method, and we can pass a lambda expression to a method. Remembering that a lambda represents an instance of a class that implements an interface can help you see how this works.

When lambdas refer to variables from their enclosing scope, those variables are "captured" and so you can refer to them when applying a lambda later. The important key here, however, is that the variables must be final or effectively final.

Then we delved into functional interfaces. A functional interface is simply an interface that has one abstract method—the functional method. Lambda expressions are related to functional interfaces because the type of a lambda expression is always a functional interface. Why? Because the method the lambda expression defines is the one and only functional method in that interface, so there is no confusion in the lambda expression syntax about which method you're writing.

The tricky part of functional interfaces is learning about the new interfaces defined in `java.util.function`. Forty-three new interfaces are there to help you shortcut the process of writing functional interfaces and also to provide a way for other JDK methods to accept objects of a functional interface type. The core functional interfaces are `Consumer`, `Supplier`, `Predicate`, and

`Function`, and all the other functional interfaces in `java.util.function` are variations on these. We talked about a few examples in which new JDK methods, like the `Logger`'s `log()` method, can now take objects of one of the types in this package.

As we covered the various functional interfaces, we looked at the functional methods defined for each one, like the `Predicate`'s `test()` method and the `Supplier`'s `get()` method, and some of the static and default methods included in some of the functional interfaces, too, like the `Consumer`'s `andThen()` method.

We talked about operators, which are just slight variations on the other functional interfaces and have a restriction that the parameter type is the same as the return value type. So, if you define a `UnaryOperator` whose functional method takes an `Integer`, that operator method must return an `Integer` too.

A lambda expression is a syntax shorthand to make your code more concise, and a method reference is a shorthand that can make your code even more concise in certain situations, such as when you're just passing on an argument to another method. A common example for this is replacing a lambda expression that simply prints its argument:

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

with a method reference:

```
Consumer<String> c = System.out::println;
```

Java can infer that you want to pass on the argument to the `Consumer` (which must have an argument; it's a `Consumer`!) to the `System.out.println()` method. Again, this syntax takes some getting used to. You'll see it again in the next chapter and get more practice with it.

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Create and Use Lambda Expressions (OCP Objective

2.6)

- A lambda expression is a shorthand syntax for an instance of a class that implements a functional interface.
- Use `->` to define a lambda expression with the arguments on the left and the body on the right of the arrow.
- Typically, we leave off the types of a lambda's arguments because they can be inferred from the functional interface definition.
- If you have multiple parameters for a lambda, then you must surround them with parentheses.
- If you have no parameters for a lambda, then you must use empty parentheses, ().
- If you specify the type of a parameter of a lambda, you must use parentheses.
- If the body of a lambda expression has multiple statements, you must use curly braces.
- Lambda expressions are often used in place of an inner class.
- If the body of a lambda expression simply evaluates an expression and returns a value, you can leave off the `return` keyword.
- The type of the return value of a lambda expression is inferred from the functional interface definition.
- You can pass lambda expressions to methods, either by name or by writing them inline.
- Lambda expressions capture variables from the enclosing scope if they are used within the body of the lambda.
- All captured variables in a lambda expression must be final or effectively final.

Iterate Using forEach Methods of List (OCP Objective 3.5)

- The `forEach()` method in collection types, like `List`, takes a `Consumer` and allows you to easily iterate through the collection. The `Consumer`'s `accept()` method argument is the current object in the

collection you are iterating over.

Use the Built-in Interfaces Included in the java.util.function Package such as Predicate, Consumer, Function, and Supplier (OCP Objective 4.1)

- A functional interface is an interface with one abstract method.
- The single abstract method in a functional interface is called the functional method.
- Functional interfaces can include any number of default and static methods in addition to the functional method.
- Functional interfaces can redefine public methods from `Object`.
- Use `@FunctionalInterface` to annotate functional interfaces.
- If you add a second abstract method to a functional interface annotated with `@FunctionalInterface`, you will get a compiler error: "Invalid '`@FunctionalInterface`' annotation; [interface name] is not a functional interface."

Core Functional Interfaces (OCP Objectives 4.1, 4.2, 4.3, 4.4)

- The JDK provides several built-in functional interfaces in `java.util.function`.
- All of these functional interfaces fall into one of four categories: suppliers, consumers, predicates, or functions.
- The basic functional interfaces from this package are `Supplier`, `Consumer`, `Predicate`, and `Function`.
  - The functional method of `Supplier` is `get()`. It returns a value.
  - The functional method of `Consumer` is `accept()`. It takes an argument and returns no value.
  - The functional method of `Predicate` is `test()`. It takes an argument and returns a boolean.
  - The functional method of `Function` is `apply()`. It takes an argument and returns a value.

Using Functional Interfaces (OCP Objectives 4.1, 4.2, 4.3, 4.4)

- You can compose consumers with the `andThen()` default method.
- Use the same type of consumer when composing two consumers together.
- Perform logical tests with predicates using the default predicate methods `and()`, `or()`, and `negate()`.
- A `Predicate`'s static `isEqual()` method returns a `Predicate` that tests to see if two objects are equal. Note that this method is only available in the `Predicate` interface, not the predicate variations in `java.util.function`.
- Compose `Functions` together with the methods `andThen()` and `compose()`.
- Check carefully to see which functional interfaces support which default and static methods.
- The `forEach()` method in collection types, like `List`, takes a `Consumer` and allows you to easily iterate through the collection. The `Consumer`'s `accept()` method argument is the current object in the collection you are iterating over.
- The `replaceAll()` method in collection types, like `List`, takes a `UnaryOperator` and allows you to replace items in the `List` with different values, ones that could be based on the current values or completely new values.
- The built-in functional interfaces are conveniences so you don't have to create your own.
- The built-in functional interfaces are used in several ways in the Java 8 JDK, including with Streams (see Chapter 9).
- Creating your own functional interfaces is no different from creating any interface except you must make sure the interface has only one abstract method.
- Brush up on generics (Chapter 6) to make sure you know how to create a functional interface with generic types.

Develop Code That Uses Primitive Versions of Functional Interfaces (OCP Objective 4.2)

- Some variations of functional interfaces in the `java.util.package` are meant to handle primitive values to avoid autoboxing.
- `IntSupplier`'s functional method, `getAsInt()`, takes no arguments and returns an `int`. This is to avoid autoboxing the result as `Integer`, in case you need a primitive.
- Likewise, `DoubleSupplier`'s functional method, `getAsDouble()`, takes no arguments and returns a `double`.
- `IntConsumer`'s functional method, `accept()`, takes an `int` and does not return any value.
- `IntPredicate`'s functional method, `test()`, takes an `int` and returns a `boolean`.
- `IntFunction`'s functional method, `apply()`, takes an `int` and returns an object value.
- The functional method name in functional interfaces is not always the same (e.g., `IntSupplier` uses `getAsInt()` rather than `get()`, and `IntToLongFunction` uses `applyAsLong()` rather than `apply()`), so note the patterns of naming conventions for functional methods.

Develop Code That Uses Binary Versions of Functional Interfaces (OCP Objective 4.3)

- Some variations of functional interfaces in the `java.util.package` are meant to allow multiple arguments.
- `BiConsumer`'s functional method, `accept()`, takes two arguments and returns no value.
- `BiPredicate`'s functional method, `test()`, takes two arguments and returns a `boolean`.
- `BiFunction`'s functional method, `apply()`, takes two arguments and returns a value.
- The `forEach()` method in the `Map` collection type takes a `BiConsumer`, and the arguments of the `accept()` method are the key and value of

the current `Map` entry.

Develop Code That Uses the UnaryOperator Interface (OCP Objective 4.4)

- The operator functional interfaces in `java.util.function` are variations on the function interfaces (such as `Function`).
- Whereas `Function` takes a value of a type and returns a value of perhaps a different type, the `UnaryOperator` takes a value of a type and returns a value of that same type.

SELF TEST

The following questions will help you measure your understanding of the material in this chapter. If you don't get them all, go back and review and try again.

1. Which of these are functional interfaces? (Choose all that apply.)

A.

```
interface Question {
   default int answer() {
      return 42;
   }
}
```

B.

```
interface Tree {
   void grow();
}
```

C.

```
interface Book {
   static void read() {
      System.out.println("Turn the page...");
   }
}
```

D.

```
interface Flower {
   boolean equals(Object f);
   default void bloom() {
      System.out.println("Petals are opening");
   }
   String pick();
}
```

2. Given the following interface:

```
@FunctionalInterface
interface FtoC {
    double convert(double f);
}
```

Which of the following expressions are legal? (Choose all that apply.)

A. `FtoC converter = (f) -> (f - 32.0) * 5/9;`

B. `FtoC converter = f -> (f - 32.0) * 5/9;`

C. `FtoC converter = f -> return ((f - 32.0) * 5/9);`

D. `double converter = f -> (f - 32.0) * 5/9;`

E. `FtoC converter = f -> { return (f - 32.0) * 5/9; };`

3. Which of the following compiles correctly?

A. `Predicate<String> p = (s) -> System.out.println(s);`

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

C. `Supplier<String> s = (s) -> System.out.println(s);`

D. `Function<String> f = (s) -> System.out.println(s);`

4. Given the code fragment:

```
System.out.format("Total = %.2f", computeTax(10.00, (p) -> p * 0.05));
```

Which method would you use for `computeTax()` so the code fragment

prints `Total = 10.50`?

A.

```
double computeTax(double price, Function<Double> op) {
   return op.apply(price) + price;
}
```

B.

```
double computeTax(double price, UnaryOperator<Double> op) {
   return op.apply(price) + price;
}
```

C.

```
double computeTax(double price, double op) {
   return op.apply(price) + price;
}
```

D.

```
Function<Double, Double> computeTax(double price, UnaryOperator<Double> op) {
   return op.apply(price) + price;
}
```

5. Given:

```
class Reading {
   int year;
   int month;
   int day;
   double value;

   Reading(int year, int month, int day, double value) {
      this.year = year; this.month = month; this.day = day; this.value = value;
   }
}
```

and the code fragment:

```
List<Reading> readings = Arrays.asList(
            new Reading(2017, 1, 1, 405.91),
            new Reading(2017, 1, 8, 405.98),
            new Reading(2017, 1, 15, 406.14),
            new Reading(2017, 1, 22, 406.48),
            new Reading(2017, 1, 29, 406.20),
            new Reading(2017, 2, 5, 406.03));
```

Which code fragment will sort the readings in ascending order by `value` and print the `value` of each reading?

A.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);
readings.forEach(System.out.println(r.value));
```

B.

```
readings.sort((r1, r2) -> r1.value < r2.value ? 1 : -1);
readings.forEach(System.out::println(r.value));
```

C.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);
readings.forEach(System.out::println);
```

D.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);
readings.forEach(r -> System.out.println(r.value));
```

6. Given the code fragments:

```
class Human {
   public Integer age;
   public String name;

   public Human(Integer age, String name) {
      this.age = age;
      this.name = name;
   }
   public Integer getAge() { return this.age; }
}
```

and

```
Supplier<Human> human = () -> new Human(34, "Joe");
Human joe = human.XXXX;
```

Which code fragment inserted at `XXXX` will cause a new `Human` object to be stored in the variable `joe`?

A. `push()`

B. `get()`

C. `apply()`

D. `test()`

E. `accept()`

7. Given:

```
class Human {
   public Integer age;
   public String name;

   public Human(Integer age, String name) {
      this.age = age;
      this.name = name;
   }
   public Integer getAge() { return this.age; }
}
```

and the code fragment:

```
Human jenny = new Human(18, "Jenny");
Human jeff = new Human(17, "Jeff");
Human jill = new Human(21, "Jill");
List<Human> people = new ArrayList<>(Arrays.asList(jenny, jeff, jill));
// L1
people.forEach(printAdults);
```

Which code fragment inserted at line `// L1` will print the names of only adults (those humans whose age is older than 17)?

A.

```
Predicate<Human> printAdults = p -> { if (p.getAge() >= 18) {
   System.out.println(p.name);
}};
```

B.

```
Predicate<Human> adult = p -> p.getAge() >= 18;
Consumer<Human> printAdults = p -> { if (p.getAge(adult.test()) >= 18) {
   System.out.println(p.name);
}};
```

C.

```
Predicate<Human> adult = p -> p.getAge() >= 18;
Consumer<Human> printAdults = p -> { if (adult.test(p)) {
   System.out.println(p.name);
}};
```

D.

```
Consumer printAdults(Human p) {
   if (p.getAge() >= 18) {
      System.out.println(p.name);
   }
}
```

8. Given:

```
List<String> birds =
       Arrays.asList("eagle", "seagull", "albatross", "buzzard", "goose");
int longest = 0;
birds.forEach(b -> {              // L3
   if (b.length() > longest) {
     longest = b.length();        // L5
   }
});
System.out.println("Longest bird name is length: " + longest);
```

What is the result?

A. `"Longest bird name is length: 9"`
B. Compilation fails because of an error on line L5
C. Compilation fails because of an error on line L3
D. A runtime exception occurs on line L5

9. Given the following code fragment:

```
Supplier<List<Double>> readingsSupplier =
    Arrays.asList(405.91, 405.98, 406.14, 406.48, 406.20, 406.03);
for (Double r : readingsSupplier.get()) { System.out.print(r + " "); }
```

What is the result?

A. `405.91`
B. `405.91 405.98 406.14 406.48 406.2 406.03`
C. An exception is thrown at runtime
D. Compilation fails

10. Given the code fragment:

```
BiFunction<Integer, String, String> foo = (n, s) -> {
    String newString = "";
    for (int i = 0; i < n; i++) {
        newString = s + " " + newString;
    }
    return newString;
};
Function<String, String> bar = (s) -> s + "bar";
System.out.println(foo.andThen(bar).apply(3, "foo"));
```

What is the result?

A. `foo foo foo bar bar bar`

B. `foo foo foo bar`

C. `foo foo foo`

D. Compilation fails

11. Given the code fragment:

```
List<String> trees = Arrays.asList("FIR", "CEDAR", "PINE");
// L1
trees.replaceAll(convert);
trees.forEach(t -> System.out.print(t + " "));
```

Which fragment(s), inserted independently at `// L1`, produce the output? (Choose all that apply.)

```
fir cedar pine
```

A. `UnaryOperator<String> convert = (t) -> t.toLowerCase();`

B. `UnaryOperator<String, String> convert = (t) -> t.toLowerCase();`

C. `Function<String, String> convert = (t) -> t.toLowerCase();`

D. `Supplier<String> convert = (t) -> t.toLowerCase();`

12. Given the code fragment:

```
Map<String, String> todos = new HashMap<String, String>();
todos.put("monday", "wash dog");
todos.put("tuesday", "weed yard");
// L1
todos.forEach(printTodo);
```

Which fragment(s), inserted independently at `// L1`, produce the to-do items, both key and value, in the `Map`? (Choose all that apply.)

A.

```
Function<String, String> printTodo =
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

B.

```
Consumer<String, String> printTodo =
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

C.

```
Consumer<Map> printTodo =
    (Map m) -> System.out.println("On " + m.keySet() + "do: " + m.values());
```

D.

```
BiConsumer<String, String> printTodo =
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

13. Given the code fragments:

```
class TodoList {
   public void checkTodoDay(Map<String, String> todos, Predicate<String> isDay) {
      todos.forEach((d, t) -> {
         if (isDay.test(d)) {
            System.out.println("You really should do this today! " + t);
         }
      });
   }
}
```

and

```
Map<String, String> todos = new HashMap<String, String>();
todos.put("monday", "wash dog");
todos.put("tuesday", "weed yard");
TodoList todoList = new TodoList();
// L1
```

Which fragment(s), inserted independently at `// L1`, display the output consisting of the to-do item for Tuesday? (Choose all that apply.)

A.

```
todoList.checkTodoDay(todos,
    (k) -> if (k.equals("tuesday")) return true; else return false; );
```

B.

```
todoList.checkTodoDay(todos, (k, v) -> {
   if (k.equals("tuesday")) {
      System.out.println("You really should do this today! " + v);
   }
});
```

C.

```
todoList.checkTodoDay(todos, (k) -> k.equals("tuesday"));
```

D.

```
todoList.checkTodoDay(todos, (k) -> k.test("tuesday"));
```

14. Given the code fragment:

```
DoubleSupplier d = () -> 42.0;
// L1
```

Which fragment(s), inserted independently at `// L1`, produce the output

(Choose all that apply.)

```
The answer is: 42.0
```

A.

```
System.out.println("The answer is: " + d.get());
```

B.

```
double answer = d.getAsDouble();
System.out.println("The answer is: " + answer);
```

C.

```
System.out.println("The answer is: " + d.getAsDouble());
```

D.

```
double answer = d.get();
System.out.println("The answer is: " + answer);
```

A SELF TEST ANSWERS

1. ☑ **B** and **D** are correct. For **B**, we just have one abstract method, so this is a functional interface. Although **D** includes the `equals()` method (implying that classes implementing this interface must implement an

`equals()` method), this method doesn't count because it's inherited from `Object`. The default method `bloom()` is default and so doesn't count; the method `pick()` is the functional method.

☒ **A** is incorrect because the interface has no functional method (one abstract method). A default method is not a functional method. Likewise, **C** is incorrect because the interface has no functional method (one abstract method). A static method is not a functional method. (OCP Objective 4.1)

2. ☑ **A, B,** and **E** are correct variations on writing lambda expressions.

   ☒ **C** and **D** are incorrect. **C** is invalid syntax for lambda expressions, with a return statement that's not enclosed in `{ }`. **D** is incorrect because it has the wrong type for the lambda expression. (OCP Objective 2.6)

3. ☑ **B** is correct. The lambda expression is a consumer, so only the type `Consumer` is correct for this lambda expression—the functional method takes an argument and returns nothing.

   ☒ **A, C,** and **D** are incorrect based on the above. (OCP Objective 4.1)

4. ☑ **B** is correct. `computeTax()` is a method that takes two arguments, a `double` and a `UnaryOperator`, and returns a `double` value. We know that the lambda expression is a `UnaryOperator` because the functional method takes a `Double` and returns a `Double` (with autoboxing and autounboxing).

   ☒ **A, C,** and **D** are incorrect. **A** could almost work because a `UnaryOperator` is a type of `Function`; however, a correct declaration of the `Function` would specify the type of both the argument and the return value, because unlike `UnaryOperator`, they are not necessarily the same type. (OCP Objective 4.4)

5. ☑ **D** is correct. The `sort()` method of the `List` requires a `Comparator`, which can be expressed as a lambda expression implementing the `compare()` functional method. This method must return a –1 or 1 (for items that are not equal) depending on the ordering; because we want ascending order, we test if object 1 is less than object 2 and return –1 if so; otherwise, 1 is returned to get ascending order. Because we are comparing the values of the `reading` objects, we use the `value` field of each `reading` object to make the comparison. In the `forEach`, we supply

a `Consumer`, whose functional method takes a `reading` object and prints the `value` field of that `reading`. **D** properly supplies a `Consumer` to `forEach` that prints just the `values`.

☒ **A, B,** and **C** are incorrect. **A** is incorrect because we don't supply a `Consumer` for the `forEach`. **B** is incorrect because we are using invalid syntax for the method reference. Here we can't use a method reference because we are printing the `reading` values, not the whole `reading` object, and so must specify more details about the argument to the `Consumer`'s functional method than allowed with a method reference, which is why **C** is incorrect. **C** prints the entire `reading` object, rather than just the `value`. (OCP Objectives 3.3, 3.5, and 4.1)

6. ☑ **B** is correct. `human` is a `Supplier`, so its functional method is `get()`. Calling `get()` returns a new `Human` with the name Joe and the age 34.

☒ **A, C, D,** and **E** are incorrect based on the above. (OCP Objective 4.1)

7. ☑ **C** is correct. We know `printAdults` must be a `Consumer` that prints adults in the `people` `ArrayList`. The functional method of the `Consumer` will take a `Human` object. Then we must test that object to see if the `Human` is 18 or older, which we can do by defining a `Predicate` whose functional method returns true if the `Human`'s `age` is 18 or older. We use the `Predicate` by calling the function method, `test()`. If the `Human` passes the test, we print the `Human`'s `name`.

☒ **A, B,** and **D** are incorrect. **A** is incorrect because we haven't defined the `printAdults` as a `Consumer`. **B** is incorrect because, although we are defining the `printAdults` `Consumer` in the body of the `Consumer`'s functional method, we are using the `Predicate` wrong. **D** can initially be tempting, but if you look carefully, the syntax is incorrect. Either we must implement the `Consumer` interface with an inner class or with a lambda expression. (OCP Objective 4.1)

8. ☑ **B** is correct. The code does not compile because the variable `longest` from the lambda's enclosing scope must be final or effectively final, and we are trying to change the value of `longest` within the lambda.

☒ **A, C,** and **D** are incorrect for the above reasons. (OCP Objective 2.6)

9. ☑ **D** is correct. The code does not compile because `readingsSupplier`

is not being assigned a `Supplier`; rather it is being assigned a `List`.

☒ **A, B,** and **C** are incorrect for the above reasons. (OCP Objective 4.1)

10. ☑ **B** is correct. In the last line of the code, we are first calling the `apply()` method of the `foo BiFunction` and then calling the `apply()` method of the `bar Function`. Looking at the first line of code, we see that `foo`'s `apply()` method is implemented by the lambda expression and takes two arguments, an `integer n` and a `String`. The method concatenates the `String n` times (with a space between) and then returns it. So by calling `apply()` with the arguments `3` and `"foo"`, the `String "foo foo foo"` is returned. Then we call the `apply()` method of `bar`. The `andThen()` method of the `foo BiFunction` passes the value returned from the first `BiFunction` to the `Function` whose `apply()` method is called, so `"foo foo foo"` gets passed to the `apply()` method of `bar`, which is implemented with a lambda expression and takes a `String`, concatenates `"bar"` to that `String`, and returns it, resulting in `"foo foo foo bar"`. (Note, too, this is an example of how you can combine a `BiFunction` with a `Function` using `andThen()`!)

☒ **A, C,** and **D** are incorrect for the above reasons. (OCP Objectives 4.1 and 4.3)

11. ☑ **A** is correct. The `List replaceAll()` method takes a `UnaryOperator`. In this case, the `UnaryOperator`'s functional method takes a `String` and returns the lowercase value of that `String`. Because it's a `UnaryOperator`, we need only specify one type parameter.

☒ **B, C,** and **D** are incorrect. **B** looks like it might be correct if you forget that `UnaryOperator` uses only one type parameter. **C** is tempting because a `UnaryOperator` is a type of `Function`, but `replaceAll()` specifies a `UnaryOperator` as an argument. **D** can't work because a `Supplier`'s functional method does not take an argument. (OCP Objective 4.4)

12. ☑ **D** is correct. To use `forEach()` with a `Map`, we need a `BiConsumer`, whose functional method takes two arguments, both `Strings`.

☒ **A, B,** and **C** are incorrect for the above reasons. (OCP Objective 4.3)

13. ☑ **C** is correct. To print the to-do item for Tuesday, we need to call the to-do list's `checkTodoDay()` method, passing the list of to-dos and a

`Predicate`. We can see in the body of the `checkTodoDay()` method that we call the `Predicate`'s `test()` method on the map key, which is the day, a `String`, and if the test passes, we display the value in the map. So the `Predicate` should test to see if the day is equal to `"tuesday"` and return true if it is.

☒ **A, B,** and **D** are incorrect. **A** could work, but the syntax is incorrect (we should use `{ }` for statements in the body of the lambda) and will cause a compile error. **B** is incorrect because the `Predicate`'s `test()` method doesn't take two values; it takes only one, the key (day). **D** is incorrect because the `test()` method is not valid for a `String`; we need `equals()` instead. (OCP Objective 4.1)

14. ☑ **B** and **C** are correct. To get a `double` value from a `DoubleSupplier`, you must use the `getAsDouble()` functional method.

☒ **A** and **D** are incorrect. **A** and **D** are incorrectly using `get()` instead of `getAsDouble()`. (OCP Objective 4.2)