# Chapter 6 Lambdas and Functional Interfaces

**OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:**

- **Programming Abstractly Through Interfaces**
  - Declare and use List and ArrayList instances
  - Understanding Lambda Expressions

When we covered the Java APIs in the previous chapter, we didn't cover the ones that use lambda syntax. This chapter remedies that! You'll learn what a lambda is used for, about common functional interfaces, how to write a lambda with variables, and the APIs on the exam that rely on lambdas.

## Writing Simple Lambdas

Java is an object-oriented language at heart. You've seen plenty of objects by now. In Java 8, the language added the ability to write code using another style.

*Functional programming* is a way of writing code more declaratively. You specify what you want to do rather than dealing with the state of objects.

You focus more on expressions than loops.

Functional programming uses lambda expressions to write code. A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an unnamed method. It has parameters and a body just like full-fledged methods do, but it doesn't have a name like a real method. Lambda expressions are often referred to as *lambdas* for short. You might also know them as closures if Java isn't your first language. If you had a bad experience with closures in the past, don't worry. They are far simpler in Java.

In other words, a lambda expression is like a method that you can pass as if it were a variable. For example, there are different ways to calculate age. One human year is equivalent to seven dog years. You want to write a method that takes an `age()` method as input. To do this in an object-oriented program, you'd need to define a `Human` subclass and a `Dog` subclass. With lambdas, you can just pass in the relevant expression to calculate age.

Lambdas allow you to write powerful code in Java. Only the simplest lambda expressions are on this exam. The goal is to get you comfortable with the syntax and the concepts. You'll see lambdas again on the 1Z0-816 exam.

In this section, we'll cover an example of why lambdas are helpful and the syntax of lambdas.

## Lambda Example

Our goal is to print out all the animals in a list according to some criteria. We'll show you how to do this without lambdas to illustrate how lambdas are useful. We start out with the `Animal` class:

```
public class Animal {
   private String species;
```

```
    private boolean canHop;
    private boolean canSwim;
    public Animal(String speciesName, boolean hopper, boolean swimmer){
       species = speciesName;
       canHop = hopper;
       canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
 }
```

The `Animal` class has three instance variables, which are set in the constructor. It has two methods that get the state of whether the animal can hop or swim. It also has a `toString()` method so we can easily identify the `Animal` in programs.

We plan to write a lot of different checks, so we want an interface. You'll learn more about interfaces in Chapter 9, "Advanced Class Design." For now, it is enough to remember that an interface specifies the methods that our class needs to implement:

```
public interface CheckTrait {
   boolean test(Animal a);
}
```

The first thing we want to check is whether the `Animal` can hop. We provide a class that can check this:

```
public class CheckIfHopper implements CheckTrait {
   public boolean test(Animal a) {
      return a.canHop();
   }
}
```

This class may seem simple—and it is. This is actually part of the problem that lambdas solve. Just bear with us for a bit. Now we have everything that we need to write our code to find the `Animal` s that hop:

```
1:  import java.util.*;
2:  public class TraditionalSearch {
3:     public static void main(String[] args) {
4:
5:        // list of animals
6:        List<Animal> animals = new ArrayList<Animal>();
7:        animals.add(new Animal("fish", false, true));
8:        animals.add(new Animal("kangaroo", true, false));
9:        animals.add(new Animal("rabbit", true, false));
10:       animals.add(new Animal("turtle", false, true));
11:
12:       // pass class that does check
13:       print(animals, new CheckIfHopper());
14:    }
15:    private static void print(List<Animal> animals,
16:       CheckTrait checker) {
17:       for (Animal animal : animals) {
18:
19:          // the general check
20:          if (checker.test(animal))
21:             System.out.print(animal + " ");
22:       }
23:       System.out.println();
24:    }
25: }
```

The `print()` method on line 13 method is very general—it can check for any trait. This is good design. It shouldn't need to know what specifically we are searching for in order to print a list of animals.

Now what happens if we want to print the `Animal` s that swim? Sigh. We need to write another class, `CheckIfSwims` . Granted, it is only a few lines.

Then we need to add a new line under line 13 that instantiates that class. That's two things just to do another check.

Why can't we just specify the logic we care about right here? Turns out that we can with lambda expressions. We could repeat that whole class here and make you find the one line that changed. Instead, we'll just show you. We could replace line 13 with the following, which uses a lambda:

```
13:    print(animals, a -> a.canHop());
```

Don't worry that the syntax looks a little funky. You'll get used to it, and we'll describe it in the next section. We'll also explain the bits that look like magic. For now, just focus on how easy it is to read. We are telling Java that we only care about `Animal` s that can hop.

It doesn't take much imagination to figure out how we would add logic to get the `Animal` s that can swim. We only have to add one line of code—no need for an extra class to do something simple. Here's that other line:

```
print(animals, a -> a.canSwim());
```

How about `Animal` s that cannot swim?

```
print(animals, a -> ! a.canSwim());
```

The point here is that it is really easy to write code that uses lambdas once you get the basics in place. This code uses a concept called deferred execution. *Deferred execution* means that code is specified now but will run later. In this case, later is when the `print()` method calls it.

## Lambda Syntax

One of the simplest lambda expressions you can write is the one you just saw:

```
a -> a.canHop()
```

Lambdas work with interfaces that have only one abstract method. In this case, Java looks at the `CheckTrait` interface that has one method. The lambda indicates that Java should call a method with an `Animal` parameter that returns a `boolean` value that's the result of `a.canHop()`. We know all this because we wrote the code. But how does Java know?

Java relies on context when figuring out what lambda expressions mean. We are passing this lambda as the second parameter of the `print()` method. That method expects a `CheckTrait` as the second parameter. Since we are passing a lambda instead, Java tries to map our lambda to that interface:

```
boolean test(Animal a);
```

Since that interface's method takes an `Animal`, that means the lambda parameter has to be an `Animal`. And since that interface's method returns a `boolean`, we know the lambda returns a `boolean`.

The syntax of lambdas is tricky because many parts are optional. These two lines do the exact same thing:

```
a -> a.canHop()

(Animal a) -> { return a.canHop(); }
```

Let's look at what is going on here. The first example, shown in Figure 6.1, has three parts:

- A single parameter specified with the name `a`

- The arrow operator to separate the parameter and body
- A body that calls a single method and returns the result of that method

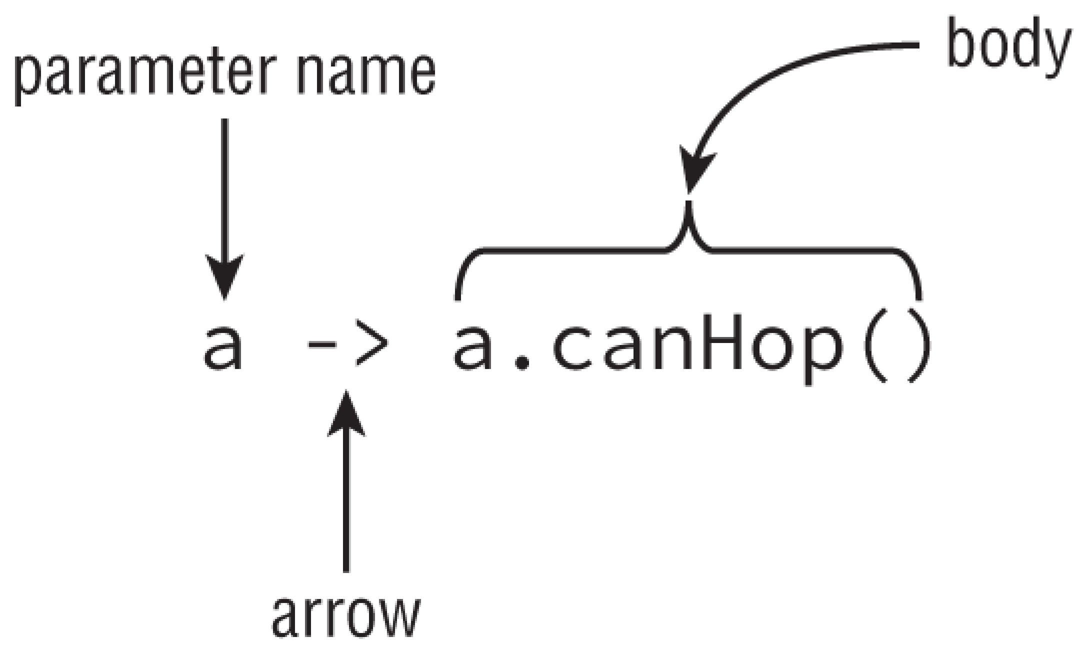

**FIGURE 6.1 Lambda syntax omitting optional parts**

The second example shows the most verbose form of a lambda that returns a `boolean` (see Figure 6.2):

- A single parameter specified with the name `a` and stating the type is `Animal`
- The arrow operator to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a `return` statement

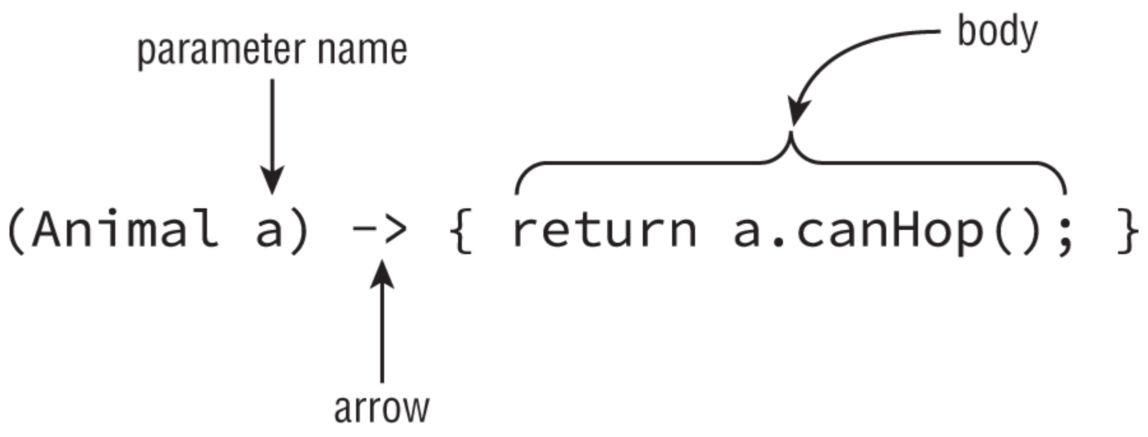

**FIGURE 6.2 Lambda syntax, including optional parts**

The parentheses can be omitted only if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do as little typing as possible.

It shouldn't be news to you that we can omit braces when we have only a single statement. We did this with `if` statements and loops already. What is different here is that the rules change when you omit the braces. Java doesn't require you to type `return` or use a semicolon when no braces are used. This special shortcut doesn't work when we have two or more statements. At least this is consistent with using `{}` to create blocks of code elsewhere.

---

**NOTE**

Here's a fun fact: `s -> {}` is a valid lambda. If there is no code on the right side of the expression, you don't need the semicolon or `return` statement.

---

Table 6.1 shows examples of valid lambdas that return a `boolean`.

**TABLE 6.1** **Valid lambdas**

| Lambda | # parameters |
|---|---|
| `() -> true` | 0 |
| `a -> a.startsWith("test")` | 1 |
| `(String a) -> a.startsWith("test")` | 1 |
| `(a, b) -> a.startsWith("test")` | 2 |
| `(String a, String b) -> a.startsWith("test")` | 2 |

Notice that all of these examples have parentheses around the parameter list except the one that takes only one parameter and doesn't specify the type. The first row takes zero parameters and always returns the `boolean` value `true`. The second row takes one parameter and calls a method on it, returning the result. The third row does the same except that it explicitly defines the type of the variable. The final two rows take two parameters and ignore one of them—there isn't a rule that says you must use all defined parameters.

Now let's make sure you can identify invalid syntax for each row in Table 6.2 where each is supposed to return a `boolean`. Make sure you understand what's wrong with each of these.

**TABLE 6.2** **Invalid lambdas that return `boolean`**

| Invalid lambda | Reason |
|---|---|
| `a, b -> a.startsWith("test")` | Missing parentheses |
| `a -> { a.startsWith("test"); }` | Missing `return` |
| `a -> { return a.startsWith("test") }` | Missing semicolon |

Remember that the parentheses are optional *only* when there is one parameter and it doesn't have a type declared.

## Introducing Functional Interfaces

In our earlier example, we created an interface with one method:

```
boolean test(Animal a);
```

Lambdas work with interfaces that have only one abstract method. These are called *functional interfaces*. (It's actually more complicated than this, but for this exam the simplified definition is fine. On the 1Z0-816 exam, you'll get to deal with the full definition of a functional interface.)

We mentioned that a functional interface has only one abstract method. Your friend Sam can help you remember this because it is officially known as a S*ingle Abstract Method (SAM)* rule.

NOTE

Java provides an annotation `@FunctionalInterface` on some, but not all, functional interfaces. This annotation means the authors of the interface promise it will be safe to use in a lambda in the future. However, just because you don't see the annotation doesn't mean it's not a functional interface. Remember that having exactly one abstract method is what makes it a functional interface, not the annotation.

There are four functional interfaces you are likely to see on the exam. The next sections take a look at `Predicate` , `Consumer` , `Supplier` , and `Comparator` .

## Predicate

You can imagine that we'd have to create lots of interfaces like this to use lambdas. We want to test `Animal` s and `String` s and `Plant` s and anything else that we come across.

Luckily, Java recognizes that this is a common problem and provides such an interface for us. It's in the package `java.util.function` and the gist of it is as follows:

```
public interface Predicate<T> {
   boolean test(T t);
}
```

That looks a lot like our `test(Animal)` method. The only difference is that it uses the type `T` instead of `Animal` . That's the syntax for generics. It's like when we created an `ArrayList` and got to specify any type that goes in it.

This means we don't need our own interface anymore and can put everything related to our search in one class:

```
1:  import java.util.*;
2:  import java.util.function.*;
3:  public class PredicateSearch {
4:     public static void main(String[] args) {
5:        List<Animal> animals = new ArrayList<Animal>();
6:        animals.add(new Animal("fish", false, true));
7:
8:        print(animals, a -> a.canHop());
9:     }
10:    private static void print(List<Animal> animals,
11:       Predicate<Animal>  checker) {
12:       for (Animal animal : animals) {
13:          if (checker.test(animal))
14:             System.out.print(animal + " ");
15:       }
16:       System.out.println();
17:    }
18: }
```

This time, line 11 is the only one that changed. We expect to have a `Predicate` passed in that uses type `Animal`. Pretty cool. We can just use it without having to write extra code.

## Consumer

The `Consumer` functional interface has one method you need to know:

```
void accept(T t)
```

Why might you want to receive a value and not return it? A common reason is when printing a message:

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

We've declared functionality to print out the value we were given. It's okay that we don't have a value yet. When the consumer is called, the value will be provided and printed then. Let's take a look at code that uses a `Consumer`:

```
public static void main(String[] args) {
   Consumer<String> consumer = x -> System.out.println(x);
   print(consumer, "Hello World");
}
private static void print(Consumer<String> consumer, String value) {
   consumer.accept(value);
}
```

This code prints `Hello World`. It's a more complicated version than the one you learned as your first program. The `print()` method accepts a `Consumer` that knows how to print a value. When the `accept()` method is called, the lambda actually runs, printing the value.

## Supplier

The `Supplier` functional interface has only one method:

```
T get()
```

A good use case for a `Supplier` is when generating values. Here are two examples:

```
Supplier<Integer> number = () ->  42;
Supplier<Integer> random = () ->  new Random().nextInt();
```

The first example returns `42` each time the lambda is called. The second generates a random number each time it is called. It could be the same number but is likely to be a different one. After all, it's random. Let's take a look at code that uses a `Supplier`:

```
public static void main(String[] args) {
   Supplier<Integer> number = () ->  42;
   System.out.println(returnNumber(number));
}

private static int returnNumber(Supplier<Integer> supplier) {
   return supplier.get();
}
```

When the `returnNumber()` method is called, it invokes the lambda to get the desired value. In this case, the method returns `42`.

### Comparator

In Chapter 5, “Core Java APIs,” we compared numbers. We didn’t supply a `Comparator` because we were using the default sort order. We did learn the rules. A negative number means the first value is smaller, zero means they are equal, and a positive number means the first value is bigger. The method signature is as follows:

```
int compare(T o1, T o2)
```

This interface is a functional interface since it has only one unimplemented method. It has many `static` and `default` methods to facilitate writing complex comparators.

---

NOTE

The `Comparator` interface existed prior to lambdas being added to Java. As a result, it is in a different package. You can find `Comparator` in `java.util`.

---

You only have to know `compare()` for the exam. Can you figure out whether this sorts in ascending or descending order?

```
Comparator<Integer> ints = (i1, i2) -> i1 - i2;
```

The `ints` comparator uses natural sort order. If the first number is bigger, it will return a positive number. Try it. Suppose we are comparing `5` and `3`. The comparator subtracts `5-3` and gets `2`. This is a positive number that means the first number is bigger and we are sorting in ascending order.

Let's try another one. Do you think these two statements would sort in ascending or descending order?

```
Comparator<String> strings = (s1, s2) -> s2.compareTo(s1);
Comparator<String> moreStrings = (s1, s2) -> - s1.compareTo(s2);
```

Both of these comparators actually do the same thing: sort in descending order. In the first example, the call to `compareTo()` is "backwards," making it descending. In the second example, the call uses the default order; however, it applies a negative sign to the result, which reverses it.

Be sure you understand Table 6.3 to identify what type of lambda you are looking at.

**TABLE 6.3** **Basic functional interfaces**

| Functional interface | # parameters | Return type |
|---|---|---|
| `Comparator` | Two | `int` |
| `Consumer` | One | `void` |
| `Predicate` | One | `boolean` |
| `Supplier` | None | One (type varies) |

## Working with Variables in Lambdas

Variables can appear in three places with respect to lambdas: the parameter list, local variables declared inside the lambda body, and variables referenced from the lambda body. All three of these are opportunities for the exam to trick you. We will explore each one so you'll be alert when tricks show up!

### Parameter List

Earlier in this chapter, you learned that specifying the type of parameters is optional. Additionally, `var` can be used in place of the specific type. That means that all three of these statements are interchangeable:

```
Predicate<String> p = x -> true;
Predicate<String> p = (var x) -> true;
Predicate<String> p = (String x) -> true;
```

The exam might ask you to identify the type of the lambda parameter. In our example, the answer is `String`. How did we figure that out? A lambda infers the types from the surrounding context. That means you get to do the same.

In this case, the lambda is being assigned to a `Predicate` that takes a `String`. Another place to look for the type is in a method signature. Let's try another example. Can you figure out the type of `x`?

```
public void whatAmI() {
   consume((var x) -> System.out.print(x), 123);
}
public void consume(Consumer<Integer> c, int num) {
   c.accept(num);
}
```

If you guessed `Integer`, you were right. The `whatAmI()` method creates a lambda to be passed to the `consume()` method. Since the `consume()` method expects an `Integer` as the generic, we know that is what the inferred type of `x` will be.

But wait; there's more. In some cases, you can determine the type without even seeing the method signature. What do you think the type of `x` is here?

```
   public void counts(List<Integer> list) {
      list.sort((var x, var y) -> x.compareTo(y));
   }
```

The answer is again `Integer`. Since we are sorting a list, we can use the type of the list to determine the type of the lambda parameter.

## Local Variables inside the Lambda Body

While it is most common for a lambda body to be a single expression, it is legal to define a block. That block can have anything that is valid in a normal Java block, including local variable declarations.

The following code does just that. It creates a local variable named `c` that is scoped to the lambda block.

```
(a, b) -> { int c = 0; return 5;}
```

---

NOTE

When writing your own code, a lambda block with a local variable is a good hint that you should extract that code into a method.

---

Now let's try another one. Do you see what's wrong here?

```
(a, b) -> { int a = 0; return 5;}     // DOES NOT COMPILE
```

We tried to redeclare `a`, which is not allowed. Java doesn't let you create a local variable with the same name as one already declared in that scope. Now let's try a hard one. How many syntax errors do you see in this method?

```
11: public void variables(int a) {
12:    int b = 1;
13:    Predicate<Integer> p1 = a -> {
14:       int b = 0;
15:       int c = 0;
16:       return b == c;}
17: }
```

There are three syntax errors. The first is on line 13. The variable `a` was already used in this scope as a method parameter, so it cannot be reused. The next syntax error comes on line 14 where the code attempts to redeclare local variable `b`. The third syntax error is quite subtle and on line 16. See it? Look really closely.

The variable `p1` is missing a semicolon at the end. There is a semicolon before the `}`, but that is inside the block. While you don't normally have to look for missing semicolons, lambdas are tricky in this space, so beware!

## Variables Referenced from the Lambda Body

Lambda bodies are allowed to reference some variables from the surrounding code. The following code is legal:

```
public class Crow {
   private String color;
   public void caw(String name) {
      String volume = "loudly";
      Consumer<String> consumer = s ->
            System.out.println(name + " says "
                  + volume + " that she is " + color);
   }
}
```

This shows that lambda can access an instance variable, method parameter, or local variable under certain conditions. Instance variables (and class variables) are always allowed.

Method parameters and local variables are allowed to be referenced if they are *effectively final*. This means that the value of a variable doesn't change after it is set, regardless of whether it is explicitly marked as `final`. If you aren't sure whether a variable is effectively final, add the `final` keyword. If the code would still compile, the variable is effectively final. You can think of it as if we had written this:

```
public class Crow {
   private String color;
   public void caw(final String name) {
      final String volume = "loudly";
      Consumer<String> consumer = s ->
         System.out.println(name + " says "
            + volume + " that she is " + color);
   }
}
```

It gets even more interesting when you look at where the compiler errors occur when the variables are not effectively final.

```
2:  public class Crow {
3:     private String color;
4:     public void caw(String name) {
5:        String volume = "loudly";
6:        name = "Caty";
7:        color = "black";
8:
9:        Consumer<String> consumer = s ->
10:          System.out.println(name + " says "
11:             + volume + " that she is " + color);
12:       volume = "softly";
13:    }
14: }
```

In this example, `name` is not effectively final because it is set on line 6. However, the compiler error occurs on line 10. It's not a problem to assign a value to a nonfinal variable. However, once the lambda tries to use it, we do have a problem. The variable is no longer effectively final, so the lambda is not allowed to use the variable.

The variable `volume` is not effectively final either since it is updated on line 12. In this case, the compiler error is on line 11. That's before the assignment! Again, the act of assigning a value is only a problem from the point of view of the lambda. Therefore, the lambda has to be the one to generate the compiler error.

To review, make sure you've memorized Table 6.4.

**TABLE 6.4 Rules for accessing a variable from a lambda body inside a method**

| Variable type | Rule |
|---|---|
| Instance variable | Allowed |
| Static variable | Allowed |
| Local variable | Allowed if effectively final |
| Method parameter | Allowed if effectively final |
| Lambda parameter | Allowed |

## Calling APIs with Lambdas

Now that you are familiar with lambdas and functional interfaces, we can look at the most common methods that use them on the exam. The 1Z0-816 will cover streams and many more APIs that use lambdas.

### *removeIf()*

`List` and `Set` declare a `removeIf()` method that takes a `Predicate`. Imagine we have a list of names for pet bunnies. We decide we want to remove all of the bunny names that don't begin with the letter `h` because our little cousin really wants us to choose an `h` name. We could solve this problem by writing a loop. Or we could solve it in one line:

```
List<String> bunnies = new ArrayList<>();
bunnies.add("long ear");
bunnies.add("floppy");
bunnies.add("hoppy");
```

```
7: System.out.println(bunnies);     // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies);     // [hoppy]
```

Line 8 takes care of everything for us. It defines a predicate that takes a `String` and returns a `boolean`. The `removeIf()` method does the rest.

The `removeIf()` method works the same way on a `Set`. It removes any values in the set that match the `Predicate`. There isn't a `removeIf()` method on a `Map`. Remember that maps have both keys and values. It wouldn't be clear what one was removing!

### *sort()*

While you can call `Collections.sort(list)`, you can now sort directly on the list object.

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);     // [long ear, floppy, hoppy]
8: bunnies.sort((b1, b2) -> b1.compareTo(b2));
9: System.out.println(bunnies);     // [floppy, hoppy, long ear]
```

On line 8, we sort the list alphabetically. The `sort()` method takes `Comparator` that provides the sort order. Remember that `Comparator` takes two parameters and returns an `int`. If you need a review of what the return value of a `compare()` operation means, check the `Comparator` section in this chapter or the Comparing section in Chapter 5. This is really important to memorize!

There is not a sort method on `Set` or `Map`. Neither of those types has indexing, so it wouldn't make sense to sort them.

### *forEach()*

Our final method is `forEach()`. It takes a `Consumer` and calls that lambda for each element encountered.

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7:
8: bunnies.forEach(b -> System.out.println(b));
9: System.out.println(bunnies);
```

This code prints the following:

```
long ear
floppy
hoppy
[long ear, floppy, hoppy]
```

The method on line 8 prints one entry per line. The method on line 9 prints the entire list on one line.

We can use `forEach()` with a `Set` or `Map`. For a `Set`, it works the same way as a `List`.

```
Set<String> bunnies = Set.of("long ear", "floppy", "hoppy");
bunnies.forEach(b -> System.out.println(b));
```

For a `Map`, you have to choose whether you want to go through the keys or values:

```
Map<String, Integer> bunnies =  new HashMap<>();
bunnies.put("long ear", 3);
bunnies.put("floppy", 8);
bunnies.put("hoppy", 1);
bunnies.keySet().forEach(b -> System.out.println(b));
bunnies.values().forEach(b -> System.out.println(b));
```

It turns out the `keySet()` and `values()` methods each return a `Set`. Since we know how to use `forEach()` with a `Set`, this is easy!

---

**Real World Scenario**

**USING *FOREACH()* WITH A MAP DIRECTLY**

You don't need to know this for the exam, but Java has a functional interface called `BiConsumer`. It works just like `Consumer` except it can take two parameters. This functional interface allows you to use `forEach()` with key/value pairs from `Map`.

```
Map<String, Integer> bunnies = new HashMap<>();
bunnies.put("long ear", 3);
bunnies.put("floppy", 8);
bunnies.put("hoppy", 1);
bunnies.forEach((k, v) -> System.out.println(k + " " + v));
```

---

## Summary

Lambda expressions, or lambdas, allow passing around blocks of code. The full syntax looks like this:

```
(String a, String b) -> { return a.equals(b); }
```

The parameter types can be omitted. When only one parameter is specified without a type the parentheses can also be omitted. The braces and `return` statement can be omitted for a single statement, making the short form as follows:

```
a -> a.equals(b)
```

Lambdas are passed to a method expecting an instance of a functional interface.

A functional interface is one with a single abstract method. `Predicate` is a common interface that returns a `boolean` and takes any type. `Consumer` takes any type and doesn't return a value. `Supplier` returns a value and does not take any parameters. `Comparator` takes two parameters and returns an `int`.

A lambda can define parameters or variables in the body as long as their names are different from existing local variables. The body of a lambda is allowed to use any instance or class variables. Additionally, it can use any local variables or method parameters that are effectively final.

We covered three common APIs that use lambdas. The `removeIf()` method on a `List` and a `Set` takes a `Predicate`. The `sort()` method on a `List` interface takes a `Comparator`. The `forEach()` methods on a `List` and a `Set` interface both take a `Consumer`.

## Exam Essentials

**Write simple lambda expressions.** Look for the presence or absence of optional elements in lambda code. Parameter types are optional. Braces and the `return` keyword are optional when the body is a single statement. Parentheses are optional when only one parameter is specified and the type is implicit.

**Identify common functional interfaces.** From a code snippet, identify whether the lambda is a `Comparator`, `Consumer`, `Predicate`, or `Supplier`. You can use the number of parameters and return type to tell them apart.

**Determine whether a variable can be used in a lambda body.** Local variables and method parameters must be effectively final to be referenced. This means the code must compile if you were to add the `final` keyword to these variables. Instance and class variables are always allowed.

**Use common APIs with lambdas.** Be able to read and write code using `forEach()`, `removeIf()`, and `sort()`.

## Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What is the result of the following class?

```
1:  import java.util.function.*;
2:
3:  public class Panda {
4:     int age;
5:     public static void main(String[] args) {
6:        Panda p1 = new Panda();
7:        p1.age = 1;
8:        check(p1, p -> p.age < 5);
9:     }
10:    private static void check(Panda panda,
11:       Predicate<Panda> pred) {
12:       String result =
13:          pred.test(panda) ? "match" : "not match";
14:       System.out.print(result);
15: } }
```

   1. `match`
   2. `not match`
   3. Compiler error on line 8.
   4. Compiler error on lines 10 and 11.

5. Compiler error on lines 12 and 13.
6. A runtime exception is thrown.

2. What is the result of the following code?

```
1:  interface Climb {
2:     boolean isTooHigh(int height, int limit);
3:  }
4:
5:  public class Climber {
6:     public static void main(String[] args) {
7:        check((h, m) -> h.append(m).isEmpty(), 5);
8:     }
9:     private static void check(Climb climb, int height) {
10:       if (climb.isTooHigh(height, 10))
11:          System.out.println("too high");
12:       else
13:          System.out.println("ok");
14:    }
15: }
```

1. `ok`
2. `too high`
3. Compiler error on line 7.
4. Compiler error on line 10.
5. Compiler error on a different line.
6. A runtime exception is thrown.

3. Which of the following lambda expressions can fill in the blank? (Choose all that apply.)

```
List<String> list = new ArrayList<>();
list.removeIf(_________________);
```

1. `s -> s.isEmpty()`
2. `s -> {s.isEmpty()}`
3. `s -> {s.isEmpty();}`
4. `s -> {return s.isEmpty();}`

5. `String s -> s.isEmpty()`
6. `(String s) -> s.isEmpty()`

4. Which lambda can replace the `MySecret` class to return the same value? (Choose all that apply.)

```
interface Secret {
   String magic(double d);
}

class MySecret implements Secret {
   public String magic(double d) {
      return "Poof";
   }
}
```

   1. `(e) -> "Poof"`
   2. `(e) -> {"Poof"}`
   3. `(e) -> { String e = ""; "Poof" }`
   4. `(e) -> { String e = ""; return "Poof"; }`
   5. `(e) -> { String e = ""; return "Poof" }`
   6. `(e) -> { String f = ""; return "Poof"; }`

5. Which of the following lambda expressions can be passed to a function of `Predicate<String>` type? (Choose all that apply.)
   1. `() -> s.isEmpty()`
   2. `s -> s.isEmpty()`
   3. `String s -> s.isEmpty()`
   4. `(String s) -> s.isEmpty()`
   5. `(s1) -> s.isEmpty()`
   6. `(s1, s2) -> s1.isEmpty()`

6. Which of these statements is true about the following code?

```
public void method() {
   x((var x) -> {}, (var x, var y) -> 0);
}
public void x(Consumer<String> x, Comparator<Boolean> y) {
}
```

   1. `The code does not compile because of one of the variables named x.`
   2. `The code does not compile because of one of the variables named y.`
   3. `The code does not compile for another reason.`
   4. The code compiles, and the `var` in each lambda refers to the same type.
   5. The code compiles, and the `var` in each lambda refers to a different type.
7. Which of the following will compile when filling in the blank? (Choose all that apply.)

```
List list = List.of(1, 2, 3);
Set set = Set.of(1, 2, 3);
Map map = Map.of(1, 2, 3, 4);

_______.forEach(x -> System.out.println(x));
```

   1. `list`
   2. `set`
   3. `map`
   4. `map.keys()`
   5. `map.keySet()`
   6. `map.values()`
   7. `map.valueSet()`
8. Which statements are true?
   1. The `Consumer` interface is best for printing out an existing value.
   2. The `Supplier` interface is best for printing out an existing value.
   3. The `Comparator` interface returns an `int`.
   4. The `Predicate` interface returns an `int`.
   5. The `Comparator` interface has a method named `test()`.
   6. The `Predicate` interface has a method named `test()`.
9. Which of the following can be inserted without causing a compilation error? (Choose all that apply.)

```
public void remove(List<Character> chars) {
   char end = 'z';
   chars.removeIf(c -> {
      char start = 'a'; return start <= c && c <= end; });
      // INSERT LINE HERE
}
```

1. `char start = 'a';`
2. `char c = 'x';`
3. `chars = null;`
4. `end = '1';`
5. None of the above

10. How many lines does this code output?

```
Set<String> set = Set.of("mickey", "minnie");
List<String> list = new ArrayList<>(set);

set.forEach(s -> System.out.println(s));
list.forEach(s -> System.out.println(s));
```

1. 0
2. 2
3. 4
4. The code does not compile.
5. A runtime exception is thrown.

11. What is the output of the following code?

```
List<String> cats = new ArrayList<>();
cats.add("leo");
cats.add("Olivia");

cats.sort((c1, c2) -> -c1.compareTo(c2)); // line X
System.out.println(cats);
```

1. [leo, Olivia]

2. `[Olivia, leo]`
3. `The code does not compile because of line X.`
4. `The code does not compile for another reason.`
5. `A runtime exception is thrown.`

12. Which pieces of code can fill in the blanks? (Choose all that apply.)

```
_______________ first = () -> Set.of(1.23);
_______________ second = x -> true;
```

1. `Consumer<Set<Double>>`
2. `Consumer<Set<Float>>`
3. `Predicate<Set<Double>>`
4. `Predicate<Set<Float>>`
5. `Supplier<Set<Double>>`
6. `Supplier<Set<Float>>`

13. Which is true of the following code?

```
int length = 3;

for (int i = 0; i<3; i++) {
   if (i%2 == 0) {
      Supplier<Integer> supplier = () -> length; // A
      System.out.println(supplier.get());        // B
   } else {
      int j = i;
      Supplier<Integer> supplier = () -> j;      // C
      System.out.println(supplier.get());        // D
   }
}
```

1. `The first compiler error is on line A.`
2. `The first compiler error is on line B.`
3. `The first compiler error is on line C.`
4. `The first compiler error is on line D.`
5. `The code compiles successfully.`

14. Which of the following can be inserted without causing a compilation error? (Choose all that apply.)

```
public void remove(List<Character> chars) {
   char end = 'z';
   // INSERT LINE HERE
   chars.removeIf(c -> {
      char start = 'a'; return start <= c && c <= end; });
}
```

   1. `char start = 'a';`
   2. `char c = 'x';`
   3. `chars = null;`
   4. `end = '1';`
   5. `None of the above`

15. What is the output of the following code?

```
Set<String> cats = new HashSet<>();
cats.add("leo");
cats.add("Olivia");

cats.sort((c1, c2) -> -c1.compareTo(c2)); // line X
System.out.println(cats);
```

   1. `[leo, Olivia]`
   2. `[Olivia, leo]`
   3. `The code does not compile because of line X.`
   4. `The code does not compile for another reason.`
   5. `A runtime exception is thrown.`

16. Which variables are effectively final? (Choose all that apply.)

```
public void isIt(String param1, String param2) {
   String local1 = param1 + param2;
   String local2 = param1 + param2;

   param1 = null;
```

```
      local2 = null;
   }
```

1. `local1`
2. `local2`
3. `param1`
4. `param2`
5. `None of the above`

17. What is the result of the following class?

```
1:  import java.util.function.*;
2:
3:  public class Panda {
4:     int age;
5:     public static void main(String[] args) {
6:        Panda p1 = new Panda();
7:        p1.age = 1;
8:        check(p1, p -> {p.age < 5});
9:     }
10:    private static void check(Panda panda,
11:       Predicate<Panda> pred) {
12:       String result = pred.test(panda)
13:          ? "match" : "not match";
14:       System.out.print(result);
15: } }
```

1. `match`
2. `not match`
3. Compiler error on line 8.
4. Compiler error on line 10.
5. Compile error on line 12.
6. A runtime exception is thrown.

18. How many lines does this code output?

```
Set<String> s = Set.of("mickey", "minnie");
List<String> x = new ArrayList<>(s);
```

```
s.forEach(s -> System.out.println(s));
x.forEach(x -> System.out.println(x));
```

1. `0`
2. `2`
3. `4`
4. `The code does not compile.`
5. `A runtime exception is thrown.`

19. Which lambda can replace the `MySecret` class? (Choose all that apply.)

```
interface Secret {
   String concat(String a, String b);
}

class MySecret implements Secret {
   public String concat(String a, String b) {
      return a + b;
   }
}
```

1. `(a, b) -> a + b`
2. `(String a, b) -> a + b`
3. `(String a, String b) -> a + b`
4. `(a, b) , a + b`
5. `(String a, b) , a + b`
6. `(String a, String b) , a + b`

20. Which of the following lambda expressions can be passed to a function of `Predicate<String>` type? (Choose all that apply.)
    1. `s -> s.isEmpty()`
    2. `s --> s.isEmpty()`
    3. `(String s) -> s.isEmpty()`
    4. `(String s) --> s.isEmpty()`
    5. `(StringBuilder s) -> s.isEmpty()`

6. `(StringBuilder s) --> s.isEmpty()`