Chapter 8
Lambdas and Functional Interfaces

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Using Object-Oriented Concepts in Java**
  - Understand variable scopes, apply encapsulation, and create immutable objects. Use local variable type inference.
  - Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods.

In this chapter, we start by introducing lambdas, a new piece of syntax. Lambdas allow you to specify code that will be run later in the program.

Next, we introduce the concept of functional interfaces, showing how to write your own and identify whether an interface is a functional interface. After that, we introduce another new piece of syntax: method references. These are like a shorter form of lambdas.

Then we introduce the functional interfaces you need to know for the exam. Finally, we emphasize how variables fit into lambdas.

Lambdas, method references, and functional interfaces are used quite a bit in Chapter 9, "Collections and Generics," and Chapter 10, "Streams."

Writing Simple Lambdas

Java is an object-oriented language at heart. You've seen plenty of objects by now. *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 existing inside an anonymous class like the ones you saw in Chapter 7, “Beyond Classes.” 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.

Lambdas allow you to write powerful code in Java. In this section, we cover an example of why lambdas are helpful and the syntax of lambdas.

Looking at a Lambda Example

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

```
public record Animal(String species, boolean canHop, boolean
canSwim) { }
```

The `Animal` record has three fields. Let’s say we have a list of animals, and we want to process the data based on a particular attribute. For example, we want to print all animals that can hop. We can define an interface to generalize this concept and support a large variety of checks.

```
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 implements our interface.

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

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

```
1:  import java.util.*;
2:  public class TraditionalSearch {
3:     public static void main(String[] args) {
4:
5:        // list of animals
6:        var 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,
CheckTrait checker) {
16:       for (Animal animal : animals) {
17:
18:          // General check
19:          if (checker.test(animal))
20:             System.out.print(animal + " ");
21:       }
22:       System.out.println();
23:    }
24: }
```

Line 6 shows configuring an `ArrayList` with a specific type of `Animal`. The `print()` method on line 15 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.

What happens if we want to print the `Animals` that swim? Sigh. We need to write another class, `CheckIfSwims`. Granted, it is only a few lines, but it is a whole new file. 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 specify the logic we care about right here? It turns out that we can, with lambda expressions. We could repeat the whole class here and make you find the one line that changed. Instead, we just show you that we can keep our `print()` method declaration unchanged. Let's 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 describe it in the next section. We 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 if an `Animal` 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:

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

How about `Animal`s that cannot swim?

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

The point 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*, which means that code is specified now but will run later. In this case, “later” is inside the `print()` method body, as opposed to when it is passed to the method.

Learning 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 exactly one abstract method. In this case, Java looks at the `CheckTrait` interface, which has one method. The lambda in our example suggests 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. Context refers to where and how the lambda is interpreted. For example, if we see someone in line to enter the zoo and they have their wallet out, it is fair to assume they want to buy zoo tickets. Alternatively, if they are in the concession line with their wallet out, they are probably hungry.

Referring to our earlier example, we passed the lambda as the second parameter of the print `method()`.

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

The `print()` method expects a `CheckTrait` as the second parameter.

```
private static void print(List<Animal> animals, CheckTrait checker) { … }
```

Since we are passing a lambda instead, Java tries to map our lambda to the abstract method declaration in the `CheckTrait` interface.

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

Since that interface's method takes an `Animal`, 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 8.1, 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

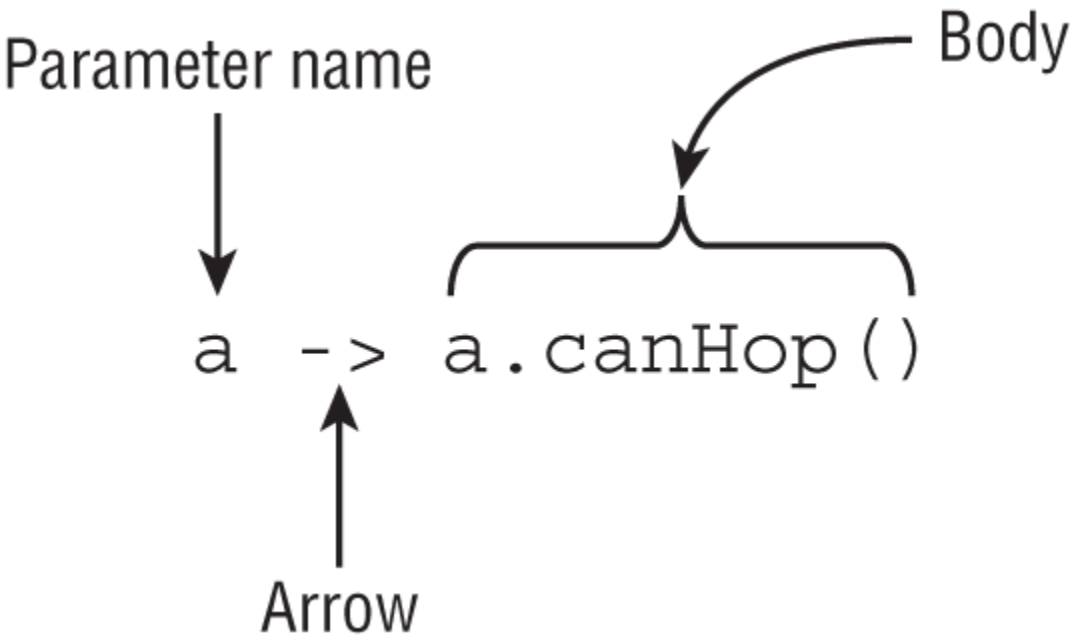

FIGURE 8.1 Lambda syntax omitting optional parts

The second example shows the most verbose form of a lambda that returns a `boolean` (see Figure 8.2).

- A single parameter specified with the name `a` and stating that 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

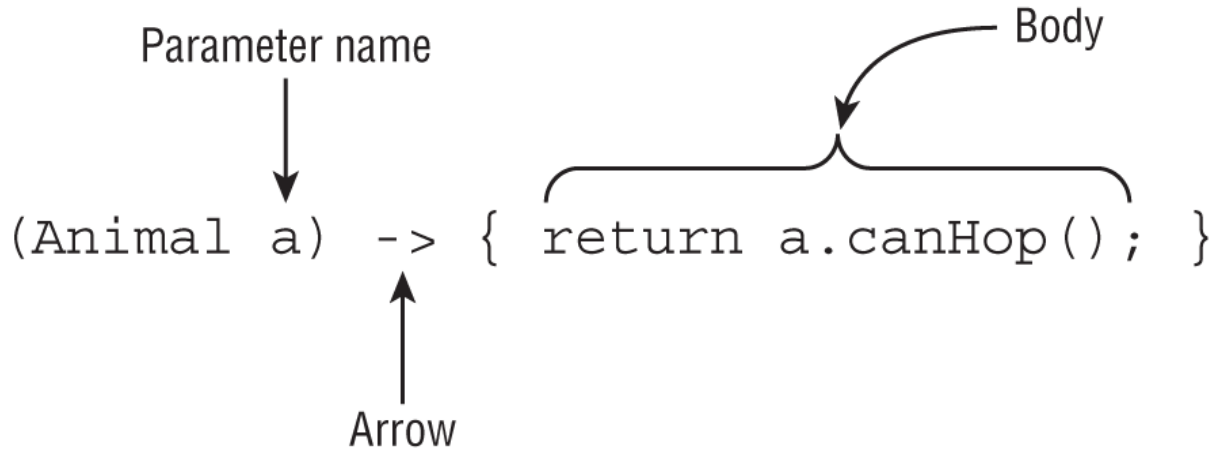

FIGURE 8.2 Lambda syntax including optional parts

The parentheses around the lambda parameters 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 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. Java allows you to omit a `return` statement and semicolon (`;`) when no braces are used. This special shortcut doesn't work when you have two or more statements. At least this is consistent with using `{}` to create blocks of code elsewhere.

The syntax in Figure 8.1 and Figure 8.2 can be mixed and matched. For example, the following are valid:

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

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 8.1 shows examples of valid lambdas that return a `boolean`.

TABLE 8.1 Valid lambdas that return a `boolean`

Lambda	# of parameters
`() -> true`	0
`x -> x.startsWith("test")`	1
`(String x) -> x.startsWith("test")`	1
`(x, y) -> { return x.startsWith("test"); }`	2
`(String x, String y) -> x.startsWith("test")`	2

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 8.2, where each lambda is supposed to return a `boolean`. Make sure you understand what's wrong with these.

TABLE 8.2 Invalid lambdas that should return a `boolean`

Invalid lambda	Reason
`x, y -> x.startsWith("fish")`	Missing parentheses on left
`x -> { x.startsWith("camel"); }`	Missing `return` on right
`x -> { return x.startsWith("giraffe") }`	Missing semicolon inside braces
`String x -> x.endsWith("eagle")`	Missing parentheses on left

Remember that the parentheses are optional *only* when there is one parameter and it doesn't have a type declared. Those are the basics of writing a lambda. At the end of the chapter, we cover additional rules about using variables in a lambda.

Assigning Lambdas to *var*

Why do you think this line of code doesn't compile?

```
var invalid = (Animal a) -> a.canHop();  // DOES NOT COMPILE
```

Remember when we talked about Java inferring information about the lambda from the context? Well, `var` assumes the type based on the context as well. There's not enough context here! Neither the lambda nor `var` have enough information to determine what type of functional interface should be used.

Coding Functional Interfaces

Earlier in the chapter, we declared the `CheckTrait` interface, which has exactly one method for implementers to write. Lambdas have a special relationship with such interfaces. In fact, these interfaces have a name. A *functional interface* is an interface that contains a single abstract method. Your friend Sam can help you remember this because it is officially known as a *single abstract method (SAM)* rule.

Defining a Functional Interface

Let's take a look at an example of a functional interface and a class that implements it:

```
@FunctionalInterface
public interface Sprint {
   public void sprint(int speed);
}

public class Tiger implements Sprint {
   public void sprint(int speed) {
      System.out.println("Animal is sprinting fast! " + speed);
   }
}
```

In this example, the `Sprint` interface is a functional interface because it contains exactly one abstract method, and the `Tiger` class is a valid class that implements the interface.

The *@FunctionalInterface* Annotation

The `@FunctionalInterface` annotation tells the compiler that you intend for the code to be a functional interface. If the interface does not follow the rules for a functional interface, the compiler will give you an error.

```
@FunctionalInterface  // DOES NOT COMPILE
public interface Dance {
   void move();
   void rest();
}
```

Java includes `@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.

Consider the following four interfaces. Given our previous `Sprint` functional interface, which of the following are functional interfaces?

```
public interface Dash extends Sprint {}

public interface Skip extends Sprint {
   void skip();
}

public interface Sleep {
   private void snore() {}
   default int getZzz() { return 1; }
}

public interface Climb {
```

```
   void reach();
   default void fall() {}
   static int getBackUp() { return 100; }
   private static boolean checkHeight() { return true; }
}
```

All four of these are valid interfaces, but not all of them are functional interfaces. The `Dash` interface is a functional interface because it extends the `Sprint` interface and inherits the single abstract method `sprint()`. The `Skip` interface is not a valid functional interface because it has two abstract methods: the inherited `sprint()` method and the declared `skip()` method.

The `Sleep` interface is also not a valid functional interface. Neither `snore()` nor `getZzz()` meets the criteria of a single abstract method. Even though `default` methods function like abstract methods, in that they can be overridden in a class implementing the interface, they are insufficient for satisfying the single abstract method requirement.

Finally, the `Climb` interface is a functional interface. Despite defining a slew of `static`, `private`, and `default` methods, it contains only one abstract method: `reach()`.

Adding Object Methods

All classes inherit certain methods from `Object`. For the exam, you should know the following `Object` method signatures:

- `public String toString()`
- `public boolean equals(Object)`
- `public int hashCode()`

We bring this up now because there is one exception to the single abstract method rule that you should be familiar with. If a functional interface includes an abstract method with the same signature as a `public` method found in `Object`, *those methods do not count toward the single abstract method test*. The motivation behind this rule is that any class that implements the interface will inherit from `Object`, as all classes do, and therefore always implement these methods.

Since Java assumes all classes extend from `Object`, you also cannot declare an interface method that is incompatible with `Object`. For example, declaring an abstract method `int toString()` in an interface would not compile since `Object`'s version of the method returns a `String`.

Let's take a look at an example. Is the `Soar` class a functional interface?

```
public interface Soar {
   abstract String toString();
}
```

It is not. Since `toString()` is a `public` method implemented in `Object`, it does not count toward the single abstract method test. On the other hand, the following implementation of `Dive` is a functional interface:

```
public interface Dive {
   String toString();
   public boolean equals(Object o);
   public abstract int hashCode();
   public void dive();
}
```

The `dive()` method is the single abstract method, while the others are not counted since they are `public` methods defined in the `Object` class.

Be wary of examples that resemble methods in the `Object` class but are not actually defined in the `Object` class. Do you see why the following is not a valid functional interface?

```
public interface Hibernate {
   String toString();
   public boolean equals(Hibernate o);
   public abstract int hashCode();
   public void rest();
}
```

Despite looking a lot like our `Dive` interface, the `Hibernate` interface uses `equals(Hibernate)` instead of `equals(Object)`. Because this does not match the method signature of the `equals(Object)` method defined in the `Object` class, this interface is counted as containing two abstract methods: `equals(Hibernate)` and `rest()`.

Using Method References

Method references are another way to make the code easier to read, such as simply mentioning the name of the method. Like lambdas, it takes time to get used to the new syntax. In this section, we show the syntax along with the four types of method references. We also mix in lambdas with method references.

Suppose we are coding a duckling that is trying to learn how to quack. First we have a functional interface:

```
public interface LearnToSpeak {
   void speak(String sound);
}
```

Next, we discover that our duckling is lucky. There is a helper class that the duckling can work with. We've omitted the details of teaching the duckling how to quack and left the part that calls the functional interface:

```
public class DuckHelper {
   public static void teacher(String name, LearnToSpeak
learner) {
      // Exercise patience (omitted)
      learner.speak(name);
   }
}
```

Finally, it is time to put it all together and meet our little `Duckling`. This code implements the functional interface using a lambda:

```
public class Duckling {
   public static void makeSound(String sound) {
      LearnToSpeak learner = s -> System.out.println(s);
      DuckHelper.teacher(sound, learner);
   }
}
```

Not bad. There's a bit of redundancy, though. The lambda declares one parameter named `s`. However, it does nothing other than pass that parameter to another method. A method reference lets us remove that redundancy and instead write this:

```
LearnToSpeak learner = System.out::println;
```

The `::` operator tells Java to call the `println()` method later. It will take a little while to get used to the syntax. Once you do, you may find your code is shorter and less distracting without writing as many lambdas.

Remember that `::` is like a lambda, and it is used for deferred execution with a functional interface. You can even imagine the method reference as a lambda if it helps you.

A method reference and a lambda behave the same way at runtime. You can pretend the compiler turns your method references into lambdas for you.

There are four formats for method references.

- `static` methods
- Instance methods on a particular object
- Instance methods on a parameter to be determined at runtime
- Constructors

Let's take a brief look at each of these in turn. In each example, we show the method reference and its lambda equivalent. For now, we create a separate functional interface for each example. In the next section, we introduce built-in functional interfaces so you don't have to keep writing your own.

Calling *static* Methods

For this first example, we use a functional interface that converts a `double` to a `long`:

```
interface Converter {
   long round(double num);
}
```

We can implement this interface with the `round()` method in `Math`. Here we assign a method reference and a lambda to this functional interface:

```
14: Converter methodRef = Math::round;
15: Converter lambda = x -> Math.round(x);
16:
17: System.out.println(methodRef.round(100.1));  // 100
```

On line 14, we reference a method with one parameter, and Java knows that it's like a lambda with one parameter. Additionally, Java knows to pass that parameter to the method.

Wait a minute. You might be aware that the `round()` method is overloaded—it can take a `double` or a `float`. How does Java know that we want to call the version with a `double`? With both lambdas and method references, Java infers information from the *context*. In this case, we said that we were declaring a `Converter`, which has a method taking a `double` parameter. Java looks for a method that matches that description. If it can't find it or finds multiple matches, then the compiler will report an error. The latter is sometimes called an *ambiguous* type error.

Calling Instance Methods on a Particular Object

For this example, our functional interface checks if a `String` starts with a specified value.

```
interface StringStart {
   boolean beginningCheck(String prefix);
}
```

Conveniently, the `String` class has a `startsWith()` method that takes one parameter and returns a `boolean`. Let's look at how to use method references with this code:

```
18: var str = "Zoo";
19: StringStart methodRef = str::startsWith;
```

```
20: StringStart lambda = s -> str.startsWith(s);
21:
22: System.out.println(methodRef.beginningCheck("A"));  //
false
```

Line 19 shows that we want to call `str.startsWith()` and pass a single parameter to be supplied at runtime. This would be a nice way of filtering the data in a list.

A method reference doesn't have to take any parameters. In this example, we create a functional interface with a method that doesn't take any parameters but returns a value:

```
interface StringChecker {
   boolean check();
}
```

We implement it by checking if the `String` is empty.

```
18: var str = "";
19: StringChecker methodRef = str::isEmpty;
20: StringChecker lambda = () -> str.isEmpty();
21:
22: System.out.print(methodRef.check());  // true
```

Since the method on `String` is an instance method, we call the method reference on an instance of the `String` class.

While all method references can be turned into lambdas, the opposite is not always true. For example, consider this code:

```
var str = "";
StringChecker lambda = () -> str.startsWith("Zoo");
```

How might we write this as a method reference? You might try one of the following:

```
StringChecker methodReference = str::startsWith;         //
DOES NOT COMPILE

StringChecker methodReference = str::startsWith("Zoo");  //
DOES NOT COMPILE
```

Neither of these works! While we can pass the `str` as part of the method reference, there's no way to pass the `"Zoo"` parameter with it. Therefore, it

is not possible to write this lambda as a method reference.

Calling Instance Methods on a Parameter

This time, we are going to call the same instance method that doesn't take any parameters. The trick is that we will do so without knowing the instance in advance. We need a different functional interface this time since it needs to know about the `String`.

```
interface StringParameterChecker {
   boolean check(String text);
}
```

We can implement this functional interface as follows:

```
23: StringParameterChecker methodRef = String::isEmpty;
24: StringParameterChecker lambda = s -> s.isEmpty();
25:
26: System.out.println(methodRef.check("Zoo"));  // false
```

Line 23 says the method that we want to call is declared in `String`. It looks like a `static` method, but it isn't. Instead, Java knows that `isEmpty()` is an instance method that does not take any parameters. Java uses the parameter supplied at runtime as the instance on which the method is called.

Compare lines 23 and 24 with lines 19 and 20 of our instance example. They look similar, although one references a local variable named `str`, while the other only references the functional interface parameters.

You can even combine the two types of instance method references. Again, we need a new functional interface that takes two parameters.

```
interface StringTwoParameterChecker {
   boolean check(String text, String prefix);
}
```

Pay attention to the parameter order when reading the implementation.

```
26: StringTwoParameterChecker methodRef = String::startsWith;
27: StringTwoParameterChecker lambda = (s, p) ->
s.startsWith(p);
28:
29: System.out.println(methodRef.check("Zoo", "A"));  // false
```

Since the functional interface takes two parameters, Java has to figure out what they represent. The first one will always be the instance of the object for instance methods. Any others are to be method parameters.

Remember that line 26 may look like a `static` method, but it is really a method reference declaring that the instance of the object will be specified later. Line 27 shows some of the power of a method reference. We were able to replace two lambda parameters this time.

Calling Constructors

A *constructor reference* is a special type of method reference that uses `new` instead of a method and instantiates an object. For this example, our functional interface will not take any parameters but will return a `String`.

```
interface EmptyStringCreator {
   String create();
}
```

To call this, we use `new` as if it were a method name.

```
30: EmptyStringCreator methodRef = String::new;
31: EmptyStringCreator lambda = () -> new String();
32:
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
```

It expands like the method references you have seen so far. In the previous example, the lambda doesn't have any parameters.

Method references can be tricky. This time we create a functional interface that takes one parameter and returns a result:

```
interface StringCopier {
   String copy(String value);
}
```

In the implementation, notice that line 32 in the following example has the same method reference as line 30 in the previous example:

```
32: StringCopier methodRef = String::new;
33: StringCopier lambda = x -> new String(x);
34:
```

```
35: var myString = methodRef.copy("Zebra");
36: System.out.println(myString.equals("Zebra"));  // true
```

This means you can't always determine which method can be called by looking at the method reference. Instead, you have to look at the context to see what parameters are used and if there is a return type. In this example, Java sees that we are passing a `String` parameter and calls the constructor of `String` that takes such a parameter.

Reviewing Method References

Reading method references is helpful in understanding the code. Table 8.3 shows the four types of method references. If this table doesn't make sense, please reread the previous section. It can take a few tries before method references start to add up.

TABLE 8.3 Method references

Type	Before colon	After colon	Example
`static` methods	Class name	Method name	`Math::random`
Instance methods on a particular object	Instance variable name	Method name	`str::startsWith`
Instance methods on a parameter	Class name	Method name	`String::isEmpty`
Constructor	Class name	`New`	`String::new`

Working with Built-in Functional Interfaces

It would be inconvenient to write your own functional interface any time you want to write a lambda. Luckily, a large number of general-purpose functional interfaces are provided for you. We cover them in this section.

The core functional interfaces in Table 8.4 are provided in the `java.util.function` package. We cover generics in the next chapter, but for now, you just need to know that `<T>` allows the interface to take an object of a specified type. If a second type parameter is needed, we use the

next letter, `U`. If a distinct return type is needed, we choose `R` for *return* as the generic type.

TABLE 8.4 Common functional interfaces

Functional interface	Return type	Method name	# of parameters
`Supplier<T>`	`T`	`get()`	0
`Consumer<T>`	`void`	`accept(T)`	1 `(T)`
`BiConsumer<T, U>`	`void`	`accept(T,U)`	2 `(T, U)`
`Predicate<T>`	`boolean`	`test(T)`	1 `(T)`
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 `(T, U)`
`Function<T, R>`	`R`	`apply(T)`	1 `(T)`
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 `(T, U)`
`UnaryOperator<T>`	`T`	`apply(T)`	1 `(T)`
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 `(T, T)`

For the exam, you need to memorize Table 8.4. We will give you lots of practice in this section to help make it memorable. Before you ask, most of the time we don't assign the implementation of the interface to a variable. The interface name is implied, and it is passed directly to the method that needs it. We are introducing the names so that you can better understand and remember what is going on. By the next chapter, we will assume that you have this down and stop creating the intermediate variable.

You learn about a few more functional interfaces later in the book. In the next chapter, we cover `Comparator`. In Chapter 13, "Concurrency," we discuss `Runnable` and `Callable`. These may show up on the exam when you are asked to recognize functional interfaces.

Let’s look at how to implement each of these interfaces. Since both lambdas and method references appear all over the exam, we show an implementation using both where possible. After introducing the interfaces, we also cover some convenience methods available on these interfaces.

Implementing *Supplier*

A `Supplier` is used when you want to generate or supply values without taking any input. The `Supplier` interface is defined as follows:

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

You can create a `LocalDate` object using the factory method `now()`. This example shows how to use a `Supplier` to call this factory:

```
Supplier<LocalDate> s1 = LocalDate::now;
Supplier<LocalDate> s2 = () -> LocalDate.now();

LocalDate d1 = s1.get();
LocalDate d2 = s2.get();

System.out.println(d1);  // 2025-02-20
System.out.println(d2);  // 2025-02-20
```

This example prints a date twice. It’s also a good opportunity to review `static` method references. The `LocalDate::now` method reference is used to create a `Supplier` to assign to an intermediate variable `s1`. A `Supplier` is often used when constructing new objects. For example, we can print two empty `StringBuilder` objects.

```
Supplier<StringBuilder> s1 = StringBuilder::new;
Supplier<StringBuilder> s2 = () -> new StringBuilder();

System.out.println(s1.get());  // Empty string
System.out.println(s2.get());  // Empty string
```

This time, we used a constructor reference to create the object. We’ve been using generics to declare what type of `Supplier` we are using. This can be a little long to read. Can you figure out what the following does? Just take it one step at a time.

```
Supplier<ArrayList<String>> s3 = ArrayList::new;
ArrayList<String> a1 = s3.get();
System.out.println(a1);  // []
```

We have a `Supplier` of a certain type. That type happens to be `ArrayList<String>`. Then calling `get()` creates a new instance of `ArrayList<String>`, which is the generic type of the `Supplier`—in other words, a generic that contains another generic. Be sure to look at the code carefully when this type of thing comes up.

Notice how we called `get()` on the functional interface. What would happen if we tried to print out `s3` itself?

```
System.out.println(s3);
```

The code prints something like this:

```
functionalinterface.BuiltIns$$Lambda$1/0x0000000800066840@4909b
8da
```

That's the result of calling `toString()` on a lambda. Yuck. This actually does mean something. Our test class is named `BuiltIns`, and it is in a package that we created named `functionalinterface`. Then comes `$$`, which means the class doesn't exist in a class file on the file system. It exists only in memory. You don't need to worry about the rest.

Implementing *Consumer* and *BiConsumer*

You use a `Consumer` when you want to do something with a parameter but not return anything. `BiConsumer` does the same thing, except that it takes two parameters. The interfaces are defined as follows:

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
   // omitted default method
}

@FunctionalInterface
public interface BiConsumer<T, U> {
   void accept(T t, U u);
   // omitted default method
}
```

You'll notice this pattern. *Bi* means two. It comes from Latin, but you can remember it from English words like *binary* (0 or 1) or *bicycle* (two wheels). The interface method will always take two inputs when you see *Bi*.

Printing is a common use of the `Consumer` interface.

```
Consumer<String> c1 = System.out::println;
Consumer<String> c2 = x -> System.out.println(x);

c1.accept("Annie");  // Annie
c2.accept("Annie");  // Annie
```

`BiConsumer` is called with two parameters. They don't have to be the same type. For example, we can put a key and a value in a map using this interface:

```
var map = new HashMap<String, Integer>();
BiConsumer<String, Integer> b1 = map::put;
BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);

b1.accept("chicken", 7);
b2.accept("chick", 1);

System.out.println(map);  // {chicken=7, chick=1}
```

The output is `{chicken=7, chick=1}`, which shows that both `BiConsumer` implementations were called. When declaring `b1`, we used an instance method reference on an object since we want to call a method on the local variable `map`. The code to instantiate `b1` is a good bit shorter than the code for `b2`. This is probably why the exam is so fond of method references.

As another example, we use the same type for both generic parameters:

```
var map = new HashMap<String, String>();
BiConsumer<String, String> b1 = map::put;
BiConsumer<String, String> b2 = (k, v) -> map.put(k, v);
```

```
b1.accept("chicken", "Cluck");
b2.accept("chick", "Tweep");

System.out.println(map);  // {chicken=Cluck, chick=Tweep}
```

This shows that a `BiConsumer` can use the same type for both the `T` and `U` generic parameters.

Implementing *Predicate* and *BiPredicate*

`Predicate` is often used when filtering or matching. Both are common operations. A `BiPredicate` is just like a `Predicate`, except that it takes two parameters instead of one. The interfaces are defined as follows:

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

@FunctionalInterface
public interface BiPredicate<T, U> {
   boolean test(T t, U u);
   // omitted default methods
}
```

You can use a `Predicate` to test a condition.

```
Predicate<String> p1 = String::isEmpty;
Predicate<String> p2 = x -> x.isEmpty();

System.out.println(p1.test(""));  // true
System.out.println(p2.test(""));  // true
```

This prints `true` twice. More interesting is a `BiPredicate`. This example also prints `true` twice:

```
BiPredicate<String, String> b1 = String::startsWith;
BiPredicate<String, String> b2 =
   (string, prefix) -> string.startsWith(prefix);

System.out.println(b1.test("chicken", "chick"));  // true
System.out.println(b2.test("chicken", "chick"));  // true
```

The method reference includes both the instance variable and parameter for `startsWith()`. This is a good example of how method references save quite a lot of typing. The downside is that they are less explicit, and you really have to understand what is going on!

Implementing *Function* and *BiFunction*

A `Function` is responsible for turning one parameter into a value of a potentially different type and returning it. Similarly, a `BiFunction` is responsible for turning two parameters into a value and returning it. The interfaces are defined as follows:

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
   // omitted default and static methods
}

@FunctionalInterface
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
   // omitted default method
}
```

For example, this function converts a `String` to the length of the `String`:

```
Function<String, Integer> f1 = String::length;
Function<String, Integer> f2 = x -> x.length();

System.out.println(f1.apply("cluck"));  // 5
System.out.println(f2.apply("cluck"));  // 5
```

This function turns a `String` into an `Integer`. Well, technically, it turns the `String` into an `int`, which is autoboxed into an `Integer`. The types don't have to be different. The following combines two `String` objects and produces another `String`:

```
BiFunction<String, String, String> b1 = String::concat;
BiFunction<String, String, String> b2 =
   (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick"));  // baby chick
System.out.println(b2.apply("baby ", "chick"));  // baby chick
```

The first two types in the `BiFunction` are the input types. The third is the result type. For the method reference, the first parameter is the instance that `concat()` is called on, and the second is passed to `concat()`.

Implementing *UnaryOperator* and *BinaryOperator*

`UnaryOperator` and `BinaryOperator` are special cases of a `Function`. They require all type parameters to be the same type. A `UnaryOperator` transforms its value into one of the same type. For example, incrementing by one is a unary operation. In fact, `UnaryOperator` extends `Function`. A `BinaryOperator` merges two values into one of the same type. Adding two numbers is a binary operation. Similarly, `BinaryOperator` extends `BiFunction`. The interfaces are defined as follows:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
   // omitted static method
}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T>
{
   // omitted static methods
}
```

This means the method signatures look like this:

```
T apply(T t);         // UnaryOperator

T apply(T t1, T t2);  // BinaryOperator
```

In the Javadoc, you'll notice that these methods are inherited from the `Function`/`BiFunction` superclass. The generic declarations on the subclass are what force the type to be the same. For the unary example, notice how the return type is the same type as the parameter.

```
UnaryOperator<String> u1 = String::toUpperCase;
UnaryOperator<String> u2 = x -> x.toUpperCase();

System.out.println(u1.apply("chirp"));  // CHIRP
System.out.println(u2.apply("chirp"));  // CHIRP
```

This prints `CHIRP` twice. We don't need to specify the return type in the generics because `UnaryOperator` requires it to be the same as the parameter. And now here's the binary example:

```
BinaryOperator<String> b1 = String::concat;
BinaryOperator<String> b2 = (string, toAdd) ->
string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick"));  // baby chick
System.out.println(b2.apply("baby ", "chick"));  // baby chick
```

Notice that this does the same thing as the `BiFunction` example. The code is more succinct, which shows the importance of using the best functional interface. It's nice to have one generic type specified instead of three.

Checking Functional Interfaces

It's really important to know the number of parameters, types, return value, and method name for each of the functional interfaces. Now would be a good time to memorize Table 8.4 if you haven't done so already. Let's do some examples to practice.

What functional interface would you use in these three situations?

- Returns a `String` without taking any parameters
- Returns a `Boolean` and takes a `String`
- Returns an `Integer` and takes two `Integers`

Ready? Think about what your answers are before continuing. Really. You have to know this cold. OK, the first one is a `Supplier<String>` because it generates an object and takes zero parameters. The second one is a `Function<String,Boolean>` because it takes one parameter and returns another type. It's a little tricky. You might think it is a `Predicate<String>`. Note that a `Predicate` returns a `boolean` primitive and not a `Boolean` object.

Finally, the third one is either a `BinaryOperator<Integer>` or a `BiFunction<Integer,Integer,Integer>`. Since `BinaryOperator` is a special case of `BiFunction`, either is a correct answer.

`BinaryOperator<Integer>` is the better answer of the two since it is more specific.

Let's try this exercise again but with code. It's harder with code. The first thing you do is look at how many parameters the lambda takes and whether there is a return value. What functional interface would you use to fill in the blanks for these?

```
6:  ___________<List> ex1 = x -> "".equals(x.get(0));
7:  ___________<Long> ex2 = (Long l) -> System.out.println(l);
8:  ___________<String, String> ex3 = (s1, s2) -> false;
```

Again, think about the answers before continuing. Ready? Line 6 passes one `List` parameter to the lambda and returns a `boolean`. This tells us that it is a `Predicate` or `Function`. Since the generic declaration has only one parameter, it is a `Predicate`.

Line 7 passes one `Long` parameter to the lambda and doesn't return anything. This tells us that it is a `Consumer`. Line 8 takes two parameters and returns a `boolean`. When you see a `boolean` returned, think `Predicate` unless the generics specify a `Boolean` return type. In this case, there are two parameters, so it is a `BiPredicate`.

Are you finding these easy? If not, review Table 8.4 again. We aren't kidding. You need to know the table really well. Now that you are fresh from studying the table, we are going to play "identify the error." These are meant to be tricky.

```
6: Function<List<String>> ex1 = x -> x.get(0); // DOES NOT
COMPILE
7: UnaryOperator<Long> ex2 = (Long l) -> 3.14; // DOES NOT
COMPILE
```

Line 6 claims to be a `Function`. A `Function` needs to specify two generic types: the input parameter type and the return value type. The return value type is missing from line 6, causing the code not to compile. Line 7 is a `UnaryOperator`, which returns the same type as it is passed in. The example returns a `double` rather than a `Long`, causing the code not to compile.

Using Convenience Methods on Functional Interfaces

By definition, all functional interfaces have a single abstract method. This doesn't mean they can have only one method, though. Several of the common functional interfaces provide a number of helpful `default` interface methods.

Table 8.5 shows the convenience methods on the built-in functional interfaces that you need to know for the exam. All of these facilitate modifying or combining functional interfaces of the same type. Note that Table 8.5 shows only the main interfaces. The `BiConsumer`, `BiFunction`, and `BiPredicate` interfaces have similar methods available.

TABLE 8.5 Convenience methods

Interface instance	Method return type	Method name	Method parameters
`Consumer`	`Consumer`	`andThen()`	`Consumer`
`Function`	`Function`	`andThen()`	`Function`
`Function`	`Function`	`compose()`	`Function`
`Predicate`	`Predicate`	`and()`	`Predicate`
`Predicate`	`Predicate`	`negate()`	—
`Predicate`	`Predicate`	`or()`	`Predicate`

Let's start with these two `Predicate` variables:

```
Predicate<String> egg = s -> s.contains("egg");
Predicate<String> brown = s -> s.contains("brown");
```

Now we want a `Predicate` for brown eggs and another for all other colors of eggs. We could write this by hand, as shown here:

```
Predicate<String> brownEggs = s -> s.contains("egg") &&
s.contains("brown");
Predicate<String> otherEggs = s -> s.contains("egg") &&
!s.contains("brown");
```

This works, but it's not great. It's a bit long to read, and it contains duplication. What if we decide the letter *e* should be capitalized in *egg*? We'd have to change it in three variables: `egg`, `brownEggs`, and `otherEggs`.

A better way to deal with this situation is to use two of the `default` methods on `Predicate`.

```
Predicate<String> brownEggs = egg.and(brown);
Predicate<String> otherEggs = egg.and(brown.negate());
```

Neat! Now we are reusing the logic in the original `Predicate` variables to build two new ones. It's shorter and clearer what the relationship is between variables. We can also change the spelling of *egg* in one place, and the other two objects will have new logic because they reference it.

Moving on to `Consumer`, let's take a look at the `andThen()` method, which runs two functional interfaces in sequence.

```
Consumer<String> c1 = x -> System.out.print("1: " + x);
Consumer<String> c2 = x -> System.out.print(",2: " + x);

Consumer<String> combined = c1.andThen(c2);
combined.accept("Annie");  // 1: Annie,2: Annie
```

Notice how the same parameter is passed to both `c1` and `c2`. This shows that the `Consumer` instances are run in sequence and are independent of each other. By contrast, the `compose()` method on `Function` chains functional interfaces. However, it passes along the output of one to the input of another.

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3));   // 8
```

This time, the `before` runs first, turning the `3` into `4`. Then the `after` runs, doubling the `4` to `8`. All of the methods in this section are helpful for simplifying your code as you work with functional interfaces.

Learning the Functional Interfaces for Primitives

Remember when we told you to memorize Table 8.4 with the common functional interfaces? Did you? If you didn't, go do it now. We'll wait. We are about to make it more involved. There are also a large number of special functional interfaces for primitives. These are useful in Chapter 10 when we cover streams and optionals.

Most of them are for the `double`, `int`, and `long` types. There is one exception, which is `BooleanSupplier`. We cover that before introducing the functional interfaces for `double`, `int`, and `long`.

Functional Interfaces for *boolean*

`BooleanSupplier` is a separate type. It has one method to implement.

```
@FunctionalInterface
public interface BooleanSupplier {
   boolean getAsBoolean();
}
```

It works just as you've come to expect from functional interfaces. Here's an example:

```
12: BooleanSupplier b1 = () -> true;
13: BooleanSupplier b2 = () -> Math.random()> .5;
14: System.out.println(b1.getAsBoolean());  // true
15: System.out.println(b2.getAsBoolean());  // false
```

Lines 12 and 13 each create a `BooleanSupplier`, which is the only functional interface for `boolean`. Line 14 prints `true`, since it is the result of `b1`. Line 15 prints `true` or `false`, depending on the random value generated.

Functional Interfaces for *double*, *int*, and *long*

Most of the functional interfaces are for `double`, `int`, and `long`. Table 8.6 shows the equivalent of Table 8.4 for these primitives. You probably won't be surprised that you have to memorize it. Luckily, you've memorized Table 8.4 by now and can apply what you've learned to Table 8.6.

TABLE 8.6 Common functional interfaces for primitives

Functional interfaces	Return type	Single abstract method	# of parameters

`DoubleSupplier`
`IntSupplier`
`LongSupplier`
	`double`
	`int`
	`long`
		`getAsDouble`
		`getAsInt`
		`getAsLong`
			0

`DoubleConsumer`
`IntConsumer`
`LongConsumer`
	`void`	`accept`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoublePredicate`
`IntPredicate`
`LongPredicate`
	`boolean`	`test`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleFunction<R>`
`IntFunction<R>`
`LongFunction<R>`
	`R`	`apply`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleUnaryOperator`
`IntUnaryOperator`
`LongUnaryOperator`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleBinaryOperator`
`IntBinaryOperator`
`LongBinaryOperator`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			2 (`double`, `double`)
			2 (`int`, `int`)
			2 (`long`, `long`)

There are a few things to notice that are different between Table 8.4 and Table 8.6.

- Generics are gone from some of the interfaces, and instead the type name tells us what primitive type is involved. In other cases, such as `IntFunction`, only the return type generic is needed because we're converting a primitive `int` into an object.
- The single abstract method is often renamed when a primitive type is returned.

In addition to Table 8.4 equivalents, some interfaces are specific to primitives. Table 8.7 lists these.

TABLE 8.7 Primitive-specific functional interfaces

Functional interfaces	Return type	Single abstract method	# of parameters

`ToDoubleFunction<T>`
`ToIntFunction<T>`
`ToLongFunction<T>`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			1 (`T`)

`ToDoubleBiFunction<T, U>`
`ToIntBiFunction<T, U>`
`ToLongBiFunction<T, U>`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			2 (`T`, `U`)

`DoubleToIntFunction`
`DoubleToLongFunction`
`IntToDoubleFunction`
`IntToLongFunction`
`LongToDoubleFunction`
`LongToIntFunction`
	`int`
	`long`
	`double`
	`long`
	`double`
	`int`
		`applyAsInt`
		`applyAsLong`
		`applyAsDouble`
		`applyAsLong`
		`applyAsDouble`
		`applyAsInt`
			1 (`double`)
			1 (`double`)
			1 (`int`)
			1 (`int`)
			1 (`long`)
			1 (`long`)

`ObjDoubleConsumer<T>`
`ObjIntConsumer<T>`
`ObjLongConsumer<T>`
	`void`	`accept`	2 (`T`, `double`)
			2 (`T`, `int`)
			2 (`T`, `long`)

We've been using functional interfaces for a while now, so you should have a good grasp of how to read the table. Let's do one example just to be sure. Which functional interface would you use to fill in the blank to make the following code compile?

```
var d = 1.0;
_____________ f1 = x -> 1;
f1.applyAsInt(d);
```

When you see a question like this, look for clues. You can see that the functional interface in question takes a `double` parameter and returns an `int`. You can also see that it has a single abstract method named `applyAsInt`. The `DoubleToIntFunction` and `ToIntFunction` functional interfaces meet all three of those criteria.

Working with Variables in Lambdas

Now that we've learned about functional interfaces, we will use them to show different approaches for variables. They 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 explore each one so you'll be alert when tricks show up!

Listing Parameters

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 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.

Since lambda parameters are just like method parameters, you can add modifiers to them. Specifically, you can add the `final` modifier or an annotation, as shown in this example:

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

While this tends to be uncommon in real life, modifiers such as these have been known to appear in passing on the exam.

Parameter List Formats

You have three formats for specifying parameter types within a lambda: without types, with types, and with `var`. The compiler requires all parameters in the lambda to use the same format. Can you see why the following are not valid?

```
5: (var x, y) -> "Hello"                  // DOES NOT
COMPILE
6: (var x, Integer y) -> true             // DOES NOT
COMPILE
7: (String x, var y, Integer z) -> true   // DOES NOT
COMPILE
8: (Integer x, y) -> "goodbye"            // DOES NOT
COMPILE
```

Lines 5 needs to remove `var` from `x` or add it to `y`. Next, lines 6 and 7 need to use the type or `var` consistently. Finally, line 8 needs to remove `Integer` from `x` or add a type to `y`.

Using Local Variables Inside a 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; }
```

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. While this kind of error is less likely to come up in real life, it has been known to appear on the exam!

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!

Real World Scenario

Keep Your Lambdas Short

Having a lambda with multiple lines and a `return` statement is often a clue that you should refactor and put that code in a method. For example, the previous example could be rewritten as follows:

```
Predicate<Integer> p1 = a -> returnSame(a);
```

This simpler form can be further refactored to use a method reference:

```
Predicate<Integer> p1 = this::returnSame;
```

You might be wondering why this is so important. In Chapter 10, lambdas and method references are used in chained method calls. The shorter the lambda, the easier it is to read the code.

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

The only thing lambdas cannot access are variables that are not `final` or effectively final. If you need a refresher on effectively final, see Chapter 5, "Methods."

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 "       // DOES
NOT COMPILE
11:             + volume + " that she is " + color);  // DOES
NOT COMPILE
12:       volume = "softly";
13:    }
14: }
```

In this example, the method parameter `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 non-`final` 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 reassignment! 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 8.8.

TABLE 8.8 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 `final` or effectively final
Method parameter	Allowed if `final` or effectively final
Lambda parameter	Allowed

Summary

We spent a lot of time in this chapter teaching you how to use lambda expressions, and with good reason. The next two chapters depend heavily on your ability to create and use lambda expressions. We recommend you understand this chapter well before moving on.

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, semicolon, and `return` statement can be omitted for a single statement, making the short form as follows:

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

Lambdas can be passed to a method expecting an instance of a functional interface. 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 `final` or effectively final.

A method reference is a compact syntax for writing lambdas that refer to methods. There are four types: `static` methods, instance methods on a

particular object, instance methods on a parameter, and constructor references.

A functional interface has a single abstract method. Any functional interface can be implemented with a lambda expression. You must know the built-in functional interfaces.

You should review the tables in the chapter. While there are many tables, some share common patterns, making it easier to remember them. You absolutely must memorize Table 8.4, which lists the common functional interfaces.

Exam Essentials

Write simple lambda expressions. Look for the presence or absence of optional elements in lambda code. Parameter types are optional. Braces, a semicolon, 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.

Determine whether a variable can be used in a lambda body. Local variables and method parameters must be `final` or 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.

Translate method references to the "long-form" lambda. Be able to convert method references into regular lambda expressions, and vice versa. For example, `System.out::print` and `x -> System.out.print(x)` are equivalent. Remember that the order of method parameters is inferred for method references.

Determine whether an interface is a functional interface. Use the single abstract method (SAM) rule to determine whether an interface is a functional interface. Other interface method types (`default`, `private`, `static`, and `private static`) do not count toward the single abstract method count, nor do any `public` methods with signatures found in `Object`.

Identify the correct functional interface given the number of parameters, return type, and method name—and vice versa. The most common functional interfaces are `Supplier`, `Consumer`, `Function`, and

`Predicate`. There are also binary versions and primitive versions of many of these methods. You can use the number of parameters and return type to tell them apart.

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: } }
```

   A. `match`

   B. `not match`

   C. Compiler error on line 8

   D. Compiler error on lines 10 and 11

   E. Compiler error on lines 12 and 13

   F. A runtime exception

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) {
```