Chapter 12
Java Fundamentals

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Java Fundamentals**
  - Create and use final classes
  - Create and use inner, nested and anonymous classes
  - Create and use enumerations
- **Java Interfaces**
  - Create and use interfaces with default methods
  - Create and use interfaces with private methods
- **Functional Interface and Lambda Expressions**
  - Define and write functional interfaces
  - Create and use lambda expressions including statement lambdas, local-variable for lambda parameters

---

Welcome to the first chapter on your road to taking the 1Z0-816 Programmer II exam! If you've recently taken the 1Z0-815 Programmer I exam, then you should be well versed in class structure, inheritance, scope, abstract types, etc. If not, you might want to review earlier chapters. The exam expects you to have a solid foundation on these topics.

In this chapter, we are going to expand your understanding of Java fundamentals including enums and nested classes, various interface members, functional interfaces, and lambda expressions. Pay attention in this chapter, as many of these topics will be used throughout the rest of this

book. Even if you use them all the time, there are subtle rules you might not be aware of.

Finally, we want to wish you a hearty congratulations on beginning your journey to prepare for the 1Z0-816 Programmer II exam!

Applying the *final* Modifier

From previous chapters, you should remember the `final` modifier can be applied to variables, methods, and classes. Marking a variable `final` means the value cannot be changed after it is assigned. Marking a method or class `final` means it cannot be overridden or extended, respectively. In this section, we will review the rules for using the `final` modifier.

If you studied `final` classes for the 1Z0-815 exam recently, then you can probably skip this section and go straight to enums.

Declaring final Local Variables

Let's start by taking a look at some local variables marked with the `final` modifier:

```
private void printZooInfo(boolean isWeekend) {
   final int giraffe = 5;
   final long lemur;
   if(isWeekend) lemur = 5;
   else lemur = 10;
   System.out.println(giraffe+" "+lemur);
}
```

As shown with the `lemur` variable, we don't need to assign a value when a `final` variable is declared. The rule is only that it must be assigned a value before it can be used. Contrast this with the following example:

```
private void printZooInfo(boolean isWeekend) {
   final int giraffe = 5;
   final long lemur;
   if(isWeekend) lemur = 5;
   giraffe = 3; // DOES NOT COMPILE
   System.out.println(giraffe+" "+lemur); // DOES NOT COMPILE
}
```

This snippet contains two compilation errors. The `giraffe` variable is already assigned a value, so attempting to assign it a new value is not permitted. The second compilation error is from attempting to use the `lemur` variable, which would not be assigned a value if `isWeekend` is `false`. The compiler does not allow the use of local variables that may not have been assigned a value, whether they are marked `final` or not.

Just because a variable reference is marked `final` does not mean the object associated with it cannot be modified. Consider the following code snippet:

```
final StringBuilder cobra = new StringBuilder();
cobra.append("Hssssss");
cobra.append("Hssssss!!!");
```

In the `cobra` example, the object reference is constant, but that doesn't mean the data in the class is constant.

Throughout this book, you'll see a lot of examples involving zoos and animals. We chose this topic because it is a fruitful topic for data modeling. There are a wide variety of species, with very interesting attributes, relationships, and hierarchies. In addition, there are a lot of complex tasks involved in managing a zoo. Ideally, you enjoy the topic and learn about animals along the way!

Adding *final* to Instance and *static* Variables

Instance and `static` class variables can also be marked `final`. If an instance variable is marked `final`, then it must be assigned a value when it is declared or when the object is instantiated. Like a local `final` variable, it cannot be assigned a value more than once, though. The following `PolarBear` class demonstrates these properties:

```
public class PolarBear {
   final int age = 10;
   final int fishEaten;
   final String name;

   { fishEaten = 10; }

   public PolarBear() {
      name = "Robert";
   }
   public PolarBear(int height) {
      this();
   }
}
```

The `age` variable is given a value when it is declared, while the `fishEaten` variable is assigned a value in an instance initializer. The

`name` variable is given a value in the no-argument constructor. Notice that the second constructor does not assign a value to `name` , but since it calls the no-argument constructor first, `name` is guaranteed to be assigned a value in the first line of this constructor.

The rules for `static final` variables are similar to instance `final` variables, except they do not use `static` constructors (there is no such thing!) and use `static` initializers instead of instance initializers.

```
public class Panda {
   final static String name = "Ronda";
   static final int bamboo;
   static final double height; // DOES NOT COMPILE
   static { bamboo = 5;}
}
```

The `name` variable is assigned a value when it is declared, while the `bamboo` variable is assigned a value in a `static` initializer. The `height` variable is not assigned a value anywhere in the class definition, so that line does not compile.

Writing *final* Methods

Methods marked `final` cannot be overridden by a subclass. This essentially prevents any polymorphic behavior on the method call and ensures that a specific version of the method is always called.

Remember that methods can be assigned an `abstract` or `final` modifier. An `abstract` method is one that does not define a method body and can appear only inside an abstract class or interface. A `final` method is one that cannot be overridden by a subclass. Let's take a look at an example:

```
public abstract class Animal {
   abstract void chew();
```

```
}

public class Hippo extends Animal {
   final void chew() {}
}

public class PygmyHippo extends Hippo {
   void chew() {} // DOES NOT COMPILE
}
```

The `chew()` method is declared `abstract` in the `Animal` class. It is then implemented in the concrete `Hippo` class, where it is marked `final`, thereby preventing a subclass of `Hippo` from overriding it. For this reason, the `chew()` method in `PygmyHippo` does not compile since it is inherited as `final`.

What happens if a method is marked both `abstract` and `final`? Well, that's like saying, "I want to declare a method that someone else will provide an implementation for, while also telling that person that they are not allowed to provide an implementation." For this reason, the compiler does not allow it.

```
abstract class ZooKeeper {
   public abstract final void openZoo(); // DOES NOT COMPILE
}
```

Marking Classes *final*

Lastly, the `final` modifier can be applied to class declarations as well. A `final` class is one that cannot be extended. For example, the following does not compile:

```
public final class Reptile {}

public class Snake extends Reptile {} // DOES NOT COMPILE
```

Like we saw with `final` methods, classes cannot be marked both `abstract` and `final`. For example, the following two declarations do not compile:

```
public abstract final class Eagle {} // DOES NOT COMPILE

public final interface Hawk {} // DOES NOT COMPILE
```

It is not possible to write a class that provides a concrete implementation of the abstract `Eagle` class, as it is marked `final` and cannot be extended. The `Hawk` interface also does not compile, although the reason is subtler. The compiler automatically applies the implicit `abstract` modifier to each interface declaration. Just like abstract classes, interfaces cannot be marked `final`.

As you may remember from Chapter 9, "Advanced Class Design," a modifier that is inserted automatically by the compiler is referred to as an *implicit modifier*. Think of an implicit modifier as being present, even if it is not written out. For example, a method that is implicitly `public` cannot be marked `private`. Implicit modifiers are common in interface members, as we will see later in this chapter.

Working with Enums

In programming, it is common to have a type that can only have a finite set of values, such as days of the week, seasons of the year, primary colors, etc. An *enumeration* is like a fixed set of constants. In Java, an *enum*, short for "enumerated type," can be a top-level type like a class or interface, as well as a nested type like an inner class.

Using an enum is much better than using a bunch of constants because it provides type-safe checking. With numeric or `String` constants, you can pass an invalid value and not find out until runtime. With enums, it is impossible to create an invalid enum value without introducing a compiler error.

Enumerations show up whenever you have a set of items whose types are known at compile time. Common examples include the compass directions, the months of the year, the planets in the solar system, or the cards in a deck (well, maybe not the planets in a solar system, given that Pluto had its planetary status revoked).

Creating Simple Enums

To create an enum, use the `enum` keyword instead of the `class` or `interface` keyword. Then list all of the valid types for that enum.

```
public enum Season {
   WINTER, SPRING, SUMMER, FALL
}
```

Keep the `Season` enum handy, as we will be using it throughout this section.

Enum values are considered constants and are commonly written using snake case, often stylized as snake_case. This style uses an underscore (`_`) to separate words with constant values commonly written in all uppercase. For example, an enum declaring a list of ice cream flavors might include values like `VANILLA`, `ROCKY_ROAD`, `MINT_CHOCOLATE_CHIP`, and so on.

Behind the scenes, an enum is a type of class that mainly contains `static` members. It also includes some helper methods like `name()`. Using an enum is easy.

```
Season s = Season.SUMMER;
System.out.println(Season.SUMMER); // SUMMER
System.out.println(s == Season.SUMMER); // true
```

As you can see, enums print the name of the enum when `toString()` is called. They can be compared using `==` because they are like `static final` constants. In other words, you can use `equals()` or `==` to compare enums, since each enum value is initialized only once in the Java Virtual Machine (JVM).

An enum provides a `values()` method to get an array of all of the values. You can use this like any normal array, including in an enhanced `for` loop, often called a *for-each loop*.

```
for(Season season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

The output shows that each enum value has a corresponding `int` value, and the values are listed in the order in which they are declared.

```
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

The `int` value will remain the same during your program, but the program is easier to read if you stick to the human-readable enum value.

You can't compare an `int` and enum value directly anyway since an enum is a type, like a Java class, and *not* a primitive `int`.

```
if ( Season.SUMMER == 2) {} // DOES NOT COMPILE
```

Another useful feature is retrieving an enum value from a `String` using the `valueOf()` method. This is helpful when working with older code. The `String` passed in must match the enum value exactly, though.

```
Season s = Season.valueOf("SUMMER"); // SUMMER

Season t = Season.valueOf("summer"); // Throws an exception at runtime
```

The first statement works and assigns the proper enum value to `s`. Note that this line is not creating an enum value, at least not directly. Each enum value is created once when the enum is first loaded. Once the enum has been loaded, it retrieves the single enum value with the matching name.

The second statement encounters a problem. There is no enum value with the lowercase name `summer`. Java throws up its hands in defeat and throws an `IllegalArgumentException`.

```
Exception in thread "main" java.lang.IllegalArgumentException:
   No enum constant enums.Season.summer
```

One thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season { } // DOES NOT COMPILE
```

The values in an enum are all that are allowed. You cannot add more by extending the enum.

Using Enums in Switch Statements

Enums can be used in `switch` statements. Pay attention to the `case` values in this code:

```
Season summer = Season.SUMMER;
switch (summer) {
   case WINTER:
      System.out.println("Get out the sled!");
      break;
   case SUMMER:
      System.out.println("Time for the pool!");
      break;
   default:
      System.out.println("Is it summer yet?");
}
```

The code prints `"Time for the pool!"` since it matches `SUMMER`. In each `case` statement, we just typed the value of the enum rather than writing `Season.WINTER`. After all, the compiler already knows that the only possible matches can be enum values. Java treats the enum type as implicit. In fact, if you were to type `case Season.WINTER`, it would not compile. Don't believe us? Take a look at the following example:

```
switch (summer) {
   case Season.FALL:  // DOES NOT COMPILE
      System.out.println("Rake some leaves!");
      break;
   case 0:            // DOES NOT COMPILE
      System.out.println("Get out the sled!");
      break;
}
```

The first `case` statement does not compile because `Season` is used in the `case` value. If we changed `Season.FALL` to just `FALL`, then the line would compile. What about the second `case` statement? Just as earlier we said that you can't compare enums with `int` values, you cannot use them in a `switch` statement with `int` values either. On the exam, pay

special attention when working with enums that they are used only as enums.

Adding Constructors, Fields, and Methods

Enums can have more in them than just a list of values. Let's say our zoo wants to keep track of traffic patterns for which seasons get the most visitors.

```
1: public enum Season {
2:    WINTER("Low"), SPRING("Medium"), SUMMER("High"), FALL("Medium");
3:    private final String expectedVisitors;
4:    private Season(String expectedVisitors) {
5:       this.expectedVisitors = expectedVisitors;
6:    }
7:    public void printExpectedVisitors() {
8:       System.out.println(expectedVisitors);
9:    } }
```

There are a few things to notice here. On line 2, the list of enum values ends with a semicolon (`;`). While this is optional when our enum is composed solely of a list of values, it is required if there is anything in the enum besides the values.

Lines 3–9 are regular Java code. We have an instance variable, a constructor, and a method. We mark the instance variable `final` on line 3 so that our enum values are considered immutable. Although this is certainly not required, it is considered a good coding practice to do so. Since enum values are shared by all processes in the JVM, it would be problematic if one of them could change the value inside an enum.

Real World Scenario

CREATING IMMUTABLE OBJECTS

The *immutable objects pattern* is an object-oriented design pattern in which an object cannot be modified after it is created. Instead of modifying an immutable object, you create a new object that contains any properties from the original object you want copied over.

Many Java libraries contain immutable objects, including `String` and classes in the `java.time` package, just to name a few. Immutable objects are invaluable in concurrent applications since the state of the object cannot change or be corrupted by a rogue thread. You'll learn more about threads in Chapter 18, “Concurrency.”

All enum constructors are implicitly `private` , with the modifier being optional. This is reasonable since you can't extend an enum and the constructors can be called only within the enum itself. An enum constructor will not compile if it contains a `public` or `protected` modifier.

How do we call an enum method? It's easy.

```
Season.SUMMER.printExpectedVisitors();
```

Notice how we don't appear to call the constructor. We just say that we want the enum value. The first time that we ask for any of the enum values, Java constructs all of the enum values. After that, Java just returns the already constructed enum values. Given that explanation, you can see why this calls the constructor only once:

```
public enum OnlyOne {
   ONCE(true);
   private OnlyOne(boolean b) {
      System.out.print("constructing,");
   }
```

```
}

public class PrintTheOne {
   public static void main(String[] args) {
      System.out.print("begin,");
      OnlyOne firstCall = OnlyOne.ONCE;  // prints constructing,
      OnlyOne secondCall = OnlyOne.ONCE; // doesn't print anything
      System.out.print("end");
   }
}
```

This class prints the following:

```
begin,constructing,end
```

If the `OnlyOne` enum was used earlier, and therefore initialized sooner, then the line that declares the `firstCall` variable would not print anything.

This technique of a constructor and state allows you to combine logic with the benefit of a list of values. Sometimes, you want to do more. For example, our zoo has different seasonal hours. It is cold and gets dark early in the winter. We could keep track of the hours through instance variables, or we can let each enum value manage hours itself.

```
public enum Season {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   FALL {
      public String getHours() { return "9am-5pm"; }
```

```
    };
    public abstract String getHours();
}
```

What's going on here? It looks like we created an `abstract` class and a bunch of tiny subclasses. In a way we did. The enum itself has an `abstract` method. This means that each and every enum value is required to implement this method. If we forget to implement the method for one of the values, then we get a compiler error.

```
The enum constant WINTER must implement the abstract method getHours()
```

If we don't want each and every enum value to have a method, we can create a default implementation and override it only for the special cases.

```
public enum Season {
    WINTER {
        public String getHours() { return "10am-3pm"; }
    },
    SUMMER {
        public String getHours() { return "9am-7pm"; }
    },
    SPRING, FALL;
    public String getHours() { return "9am-5pm"; }
}
```

This one looks better. We only code the special cases and let the others use the enum-provided implementation. Of course, overriding `getHours()` is possible only if it is not marked `final`.

Just because an enum can have lots of methods doesn't mean that it should. Try to keep your enums simple. If your enum is more than a page or two, it is way too long. Most enums are just a handful of lines. The main reason they get long is that when you start with a one- or two-line method and then declare it for each of your dozen enum types, it grows

long. When they get too long or too complex, it makes the enum hard to read.

You might have noticed that in each of these enum examples, the list of values came first. This was not an accident. Whether the enum is simple or contains a ton of methods, constructors, and variables, the compiler requires that the list of values always be declared first.

Creating Nested Classes

A *nested class* is a class that is defined within another class. A nested class can come in one of four flavors.

- *Inner class*: A non-`static` type defined at the member level of a class
- *Static nested class:* A `static` type defined at the member level of a class
- *Local class*: A class defined within a method body
- *Anonymous class*: A special case of a local class that does not have a name

There are many benefits of using nested classes. They can encapsulate helper classes by restricting them to the containing class. They can make it easy to create a class that will be used in only one place. They can make the code cleaner and easier to read. This section covers all four types of nested classes.

By convention and throughout this chapter, we often use the term *inner* or *nested class* to apply to other Java types, including interfaces and enums. Although you are unlikely to encounter this on the exam, interfaces and enums can be declared as both inner classes and `static` nested classes, but not as local or anonymous classes.

When used improperly, though, nested classes can sometimes make the code harder to read. They also tend to tightly couple the enclosing and inner class, whereas there may be cases where you want to use the inner class by itself. In this case, you should move the inner class out into a separate top-level class.

Unfortunately, the exam tests these edge cases where programmers wouldn't typically use a nested class. For example, the exam might have an inner class within another inner class. This tends to create code that is difficult to read, so please never do this in practice!

Declaring an Inner Class

An *inner class*, also called a *member inner class*, is a non-`static` type defined at the member level of a class (the same level as the methods, instance variables, and constructors). Inner classes have the following properties:

- Can be declared `public`, `protected`, package-private (default), or `private`
- Can extend any class and implement interfaces
- Can be marked `abstract` or `final`
- Cannot declare `static` fields or methods, except for `static final` fields
- Can access members of the outer class including `private` members

The last property is actually pretty cool. It means that the inner class can access variables in the outer class without doing anything special. Ready for a complicated way to print `Hi` three times?

```
1:  public class Outer {
2:     private String greeting = "Hi";
3:
4:     protected class Inner {
5:        public int repeat = 3;
6:        public void go() {
7:           for (int i = 0; i < repeat; i++)
8:              System.out.println(greeting);
9:        }
10:    }
11:
12:    public void callInner() {
13:       Inner inner = new Inner();
14:       inner.go();
15:    }
16:    public static void main(String[] args) {
17:       Outer outer = new Outer();
18:       outer.callInner();
19: } }
```

An inner class declaration looks just like a stand-alone class declaration except that it happens to be located inside another class. Line 8 shows that the inner class just refers to `greeting` as if it were available. This works because it is in fact available. Even though the variable is `private`, it is within that same class.

Since an inner class is not `static`, it has to be used with an instance of the outer class. Line 13 shows that an instance of the outer class can instantiate `Inner` normally. This works because `callInner()` is an instance method on `Outer`. Both `Inner` and `callInner()` are members of `Outer`. Since they are peers, they just write the name.

There is another way to instantiate `Inner` that looks odd at first. OK, well maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:    public static void main(String[] args) {
21:       Outer outer = new Outer();
22:       Inner inner = outer.new Inner(); // create the inner class
23:       inner.go();
24:    }
```

Let's take a closer look at line 22. We need an instance of `Outer` to create `Inner`. We can't just call `new Inner()` because Java won't know with which instance of `Outer` it is associated. Java solves this by calling `new` as if it were a method on the `outer` variable.

.*CLASS* FILES FOR INNER CLASSES

Compiling the `Outer.java` class with which we have been working creates two class files. `Outer.class` you should be expecting. For the inner class, the compiler creates `Outer$Inner.class`. You don't need to know this syntax for the exam. We mention it so that you aren't surprised to see files with `$` appearing in your directories. You do need to understand that multiple class files are created.

Inner classes can have the same variable names as outer classes, making scope a little tricky. There is a special way of calling `this` to say which variable you want to access. This is something you might see on the exam but ideally not in the real world.

In fact, you aren't limited to just one inner class. Please never do this in code you write. Here is how to nest multiple classes and access a variable with the same name in each:

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);        // 30
9:              System.out.println(this.x);   // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

Yes, this code makes us cringe too. It has two nested classes. Line 14 instantiates the outermost one. Line 15 uses the awkward syntax to instantiate a `B`. Notice the type is `A.B`. We could have written `B` as the type because that is available at the member level of `B`. Java knows where to look for it. On line 16, we instantiate a `C`. This time, the `A.B.C` type is necessary to specify. `C` is too deep for Java to know where to look. Then line 17 calls a method on `c`.

Lines 8 and 9 are the type of code that we are used to seeing. They refer to the instance variable on the current class—the one declared on line 6 to be precise. Line 10 uses `this` in a special way. We still want an instance variable. But this time we want the one on the `B` class, which is the variable on line 4. Line 11 does the same thing for class `A`, getting the variable from line 2.

INNER CLASSES REQUIRE AN INSTANCE

Take a look at the following and see whether you can figure out why two of the three constructor calls do not compile:

```
public class Fox {
   private class Den {}
   public void goHome() {
      new Den();
   }
   public static void visitFriend() {
      new Den();  // DOES NOT COMPILE
   }
}

public class Squirrel {
   public void visitFox() {
      new Den();  // DOES NOT COMPILE
   }
}
```

The first constructor call compiles because `goHome()` is an instance method, and therefore the call is associated with the `this` instance. The second call does not compile because it is called inside a `static` method. You can still call the constructor, but you have to explicitly give it a reference to a `Fox` instance.

The last constructor call does not compile for two reasons. Even though it is an instance method, it is not an instance method inside the `Fox` class. Adding a `Fox` reference would not fix the problem entirely, though. `Den` is `private` and not accessible in the `Squirrel` class.

Creating a *static* Nested Class

A *static nested class* is a `static` type defined at the member level. Unlike an inner class, a `static` nested class can be instantiated without an instance of the enclosing class. The trade-off, though, is it can't access instance variables or methods in the outer class directly. It can be done but requires an explicit reference to an outer class variable.

In other words, it is like a top-level class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it.
- It can be made `private` or use one of the other access modifiers to encapsulate it.
- The enclosing class can refer to the fields and methods of the `static` nested class.

Let's take a look at an example:

```
1: public class Enclosing {
2:    static class Nested {
3:       private int price = 6;
4:    }
5:    public static void main(String[] args) {
6:       Nested nested = new Nested();
7:       System.out.println(nested.price);
8: } }
```

Line 6 instantiates the nested class. Since the class is `static` , you do not need an instance of `Enclosing` to use it. You are allowed to access `private` instance variables, which is shown on line 7.

IMPORTING A *STATIC* NESTED CLASS

Importing a `static` nested class is interesting. You can import it using a regular import.

```
// Toucan.java
package bird;
public class Toucan {
   public static class Beak {}
}

// BirdWatcher.java
package watcher;
import bird.Toucan.Beak; // regular import ok
public class BirdWatcher {
   Beak beak;
}
```

Since it is `static`, you can also use a `static` import.

```
import static bird.Toucan.Beak;
```

Either one will compile. Surprising, isn't it? Remember, Java treats the enclosing class as if it were a namespace.

Writing a Local Class

A *local class* is a nested class defined within a method. Like local variables, a local class declaration does not exist until the method is invoked, and it goes out of scope when the method returns. This means you can create instances only from within the method. Those instances can still be returned from the method. This is just how local variables work.

Local classes are not limited to being declared only inside methods. They can be declared inside constructors and initializers too. For simplicity, we limit our discussion to methods in this chapter.

Local classes have the following properties:

- They do not have an access modifier.
- They cannot be declared `static` and cannot declare `static` fields or methods, except for `static final` fields.
- They have access to all fields and methods of the enclosing class (when defined in an instance method).
- They can access local variables if the variables are `final` or effectively final.

As you saw in Chapter 6, "Lambdas and Functional Interfaces," *effectively final* refers to a local variable whose value does not change after it is set. A simple test for effectively final is to add the `final` modifier to the local variable declaration. If it still compiles, then the local variable is effectively final.

Ready for an example? Here's a complicated way to multiply two numbers:

```
1:  public class PrintNumbers {
2:     private int length = 5;
3:     public void calculate() {
4:        final int width = 20;
5:        class MyLocalClass {
```

```
6:           public void multiply() {
7:              System.out.print(length * width);
8:           }
9:        }
10:       MyLocalClass local = new MyLocalClass();
11:       local.multiply();
12:    }
13:    public static void main(String[] args) {
14:       PrintNumbers outer = new PrintNumbers();
15:       outer.calculate();
16:    }
17: }
```

Lines 5 through 9 are the local class. That class's scope ends on line 12 where the method ends. Line 7 refers to an instance variable and a final local variable, so both variable references are allowed from within the local class.

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. Let's talk about that now. The compiler is generating a `.class` file from your local class. A separate class has no way to refer to local variables. If the local variable is `final`, Java can handle it by passing it to the constructor of the local class or by storing it in the `.class` file. If it weren't effectively final, these tricks wouldn't work because the value could change after the copy was made.

As an illustrative example, consider the following:

```
public void processData() {
   final int length = 5;
   int width = 10;
   int height = 2;
   class VolumeCalculator {
      public int multiply() {
         return length * width * height; // DOES NOT COMPILE
      }
   }
```

```
      width = 2;
   }
```

The length and height variables are `final` and effectively final, respectively, so neither causes a compilation issue. On the other hand, the `width` variable is reassigned during the method so it cannot be effectively final. For this reason, the local class declaration does not compile.

Defining an Anonymous Class

An *anonymous class* is a specialized form of a local class that does not have a name. It is declared and instantiated all in one statement using the `new` keyword, a type name with parentheses, and a set of braces `{}`. Anonymous classes are required to extend an existing class or implement an existing interface. They are useful when you have a short implementation that will not be used anywhere else. Here's an example:

```
1:  public class ZooGiftShop {
2:     abstract class SaleTodayOnly {
3:        abstract int dollarsOff();
4:     }
5:     public int admission(int basePrice) {
6:        SaleTodayOnly sale = new SaleTodayOnly() {
7:           int dollarsOff() { return 3; }
8:        };  // Don't forget the semicolon!
9:        return basePrice - sale.dollarsOff();
10: } }
```

Lines 2 through 4 define an `abstract` class. Lines 6 through 8 define the anonymous class. Notice how this anonymous class does not have a name. The code says to instantiate a new `SaleTodayOnly` object. But wait, `SaleTodayOnly` is `abstract`. This is OK because we provide the class body right there—anonymously. In this example, writing an anonymous class is equivalent to writing a local class with an unspecified name that extends `SaleTodayOnly` and then immediately using it.

Pay special attention to the semicolon on line 8. We are declaring a local variable on these lines. Local variable declarations are required to end with semicolons, just like other Java statements—even if they are long and happen to contain an anonymous class.

Now we convert this same example to implement an `interface` instead of extending an `abstract` class.

```
1:  public class ZooGiftShop {
2:     interface SaleTodayOnly {
3:        int dollarsOff();
4:     }
5:     public int admission(int basePrice) {
6:        SaleTodayOnly sale = new SaleTodayOnly() {
7:           public int dollarsOff() { return 3; }
8:        };
9:        return basePrice - sale.dollarsOff();
10: } }
```

The most interesting thing here is how little has changed. Lines 2 through 4 declare an `interface` instead of an `abstract` class. Line 7 is `public` instead of using default access since interfaces require `public` methods. And that is it. The anonymous class is the same whether you implement an interface or extend a class! Java figures out which one you want automatically. Just remember that in this second example, an instance of a class is created on line 6, not an interface.

But what if we want to implement both an `interface` and extend a class? You can't with an anonymous class, unless the class to extend is `java.lang.Object`. The `Object` class doesn't count in the rule. Remember that an anonymous class is just an unnamed local class. You can write a local class and give it a name if you have this problem. Then you can extend a class and implement as many `interface`s as you like. If your code is this complex, a local class probably isn't the most readable option anyway.

There is one more thing that you can do with anonymous classes. You can define them right where they are needed, even if that is an argument to another method.

```
1:  public class ZooGiftShop {
2:     interface SaleTodayOnly {
3:        int dollarsOff();
4:     }
5:     public int pay() {
6:        return admission(5, new SaleTodayOnly() {
7:           public int dollarsOff() { return 3; }
8:        });
9:     }
10:    public int admission(int basePrice, SaleTodayOnly sale) {
11:       return basePrice - sale.dollarsOff();
12: }}
```

Lines 6 through 8 are the anonymous class. We don't even store it in a local variable. Instead, we pass it directly to the method that needs it. Reading this style of code does take some getting used to. But it is a concise way to create a class that you will use only once.

You can even define anonymous classes outside a method body. The following may look like we are instantiating an interface as an instance variable, but the `{}` after the interface name indicates that this is an anonymous inner class implementing the interface.

```
public class Gorilla {
   interface Climb {}
   Climb climbing = new Climb() {};
}
```

Real World Scenario

ANONYMOUS CLASSES AND LAMBDA EXPRESSIONS

Prior to Java 8, anonymous classes were frequently used for asynchronous tasks and event handlers. For example, the following shows an anonymous class used as an event handler in a JavaFX application:

```
Button redButton = new Button();
redButton.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent e) {
      System.out.println("Red button pressed!");
   }
});
```

Since Java 8, though, lambda expressions are a much more concise way of expressing the same thing.

```
Button redButton = new Button();
redButton.setOnAction(e -> System.out.println("Red button pressed!"));
```

The only restriction is that the variable type must be a functional interface. If you haven't worked with functional interfaces and lambda expressions before, don't worry. We'll be reviewing them in this chapter.

Reviewing Nested Classes

For the exam, make sure that you know the information in Table 12.1 and Table 12.2 about which syntax rules are permitted in Java.

TABLE 12.1 Modifiers in nested classes

Permitted Modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`Final`	Yes	Yes	Yes	No

TABLE 12.2 Members in nested classes

Permitted Members	Inner class	`static` nested class	Local class	Anonymous class
Instance methods	Yes	Yes	Yes	Yes
Instance variables	Yes	Yes	Yes	Yes
`static` methods	No	Yes	No	No
`static` variables	Yes (if `final`)	Yes	Yes (if `final`)	Yes (if `final`)

You should also know the information in Table 12.3 about types of access. For example, the exam might try to trick you by having a `static` class

access an outer class instance variable without a reference to the outer class.

TABLE 12.3 Nested class access rules

	Inner class	`static` nested class	Local class	Anonymous class
Can extend any class or implement any number of interfaces	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class without a reference	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Understanding Interface Members

When Java was first released, there were only two types of members an interface declaration could include: abstract methods and constant (`static final`) variables. Since Java 8 and 9 were released, four new

method types have been added that we will cover in this section. Keep Table 12.4 handy as we discuss the various interface types in this section.

TABLE 12.4 Interface member types

	Since Java version	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	1.0	Class	—	`public static final`	Yes
Abstract method	1.0	Instance	—	`public abstract`	No
Default method	`8`	Instance	`default`	`public`	Yes
Static method	`8`	Class	`static`	`public`	Yes
Private method	`9`	Instance	`private`	—	Yes
Private static method	`9`	Class	`private static`	—	Yes

We assume from your previous studies that you know how to define a constant variable and abstract method, so we'll move on to the newer interface member types.

Relying on a *default* Interface Method

A *default method* is a method defined in an interface with the `default` keyword and includes a method body. Contrast `default` methods with abstract methods in an interface, which do not define a method body.

A `default` method may be overridden by a class implementing the interface. The name *default* comes from the concept that it is viewed as an abstract interface method with a default implementation. The class has the option of overriding the `default` method, but if it does not, then the default implementation will be used.

PURPOSE OF *DEFAULT* METHODS

One motivation for adding `default` methods to the Java language was for backward compatibility. A `default` method allows you to add a new method to an existing interface, without the need to modify older code that implements the interface.

Another motivation for adding `default` methods to Java is for convenience. For instance, the `Comparator` interface includes a `default reversed()` method that returns a new `Comparator` in the order reversed. While these can be written in every class implementing the interface, having it defined in the interface as a `default` method is quite useful.

The following is an example of a `default` method defined in an interface:

```
public interface IsWarmBlooded {
   boolean hasScales();
   default double getTemperature() {
      return 10.0;
   }
}
```

This example defines two interface methods: one is the abstract `hasScales()` method, and the other is the `default getTemperature()` method. Any class that implements `IsWarmBlooded` may rely on the default implementation of `getTemperature()` or override the method with its own version. Both of these methods include the implicit `public` modifier, so overriding them with a different access modifier is not allowed.

Note that the `default` interface method modifier is not the same as the `default` label used in `switch` statements. Likewise, although package-private access is commonly referred to as default access, that feature is implemented by omitting an access modifier. Sorry if this is confusing! We agree Java has overused the word *default* over the years.

For the exam, you should be familiar with various rules for declaring `default` methods.

Default Interface Method Definition Rules

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is assumed to be `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

The first rule should give you some comfort in that you'll only see `default` methods in interfaces. If you see them in a class or enum on the exam, something is wrong. The second rule just denotes syntax, as `default` methods must use the `default` keyword. For example, the following code snippets will not compile:

```
public interface Carnivore {
   public default void eatMeat();        // DOES NOT COMPILE
   public int getRequiredFoodAmount() {  // DOES NOT COMPILE
      return 13;
   }
}
```

The first method, `eatMeat()`, doesn't compile because it is marked as `default` but doesn't provide a method body. The second method, `getRequiredFoodAmount()`, also doesn't compile because it provides a method body but is not marked with the `default` keyword.

What about our third, fourth, and fifth rules? Like abstract interface methods, `default` methods are implicitly `public`. Unlike abstract methods, though, `default` interface methods cannot be marked `abstract` and must provide a body. They also cannot be marked as `final`, because they can always be overridden in classes implementing the interface. Finally, they cannot be marked `static` since they are associated with the instance of the class implementing the interface.

Inheriting Duplicate default Methods

We have one last rule for `default` methods that warrants some discussion. You may have realized that by allowing `default` methods in interfaces, coupled with the fact that a class may implement multiple interfaces, Java has essentially opened the door to multiple inheritance problems. For example, what value would the following code output?

```
public interface Walk {
   public default int getSpeed() { return 5; }
```

```
}

public interface Run {
   public default int getSpeed() { return 10; }
}

public class Cat implements Walk, Run {  // DOES NOT COMPILE
   public static void main(String[] args) {
      System.out.println(new Cat().getSpeed());
   }
}
```

In this example, `Cat` inherits the two `default` methods for `getSpeed()`, so which does it use? Since `Walk` and `Run` are considered siblings in terms of how they are used in the `Cat` class, it is not clear whether the code should output `5` or `10`. In this case, Java throws up its hands and says "Too hard, I give up!" and fails to compile.

If a class implements two interfaces that have `default` methods with the same method signature, the compiler will report an error. This rule holds true even for abstract classes because the duplicate method could be called within a concrete method within the abstract class. All is not lost, though. If the class implementing the interfaces *overrides* the duplicate `default` method, then the code will compile without issue.

By overriding the conflicting method, the ambiguity about which version of the method to call has been removed. For example, the following modified implementation of `Cat` will compile and output `1`:

```
public class Cat implements Walk, Run {
   public int getSpeed() { return 1; }

   public static void main(String[] args) {
      System.out.println(new Cat().getSpeed());
   }
}
```

In this section, all of our conflicting methods had identical declarations. These rules also apply to methods with the same signature but different return types or declared exceptions. If a `default` method is overridden in the concrete class, then it must use a declaration that is compatible, following the rules for overriding methods introduced in Chapter 8, "Class Design".

Calling a Hidden *default* Method

Let's conclude our discussion of `default` methods by revisiting the `Cat` example, with two inherited `default getSpeed()` methods. Given our corrected implementation of `Cat` that overrides the `getSpeed()` method and returns `1`, how would you call the version of the `default` method in the `Walk` interface? Take a few minutes to think about the following incomplete code:

```
public class Cat implements Walk, Run {
   public int getSpeed() { return 1; }

   public int getWalkSpeed() {
      return ___________; // TODO: Call Walk's version of getSpeed()
   }

   public static void main(String[] args) {
      System.out.println(new Cat().getWalkSpeed());
   }
}
```

This is an area where a `default` method exhibits properties of both a `static` and instance method. Ready for the answer? Well, first off, you definitely can't call `Walk.getSpeed()`. A `default` method is treated as

part of the instance since they can be overridden, so they cannot be called like a `static` method.

What about calling `super.getSpeed()`? That gets us a little closer, but which of the two inherited `default` methods is called? It's ambiguous and therefore not allowed. In fact, the compiler won't allow this even if there is only one inherited `default` method, as an interface is not part of the class hierarchy.

The solution is a combination of both of these answers. Take a look at the `getWalkSpeed()` method in this implementation of the `Cat` class:

```
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   }

   public static void main(String[] args) {
      System.out.println(new Cat().getWalkSpeed());
   }
}
```

In this example, we first use the interface name, followed by the `super` keyword, followed by the `default` method we want to call. We also put the call to the inherited `default` method inside the instance method `getWalkSpeed()`, as `super` is not accessible in the `main()` method.

Congratulations—if you understood this section, then you are prepared for the most complicated thing the exam can throw at you on `default` methods!

Using *static* Interface Methods

If you've been using an older version of Java, you might not be aware that Java now supports `static` interface methods. These methods are defined explicitly with the `static` keyword and for the most part behave just like `static` methods defined in classes.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is assumed to be `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

These rules should follow from what you know so far of classes, interfaces, and `static` methods. For example, you can't declare `static` methods without a body in classes either. Like `default` and abstract interface methods, `static` interface methods are implicitly `public` if they are declared without an access modifier. As we'll see shortly, you can use the `private` access modifier with `static` methods.

Let's take a look at a `static` interface method.

```
public interface Hop {
   static int getJumpHeight() {
      return 8;
   }
}
```

The method `getJumpHeight()` works just like a `static` method as defined in a class. In other words, it can be accessed without an instance of a class using the `Hop.getJumpHeight()` syntax. Since the method was defined without an access modifier, the compiler will automatically insert the `public` access modifier.

The fourth rule about inheritance might be a little confusing, so let's look at an example. The following is an example of a class `Bunny` that implements `Hop` and does not compile:

```
public class Bunny implements Hop {
   public void printDetails() {
      System.out.println(getJumpHeight());  // DOES NOT COMPILE
   }
}
```

Without an explicit reference to the name of the interface, the code will not compile, even though `Bunny` implements `Hop` . In this manner, the `static` interface methods are not inherited by a class implementing the interface, as they would if the method were defined in a parent class. Because `static` methods do not require an instance of the class, the problem can be easily fixed by using the interface name and calling the `public static` method.

```
public class Bunny implements Hop {
   public void printDetails() {
      System.out.println(Hop.getJumpHeight());
   }
}
```

Java "solved" the multiple inheritance problem of `static` interface methods by not allowing them to be inherited. This applies to both subinterfaces and classes that implement the interface. For example, a class that implements two interfaces containing `static` methods with the same signature will still compile. Contrast this with the behavior you saw for `default` interface methods in the previous section.

Introducing *private* Interface Methods

New to Java 9, interfaces may now include `private` interface methods. Putting on our thinking cap for a minute, what do you think `private` in-

terface methods are useful for? Since they are `private`, they cannot be used outside the interface definition. They also cannot be used in `static` interface methods without a `static` method modifier, as we'll see in the next section. With all these restrictions, why were they added to the Java language?

Give up? The answer is that `private` interface methods can be used to reduce code duplication. For example, let's say we had a `Schedule` interface with a bunch of `default` methods. In each `default` method, we want to check some value and log some information based on the `hour` value. We could copy and paste the same code into each method, or we could use a `private` interface method. Take a look at the following example:

```
public interface Schedule {
   default void wakeUp()        { checkTime(7);  }
   default void haveBreakfast() { checkTime(9);  }
   default void haveLunch()     { checkTime(12); }
   default void workOut()       { checkTime(18); }
   private void checkTime(int hour) {
      if (hour> 17) {
         System.out.println("You're late!");
      } else {
         System.out.println("You have "+(17-hour)+" hours left "
               + "to make the appointment");
      }
   }
}
```

While you can write this code without using a `private` interface method by copying the contents of the `checkTime()` method into every `default` method, it's a lot shorter and easier to read if we don't. Since the authors of Java were nice enough to add this feature for our convenience, we might as well make use of it!

The rules for `private` interface methods are pretty straightforward.

Private Interface Method Definition Rules

1. A `private` interface method must be marked with the `private` modifier and include a method body.
2. A `private` interface method may be called only by `default` and `private` (non-`static`) methods within the interface definition.

Private interface methods behave a lot like instance methods within a class. Like `private` methods in a class, they cannot be declared `abstract` since they are not inherited.

Introducing *private static* Interface Methods

Alongside `private` interface methods, Java 9 added `private static` interface methods. As you might have already guessed, the purpose of `private static` interface methods is to reduce code duplication in `static` methods within the interface declaration. Furthermore, because instance methods can access `static` methods within a class, they can also be accessed by `default` and `private` methods.

The following is an example of a `Swim` interface that uses a `private static` method to reduce code duplication within other methods declared in the interface:

```
public interface Swim {
   private static void breathe(String type) {
      System.out.println("Inhale");
      System.out.println("Performing stroke: " + type);
      System.out.println("Exhale");
   }
   static void butterfly()        { breathe("butterfly");  }
   public static void freestyle() { breathe("freestyle");  }
   default void backstroke()      { breathe("backstroke"); }
   private void breaststroke()    { breathe("breaststroke"); }
}
```

The `breathe()` method is able to be called in the `static butterfly()` and `freestyle()` methods, as well as the `default backstroke()` and `private breaststroke()` methods. Also, notice that `butterfly()` is assumed to be `public static` without any access modifier. The rules for `private static` interface methods are nearly the same as the rules for `private` interface methods.

Private Static Interface Method Definition Rules

1. A `private static` method must be marked with the `private` and `static` modifiers and include a method body.
2. A `private static` interface method may be called only by other methods within the interface definition.

Both `private` and `private static` methods can be called from `default` and `private` methods. This is equivalent to how an instance method is able to call both `static` and instance methods. On the other hand, a `private` method cannot be called from a `private static` method. This would be like trying to access an instance method from a `static` method in a class.

WHY MARK INTERFACE METHODS *PRIVATE*?

Instead of `private` and `private static` methods, we could have created `default` and `public static` methods, respectively. The code would have compiled just the same, so why mark them `private` at all?

The answer is to improve encapsulation, as we might not want these methods exposed outside the interface declaration. Encapsulation and security work best when the outside caller knows as little as possible about the internal implementation of a class or an interface. Using `private` interface methods doesn't just provide a way to reduce code duplication, but also a way to hide some of the underlying implementation details from users of the interface.

Reviewing Interface Members

We conclude our discussion of interface members with Table 12.5.

TABLE 12.5 Interface member access

	Accessible from `default` and `private` methods within the interface definition?	**Accessible from `static` methods within the interface definition?**	**Accessible from instance methods implementing or extending the interface?**	**Accessible outside the interface without an instance of interface?**
Constant variable	Yes	Yes	Yes	Yes
abstract method	Yes	No	Yes	No
default method	Yes	No	Yes	No
private method	Yes	No	No	No
static method	Yes	Yes	Yes	Yes
private static method	Yes	Yes	No	No

The first two data columns of Table 12.5 refer to access within the same interface definition. For example, a `private` method can access other `private` and `private static` methods defined within the same interface declaration.

When working with interfaces, we consider abstract, `default`, and `private` interface methods as instance methods. With that thought in mind, the last two columns of Table 12.5 should follow from what you know about class access modifiers and `private` members. Recall that instance methods can access `static` members within the class, but `static` members cannot access instance methods without a reference to the instance. Also, `private` members are never inherited, so they are never accessible directly by a class implementing an interface.

Real World Scenario

ABSTRACT CLASSES VS. INTERFACES

By introducing six different interface member types, Java has certainly blurred the lines between an abstract class and an interface. A key distinction, though, is that interfaces do not implement constructors and are not part of the class hierarchy. While a class can implement multiple interfaces, it can only directly extend a single class.

In fact, a common interview question is to ask an interviewee to describe the difference between an abstract class and an interface. These days, the question is more useful in determining which version of Java the candidate has most recently worked with. If you do happen to get this question on an interview, an appropriate tongue-in-cheek response would be, “How much time have you got?”

Introducing Functional Programming

Functional interfaces are used as the basis for lambda expressions in functional programming. 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.

A *lambda expression* is a block of code that gets passed around, sort of like an anonymous class that defines one method. As you'll see in this section, it can be written in a variety of short or long forms.

Since lambdas were part of the 1Z0-815 exam, some of this you should already know. Considering how important functional interfaces and lambda expressions are to passing the exam, you should read this section carefully, even if some of it is review.

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.

We'll cover the meaning of the `@FunctionalInterface` annotation in Chapter 13, "Annotations." For now, you just need to know that adding the annotation to a functional interface is optional.

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()` meet 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 methods, it contains only one abstract method: `reach()`.

Declaring a Functional Interface with *Object* Methods

As you may remember from your previous studies, all classes inherit certain methods from `Object`. For the exam, you should be familiar with the following `Object` method declarations:

- `String toString()`
- `boolean equals(Object)`
- `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`, then 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()`.

Real World Scenario

OVERRIDING *TOSTRING()*, *EQUALS(OBJECT)*, AND *HASHCODE()*

While knowing how to properly override `toString()`, `equals(Object)`, and `hashCode()` was part of Java certification exams prior to Java 11, this requirement was removed on all of the Java 11 exams. As a professional Java developer, it is important for you to know at least the basic rules for overriding each of these methods.

- `toString()`: The `toString()` method is called when you try to print an object or concatenate the object with a `String`. It is commonly overridden with a version that prints a unique description of the instance using its instance fields.
- `equals(Object)`: The `equals(Object)` method is used to compare objects, with the default implementation just using the `==` operator. You should override the `equals(Object)` method anytime you want to conveniently compare elements for equality, especially if this requires checking numerous fields.
- `hashCode()`: Any time you override `equals(Object)`, you must override `hashCode()` to be consistent. This means that for any two objects, if `a.equals(b)` is `true`, then `a.hashCode()==b.hashCode()` must also be `true`. If they are not consistent, then this could lead to invalid data and side effects in hash-based collections such as `HashMap` and `HashSet`.

All of these methods provide a default implementation in `Object`, but if you want to make intelligent use out of them, then you should override them.

Implementing Functional Interfaces with Lambdas

In addition to functional interfaces you write yourself, Java provides a number of predefined ones. You'll learn about many of these in Chapter 15, “Functional Programming.” For now, let's work with the `Predicate`

interface. Excluding any `static` or `default` methods defined in the interface, we have the following:

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

We'll review generics in Chapter 14, "Generics and Collections," but for now you just need to know that `<T>` allows the interface to take an object of a specified type. Now that we have a functional interface, we'll show you how to implement it using a lambda expression. The relationship between functional interfaces and lambda expressions is as follows: *any functional interface can be implemented as a lambda expression*.

Even older Java interfaces that pass the single abstract method test are functional interfaces, which can be implemented with lambda expressions.

Let's try an illustrative example. Our goal is to print out all the animals in a list according to some criteria. 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.

Now we have everything that we need to write our code to find each `Animal` that hops.

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

This program compiles and prints `kangaroo rabbit` at runtime. The `print()` method on line 14 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? We only have to add one line of code—no need for an extra class to do something simple. Here's that other line:

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

This prints `fish kangaroo turtle` at runtime. How about `Animal`s that cannot swim?

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

This prints `rabbit` by itself. The point here is that it is really easy to write code that uses lambdas once you get the basics in place.

Lambda expressions rely on the notion of deferred execution. *Deferred execution* means that code is specified now but runs later. In this case, *later* is when the `print()` method calls it. Even though the execution is deferred, the compiler will still validate that the code syntax is correct.

Writing Lambda Expressions

The syntax of lambda expressions is tricky because many parts are optional. Despite this, the overall structure is the same. The left side of the lambda expression lists the variables. It must be compatible with the type and number of input parameters of the functional interface's single abstract method.

The right side of the lambda expression represents the body of the expression. It must be compatible with the return type of the functional interface's abstract method. For example, if the abstract method returns `int`, then the lambda expression must return an `int`, a value that can be implicitly cast to an `int`, or throw an exception.

Let's take a look at a functional interface in both its short and long forms. Figure 12.1 shows the short form of this functional interface and 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

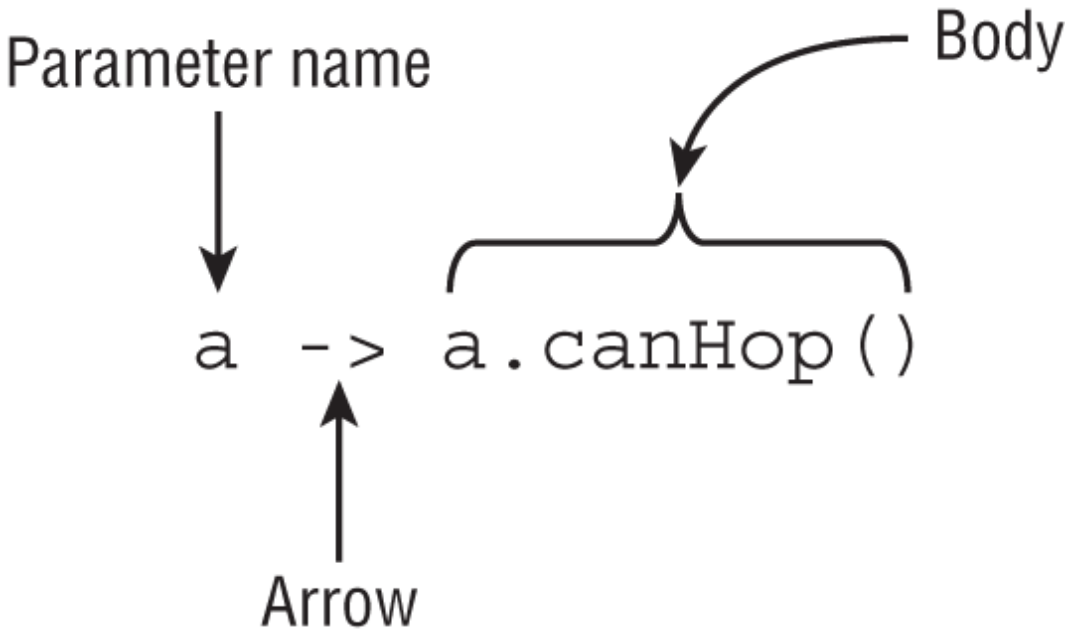

FIGURE 12.1 Lambda syntax omitting optional parts

Now let's look at a more verbose version of this lambda expression, shown in Figure 12.2. It also contains three parts.

- 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

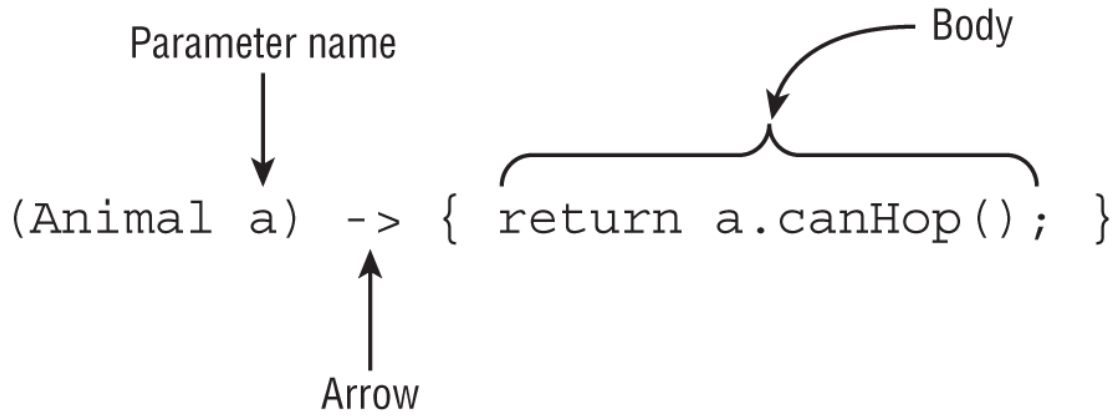

FIGURE 12.2 Lambda syntax, including optional parts

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 so 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.

As a fun fact, `s -> {}` is a valid lambda. If the return type of the functional interface method is `void`, then you don't need the semicolon or `return` statement.

Let's take a look at some examples. The following are all valid lambda expressions, assuming that there are functional interfaces that can consume them:

```
() -> new Duck()
d -> {return d.quack();}
(Duck d) -> d.quack()
(Animal a, Duck d) -> d.quack()
```

The first lambda expression could be used by a functional interface containing a method that takes no arguments and returns a `Duck` object. The second and third lambda expressions both can be used by a functional interface that takes a `Duck` as input and returns whatever the return type of `quack()` is. The last lambda expression can be used by a functional interface that takes as input `Animal` and `Duck` objects and returns whatever the return type of `quack()` is.

Now let's make sure you can identify invalid syntax. Let's assume we needed a lambda that returns a `boolean` value. Do you see what's wrong with each of these?

```
3: a, b -> a.startsWith("test")         // DOES NOT COMPILE
4: Duck d -> d.canQuack();              // DOES NOT COMPILE
5: a -> { a.startsWith("test"); }       // DOES NOT COMPILE
6: a -> { return a.startsWith("test") } // DOES NOT COMPILE
7: (Swan s, t) -> s.compareTo(t) != 0   // DOES NOT COMPILE
```

Lines 3 and 4 require parentheses around each parameter list. Remember that the parentheses are optional *only* when there is one parameter and it doesn't have a type declared. Line 5 is missing the `return` keyword, which is required since we said the lambda must return a `boolean`. Line 6 is missing the semicolon inside of the braces, `{}`. Finally, line 7 is missing the parameter type for `t`. If the parameter type is specified for one of the parameters, then it must be specified for all of them.

Working with Lambda Variables

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.

Parameter List

Earlier you learned that specifying the type of parameters is optional. Now `var` can be used in a lambda parameter list. 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 the previous example, the answer is `String`. OK, but 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() {
   test((var x) -> x>2, 123);
}

public void test(Predicate<Integer> c, int num) {
   c.test(num);
}
```

If you guessed `Integer`, you were right. The `whatAmI()` method creates a lambda to be passed to the `test()` method. Since the `test()` 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.

Restrictions on Using var in the Parameter List

While you can use `var` inside a lambda parameter list, there is a rule you need to be aware of. If `var` is used for one of the types in the parameter list, then it must be used for all parameters in the list. Given this rule, which of the following lambda expressions do not compile if they were assigned to a variable?

```
3: (var num) -> 1
4: var w -> 99
5: (var a, var b) -> "Hello"
6: (var a, Integer b) -> true
7: (String x, var y, Integer z) -> true
8: (var b, var k, var m) -> 3.14159
9: (var x, y) -> "goodbye"
```

Line 3 compiles and is similar to our previous examples. Line 4 does not compile because parentheses, `()`, are required when using the parameter name. Lines 5 and 8 compile because all of the parameters in the list use `var`. Lines 6 and 7 do not compile, though, because the parameter types include a mix of `var` and type names. Finally, line 9 does not compile because the parameter type is missing for the second parameter, `y`.

Even when using `var` for all the parameter types, each parameter type must be written out.

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;}
```

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 actually 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 use `static` variables, instance variables, and local variables if they are `final` or effectively final. Sound familiar? Lambdas follow the same rules for access as local and anonymous classes! This is not a coincidence, as behind the scenes, anonymous classes are used for lambda expressions. Let's take a look at an example:

```
4:  public class Crow {
5:     private String color;
6:     public void caw(String name) {
7:        String volume = "loudly";
8:        Predicate<String> p = s -> (name+volume+color).length()==10;
9:     }
10: }
```

On the other hand, if the local variable is not `final` or effectively final, then the code does not compile.

```
4:  public class Crow {
5:     private String color;
```

```
6:     public void caw(String name) {
7:        String volume = "loudly";
8:        color = "allowed";
9:        name = "not allowed";
10:       volume = "not allowed";
11:       Predicate<String> p =
12:          s -> (name+volume+color).length()==9; // DOES NOT COMPILE
13:    }
14: }
```

In this example, the values of `name` and `volume` are assigned new values on lines 9 and 10. For this reason, the lambda expression declared on lines 11 and 12 does not compile since it references local variables that are not `final` or effectively final. If lines 9 and 10 were removed, then the class would compile.

Summary

This chapter focused on core fundamentals of the Java language that you will use throughout this book. We started with the `final` modifier and showed how it could be applied to local, instance, and `static` variables, as well as methods and classes.

We next moved on to enumerated types, which define a list of fixed values. Like `boolean` values, enums are not integers and cannot be compared this way. Enums can be used in `switch` statements. Besides the list of values, enums can include instance variables, constructors, and methods. Methods can even be `abstract` , in which case all enum values must provide an implementation. Alternatively, if an enum method is not marked `final` , then it can be overridden by one of its value declarations.

There are four types of nested classes. An inner class requires an instance of the outer class to use, while a `static` nested class does not. A local class is one defined within a method. Local classes can access `final` and effectively final local variables. Anonymous classes are a special type of local class that does not have a name. Anonymous classes are required to

extend exactly one class by name or implement exactly one `interface`. Inner, local, and anonymous classes can access `private` members of the class in which they are defined, provided the latter two are used inside an instance method.

As of Java 9, interfaces now support six different members. Constant variables (`static final`) and `abstract` methods should have been familiar to you. Newer member types include `default`, `static`, `private`, and `private static` methods. While interfaces now contain a lot of member types, they are still distinct from abstract classes and do not participate in the class instantiation.

Last but certainly not least, this chapter included an introduction to functional interfaces and lambda expressions. A functional interface is an interface that contains exactly one abstract method. Any functional interface can be implemented with a lambda expression. A lambda expression can be written in a number of different forms, since many of the parts are optional. Make sure you understand the basics of writing lambda expressions as you will be using them throughout the book.

Exam Essentials

- **Be able to correctly apply the *final* modifier.** Applying the `final` modifier to a variable means its value cannot change after it has been assigned, although its contents can be modified. An instance `final` variable must be assigned a value when it is declared, in an instance initializer, or in a constructor at most once. A `static final` variable must be assigned a value when it is declared or in a `static` initializer. A `final` method is one that cannot be overridden by a subclass, while a `final` class is one that cannot be extended.
- **Be able to create and use enum types.** An enum is a data structure that defines a list of values. If the enum does not contain any other elements, then the semicolon (`;`) after the values is optional. An enum can have instance variables, constructors, and methods. Enum constructors are implicitly `private`. Enums can include methods, both as

members or within individual enum values. If the enum declares an `abstract` method, each enum value must implement it.

- **Identify and distinguish between types of nested classes.** There are four types of nested types: inner classes, `static` classes, local classes, and anonymous classes. The first two are defined as part of a class declaration. Local classes are used inside method bodies and scoped to the end of the current block of code. Anonymous classes are created and used once, often on the fly. More recently, they are commonly implemented as lambda expressions.
- **Be able to declare and use nested classes.** Instantiating an inner class requires an instance of the outer class, such as calling `new Outer.new Inner()`. On the other hand, `static` nested classes can be created without a reference to the outer class, although they cannot access instance members of the outer class without a reference. Local and anonymous classes cannot be declared with an access modifier. Anonymous classes are limited to extending a single class or implementing one interface.
- **Be able to create *default, static, private*, and *private static* interface methods.** A `default` interface method is a `public` interface that contains a body, which can be overridden by a class implementing the interface. If a class inherits two `default` methods with the same signature, then the class must override the `default` method with its own implementation. An interface can include `public static` and `private static` methods, the latter of which can be accessed only by methods declared within the interface. An interface can also include `private` methods, which can be called only by `default` and other `private` methods in the interface declaration.
- **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`.
- **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. If one of the parameters is a `var`, then they all must use `var`.

- **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 in a lambda expression. Class variables are always allowed. Instance variables are allowed if the lambda is used inside an instance method.

Review Questions

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

1. Which statements about the `final` modifier are correct? (Choose all that apply.)
   A. Instance and static variables can be marked `final`.
   B. A variable is effectively final if it is marked `final`.
   C. The `final` modifier can be applied to classes and interfaces.
   D. A `final` class cannot be extended.
   E. An object that is marked `final` cannot be modified.
   F. Local variables cannot be declared with type `var` and the `final` modifier.
2. What is the result of the following program?

```
public class FlavorsEnum {
   enum Flavors {
      VANILLA, CHOCOLATE, STRAWBERRY
      static final Flavors DEFAULT = STRAWBERRY;
   }
   public static void main(String[] args) {
      for(final var e : Flavors.values())
         System.out.print(e.ordinal()+" ");
   }
}
```

A. `0 1 2`

B. `1 2 3`

C. Exactly one line of code does not compile.

D. More than one line of code does not compile.

E. The code compiles but produces an exception at runtime.

F. None of the above

3. What is the result of the following code? (Choose all that apply.)

```
1:  public class Movie {
2:     private int butter = 5;
3:     private Movie() {}
4:     protected class Popcorn {
5:        private Popcorn() {}
6:        public static int butter = 10;
7:        public void startMovie() {
8:           System.out.println(butter);
9:        }
10:    }
11:    public static void main(String[] args) {
12:       var movie = new Movie();
13:       Movie.Popcorn in = new Movie().new Popcorn();
14:       in.startMovie();
15:    } }
```

A. The output is `5`.

B. The output is `10`.

C. Line 6 generates a compiler error.

D. Line 12 generates a compiler error.

E. Line 13 generates a compiler error.

F. The code compiles but produces an exception at runtime.

4. Which statements about `default` and `private` interface methods are correct? (Choose all that apply.)

A. A `default` interface method can be declared `private`.

B. A `default` interface method can be declared `public`.

C. A `default` interface method can be declared `static`.

D. A `private` interface method can be declared `abstract`.

E. A `private` interface method can be declared `protected`.

F. A `private` interface method can be declared `static`.

5. Which of the following are valid lambda expressions? (Choose all that apply.)

A. `(Wolf w, var c) -> 39`

B. `(final Camel c) -> {}`

C. `(a,b,c) -> {int b = 3; return 2;}`

D. `(x,y) -> new RuntimeException()`

E. `(var y) -> return 0;`

F. `() -> {float r}`

G. `(Cat a, b) -> {}`

6. What are some advantages of using `private` interface methods? (Choose all that apply.)

A. Improve polymorphism

B. Improve performance at runtime

C. Reduce code duplication

D. Backward compatibility

E. Encapsulate interface implementation

F. Portability

7. What is the result of the following program?

```
public class IceCream {
   enum Flavors {
      CHOCOLATE, STRAWBERRY, VANILLA
   }

   public static void main(String[] args) {
      Flavors STRAWBERRY = null;
      switch (STRAWBERRY) {
         case Flavors.VANILLA: System.out.print("v");
         case Flavors.CHOCOLATE: System.out.print("c");
         case Flavors.STRAWBERRY: System.out.print("s");
         break;
         default: System.out.println("missing flavor"); }
   }
}
```

A. `v`

B. `vc`

C. `s`

D. `missing flavor`

E. Exactly one line of code does not compile.

F. More than one line of code does not compile.

G. The code compiles but produces an exception at runtime.

8. Which statements about functional interfaces are true? (Choose all that apply.)

A. A functional interface can contain `default` and `private` methods.

B. A functional interface can be defined by a class or interface.

C. Abstract methods with signatures that are contained in `public` methods of `java.lang.Object` do not count toward the abstract method count for a functional interface.

D. A functional interface cannot contain `static` or `private static` methods.

E. A functional interface contains at least one abstract method.

F. A functional interface must be marked with the `@FunctionalInterface` annotation.

9. Which lines, when entered independently into the blank, allow the code to print `Not scared` at runtime? (Choose all that apply.)

```
public class Ghost {
   public static void boo() {
      System.out.println("Not scared");
   }
   protected final class Spirit {
      public void boo() {
         System.out.println("Booo!!!");
      }
   }
   public static void main(String… haunt) {
      var g = new Ghost().new Spirit() {};
      _____________________________;
```

```
   }
}
```

A. `g.boo()`
B. `g.super.boo()`
C. `new Ghost().boo()`
D. `g.Ghost.boo()`
E. `new Spirit().boo()`
F. `Ghost.boo()`
G. None of the above

10. The following code appears in a file named `Ostrich.java`. What is the result of compiling the source file?

```
1: public class Ostrich {
2:    private int count;
3:    private interface Wild {}
4:    static class OstrichWrangler implements Wild {
5:       public int stampede() {
6:          return count;
7:       } } }
```

A. The code compiles successfully, and one bytecode file is generated: `Ostrich.class`.
B. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `OstrichWrangler.class`.
C. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `Ostrich$OstrichWrangler.class`.
D. A compiler error occurs on line 4.
E. A compiler error occurs on line 6.

11. What is the result of the following code?

```
1:  public interface CanWalk {
2:     default void walk() { System.out.print("Walking"); }
3:     private void testWalk() {}
4:  }
5:  public interface CanRun {
```

```
6:     abstract public void run();
7:     private void testWalk() {}
8:     default void walk() { System.out.print("Running"); }
9:  }
10: public interface CanSprint extends CanWalk, CanRun {
11:    void sprint();
12:    default void walk(int speed) {
13:       System.out.print("Sprinting");
14:    }
15:    private void testWalk() {}
16: }
```

A. The code compiles without issue.

B. The code will not compile because of line 6.

C. The code will not compile because of line 8.

D. The code will not compile because of line 10.

E. The code will not compile because of line 12.

F. None of the above

12. What is the result of executing the following program?

```
interface Sing {
   boolean isTooLoud(int volume, int limit);
}
public class OperaSinger {
   public static void main(String[] args) {
      check((h, l) -> h.toString(), 5);  // m1
   }
   private static void check(Sing sing, int volume) {
      if (sing.isTooLoud(volume, 10))   // m2
         System.out.println("not so great");
      else System.out.println("great");
   }
}
```

A. `great`

B. `not so great`

C. Compiler error on line `m1`

D. Compiler error on line `m2`

E. Compiler error on a different line

F. A runtime exception is thrown.

13. Which lines of the following interface declaration do not compile? (Choose all that apply.)

```
1: public interface Herbivore {
2:    int amount = 10;
3:    static boolean gather = true;
4:    static void eatGrass() {}
5:    int findMore() { return 2; }
6:    default float rest() { return 2; }
7:    protected int chew() { return 13; }
8:    private static void eatLeaves() {}
9: }
```

A. All of the lines compile without issue.

B. Line 2

C. Line 3

D. Line 4

E. Line 5

F. Line 6

G. Line 7

H. Line 8

14. What is printed by the following program?

```
public class Deer {
   enum Food {APPLES, BERRIES, GRASS}
   protected class Diet {
      private Food getFavorite() {
         return Food.BERRIES;
      }
   }
   public static void main(String[] seasons) {
      switch(new Diet().getFavorite()) {
         case APPLES: System.out.print("a");
         case BERRIES: System.out.print("b");
```

```
                default: System.out.print("c");
            }
        }
    }
```

A. `b`

B. `bc`

C. `abc`

D. The code declaration of the `Diet` class does not compile.

E. The `main()` method does not compile.

F. The code compiles but produces an exception at runtime.

G. None of the above

15. Which of the following are printed by the `Bear` program? (Choose all that apply.)

```
public class Bear {
   enum FOOD {
      BERRIES, INSECTS {
         public boolean isHealthy() { return true; }},
      FISH, ROOTS, COOKIES, HONEY;
      public abstract boolean isHealthy();
   }
   public static void main(String[] args) {
      System.out.print(FOOD.INSECTS);
      System.out.print(FOOD.INSECTS.ordinal());
      System.out.print(FOOD.INSECTS.isHealthy());
      System.out.print(FOOD.COOKIES.isHealthy());
   }
}
```

A. `insects`

B. `INSECTS`

C. `0`

D. `1`

E. `false`

F. `true`

G. The code does not compile.

16. Which of the following are valid functional interfaces? (Choose all that apply.)

```
public interface Transport {
   public int go();
   public boolean equals(Object o);
}

public abstract class Car {
   public abstract Object swim(double speed, int duration);
}

public interface Locomotive extends Train {
   public int getSpeed();
}

public interface Train extends Transport {}

abstract interface Spaceship extends Transport {
   default int blastOff();
}

public interface Boat {
   int hashCode();
   int hashCode(String input);
}
```

A. `Boat`
B. `Car`
C. `Locomotive`
D. `Tranport`
E. `Train`
F. `Spaceship`
G. None of these is a valid functional interface.

17. Which lambda expression when entered into the blank line in the following code causes the program to print `hahaha` ? (Choose all that

apply.)

```
import java.util.function.Predicate;
public class Hyena {
   private int age = 1;
   public static void main(String[] args) {
      var p = new Hyena();
      double height = 10;
      int age = 1;
      testLaugh(p, _____________________);
      age = 2;
   }
   static void testLaugh(Hyena panda, Predicate<Hyena> joke) {
      var r = joke.test(panda) ? "hahaha" : "silence";
      System.out.print(r);
   }
}
```

A. `var -> p.age <= 10`

B. `shenzi -> age==1`

C. `p -> true`

D. `age==1`

E. `shenzi -> age==2`

F. `h -> h.age < 5`

G. None of the above, as the code does not compile.

18. Which of the following can be inserted in the `rest()` method? (Choose all that apply.)

```
public class Lion {
   class Cub {}
   static class Den {}
   static void rest() {
   _____________________;
   } }
```

A. `Cub a = Lion.new Cub()`

B. `Lion.Cub b = new Lion().Cub()`

C. `Lion.Cub c = new Lion().new Cub()`

D. `var d = new Den()`

E. `var e = Lion.new Cub()`

F. `Lion.Den f = Lion.new Den()`

G. `Lion.Den g = new Lion.Den()`

H. `var h = new Cub()`

19. Given the following program, what can be inserted into the blank line that would allow it to print `Swim!` at runtime?

```
interface Swim {
   default void perform() { System.out.print("Swim!"); }
}
interface Dance {
   default void perform() { System.out.print("Dance!"); }
}
public class Penguin implements Swim, Dance {
   public void perform() { System.out.print("Smile!"); }
   private void doShow() {
      ____________________
   }
   public static void main(String[] eggs) {
      new Penguin().doShow();
   }
}
```

A. `super.perform();`

B. `Swim.perform();`

C. `super.Swim.perform();`

D. `Swim.super.perform();`

E. The code does not compile regardless of what is inserted into the blank.

F. The code compiles, but due to polymorphism, it is not possible to produce the requested output without creating a new object.

20. Which statements about effectively final variables are true? (Choose all that apply.)

A. The value of an effectively final variable is not modified after it is set.
B. A lambda expression can reference effectively final variables.
C. A lambda expression can reference `final` variables.
D. If the `final` modifier is added, the code still compiles.
E. Instance variables can be effectively final.
F. Static variables can be effectively final.

21. Which lines of the following interface do not compile? (Choose all that apply.)

```
1: public interface BigCat {
2:    abstract String getName();
3:    static int hunt() { getName(); return 5; }
4:    default void climb() { rest(); }
5:    private void roar() { getName();  climb(); hunt(); }
6:    private static boolean sneak() { roar(); return true; }
7:    private int rest() { return 2; };
8: }
```

A. Line 2
B. Line 3
C. Line 4
D. Line 5
E. Line 6
F. Line 7
G. None of the above

22. What are some advantages of using `default` interface methods? (Choose all that apply.)
A. Automatic resource management
B. Improve performance at runtime
C. Better exception handling
D. Backward compatibility
E. Highly concurrent execution
F. Convenience in classes implementing the interface

23. Which statements about the following enum are true? (Choose all that apply.)

```
1:  public enum AnimalClasses {
2:     MAMMAL(true), INVERTIBRATE(Boolean.FALSE), BIRD(false),
3:     REPTILE(false), AMPHIBIAN(false), FISH(false) {
4:        public int swim() { return 4; }
5:     }
6:     final boolean hasHair;
7:     public AnimalClasses(boolean hasHair) {
8:        this.hasHair = hasHair;
9:     }
10:    public boolean hasHair() { return hasHair; }
11:    public int swim() { return 0; }
12: }
```

A. Compiler error on line 2
B. Compiler error on line 3
C. Compiler error on line 7
D. Compiler error on line 8
E. Compiler error on line 10
F. Compiler error on another line
G. The code compiles successfully.

24. Which lambdas can replace the `new Sloth()` call in the `main()` method and produce the same output at runtime? (Choose all that apply.)

```
import java.util.List;
interface Yawn {
   String yawn(double d, List<Integer> time);
}
class Sloth implements Yawn {
   public String yawn(double zzz, List<Integer> time) {
      return "Sleep: " + zzz;
   }
}
public class Vet {
```

```
   public static String takeNap(Yawn y) {
      return y.yawn(10, null);
   }
   public static void main(String… unused) {
      System.out.print(takeNap(new Sloth()));
   }
}
```

A. `(z,f) -> { String x = ""; return "Sleep: " + x }`

B. `(t,s) -> { String t = ""; return "Sleep: " + t; }`

C. `(w,q) -> {"Sleep: " + w}`

D. `(e,u) -> { String g = ""; "Sleep: " + e }`

E. `(a,b) -> "Sleep: " + (double)(b==null ? a : a)`

F. `(r,k) -> { String g = ""; return "Sleep:"; }`

G. None of the above, as the program does not compile.

25. What does the following program print?

```
1:  public class Zebra {
2:     private int x = 24;
3:     public int hunt() {
4:        String message = "x is ";
5:        abstract class Stripes {
6:           private int x = 0;
7:           public void print() {
8:              System.out.print(message + Zebra.this.x);
9:           }
10:       }
11:       var s = new Stripes() {};
12:       s.print();
13:       return x;
14:    }
15:    public static void main(String[] args) {
16:       new Zebra().hunt();
17:    } }
```

A. `x is 0`

B. `x is 24`

C. Line 6 generates a compiler error.

D. Line 8 generates a compiler error.

E. Line 11 generates a compiler error.

F. None of the above