# Chapter 7
# Methods and Encapsulation

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

- **Creating and Using Methods**
  - Create methods and constructors with arguments and return values
  - Create and invoke overloaded methods
  - Apply the static keyword to methods and fields
- **Applying Encapsulation**
  - Apply access modifiers
  - Apply encapsulation principles to a class

In previous chapters, you learned how to use methods without examining them in detail. In this chapter, you'll explore methods in depth, including overloading. This chapter discusses instance variables, access modifiers, and encapsulation.

## Designing Methods

Every interesting Java program we've seen has had a `main()` method. You can write other methods, too. For example, you can write a basic method to take a nap, as shown in Figure 7.1.

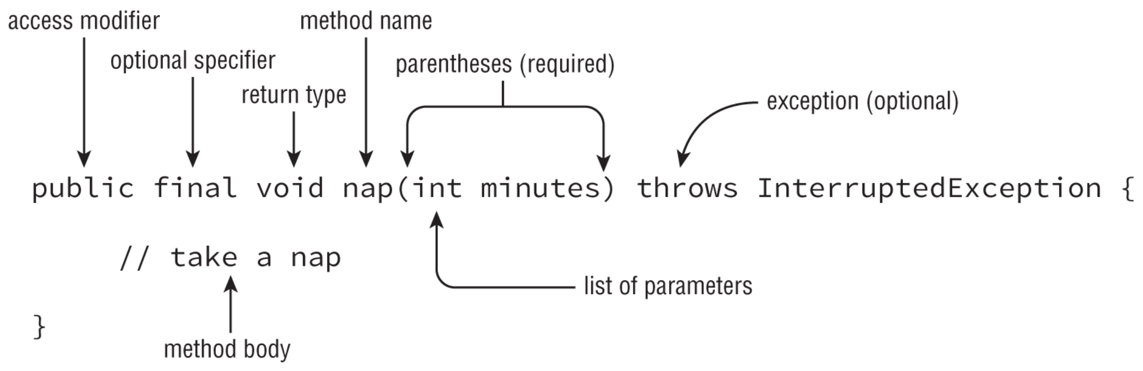

**FIGURE 7.1** **Method declaration**

This is called a *method declaration*, which specifies all the information needed to call the method. There are a lot of parts, and we'll cover each one in more detail. Two of the parts—the method name and parameter list—are called the *method signature*.

Table 7.1 is a brief reference to the elements of a method declaration. Don't worry if it seems like a lot of information—by the time you finish this chapter, it will all fit together.

TABLE 7.1 **Parts of a method declaration**

| **Element** | **Value in `nap()` example** | **Required?** |
|---|---|---|
| Access modifier | `public` | No |
| Optional specifier | `final` | No |
| Return type | `void` | Yes |
| Method name | `nap` | Yes |
| Parameter list | `(int minutes)` | Yes, but can be empty parentheses |
| Optional exception list | `throws InterruptedException` | No |
| Method body* | `{`<br>`// take a nap`<br>`}` | Yes, but can be empty braces |

* Body omitted for `abstract` methods, which we will cover in the later in the book.

To call this method, just type its name, followed by a single `int` value in parentheses:

```
nap(10);
```

Let's start by taking a look at each of these parts of a basic method.

## Access Modifiers

Java offers four choices of access modifier:

***private*** The `private` modifier means the method can be called only from within the same class.

**Default (Package-Private) Access** With default access, the method can be called only from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

***protected*** The `protected` modifier means the method can be called only from classes in the same package or subclasses. You'll learn about subclasses in Chapter 8, "Class Design."

***public*** The `public` modifier means the method can be called from any class.

NOTE

There's a `default` keyword in Java. You saw it in the `switch` statement in Chapter 4, "Making Decisions," and you'll see it again in the Chapter 9, "Advanced Class Design," when I discuss interfaces. It's not used for access control.

We'll explore the impact of the various access modifiers later in this chapter. For now, just master identifying valid syntax of methods. The exam creators like to trick you by putting method elements in the wrong order or using incorrect values.

We'll see practice examples as we go through each of the method elements in this section. Make sure you understand why each of these is a valid or invalid method declaration. Pay attention to the access modifiers as you figure out what is wrong with the ones that don't compile when inserted into a class:

```
public void walk1() {}
default void walk2() {} // DOES NOT COMPILE
void public walk3() {}  // DOES NOT COMPILE
void walk4() {}
```

The `walk1()` method is a valid declaration with public access. The `walk4()` method is a valid declaration with default access. The `walk2()` method doesn't compile because `default` is not a valid access modifier. The `walk3()` method doesn't compile because the access modifier is specified after the return type.

## Optional Specifiers

There are a number of optional specifiers, but most of them aren't on the exam. Optional specifiers come from the following list. Unlike with access modifiers, you can have multiple specifiers in the same method (although not all combinations are legal). When this happens, you can specify them in any order. And since these specifiers are optional, you are allowed to not have any of them at all. This means you can have zero or more specifiers in a method declaration.

***static*** The `static` modifier is used for class methods and will be covered later in this chapter.

***abstract*** The `abstract` modifier is used when a method body is not provided. It will be covered in Chapter 9.

***final*** The `final` modifier is used when a method is not allowed to be overridden by a subclass. It will also be covered in Chapter 8.

***synchronized*** The `synchronized` modifier is used with multithreaded code. It is on the 1Z0-816 exam, but not the 1Z0-815 exam.

***native*** The `native` modifier is used when interacting with code written in another language such as C++. It is not on either OCP 11 exam.

***strictfp*** The `strictfp` modifier is used for making floating-point calculations portable. It is not on either OCP 11 exam.

Again, just focus on syntax for now. Do you see why these compile or don't compile?

```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {}       // DOES NOT COMPILE
public void final walk6() {}          // DOES NOT COMPILE
final public void walk7() {}
```

The `walk1()` method is a valid declaration with no optional specifier. This is okay—it is optional after all. The `walk2()` method is a valid declaration, with `final` as the optional specifier. The `walk3()` and `walk4()` methods are valid declarations with both `final` and `static` as optional specifiers. The order of these two keywords doesn't matter. The `walk5()` method doesn't compile because `modifier` is not a valid optional specifier. The `walk6()` method doesn't compile because the optional specifier is after the return type.

The `walk7()` method does compile. Java allows the optional specifiers to appear before the access modifier. This is a weird case and not one you need to know for the exam. We are mentioning it so you don't get confused when practicing.

## Return Type

The next item in a method declaration is the return type. The return type might be an actual Java type such as `String` or `int`. If there is no return type, the `void` keyword is used. This special return type comes from the English language: *void* means without contents. In Java, there is no type there.

---

NOTE

Remember that a method must have a return type. If no value is returned, the return type is `void`. You cannot omit the return type.

---

When checking return types, you also have to look inside the method body. Methods with a return type other than `void` are required to have a `return` statement inside the method body. This `return` statement must include the primitive or object to be returned. Methods that have a return type of `void` are permitted to have a `return` statement with no value returned or omit the `return` statement entirely.

Ready for some examples? Can you explain why these methods compile or don't?

```
public void walk1() {}
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() {}                          // DOES NOT COMPILE
public walk5() {}                                 // DOES NOT COMPILE
public String int walk6() { }                     // DOES NOT COMPILE
String walk7(int a) { if (a == 4) return ""; } // DOES NOT COMPILE
```

Since the return type of the `walk1()` method is `void`, the `return` statement is optional. The `walk2()` method shows the optional `return` statement that correctly doesn't return anything. The `walk3()` method is a valid declaration with a `String` return type and a `return` statement that returns a `String`. The `walk4()` method doesn't compile because the `return` statement is missing. The `walk5()` method doesn't compile because the return type is missing. The `walk6()` method doesn't compile because it attempts to use two return types. You get only one return type.

The `walk7()` method is a little tricky. There is a `return` statement, but it doesn't always get run. If `a` is 6, the `return` statement doesn't get executed. Since the `String` always needs to be returned, the compiler complains.

When returning a value, it needs to be assignable to the return type. Imagine there is a local variable of that type to which it is assigned before being returned. Can you think of how to add a line of code with a local variable in these two methods?

```
int integer() {
   return 9;
}
int longMethod() {
   return 9L; // DOES NOT COMPILE
}
```

It is a fairly mechanical exercise. You just add a line with a local variable. The type of the local variable matches the return type of the method. Then you return that local variable instead of the value directly:

```
int integerExpanded() {
   int temp = 9;
   return temp;
}
int longExpanded() {
   int temp = 9L; // DOES NOT COMPILE
   return temp;
}
```

This shows more clearly why you can't return a `long` primitive in a method that returns an `int`. You can't stuff that `long` into an `int` variable, so you can't return it directly either.

### Method Name

Method names follow the same rules as we practiced with variable names in Chapter 2, "Java Building Blocks." To review, an identifier may only contain letters, numbers, `$`, or `_`. Also, the first character is not allowed to be a number, and reserved words are not allowed. Finally, the single underscore character is not allowed. By convention, methods begin with a lowercase letter but are not required to. Since this is a review of Chapter 2, we can jump right into practicing with some examples:

```
public void walk1() {}
public void 2walk() {}   // DOES NOT COMPILE
public walk3 void() {}   // DOES NOT COMPILE
public void Walk_$() {}
public _() {}            // DOES NOT COMPILE
public void() {}         // DOES NOT COMPILE
```

The `walk1()` method is a valid declaration with a traditional name. The `2walk()` method doesn't compile because identifiers are not allowed to begin with numbers. The `walk3()` method doesn't compile because the method name is before the return type. The `Walk_$()` method is a valid declaration. While it certainly isn't good practice to start a method name with a capital letter and end with punctuation, it is legal. The `_` method is not allowed since it consists of a single underscore. The final line of code doesn't compile because the method name is missing.

### Parameter List

Although the parameter list is required, it doesn't have to contain any parameters. This means you can just have an empty pair of parentheses after the method name, as follows:

```
void nap(){}
```

If you do have multiple parameters, you separate them with a comma. There are a couple more rules for the parameter list that you'll see when we cover varargs shortly. For now, let's practice looking at method declaration with "regular" parameters:

```
public void walk1() {}
public void walk2 {}                // DOES NOT COMPILE
public void walk3(int a) {}
public void walk4(int a; int b) {}  // DOES NOT COMPILE
public void walk5(int a, int b) {}
```

The `walk1()` method is a valid declaration without any parameters. The `walk2()` method doesn't compile because it is missing the parentheses around the parameter list. The `walk3()` method is a valid declaration with one parameter. The `walk4()` method doesn't compile because the parameters are separated by a semicolon rather than a comma. Semicolons are for separating statements, not for parameter lists. The `walk5()` method is a valid declaration with two parameters.

### Optional Exception List

In Java, code can indicate that something went wrong by throwing an exception. We'll cover this in Chapter 10, "Exceptions." For now, you just need to know that it is optional and where in the method declaration it goes if present. For example, `InterruptedException` is a type of `Exception`. You can list as many types of exceptions as you want in this clause separated by commas. Here's an example:

```
public void zeroExceptions() {}
public void oneException() throws IllegalArgumentException {}
public void twoExceptions() throws
   IllegalArgumentException, InterruptedException {}
```

You might be wondering what methods do with these exceptions. The calling method can throw the same exceptions or handle them. You'll learn more about this in Chapter 10.

### Method Body

The final part of a method declaration is the method body (except for abstract methods and interfaces, but you don't need to know about either of those yet). A method body is simply a code block. It has braces that contain zero or more Java statements. We've spent several chapters looking at Java statements by now, so you should find it easy to figure out why these compile or don't:

```
public void walk1() {}
public void walk2()     // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

The `walk1()` method is a valid declaration with an empty method body. The `walk2()` method doesn't compile because it is missing the braces around the empty method body. The `walk3()` method is a valid declaration with one statement in the method body.

You've made it through the basics of identifying correct and incorrect method declarations. Now you can delve into more detail.

## Working with Varargs

As you saw in Chapter 5, "Core Java APIs," a method may use a varargs parameter (variable argument) as if it is an array. It is a little different than an array, though. A varargs parameter must be the last element in a method's parameter list. This means you are allowed to have only one varargs parameter per method.

Can you identify why each of these does or doesn't compile? (Yes, there is a lot of practice in this chapter. You have to be really good at identifying valid and invalid methods for the exam.)

```
public void walk1(int... nums) {}
public void walk2(int start, int... nums) {}
public void walk3(int... nums, int start) {}    // DOES NOT COMPILE
public void walk4(int... start, int... nums) {} // DOES NOT COMPILE
```

The `walk1()` method is a valid declaration with one varargs parameter. The `walk2()` method is a valid declaration with one `int` parameter and one varargs parameter. The `walk3()` and `walk4()` methods do not compile because they have a varargs parameter in a position that is not the last one.

When calling a method with a varargs parameter, you have a choice. You can pass in an array, or you can list the elements of the array and let Java create it for you. You can even omit the varargs values in the method call and Java will create an array of length zero for you.

Finally! You get to do something other than identify whether method declarations are valid. Instead, you get to look at method calls. Can you figure out why each method call outputs what it does?

```
15: public static void walk(int start, int... nums) {
16:    System.out.println(nums.length);
17: }
18: public static void main(String[] args) {
19:    walk(1);                          // 0
20:    walk(1, 2);                       // 1
21:    walk(1, 2, 3);                    // 2
22:    walk(1, new int[] {4, 5});        // 2
23: }
```

Line 19 passes 1 as `start` but nothing else. This means Java creates an array of length 0 for `nums`. Line 20 passes 1 as `start` and one more value. Java converts this one value to an array of length 1. Line 21 passes 1 as `start` and two more values. Java converts these two values to an array of length 2. Line 22 passes 1 as `start` and an array of length 2 directly as `nums`.

You've seen that Java will create an empty array if no parameters are passed for a vararg. However, it is still possible to pass `null` explicitly:

```
walk(1, null);     // throws a NullPointerException in walk()
```

Since `null` isn't an `int`, Java treats it as an array reference that happens to be `null`. It just passes on the `null` array object to `walk`. Then the `walk()` method throws an exception because it tries to determine the length of `null`.

Accessing a varargs parameter is just like accessing an array. It uses array indexing. Here's an example:

```
16: public static void run(int... nums) {
17:    System.out.println(nums[1]);
18: }
19: public static void main(String[] args) {
20:    run(11, 22);     // 22
21: }
```

Line 20 calls a varargs method with two parameters. When the method gets called, it sees an array of size 2. Since indexes are 0 based, 22 is printed.

## Applying Access Modifiers

You already saw that there are four access modifiers: `public`, `private`, `protected`, and default access. We are going to discuss them in order from most restrictive to least restrictive:

- `private`: Only accessible within the same class

- Default (package-private) access: `private` plus other classes in the same package
- `protected` : Default access plus child classes
- `public` : `protected` plus classes in the other packages

We will explore the impact of these four levels of access on members of a class. As you learned in Chapter 1, “Welcome to Java,” a member is an instance variable or instance method.

## Private Access

Private access is easy. Only code in the same class can call private methods or access private fields.

First, take a look at Figure 7.2. It shows the classes you’ll use to explore private and default access. The big boxes are the names of the packages. The smaller boxes inside them are the classes in each package. You can refer back to this figure if you want to quickly see how the classes relate.

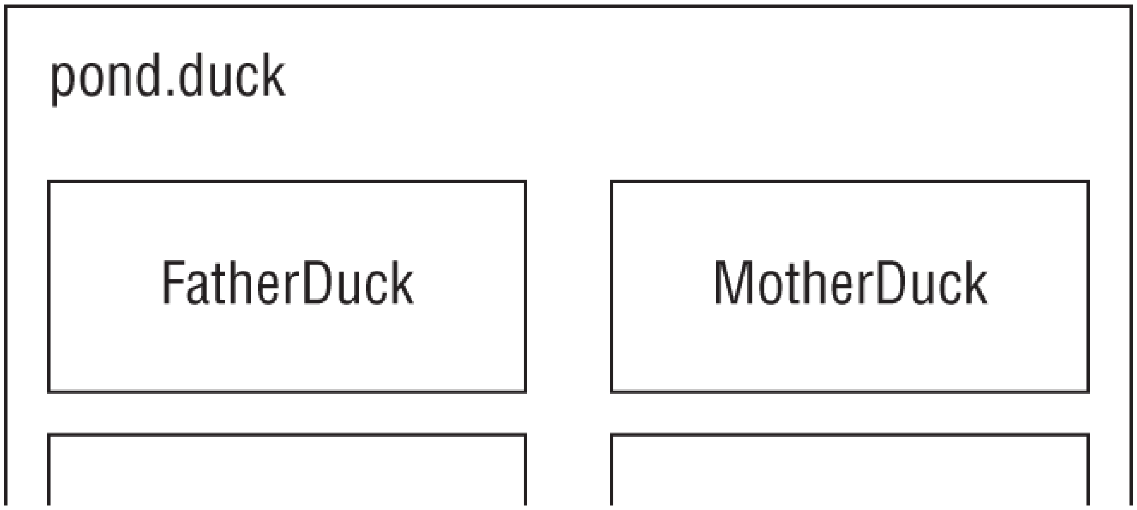

**FIGURE 7.2** **Classes used to show private and default access**

This is perfectly legal code because everything is one class:

```
package pond.duck;
public class FatherDuck {
   private String noise = "quack";
   private void quack() {
      System.out.println(noise);     // private access is ok
   }
   private void makeNoise() {
      quack();                       // private access is ok
   } }
```

So far, so good. `FatherDuck` makes a call to private method `quack()` on line 8 and uses private instance variable `noise` on line 5.

Now we add another class:

```
1: package pond.duck;
2: public class BadDuckling {
3:    public void makeNoise() {
4:       FatherDuck duck = new FatherDuck();
5:       duck.quack();                          // DOES NOT COMPILE
6:       System.out.println(duck.noise);     // DOES NOT COMPILE
7:    } }
```

`BadDuckling` is trying to access an instance variable and a method it has no business touching. On line 5, it tries to access a private method in another class. On line 6, it tries to access a private instance variable in another class. Both generate compiler errors. Bad duckling!

Our bad duckling is only a few days old and doesn't know better yet. Luckily, you know that accessing private members of other classes is not allowed and you need to use a different type of access.

## Default (Package-Private) Access

Luckily, `MotherDuck` is more accommodating about what her ducklings can do. She allows classes in the same package to access her members. When there is no access modifier, Java uses the default, which is package-private access. This means that the member is "private" to classes in the same package. In other words, only classes in the package may access it.

```
package pond.duck;
public class MotherDuck {
   String noise = "quack";
   void quack() {
      System.out.println(noise);     // default access is ok
   }
   private void makeNoise() {
      quack();                       // default access is ok
   }
}
```

`MotherDuck` can refer to `noise` and call `quack()`. After all, members in the same class are certainly in the same package. The big difference is `MotherDuck` lets other classes in the same package access members (due to being package-private), whereas `FatherDuck` doesn't (due to being private). `GoodDuckling` has a much better experience than `BadDuckling`:

```
package pond.duck;
public class GoodDuckling {
   public void makeNoise() {
      MotherDuck duck = new MotherDuck();
      duck.quack();                             // default access
      System.out.println(duck.noise);           // default access
   }
}
```

`GoodDuckling` succeeds in learning to `quack()` and make `noise` by copying its mother. Notice that all the classes covered so far are in the same package `pond.duck`. This allows default (package-private) access to work.

In this same pond, a swan just gave birth to a baby swan. A baby swan is called a *cygnet*. The cygnet sees the ducklings learning to quack and decides to learn from `MotherDuck` as well.

```
package pond.swan;
import pond.duck.MotherDuck;              // import another package
public class BadCygnet {
   public void makeNoise() {
      MotherDuck duck = new MotherDuck();
      duck.quack();                       // DOES NOT COMPILE
      System.out.println(duck.noise);     // DOES NOT COMPILE
    }
  }
```

Oh no! `MotherDuck` only allows lessons to other ducks by restricting access to the `pond.duck` package. Poor little `BadCygnet` is in the `pond.swan` package, and the code doesn't compile.

Remember that when there is no access modifier on a member, only classes in the same package can access the member.

### Protected Access

Protected access allows everything that default (package-private) access allows and more. The `protected` access modifier adds the ability to access members of a parent class. We'll cover creating subclasses in depth in Chapter 8. For now, we'll cover the simplest possible use of a child class.

Figure 7.3 shows the many classes we will create in this section. There are a number of classes and packages, so don't worry about keeping them all in your head. Just check back with this figure as you go.

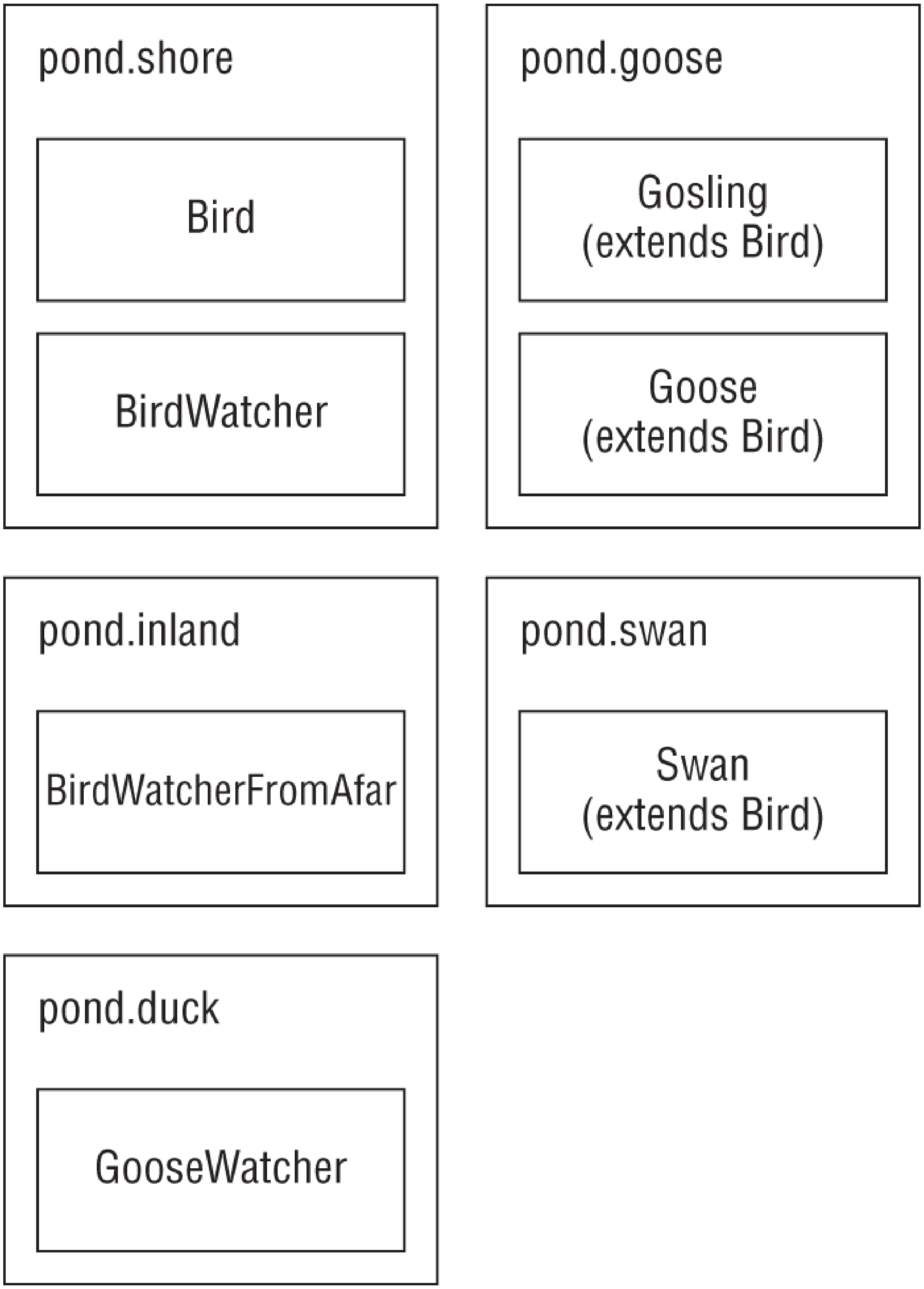

**FIGURE 7.3 Classes used to show protected access**

First, create a `Bird` class and give `protected` access to its members:

```
package pond.shore;
public class Bird {
   protected String text = "floating";           // protected access
   protected void floatInWater() {               // protected access
      System.out.println(text);
   }
}
```

Next, we create a subclass:

```
   package pond.goose;
   import pond.shore.Bird;                // in a different package
   public class Gosling extends Bird {    // extends means create subclass
      public void swim() {
         floatInWater();                  // calling protected member
         System.out.println(text);        // accessing protected member
      }
   }
```

This is a simple subclass. It *extends* the `Bird` class. Extending means creating a subclass that has access to any `protected` or `public` members of the parent class. Running this code prints `floating` twice: once from calling `floatInWater()` , and once from the print statement in `swim()` . Since `Gosling` is a subclass of `Bird` , it can access these members even though it is in a different package.

Remember that `protected` also gives us access to everything that default access does. This means that a class in the same package as `Bird` can access its `protected` members.

```
   package pond.shore;                       // same package as Bird
   public class BirdWatcher {
      public void watchBird() {
         Bird bird = new Bird();
         bird.floatInWater();                // calling protected member
         System.out.println(bird.text);      // accessing protected member
      }
   }
```

Since `Bird` and `BirdWatcher` are in the same package, `BirdWatcher` can access members of the `bird` variable. The definition of `protected` allows access to subclasses and classes in the same package. This example uses the same package part of that definition.

Now let's try the same thing from a different package:

```
package pond.inland;
import pond.shore.Bird;               // different package than Bird
public class BirdWatcherFromAfar {
   public void watchBird() {
      Bird bird = new Bird();
      bird.floatInWater();               // DOES NOT COMPILE
      System.out.println(bird.text);     // DOES NOT COMPILE
   }
}
```

`BirdWatcherFromAfar` is not in the same package as `Bird`, and it doesn't inherit from `Bird`. This means that it is not allowed to access `protected` members of `Bird`.

Got that? Subclasses and classes in the same package are the only ones allowed to access `protected` members.

There is one gotcha for `protected` access. Consider this class:

```
package pond.swan;
import pond.shore.Bird;     // in different package than Bird
public class Swan extends Bird {     // but subclass of Bird
   public void swim() {
      floatInWater();              // subclass access to superclass
      System.out.println(text);    // subclass access to superclass
   }
   public void helpOtherSwanSwim() {
      Swan other = new Swan();
      other.floatInWater();        // subclass access to superclass
      System.out.println(other.text);  // subclass access
12:                                       // to superclass
13:    }
14:    public void helpOtherBirdSwim() {
15:       Bird other = new Bird();
16:       other.floatInWater();               // DOES NOT COMPILE
17:       System.out.println(other.text);     // DOES NOT COMPILE
18:    }
19: }
```

Take a deep breath. This is interesting. `Swan` is not in the same package as `Bird` but does extend it—which implies it has access to the `protected` members of `Bird` since it is a subclass. And it does. Lines 5 and 6 refer to `protected` members via inheriting them.

Lines 10 and 11 also successfully use `protected` members of `Bird`. This is allowed because these lines refer to a `Swan` object. `Swan` inherits from `Bird`, so this is okay. It is sort of a two-phase check. The `Swan` class is allowed to use `protected` members of `Bird`, and we are referring to a `Swan` object. Granted, it is a `Swan` object created on line 9 rather than an inherited one, but it is still a `Swan` object.

Lines 16 and 17 do *not* compile. Wait a minute. They are almost exactly the same as lines 10 and 11! There's one key difference. This time a `Bird` reference is used rather than inheritance. It is created on line 15. `Bird` is in a different package, and this code isn't inheriting from `Bird`, so it doesn't get to use `protected` members. Say what now? We just got through saying repeatedly that `Swan` inherits from `Bird`. And it does. However, the variable reference isn't a `Swan`. The code just happens to be in the `Swan` class.

It's okay to be confused. This is arguably one of the most confusing points on the exam. Looking at it a different way, the `protected` rules apply under two scenarios:

- A member is used without referring to a variable. This is the case on lines 5 and 6. In this case, we are taking advantage of inheritance and `protected` access is allowed.

- A member is used through a variable. This is the case on lines 10, 11, 16, and 17. In this case, the rules for the reference type of the variable are what matter. If it is a subclass, `protected` access is allowed. This works for references to the same class or a subclass.

We're going to try this again to make sure you understand what is going on. Can you figure out why these examples don't compile?

```
package pond.goose;
import pond.shore.Bird;
public class Goose extends Bird {
   public void helpGooseSwim() {
      Goose other = new Goose();
      other.floatInWater();
      System.out.println(other.text);
   }
   public void helpOtherGooseSwim() {
      Bird other = new Goose();
      other.floatInWater();           // DOES NOT COMPILE
      System.out.println(other.text); // DOES NOT COMPILE
   }
}
```

The first method is fine. In fact, it is equivalent to the `Swan` example. `Goose` extends `Bird`. Since we are in the `Goose` subclass and referring to a `Goose` reference, it can access `protected` members. The second method is a problem. Although the object happens to be a `Goose`, it is stored in a `Bird` reference. We are not allowed to refer to members of the `Bird` class since we are not in the same package and the reference type of `other` is not a subclass of `Goose`.

What about this one?

```
package pond.duck;
import pond.goose.Goose;
```

```
public class GooseWatcher {
   public void watch() {
      Goose goose = new Goose();
      goose.floatInWater();     // DOES NOT COMPILE
   }
}
```

This code doesn't compile because we are not in the `goose` object. The `floatInWater()` method is declared in `Bird`. `GooseWatcher` is not in the same package as `Bird`, nor does it extend `Bird`. `Goose` extends `Bird`. That only lets `Goose` refer to `floatInWater()` and not callers of `Goose`.

If this is still puzzling, try it. Type in the code and try to make it compile. Then reread this section. Don't worry—it wasn't obvious to us the first time either!

## Public Access

Protected access was a tough concept. Luckily, the last type of access modifier is easy: `public` means anyone can access the member from anywhere.

---

NOTE

The Java module system redefines "anywhere," and it becomes possible to restrict access to `public` code. When given a code sample, you can assume it isn't in a module unless explicitly stated otherwise.

---

Let's create a class that has `public` members:

```
package pond.duck;
public class DuckTeacher {
   public String name = "helpful";     // public access
```

```
    public void swim() {                // public access
       System.out.println("swim");
    }
 }
```

`DuckTeacher` allows access to any class that wants it. Now we can try it:

```
 package pond.goose;
 import pond.duck.DuckTeacher;
 public class LostDuckling {
    public void swim() {
       DuckTeacher teacher = new DuckTeacher();
       teacher.swim();                                 // allowed
       System.out.println("Thanks" + teacher.name);     // allowed
    }
 }
```

`LostDuckling` is able to refer to `swim()` and `name` on `DuckTeacher` because they are public. The story has a happy ending. `LostDuckling` has learned to swim and can find its parents—all because `DuckTeacher` made members public.

To review access modifiers, make sure you know why everything in Table 7.2 is true. Remember that a member is a method or field.

**TABLE 7.2** **Access modifiers**

| **A method in ________ can access a ________ member** | `private` | **Default (package-private)** | `protected` | `public` |
|---|---|---|---|---|
| the same class | Yes | Yes | Yes | Yes |
| another class in the same package | No | Yes | Yes | Yes |
| in a subclass in a different package | No | No | Yes | Yes |
| an unrelated class in a different package | No | No | No | Yes |

## Applying the *static* Keyword

When the `static` keyword is applied to a variable, method, or class, it applies to the class rather than a specific instance of the class. In this section, you will see that the `static` keyword can also be applied to `import` statements.

### Designing *static* Methods and Fields

Except for the `main()` method, we've been looking at instance methods. `static` methods don't require an instance of the class. They are shared among all users of the class. You can think of a `static` variable as being a member of the single class object that exists independently of any instances of that class.

You have seen one `static` method since Chapter 1. The `main()` method is a `static` method. That means you can call it using the class name:

```
public class Koala {
   public static int count = 0;               // static variable
   public static void main(String[] args) {   // static method
      System.out.println(count);
   }
}
```

Here the JVM basically calls `Koala.main()` to get the program started. You can do this too. We can have a `KoalaTester` that does nothing but call the `main()` method:

```
public class KoalaTester {
   public static void main(String[] args) {
      Koala.main(new String[0]);          // call static method
   }
}
```

Quite a complicated way to print `0`, isn't it? When we run `KoalaTester`, it makes a call to the `main()` method of `Koala`, which prints the value of `count`. The purpose of all these examples is to show that `main()` can be called just like any other `static` method.

In addition to `main()` methods, `static` methods have two main purposes:

- For utility or helper methods that don't require any object state. Since there is no need to access instance variables, having `static` methods eliminates the need for the caller to instantiate an object just to call the method.

- For state that is shared by all instances of a class, like a counter. All instances must share the same state. Methods that merely use that state should be `static` as well.

In the following sections, we will look at some examples covering other static concepts.

### Accessing a *static* Variable or Method

Usually, accessing a `static` member like `count` is easy. You just put the class name before the method or variable and you are done. Here's an example:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Both of these are nice and easy. There is one rule that is trickier. You can use an instance of the object to call a `static` method. The compiler checks for the type of the reference and uses that instead of the object—which is sneaky of Java. This code is perfectly legal:

```
5: Koala k = new Koala();
6: System.out.println(k.count);          // k is a Koala
7: k = null;
8: System.out.println(k.count);          // k is still a Koala
```

Believe it or not, this code outputs 0 twice. Line 6 sees that `k` is a `Koala` and `count` is a `static` variable, so it reads that `static` variable. Line 8 does the same thing. Java doesn't care that `k` happens to be `null`. Since we are looking for a `static`, it doesn't matter.

Remember to look at the reference type for a variable when you see a `static` method or variable. The exam creators will try to trick you into thinking a `NullPointerException` is thrown because the variable happens to be `null`. Don't be fooled!

One more time because this is really important: what does the following output?

```
Koala.count = 4;
Koala koala1 = new Koala();
Koala koala2 = new Koala();
koala1.count = 6;
koala2.count = 5;
System.out.println(Koala.count);
```

We hope you answered 5. There is only one `count` variable since it is `static`. It is set to 4, then 6, and finally winds up as 5. All the `Koala` variables are just distractions.

### Static vs. Instance

There's another way the exam creators will try to trick you regarding `static` and instance members. A `static` member cannot call an instance member without referencing an instance of the class. This shouldn't be a surprise since `static` doesn't require any instances of the class to even exist.

The following is a common mistake for rookie programmers to make:

```
public class Static {
   private String name = "Static class";
```

```
    public static void first() {  }
    public static void second() {  }
    public void third() {  System.out.println(name); }
    public static void main(String args[]) {
       first();
       second();
       third();          // DOES NOT COMPILE
    }
 }
```

The compiler will give you an error about making a `static` reference to a nonstatic method. If we fix this by adding `static` to `third()`, we create a new problem. Can you figure out what it is?

All this does is move the problem. Now, `third()` is referring to nonstatic `name`. Adding `static` to `name` as well would solve the problem. Another solution would have been to call `third` as an instance method—for example, `new Static().third();`.

The exam creators like this topic. A `static` method or instance method can call a `static` method because `static` methods don't require an object to use. Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object. Similar logic applies for the instance and `static` variables.

Suppose we have a `Giraffe` class:

```
public class Giraffe {
   public void eat(Giraffe g) {}
   public void drink() {};
   public static void allGiraffeGoHome(Giraffe g) {}
   public static void allGiraffeComeOut() {}
}
```

Make sure you understand Table 7.3 before continuing.

**TABLE 7.3 Static vs. instance calls**

| Type | Calling | Legal? |
|---|---|---|
| `allGiraffeGoHome()` | `allGiraffeComeOut()` | Yes |
| `allGiraffeGoHome()` | `drink()` | No |
| `allGiraffeGoHome()` | `g.eat()` | Yes |
| `eat()` | `allGiraffeComeOut()` | Yes |
| `eat()` | `drink()` | Yes |
| `eat()` | `g.eat()` | Yes |

Let's try one more example so you have more practice at recognizing this scenario. Do you understand why the following lines fail to compile?

```java
public class Gorilla {
   public static int count;
   public static void addGorilla() { count++; }
   public void babyGorilla() { count++; }
   public void announceBabies() {
      addGorilla();
      babyGorilla();
   }
   public static void announceBabiesToEveryone() {
      addGorilla();
      babyGorilla();     // DOES NOT COMPILE
   }
   public int total;
   public static double average
      = total / count;  // DOES NOT COMPILE
}
```

Lines 3 and 4 are fine because both `static` and instance methods can refer to a `static` variable. Lines 5–8 are fine because an instance method can call a `static` method. Line 11 doesn't compile because a `static` method cannot call an instance method. Similarly, line 15 doesn't compile because a `static` variable is trying to use an instance variable.

A common use for `static` variables is counting the number of instances:

```
public class Counter {
   private static int count;
   public Counter() { count++; }
   public static void main(String[] args) {
      Counter c1 = new Counter();
      Counter c2 = new Counter();
      Counter c3 = new Counter();
      System.out.println(count);          // 3
   }
}
```

Each time the constructor gets called, it increments `count` by 1. This example relies on the fact that `static` (and instance) variables are automatically initialized to the default value for that type, which is 0 for `int`. See Chapter 2 to review the default values.

Also notice that we didn't write `Counter.count`. We could have. It isn't necessary because we are already in that class so the compiler can infer it.

DOES EACH INSTANCE HAVE ITS OWN COPY OF THE CODE?

Each object has a copy of the instance variables. There is only one copy of the code for the instance methods. Each instance of the class can call it as many times as it would like. However, each call of an instance method (or any method) gets space on the stack for method parameters and local variables.

The same thing happens for `static` methods. There is one copy of the code. Parameters and local variables go on the stack.

Just remember that only data gets its "own copy." There is no need to duplicate copies of the code itself.

## *static* Variables

Some `static` variables are meant to change as the program runs. Counters are a common example of this. We want the count to increase over time. Just as with instance variables, you can initialize a `static` variable on the line it is declared:

```
public class Initializers {
   private static int counter = 0;          // initialization
}
```

Other `static` variables are meant to never change during the program. This type of variable is known as a *constant*. It uses the `final` modifier to ensure the variable never changes. Constants use the modifier `static final` and a different naming convention than other variables. They use all uppercase letters with underscores between "words." Here's an example:

```
public class Initializers {
   private static final int NUM_BUCKETS = 45;
```

```
      public static void main(String[] args) {
         NUM_BUCKETS = 5;  // DOES NOT COMPILE
      }
   }
```

The compiler will make sure that you do not accidentally try to update a final variable. This can get interesting. Do you think the following compiles?

```
   private static final ArrayList<String> values = new ArrayList<>();
   public static void main(String[] args) {
      values.add("changed");
   }
```

It actually does compile since `values` is a reference variable. We are allowed to call methods on reference variables. All the compiler can do is check that we don't try to reassign the `final values` to point to a different object.

## Static Initialization

In Chapter 2, we covered instance initializers that looked like unnamed methods—just code inside braces. Static initializers look similar. They add the `static` keyword to specify they should be run when the class is first loaded. Here's an example:

```
   private static final int NUM_SECONDS_PER_MINUTE;
   private static final int NUM_MINUTES_PER_HOUR;
   private static final int NUM_SECONDS_PER_HOUR;
   static {
      NUM_SECONDS_PER_MINUTE = 60;
      NUM_MINUTES_PER_HOUR = 60;
   }
   static {
      NUM_SECONDS_PER_HOUR
      = NUM_SECONDS_PER_MINUTE * NUM_MINUTES_PER_HOUR;
}
```

All `static` initializers run when the class is first used in the order they are defined. The statements in them run and assign any `static` variables as needed. There is something interesting about this example. We just got through saying that `final` variables aren't allowed to be reassigned. The key here is that the `static` initializer is the first assignment. And since it occurs up front, it is okay.

Let's try another example to make sure you understand the distinction:

```
14: private static int one;
15: private static final int two;
16: private static final int three = 3;
17: private static final int four;    // DOES NOT COMPILE
18: static {
19:    one = 1;
20:    two = 2;
21:    three = 3;                     // DOES NOT COMPILE
22:    two = 4;                       // DOES NOT COMPILE
23: }
```

Line 14 declares a `static` variable that is not `final`. It can be assigned as many times as we like. Line 15 declares a `final` variable without initializing it. This means we can initialize it exactly once in a `static` block. Line 22 doesn't compile because this is the second attempt. Line 16 declares a `final` variable and initializes it at the same time. We are not allowed to assign it again, so line 21 doesn't compile. Line 17 declares a `final` variable that never gets initialized. The compiler gives a compiler error because it knows that the `static` blocks are the only place the variable could possibly get initialized. Since the programmer forgot, this is clearly an error.

TRY TO AVOID STATIC AND INSTANCE INITIALIZERS

Using `static` and instance initializers can make your code much harder to read. Everything that could be done in an instance initializer could be done in a constructor instead. Many people find the constructor approach is easier to read.

There is a common case to use a `static` initializer: when you need to initialize a `static` field and the code to do so requires more than one line. This often occurs when you want to initialize a collection like an `ArrayList`. When you do need to use a `static` initializer, put all the `static` initialization in the same block. That way, the order is obvious.

## Static Imports

In Chapter 1, you saw that we could import a specific class or all the classes in a package:

```
import java.util.ArrayList;
import java.util.*;
```

We could use this technique to import two classes:

```
import java.util.List;
import java.util.Arrays;
public class Imports {
   public static void main(String[] args) {
      List<String> list = Arrays.asList("one", "two");
   }
}
```

Imports are convenient because you don't need to specify where each class comes from each time you use it. There is another type of import called a *static import*. Regular imports are for importing classes. Static imports are for importing `static` members of classes. Just like regular imports, you can use a wildcard or import a specific member. The idea is that you shouldn't have to specify where each `static` method or variable comes from each time you use it. An example of when static imports shine is when you are referring to a lot of constants in another class.

---

NOTE

In a large program, static imports can be overused. When importing from too many places, it can be hard to remember where each `static` member comes from.

---

The previous method has one `static` method call: `Arrays.asList`. Rewriting the code to use a static import yields the following:

```
import java.util.List;
import static java.util.Arrays.asList;          // static import
public class StaticImports {
   public static void main(String[] args) {
      List<String> list = asList("one", "two"); // no Arrays.
   }
}
```

In this example, we are specifically importing the `asList` method. This means that any time we refer to `asList` in the class, it will call `Arrays.asList()`.

An interesting case is what would happen if we created an `asList` method in our `StaticImports` class. Java would give it preference over the imported one, and the method we coded would be used.

The exam will try to trick you with misusing static imports. This example shows almost everything you can do wrong. Can you figure out what is wrong with each one?

```
1: import static java.util.Arrays;       // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*;     // DOES NOT COMPILE
4: public class BadStaticImports {
5:    public static void main(String[] args) {
6:       Arrays.asList("one");           // DOES NOT COMPILE
7:    } }
```

Line 1 tries to use a static import to import a class. Remember that static imports are only for importing `static` members. Regular imports are for importing a class. Line 3 tries to see whether you are paying attention to the order of keywords. The syntax is `import static` and not vice versa. Line 6 is sneaky. The `asList` method is imported on line 2. However, the `Arrays` class is not imported anywhere. This makes it okay to write `asList("one")` but not `Arrays.asList("one")`.

There's only one more scenario with static imports. In Chapter 1, you learned that importing two classes with the same name gives a compiler error. This is true of static imports as well. The compiler will complain if you try to explicitly do a static import of two methods with the same name or two `static` variables with the same name. Here's an example:

```
import static statics.A.TYPE;
import static statics.B.TYPE;     // DOES NOT COMPILE
```

Luckily, when this happens, we can just refer to the `static` members via their class name in the code instead of trying to use a static import.

## Passing Data among Methods

Java is a “pass-by-value” language. This means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller. Let’s look at an example:

```
2: public static void main(String[] args) {
3:    int num = 4;
4:    newNumber(num);
5:    System.out.println(num);     // 4
6: }
7: public static void newNumber(int num) {
8:    num = 8;
9: }
```

On line 3, `num` is assigned the value of `4`. On line 4, we call a method. On line 8, the `num` parameter in the method gets set to `8`. Although this parameter has the same name as the variable on line 3, this is a coincidence. The name could be anything. The exam will often use the same name to try to confuse you. The variable on line 3 never changes because no assignments are made to it.

Now that you’ve seen primitives, let’s try an example with a reference type. What do you think is output by the following code?

```
public static void main(String[] args) {
   String name = "Webby";
   speak(name);
   System.out.println(name);
}
public static void speak(String name) {
   name = "Sparky";
}
```

The correct answer is `Webby`. Just as in the primitive example, the variable assignment is only to the method parameter and doesn’t affect the caller.

Notice how we keep talking about variable assignments. This is because we can call methods on the parameters. As an example, here is code that calls a method on the `StringBuilder` passed into the method:

```
public static void main(String[] args) {
   StringBuilder name = new StringBuilder();
   speak(name);
   System.out.println(name); // Webby
}
public static void speak(StringBuilder s) {
   s.append("Webby");
}
```

In this case, the output is `Webby` because the method merely calls a method on the parameter. It doesn't reassign `name` to a different object. In Figure 7.4, you can see how pass-by-value is still used. The variable `s` is a copy of the variable `name`. Both point to the same `StringBuilder`, which means that changes made to the `StringBuilder` are available to both references.

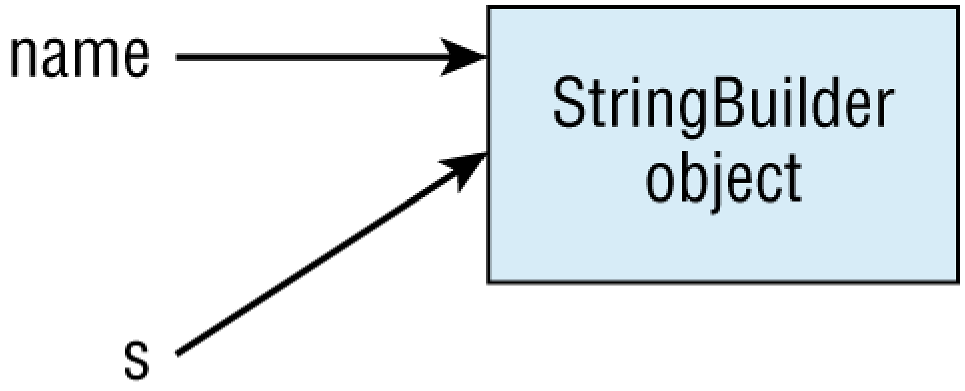

**FIGURE 7.4** **Copying a reference with pass-by-value**

**Real World Scenario**

**PASS-BY-VALUE VS. PASS-BY-REFERENCE**

Different languages handle parameters in different ways. Pass-by-value is used by many languages, including Java. In this example, the swap method does not change the original values. It only changes `a` and `b` within the method.

```
public static void main(String[] args) {
   int original1 = 1;
   int original2 = 2;
   swap(original1, original2);
   System.out.println(original1);   // 1
   System.out.println(original2);   // 2
}
public static void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}
```

The other approach is pass-by-reference. It is used by default in a few languages, such as Perl. We aren't going to show you Perl code here because you are studying for the Java exam and we don't want to confuse you. The following example is in a made-up language that shows pass-by-reference:

```
original1 = 1;
original2 = 2;
swapByReference(original1, original2);
print(original1);   // 2 (not in Java)
print(original2);   // 1 (not in Java)

swapByReference(a, b) {
   temp = a;
   a = b;
```

```
      b = temp;
   }
```

See the difference? In our made-up language, the caller is affected by variable assignments made in the method.

---

To review, Java uses pass-by-value to get data into a method. Assigning a new primitive or reference to a parameter doesn’t change the caller. Calling methods on a reference to an object can affect the caller.

Getting data back from a method is easier. A copy is made of the primitive or reference and returned from the method. Most of the time, this returned value is used. For example, it might be stored in a variable. If the returned value is not used, the result is ignored. Watch for this on the exam. Ignored returned values are tricky.

Let’s try an example. Pay attention to the return types.

```
public class ReturningValues {
   public static void main(String[] args) {
      int number = 1;                             // number=1
      String letters = "abc";                     // letters=abc
      number(number);                             // number=1
      letters = letters(letters);                 // letters=abcd
      System.out.println(number + letters);     // 1abcd
   }
   public static int number(int number) {
      number++;
      return number;
   }
   public static String letters(String letters) {
      letters += "d";
      return letters;
   }
}
```

This is a tricky one because there is a lot to keep track of. When you see such questions on the exam, write down the values of each variable. Lines 3 and 4 are straightforward assignments. Line 5 calls a method. Line 10 increments the method parameter to 2 but leaves the `number` variable in the `main()` method as 1. While line 11 returns the value, the caller ignores it. The method call on line 6 doesn't ignore the result, so `letters` becomes `"abcd"`. Remember that this is happening because of the returned value and not the method parameter.

## Overloading Methods

Now that you are familiar with the rules for declaring methods, it is time to look at creating methods with the same name in the same class. *Method overloading* occurs when methods have the same name but different method signatures, which means they differ by method parameters. (Overloading differs from overriding, which you'll learn about in Chapter 8.)

We've been showing how to call overloaded methods for a while. `System.out.println` and `StringBuilder`'s `append` methods provide many overloaded versions, so you can pass just about anything to them without having to think about it. In both of these examples, the only change was the type of the parameter. Overloading also allows different numbers of parameters.

Everything other than the method name can vary for overloading methods. This means there can be different access modifiers, specifiers (like `static`), return types, and exception lists.

These are all valid overloaded methods:

```
public void fly(int numMiles) {}
public void fly(short numFeet) {}
public boolean fly() { return false; }
```

```
void fly(int numMiles, short numFeet) {}
public void fly(short numFeet, int numMiles) throws Exception {}
```

As you can see, we can overload by changing anything in the parameter list. We can have a different type, more types, or the same types in a different order. Also notice that the return type, access modifier, and exception list are irrelevant to overloading.

Now let's look at an example that is not valid overloading:

```
public void fly(int numMiles) {}
public int fly(int numMiles) {}     // DOES NOT COMPILE
```

This method doesn't compile because it differs from the original only by return type. The parameter lists are the same, so they are duplicate methods as far as Java is concerned.

What about these two? Why does the second not compile?

```
public void fly(int numMiles) {}
public static void fly(int numMiles) {}     // DOES NOT COMPILE
```

Again, the parameter list is the same. You cannot have methods where the only difference is that one is an instance method and one is a `static` method.

Calling overloaded methods is easy. You just write code and Java calls the right one. For example, look at these two methods:

```
public void fly(int numMiles) {
   System.out.println("int");
}
public void fly(short numFeet) {
   System.out.println("short");
}
```

The call `fly((short) 1)` prints `short`. It looks for matching types and calls the appropriate method. Of course, it can be more complicated than this.

Now that you know the basics of overloading, let's look at some more complex scenarios that you may encounter on the exam.

## Varargs

Which method do you think is called if we pass an `int[]`?

```
public void fly(int[] lengths) {}
public void fly(int... lengths) {}     // DOES NOT COMPILE
```

Trick question! Remember that Java treats varargs as if they were an array. This means that the method signature is the same for both methods. Since we are not allowed to overload methods with the same parameter list, this code doesn't compile. Even though the code doesn't look the same, it compiles to the same parameter list.

Now that we've just gotten through explaining that they are the same, it is time to mention how they are not the same. It shouldn't be a surprise that you can call either method by passing an array:

```
fly(new int[] { 1, 2, 3 });
```

However, you can only call the varargs version with stand-alone parameters:

```
fly(1, 2, 3);
```

Obviously, this means they don't compile *exactly* the same. The parameter list is the same, though, and that is what you need to know with respect to overloading for the exam.

## Autoboxing

In Chapter 5, you saw how Java will convert a primitive `int` to an object `Integer` to add it to an `ArrayList` through the wonders of autoboxing. This works for code you write too.

```
public void fly(Integer numMiles) {}
```

This means calling `fly(3)` will call the previous method as expected. However, what happens if you have both a primitive and an integer version?

```
public void fly(int numMiles) {}
public void fly(Integer numMiles) {}
```

Java will match the `int numMiles` version. Java tries to use the most specific parameter list it can find. When the primitive `int` version isn't present, it will autobox. However, when the primitive `int` version is provided, there is no reason for Java to do the extra work of autoboxing.

## Reference Types

Given the rule about Java picking the most specific version of a method that it can, what do you think this code outputs?

```
public class ReferenceTypes {
   public void fly(String s) {
      System.out.print("string");
   }

   public void fly(Object o) {
      System.out.print("object");
   }
   public static void main(String[] args) {
      ReferenceTypes r = new ReferenceTypes();
      r.fly("test");
      System.out.print("-");
      r.fly(56);
   }
}
```

The answer is `string-object`. The first call is a `String` and finds a direct match. There's no reason to use the `Object` version when there is a nice `String` parameter list just waiting to be called. The second call looks for an `int` parameter list. When it doesn't find one, it autoboxes to `Integer`. Since it still doesn't find a match, it goes to the `Object` one.

Let's try another one. What does this print?

```
public static void print(Iterable i) {
   System.out.print("I");
}
public static void print(CharSequence c) {
   System.out.print("C");
}
public static void print(Object o) {
   System.out.print("O");
}
public static void main(String[] args){
   print("abc");
   print(new ArrayList<>());
   print(LocalDate.of(2019, Month.JULY, 4));
}
```

The answer is `CIO`. The code is due for a promotion! The first call to `print()` passes a `String`. As you learned in Chapter 5, `String` and `StringBuilder` implement the `CharSequence` interface.

The second call to `print()` passes an `ArrayList`. Remember that you get to assume unknown APIs do what they sound like. In this case, `Iterable` is an interface for classes you can iterate over.

The final call to `print()` passes a `LocalDate`. This is another class you might not know, but that's okay. It clearly isn't a sequence of characters or something to loop through. That means the `Object` method signature is used.

## Primitives

Primitives work in a way that's similar to reference variables. Java tries to find the most specific matching overloaded method. What do you think happens here?

```
public class Plane {
   public void fly(int i) {
      System.out.print("int");
   }
   public void fly(long l) {
      System.out.print("long");
   }
   public static void main(String[] args) {
      Plane p = new Plane();
      p.fly(123);
      System.out.print("-");
      p.fly(123L);
   }
}
```

The answer is `int-long`. The first call passes an `int` and sees an exact match. The second call passes a `long` and also sees an exact match. If we comment out the overloaded method with the `int` parameter list, the output becomes `long-long`. Java has no problem calling a larger primitive. However, it will not do so unless a better match is not found.

Note that Java can only accept wider types. An `int` can be passed to a method taking a `long` parameter. Java will not automatically convert to a narrower type. If you want to pass a `long` to a method taking an `int` parameter, you have to add a cast to explicitly say narrowing is okay.

## Generics

You might be surprised to learn that these are not valid overloads:

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {}    // DOES NOT COMPILE
```

Java has a concept called *type erasure* where generics are used only at compile time. That means the compiled code looks like this:

```
public void walk(List strings) {}
public void walk(List integers) {}    // DOES NOT COMPILE
```

We clearly can't have two methods with the same method signature, so this doesn't compile. Remember that method overloads must differ in at least one of the method parameters.

## Arrays

Unlike the previous example, this code is just fine:

```
public static void walk(int[] ints) {}
public static void walk(Integer[] integers) {}
```

Arrays have been around since the beginning of Java. They specify their actual types and don't participate in type erasure.

## Putting It All Together

So far, all the rules for when an overloaded method is called should be logical. Java calls the most specific method it can. When some of the types interact, the Java rules focus on backward compatibility. A long time ago, autoboxing and varargs didn't exist. Since old code still needs to work, this means autoboxing and varargs come last when Java looks at overloaded methods. Ready for the official order? Table 7.4 lays it out for you.

**TABLE 7.4 The order that Java uses to choose the right overloaded method**

| Rule | Example of what will be chosen for `glide(1,2)` |
|---|---|
| Exact match by type | `String glide(int i, int j)` |
| Larger primitive type | `String glide(long i, long j)` |
| Autoboxed type | `String glide(Integer i, Integer j)` |
| Varargs | `String glide(int... nums)` |

Let's give this a practice run using the rules in Table 7.4. What do you think this outputs?

```
public class Glider2 {
   public static String glide(String s) {
      return "1";
   }
   public static String glide(String... s) {
      return "2";
   }
   public static String glide(Object o) {
      return "3";
   }
   public static String glide(String s, String t) {
      return "4";
   }
   public static void main(String[] args) {
      System.out.print(glide("a"));
      System.out.print(glide("a", "b"));
      System.out.print(glide("a", "b", "c"));
   }
}
```

It prints out `142`. The first call matches the signature taking a single `String` because that is the most specific match. The second call matches the signature, taking two `String` parameters since that is an exact match. It isn't until the third call that the varargs version is used since there are no better matches.

As accommodating as Java is with trying to find a match, it will do only one conversion:

```
public class TooManyConversions {
   public static void play(Long l) {}
   public static void play(Long... l) {}
   public static void main(String[] args) {
      play(4);      // DOES NOT COMPILE
      play(4L);     // calls the Long version
   }
}
```

Here we have a problem. Java is happy to convert the `int` 4 to a `long` 4 or an `Integer` 4. It cannot handle converting to a `long` and then to a `Long`. If we had `public static void play(Object o) {}`, it would match because only one conversion would be necessary: from `int` to `Integer`. Remember, if a variable is not a primitive, it is an `Object`, as you'll see in Chapter 8.

## Encapsulating Data

In Chapter 2, you saw an example of a class with a field that wasn't private:

```
public class Swan {
   int numberEggs;     // instance variable
}
```

Why do we care? Since there is default (package-private) access, that means any class in the package can set `numberEggs`. We no longer have control of what gets set in your own class. A caller could even write this:

```
mother.numberEggs = -1;
```

This is clearly no good. We do not want the mother `Swan` to have a negative number of eggs!

Encapsulation to the rescue. *Encapsulation* means only methods in the class with the variables can refer to the instance variables. Callers are required to use these methods. Let's take a look at the newly encapsulated `Swan` class:

```
1: public class Swan {
2:    private int numberEggs;                     // private
3:    public int getNumberEggs() {                // getter
4:       return numberEggs;
5:    }
6:    public void setNumberEggs(int newNumber) { // setter
7:       if (newNumber >= 0)                     // guard condition
8:          numberEggs = newNumber;
9:    } }
```

Note that `numberEggs` is now private on line 2. This means only code within the class can read or write the value of `numberEggs`. Since we wrote the class, we know better than to set a negative number of eggs. We added a method on lines 3–5 to read the value, which is called an *accessor method* or a getter. We also added a method on lines 6–9 to update the value, which is called a *mutator method* or a setter. The setter has an `if` statement in this example to prevent setting the instance variable to an invalid value. This guard condition protects the instance variable.

For encapsulation, remember that data (an instance variable) is private and getters/setters are public. Java defines a naming convention for getters and setters listed in Table 7.5.

**TABLE 7.5 Naming conventions for getters and setters**

| Rule | Example |
|---|---|
| Getter methods most frequently begin with `is` if the property is a `boolean`. | `public boolean isHappy() { return happy; }` |
| Getter methods begin with `get` if the property is not a `boolean`. | `public int getNumberEggs() { return numberEggs; }` |
| Setter methods begin with `set`. | `public void setHappy(boolean _happy) { happy = _happy; }` |

In the last example in Table 7.5, you probably noticed that you can name the method parameter to anything you want. Only the method name and property name have naming conventions here.

It's time for some practice. See whether you can figure out which lines follow these naming conventions:

```
12: private boolean playing;
13: private String name;
14: public boolean isPlaying() { return playing; }
15: public String name() { return name; }
16: public void updateName(String n) { name = n; }
17: public void setName(String n) { name = n; }
```

Lines 12 and 13 are good. They are private instance variables. Line 14 is correct. Since `playing` is a `boolean`, line 14 is a correct getter. Line 15 doesn't follow the naming conventions because it should be called `getName()`. Line 16 does not follow the naming convention for a setter, but line 17 does.

For data to be encapsulated, you don't have to provide getters and setters. As long as the instance variables are `private`, you are good. For example, this is a well-encapsulated class:

```
public class Swan {
   private int numEggs;
   public void layEgg() {
      numEggs++;
   }
   public void printEggCount() {
      System.out.println(numEggs);
   }
}
```

To review, you can tell it is a well-encapsulated class because the `numEggs` instance variable is `private`. Only methods can retrieve and update the value.

## Summary

As you learned in this chapter, Java methods start with an access modifier of `public`, `private`, `protected`, or blank (default access). This is followed by an optional specifier such as `static`, `final`, or `abstract`. Next comes the return type, which is `void` or a Java type. The method name follows, using standard Java identifier rules. Zero or more parameters go in parentheses as the parameter list. Next come any optional exception types. Finally, zero or more statements go in braces to make up the method body.

Using the `private` keyword means the code is only available from within the same class. Default (package-private) access means the code is available only from within the same package. Using the `protected` keyword means the code is available from the same package or subclasses. Using the `public` keyword means the code is available from anywhere. Both `static` methods and `static` variables are shared by all instances of the class. When referenced from outside the class, they are called using the classname—for example, `StaticClass.method()`. Instance members are allowed to call `static` members, but `static` members are not allowed to call instance members. Static imports are used to import `static` members.

Java uses pass-by-value, which means that calls to methods create a copy of the parameters. Assigning new values to those parameters in the method doesn't affect the caller's variables. Calling methods on objects that are method parameters changes the state of those objects and is reflected in the caller.

Overloaded methods are methods with the same name but a different parameter list. Java calls the most specific method it can find. Exact matches are preferred, followed by wider primitives. After that comes autoboxing and finally varargs.

Encapsulation refers to preventing callers from changing the instance variables directly. This is done by making instance variables `private` and getters/setters `public`.

## Exam Essentials

**Be able to identify correct and incorrect method declarations.** A sample method declaration is `public static void method(String... args) throws Exception {}`.

**Identify when a method or field is accessible.** Recognize when a method or field is accessed when the access modifier ( `private` , `protected` , `public` , or default access) does not allow it.

**Recognize valid and invalid uses of static imports.** Static imports import `static` members. They are written as `import static` , not *static import*. Make sure they are importing `static` methods or variables rather than class names.

**State the output of code involving methods.** Identify when to call `static` rather than instance methods based on whether the class name or object comes before the method. Recognize that instance methods can call `static` methods and that `static` methods need an instance of the object in order to call an instance method.

**Recognize the correct overloaded method.** Exact matches are used first, followed by wider primitives, followed by autoboxing, followed by varargs. Assigning new values to method parameters does not change the caller, but calling methods on them does.

**Identify properly encapsulated classes.** Instance variables in encapsulated classes are `private` . All code that retrieves the value or updates it uses methods. These methods are allowed to be `public` .

## Review Questions

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

1. Which of the following can fill in the blank in this code to make it compile? (Choose all that apply.)

   ```
   public class Ant {
       _____ void method() {}
   }
   ```

   1. `default`
   2. `final`
   3. `private`
   4. `Public`
   5. `String`
   6. `zzz:`
2. Which of the following methods compile? (Choose all that apply.)
   1. `final static void method4() {}`
   2. `public final int void method() {}`
   3. `private void int method() {}`
   4. `static final void method3() {}`
   5. `void final method() {}`
   6. `void public method() {}`
3. Which of the following methods compile? (Choose all that apply.)
   1. `public void methodA() { return;}`
   2. `public int methodB() { return null;}`
   3. `public void methodC() {}`
   4. `public int methodD() { return 9;}`
   5. `public int methodE() { return 9.0;}`
   6. `public int methodF() { return;}`
4. Which of the following methods compile? (Choose all that apply.)
   1. `public void moreA(int... nums) {}`
   2. `public void moreB(String values, int... nums) {}`
   3. `public void moreC(int... nums, String values) {}`
   4. `public void moreD(String... values, int... nums) {}`
   5. `public void moreE(String[] values, ...int nums) {}`
   6. `public void moreG(String[] values, int[] nums) {}`

5. Given the following method, which of the method calls return `2`? (Choose all that apply.)

```
public int howMany(boolean b, boolean... b2) {
 return b2.length;
}
```

   1. `howMany();`
   2. `howMany(true);`
   3. `howMany(true, true);`
   4. `howMany(true, true, true);`
   5. `howMany(true, {true, true});`
   6. `howMany(true, new boolean[2]);`

6. Which of the following statements is true?
   1. Package-private access is more lenient than protected access.
   2. A `public` class that has private fields and package-private methods is not visible to classes outside the package.
   3. You can use access modifiers so only some of the classes in a package see a particular package-private class.
   4. You can use access modifiers to allow access to all methods and not any instance variables.
   5. You can use access modifiers to restrict access to all classes that begin with the word `Test`.

7. Given the following `my.school.` Classroom and `my.city.School` class definitions, which line numbers in `main()` generate a compiler error? (Choose all that apply.)

```
1: package my.school;
2: public class Classroom {
3:    private int roomNumber;
4:    protected static String teacherName;
5:    static int globalKey = 54321;
6:    public static int floor = 3;
7:    Classroom(int r, String t) {
8:       roomNumber = r;
9:       teacherName = t; } }

1: package my.city;
2: import my.school.*;
3: public class School {
4:    public static void main(String[] args) {
5:       System.out.println(Classroom.globalKey);
6:       Classroom room = new Classroom(101, "Mrs. Anderson");
7:       System.out.println(room.roomNumber);
8:       System.out.println(Classroom.floor);
9:       System.out.println(Classroom.teacherName); } }
```

1. None, the code compiles fine.
2. Line 5
3. Line 6
4. Line 7
5. Line 8
6. Line 9

8. Which of the following are true about encapsulation? (Choose all that apply.)
   1. It allows getters.
   2. It allows setters.
   3. It requires specific naming conventions.
   4. It uses package-private instance variables.
   5. It uses private instance variables.
9. Which pairs of methods are valid overloaded pairs? (Choose all that apply.)
   1. 
      ```
      public void hiss(Iterable i) {}
      ```
      and
      ```
      public int hiss(Iterable i) { return 0; }
      ```
   2. 
      ```
      public void baa(CharSequence c) {}
      ```
      and
      ```
      public void baa(String s) {}
      ```
   3.

```
public var meow(List<String> l) {}
```

and

```
public var meow(String s) {}
```

4.
```
public void moo(Object o) {}
```
and
```
public void moo(String s) {}
```

5.
```
public void roar(List<Boolean> b) {}
```
and
```
public void roar(List<Character> c) {}
```

6.
```
public void woof(boolean[] b1) {}
```
and
```
public void woof(Boolean[] b) {}
```

10. What is the output of the following code?

```
package rope;
public class Rope {
   public static int LENGTH = 5;
   static {
      LENGTH = 10;
   }
   public static void swing() {
      System.out.print("swing ");
   } }
```

```
import rope.*;
import static rope.Rope.*;
public class Chimp {
   public static void main(String[] args) {
      Rope.swing();
      new Rope().swing();
      System.out.println(LENGTH);
   } }
```

1. `swing swing 5`

2. `swing swing 10`
3. Compiler error on line 2 of `Chimp`
4. Compiler error on line 5 of `Chimp`
5. Compiler error on line 6 of `Chimp`
6. Compiler error on line 7 of `Chimp`

11. Which statements are true of the following code? (Choose all that apply.)

```
public class Rope {
   public static void swing() {
      System.out.print("swing");
   }
   public void climb() {
      System.out.println("climb");
   }
   public static void play() {
      swing();
      climb();
   }
   public static void main(String[] args) {
      Rope rope = new Rope();
      rope.play();
      Rope rope2 = null;
      System.out.print("-");
      rope2.play();
   } }
```

1. The code compiles as is.
2. There is exactly one compiler error in the code.
3. There are exactly two compiler errors in the code.
4. If the line(s) with compiler errors are removed, the output is `swing-climb`.
5. If the line(s) with compiler errors are removed, the output is `swing-swing`.
6. If the line(s) with compile errors are removed, the code throws a `NullPointerException`.

12. What is the output of the following code?

```java
import rope.*;
import static rope.Rope.*;
public class RopeSwing {
   private static Rope rope1 = new Rope();
   private static Rope rope2 = new Rope();
   {
      System.out.println(rope1.length);
   }
   public static void main(String[] args) {
      rope1.length = 2;
      rope2.length = 8;
      System.out.println(rope1.length);
   }
}

package rope;
public class Rope {
   public static int length = 0;
}
```

1. `02`
2. `08`
3. `2`
4. `8`
5. The code does not compile.
6. An exception is thrown.

13. How many lines in the following code have compiler errors?

```java
1:  public class RopeSwing {
2:     private static final String leftRope;
3:     private static final String rightRope;
4:     private static final String bench;
5:     private static final String name = "name";
6:     static {
7:        leftRope = "left";
8:        rightRope = "right";
9:     }
10:    static {
```

```
11:       name = "name";
12:       rightRope = "right";
13:    }
14:    public static void main(String[] args) {
15:       bench = "bench";
16:    }
17: }
```

1. 0
2. 1
3. 2
4. 3
5. 4
6. 5

14. Which of the following can replace line 2 to make this code compile? (Choose all that apply.)

```
1: import java.util.*;
2: // INSERT CODE HERE
3: public class Imports {
4:    public void method(ArrayList<String> list) {
5:       sort(list);
6:    }
7: }
```

1. `import static java.util.Collections;`
2. `import static java.util.Collections.*;`
3. `import static java.util.Collections.sort(ArrayList<String>);`
4. `static import java.util.Collections;`
5. `static import java.util.Collections.*;`
6. `static import java.util.Collections.sort(ArrayList<String>);`

15. What is the result of the following statements?

```java
public class Test {
   public void print(byte x) {
      System.out.print("byte-");
   }
   public void print(int x) {
      System.out.print("int-");
   }
   public void print(float x) {
      System.out.print("float-");
   }
   public void print(Object x) {
      System.out.print("Object-");
   }
   public static void main(String[] args) {
      Test t = new Test();
      short s = 123;
      t.print(s);
      t.print(true);
      t.print(6.789);
   }
}
```

1. `byte-float-Object-`
2. `int-float-Object-`
3. `byte-Object-float-`
4. `int-Object-float-`
5. `int-Object-Object-`
6. `byte-Object-Object-`

16. What is the result of the following program?

```java
public class Squares {
   public static long square(int x) {
      var y = x * (long) x;
      x = -1;
      return y;
   }
   public static void main(String[] args) {
      var value = 9;
9:        var result = square(value);
10:       System.out.println(value);
11:    } }
```

1. `-1`
2. `9`
3. `81`
4. Compiler error on line 9
5. Compiler error on a different line

17. Which of the following are output by the following code? (Choose all that apply.)

```
public class StringBuilders {
   public static StringBuilder work(StringBuilder a,
      StringBuilder b) {
      a = new StringBuilder("a");
      b.append("b");
      return a;
   }
   public static void main(String[] args) {
      var s1 = new StringBuilder("s1");
      var s2 = new StringBuilder("s2");
      var s3 = work(s1, s2);
      System.out.println("s1 = " + s1);
      System.out.println("s2 = " + s2);
      System.out.println("s3 = " + s3);
   }
}
```

1. `s1 = a`
2. `s1 = s1`
3. `s2 = s2`
4. `s2 = s2b`
5. `s3 = a`
6. The code does not compile.

18. Which of the following will compile when independently inserted in the following code? (Choose all that apply.)

```
1:  public class Order3 {
2:     final String value1 = "red";
3:     static String value2 = "blue";
4:     String value3 = "yellow";
5:     {
6:        // CODE SNIPPET 1
7:     }
8:     static {
9:        // CODE SNIPPET 2
10:    } }
```

1. Insert at line 6: `value1 = "green";`
2. Insert at line 6: `value2 = "purple";`
3. Insert at line 6: `value3 = "orange";`
4. Insert at line 9: `value1 = "magenta";`
5. Insert at line 9: `value2 = "cyan";`
6. Insert at line 9: `value3 = "turquoise";`

19. Which of the following are true about the following code? (Choose all that apply.)

```
public class Run {
   static void execute() {
      System.out.print("1-");
   }
   static void execute(int num) {
      System.out.print("2-");
   }
   static void execute(Integer num) {
      System.out.print("3-");
   }
   static void execute(Object num) {
      System.out.print("4-");
   }
   static void execute(int... nums) {
```

```
            System.out.print("5-");
         }
         public static void main(String[] args) {
            Run.execute(100);
            Run.execute(100L);
         }
      }
```

1. The code prints out `2-4-`.
2. The code prints out `3-4-`.
3. The code prints out `4-2-`.
4. The code prints out `4-4-`.
5. The code prints `3-4-` if you remove the method `static void execute(int num)`.
6. The code prints `4-4-` if you remove the method `static void execute(int num)`.

20. Which pairs of methods are valid overloaded pairs? (Choose all that apply.)
    1. 
       ```
       public void hiss(Set<String> s) {}
       ```
       and
       ```
       public void hiss(List<String> l) {}
       ```
    2. 
       ```
       public void baa(var c) {}
       ```
       and
       ```
       public void baa(String s) {}
       ```
    3. 
       ```
       public void meow(char ch) {}
       ```
       and
       ```
       public void meow(String s) {}
       ```
    4. 
       ```
       public void moo(char ch) {}
       ```
       and
       ```
       public void moo(char ch) {}
       ```
    5.

```
public void roar(long... longs){}
```
and
```
public void roar(long long) {}
```
6.
```
public void woof(char... chars) {}
```
and
```
public void woof(Character c) {}
```
21. Which can fill in the blank to create a properly encapsulated class? (Choose all that apply.)

```
public class Rabbits {
   ______ int numRabbits = 0;
   ______ void multiply() {
      numRabbits *= 6;
   }
   ______ int getNumberOfRabbits() {
      return numRabbits;
   }
}
```

1. `private` , `public` , and `public`
2. `private` , `protected` , and `private`
3. `private` , `private` , and `protected`
4. `public` , `public` , and `public`
5. None of the above since `multiply()` does not begin with set
6. None of the above for a reason other than the `multiply()` method