

Chapter 5

Methods

OCJP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ Using Object-Oriented Concepts in Java

- Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers.
- Implement overloaded methods, including var-arg methods.

In previous chapters, you learned how to write snippets of code without much thought about the methods that contained the code. In this chapter, you explore methods in depth including modifiers, arguments, varargs, overloading, and autoboxing. Many of these fundamentals, such as `access` and `static` modifiers, are applicable to classes and other types throughout the rest of the book. If you're having difficulty, you might want to read this chapter twice!

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 5.1](#).

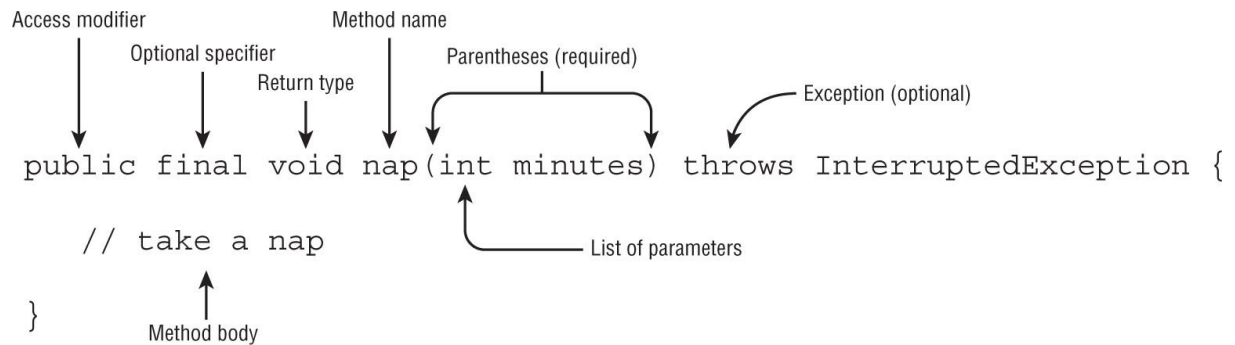


FIGURE 5.1 Method declaration

This is called a *method declaration*, which specifies all the information needed to call the method. There are a lot of parts, and we cover each one in more detail. Two of the parts—the method name and parameter list—are called the *method signature*. The method signature provides instructions for *how* callers can reference this method. The method signature does not include the return type and access modifiers, which control *where* the method can be referenced.

[Table 5.1](#) 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 5.1 Parts of a method declaration in [Figure 5.1](#)

Element	Value in <code>nap()</code> example	Required?
Access modifier	<code>public</code>	No
Optional specifier	<code>final</code>	No
Return type	<code>void</code>	Yes
Method name	<code>nap</code>	Yes
Parameter list	<code>(int minutes)</code>	Yes, but can be empty parentheses
Method signature	<code>nap(int minutes)</code>	Yes
Exception list	<code>throws InterruptedException</code>	No
Method body	<code>{ // take a nap }</code>	Yes, except for abstract methods

To call this method, just use the method signature and provide an `int` value in parentheses:

```
nap(10);
```

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

Access Modifiers

An access modifier determines what classes a method can be accessed from. Think of it like a security guard. Some classes are good friends, some are distant relatives, and some are complete strangers. Access modifiers help to enforce when these components are allowed to talk to each other. Java offers four choices of access:

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

Package Access With package access, the method can be called only from a class in the same package. This one is tricky because there is no keyword. You simply omit the access modifier. Package access is sometimes referred to as package-private or default access (even within this book!).

protected The `protected` modifier means the method can be called only from a class in the same package or a subclass.

public The `public` modifier means the method can be called from anywhere.



For simplicity, we're primarily concerned with access modifiers applied to methods and fields in this chapter. Rules for access modifiers are also applicable to classes and other types you learn about in [Chapter 7](#), “Beyond Classes,” such as interfaces, enums, and records.

We 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 chapter. 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 class ParkTrip {  
    public void skip1() {}  
    default void skip2() {} // DOES NOT COMPILE  
    void public skip3() {} // DOES NOT COMPILE  
    void skip4() {}  
}
```

The `skip1()` method is a valid declaration with `public` access. The `skip4()` method is a valid declaration with `package` access. The `skip2()` method doesn't compile because `default` is not a valid access modifier. There is a `default` keyword, which is used in `switch` statements and interfaces, but `default` is never used as an access modifier. The `skip3()` method doesn't compile because the access modifier is specified after the return type.

Optional Specifiers

There are a number of optional specifiers for methods, shown in [Table 5.2](#). 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.

As you can see in [Table 5.2](#), four of the method modifiers are covered in later chapters, and the last two aren't even in scope for the exam (and are seldom used in real life). In this chapter, we focus on introducing you to these modifiers. Using them often requires a lot more rules.

TABLE 5.2 Optional specifiers for methods

Modifier	Description	Chapter covered
<code>static</code>	Indicates the method is a member of the shared class object	Chapter 5
<code>abstract</code>	Used in an abstract class or interface when the method body is excluded	Chapter 6
<code>final</code>	Specifies that the method may not be overridden in a subclass	Chapter 6
<code>default</code>	Used in an interface to provide a default implementation of a method for classes that implement the interface	Chapter 7
<code>synchronized</code>	Used with multithreaded code	Chapter 13
<code>native</code>	Used when interacting with code written in another language, such as C++	Out of scope
<code>strictfp</code>	Used for making floating-point calculations portable	Out of scope

While access modifiers and optional specifiers can appear in any order, *they must all appear before the return type*. In practice, it is common to list the access modifier first. As you'll also learn in upcoming chapters, some specifiers are not compatible with one another. For example, you can't declare a method (or class) both `final` and `abstract`.



Remember, access modifiers and optional specifiers can be listed in any order, but once the return type is specified, the rest of the parts of the method are written in a specific order: name, parameter list, exception list, body.

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

```
public class Exercise {  
    public void bike1() {}  
    public final void bike2() {}  
    public static final void bike3() {}  
    public final static void bike4() {}  
    public modifier void bike5() {}           // DOES NOT COMPILE  
    public void final bike6() {}             // DOES NOT COMPILE  
    final public void bike7() {}  
}
```

The `bike1()` method is a valid declaration with no optional specifier. This is OK—it is optional, after all. The `bike2()` method is a valid declaration, with `final` as the optional specifier. The `bike3()` and `bike4()` methods are valid declarations with both `final` and `static` as optional specifiers. The order of these two keywords doesn't matter. The `bike5()` method doesn't compile because `modifier` is not a valid optional specifier. The `bike6()` method doesn't compile because the optional specifier is after the return type.

The `bike7()` 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. It must appear after any access modifiers or optional specifiers and before the method name.

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.”



Remember that a method must have a return type. If no value is returned, the `void` keyword must be used. 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. Think of a `return` statement in a `void` method as the method saying, “I’m done!” and quitting early, such as the following:

```
public void swim(int distance) {
    if(distance <= 0) {
        // Exit early, nothing to do!
        return;
    }
    System.out.print("Fish is swimming " + distance + "
meters");
}
```

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

```
public class Hike {
    public void hike1() {}
    public void hike2() { return; }
    public String hike3() { return ""; }
    public String hike4() {} // DOES NOT COMPILE
    public hike5() {} // DOES NOT COMPILE
    public String hike6() { } // DOES NOT COMPILE
    String hike7(int a) { // DOES NOT COMPILE
        if (1 < 2) return "orange";
    }
}
```


Since the return type of the `hike1()` method is `void`, the `return` statement is optional. The `hike2()` method shows the optional `return` statement that correctly doesn't return anything. The `hike3()` method is a valid declaration with a `String` return type and a `return` statement that returns a `String`. The `hike4()` method doesn't compile because the `return` statement is missing. The `hike5()` method doesn't compile because the return type is missing. The `hike6()` method doesn't compile because it attempts to use two return types. You get only one return type.

The `hike7()` method is a little tricky. There is a `return` statement, but it doesn't always get run. Even though `1` is always less than `2`, the compiler won't fully evaluate the `if` statement and requires a `return` statement if this condition is `false`. What about this modified version?

```
String hike8(int a) {  
    if (1 < 2) return "orange";  
    return "apple";  
} // COMPILER WARNING
```

The code compiles, although the compiler will produce a warning about *unreachable code* (or *dead code*). This means the compiler was smart enough to realize you wrote code that cannot possibly be reached.

When returning a value, it needs to be assignable to the return type. Can you spot what's wrong with two of these examples?

```
public class Measurement {  
    int getHeight1() {  
        int temp = 9;  
        return temp;  
    }  
    int getHeight2() {  
        int temp = 9L; // DOES NOT COMPILE  
        return temp;  
    }  
    int getHeight3() {  
        long temp = 9L;  
        return temp; // DOES NOT COMPILE  
    }  
}
```

The `getHeight2()` method doesn't compile because you can't assign a `long` to an `int`. The `getHeight3()` method doesn't compile because you can't

return a `long` value as an `int`. If this wasn't clear to you, you should go back to [Chapter 2](#), "Operators," and reread the sections about numeric types and casting.

Method Name

Method names follow the same rules we practiced with variable names in [Chapter 1](#), "Building Blocks." To review, an identifier may only contain letters, numbers, currency symbols, 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 they are not required to. Since this is a review of [Chapter 1](#), we can jump right into practicing with some examples:

```
public class BeachTrip {
    public void jog1() {}
    public void 2jog() {} // DOES NOT COMPILE
    public jog3 void() {} // DOES NOT COMPILE
    public void Jog_$() {}
    public _() {} // DOES NOT COMPILE
    public void() {} // DOES NOT COMPILE
}
```

The `jog1()` method is a valid declaration with a traditional name. The `2jog()` method doesn't compile because identifiers are not allowed to begin with numbers. The `jog3()` method doesn't compile because the method name is before the return type. The `Jog_$()` 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:

```
public class Sleep {
    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 declarations with "regular" parameters:

```
public class PhysicalEducation {  
    public void run1() {}  
    public void run2 {} // DOES NOT COMPILE  
    public void run3(int a) {}  
    public void run4(int a; int b) {} // DOES NOT COMPILE  
    public void run5(int a, int b) {}  
}
```

The `run1()` method is a valid declaration without any parameters. The `run2()` method doesn't compile because it is missing the parentheses around the parameter list. The `run3()` method is a valid declaration with one parameter. The `run4()` 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 `run5()` method is a valid declaration with two parameters.

Method Signature

A method signature, composed of the method name and parameter list, is what Java uses to uniquely determine exactly which method you are attempting to call. Once it determines *which* method you are trying to call, it then determines *if* the call is allowed. For example, attempting to access a `private` method outside the class or assigning the return value of a `void` method to an `int` variable results in compiler errors. Neither of these compiler errors is related to the method signature, though.

It's important to note that the names of the parameters in the method signature are not used as part of a method signature. The parameter list is about the *types* of parameters and their *order*. For example, the following two methods have the exact same signature:

```
// DOES NOT COMPILE  
public class Trip {  
    public void visitZoo(String name, int waitTime) {}  
}
```

```
public void visitZoo(String attraction, int rainFall) {}  
}
```

Despite having different parameter names, these two methods have the same signature and cannot be declared within the same class. Changing the order of parameter types does allow the method to compile, though:

```
public class Trip {  
    public void visitZoo(String name, int waitTime) {}  
    public void visitZoo(int rainFall, String attraction) {}  
}
```

We cover these rules in more detail when we get to method overloading later in this chapter.

Exception List

In Java, code can indicate that something went wrong by throwing an exception. We cover this in [Chapter 11](#), “Exceptions and Localization.” 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 class ZooMonorail {  
    public void zeroExceptions() {}  
  
    public void oneException() throws IllegalArgumentException  
{}  
  
    public void twoExceptions() throws  
        IllegalArgumentException, InterruptedException {}  
}
```

While the list of exceptions is optional, it may be required by the compiler, depending on what appears inside the method body. You learn more about this, as well as how methods calling them may be required to handle these exception declarations, in [Chapter 11](#).

Method Body

The final part of a method declaration is the method body. 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 the following compile or don't:

```
public class Bird {  
    public void fly1() {}  
    public void fly2()          // DOES NOT COMPILE  
    public void fly3(int a) { int name = 5; }  
}
```

The `fly1()` method is a valid declaration with an empty method body. The `fly2()` method doesn't compile because it is missing the braces around the empty method body. Methods are required to have a body unless they are declared `abstract`. We cover `abstract` methods in [Chapter 6](#), "Class Design." The `fly3()` method is a valid declaration with one statement in the method body.

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

Declaring Local and Instance Variables

Now that we have methods, we need to talk a little bit about the variables that they can create or use. As you might recall from [Chapter 1](#), local variables are those defined within a method or block, while instance variables are those that are defined as a member of a class. Let's take a look at an example:

```
public class Lion {  
    int hunger = 4;  
  
    public int feedZooAnimals() {  
        int snack = 10; // Local variable  
        if (snack > 4) {  
            long dinnerTime = snack++;  
            hunger--;  
        }  
        return snack;  
    }  
}
```

In the `Lion` class, `snack` and `dinnertime` are local variables accessible only within their respective code blocks, while `hunger` is an instance variable

and created in every object of the `Lion` class.

The object or value returned by a method may be available outside the method, but the variable reference `snack` is gone. Keep this in mind while reading this chapter: all local variable references are destroyed after the block is executed, but the objects they point to may still be accessible.

Local Variable Modifiers

There's only one modifier that can be applied to a local variable: `final`. Easy to remember, right? When writing methods, developers may want to create a variable that does not change during the course of the method. In this code sample, trying to change the value or object these variables reference results in a compiler error:

```
public void zooAnimalCheckup(boolean isWeekend) {
    final int rest;
    if(isWeekend) rest = 5; else rest = 20;
    System.out.print(rest);

    final var giraffe = new Animal();
    final int[] friends = new int[5];

    rest = 10;                // DOES NOT COMPILE
    giraffe = new Animal();   // DOES NOT COMPILE
    friends = null;           // DOES NOT COMPILE
}
```

As shown with the `rest` 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. We can even use `var` and `final` together. Contrast this with the following example:

```
public void zooAnimalCheckup(boolean isWeekend) {
    final int rest;
    if(isWeekend) rest = 5;
    System.out.print(rest);    // DOES NOT COMPILE
}
```

The `rest` variable might not have been assigned a value, such as if `isWeekend` is `false`. Since the compiler does not allow the use of local variables that may not have been assigned a value, the code does not compile.

Does using the `final` modifier mean we can't modify the data? Nope. The `final` attribute refers only to the variable reference; the contents can be freely modified (assuming the object isn't immutable).

```
public void zooAnimalCheckup() {  
    final int rest = 5;  
    final Animal giraffe = new Animal();  
    final int[] friends = new int[5];  
  
    giraffe.setName("George");  
    friends[2] = 2;  
}
```

The `rest` variable is a primitive, so it's just a value that can't be modified. On the other hand, the contents of the `giraffe` and `friends` variables can be freely modified, provided the variables aren't reassigned.



While it might not seem obvious, marking a local variable `final` is often a good practice. For example, you may have a complex method in which a variable is referenced dozens of times. It would be really bad if someone came in and reassigned the variable in the middle of the method. Using the `final` attribute is like sending a message to other developers to leave the variable alone!

Effectively Final Variables

An *effectively final* local variable is one that is not modified after it is assigned. This means that the value of a variable doesn't change after it is set, regardless of whether it is explicitly marked as `final`. If you aren't sure whether a local variable is effectively final, just add the `final` keyword. If the code still compiles, the variable is effectively final.

Given this definition, which of the following variables are effectively final?

```
11: public String zooFriends() {  
12:     String name = "Harry the Hippo";
```

```
13:    var size = 10;
14:    boolean wet;
15:    if(size > 100) size++;
16:    name.substring(0);
17:    wet = true;
18:    return name;
19: }
```

Remember, a quick test of effectively final is to just add `final` to the variable declaration and see if it still compiles. In this example, `name` and `wet` are effectively final and can be updated with the `final` modifier, but not `size`. The `name` variable is assigned a value on line 12 and not reassigned. Line 16 creates a value that is never used. Remember from [Chapter 4](#), “Core APIs,” that strings are immutable. The `size` variable is not effectively final because it could be incremented on line 15. The `wet` variable is assigned a value only once and not modified afterward.

Effectively Final Parameters

Recall from [Chapter 1](#) that *method and constructor parameters are local variables that have been pre-initialized*. In the context of local variables, the same rules around `final` and effectively final apply. This is especially important in [Chapter 7](#) and [Chapter 8](#), “Lambdas and Functional Interfaces,” since local classes and lambda expressions declared within a method can only reference local variables that are `final` or effectively final.

Instance Variable Modifiers

Like methods, instance variables can have different access levels, such as `private`, `package`, `protected`, and `public`. Remember, package access is indicated by the lack of any modifiers. We cover each of the different access modifiers shortly in this chapter. Instance variables can also use optional specifiers, described in [Table 5.3](#).

TABLE 5.C Optional specifiers for instance variables

Modifier	Description	Chapter Covered
<code>final</code>	Specifies that the instance variable must be initialized with each instance of the class exactly once	Chapter 5
<code>volatile</code>	Instructs the JVM that the value in this variable may be modified by other threads	Chapter 13
<code>transient</code>	Used to indicate that an instance variable should not be serialized with the class	Chapter 14

Looks like we only need to discuss `final` in this chapter! 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";
    }
}
```

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. Failing to initialize an instance variable (or assigning a value more than once) will lead to a compiler error. We talk about `final` variable initialization in more detail when we cover constructors in the next chapter.



In [Chapter 1](#), we show that instance variables receive default values based on their type when not set. For example, `int` receives a default value of `0`, while an object reference receives a default value of `null`. The compiler does not apply a default value to `final` variables, though. A `final` instance or `final static` variable must receive a value when it is declared or as part of initialization.

Working with Varargs

As mentioned in [Chapter 4](#), a method may use a varargs parameter (variable argument) as if it is an array. Creating a method with a varargs parameter is a bit more complicated. In fact, calling such a method may not use an array at all.

Creating Methods with Varargs

There are a number of important rules for creating a method with a varargs parameter.

Rules for Creating a Method with a Varargs Parameter

1. A method can have at most one varargs parameter.
2. If a method contains a varargs parameter, it must be the last parameter in the list.

Given these rules, 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 class VisitAttractions {  
    public void walk1(int... steps) {}  
    public void walk2(int start, int... steps) {}  
    public void walk3(int... steps, int start) {}           // DOES NOT
```

```
COMPILE
    public void walk4(int... start, int... steps) {}    // 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.

Calling Methods with Varargs

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. Given our previous `walk1()` method, which takes a varargs parameter, we can call it one of two ways:

```
// Pass an array
int[] data = new int[] {1, 2, 3};
walk1(data);

// Pass a list of values
walk1(1,2,3);
```

Regardless of which one you use to call the method, the method will receive an array containing the elements. We can reinforce this with the following example:

```
public void walk1(int... steps) {
    int[] step2 = steps;    // Not necessary, but shows steps
    is of type int[]
    System.out.print(step2.length);
}
```

You can even omit the varargs values in the method call, and Java will create an array of length zero for you.

```
walk1();
```

Accessing Elements of a Vararg

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

```

16: public static void run(int... steps) {
17:     System.out.print(steps[1]);
18: }
19: public static void main(String[] args) {
20:     run(11, 77);    // 77
21: }

```

Line 20 calls a varargs method with two parameters. When the method is called, it sees an array of size 2. Since indexes are zero-based, 77 is printed.

Using Varargs with Other Method Parameters

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? For now, feel free to ignore the `static` modifier in the `walkDog()` method declaration; we cover that later in the chapter.

```

1: public class DogWalker {
2:     public static void walkDog(int start, int... steps) {
3:         System.out.println(steps.length);
4:     }
5:     public static void main(String[] args) {
6:         walkDog(1);           // 0
7:         walkDog(1, 2);        // 1
8:         walkDog(1, 2, 3);     // 2
9:         walkDog(1, new int[] {4, 5}); // 2
10:    } }

```

Line 6 passes 1 as `start` but nothing else. This means Java creates an array of length 0 for `steps`. Line 7 passes 1 as `start` and one more value. Java converts this one value to an array of length 1. Line 8 passes 1 as `start` and two more values. Java converts these two values to an array of length 2. Line 9 passes 1 as `start` and an array of length 2 directly as `steps`.

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. The following snippet does compile:

```

walkDog(1, null);    // Triggers NullPointerException in
walkDog()

```

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 `walkDog()`. Then the

`walkDog()` method throws an exception because it tries to determine the length of `null`.

Applying Access Modifiers

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

- `private`: Only accessible within the same class.
- **Package access**: `private` plus other members of the same package. Sometimes referred to as package-private or default access.
- `protected`: Package access plus access within subclasses.
- `public`: `protected` plus classes in the other packages.

We will explore the impact of these four levels of access on members of a class.

Private Access

Let's start with `private` access, which is the simplest. Only code in the same class can call `private` methods or access `private` fields.

First, take a look at [Figure 5.2](#). It shows the classes you'll use to explore `private` and `package` 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.

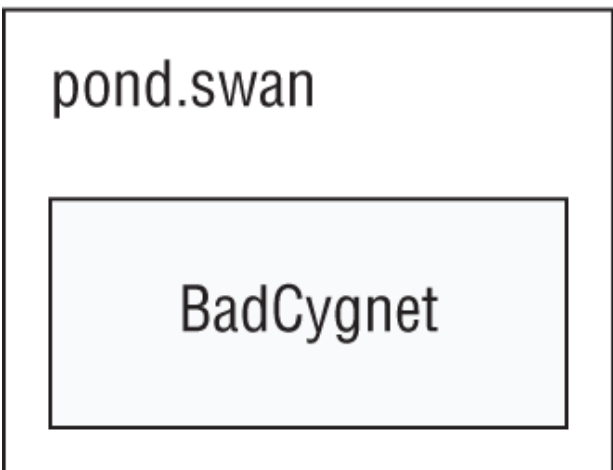
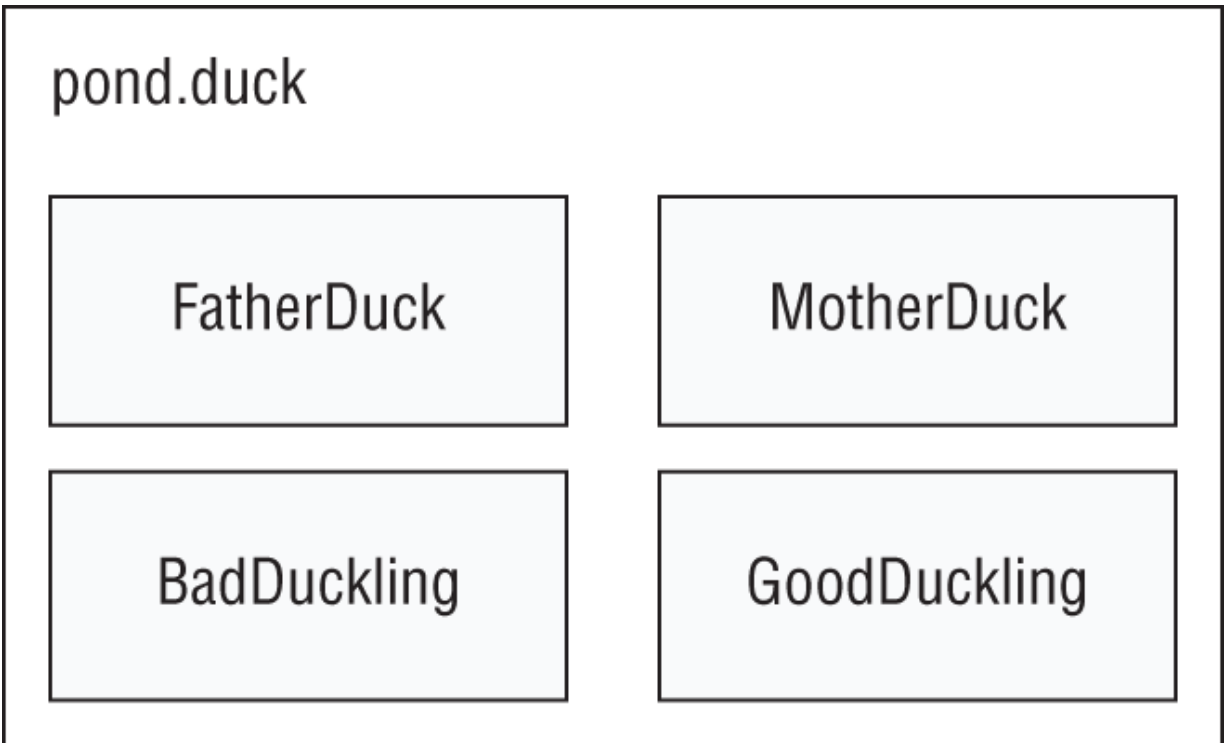


FIGURE 5.2 Classes used to show `private` and package access

This is perfectly legal code because everything is one class:

```
1: package pond.duck;
2: public class FatherDuck {
3:     private String noise = "quack";
4:     private void quack() {
5:         System.out.print(noise);           // private access is
ok
```

```
6:     }  
7: }
```

So far, so good. `FatherDuck` declares a `private` method `quack()` 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:         var duck = new FatherDuck();  
5:         duck.quack();           // DOES NOT COMPILE  
6:         System.out.print(duck.noise); // DOES NOT COMPILE  
7:     }  
8: }
```

`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.



In the previous example, `FatherDuck` and `BadDuckling` are in separate files, but what if they were declared in the same file? Even then, the code would still not compile as Java prevents access outside the class.

Package 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 assumes package access.

```

package pond.duck;
public class MotherDuck {
    String noise = "quack";
    void quack() {
        System.out.print(noise);           // package 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 that MotherDuck lets other classes in the same package access members, 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() {
        var duck = new MotherDuck();
        duck.quack();                     // package access is
ok
        System.out.print(duck.noise);      // package access is
ok
    }
}

```

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 package 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() {
        var duck = new MotherDuck();
        duck.quack();                   // DOES NOT COMPILE
        System.out.print(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 package access does, and more. The `protected` access modifier adds the ability to access members of a parent class. We cover creating subclasses in depth in [Chapter 6](#). For now, we cover the simplest possible use of a subclass. In the following example, the “child” `ClownFish` class is a subclass of the “parent” `Fish` class, using the `extends` keyword to connect them:

```
public class Fish {}

public class ClownFish extends Fish {}
```

By extending a class, the subclass gains access to all `protected` and `public` members of the parent class, as if they were declared in the subclass. If the two classes are in the same package, then the subclass also gains access to all package members.

[Figure 5.3](#) shows the many classes we 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.

pond.shore

Bird

BirdWatcher

pond.goose

Gosling
(extends Bird)

Goose
(extends Bird)

pond.inland

BirdWatcherFromAfar

pond.swan

Swan
(extends Bird)

pond.duck

GooseWatcher

FIGURE 5.C 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 void floatInWater() {
        System.out.print(text);           // protected access is ok
    }
}
```

Next, we create a subclass:

```
package pond.goose;                               // Different package than
Bird                                              Bird
import pond.shore.Bird;
public class Gosling extends Bird {              // Gosling is a subclass
of Bird
    public void swim() {
        floatInWater();                       // protected access is ok
        System.out.print(text);               // protected access is ok
    }
    public static void main(String[] args) {
        new Gosling().swim();
    }
}
```

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 program 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 package access does. This means a class in the same package as `Bird` can access its protected members.

```
package pond.shore;                               // Same package as
Bird                                              Bird
public class BirdWatcher {
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater();                 // protected access is
```

```

ok      System.out.print(bird.text);          // protected access is
ok      }
ok      }

```

Since `Bird` and `BirdWatcher` are in the same package, `BirdWatcher` can access package 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;                // Different package
than Bird
import pond.shore.Bird;
public class BirdWatcherFromAfar {    // Not a subclass of
Bird
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater();          // DOES NOT COMPILE
        System.out.print(bird.text);  // DOES NOT COMPILE
    }
}

```

`BirdWatcherFromAfar` is not in the same package as `Bird`, and it doesn't inherit from `Bird`. This means 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:

```

1: package pond.swan;                // Different package
than Bird
2: import pond.shore.Bird;
3: public class Swan extends Bird {   // Swan is a subclass
of Bird
4:     public void swim() {
5:         floatInWater();            // protected access is
ok
6:         System.out.print(text);      // protected access is
ok
7:     }
8:     public void helpOtherSwanSwim() {

```

```

9:         Swan other = new Swan();
10:        other.floatInWater();           // subclass access to
superclass
11:        System.out.print(other.text);   // subclass access to
superclass
12:    }
13:    public void helpOtherBirdSwim() {
14:        Bird other = new Bird();
15:        other.floatInWater();           // DOES NOT COMPILE
16:        System.out.print(other.text);   // DOES NOT COMPILE
17:    }
18: }

```

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 OK. 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 15 and 16 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 14. `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 OK 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, 15, and 16. 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.print(other.text);
    }
    public void helpOtherGooseSwim() {
        Bird other = new Goose();
        other.floatInWater();           // DOES NOT COMPILE
        System.out.print(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` class. 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()`, 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.



The Java module system redefines “anywhere,” and it becomes possible to restrict access to `public` code outside a module. We cover this in more detail in [Chapter 12](#), “Modules.” When given code samples, you can assume they are in the same 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 void swim() {
        System.out.print(name);           // public access is
ok
    }
}
```

`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() {
        var teacher = new DuckTeacher();
    }
}
```

```

        teacher.swim();                                //
allowed
        System.out.print("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`.

Reviewing Access Modifiers

Make sure you know why everything in [Table 5.4](#) is true. Use the first column for the first blank and the first row for the second blank. Also, remember that a member is a method or field.

TABLE 5.4 A method in _____ can access a _____ member.

	private	package	protected	public
the same class	Yes	Yes	Yes	Yes
another class in the same package	No	Yes	Yes	Yes
a subclass in a different package	No	No	Yes	Yes
an unrelated class in a different package	No	No	No	Yes

Accessing Static Data

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

Designing Static Methods and Variables

Except for the `main()` method, we've been looking at instance methods. Methods and variables declared `static` don't require an instance of the

class. They are shared among all users of the class. For instance, take a look at the following `Penguin` class:

```
public class Penguin {  
    String name;  
    static String nameOfTallestPenguin;  
}
```

In this class, every `Penguin` instance has its own name like `Willy` or `Lilly`, but only one `Penguin` among all the instances is the tallest. 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. Consider the following example:

```
public static void main(String[] unused) {  
    var p1 = new Penguin();  
    p1.name = "Lilly";  
    p1.nameOfTallestPenguin = "Lilly";  
    var p2 = new Penguin();  
    p2.name = "Willy";  
    p2.nameOfTallestPenguin = "Willy";  
  
    System.out.println(p1.name);           // Lilly  
    System.out.println(p1.nameOfTallestPenguin); // Willy  
    System.out.println(p2.name);           // Willy  
    System.out.println(p2.nameOfTallestPenguin); // Willy  
}
```

We see that each penguin instance is updated with its own unique name. The `nameOfTallestPenguin` field is static and therefore shared, though, so anytime it is updated, it impacts all instances of the 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.print(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 look at some examples covering other `static` concepts.

Accessing a Static Variable or Method

Usually, accessing a `static` member is easy.

```
public class Snake {  
    public static long hiss = 2;  
}
```

You just put the class name before the method or variable, and you are done. Here's an example:

```
System.out.println(Snake.hiss);
```

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: Snake s = new Snake();
6: System.out.println(s.hiss); // s is a Snake
7: s = null;
8: System.out.println(s.hiss); // s is still a Snake
```

Believe it or not, this code outputs 2 twice. Line 6 sees that `s` is a `Snake` and `hiss` is a `static` variable, so it reads that `static` variable. Line 8 does the same thing. Java doesn't care that `s` happens to be `null`. Since we are looking for a `static` variable, 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?

```
Snake.hiss = 4;
Snake snake1 = new Snake();
Snake snake2 = new Snake();
snake1.hiss = 6;
snake2.hiss = 5;
System.out.println(Snake.hiss);
```

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

Class vs. Instance Membership

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 MantaRay {
    private String name = "Sammy";
    public static void first() { }
    public static void second() { }
    public void third() { System.out.print(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 an instance method. If we fix this by adding `static` to `third()`, we create a new problem. Can you figure out what it is?

```
    public static void third() { System.out.print(name); } //
DOES NOT COMPILE
```

All this does is move the problem. Now, `third()` is referring to an instance variable `name`. There are two ways we could fix this. The first is to add `static` to the `name` variable as well.

```
public class MantaRay {
    private static String name = "Sammy";
    ...
    public static void third() { System.out.print(name); }
    ...
}
```

The second solution would have been to call `third()` as an instance method and not use `static` for the method or the variable.

```
public class MantaRay {
    private String name = "Sammy";
    ...
    public void third() { System.out.print(name); }
    public static void main(String args[]) {
        ...
    }
}
```

```

        var ray = new MantaRay();
        ray.third();
    }
}

```

The exam creators like this topic—a lot. 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 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 5.5](#) before continuing.

TABLE 5.5 Static vs. instance calls

Method	Calling	Legal?
<code>allGiraffeGoHome()</code>	<code>allGiraffeComeOut()</code>	Yes
<code>allGiraffeGoHome()</code>	<code>drink()</code>	No
<code>allGiraffeGoHome()</code>	<code>g.eat()</code>	Yes
<code>eat()</code>	<code>allGiraffeComeOut()</code>	Yes
<code>eat()</code>	<code>drink()</code>	Yes
<code>eat()</code>	<code>g.eat()</code>	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?

```

1: public class Gorilla {
2:     public static int count;
3:     public static void addGorilla() { count++; }
4:     public void babyGorilla() { count++; }
5:     public void announceBabies() {
6:         addGorilla();

```

```

7:         babyGorilla();
8:     }
9:     public static void announceBabiesToEveryone() {
10:         addGorilla();
11:         babyGorilla();        // DOES NOT COMPILE
12:     }
13:     public int total;
14:     public static double average
15:         = total / count;    // DOES NOT COMPILE
16: }

```

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 is called, it increments `count` by one. 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 1](#) 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.



Make sure you understand this section really well. It comes up throughout this book. You even see a similar topic when we discuss interfaces in [Chapter 7](#). For example, a `static` interface method cannot call a `default` interface method without a reference, much the same way that within a class, a `static` method cannot call an instance method without a reference.

Static Variable Modifiers

Referring back to [Table 5.3](#), `static` variables can be declared with the same modifiers as instance variables, such as `final`, `transient`, and `volatile`. While some `static` variables are meant to change as the program runs, like our `count` example, others are meant to never change. This type of `static` 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 ZooPen {
    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?

```
import java.util.*;
public class ZooInventoryManager {
    private static final String[] treats = new String[10];
    public static void main(String[] args) {
        treats[0] = "popcorn";
    }
}
```

```
    }  
}
```

It actually does compile since `treats` is a reference variable. We are allowed to modify the referenced object or array's contents. All the compiler can do is check that we don't try to reassign `treats` to point to a different object.

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. Remember, `final` variables must be initialized with a value. Next, we cover `static` initializers.

Static Initializers

In [Chapter 1](#), we covered instance initializers that looked like unnamed methods—just code inside braces. Now we introduce `static` initializers, which look similar. We just add the `static` keyword to specify that 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 OK.

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 be initialized. Since the programmer forgot, this is clearly an error.

Real World Scenario

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 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` or a `HashMap`. 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 you can import a specific class or all the classes in a package. If you haven't seen `ArrayList` or `List` before, don't worry, because we cover them in detail in [Chapter 9](#), "Collections and Generics."

```
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, while `static`

imports are for importing `static` members of classes like variables and methods.

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.

We can rewrite our previous example to use a `static` import. Doing so yields the following:

```
import java.util.List;
import static java.util.Arrays.asList;           // static
import
public class ZooParking {
    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 `ZooParking` 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 by 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 BadZooParking {
5:     public static void main(String[] args) {
6:         Arrays.asList("one");             // DOES NOT COMPILE
7:     }
8: }
```

Line 1 tries to use a `static import` to import a class. Remember that `static imports` are only for importing `static` members like a method or variable. 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 OK 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 zoo.A.TYPE;
import static zoo.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`.



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. Use them sparingly!

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.print(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 is 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.

Passing Objects

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 class Dog {
    public static void main(String[] args) {
        String name = "Webby";
        speak(name);
        System.out.print(name);
    }
    public static void speak(String name) {
        name = "Georgette";
    }
}

```

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 class Dog {
    public static void main(String[] args) {
        var name = new StringBuilder("Webby");
        speak(name);
        System.out.print(name);    // WebbyGeorgette
    }
    public static void speak(StringBuilder s) {
        s.append("Georgette");
    }
}

```

In this case, `speak()` calls a method on the parameter. It doesn't reassign `s` to a different object. In [Figure 5.4](#), 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.

An image structure of a pass by value. The parameter string builder object is pointed out by arrows from `name` and `s` variable.

[FIGURE 5.4](#) 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. In a pass-by-reference language, the variables would be swapped and the output would be reversed.

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.

Returning Objects

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.

```
1: public class ZooTickets {
2:     public static void main(String[] args) {
3:         int tickets = 2;                                // tickets
= 2
4:         String guests = "abc";                          // guests
= abc
5:         addTickets(tickets);                            // tickets
= 2
6:         guests = addGuests(guests);                    // guests
= abcd
7:         System.out.println(tickets + guests);          // 2abcd
8:     }
9:     public static int addTickets(int tickets) {
10:         tickets++;
11:         return tickets;
12:     }
13:     public static String addGuests(String guests) {
14:         guests += "d";
15:         return guests;
16:     }
17: }
```

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 3 but leaves the `tickets` variable in the `main()` method as 2. While line 11 returns the value, the caller ignores it. The method call on line 6 doesn't ignore the result, so `guests` becomes "abcd". Remember that this is happening because of the returned value and not the method parameter.

Autoboxing and Unboxing Variables

Java supports some helpful features around passing primitive and wrapper data types, such as `int` and `Integer`. Remember from [Chapter 1](#) that we can explicitly convert between primitives and wrapper classes using built-in methods.


```

5: int quack = 5;
6: Integer quackquack = Integer.valueOf(quack); // Convert int
to Integer
7: int quackquackquack = quackquack.intValue(); // Convert
Integer to int

```

Useful, but a bit verbose. Luckily, Java has handlers built into the Java language that automatically convert between primitives and wrapper classes and back again. *Autoboxing* is the process of converting a primitive into its equivalent wrapper class, while *unboxing* is the process of converting a wrapper class into its equivalent primitive.

```

5: int quack = 5;
6: Integer quackquack = quack; // Autoboxing
7: int quackquackquack = quackquack; // Unboxing

```

The new code is equivalent to the previous code, as the compiler is “doing the work” of converting the types automatically for you. Autoboxing applies to all primitives and their associated wrapper types, such as the following:

```

Short tail = 8; // Autoboxing
Character p = Character.valueOf('p');
char paw = p; // Unboxing
Boolean nose = true; // Autoboxing
Integer e = Integer.valueOf(9);
long ears = e; // Unboxing, then
implicit casting

```

Each of these examples compiles without issue. In the last line, `e` is unboxed to an `int` value. Since an `int` value can be stored in a `long` variable via implicit casting, the compiler allows the assignment.

Limits of Autoboxing and Numeric Promotion

While Java will implicitly cast a smaller primitive to a larger type, as well as autobox, it will not do both at the same time. Do you see why the following does not compile?

```
Long badGorilla = 8; // DOES NOT COMPILE
```

The compiler will automatically cast or autobox the `int` value to `long` or `Integer`, respectively. Neither of these types can be assigned to a `Long` reference variable, though, so the code does not compile. Compare this behavior to the previous example with `ears`, where the unboxed primitive value could be implicitly cast to a larger primitive type.

What do you think happens if you try to unbox a `null`?

```
10: Character elephant = null;
11: char badElephant = elephant; // NullPointerException
```

On line 10, we store `null` in a `Character` reference. This is legal because a `null` reference can be assigned to any reference variable. On line 11, we try to unbox that `null` to a `char` primitive. This is a problem. Java tries to get the `char` value of `null`. Since calling any method on `null` gives a `NullPointerException`, that is just what we get. Be careful when you see `null` in relation to autoboxing and unboxing.

Where autoboxing and unboxing really shine is when we apply them to method calls.

```
public class Chimpanzee {
    public void climb(long t) {}
    public void swing(Integer u) {}
    public void jump(int v) {}
    public static void main(String[] args) {
        var c = new Chimpanzee();
        c.climb(123);
        c.swing(123);
        c.jump(123L); // DOES NOT COMPILE
    }
}
```

```
    }  
}
```

In this example, the call to `climb()` compiles because the `int` value can be implicitly cast to a `long`. The call to `swing()` also is permitted, because the `int` value is autoboxed to an `Integer`. On the other hand, the call to `jump()` results in a compiler error because a `long` must be explicitly cast to an `int`. In other words, Java will not automatically convert to a narrower type.

As before, the same limitation around autoboxing and numeric promotion applies to method calls. For example, the following does not compile:

```
public class Gorilla {  
    public void rest(Long x) {  
        System.out.print("long");  
    }  
    public static void main(String[] args) {  
        var g = new Gorilla();  
        g.rest(8);    // DOES NOT COMPILE  
    }  
}
```

Java will cast or autobox the value automatically, but not both at the same time. Finally, autoboxing can be used when initializing an array. The following creates two arrays with `Integer` and `Double` values, respectively.

```
Integer[] openingHours    = { 9, 12 };  
Double[] temperaturesAtZoo = { 74.1, 93.2 };
```

The types have to be compatible, though, as shown in the following examples.

```
Integer[] winterHours = { 10.5, 17.0 };    // DOES NOT COMPILE  
Double[] summerHours  = { 9, 21 };         // DOES NOT COMPILE
```

Overloading Methods

Now that you are familiar with the rules for declaring and using methods, it is time to look at creating methods with the same name in the same class. *Method overloading* occurs when methods in the same class have the same name but different method signatures, which means they use different

parameter lists. (Overloading differs from overriding, which you learn about in [Chapter 6](#).)

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, optional specifiers (like `static`), return types, and exception lists.

The following shows five overloaded versions of the `fly()` method:

```
public class Falcon {
    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. Only the method name and parameter list matter.

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

```
public class Eagle {
    public void fly(int numMiles) {}
    public int fly(int numMiles) { return 1; }    // DOES NOT
COMPILE
}
```

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

What about these; why do they not compile?

```

public class Hawk {
    public void fly(int numMiles) {}
    public static void fly(int numMiles) {}    // DOES NOT
COMPILE
    public void fly(int numKilometers) {}    // DOES NOT
COMPILE
}

```

Again, the method signatures of these three methods are the same. You cannot declare methods in the same class where the only difference is that one is an instance method and one is a `static` method. You also cannot have two methods that have parameter lists with the same variable types and in the same order. As we mentioned earlier, the names of the parameters in the list do not matter when determining the method signature.

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

```

public class Dove {
    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.

Reference Types

Java picks the most specific version of a method that it can. What do you think this code outputs?

```

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

    public void fly(Object o) {
        System.out.print("object");
    }
}

```

```

    }
    public static void main(String[] args) {
        var p = new Pelican();
        p.fly("test");
        System.out.print("-");
        p.fly(56);
    }
}

```

The answer is `string-object`. The first call passes 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. What does this print?

```

import java.time.*;
import java.util.*;
public class Parrot {
    public static void print(List<Integer> 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(Arrays.asList(3));
        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 4](#), `String` and `StringBuilder` implement the `CharSequence` interface. You also learned that `Arrays.asList()` can be used to create a `List<Integer>` object, which explains the second output. The final call to `print()` passes a `LocalDate`. This is a class you might not know, but that's OK. It clearly isn't a sequence of characters or a list. 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 Ostrich {
    public void fly(int i) {
        System.out.print("int");
    }
    public void fly(long l) {
        System.out.print("long");
    }
    public static void main(String[] args) {
        var p = new Ostrich();
        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.

Autoboxing

As we saw earlier, autoboxing applies to method calls, but what happens if you have both a primitive and an integer version?

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

These method overloads are valid. *Java tries to use the most specific parameter list it can find.* This is true for autoboxing as well as other matching types we talk about in this section.

This means calling `fly(3)` will call the first method. When the primitive `int` version isn't present, Java will autobox. However, when the primitive

`int` version is provided, there is no reason for Java to do the extra work of autoboxing.

Arrays

Unlike the previous example, this code does not result in autoboxing:

```
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. What about generic types, such as `List<Integer>`? We cover this topic in [Chapter 9](#).

Varargs

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

```
public class Toucan {  
    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 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 the two methods are similar, it is time to mention how they are different. It shouldn't be a surprise that you can call either method by passing an array:

```
fly(new int[] { 1, 2, 3 }); // Allowed to call either fly()  
method
```

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

```
fly(1, 2, 3); // Allowed to call only the fly() method using  
varargs
```


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.

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 5.6](#) lays it out for you.

[TABLE 5.6](#) The order that Java uses to choose the right overloaded method

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

Let's give this a practice run using the rules in [Table 5.6](#). What do you think this outputs?

```
public class Glider {
    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.

Summary

In this chapter, we presented a lot of rules for declaring methods and variables. Methods start with access modifiers and optional specifiers in any order (although commonly with access modifiers first). The access modifiers we discussed in this chapter are `private`, `package` (omitted), `protected`, and `public`. The optional specifier for methods we covered in this chapter is `static`. We cover additional method modifiers in future chapters.

Next comes the method return type, which is `void` if there is no return value. The method name and parameter list are provided next, which compose the unique method signature. The method name uses standard Java identifier rules, while the parameter list is composed of zero or more types with names. An optional list of exceptions may also be added following the parameter list. Finally, a block defines the method body (which is omitted for `abstract` methods).

Access modifiers are used for a lot more than just methods, so make sure you understand them well. Using the `private` keyword means the code is only available from within the same class. Package 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 class name—for example, `Pigeon.fly()`. Instance members are allowed to call `static` members, but `static` members are not allowed to call instance members. In addition, `static` imports are used to import `static` members.

We also presented the `final` modifier and showed how it can be applied to local, instance, and `static` variables. Remember, a local variable is effectively final if it is not modified after it is assigned. One quick test for this is to add the `final` modifier and see if the code still compiles.

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. Java supports autoboxing and unboxing of primitives and wrappers automatically within a method and through method calls.

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.

Make sure you understand everything in this chapter. It sets the foundation of what you learn in the next chapters.

Exam Essentials

Be able to identify correct and incorrect method declarations. Be able to view a method signature and know if it is correct, contains invalid or conflicting elements, or contains elements in the wrong order.

Identify when a method or field is accessible. Recognize when a method or field is accessible when the access modifier is: `private`, `package` (omitted), `protected`, or `public`.

Understand how to declare and use final variables. Local, instance, and `static` variables may be declared `final`. Be able to understand how to declare them and how they can (or cannot) be used.

Be able to spot effectively final variables. Effectively final variables are local variables that are not modified after being assigned. Given a local variable, be able to determine if it is effectively final.

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.

Apply autoboxing and unboxing. The process of automatically converting from a primitive value to a wrapper class is called autoboxing, while the reciprocal process is called unboxing. Watch for a `NullPointerException` when performing unboxing.

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 can.