Appendix C

Java Classes

Contents

Prerequisites

Appendix B Java Basics

This appendix reviews the use and creation of Java classes, methods, and packages. Even if you are familiar with this material, you should at least skim it to learn our terminology.

Objects and Classes

C.1 Appendix B introduced the basics of classes and objects. Recall that an object contains data and can perform certain actions. An object belongs to a class, which defines its data type. A class specifies the kind of data the objects of that class have. A class also specifies what actions the

objects can take and how they accomplish those actions. **Object-oriented programming**, or **OOP**, views a program as a sort of world consisting of objects that interact with one another by means of actions. For example, in a program that simulates automobiles, each automobile is an object.

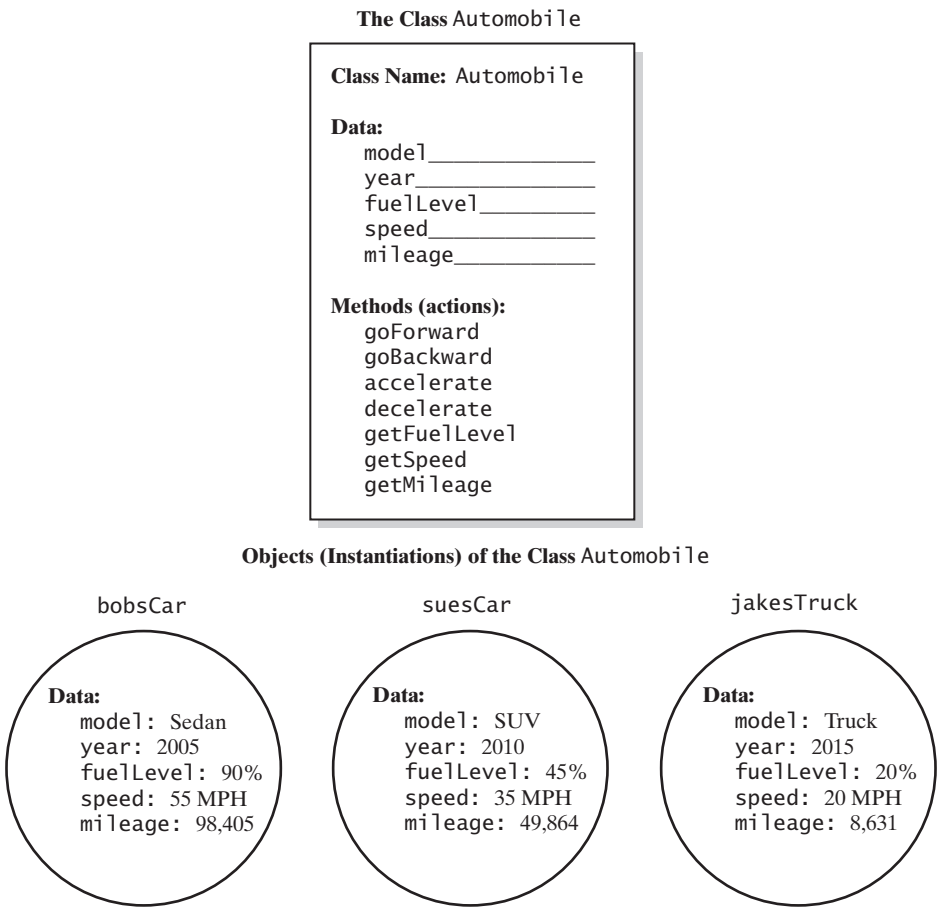
When you define a class in Java, the class is like a plan or a blueprint for constructing specific objects. As an example, Figure C-1 describes a class called `Automobile` and shows three `Automobile` objects. The class is a general description of what an automobile is and what it can do.

Each **instance**, or object, of the class `Automobile` is a particular automobile. You can name each object that you create, or **instantiate**. In Figure C-1, the names are `bobsCar`, `suesCar`, and `jakesTruck`. In a Java program, `bobsCar`, `suesCar`, and `jakesTruck` would be variables of type `Automobile`.

The definition of the `Automobile` class says that an `Automobile` object has data such as its model, its year, and how much fuel is in its tank. The class definition contains no actual data—no string and no numbers. The individual objects have the data, and the class simply specifies what kind of data they have.

The `Automobile` class also defines methods such as `goForward` and `goBackward`. In a program that uses the class `Automobile`, the only actions an `Automobile` object can take are defined by those methods. All objects of a given class have exactly the same actions. The implementations of the methods indicate how the actions are performed and are included in the class definition. The objects themselves actually perform the method's actions, however.

FIGURE C-1 An outline of a class and three of its instances



The objects in a single class can have different characteristics. Even though these objects have the same types of data and the same actions, the individual objects can differ in the values of their data.

Note: An object is a program construct that contains data and performs actions. The objects in a Java program interact, and this interaction forms the solution to a given problem. The actions performed by objects are defined by methods.

Note: A class is a type or kind of object. All objects in the same class have the same kinds of data and the same actions. A class definition is a general description of what that object is and what it can do.

Note: You can view a class in several different ways when programming. When you instantiate an object of a class, you view the class as a data type. When you implement a class, you can view it as a plan or a blueprint for constructing objects—that is, as a definition of the objects' data and actions. At other times, you can think of a class as a collection of objects that have the same type.

Using the Methods in a Java Class

C.2 Let's assume that someone has written a Java class called `Name` to represent a person's name. We will describe how to use this class and, in doing so, we will show you how to use a class's methods. A program component that uses a class is called a **client** of the class.We will reserve the term "user" to mean a person who uses a program.

To declare a variable of data type `Name`, you would write, for example,

```
Name joe;
```

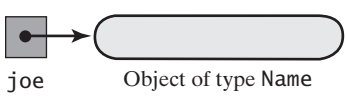
At this point, the variable `joe` contains nothing in particular; it is uninitialized. To create a specific object of data type `Name`—that is, to create an instance of `Name`—called `joe`, you write

```
joe = new Name();
```

The `new` operator creates an instance of `Name` by invoking a special method within the class, known as a **constructor**. The memory address of the new object is assigned to `joe`, as Figure C-2 illustrates. We will show you how to define constructors a bit later, in Segment C.17. Note that you can combine the previous two Java statements into one:

```
Name joe = new Name();
```

FIGURE C-2 A variable that references an object



C.3 Suppose that a person's name has only two parts: a first name and a last name. The data associated with the object `joe` then consists of two strings that represent the first and last names. Since you want to be able to **set**—that is, initialize or change—a person's name, the `Name` class should have methods that give you this capability. To set `joe`'s first and last names, you can use two methods from the class `Name`—`setFirst` and `setLast`—as follows:

```
joe.setFirst("Joseph");
joe.setLast("Brown");
```

You usually invoke a method by writing the name of the **receiving object** first, followed by a dot, the name of the method to be invoked, and finally a set of parentheses that contain **arguments**. In this example, `joe` is the receiving object, as it receives the call to perform an action, and the arguments are strings that represent inputs to the methods. The methods set the object's data fields to the specific values given as arguments.

The methods `setFirst` and `setLast` are examples of void methods, in that they do not return a value. As we mentioned in Segment B.2 of Appendix B, a second kind of method—the valued method—returns a single value. For example, the method `getFirst` returns a string that is the first name of the object that received the method call. Similarly, the method `getLast` returns the last name.

You can invoke a valued method anywhere that you can use a value of the type returned by the method. For example, `getFirst` returns a value of type `String`, and so you can use a method invocation such as `joe.getFirst()` anywhere that it is legal to use a value of type `String`. Such places could be in an assignment statement, like

```
String hisName = joe.getFirst();
```

or within a `println` statement, like

```
System.out.println("Joe's first name is " + joe.getFirst());
```

Notice that the methods `getFirst` and `getLast` have no arguments in their parentheses. Any method—valued or void—can require zero or more arguments.

Note: Valued methods return a single value; void methods do not return a value. For example, the valued method `getFirst` returns the string that represents the first name. The void method `setFirst` sets the first name to a given string but does not return a value. For now, you will distinguish valued methods and void methods by the description of what they do. Later, in Segments C.7 through C.9, you will see that their Java definitions distinguish one kind of method from another.

Question 1 Write Java statements that create an object of type `Name` to represent your name.

Question 2 Write a Java statement that uses the object you created in Question 1 to display your name in the form *last name, comma, first name*.

Question 3 Which methods of the class `Automobile`, as given in Figure C-1, are most likely valued methods, and which are most likely void methods?

References and Aliases

C.4 Java has eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. A variable of a primitive type actually contains the primitive value. All other data types are reference—that is, class or array—types. The `String` variable `greeting` in

```
String greeting = "Hello";
```

is a reference variable. As we discussed in Segment B.14, reference variable contains the address in memory of an actual object. This address is called a reference. It is not important here to know that `greeting` contains a reference to the string `"Hello"` instead of the actual string. In such cases, it is easier to talk about the string `greeting`, when in fact this is not an accurate description of that variable. This book makes the distinction between an object and a reference to an object when it is important to do so.

Now suppose that you write

```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");
Name friend = jamie;
```

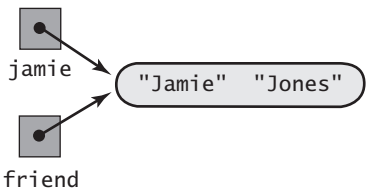
The two variables `jamie` and `friend` reference the same instance of `Name`, as Figure C-3 shows. We say that `jamie` and `friend` are aliases, because they are two different names for the same object. You can use `jamie` and `friend` interchangeably when referencing the object.

For example, if you use the variable `jamie` to change Jamie Jones's last name, you can use the variable `friend` to access it. Thus, the statements

```
jamie.setLast("Smith");
System.out.println(friend.getLast());
```

display *Smith*. Also note that the boolean expression `jamie == friend` is true, since both variables contain the same address.

FIGURE C-3 Aliases of an object



Defining a Java Class

C.5 We now show you how to write the Java class `Name` that represents a person's name. You store a class definition in a file whose name is the name of the class followed by `.java`. Thus, the class `Name` should be in the file `Name.java`. Typically, you store only one class per file.

The data in a `Name` object consists of the person's first and last names as strings. The methods in the class will enable you to set and look at these strings. The class has the following form:

```
public class Name
{
   private String first; // First name
   private String last;  // Last name

   < Definitions of methods are here >

   . . .
} // end Name
```

The word `public` simply means that there are no restrictions on where the class is used. That is, the class `Name` is available for use in any other Java class. The two strings `first` and `last` are called the class's **data fields** or **instance variables** or **data members**. Each object of this class will have these two data fields inside of it. The word `private` that precedes the declaration of each data field means that only the methods within the class can refer to the data fields by their names `first` and `last`. No other class will be able to do this. The words `public` and `private` are examples of an **access modifier** or **visibility modifier**, which specifies where a class, data field, or method can be used. A third access modifier, `protected`, is possible, as you will see in Appendix D.

Note: Access (visibility) modifiers
The words `public` and `private` are examples of access modifiers that specify where a class, method, or data field can be used. Any class can use a public method, but a private method can be used only by the class that defines it. Appendix D discusses the access modifier `protected`, and the section "Packages" of this appendix shows when you can omit the access modifier.

C.6 Since the data fields are private, how will a class that uses the class `Name` be able to change or look at their values? You can define methods in a class that look at or change the values of its data fields. You declare such methods to be public, so that anyone can use them. A method that enables you to look at the value of a data field is called an **accessor method** or **query method**. A method that changes the value of a data field is called a **mutator method**. Java programmers typically begin the names of accessor methods with `get` and the names of mutator methods with `set`. Because of this convention, accessor methods are sometimes called **get methods** or **getters**, and mutator methods are called **set methods** or **setters**. For example, the class `Name` will have methods that include `getFirst`, `getLast`, `setFirst`, and `setLast`.

You may think that accessor methods and mutator methods defeat the purpose of making data fields private. On the contrary, they give the class control over its data fields. For example, a mutator method can check that any change to a data field is appropriate and warn you if there is a problem. The class would be unable to make this check if its data fields were public, since anyone could alter the fields.

Note: An accessor (query) method enables you to look at the value of a data field. A mutator method changes the value of a data field. Typically, you begin the names of accessor methods with `get` and the names of mutator methods with `set`.

Programming Tip: You should make each data field in a class private by beginning its declaration with the access modifier `private`. You cannot make any direct reference to a private data field's name outside of the class definition. The programmer who uses the class is forced to manipulate the data fields only via methods in the class. The class then can control how a programmer accesses or changes the data fields. Within any of the class's method definitions, however, you can use the name of the data field in any way you wish. In particular, you can directly change the value of the data field.

Question 4 Is the method `setFirst` an accessor method or a mutator method?

Question 5 Should a typical accessor method be valued or void?

Question 6 Should a typical mutator method be valued or void?

Question 7 What is a disadvantage of making a data field in a class public?

Method Definitions

C.7 The definition of a method has the following general form:

```
access-modifier use-modifier return-type method-name(parameter-list)
{
  method-body
}
```

The **use modifier** is optional and in most cases is omitted. When present, it can be either `abstract`, `final`, or `static`. Briefly, an abstract method has no definition and must be overridden in a derived class. A final method cannot be overridden in a derived class. A static method is shared by all instances of the class. You will encounter these use modifiers later.

Next comes the **return type**, which for a valued method is the data type of the value that the method returns. For a void method, the return type is `void`. You then write the name of the method and a pair of parentheses that contain an optional list of **parameters** and their data types. The parameters specify values or objects that are inputs to the method.

So far, we have described the first line of the method definition, which is called the method's **header** or **declaration**. After the header is the method's **body**—which is simply a sequence of Java statements—enclosed in curly braces.

C.8 As an example of a valued method, here is the definition of the method `getFirst`:

```
public String getFirst()  <-- Header
{
   return first;          } Body
} // end getFirst
```

This method returns the string in the data field `first`. The return type of this method is, therefore, `String`. A valued method must always execute a `return` statement as its last action. The data type of the value returned must match the data type declared as the return type in the method's header. Notice that this particular method does not have parameters.

C.9 Now let's look at an example of a void method. The void method `setFirst` sets the data field `first` to a string that represents a first name. The method definition is as follows:

```
public void setFirst(String firstName)
{
   first = firstName;
} // end setFirst
```

This method does not return a value, so its return type is `void`. The method has one parameter, `firstName`, that has the data type `String`. It represents the string that the method should assign to the data field `first`. The declaration of a parameter always consists of a data type and a name. If you have more than one parameter, you separate their declarations with commas.

C.10 **The object `this`**. Notice that the bodies of the previous two method definitions refer to the data field `first` by name. This is perfectly legal. Exactly whose data field is involved here? Remember that each object of this class contains a data field `first`. The data field `first` that belongs to the object receiving the call to the method is the one involved. Java has a name for this object when you want to refer to it within the body of a method definition. It is simply `this`. For example, in the method `setFirst` you could write the statement

```
first = firstName;
```

as

```
this.first = firstName;
```

Some programmers use `this` in this way either for clarity or when they want to give the parameter the same name as the data field. For example, you could name `setFirst`'s parameter `first` instead of `firstName`. Clearly, the statement

```
first = first;
```

in the method's body would not work correctly, so instead you would write

```
this.first = first;
```

We typically will not use `this` in the ways shown in these examples. However, Segment C.26 will show an essential use of `this`.

Note: Members
Both the data fields and the methods of an object are sometimes called **members** of the object, because they belong to the object.

Note: Naming classes and methods
The normal convention when naming classes and methods is to start all class names with an uppercase letter and to start all method names with a lowercase letter. Use a noun or descriptive phrase to name a class. Use a verb or action phrase to name a method.

Note: Local variables
A variable declared within a method definition is called a **local variable**. The value of a local variable is not available outside of the method definition. If two methods each have a local variable of the same name, the variables are different, even though they have the same name.

C.11 Methods should be self-contained units. You should design methods separately from the incidental details of other methods of the class and separately from any program that uses the class. One incidental detail is the name of a method's parameters. Fortunately, parameters behave like local variables, and so their meanings are confined to their respective method definitions. Thus, you can choose the parameter names without any concern that they will be the same as some

other identifier used in some other method. For team programming projects, one programmer can write a method definition while another programmer writes another part of the program that uses that method. The two programmers need not agree on the names they use for parameters or local variables. They can choose their identifier names completely independently, without any concern that some, all, or none of their identifiers might be the same.

Arguments and Parameters

C.12 Earlier you saw that an object of a class usually receives a call to a method defined within that class. For example, you saw that the statements

```
Name joe = new Name();
joe.setFirst("Joseph");
joe.setLast("Brown");
```

set the first and last names for the object `joe`. The strings `"Joseph"` and `"Brown"` are the arguments. These arguments must correspond to the parameters of the method definition. In the case of `setFirst`, for example, the parameter is the string `firstName`. The argument is the string `"Joseph"`. The argument is plugged in for the corresponding parameter. Thus, in the body of the method, `firstName` represents the string `"Joseph"` and behaves like a local variable.

A method invocation must provide exactly as many arguments as there are parameters in the corresponding method definition. In addition, the arguments in the invocation must correspond to the parameters in the method's definition with respect to both the order in which they occur and their data types. In some cases, however, Java will perform an automatic type conversion when the data types do not match.

Java does have a notation for parameters that allows a variable number of arguments. Since we really do not need this feature, we will not cover it.

Note: The arguments in the invocation of a method must correspond to the parameters in the method's definition with respect to number, order, and data type.

Aside: Use of the terms "parameter" and "argument"

The use of the terms "parameter" and "argument" in this book is consistent with common usage, but some people use the terms "parameter" and "argument" interchangeably. Some people use the term "parameter" for both what we call parameters and what we call arguments. Other people use the term "argument" for both what we call parameters and what we call arguments.

Passing Arguments

C.13 When a parameter has a primitive type, such as `int` or `char`, the parameter is initialized to the value of the corresponding argument in the method invocation. The argument in a method invocation can be a literal constant—like 2 or `'A'`—or it can be a variable or any expression that yields a value of the appropriate type. Note that the method cannot change the value of an argument

that has a primitive data type. Such an argument serves as an input value only. This mechanism is described as **call-by-value**.

For example, suppose that the class `Name` provided for a middle initial by defining another data field and the method `setMiddleInitial`. Thus, the class might appear as follows:

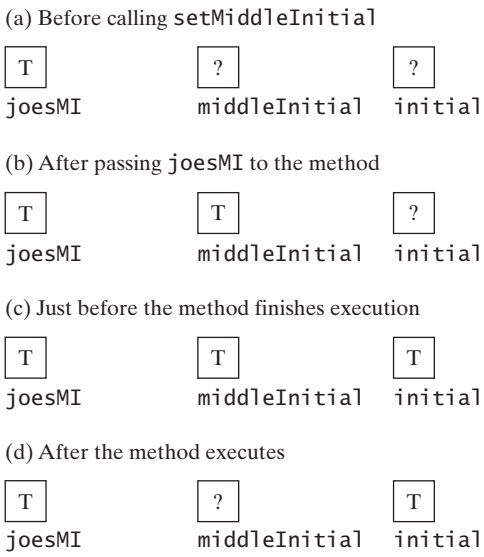
```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial

   . . .
```

A client of this class could contain the following statements:

```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```

Figure C-4 shows the argument `joesMI`, the parameter `middleInitial`, and the data field `initial` as the method `setMiddleInitial` executes. (Although the data field has an initial value, it is not relevant, so the figure shows it as a question mark.)

FIGURE C-4 The effect of executing the method `setMiddleInitial` on its argument `joesMI`, its parameter `middleInitial`, and the data field `initial`



If a method changes the value of its parameter, the corresponding argument will be unaffected. So if, for example, `setMiddleInitial` contained the statements

```
initial = middleInitial;
middleInitial = 'X'
```

the value of `middleInitial` in Figure C-4c would be *X*, but the rest of the figure would not change. In particular, the value of `joesMI` would not change.

C.14 When a parameter has a class type, the corresponding argument in the method invocation must be an object of that class type. The parameter is initialized to the memory address of that object.[1] Thus, the parameter will serve as an alternative name for the object. This implies that the method can change the data in the object, if the class has mutator methods. The method, however, cannot replace an object that is an argument with another object.

For example, if you adopt a child, you might give that child your last name. Suppose that you add the following method `giveLastNameTo` to the class `Name` that makes this change of name:

```
public void giveLastNameTo(Name child)
{
   child.setLast(last);
} // end giveLastNameTo
```

Notice that the parameter of this method has the type `Name`.

Now if Jamie Jones adopts Jane Doe, the following statements would change Jane's last name to Jones:

```
public static void main(String[] args)[2]
{
   Name jamie = new Name();
   jamie.setFirst("Jamie");
   jamie.setLast("Jones");

   Name jane = new Name();
   jane.setFirst("Jane");
   jane.setLast("Doe");

   jamie.giveLastNameTo(jane);
   . . .
} // end main
```

Figure C-5 shows the argument `jane` and the parameter `child` as the method `giveLastNameTo` executes.

C.15 What happens if you change the method definition to allocate a new name, as follows?

```
public void giveLastNameTo2(Name child)
{
   String firstName = child.getFirst();
   child = new Name();
   child.setFirst(firstName);
   child.setLast(last);
} // end giveLastNameTo2
```

With this change, the invoking statement

```
jamie.giveLastNameTo2(jane);
```

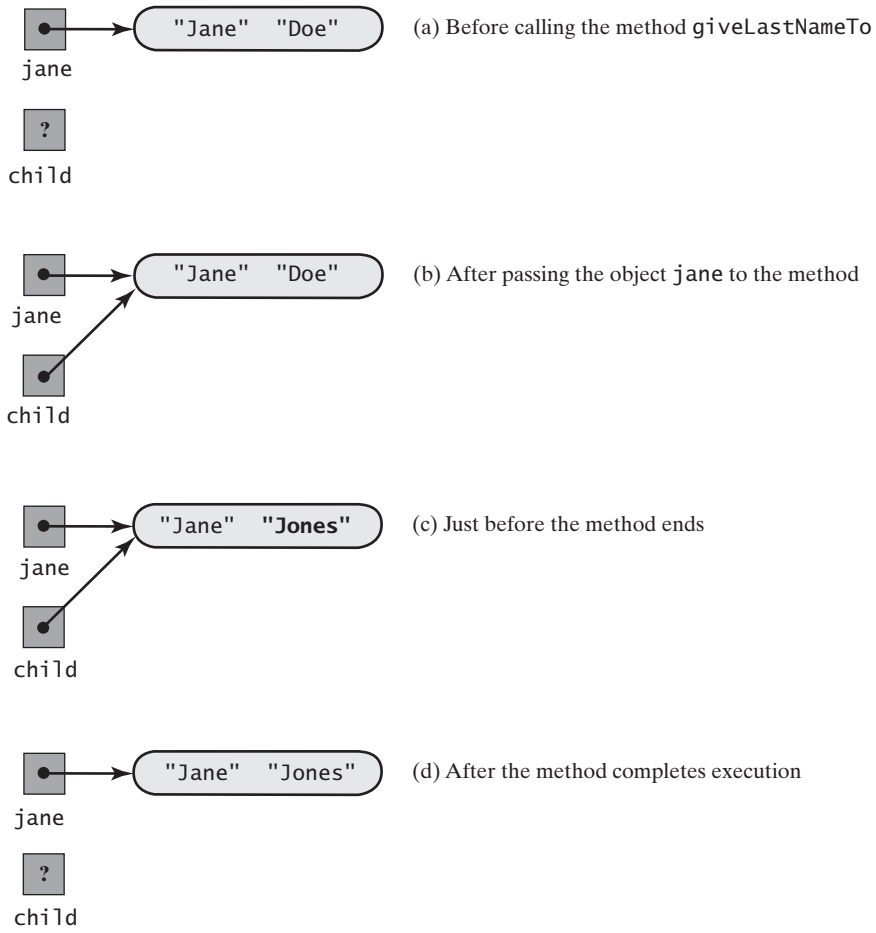
has no effect on `jane`, as Figure C-6 illustrates. The parameter `child` behaves like a local variable, so its value is not available outside of the method definition.

---

1. The parameter mechanism for parameters of a class type is similar to **call-by-reference** parameter passing. If you are familiar with this terminology, be aware that parameters of a class type in Java behave a bit differently from call-by-reference parameters in other languages.

2. If you are not familiar with `main` methods and application programs, consult the beginning of Appendix B.

FIGURE C-5 The method `giveLastNameTo` modifies the object passed to it as an argument



Question 8 Consider a method definition that begins with the statement

```
public void process(int number, Name aName)
```

If `jamie` is defined as in Segment C.14 and you invoke this method with the statement

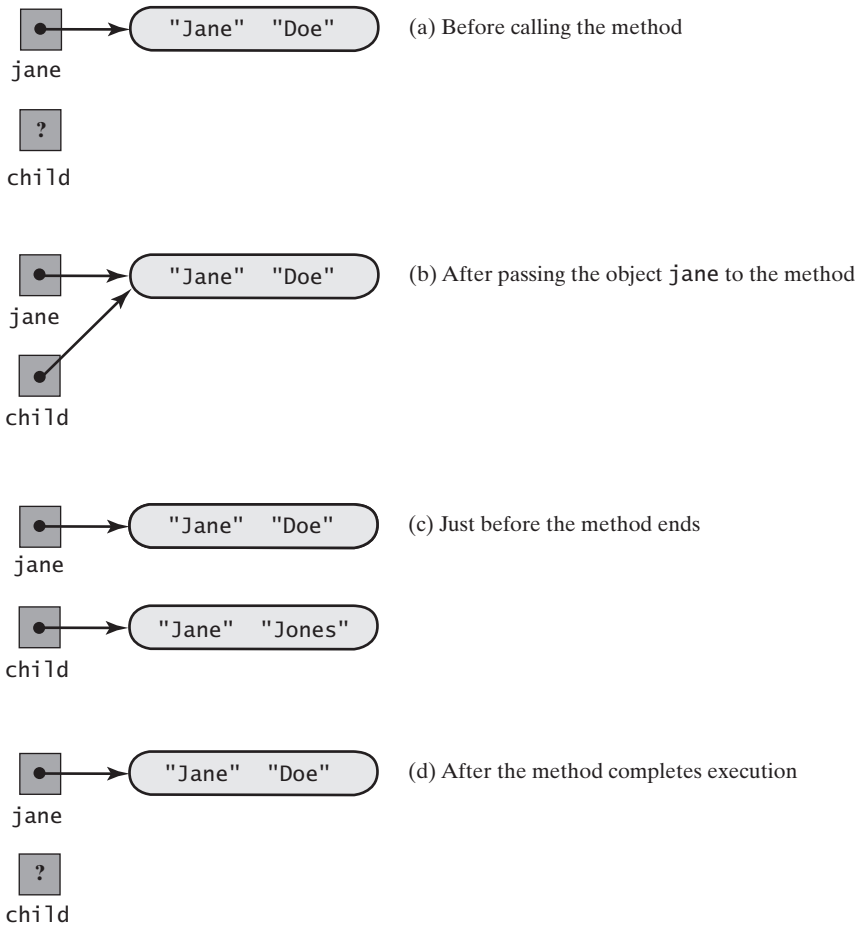
```
someObject.process(5, jamie);
```

what values are given to the parameters within the definition of the method?

Question 9 In Question 8, can the method `process` change the data fields in `jamie`?

Question 10 In Question 8, can the method `process` assign a new object to `jamie`?

FIGURE C-6 A method cannot replace an object passed to it as an argument



A Definition of the Class `Name`

C.16 A complete definition for the class `Name` appears in Listing C-1. We typically place data field declarations at the beginning of the class, but some people place them last. Although Java allows you to intermix method definitions and data field declarations, we prefer that you do not.

The sections that follow examine some other details of this class definition.

LISTING C-1 The class `Name`

```
1  public class Name
2  {
3     private String first; // First name
4     private String last;  // Last name
5
6     public Name()
7     {
8     } // end default constructor
9
```

```
10    public Name(String firstName, String lastName)
11    {
12       first = firstName;
13       last = lastName;
14    } // end constructor
15
16    public void setName(String firstName, String lastName)
17    {
18       setFirst(firstName);
19       setLast(lastName);
20    } // end setName
21
22    public String getName()
23    {
24       return toString();
25    } // end getName
26
27    public void setFirst(String firstName)
28    {
29       first = firstName;
30    } // end setFirst
31
32    public String getFirst()
33    {
34       return first;
35    } // end getFirst
36
37    public void setLast(String lastName)
38    {
39       last = lastName;
40    } // end setLast
41
42    public String getLast()
43    {
44       return last;
45    } // end getLast
46
47    public void giveLastNameTo(Name aName)
48    {
49       aName.setLast(last);
50    } // end giveLastNameTo
51
52    public String toString()
53    {
54       return first + " " + last;
55    } // end toString
56 } // end Name
```

Constructors

C.17 Segment C.2 mentioned that you create an object by using the `new` operator to invoke a special method called a constructor. A **constructor** allocates memory for the object and initializes the data fields. The method definition of a constructor has certain special properties. A constructor

- Has the same name as the class
- Has no return type, not even `void`
- Has any number of parameters, including no parameters

A class can have several constructors that differ in the number or type of parameters.

A constructor without parameters is called the **default constructor**. A class can have only one default constructor. The definition of the default constructor for `Name` is

```
public Name()
{
} // end default constructor
```

This particular default constructor has an empty body, but it need not be empty. It could explicitly initialize the data fields `first` and `last` to values other than the ones Java assigns by default.

For example, we could have defined the constructor as follows:

```
public Name()
{
   first = "";
   last = "";
} // end default constructor
```

Here we initialize the fields `first` and `last` to an empty string. If the constructor had an empty body, it would initialize these fields to `null` by default.

Note: In the absence of any explicit initialization within a constructor, data fields are set to default values: Reference types are `null`, primitive numeric types are zero, and boolean types are false.

Programming Tip: If a class depends on a data field's initial value, its constructor should set these values explicitly. Do not rely on standard default values.

Programming Tip: If a data field has a reference type, initialize it to something other than `null`. Failure to do so might cause a `NullPointerException` when the class is used.

C.18 If you do not define any constructors for a class, Java will automatically provide a default constructor—that is, a constructor with no parameters. If you define a constructor that has parameters but you do not define a default constructor—one without parameters—Java will not provide a default constructor for you. Because classes are often reused again and again, and because eventually you might want to create a new object without specifying parameters, your classes typically should include a default constructor.

Note: Once you start defining constructors, Java will not define any other constructors for you. Most of the classes you define should include a default constructor.

C.19 The class `Name` contains a second constructor, one that initializes the data fields to values given as arguments when the client invokes the constructor:

```
public Name(String firstName, String lastName)
{
   first = firstName;
   last = lastName;
} // end constructor
```

This constructor has two parameters, `firstName` and `lastName`. You invoke it with a statement such as

```
Name jill = new Name("Jill", "Jones");
```

that passes first and last names as arguments.

C.20 After creating the object `jill`, you can change the values of its data fields by using the class's set (mutator) methods. You saw that this step was in fact necessary for the object `joe` in Segments C.2 and C.3, since `joe` was created by the default constructor and had default values—probably `null`— as its first and last names.

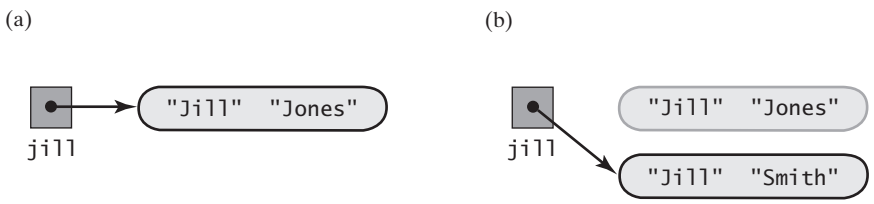
Let's see what would happen in the case of `jill` if you tried to use the constructor to change the values of `jill's` data fields. After you created the object, the variable `jill` contained the memory address of that object, as Figure C-7a illustrates. If you now write the statement

```
jill = new Name("Jill", "Smith");
```

a new object is created, and `jill` contains its memory address. The original object is lost, because no program variable has its address, as shown in Figure C-7b.

What happens to a memory location when the variables in your program no longer reference it? Periodically, the Java run-time environment **deallocates** such memory locations by returning them to the operating system so that they can be used again. In effect, the memory is recycled. This process is called **automatic garbage collection**.

FIGURE C-7 An object (a) after its initial creation; (b) after its reference is lost



Note: Memory leak

If the Java run-time environment did not track and recycle memory that a program no longer references, a program could use all the memory available to it and subsequently fail. If you use another programming language—C++, for example—you would be responsible for returning unneeded memory to the operating system for reuse. A program that failed to return such memory would have what is known as a **memory leak**. Java programs do not have this problem.

Note: Classes without mutator methods

After you create an object of a class that has no set methods, you cannot change the values of its data fields. If they require change, you must use the constructor to create a new object. Java Interlude 6 discusses these classes further.

Question 11 What is a default constructor?

Question 12 How do you invoke a constructor?

Question 13 What happens if you do not define constructors for a class?

Question 14 What happens if you do not define a default constructor but you do define a constructor that has parameters?

Question 15 What happens when an object no longer has a variable that references it?

The Method `toString`

C.21 The method `toString` in the class `Name` returns a string that is the person's full name. You can use this method, for example, to display the name that the object `jill` represents by writing

```
System.out.println(jill.toString());
```

What is remarkable about `toString` is that Java will invoke it automatically when you write

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

For this reason, providing a class with a method `toString` is a good idea in general. If you fail to do so, Java will provide its own `toString` method, which produces a string that will have little meaning to you. Appendix D provides more detail about the `toString` method.

Methods That Call Other Methods

C.22 Notice the method `setName` in the class definition for `Name`. Although `setName` could use assignment statements to initialize `first` and `last`, it instead invokes the methods `setFirst` and `setLast`. Since these methods are members of the class, `setName` can invoke them without preceding the name with an object variable and a dot. If you prefer, you can use `this`, and write the invocation as

```
this.setFirst(firstName);
```

When the logic of a method's definition is complex, you should divide the logic into smaller pieces and implement each piece as a separate method. Your method can then invoke these other methods. Such helping methods, however, might be inappropriate for a client to use. If so, declare them as private instead of public so that only your class can invoke them.

Programming Tip: If a helping method is not appropriate for public use, declare it as private.

C.23 The method `getName` in the class `Name` also invokes another of `Name`'s methods, namely `toString`. Here, we want both `getName` and `toString` to return the same string. Rather than writing the same statements in both methods, we have one method call the other. This ensures that both methods will always return the same values. If you later revise the definition of `toString`, you will automatically revise the string that `getName` returns.

Programming Tip: If you want two methods to have the same behavior, one of them should call the other.

C.24 Although it generally is a good idea for methods to call other methods to avoid repeating code, you need to be careful if you call public methods from the body of a constructor. For example, it is tempting to have the constructor mentioned in Segment C.19 call `setName`. But another class derived from your class could change the effect of `setName` and hence of your constructor. One solution is to define a private method that both the constructor and `setName` call. Another approach is given in Segment D.18 of the next appendix.

C.25 **Using `this` to invoke a constructor.** You can use the reserved word `this` to call a constructor from within the body of another constructor. For example, the class `Name` has two constructors. The default constructor, as given in Segment C.16, has an empty body. Segment C.17 suggested that it is a good idea to have the default constructor initialize the class's data fields explicitly, so we rewrote it, as follows:

```
public Name()
{
   first = "";
   last = "";
} // end default constructor
```

We could accomplish the same thing by revising the default constructor so that it initializes `first` and `last` by calling the second constructor, as follows:

```
public Name()
{
   this("", "");
} // end default constructor
```

The statement

```
this("", "");
```

calls the constructor that has two parameters. In this way, the initialization occurs in one place.

Programming Tip: Link the definitions of several constructors by using `this` to invoke one of them. Any use of `this` must be first in the body of the constructor's definition.

Question 16 A third constructor for the class `Name` could have the following header:

```
public Name(Name aName)
```

This constructor creates a `Name` object whose data fields match those of the object `aName`. Implement this new constructor by invoking one of the existing constructors.

Methods That Return an Instance of Their Class

C.26 The method `setName` in the class `Name` is a void method that sets both the first and last names of a `Name` object. We might use this method as follows:

```
Name jill = new Name();
jill.setName("Jill", "Greene");
```

Here, `setName` sets the first and last names of the receiving object `jill`.

Instead of defining `setName` as a void method, we could have it return a reference to the revised instance of `Name`, as follows:

```
public Name setName(String firstName, String lastName)
{
   setFirst(firstName);
   setLast(lastName);

   return this;
} // end setName
```

Here `this` represents the receiving object, whose first and last names were just set.

We can call this definition of `setName` just as we called its void version, or we could invoke it as follows:

```
Name jill = new Name();
Name myFriend = jill.setName("Jill", "Greene");
```

As before, `setName` sets the first and last names of the receiving object `jill`. But then it returns a reference to the receiving object. Here we used an assignment statement to retain this reference as an alias for `jill`. However, the invocation of `setName` could appear as an argument to another method.

Methods that return an instance of their class are not unusual within the classes of the Java Class Library.

Static Fields and Methods

C.27 **Static fields.** Sometimes you need a data field that does not belong to any one object. For example, a class could track how many invocations of the class's methods are made by all objects of the class. Such a data field is called a **static field**, **static variable**, or **class variable**. You declare a static field by adding the reserved word `static`. For example, the declaration

```
private static int numberOfInvocations = 0;
```

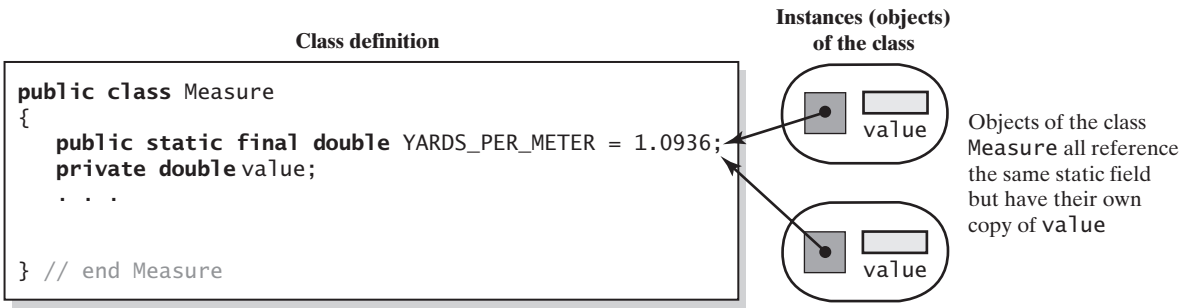
defines one copy of `numberOfInvocations` that every object of the class can access. Objects can use a static field to communicate with each other or to perform some joint action. In this example, each method increments `numberOfInvocations`. Such static fields normally should be private to ensure that access occurs only through appropriate accessor and mutator methods.

The definition of a named constant provides another example of a static field. The statement

```
public static final double YARDS_PER_METER = 1.0936;
```

defines a static field `YARDS_PER_METER`. The class has one copy of `YARDS_PER_METER`, rather than each object of the class having its own copy, as Figure C-8 illustrates. Since `YARDS_PER_METER` is also declared as final, its value cannot change, so we can safely make it public. But static fields in general can change value if you omit the modifier `final`.

FIGURE C-8 A static field `YARDS_PER_METER` versus a nonstatic field `value`



Note: Static does not mean constant

A static field is shared by all objects of its class, but its value can change. If you want a field's value to remain constant, you must declare it as final. Although they often appear together, the modifiers `static` and `final` are not related. Thus, a field can be static, final, or both static and final.

C.28 **Static methods.** Sometimes you need a method that does not belong to an object of any kind. For example, you might need a method to compute the maximum of two integers or a method to compute the square root of a number. These methods have no obvious object to which they should belong. In these cases, you can define the method as static by adding the reserved word `static` to the header of the method.

A **static method** or **class method** is still a member of a class. However, you use the class name instead of an object name to invoke the method. For example, Java's predefined class `Math` contains several standard mathematical methods, such as `max` and `sqrt`. All of these methods are static, so you do not need—and in fact have no real use for—an object of the class `Math`. You call these methods by using the class name in place of an object name. Thus, you write statements such as

```
int maximum = Math.max(2, 3);
double root = Math.sqrt(4.2);
```

The definition of a static method cannot reference any data field in its class that is not static. It can, however, reference its class's static fields. Likewise, it cannot invoke a nonstatic method of the class, unless it creates a local object of the class and uses it to invoke the nonstatic method. However, a static method can call other static methods within its class. Since every application program's `main` method is static, these restrictions apply to `main` methods.

Programming Tip: Every class can have a `main` method

You can include a test of a class as a `main` method in the class's definition. Anytime you suspect something is wrong, you can easily test the class definition. Since you—and others—can see what tests you performed, flaws in your testing will become apparent. If you use the class as a program, the `main` method is invoked. When you use the class to create objects in another class or program, the `main` method is ignored.

Note: Constructors cannot be static

A constructor creates an object of its class, so it makes no sense to disassociate a constructor from such objects by making it static.

Question 17 What happens if you do not declare a constant data field as static?

Overloading Methods

C.29 Several methods within the same class can have the same name, as long as the methods do not have identical parameters. Java is able to distinguish among these methods since their parameters differ in number or data type. We say that these methods are **overloaded**.

For example, the class `Name` has the method `setName` whose header is

```
public void setName(String firstName, String lastName)
```

Imagine that we want another method that gives a `Name` object the same first name and last name as another `Name` object. The header for this method could be, for example,

```
public void setName(Name otherName)
```

The two versions of `setName` are not exactly the same, as they have different numbers of parameters.

We could then overload `setName` with a third method whose header is

```
public void setName(String firstName, Name otherName)
```

Imagine that this method sets the first name to the string `firstName` and the last name to `otherName`'s last name. Although two of the three versions of `setName` have two parameters each, the data types of the parameters are not exactly the same. The data types of the first parameter in each method match, but the data types of the second parameters do not.

Simply changing the names of the parameters is not enough to overload a method. The parameter names are not relevant. When two methods have the same name and the same number of parameters, at least one pair of corresponding parameters must differ in data type. Also, changing only the return type is insufficient. The compiler cannot distinguish between two methods that differ only in their return types.

Finally, note that by defining more than one constructor for a class, you are actually overloading them. Thus, their parameters must differ in either number or data type.

Note: Overloading a method definition
A method in a class overloads another method in the same class when both methods have the same name but differ in the number or types of parameters.

Note: The signature of a method
A method's **signature** consists of its name and the number, types, and order of its parameters. Thus, overloaded methods have the same name but different signatures.

Question 18 If a method overloads another method, can the two methods have different return types?

Enumeration as a Class

Appendix B introduced enumerations. As Segments B.54 and B.56 mentioned, the compiler creates a class when it encounters an enumeration. This section expands that discussion. Although you should consider using enumerations in your programs, they are not central to the presentation in this book.

C.30 When you define an enumeration, the class created has methods such as `toString`, `equals`, `ordinal`, and `valueOf`. For example, let's define a simple enumeration for the suits of playing cards, as follows:

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

We then can use these methods in the following ways:

- `Suit.CLUBS.toString()` returns the string *CLUBS*. That is, `toString` returns the name of its receiving object.
- `System.out.println(Suit.CLUBS)` calls `toString` implicitly, and so it displays *CLUBS*.
- `s.equals(Suit.DIAMONDS)` tests whether `s`, an instance of `Suit`, equals `DIAMONDS`.
- `Suit.HEARTS.ordinal()` returns 2, the ordinal position of `HEARTS` in the enumeration.
- `Suit.valueOf("HEARTS")` returns `Suit.HEARTS`.

C.31 You can define additional methods—including constructors—for any enumeration. By defining a private data field, you can assign values to each of the objects in the enumeration. Adding a get method will provide a way for the client to access these values. Listing C-2 contains a new definition for the enumeration `Suit` that shows how these ideas are realized.

LISTING C-2 The enumeration `Suit`

```
1  /** An enumeration of card suits. */
2  enum Suit
3  {
4     CLUBS("black"), DIAMONDS("red"), HEARTS("red"), SPADES("black");
5
6     private final String color;
7
8     private Suit(String suitColor)
9     {
10       color = suitColor;
11    } // end constructor
12
13    public String getColor()
14    {
15       return color;
16    } // end getColor
17 } // end Suit
```

We have chosen strings as the values for the enumerated objects. We set the value of `CLUBS`, for example, by writing `CLUBS("black")`. This notation invokes the constructor that we have provided and sets the value of `CLUBS`'s private data field `color` to the string *black*. Note that `color`'s value cannot change, since it is declared as final. Also observe that the constructor is private, so it is not available to the client. It is called only within the definition of `Suit`. The method `getColor` provides public access to the value of `color`.

Note: Constructors within enumerations must be private.

C.32 The class in Listing C-3 provides a simple demonstration of the enumeration `Suit` that appears in the previous segment. We use a for-each loop, described in Segment B.63 of Appendix B. In addition to the methods that we defined in `Suit`, the enumeration also has the methods `equals`, `ordinal`, and `valueOf` described in Segment C.30 of this appendix and Segment B.56 of Appendix B.

LISTING C-3 A class that demonstrates the enumeration `Suit` given in Listing C-2 of Segment C.31

```
1  /** A demonstration of the enumeration Suit. */
2  public class SuitDemo
3  {
4     private enum Suit
5     {
6        . . . < See Listing C-2>
7     } // end Suit
8
```

```
 9    public static void main(String[] args)
10    {
11       for (Suit nextSuit : Suit.values())
12       {
13          System.out.println(nextSuit + " are " + nextSuit.getColor() +
14                             " and have an ordinal value of " +
15                             nextSuit.ordinal());
16       } // end for
17    } // end main
18 } // end SuitDemo
```

Output

```
CLUBS are black and have an ordinal value of 0
DIAMONDS are red and have an ordinal value of 1
HEARTS are red and have an ordinal value of 2
SPADES are black and have an ordinal value of 3
```

Note: Enumerations can have an access modifier such as `public` or `private`. If you omit the access modifier, the enumeration is private. You can define a public enumeration within its own file, just as you would define any other public class.

C.33 

Example. Segment B.54 in Appendix B defined an enumeration for the letter grades A, B, C, D, and F. Here we expand that definition to include plus and minus grades as well as the quality-point values associated with each grade. As in the previous definition of `Suit`, we provide private data fields and a private constructor to represent and initialize the string representation and numeric value for each grade. We also provide accessor methods for the data fields and override the method `toString`.

Listing C-4 shows our new definition of the enumeration `LetterGrade`. We have made it public and will store it in the file `LetterGrade.java`.

LISTING C-4 The enumeration LetterGrade

```
1  public enum LetterGrade
2  {
3     A("A", 4.0), A_MINUS("A-", 3.7), B_PLUS("B+", 3.3), B("B", 3.0),
4     B_MINUS("B-", 2.7), C_PLUS("C+", 2.3), C("C", 2.0), C_MINUS("C-", 1.7),
5     D_PLUS("D+", 1.3), D("D", 1.0), F("F", 0.0);
6
7     private final String grade;
8     private final double points;
9
10    private LetterGrade(String letterGrade, double qualityPoints)
11    {
12       grade = letterGrade;
13       points = qualityPoints;
14    } // end constructor
15
16    public String getGrade()
17    {
18       return grade;
19    } // end getGrade
20
```

```
21    public double getQualityPoints()
22    {
23       return points;
24    } // end getQualityPoints
25
26    public String toString()
27    {
28       return getGrade();
29    } // end toString
30 } // end LetterGrade
```

If we define

```
LetterGrade myGrade = LetterGrade.B_PLUS;
```

then

- `myGrade.toString()` returns the string *B*+.
- `System.out.println(myGrade)` displays *B*+, since it calls `toString` implicitly.
- `myGrade.getGrade()` returns the string *B*+.
- `myGrade.getQualityPoints()` returns 3.3.

If we had not overriden the method `toString` with our own definition, `myGrade.toString()` would return the string *B_PLUS*.

Like the enumeration `Suit` given in Segment C.31, `LetterGrade` has the methods `equals`, `ordinal`, and `valueOf`.

Question 19 If `myGrade` is an instance of `LetterGrade` and has the value `LetterGrade.B_PLUS`, what is returned by each of the following expressions?

a. `myGrade.ordinal()`
b. `myGrade.equals(LetterGrade.A_MINUS)`
c. `LetterGrade.valueOf("A_MINUS")`

Question 20 What does the following statement display?

```
System.out.println(LetterGrade.valueOf("A_MINUS"));
```

Packages

C.34 Using several related classes is more convenient if you group them together within a Java **package**. To identify a class as part of a particular package, you begin the file that contains the class with a statement like

```
package myStuff;
```

You then place all of the files within one directory or folder and give it the same name as the package.

To use a package in your program, you begin the program with a statement such as

```
import myStuff.*;
```

The asterisk makes all public classes within the package available to the program. You could, however, replace the asterisk with the name of a particular class in the package that you want to use. You probably have already used packages provided by Java, such as the package `java.util`.

Why did we just say "public classes?" What other kind of class is there? You can use an access modifier to control access to a class just as you can to control access to a data field or method. A public class—whether it is within a package or not—is available to any other class. If you omit the class's access modifier entirely, the class is available only to other classes within the same package. This kind of class is said to have **package access**. Similarly, if you omit the access modifier on data fields or methods, they are available by name inside the definition of any class within the same package but not outside of the package. You can use package access in situations where you have a package of cooperating classes that act as a single encapsulated unit. If you control the package directory, you control who is allowed to access the package.

The Java Class Library

C.35 Java comes with a collection of many classes that you can use in your programs. For example, Segment C.28 mentioned the class `Math`, which contains several standard mathematical methods such as `sqrt`. This collection of classes is known as the **Java Class Library**, and sometimes as the **Java Application Programming Interface**, or **API**. The classes in this library are organized into several standard packages. For example, the class `Math` is a part of the package `java.lang`. Note that no `import` statement is needed when you use a class from this particular package.

You should become familiar with the documentation provided for the Java Class Library at `docs.oracle.com/javase/8/docs/api/`.

Answers to Self-Test Questions

1.
```
Name myName = new Name();
myName.setFirst("Joseph");
myName.setLast("Brown");
```

2.
```
System.out.println(myName.getLast() + ", " + myName.getFirst());
```

3. Valued: `getFuelLevel`, `getSpeed`, `getMileage`.
Void: `goForward`, `goBackward`, `accelerate`, `decelerate`.

4. Mutator.

5. Valued.

6. Void.

7. A client could set the data field to an illegal value.

8. `number` is 5 and `aName` references `jamie`.

9. Yes, by using `Name`'s set methods.

10. No.

11. A default constructor is a constructor that has no parameters.

12. You invoke a constructor by using the `new` operator.

13. If you do not define a constructor, the compiler defines a default constructor.

14. The compiler does not define additional constructors. Thus, the class will not have a default constructor.

15. An object that is not referenced is marked for garbage collection. Eventually, the Java run-time environment deallocates the object by returning its memory locations to the operating system so that they can be used again.

16.
```
public Name(Name aName)
{
   this(aName.getFirst(), aName.getLast());
} // end constructor
```

17. Each instance (object) of a class will have a copy of a constant data field that is not static.

18. Yes, as long as the methods also have different parameters. Return type alone is not sufficient to distinguish the methods.

19. **a.** 2
b. false
c. `LetterGrade.A_MINUS`

20. A-, because the method `toString` is called implicitly.