

Chapter 8

Classes

8.1 Object-Oriented Programming

- Classes and Objects
- Point Objects

8.2 Object State and Behavior

- Object State: Fields
- Object Behavior: Methods
- The Implicit Parameter
- Mutators and Accessors
- The toString Method

8.3 Object Initialization: Constructors

- The Keyword this
- Multiple Constructors

8.4 Encapsulation

- Private Fields
- Class Invariants
- Changing Internal Implementations

8.5 Case Study: Designing a Stock Class

- Object-Oriented Design Heuristics
- Stock Fields and Method Headers
- Stock Method and Constructor Implementation

Introduction

Now that you've mastered the basics of procedural-style programming in Java, you're finally ready to explore what Java was designed for: object-oriented programming. This chapter introduces the basic terminology that you should use to talk about objects and shows you how to declare your own classes to create your own objects.

Objects are entities that contain state and behavior and that can be used as parts of larger programs. We'll discuss the concepts of abstraction and encapsulation, which allow you to use objects at a high level without understanding their inner details. We'll also discuss ideas for designing new classes of objects and implementing the programs that utilize them.

8.1 Object-Oriented Programming

Most of our focus so far has been on procedural decomposition, the technique of breaking complex tasks into smaller subtasks. This is the oldest style of programming, and even in a language like Java we still use procedural techniques. But Java also provides a different approach to programming that we call *object-oriented programming*.

Object-Oriented Programming (OOP)

Reasoning about a program as a set of objects rather than as a set of actions.

Object-oriented programming involves a particular view of programming that has its own terminology. Let's explore that terminology with nonprogramming examples first. Recall the definition of *object* from Chapter 3.

Object

A programming entity that contains state (data) and behavior (methods).

To truly understand this definition, you have to understand the terms “state” and “behavior.” These are some of the most fundamental concepts in object-oriented programming.

Let's consider the class of objects we call radios. A radio can be in different states. It can be turned on or turned off. It can be tuned to one of many different stations, and it can be set to one of many different volumes. Any given radio has to “know” what state it is in, which means that it has to keep track of this information internally. We call the collection of such internal values the *state* of an object.

State

A set of values (internal data) stored in an object.

What are the behaviors of a radio? The most obvious one is that it produces sound when it is turned on and the volume is turned up. But there are actions that you can perform on a radio that manipulate its internal state. We can turn a radio on or off, and we can change the station or volume. We can also check what station the radio is set to right now. We call the collection of these operations the *behavior* of an object.

Behavior

A set of actions an object can perform, often reporting or modifying its internal state.

Objects themselves are not complete programs; they are components that are given distinct roles and responsibilities. Objects can be used as part of larger programs to solve problems. The pieces of code that create and use objects are known as *clients*.

Client (or Client Code)

Code that interacts with a class or objects of that class.

Client programs interact with objects by sending messages to them and asking them to perform behaviors. A major benefit of objects is that they provide reusable pieces of code that can be used in many client programs. You've already used several interesting objects, such as those of type `String`, `Scanner`, `Random`, and `File`. In other words, you and your programs have been clients of these objects. Java's class libraries contain thousands of existing classes of objects.

As you write larger programs, however, you'll find that Java doesn't always have a pre-existing object for the problem you're solving. For example, if you were creating a calendar application, you might want to use objects to represent dates, contacts, and appointments. If you were creating a three-dimensional graphical simulation, you might want objects to represent three-dimensional points, vectors, and matrices. If you were writing a financial program, you might want classes to represent your various assets, transactions, and expenses. In this chapter you'll learn how to create your own classes of objects that can be used by client programs like these.

Our definition of object-oriented programming is somewhat simplified. A full exploration of this programming paradigm includes other advanced concepts called polymorphism and inheritance that will be discussed in the next chapter.

Classes and Objects

In the previous chapters, we've considered the words "class" and "program" to be roughly synonymous. We wrote programs by creating new classes and placing static main methods into them.

But classes have another use in Java: to serve as blueprints for new types of objects. To create a new type of object in Java, we must create a class and add code to it that specifies the following elements:

- The state stored in each object
- The behavior each object can perform
- How to construct objects of that type

Once we have written the appropriate code, we can use the class to create objects of its type. We can then use those objects in our client programs. We say that the created objects are *instances* of the class because one class can be used to construct many objects. This is similar to the way that a blueprint works: One blueprint can be used to create many similar houses, each of which is an instance of the original blueprint.

Did You Know?**Operating Systems History and Objects**

In 1983 the IBM PC and its “clones” dominated the PC market, and most people ran an operating system called DOS. DOS uses what we call a “command-line interface,” in which the user types commands at a prompt. The console window is a similar interface. To delete a file in DOS, for example, you would give the command “del” (short for “delete”) followed by the file name:

```
del data.txt
```

This interface can be described in simple terms as “verb noun.” In fact, if you look at a DOS manual, you will find that it is full of verbs. This structure closely parallels the procedural approach to programming. When we want to accomplish some task, we issue a command (the verb) and then mention the object of the action (the noun, the thing we want to affect).

In 1984, Apple Computer released a new computer called a Macintosh that used what we call a graphical user interface, or GUI. The GUI interface uses a graphical “desktop” metaphor that has become so well known that people now tend to forget it is a metaphor. Later, Microsoft brought this functionality to IBM PCs with its Windows operating system.

To delete a file on a Macintosh or on a Windows machine, you locate the icon for the file and click on it. Then you have several options. You can drag it to the trash/recycling bin, or you can select a “delete” command from the menu. Either way, you start with the object you want to delete and then give the command you want to perform. This is a reversal of the fundamental paradigm: With a GUI it’s “noun verb.” This different method of interaction is the core of object-oriented programming.

Most modern programs use GUIs because we have learned that people find it more natural to work this way. We are used to pointing at things, picking up things, grabbing things. Starting with the object is very natural for us. This approach has also proved to be a helpful way to structure our programs, enabling us to divide our programs into different objects that each can do a certain task, rather than dividing up the central task into subtasks.



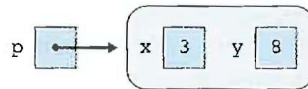
Point Objects

To learn about objects we will first examine an existing Java class, and then we will implement our own version of that class from scratch. The `java.awt` package has a class named `Point`. A `Point` object stores the (x, y) coordinates of a position in two-dimensional space. These coordinates are expressed as integers, although there are also variations for storing points using floating-point numbers. `Point` objects are useful for applications that store many two-dimensional locations, such as maps of cities, graphical animations, and games.

Like most objects, `Point` objects have to be explicitly constructed by calling a constructor. To construct a specific `Point` object, you have to pass the values you want for x and y :

```
Point p = new Point(3, 8);
```

After the program executes the previous line of code, you have the following situation:



Once you have constructed a `Point` object, what can you do with it? One of the most common things you do with an object is print it to the console. A `Point` object, like many Java objects, can be printed with the `println` statement.

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

The `println` statement produces the following output. The format is a little ugly, but it lets you see the x and y values inside a given `Point`.

```
java.awt.Point[x=3,y=8]
```

`Point` objects also have a method called `translate` that can be used to shift the coordinates by a specific delta- x and delta- y , which are passed as parameters. When you translate a `Point`, you shift its location by the specified amount. For example, you might say:

```
p.translate(-1, -2); // subtract 1 from x, subtract 2 from y
```

Given that the `Point` started out with coordinates $(3, 8)$, this translation would leave the `Point` with coordinates $(2, 6)$. Thus, after this line of code is executed, you'd end up with the following situation:

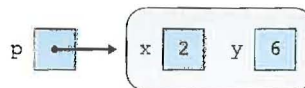


Table 8.1 Useful Methods of Point Objects

Method	Description
<code>translate(dx, dy)</code>	Translates the coordinates by the given amounts
<code>setLocation(x, y)</code>	Sets the coordinates to the given values
<code>distance(p2)</code>	Returns the distance from this point to p2

One of the other things you can do with a `Point` object is to refer to its `x` and `y` values using the dot notation:

```
int sum = p.x + p.y;
System.out.println("Sum of coordinates = " + sum);
```

You can even change these internal values directly:

```
p.x = 12;
p.y = 15;
```

Table 8.1 includes some useful methods of each `Point` object.

Here is a complete program that constructs a `Point` object and translates its coordinates, using `println` statements to examine the coordinates before and after the call:

```
1 import java.awt.*;
2
3 public class PointExample1 {
4     public static void main(String[] args) {
5         Point p = new Point(3, 8);
6         System.out.println("initially p = " + p);
7         p.translate(-1, -2);
8         System.out.println("after translating p = " + p);
9     }
10 }
```

This code produces the following output:

```
initially p = java.awt.Point[x=3,y=8]
after translating p = java.awt.Point[x=2,y=6]
```

8.2 Object State and Behavior



In the next few sections, we'll explore the structure of classes by writing a new class incrementally. We'll write our own version of the `Point` class that was just described.

Here are the main components of a class that we'll see in the sections that follow:

- Fields (the data stored in each object)
- Methods (the behavior each object can execute)
- Constructors (code that initializes an object as it is being constructed with the `new` keyword)
- Encapsulation (protects an object's data from outside access)

We'll focus on these concepts by creating several major versions of the `Point` class. The first version will give us `Point` objects that contain only data. The second version will add behavior to the objects. The third version will allow us to construct `Points` at any initial position. The finished code will encapsulate each `Point` object's internal data to protect it from unwanted outside access. The early versions of the class will be incomplete and will be used to illustrate each feature of a class in isolation. Only the finished version of the `Point` class will be written in proper object-oriented style.

Object State: Fields

The first version of our `Point` class will contain state only. To specify each object's state, we declare special variables inside the class called *fields*. There are many synonyms for "field" that come from other programming languages and environments, such as "instance variable," "data member," and "attribute."

Field

A variable inside an object that makes up part of its internal state.

The syntax for declaring a field is the same as the syntax for declaring normal variables: a type followed by a name and a semicolon. But unlike normal variables, fields are declared directly inside the `{` and `}` braces of your class. When we declare a field, we're saying that we want every object of this class to have that variable inside it.

In previous chapters we've seen that every class should be placed into its own file. The following code, written in the file `Point.java`, defines the first version of our `Point` class:

```
1 // A Point object represents a pair of (x, y) coordinates.
2 // First version: state only.
3
4 public class Point {
5     int x;
6     int y;
7 }
```

This code specifies that each `Point` object will contain two fields (an integer called `x` and an integer called `y`). It may look as though the code declares a pair of

int variables, x and y. But actually it indicates that *each* Point object will contain two int variables inside it, called x and y. If we create 100 Point objects, we'll have 100 pairs of x and y fields, one in each instance of the class.

The Point class isn't itself an executable Java program; it simply defines a new class of objects for client programs to use. The client code that uses Point will be a separate class that we will store in a separate file. Client programs can create Point objects using the new keyword and empty parentheses:

```
Point origin = new Point();
```

When a Point object is constructed, its fields are given default initial values of 0, so a new Point object always begins at the origin of (0, 0) unless you change its x or y value. This is another example of auto-initialization, similar to the way that array elements are automatically given default values.

The following lines of code form the first version of a client program that uses our Point class (the code is saved in a file called PointMain.java, which should be in the same folder or project as Point.java in order for the program to compile successfully):

```
1 // A program that deals with points.
2 // First version, to accompany Point class with state only.
3
4 public class PointMain {
5     public static void main(String[] args) {
6         // create two Point objects
7         Point p1 = new Point();
8         p1.x = 7;
9         p1.y = 2;
10
11         Point p2 = new Point();
12         p2.x = 4;
13         p2.y = 3;
14
15         // print each point and its distance from the origin
16         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
17         double dist1 = Math.sqrt(p1.x * p1.x + p1.y * p1.y);
18         System.out.println("distance from origin = " + dist1);
19
20         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
21         double dist2 = Math.sqrt(p2.x * p2.x + p2.y * p2.y);
22         System.out.println("distance from origin = " + dist2);
23         System.out.println();
24     }
```

```
25         // translate each point to a new location
26         p1.x += 11;
27         p1.y += 6;
28         p2.x += 1;
29         p2.y += 7;
30
31         // print the points again
32         System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
33         System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
34     }
35 }
```

The code produces the following output:

```
p1 is (7, 2)
distance from origin = 7.280109889280518
p2 is (4, 3)
distance from origin = 5.0

p1 is (18, 8)
p2 is (5, 10)
```

The client program has some redundancy that we'll eliminate as we improve our `Point` class in the sections that follow.

Our initial `Point` class essentially serves as a way to group two `int` values into one object. This technique is somewhat useful for the client program, but the client could have been written using primitive `ints` instead. Using `Point` objects is not yet substantially better than using primitive `int` values, because our `Point` objects do not yet have any behavior. An object that contains state, but no behavior, is sometimes called a *record* or *struct*. In the sections that follow, we'll grow our `Point` class from a minimal implementation into a proper Java class.

Object Behavior: Methods

The second version of our `Point` class will contain both state and behavior. Behavior of objects is specified by writing *instance methods*. The instance methods of an object describe the messages to which that object can respond.

Instance Method

A method inside an object that operates on that object.

The objects introduced in previous chapters all contained instance methods that represented their behavior. For example, a `String` object has a `length` method and a `Scanner` object has a `nextInt` method. We think of these methods as being stored

inside the object. As you recall, they use different call syntax than static methods: You write the object's name, then a dot, and then the method's name and parameters. Each object's methods are able to interact with the data stored inside that object.

The preceding client program translates the position of two `Point` objects. It does this by manually adjusting their `x` and `y` values:

```
p1.x += 11; // client code translating a Point
p1.y += 6;
```

Since translating points is a common operation, we should represent it as a method. One option would be to write a static `translate` method in the client code that accepts a `Point`, a `delta-x`, and a `delta-y` as parameters. Its code would look like the following:

```
// a static method to translate a Point;
// not a good choice in this case
public static void translate(Point p, int dx, int dy) {
    p.x += dx;
    p.y += dy;
}
```

A call to the static method would look like the following line of code:

```
translate(p1, 11, 6); // calling a translate static method
```

However, a static method isn't the best way to implement the behavior. The `Point` class is supposed to be reusable so that many client programs can use it. If the `translate` method is placed into our `PointMain` client, other clients won't be able to use it without copying and pasting its code redundantly. Also, one of the biggest benefits of programming with objects is that we can put related data and behavior together. The ability of a `Point` to translate data is closely related to that `Point` object's (`x`, `y`) data, so it is better to specify that each `Point` object will know how to translate itself. We'll do this by writing an instance method in the `Point` class.

We know from experience with objects that you can call an instance method called `translate` using "dot notation":

```
p1.translate(11, 6); // calling a translate instance method
```

Notice that the instance method needs just two parameters: `dx` and `dy`. The client doesn't pass the `Point` as a parameter, because the call begins by indicating which `Point` object it wants to translate (`p1`). In our example, the client is sending a `translate` message to the object to which `p1` refers.

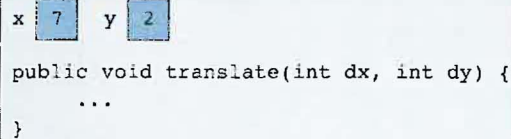
Instance method headers do not have the `static` keyword found in static method headers, but they still include the `public` keyword, the method's return type, its name,

and any parameters that the method accepts. Here's the start of a `Point` class with a `translate` method, with the header declared but the body blank:

```
public class Point {
    int x;
    int y;

    public void translate(int dx, int dy) {
        ...
    }
}
```

When we declare a `translate` method in the `Point` class, we are saying that *each* `Point` object has its own copy of that method. Each `Point` object also has its own `x` and `y` values. A `Point` object would look like the following:



```
public void translate(int dx, int dy) {
    ...
}
```

Whenever an instance method is called, it is called on a particular object. So, when we're writing the body of the `translate` method, we'll think of that code from the perspective of the particular `Point` object that receives the message: "The client has given me a `dx` and `dy` and wants me to change my `x` and `y` values by those amounts." Essentially, we need to write code to match the following pseudocode:

```
public void translate(int dx, int dy) {
    add dx to this Point object's x value.
    add dy to this Point object's y value.
}
```

It's helpful to know that an object's instance methods can refer to its fields. In previous chapters we've talked about *scope*, the range in which a variable can be seen and used. The scope of a variable is the set of braces in which it is declared. The same rule applies to fields: Since they are declared directly inside a class, their scope is the entire class.

This rule means that the `translate` method can directly refer to the fields `x` and `y`. For example, the statement `x += 3;` would increase the `Point` object's `x` value by 3.

Here is a working `translate` method that adjusts the `Point` object's location:

```
public void translate(int dx, int dy) {
    x += dx;
    y += dy;
}
```

The `translate` method can refer to `x` and `y` directly without being more specific about which object it is affecting. It's as though you were riding inside a car and wanted the driver to turn left; you'd simply say, "Turn left." Though there are millions of cars in the world, you wouldn't feel a need to specify which car you meant. It is implied that you mean the car you're currently occupying. Similarly, in instance methods we don't need to specify which object's `x` or `y` we're using, because it is implied that we want to use the fields of the object that receives the message.

Here's the complete `Point` class that contains the `translate` method. The Java style guidelines suggest declaring fields at the top of the class, with methods below, but in general it is legal for a class's contents to appear in any order.

```
public class Point {
    int x;
    int y;

    // shifts this point's location by the given amount
    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```

Here is the general syntax for instance methods:

```
public <type> <name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Methods like `translate` are useful because they give our objects useful behavior that lets us write more expressive and concise client programs. Having the client code manually adjust the `x` and `y` values of `Point` objects to move them is tedious, especially in larger client programs that translate many times. By adding the `translate` method, we have provided a clean way to adjust the location of a `Point` object in a single statement.

The Implicit Parameter

The code for an instance method has an implied knowledge of the object on which it operates. This object is called the *implicit parameter*.

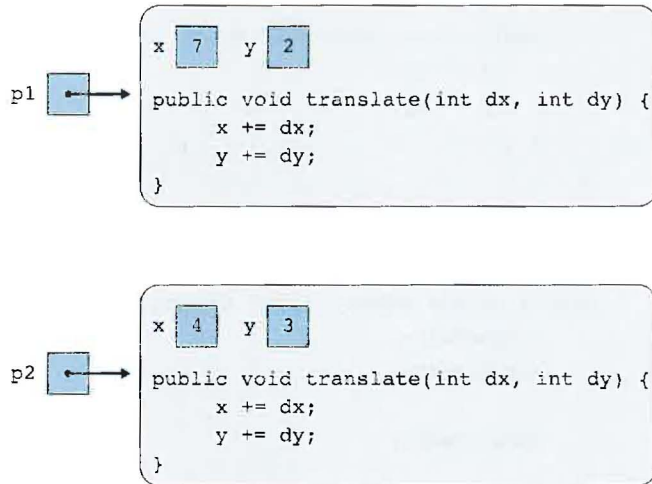
Implicit Parameter

The object that is referenced during an instance method call.

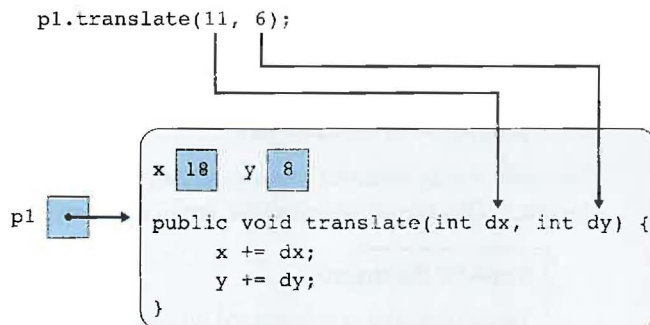
Let's walk through an example to demonstrate exactly how instance methods use the implicit parameter. The following client code constructs two `Point` objects and sets initial locations for them:

```
// construct two Point objects
Point p1 = new Point();
p1.x = 7;
p1.y = 2;
Point p2 = new Point();
p2.x = 4;
p2.y = 3;
```

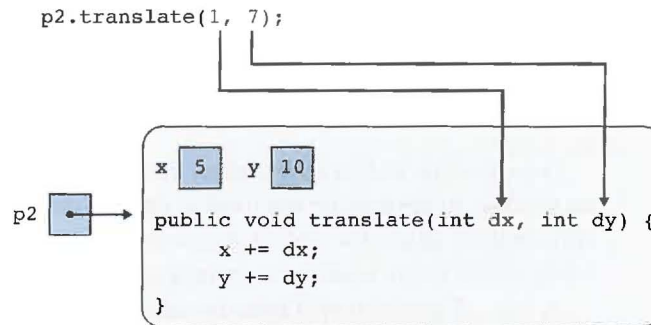
After the preceding code has executed, the variables and objects in memory would appear as follows (remember that each object has its own copy of the `translate` method):



Now we'll call the `translate` method on each object. First, `p1` is translated. During this call, `p1`'s `translate` method is passed the parameters 11 and 6. The implicit parameter here is `p1`'s object, so the statements `x += dx;` and `y += dy;` affect `p1.x` and `p1.y`:



During the second method call `p2`'s `translate` method is executed, so the lines in the body of the `translate` method change `p2.x` and `p2.y`:



Mutators and Accessors

The `translate` method is an example of a *mutator*.

Mutator

An instance method that modifies the object's internal state.

Generally, a mutator assigns a new value to one of the object's fields. Going back to the radio example, the mutators would be the switches and knobs that turn the radio on and off or change the station or volume.

It is a common convention for a mutator method's name to begin with "set," as in `setID` or `setTitle`. Usually, a mutator method has a `void` return type. Mutators often accept parameters that specify the new state of the object or the amount by which to modify the object's current state.

Accessors form a second important category of instance methods.

Accessor

An instance method that provides information about the state of an object without modifying it.

Generally, an accessor returns the value of one of the object's fields. Using our radio analogy, an accessor might return the current station or volume. Examples of accessor methods you have seen in previous chapters include the `length` and `substring` methods of `String` objects and the `exists` method of `File` objects.

Our client program computes the distance of two `Point`s from the origin, (0, 0). Since this is a common operation related to the data in a `Point`, let's give each `Point` object an accessor called `distanceFromOrigin` that computes and returns that `Point` object's distance from the origin. The method accepts no parameters and returns the distance as a `double`.

The distance from the origin is computed using the Pythagorean Theorem, taking the square root of the sum of the squares of the `x` and `y` values. As we did when we

used the `translate` method, we'll refer to the `Point` object's `x` and `y` fields directly in our computation:

```
// returns the distance between this point and (0, 0)
public double distanceFromOrigin() {
    return Math.sqrt(x * x + y * y);
}
```

Note that the `distanceFromOrigin` method doesn't change the `Point` object's `x` or `y` value. Accessors are not used to change the state of the object—they only report information about the object. You can think of accessors as read-only operations while mutators are read/write operations.

A typical accessor will have no parameters and will not have a `void` return type, because it must return a piece of information. An accessor returns a value that is part of the state of the object or is derived from it. The names of many accessors begin with "get" or "is," as in `getBalance` or `isEmpty`.

Here's the complete second version of our `Point` class that now contains both state and behavior:

```
1 // A Point object represents a pair of (x, y) coordinates.
2 // Second version: state and behavior.
3
4 public class Point {
5     int x;
6     int y;
7
8     // returns the distance between this point and (0, 0)
9     public double distanceFromOrigin() {
10         return Math.sqrt(x * x + y * y);
11     }
12
13     // shifts this point's location by the given amount
14     public void translate(int dx, int dy) {
15         x += dx;
16         y += dy;
17     }
18 }
```

The client program can now use the new behavior of the `Point` class. The program produces the same output as before, but it is shorter and more readable than the original. The following lines show examples of the changes made to `PointMain`:

```
System.out.println("distance from origin = " + p1.distanceFromOrigin());
...
p1.translate(11, 6);
```

The toString Method

The designers of Java felt that it was important for all types of values to work well with `Strings`. You've seen that you can concatenate `Strings` with any other type of value, such as primitive `ints` or other objects. Consider the following code:

```
int i = 42;
String s = "hello";
Point p = new Point();

System.out.println("i is " + i);
System.out.println("s is " + s);
System.out.println("p is " + p);
```

Using the `Point` class that we've written so far, the preceding code produces output like the following:

```
i is 42
s is hello
p is Point@119c082
```

Notice that printing `p` generated a strange result. We'd rather have it print the object's state of `(0, 0)`, but Java doesn't know how to do so unless we write a special method in our `Point` class.

When a Java program is printing an object or concatenating it with a `String`, the program calls a special method called `toString` on the object to convert it into a `String`. The `toString` method is an instance method that returns a `String` representation of the object. A `toString` method accepts no parameters and has a `String` return type:

```
public String toString() {
    <code to produce and return the desired string>;
}
```

If you don't write a `toString` method in your class, your class will use a default version that returns the class name followed by an `@` sign and some letters and numbers related to the object's address in memory. If you define your own `toString` method, it replaces this default version.

The following code implements a `toString` method for our `Point` objects and returns a `String` such as `"(0, 0)"`:

```
// returns a String representation of this point
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Now that our class has this method, the preceding client code produces the following output.

```
i is 42  
s is hello  
p is (0, 0)
```

Note that the client code didn't explicitly call the `toString` method; the compiler did it automatically because the `Point` object was being concatenated with a `String`. The `toString` method is also implicitly called when printing an object by itself, as in the following code:

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

In order for this implicit calling behavior to work properly, your `toString` method's signature must exactly match the one shown in this section. Changing the name or signature even slightly (for example, naming the method `ToString` with a capital T, or `convertToString`) will cause the class to produce the old output (e.g., `"Point@119c082"`). The reason has to do with concepts called inheritance and overriding that we will explore in the next chapter.

It is also legal to call `toString` explicitly if you prefer. The following client code uses an explicit `toString` call and produces the same output as the original client code:

```
System.out.println("p is " + p.toString());
```

The Java guidelines recommend writing a `toString` method in every class you write.

Common Programming Error

println Statement in toString Method

Since the `toString` method is closely related to printing, some students mistakenly think that they should place `println` statements in their `toString` methods, as in the following method:

```
// this toString method is flawed;  
// it should return the String rather than printing it  
public String toString() {  
    System.out.println("(" + x + ", " + y + ")");  
    return "";  
}
```



Continued on next page

Continued from previous page

A key idea to understand about `toString` is that it doesn't directly print anything: It simply returns a `String` that the client can use in a `println` statement.

In fact, many well-formed classes of objects do not contain any `println` statements at all. The inclusion of `println` statements in a class binds that class to a particular style of output. For example, the preceding code prints a `Point` object on its own line, making the class unsuitable for a client that doesn't want the output to appear exactly this way (say, a client that wants to print many `Point` objects on the same line).

You may wonder why the designers of Java chose to use a `toString` method rather than, say, a `print` method that would output the object to the console. The reason is that `toString` is more versatile. You can use `toString` to output the object to a file, display it on a graphical user interface, or even send the text over a network.

8.3 Object Initialization: Constructors



Our third version of the `Point` class will include the ability to create `Point` objects at any initial location. The initial state of objects is specified by writing *constructors*, which were introduced in Chapter 3. Recall that a constructor is a piece of code that initializes the state of new objects as they are created.

A clumsy aspect of our existing client code is that it takes three lines to create and initialize the state of one `Point` object:

```
// client needs 3 statements to initialize one Point object
Point p1 = new Point();
p1.x = 7;
p1.y = 2;
```

In general, when we have constructed objects, we have been able to declare and initialize them in a single statement. We might expect that we could initialize a `Point` by writing its initial (x, y) values in parentheses as we constructed it:

```
Point p1 = new Point(7, 2); // desired behavior
```

However, such a statement wouldn't be legal for our `Point` class, because we haven't written any code specifying how to create a `Point` with an initial (x, y) location. We can specify how to do this by writing a constructor in our `Point` class. The

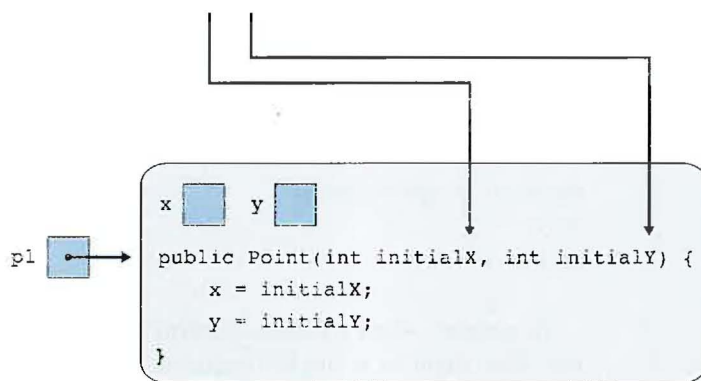
constructor executes when the client uses the `new` keyword to create a new object. When you write a constructor, you specify what parameters must be passed when clients use the `new` keyword with your type and how those parameters should be used to initialize the newly created object.

A constructor's header begins with the keyword `public`, followed by the class's name and any parameters. It looks like a method header with the same name as the class, except that you do not specify a return type. A constructor often has parameters that specify the object's initial state. Our constructor for the `Point` class will accept initial `x` and `y` values as parameters and store them into the new `Point` object's `x` and `y` fields:

```
// constructs a new point with the given (x, y) location
public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

Like instance methods, constructors execute on a particular object (the one that's being created with the `new` keyword) and can refer to that object's fields and methods directly. In this case, we store `initialX` and `initialY` parameter values into the new `Point` object's `x` and `y` fields:

```
Point p1 = new Point(7, 2);
```



Now that we are exploring constructors, it makes sense to think about the process of creating objects in more detail. When an executing Java program reaches a statement that creates a new `Point` object, several operations occur:

1. A new `Point` object is created and allocated in memory.
2. The `Point` constructor is called on the newly created object, passing 7 and 2 as the `initialX` and `initialY` parameter values.
3. A `Point` reference variable named `p` is created and set to refer to the newly created object.

Here is the complete code for the third version of our `Point` class, which now contains a constructor:

```
1 // A Point object represents a pair of (x, y) coordinates.
2 // Third version: state and behavior with constructor.
3
4 public class Point {
5     int x;
6     int y;
7
8     // constructs a new point with the given (x, y) location
9     public Point(int initialX, int initialY) {
10         x = initialX;
11         y = initialY;
12     }
13
14     // returns the distance between this point and (0, 0)
15     public double distanceFromOrigin() {
16         return Math.sqrt(x * x + y * y);
17     }
18
19     // returns a String representation of this Point
20     public String toString() {
21         return "(" + x + ", " + y + ")";
22     }
23
24     // shifts this point's location by the given amount
25     public void translate(int dx, int dy) {
26         x += dx;
27         y += dy;
28     }
29 }
```

Calling a constructor with parameters is similar to ordering a car from a factory: “I’d like the yellow one with power windows and leather seats.” You might not need to specify every detail about the car, such as the fact that it should have four wheels and headlights, but you do specify some initial attributes that are important to you.

The general syntax for constructors is the following:

```
public <class name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

When a class doesn't have a constructor, as in our previous versions of the `Point` class, Java automatically supplies a *default constructor* with no parameters. That is why it was previously legal to construct a new `Point()`. The default constructor auto-initializes all fields to zero-equivalent values. However, Java doesn't supply the default empty constructor when we supply a constructor of our own, so it is illegal to construct `Point` objects without passing in the initial `x` and `y` parameters:

 `Point p1 = new Point();` // will not compile for this version

In the next sections, we'll write additional code to restore this ability.

Common Programming Error

Using `void` with a Constructor

Many new programmers accidentally include the keyword `void` in the header of a constructor, since they've gotten used to writing a return type for every method:

 `// this code has a bug`

```
public void Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

This is actually a very tricky and annoying bug. Constructors aren't supposed to have return types. When you write a return type such as `void`, what you've created is not a constructor, but rather a normal instance method called `Point` that accepts `x` and `y` parameters and has a `void` return type. This error is tough to catch, because the `Point.java` file still compiles successfully.

You will see an error when you try to call the constructor that you thought you just wrote, though, because it isn't actually a constructor. The client code that

Continued on next page

Continued from previous page

tries to construct the `Point` object will indicate that it can't find an `(int, int)` constructor for a `Point`:

```
PointMain.java:7: cannot find symbol
symbol : constructor Point(int, int)
location: class Point
    Point p1 = new Point(7, 2);
```

If you see “cannot find symbol” constructor errors and you were positive that you wrote a constructor, double-check its header to make sure there's no return type.

Common Programming Error

Redeclaring Fields in a Constructor

Another common bug associated with constructors occurs when you mistakenly redeclare fields by writing their types. Here's an example that shows this mistake:



```
// this constructor code has a bug
public Point(int initialX, int initialY) {
    int x = initialX;
    int y = initialY;
}
```

The preceding code behaves in an odd way. It compiles successfully, but when the client code constructs a `Point` object its initial coordinates are always `(0, 0)`, regardless of the parameter values that are passed to the constructor:

```
// this client code will print that p1 is (0, 0)
Point p1 = new Point(7, 2);
System.out.println("p1 is " + p1);
```

The problem is that rather than storing `initialX` and `initialY` in the `Point` object's `x` and `y` fields, we've actually declared local variables called `x` and `y` inside the `Point` constructor. We store `initialX` and `initialY` in those local variables, which are thrown away when the constructor finishes running. No values are ever assigned to the `x` and `y` fields in the constructor, so they are automatically initialized to 0. We say that these local `x` and `y` variables *shadow* our `x` and `y` fields because they obscure the fields we intended to set.

If you observe that your constructor doesn't seem to be setting your object's fields, check closely to make sure that you didn't accidentally declare local variables that shadow your fields. The key thing is not to include a type at the front of the statement when you assign a value to a field.

The Keyword `this`

When we discussed instance methods we mentioned that an object's instance methods can refer to its other methods and fields, because the instance method code knows which object it's operating on. We called this idea the "implicit parameter." Now we will explore the mechanics behind the implicit parameter and introduce a keyword that allows us to refer to it directly.

The implicit parameter is actually a special reference that is set each time an instance method is called. You can access this reference in your code using the keyword `this`.

`this`

A Java keyword that allows you to refer to the implicit parameter inside a class.

When you refer to a field such as `x` in your code, you are actually using shorthand. The compiler converts an expression such as `x` to `this.x`. You can use the longer form in your code if you want to be more explicit. For example, our `translate` method could be rewritten as follows:

```
public void translate(int dx, int dy) {  
    this.x += dx;  
    this.y += dy;  
}
```

The code behaves the same way as the original version of the method. The explicit style is less common, but some programmers prefer it because it's clearer. It also more closely matches the style used in client code, where all messages to objects begin with a variable name and a dot.

The general syntax for using the keyword `this` to refer to fields is

```
this.<field name>
```

Similarly, when you call an instance method such as `translate`, you're actually using shorthand for a call of `this.translate`. You can use the longer form if you prefer. It has the following general syntax:

```
this.<method name>(<expression>, <expression>, ..., <expression>);
```

When the implicit parameter was introduced, we diagrammed the behavior of some method calls on two `Point` objects. Let's revisit the same example using the keyword `this`. Consider the following two `Point` objects:

```
Point p1 = new Point(7, 2);  
Point p2 = new Point(4, 3);
```

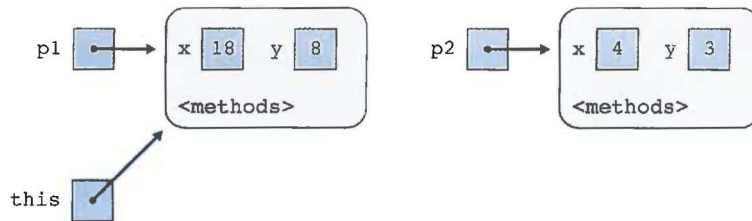
After constructing the `Point` objects, we make the following method calls:

```
p1.translate(11, 6);  
p2.translate(1, 7);
```

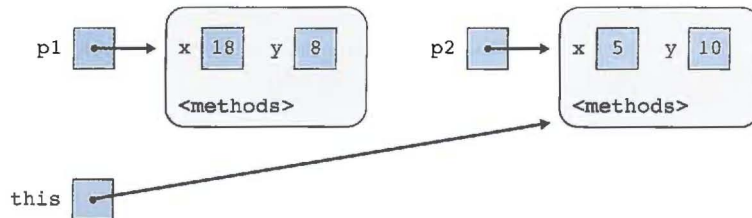
Essentially, the behavior of these two method calls is the following:

- Set `this` to refer to the same object as `p1`, and execute the `translate` method with parameters (11, 6).
- Set `this` to refer to the same object as `p2`, and execute the `translate` method with parameters (1, 7).

During the first call, `this` refers to the same object as `p1`. Therefore, the method call adjusts the (x, y) coordinates of `p1`'s object:



During the second method call `this` refers to the same object as `p2`, so the lines in the body of the `translate` method change the x and y fields of `p2`'s object:



One common usage of the keyword `this` is to deal with shadowed variables. As described earlier, shadowing occurs when a field is obscured by another variable with the same name. Shadowing can happen when a field has the same name as a parameter or local variable in a class. For example, the following is a legal header for our `Point` method, even though our fields are also called `x` and `y`:

```
public Point(int x, int y) {
```

As explained at the beginning of this section, Java would normally interpret the expression `x` to mean `this.x`. However, if a parameter or local variable called `x` exists, the program will use that one instead if you just write `x`, because the field `x` is shadowed by the parameter/variable. If you write `this.x`, though, the program will always use the field `x`.

```
public Point(int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

Of course, you can avoid this situation by naming parameters and local variables differently from fields. However, some programmers prefer the style in which a variable takes the same name as a closely related field, because it saves them from having to concoct separate parameter names like `initialX` or `newY`.

In most cases, the compiler will not allow two variables to have the same name at the same point in a program. Fields are a special case that present the risk of shadowing. Java's designers decided to allow this risk so that parameter names could match their related fields.

Multiple Constructors

A class can have multiple constructors to provide multiple ways for clients to construct objects of that class. Each constructor must have a different signature (i.e., number and type of parameters).

Our existing `Point` constructor requires two parameters (the `Point` object's initial *x*- and *y*-coordinates). Before we added the constructor, we were able to construct `Point` objects at (0, 0) without any parameters. When a class does not have a constructor, Java provides a parameterless default constructor that initializes all of the new object's fields to a zero-equivalent value. But when we added our two-parameter constructor, we lost the default constructor. This is unfortunate, because the default provided a useful shorter notation for constructing a `Point` at the origin. We can restore this ability by adding a second, parameterless constructor to our `Point` class. Our new constructor looks like this:

```
// constructs a Point object with location (0, 0)  
public Point() {  
    x = 0;  
    y = 0;  
}
```

Now it's possible to construct `Points` in two ways:

```
Point p1 = new Point(5, -2);    // (5, -2)  
Point p2 = new Point();         // (0, 0)
```

Returning to the analogy of purchasing cars, you can imagine that some customers wish to specify many details about their new cars (e.g., "I'd like a yellow Civic with gold trim, upgraded stereo system, and a sun roof"), while other customers wish to specify fewer details and want the car to contain default options instead. Having multiple constructors gives clients similar flexibility when they ask for new objects of your class.

Notice that both constructors perform similar actions; the only difference is what initial values the `x` and `y` fields receive. Another way of saying this is that the new constructor can be expressed in terms of the old constructor. The following two lines construct equivalent objects:

```
Point p1 = new Point();           // construct Point at (0, 0)
Point p2 = new Point(0, 0);       // construct Point at (0, 0)
```

A common programming practice when writing classes with multiple constructors is for one constructor to contain the true initialization code and for all other constructors to call it. This means that every object created passes through a common code path. This system can be useful when you are testing and debugging your code later. The syntax for one constructor to call another is to write the keyword `this`, followed by the parameters to pass to the other constructor in parentheses:

```
this(<expression>, <expression>, ..., <expression>);
```

This is really just the normal syntax for a method call, except that we use the special keyword `this` where we would normally put the name of a method. In our case, we want to pass parameter values of 0 and 0 to initialize each field:

```
// constructs a new point at the origin, (0, 0)
public Point() {
    this(0, 0); // calls Point(int, int) constructor
}
```

8.4 Encapsulation



Our next version of the `Point` class will protect its data from unwanted access using a concept known as *encapsulation*.

Encapsulation

Hiding the implementation details of an object from the clients of the object.

To understand the notion of encapsulation, recall the analogy of radios as objects. Almost everyone knows how to use a radio, but few people know how to build a radio or understand how the circuitry inside a radio works. It is a benefit of the radio's design that we don't need to know those details in order to use it.

The radio analogy demonstrates an important dichotomy of external versus internal views of an object. From the outside, we just see behavior. From the inside, we see the internal state that is used to accomplish that behavior (Figure 8.1).

Focusing on the radio's external behavior enables us to use it easily while ignoring the details of its inner workings that are unimportant to us. This is an example of an important computer science concept known as *abstraction*.



Figure 8.1 The internal and external views of a radio.

Abstraction

Focusing on essential properties rather than inner details.

In fact, a radio (like most other electronic devices) has a case or chassis that houses all of the electronics so that we don't see them from the outside. Dials, buttons, and displays on the case allow us to manipulate the radio without having to deal with all of the circuitry that makes it work. In fact, you wouldn't want someone to give you a fully functional radio that had wires and capacitors hanging out of it, because they would make the radio less pleasant to use.

In programming, the concept of hiding internal state from outside view is called encapsulation. When an object is properly encapsulated, its clients cannot directly access or modify its internal workings, nor do they need to do so. Only the implementer of the class needs to know about those details. Encapsulation leads to abstraction; an encapsulated object presents a more pure abstraction than one that has data which can be accessed directly.

In previous chapters you have already taken advantage of the abstraction provided by well-encapsulated objects. For example, you have used `Scanner` objects to read data from the console without knowing exactly how the `Scanner` stores and tokenizes the input data, and you have used `Random` objects to create random numbers without knowing exactly what algorithm the random number generator uses.

But so far, our `Point` class is not encapsulated. We've built a working radio, but its wires (its `x` and `y` fields) are still hanging out. Using encapsulation, we'll put a casing around our `Point` objects so that clients will only need to use the objects' methods and will not access the fields directly.

Private Fields

To encapsulate the fields of an object, we declare them to be private by writing the keyword `private` at the start of the declaration of each field. The fields of our `Point` class would be declared as follows:

```
// encapsulated fields of Point objects
private int x;
private int y;
```

We haven't yet shown a syntax template for fields because we wanted to show the preferred style with the fields private. The syntax for declaring encapsulated fields is

```
private <type> <name>;
```

Fields can also be declared with an initial value:

```
private <type> <name> = <value>;
```

Declaring fields `private` encapsulates the state of the object, in the same way that a radio's casing keeps the user from seeing the wires and circuitry inside it. Private fields are visible to all of the code inside the `Point` class (i.e., inside the `Point.java` file), but not anywhere else. This means that we can no longer directly refer to a `Point` object's `x` or `y` fields in our client code. The following client code will not compile successfully:



```
// this client code doesn't work with encapsulated points
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
```

The compiler produces error messages such as the following:

```
PointMain.java:11: x has private access in Point
PointMain.java:11: y has private access in Point
```

To preserve the functionality of our client program, we need to provide a way for client code to access a `Point` object's field values. We will do this by adding some new accessor methods to the `Point` class. If the value of an object's field might be useful externally, it is common to write an accessor to return that value. Here are the methods that provide access to a `Point` object's `x` and `y` fields:

```
// returns the x-coordinate of this point
public int getX() {
    return x;
}
// returns the y-coordinate of this point
public int getY() {
    return y;
}
```

The client code to print a `Point` object's `x` and `y` values must be changed to the following:

```
// this code works with our encapsulated Points
System.out.println("p1 is (" + p1.getX() + ", " + p1.getY() + ")");
```

It probably seems odd to grant access to a `Point` object's `x` and `y` fields when we said our goal was to encapsulate those fields, but having accessors like `getX` and `getY` doesn't actually violate the encapsulation of the object. The accessor methods just return a copy of the fields' values to the client, so that the client can see the `x` or `y` values but doesn't have any way to change them. In other words, these accessor methods give the client read-only access to the state of the object.

One drawback of encapsulating the `Point` class is that it is no longer easy for the client code to set a `Point` to a new location. For convenience, we'll add a new mutator to our encapsulated `Point` class that sets both the `x` and `y` fields of the object to new values passed as parameters:

```
// sets this point's (x, y) location to the given values
public void setLocation(int newX, int newY) {
    x = newX;
    y = newY;
}
```

Another way to set a `Point` to a new location would be to write separate methods called `setX` and `setY`. We have chosen `setLocation` partly for brevity and partly because it matches Java's actual `Point` class.

Notice that the `Point` class now has some redundancy between its two-parameter constructor and its `setLocation` method. The two bodies are essentially the same, setting the `Point` to have new `x`- and `y`-coordinates. We can eliminate this redundancy by having the constructor call `setLocation` rather than setting the field values manually. It is legal for an object to call its own instance methods from a constructor or another instance method:

```
// constructs a new point with the given (x, y) location
public Point(int x, int y) {
    setLocation(x, y);
}
```

We can eliminate a bit more redundancy using this technique. Translating a `Point` can be thought of as setting its location to the old location plus the `dx` and `dy`, so we can modify the `translate` method to call the `setLocation` method:

```
// shifts this point's location by the given amount
public void translate(int dx, int dy) {
    setLocation(x + dx, y + dy);
}
```

Did You Know?**The Perils of Poor Encapsulation**

Many novices (as well as many professional programmers) do not fully appreciate the concepts of abstraction and encapsulation. It is tempting to write classes that directly expose their data for clients to use, since private fields introduce some complexity and restrictions into a program. However, there have been some famous examples where a lack of proper encapsulation and abstraction caused a large problem.

One such example is the “Y2K” (year 2000) or “millennium bug” scare of late 1999. The issue arose because a large number of computer programs represented years by using only two digits, such as 72 for 1972. They followed this convention largely to save memory, since many of them were older programs written in a language called COBOL during a time when memory was more scarce. Once the year became 2000, the programs would incorrectly think that the year was 1900, and this might cause them to fail.

Making matters worse was the fact that many of these programs contained their own handwritten logic for representing dates, which sometimes appeared in many places in the code. In order for the program to represent a year with more than two digits, many places in the code needed to be changed. In total, over \$300 billion was spent on repairing old programs and systems to correct the Y2K problem.

If the old programs had used an encapsulated `Date` class that included a field representing the year, far less work would have been needed to fix the Y2K bug. This `Date` class could have been updated once and all the client code would have received the benefits.

Surprisingly, Java’s class libraries also contain examples of poorly encapsulated classes. In the `java.awt` package, for example, the `Point` and `Dimension` classes have public fields. (A `Dimension` object stores `width` and `height` fields to represent the size of an onscreen region.) Many client programs access the fields directly when they use these objects. Java’s developers regret this decision:

Several classes in the Java platform libraries violate the advice that public classes should not expose fields directly. Prominent examples include the `Point` and `Dimension` classes in the `java.awt` package. Rather than examples to be emulated, these classes should be regarded as cautionary tales. [...] The decision to expose the internals of the `Dimension` class resulted in a serious performance problem that could not be solved without affecting clients.

—Joshua Bloch, *Effective Java*

Now that we've introduced all the major elements of a well-encapsulated class, it's time to look at a proper syntax template for an entire class. The Java style guidelines suggest putting fields at the top of the class, followed by constructors, followed by methods:

```
public class <class name> {
    // fields
    private <type> <name>;
    private <type> <name>;
    ...
    // constructors
    public <class name>(<type> <name>, ..., <type> <name>) {
        <statement>;
        <statement>;
        ...
        <statement>;
    }
    ...
    // methods
    public <type> <name>(<type> <name>, ..., <type> <name>) {
        <statement>;
        <statement>;
        ...
        <statement>;
    }
    ...
}
```

Here is the fourth complete version of our `Point` class, including encapsulated fields and accessor methods `getX` and `getY`:

```
1 // A Point object represents a pair of (x, y) coordinates.
2 // Fourth version: encapsulated.
3
4 public class Point {
5     private int x;
6     private int y;
7
8     // constructs a new point at the origin, (0, 0)
9     public Point() {
10         this(0, 0); // calls Point(int, int) constructor
11     }
12
13     // constructs a new point with the given (x, y) location
14     public Point(int x, int y) {
```

```

15         setLocation(x, y);
16     }
17
18     // returns the distance between this Point and (0, 0)
19     public double distanceFromOrigin() {
20         return Math.sqrt(x * x + y * y);
21     }
22
23     // returns the x-coordinate of this point
24     public int getX() {
25         return x;
26     }
27
28     // returns the y-coordinate of this point
29     public int getY() {
30         return y;
31     }
32
33     // sets this point's (x, y) location to the given values
34     public void setLocation(int x, int y) {
35         this.x = x;
36         this.y = y;
37     }
38
39     // returns a String representation of this point
40     public String toString() {
41         return "(" + x + ", " + y + ")";
42     }
43
44     // shifts this point's location by the given amount
45     public void translate(int dx, int dy) {
46         setLocation(x + dx, y + dy);
47     }
48 }

```

Here's the corresponding final version of our client program, which now uses the Point constructors and methods appropriately:

```

1 // A program that deals with points.
2 // Fourth version, to accompany encapsulated Point class.
3
4 public class PointMain {
5     public static void main(String[] args) {
6         // create two Point objects
7         Point p1 = new Point(7, 2);

```

```

8      Point p2 = new Point(4, 3);
9
10     // print each point and its distance from the origin
11     System.out.println("p1 is " + p1);
12     System.out.println("distance from origin = " +
13         p1.distanceFromOrigin());
14     System.out.println("p2 is " + p2);
15     System.out.println("distance from origin = " +
16         p2.distanceFromOrigin());
17
18     // translate each point to a new location
19     p1.translate(11, 6);
20     p2.translate(1, 7);
21
22     // print the points again
23     System.out.println("p1 is " + p1);
24     System.out.println("p2 is " + p2);
25 }
26 }

```

Class Invariants

In this section we will develop another new class to illustrate a particular benefit of encapsulation. Consider a program that measures or deals with elapsed intervals of time, such as a program for a stopwatch, a scheduler, a TV recorder, or an airline flight system. A useful abstraction in such a program would be an object representing an elapsed span of time.

Let's write a class called `TimeSpan`, in which each `TimeSpan` object represents an interval of elapsed hours and minutes. For example, we could construct a `TimeSpan` representing an interval of 6 hours and 15 minutes. We'll represent only hours and minutes, ignoring larger or smaller units such as days or seconds.

Since we're representing intervals of hours and minutes, it seems natural to use these two quantities as fields in our class. We will encapsulate the class properly from the start by declaring the fields as `private`:

```

// represents a time span of elapsed hours and minutes
public class TimeSpan {
    private int hours;
    private int minutes;
    ...
}

```

The constructor for a `TimeSpan` object will accept hours and minutes as parameters and store the values into the object's fields. However, there is a potential problem: What should we do about values of minutes that are 60 or greater? Should the

client program be allowed to construct a `TimeSpan` object representing 0 hours and 157 minutes? A more natural representation of this amount would be 2 hours and 37 minutes. And what about values for hours or minutes that are negative? It doesn't make sense to have a span of -2 hours or -35 minutes. Ideally we should not allow a `TimeSpan` object to store such a value.

Let's make a design decision that we will only allow `TimeSpan` objects to store a value for minutes that is between 0 and 59 inclusive. If the client tries to construct a `TimeSpan` object with a negative number of hours or minutes, we could resolve the problem by printing an error message or by setting the fields to 0. But in cases like this, the negative value often comes from a bug or a mistake in the client's understanding of our class. The best way to handle a violation like this is to throw an exception so that the client will know the parameter values passed were illegal.

If the user tries to construct a `TimeSpan` object with more than 60 minutes, we will convert the excess minutes into hours. You might be tempted to use an `if/else` statement or a loop to handle minutes above 60, but there is a simpler solution. For a large number of minutes such as 157, dividing by 60 will produce the hours (2) and using the `%` operator by 60 will produce the remaining minutes (37). The `hours` field should really store the `hours` parameter plus the `minutes` parameter divided by 60, and the `minutes` field should store the remaining minutes:

```
public TimeSpan(int hours, int minutes) {
    if (hours < 0 || minutes < 0) {
        throw new IllegalArgumentException();
    }

    this.hours = hours + minutes / 60;
    this.minutes = minutes % 60;
}
```

A useful behavior of a `TimeSpan` object would be the ability to add more hours and minutes to the span. An airline scheduling program might use this behavior to add the elapsed times for two back-to-back flights to determine the total travel time for a passenger's trip. Let's implement this behavior as a method called `add` that accepts `hours` and `minutes` as parameters. Here's an initial incorrect version of that method:

```
// an incorrect version of an add method
public void add(int hours, int minutes) {
    this.hours += hours;
    this.minutes += minutes;
}
```

The problem with the preceding code is that it allows the client to put the object into an invalid state. If the client passes a value of `minutes` that is large enough to make the total minutes greater than 60, the `minutes` field will have an invalid value.

For example, if the client adds 45 minutes to a time span of 1 hour and 30 minutes, the result will be 1 hour and 75 minutes.

We decided that we wanted every `TimeSpan` object to store only valid numbers of minutes between 0 and 59. We wrote code in our constructor to ensure that this would be true of each object's initial state, but really we want to ensure that the condition is true for every object throughout its entire lifespan, not just when it is initially created. Such a property that is true of every object of a class is called a *class invariant*.

Class Invariant

An assertion about an object's state that is true for the lifetime of that object.

Class invariants are related to preconditions, postconditions, and assertions, as presented in Chapters 4 and 5. We cannot allow any mutator method such as `add` to break the invariants we have decided upon; a class invariant should be treated as an implicit postcondition of every method in the class. Enforcing an invariant may cause you to add preconditions to the constructors and mutator methods of your class.

We can write code at the end of the `add` method to deal with invalid numbers of minutes and hours. First we want the program to throw an exception if the hours or minutes that are passed are negative. We will also make sure that we convert each group of 60 minutes into an hour. The following code implements the behavior:

```
public void add(int hours, int minutes) {
    if (hours < 0 || minutes < 0) {
        throw new IllegalArgumentException();
    }
    this.hours += hours;
    this.minutes += minutes;

    // converts each 60 minutes into one hour
    this.hours += this.minutes / 60;
    this.minutes = this.minutes % 60;
}
```

The code now enforces its invariant in two places: the constructor and `add`. It would be preferable to solve this problem in one place rather than redundantly checking for it throughout the class. A more elegant solution is to have the constructor initialize the fields to 0 and then call the `add` method, which performs the necessary invariant checks and stores the values of hours and minutes:

```
public TimeSpan(int hours, int minutes) {
    this.hours = 0;
    this.minutes = 0;
    add(hours, minutes);
}
```

The fields would have been auto-initialized to 0 anyway, but many programmers prefer to explicitly initialize field values for clarity.

Another useful operation for `TimeSpan` objects is the ability to print them on the console. We'll add this ability by including a `toString` method that returns a `String` such as "2h 35m" for 2 hours and 35 minutes.

Here is the code for the complete `TimeSpan` class that enforces its invariant:

```
1 // Represents a time span of hours and minutes elapsed.
2 // Class invariant: hours >= 0 && minutes >= 0 && minutes < 60
3
4 public class TimeSpan {
5     private int hours;
6     private int minutes;
7
8     // Constructs a time span with the given interval.
9     // pre: hours >= 0 && minutes >= 0
10    public TimeSpan(int hours, int minutes) {
11        this.hours = 0;
12        this.minutes = 0;
13        add(hours, minutes);
14    }
15
16    // Adds the given interval to this time span.
17    // pre: hours >= 0 && minutes >= 0
18    public void add(int hours, int minutes) {
19        if (hours < 0 || minutes < 0) {
20            throw new IllegalArgumentException();
21        }
22
23        this.hours += hours;
24        this.minutes += minutes;
25
26        // converts each 60 minutes into one hour
27        this.hours += this.minutes / 60;
28        this.minutes = this.minutes % 60;
29    }
30
31    // returns a String for this time span, such as "6h 15m"
32    public String toString() {
33        return hours + "h " + minutes + "m";
34    }
35 }
```

Some additional features should be included in the class, such as accessors for the field values, but these are left as exercises.

Invariants bring to light the importance of proper encapsulation. If the `TimeSpan` class weren't encapsulated, we would not be able to properly enforce our invariant. A buggy or malicious client would be able to make a `TimeSpan` object's state invalid by setting its fields' values directly. When the class is encapsulated, it has much better control over how clients can use its objects, making it impossible for a misguided client program to violate the class invariant.

Changing Internal Implementations

Another important benefit of encapsulation is that it allows us to make internal design changes to a class without affecting its clients. A subtlety of classes is that the internal representation of a class does not necessarily have to match the external view that the client sees. A client of the `TimeSpan` class thinks of each time span as a number of hours and a number of minutes. But the `TimeSpan` object does not have to internally store its time using those two data fields. In fact, the code for `TimeSpan` becomes simpler if we simply store a single field for the total number of minutes. For example, we can represent 2 hours and 15 minutes as 135 total minutes by converting each hour into 60 minutes.

Let's rewrite the `TimeSpan` class to use only a single total minutes field:

```
// alternate implementation using only total minutes
public class TimeSpan {
    private int totalMinutes;
    ...
}
```

Because our class is encapsulated, as long as our methods still produce the same results from an external point of view, we can change the implementation of the object's internal state and the clients will not need to be modified. We can implement the same constructor, `add`, and `toString` behavior using total minutes. For example, the `add` method needs to combine the hours and minutes together and add both of them into the `totalMinutes`. We'll scale the hours by 60 as we add them to the total:

```
public void add(int hours, int minutes) {
    if (hours < 0 || minutes < 0) {
        throw new IllegalArgumentException();
    }
    totalMinutes += 60 * hours + minutes;
}
```

Notice that this new implementation makes it easier to enforce our class invariant about objects having valid state. We must still check for negative parameters, but we

no longer need to worry about storing minutes of 60 or greater. All minutes added are properly grouped into the common total.

The constructor and `toString` method also require minor modifications to account for our new representation. Here is the complete class, implemented with total minutes instead of hours and minutes. This version is shorter and simpler than the original:

```

1 // Represents a time span of elapsed hours and minutes.
2 // Second implementation using a single field for total minutes.
3 // Class invariant: totalMinutes >= 0
4
5 public class TimeSpan {
6     private int totalMinutes;
7
8     // Constructs a time span with the given interval.
9     // pre: hours >= 0 && minutes >= 0
10    public TimeSpan(int hours, int minutes) {
11        totalMinutes = 0;
12        add(hours, minutes);
13    }
14
15    // Adds the given interval to this time span.
16    // pre: hours >= 0 && minutes >= 0
17    public void add(int hours, int minutes) {
18        if (hours < 0 || minutes < 0) {
19            throw new IllegalArgumentException();
20        }
21        totalMinutes += 60 * hours + minutes;
22    }
23
24    // returns a String for this time span, such as "6h 15m"
25    public String toString() {
26        return (totalMinutes / 60) + "h " +
27            (totalMinutes % 60) + "m";
28    }
29 }

```

As another example, we could revisit our encapsulated `Point` class and change its internal structure without having to modify the client code. For example, sometimes it is useful to express two-dimensional points in polar coordinates in terms of a radius r and an angle θ . In this representation, the (x, y) coordinates of a point are not stored directly but can be computed as $(r \cos(\theta), r \sin(\theta))$. When the `Point` class is encapsulated, we can modify it to use r and θ fields internally, then modify `getX`, `getY`, and other methods so that they still return appropriate values. The

polar representation would not reap the same benefits we saw in our second implementation of `TimeSpan`, but it might make it easier for us to add certain functionality to the `Point` class later.

8.5 Case Study: Designing a Stock Class



VideoNote

So far we have written several classes, but we have not talked about how to design a class or how to break apart a programming problem into classes. In this section we'll examine a larger programming problem and design a class and client to solve it. We will create a class called `Stock` and a client program that compares the performance of stocks that the user has purchased.

Consider the task of writing a financial program to record purchases of shares of two stocks and report which has the greatest profit. The investor may have made several purchases of the same `Stock` at different times and prices. The interaction with the program might look like this:

```
First stock's symbol: AMZN
How many purchases did you make? 2
1: How many shares, at what price per share? 50 35.06
2: How many shares, at what price per share? 25 38.52
What is today's price per share? 37.29
Net profit/loss: $80.75

Second stock's symbol: INTC
How many purchases did you make? 3
1: How many shares, at what price per share? 15 16.55
2: How many shares, at what price per share? 10 18.09
3: How many shares, at what price per share? 20 17.15
What is today's price per share? 17.82
Net profit/loss: $29.75

AMZN was more profitable than INTC.
```

The program must perform several actions: prompting the user for input, calculating the amount spent on each purchase of stock, reporting profits, and so on. The client program could perform all of these actions and could keep track of the financial data using existing types such as `doubles` and `Strings`. However, recall that we began this chapter by talking about object-oriented reasoning. When you are studying complex programs, it is often useful to think about the problem in terms of the relevant objects that could solve it, rather than placing all behavior in the client program. In this particular program, we must perform several computations that involve keeping track of purchases of shares of a particular stock, so it would be useful to store the purchase information in an object.

One possible design would be to create a `Purchase` class that records information about a single purchase of shares of a particular stock. For example, if the user specified three purchases, the program should construct three `Purchase` objects. However, a more useful abstraction here would be to hold the overall information about all purchases of one stock in one place. The investor may make many purchases of the same stock, so you want to have an easy way to accumulate these shares and their total cost into a single object.

Therefore, instead of a `Purchase` class, we'll write a `Stock` class. Each `Stock` object will keep track of the investor's accumulated shares of one stock and can provide profit/loss information. Our `Stock` class will reside in a file called `Stock.java`, and the client program itself will reside in a separate file called `StockMain.java`.

Object-Oriented Design Heuristics

We now face the important task of deciding on the contents of our `Stock` class. It can be tricky to choose a good set of classes and objects to solve a complex programming problem. Chapter 4's case study introduced a set of procedural design *heuristics*, or guidelines for good design, for effectively dividing a problem into methods. There are similar guidelines for effectively breaking a large program into a set of classes and objects. The heuristics we'll discuss here are based on those listed in computer scientist Arthur Riel's influential book, *Object-Oriented Design Heuristics*.

First let's look at the overall set of responsibilities, the things that a class must know or do to solve the overall problem:

- Prompt the user for each stock's symbol and store the information somewhere.
- Prompt the user for the number of purchases of each stock.
- Read each purchase (number of shares and price per share) from the console and store the information somewhere.
- Compute the total profit/loss of each stock.
- Print the total profit/loss of each stock to the console.
- Compare the two total profits/losses and print a message to the console about which stock performed better.

It might be tempting to make most or all of these tasks responsibilities of our `Stock` class. We could make a `Stock` object store all the purchases of both stocks, prompt for information from the console, print the results, and so on. But a key guideline when writing classes is that they should have *cohesion*.

Cohesion

The extent to which the code for a class represents a single abstraction.

Placing all the responsibilities in the `Stock` class would not allow that class to represent a single clear abstraction. The abstraction we want to represent in the `Stock` class is the accumulated purchases of a single stock.

One set of responsibilities that `Stock` objects should not handle is producing the console input and output. We need to prompt the user for information and print messages, but these functions are specific to the current client program. Objects are meant to be reusable pieces of software, and other programs might wish to track stock purchases without using these exact messages or prompts. If the `Stock` class handles the prompts and printing, it will be heavily intertwined with this client program and will not be easily reusable by other clients.

In general, we want to reduce unnecessary dependencies among classes. Dependencies among classes in an object-oriented program contribute to *coupling*, the degree to which one part of a program depends on another.

Striving to avoid unnecessary coupling is a second design heuristic commonly used in object-oriented programming. A design that avoids this problem is sometimes said to have *loose coupling*.

Let's divide some of the responsibilities now, based on our heuristics. Since the `StockMain` client program will perform the console I/O, it should handle the following responsibilities:

StockMain

- Prompt for each stock's symbol.
- Prompt for the number of purchases of each stock.
- Read each purchase (number of shares and price per share) from the console.
- Print the total profit/loss of each stock.
- Compare the two total profits/losses and print a message about which stock generated higher profits.

Since the `StockMain` client program will be performing the console I/O, it might seem natural for it also to store the information about each stock purchase (that is, the number of shares and the price paid). But our `Stock` object should contain the functionality to compute a stock's total profit or loss, and it will need to have the data about all purchases to do so. This leads us to a third design heuristic: Related data and behavior should be in the same place. With that in mind, we can write out the responsibilities for the `Stock` class as follows:

Stock

- Store a stock's symbol.
- Store accumulated information about the investor's purchases of the stock.
- Record a purchase of the stock.
- Compute the total profit/loss for the stock.

When they are designing large object-oriented programs, many software engineers write information about classes as we've done here. A common technique to brainstorm ideas for classes is to write information on index cards. Each card is called a

CRC card and lists the Class, its Responsibilities, and its Collaborators (other classes to which it is coupled).

The following list summarizes the design heuristics discussed in this section:

- A class should be cohesive, representing only one abstraction.
- A class should avoid unnecessary coupling.
- Related data and behavior should be in the same class.

Note that we began our design by looking at responsibilities rather than by specifying fields as we did when we developed our `Point` class. We began writing the `Point` class by discussing fields because the data associated with a point is simple and more obvious than the data associated with stock purchases. But in many larger problems like this one, working backward from behavior and responsibilities is a better technique.

Stock Fields and Method Headers

In this section we'll decide on a design for the method names and signatures the `Stock` should use to implement its behavior. We'll use this design to determine which fields are required to implement the behavior.

We've decided that a `Stock` object should allow clients to record purchases and request the total profit or loss. Each of these tasks can be represented as a method. The recording of a purchase can be represented as a method called `purchase`. The retrieval of the total profit or loss can be represented as a method called `getProfit`.

The `purchase` method should record information about a single purchase. A purchase consists of a number of shares that the user bought (which we can assume is a whole number) and a price per share (which can include real numbers with both dollars and cents). Our `purchase` method should accept two parameters: an `int` for the number of shares bought and a `double` for the price per share. The method can use a `void` return type, since nothing needs to be returned after each purchase is recorded:

```
public void purchase(int shares, double pricePerShare)
```

The `getProfit` method will return the amount of money that the user made or lost on all accumulated purchases of this stock. Consider an investor who has made the following three purchases of a stock:

Purchase #1:	20 shares * \$10 per share = \$ 200 cost
Purchase #2:	20 shares * \$30 per share = \$ 600 cost
Purchase #3:	10 shares * \$20 per share = \$ 200 cost
	50 total shares, \$1000 total cost

If today's price per share is \$22.00, the current market value of the investor's 50 shares is $(50 * 22)$ or \$1100. Since the investor paid \$1000 total for the shares and they are now worth \$1100, the investor has made $(\$1100 - \$1000) = \$100$ of profit.

The general formula for the profit is the following:

$$\text{profit} = ((\text{total shares}) * (\text{current share price})) - (\text{total cost})$$

The total number of shares and total cost figure needed for this calculation are the accumulated information from all the purchases that have been made of this stock. This means that information will need to be stored during each call of the `purchase` method to be used later in the `getProfit` method. A key observation is that we do not need to store the number of shares, price per share, and cost for every purchase: We only need to store cumulative sums of the total shares purchased so far and the total dollars spent so far to acquire those values.

The third value we need in order to calculate the profit is the current share price. We could choose to make this a field in the `Stock` class as well, but the share price is a dynamic value that changes regularly. We use it during a single call to the `getProfit` method, but the next call may come at a later date when the price per share has changed.

This problem leads us to another design heuristic: Fields should represent values of core importance to the object and values that are used in multiple methods. Adding too many fields clutters a class and can make its code harder to read. If a value is used in only one method of the class, it's best to make it a parameter to that method rather than a field. Therefore, we'll make the share price a parameter to the `getProfit` method.

```
public double getProfit(double currentPrice)
```

One piece of state that we haven't discussed yet is that each stock has a symbol, such as "AMZN". We'll store the symbol as a `String` field in each `Stock` object.

Here's a skeleton of our `Stock` class so far:

```
// incomplete Stock class
public class Stock {
    private String symbol;
    private int totalShares;
    private double totalCost;
    ...

    public double getProfit(double currentPrice) {
        ...
    }

    public void purchase(int shares, double pricePerShare) {
        ...
    }
}
```

Stock Method and Constructor Implementation

Now that we've decided on some of the `Stock`'s state and behavior, let's think about how to construct `Stock` objects. The client program will need the ability to create two `Stocks` and record purchases of them.

It may be tempting to write a constructor that accepts three parameters: the symbol, the total number of shares purchased, and the total cost. But our `Stock` objects are accumulators of purchases, and we may want to be able to create new `Stock` objects before the program records initial purchases. Let's design our class to require only the symbol as a parameter and initialize the other fields to 0:

```
// initializes a new Stock with no shares purchased
public Stock(String theSymbol) {
    symbol = theSymbol;
    totalShares = 0;
    totalCost = 0.0;
}
```

When a constructor takes an object as a parameter (such as the `String theSymbol`), it might make sense to check that parameter's value to make sure it isn't null. One possible way to handle this case would be to throw an exception if a null symbol is passed when the program creates a `Stock` object. We could do this by inserting the following lines at the start of the `Stock`'s constructor:

```
if (theSymbol == null) {
    throw new NullPointerException();
}
```

The Java convention is to throw a `NullPointerException` when a parameter's value is null but should not be. For other invalid parameter values, throw an `IllegalArgumentException`.

Now let's write the body of the `purchase` method. The task of recording the purchase consists of adding the new number of shares to the total number of shares and adding the new price paid for these shares to the total cost. The price paid is equal to the number of shares times the price per share. Here's the code for the `purchase` method to implement this behavior:

```
// records a purchase of the given number of shares of this stock
// at the given price per share
public void purchase(int shares, double pricePerShare) {
    totalShares += shares;
    totalCost += shares * pricePerShare;
}
```

It might make sense here to check the parameters passed in to make sure they are valid, as we did with the constructor. In this case, valid numbers of shares and prices

per share must not be negative numbers. To perform this test, we can insert the following lines at the start of our purchase method:

```
if (shares < 0 || pricePerShare < 0) {
    throw new IllegalArgumentException();
}
```

Next, we'll write the body of the `getProfit` method. As we noted previously, the profit of a stock is equal to its current market value minus the amount that was paid for it:

$$\text{profit} = ((\text{total shares}) * (\text{current share price})) - (\text{total cost})$$

We can implement this formula in a straightforward manner using the `totalShares` and `totalCost` fields and the `currentPrice` parameter:

```
// Returns the total profit or loss earned on this stock,
// based on the given price per share.
public double getProfit(double currentPrice) {
    return totalShares * currentPrice - totalCost;
}
```

Note that parentheses are not needed in the code because multiplication has a higher precedence than subtraction.

As we did for the other methods, we should check for illegal parameter values. In this case, we shouldn't allow a negative current price per share. To ensure that this doesn't happen, we can place the following code at the start of the method:

```
if (currentPrice < 0.0) {
    throw new IllegalArgumentException();
}
```

After we've written all the fields, the constructor, and the methods of our `Stock`, the class will look like this:

```
1 // A Stock object represents purchases of shares of a stock.
2
3 public class Stock {
4     private String symbol;        // stock symbol, e.g. "YHOO"
5     private int totalShares;      // total shares purchased
6     private double totalCost;    // total cost for all shares
7
8     // initializes a new Stock with no shares purchased
9     // pre: symbol != null
```

```
10     public Stock(String theSymbol) {
11         if (theSymbol == null) {
12             throw new NullPointerException();
13         }
14
15         symbol = theSymbol;
16         totalShares = 0;
17         totalCost = 0.0;
18     }
19
20     // returns the total profit or loss earned on this stock,
21     // based on the given price per share
22     // pre: currentPrice >= 0.0
23     public double getProfit(double currentPrice) {
24         if (currentPrice < 0.0) {
25             throw new IllegalArgumentException();
26         }
27
28         double marketValue = totalShares * currentPrice;
29         return marketValue - totalCost;
30     }
31
32     // records purchase of the given shares at the given price
33     // pre: shares >= 0 && pricePerShare >= 0.0
34     public void purchase(int shares, double pricePerShare) {
35         if (shares < 0 || pricePerShare < 0.0) {
36             throw new IllegalArgumentException();
37         }
38
39         totalShares += shares;
40         totalCost += shares * pricePerShare;
41     }
42 }
```

Here's the client code to use the Stock class:

```
1 // This program tracks the user's purchases of two stocks,
2 // computing and reporting which stock was more profitable.
3
4 import java.util.*;
5
6 public class StockMain {
7     public static void main(String[] args) {
8         Scanner console = new Scanner(System.in);
```



```

9
10     // first stock
11     System.out.print("First stock's symbol: ");
12     String symbol1 = console.next();
13     Stock stock1 = new Stock(symbol1);
14     double profit1 = makePurchases(stock1, console);
15
16     // second stock
17     System.out.print("Second stock's symbol: ");
18     String symbol2 = console.next();
19     Stock stock2 = new Stock(symbol2);
20     double profit2 = makePurchases(stock2, console);
21
22     // report which stock made more money
23     if (profit1 > profit2) {
24         System.out.println(symbol1 + " was more " +
25             "profitable than" + symbol2 + ".");
26     } else if (profit2 > profit1) {
27         System.out.println(symbol2 + " was more " +
28             "profitable than " + symbol1 + ".");
29     } else { // profit1 == profit2
30         System.out.println(symbol1 + " and " + symbol2 +
31             " are equally profitable.");
32     }
33 }
34
35 // make purchases of stock and return the profit
36 public static double makePurchases(Stock currentStock,
37     Scanner console) {
38     System.out.print("How many purchases did you make? ");
39     int numPurchases = console.nextInt();
40
41     // ask about each purchase
42     for (int i = 1; i <= numPurchases; i++) {
43         System.out.print(i +
44             ": How many shares, at what price per share? ");
45         int numShares = console.nextInt();
46         double pricePerShare = console.nextDouble();
47
48         // ask the Stock object to record this purchase
49         currentStock.purchase(numShares, pricePerShare);
50     }
51
52     // use the Stock object to compute profit

```

```

53      System.out.print("What is today's price per share? ");
54      double currentPrice = console.nextDouble();
55
56      double profit = currentStock.getProfit(currentPrice);
57      System.out.println("Net profit/loss: $" + profit);
58      System.out.println();
59      return profit;
60  }
61 }

```

It would be useful to have a few other methods in our `Stock` objects. For example, it would be good to implement accessors for the `Stock`'s data (the symbol, number of shares, and so on), and a `toString` method to easily print `Stock` objects. We could even add a second constructor that would accept an initial number of shares and cost. These features are left for you to implement as exercises.

Chapter Summary

Object-oriented programming is a different philosophy of writing programs that focuses on nouns or entities in a program, rather than on verbs or actions of a program. In object-oriented programming, state and behavior are grouped into objects that communicate with each other.

A class serves as the blueprint for a new type of object, specifying the object's data and behavior. The class can be asked to construct many objects (also called "instances") of its type.

Java's `java.awt` package has a class named `Point`. Each object holds two `int` values, `x` and `y`. A `Point` can be constructed, translated to a new location, and printed on the console.

The data for each object are specified using special variables called fields.

The behavior of each object is specified by writing instance methods in the class. Instance methods exist inside an object and can access that object's internal state.

To make objects easily printable, write a `toString` method that returns the object's text representation.

A class can define special code called a constructor that initializes the state of new objects as they are created. The constructor will be called when external client code creates a new object of your type using the `new` keyword.

You can use the keyword `this` to have an object refer to itself. It is also used when a class has multiple constructors and one constructor wishes to call another.

Objects can protect their internal data from unwanted external modification by declaring them to be `private`, an action known as encapsulation. Encapsulation provides abstraction so that clients can use the objects without knowing about their internal implementation.

A class should represent only one key abstraction with related data and behavior, and it should be independent from its clients.

Self-Check Problems

Section 8.1: Object-Oriented Programming

1. Describe the difference between object-oriented programming and procedural programming.
2. What is an object? How is an object different from a class?
3. What is the state of a `String` object? What is its behavior?
4. What is the output of the following program?

```
public class ReferenceMystery3 {
    public static void main(String[] args) {
        int a = 7;
        int b = 9;
        Point p1 = new Point(2, 2);
        Point p2 = new Point(2, 2);
        addToXTwice(a, p1);
        System.out.println(a + " " + b + " " + p1.x + " " + p2.x);
        addToXTwice(b, p2);
        System.out.println(a + " " + b + " " + p1.x + " " + p2.x);
    }

    public static void addToXTwice(int a, Point p1) {
        a = a + a;
        p1.x = a;
        System.out.println(a + " " + p1.x);
    }
}
```

5. Imagine that you are creating a class called `Calculator`. A `Calculator` object could be used to program a simple mathematical calculator device like the ones you have used in math classes in school. What state might a `Calculator` object have? What might its behavior be?

Section 8.2: Object State and Behavior

6. Explain the differences between a field and a parameter. What is the difference in their syntax? What is the difference in their scope and the ways in which they may be used?
7. Create a class called `Name` that represents a person's name. The class should have fields representing the person's first name, last name, and middle initial. (Your class should contain only fields for now.)
8. What is the difference between an accessor and a mutator? What naming conventions are used with accessors and mutators?
9. Suppose we have written a class called `BankAccount` with a method inside it, defined as:

```
public double computeInterest(int rate)
```

If the client code has declared a `BankAccount` variable named `acct`, which of the following would be a valid call to the above method?

- a. `double result = computeInterest(acct, 42);`
- b. `acct.computeInterest(42.0, 15);`
- c. `int result = BankAccount.computeInterest(42);`


```
d. double result = acct.computeInterest(42);
```

```
e. new BankAccount(42).computeInterest();
```

10. Add a new method to the `Point` class we developed in this chapter:

```
public double distance(Point other)
```

Returns the distance between the current `Point` object and the given other `Point` object. The distance between two points is equal to the square root of the sum of the squares of the differences of their x - and y -coordinates. In other words, the distance between two points (x_1, y_1) and (x_2, y_2) can be expressed as the square root of $(x_2 - x_1)^2 + (y_2 - y_1)^2$. Two points with the same (x, y) coordinates should return a distance of 0.0.

11. (You must complete Self-Check Problem 7 before answering this question.)

Add two new methods to the `Name` class:

```
public String getNormalOrder()
```

Returns the person's name in normal order, with the first name followed by the middle initial and last name. For example, if the first name is "John", the middle initial is "Q", and the last name is "Public", returns "John Q. Public".

```
public String getReverseOrder()
```

Returns the person's name in reverse order, with the last name preceding the first name and middle initial. For example, if the first name is "John", the middle initial is "Q", and the last name is "Public", returns "Public, John Q."

12. How do you write a class whose objects can easily be printed on the console?

13. The following `println` statement (the entire line) is equivalent to what?

```
Point p1 = new Point();
```

```
...
```

```
System.out.println(p1);
```

```
a. System.out.println(toString(p1));
```

```
b. p1.toString();
```

```
c. System.out.println(p1.toString());
```

```
d. System.out.println(p1.string());
```

```
e. System.out.println(Point.toString());
```

14. The `Point` class in the `java.awt` package has a `toString` method that returns a `String` in the following format:

```
java.awt.Point[x=7,y=2]
```

Write a modified version of the `toString` method on our `Point` class that returns a result in this format.

15. (You must complete Self-Check Problem 7 before answering this question.)

Write a `toString` method for the `Name` class that returns a `String` such as "John Q. Public".

16. Finish the following client code so that it constructs two `Point` objects, translates each, and then prints their coordinates.

```
// construct two Point objects, one at (8, 2) and one at (4, 3)
```

```
System.out.println("p1 is " ...); // display the objects' state
```

```
System.out.println("p2 is " ...);
```

```
System.out.println("p1's distance from origin is " ...);
```

```
// translate p1 to (9, 4) and p2 to (3, 13)
```

```
System.out.println("p1 is now " ...); // display state again
```

```
System.out.println("p2 is now " ...);
```


Section 8.3: Object Initialization: Constructors

17. What is a constructor? How is a constructor different from other methods?

18. What are two major problems with the following constructor?

```
public void Point(int initialX, int initialY) {  
    int x = initialX;  
    int y = initialY;  
}
```

19. (You must complete Self-Check Problem 7 before answering this question.)

Add a constructor to the `Name` class that accepts a first name, middle initial, and last name as parameters and initializes the `Name` object's state with those values.

20. What is the meaning of the keyword `this`? Describe three ways that the keyword can be used.

21. Add a constructor to the `Point` class that accepts another `Point` as a parameter and initializes this new `Point` to have the same (x, y) values. Use the keyword `this` in your solution.

Section 8.4: Encapsulation

22. What is abstraction? How do objects provide abstraction?

23. What is the difference between the `public` and `private` keywords? What items should be declared `private`?

24. When fields are made `private`, client programs cannot see them directly. How do you allow classes access to read these fields' values, without letting the client break the object's encapsulation?

25. Add methods named `setX` and `setY` to the `Point` class that allow clients to change a `Point` object's x- and y-coordinates, respectively.

26. (You must complete Self-Check Problem 7 before answering this question.)

Encapsulate the `Name` class. Make its fields `private` and add appropriate accessor methods to the class.

27. (You must complete Self-Check Problem 26 before answering this question.)

Add methods called `setFirstName`, `setMiddleInitial`, and `setLastName` to your `Name` class. Give the parameters the same names as your fields, and use the `this` keyword in your solution.

28. How does encapsulation allow you to change the internal implementation of a class?

Section 8.5: Case Study: Designing a Stock Class

29. What is cohesion? How can you tell whether a class is cohesive?

30. Why didn't we choose to put the console I/O code into the `Stock` class?

31. Add accessor methods to the `Stock` class to return the stock's symbol, total shares, and total cost.

Exercises

1. Add the following accessor method to the `Point` class:

```
public int quadrant()
```

Returns which quadrant of the x/y plane the current `Point` object falls in. Quadrant 1 contains all points whose x and y values are both positive. Quadrant 2 contains all points with negative x but positive y. Quadrant 3 contains all

points with negative x and y values. Quadrant 4 contains all points with positive x but negative y . If the point lies directly on the x and/or y axis, return 0.

2. Add the following mutator method to the `Point` class:

```
public void flip()
```

Negates and swaps the x/y coordinates of the `Point` object. For example, if an object `pt` initially represents the point $(5, -3)$, after a call of `pt.flip()`; the object should represent $(3, -5)$. If the same object initially represents the point $(4, 17)$, after a call to `pt.flip()`; the object should represent $(-17, -4)$.

3. Add the following accessor method to the `Point` class:

```
public int manhattanDistance(Point other)
```

Returns the "Manhattan distance" between the current `Point` object and the given other `Point` object. The Manhattan distance refers to the distance between two places if one can travel between them only by moving horizontally or vertically, as though driving on the streets of Manhattan. In our case, the Manhattan distance is the sum of the absolute values of the differences in their coordinates; in other words, the difference in x plus the difference in y between the points.

4. Add the following accessor method to the `Point` class:

```
public boolean isVertical(Point other)
```

Returns `true` if the given `Point` lines up vertically with this `Point`, that is, if their x -coordinates are the same.

5. Add the following accessor method to the `Point` class:

```
public double slope(Point other)
```

Returns the slope of the line drawn between this `Point` and the given other `Point`. Use the formula $(y_2 - y_1) / (x_2 - x_1)$ to determine the slope between two points (x_1, y_1) and (x_2, y_2) . Note that this formula fails for points with identical x -coordinates, so throw an `IllegalArgumentException` in this case.

6. Add the following accessor method to the `Point` class:

```
public boolean isCollinear(Point p1, Point p2)
```

Returns whether this `Point` is collinear with the given two other `Points`. Points are collinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x - or y -coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and checking whether this slope is the same for all pairs of points. Use the formula $(y_2 - y_1) / (x_2 - x_1)$ to determine the slope between two points (x_1, y_1) and (x_2, y_2) . (Note that this formula fails for points with identical x -coordinates so this will have to be a special case in your code.) Since Java's `double` type is imprecise, round all slope values to a reasonable accuracy such as four digits past the decimal point before you compare them.

7. Add the following mutator method to the `TimeSpan` class:

```
public void add(TimeSpan span)
```

Adds the given amount of time to this time span.

8. Add the following mutator method to the `TimeSpan` class:

```
public void subtract(TimeSpan span)
```

Subtracts the given amount of time from this time span.

9. Add the following mutator method to the `TimeSpan` class:

```
public void scale(int factor)
```

Scales this time span by the given factor. For example, 1 hour and 45 minutes scaled by 2 equals 3 hours and 30 minutes.

10. Add the following mutator method to the `Stock` class:

```
public void clear()
```

Resets this `Stock`'s number of shares purchased and total cost to 0.

11. Suppose the following `BankAccount` class has been created:

```
1 // Each BankAccount object represents one user's account
2 // information including name and balance of money.
3 public class BankAccount {
4     String name;
5     double balance;
6
7     public void deposit(double amount) {
8         balance = balance + amount;
9     }
10
11    public void withdraw(double amount) {
12        balance = balance - amount;
13    }
14 }
```

Add a field to the `BankAccount` class named `transactionFee` for a real number representing an amount of money to deduct every time the user withdraws money. The default value is \$0.00, but the client can change the value.

Deduct the transaction fee money during every `withdraw` call (but not from deposits). Make sure that the balance cannot go negative during a withdrawal. If the withdrawal (amount plus transaction fee) would cause it to become negative, don't modify the balance at all.

12. Add a `toString` method to the `BankAccount` class from the previous exercise. Your method should return a string that contains the account's name and balance separated by a comma and space. For example, if an account object named `yana` has the name "Yana" and a balance of 3.03, the call `yana.toString()` should return the string "Yana, \$3.03".
13. Add a `transfer` method to the `BankAccount` class from the previous exercises. Your method should move money from the current bank account to another account. The method accepts two parameters: a second `BankAccount` to accept the money, and a real number for the amount of money to transfer. There is a \$5.00 fee for transferring money, so this much must be deducted from the current account's balance before any transfer. The method should modify the two `BankAccount` objects such that "this" current object has its balance decreased by the given amount plus the \$5 fee, and the other account's balance is increased by the given amount. If this account object does not have enough money to make the full transfer, transfer whatever money is left after the \$5 fee is deducted. If this account has under \$5 or the amount is 0 or less, no transfer should occur and neither account's state should be modified. The following are some example calls to the method:

```
BankAccount ben = new BankAccount();
ben.deposit(80.00);
BankAccount hal = new BankAccount();
hal.deposit(20.00);
ben.transfer(hal, 20.00); // ben $55, hal $40 (ben -$25, hal +$20)
ben.transfer(hal, 10.00); // ben $40, hal $50 (ben -$15, hal +$10)
hal.transfer(ben, 60.00); // ben $85, hal $ 0 (ben +$45, hal -$50)
```

14. Write a class called `Line` that represents a line segment between two `Points`. Your `Line` objects should have the following methods:

```
public Line(Point p1, Point p2)
```

Constructs a new `Line` that contains the given two `Points`.

```
public Point getP1()
```

Returns this `Line`'s first endpoint.

```
public Point getP2()
```

Returns this `Line`'s second endpoint.

```
public String toString()
```

Returns a `String` representation of this `Line`, such as `"[(22, 3), (4, 7)]"`.

15. Add the following accessor method to your `Line` class:

```
public double getSlope()
```

Returns the slope of this `Line`. The slope of a line between points (x_1, y_1) and (x_2, y_2) is equal to $(y_2 - y_1) / (x_2 - x_1)$. If x_2 equals x_1 the denominator is zero and the slope is undefined, so you may throw an exception in this case.

16. Add the following constructor to your `Line` class:

```
public Line(int x1, int y1, int x2, int y2)
```

Constructs a new `Line` that contains the given two `Points`.

17. Add the following accessor method to your `Line` class:

```
public boolean isCollinear(Point p)
```

Returns `true` if the given `Point` is collinear with the `Points` of this `Line`—that is, if, when this `Line` is stretched infinitely, it would eventually hit the given `Point`. Points are collinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x - or y -coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and checking whether this slope is the same for all pairs of points. Use the formula $(y_2 - y_1) / (x_2 - x_1)$ to determine the slope between two points (x_1, y_1) and (x_2, y_2) . (Note that this formula fails for points with identical x -coordinates, so this will have to be a special case in your code.) Since Java's `double` type is imprecise, round all slope values to a reasonable accuracy such as four digits past the decimal point before you compare them.

18. Write a class called `Rectangle` that represents a rectangular two-dimensional region. Your `Rectangle` objects should have the following methods:

```
public Rectangle(int x, int y, int width, int height)
```

Constructs a new `Rectangle` whose top-left corner is specified by the given coordinates and with the given width and height. Throw an `IllegalArgumentException` on a negative width or height.

```
public int getHeight()
```

Returns this `Rectangle`'s height.

```
public int getWidth()
```

Returns this `Rectangle`'s width.

```
public int getX()
```

Returns this `Rectangle`'s x -coordinate.

```
public int getY()
```


Returns this `Rectangle`'s y-coordinate.

```
public String toString()
```

Returns a `String` representation of this `Rectangle`, such as `"Rectangle[x=1,y=2,width=3,height=4]"`.

19. Add the following constructor to your `Rectangle` class:

```
public Rectangle(Point p, int width, int height)
```

Construct a new `Rectangle` whose top-left corner is specified by the given `Point` and with the given width and height.

20. Add the following accessor methods to your `Rectangle` class:

```
public boolean contains(int x, int y)
```

```
public boolean contains(Point p)
```

Returns whether the given `Point` or coordinates lie inside the bounds of this `Rectangle`.

21. Add the following method to your `Rectangle` class:

```
public Rectangle union(Rectangle rect)
```

Returns a new `Rectangle` that represents the area occupied by the tightest bounding box that contains both this `Rectangle` and the given other `Rectangle`.

22. Add the following method to your `Rectangle` class:

```
public Rectangle intersection(Rectangle rect)
```

Return a new `Rectangle` that represents the largest rectangular region completely contained within both this `Rectangle` and the given other `Rectangle`. If the `Rectangles` do not intersect at all, returns a `Rectangle` with width and height both equal to 0.

Programming Projects

1. Write a class called `RationalNumber` that represents a fraction with an integer numerator and denominator.

A `RationalNumber` object should have the following methods:

```
public RationalNumber(int numerator, int denominator)
```

Constructs a new rational number to represent the ratio (numerator/denominator). The denominator cannot be 0, so throw an `IllegalArgumentException` if 0 is passed.

```
public RationalNumber()
```

Constructs a new rational number to represent the ratio (0/1).

```
public int getDenominator()
```

Returns this rational number's denominator value; for example, if the ratio is (3/5), returns 5.

```
public int getNumerator()
```

Returns this rational number's numerator value; for example, if the ratio is (3/5), returns 3.

```
public String toString()
```

Returns a `String` representation of this rational number, such as `"3/5"`. You may wish to omit denominators of 1, returning `"4"` instead of `"4/1"`.

An extra challenge would be to maintain your `RationalNumber` objects in reduced form, avoiding rational numbers such as $3/6$ in favor of $1/2$, or avoiding $2/-3$ in favor of $-2/3$. Another possible extra feature would be methods to add, subtract, multiply, and divide two rational numbers.

2. Write a class called `Date` that represents a date consisting of a year, month, and day. A `Date` object should have the following methods:

```
public Date(int year, int month, int day)
```

Constructs a new `Date` object to represent the given date.

```
public void addDays(int days)
```

Moves this `Date` object forward in time by the given number of days.

```
public void addWeeks(int weeks)
```

Moves this `Date` object forward in time by the given number of seven-day weeks.

```
public int daysTo(Date other)
```

Returns the number of days that this `Date` must be adjusted to make it equal to the given other `Date`.

```
public int getDay()
```

Returns the day value of this date; for example, for the date 2006/07/22, returns 22.

```
public int getMonth()
```

Returns the month value of this date; for example, for the date 2006/07/22, returns 7.

```
public int getYear()
```

Returns the year value of this date; for example, for the date 2006/07/22, returns 2006.

```
public boolean isLeapYear()
```

Returns true if the year of this date is a leap year. A leap year occurs every four years, except for multiples of 100 that are not multiples of 400. For example, 1956, 1844, 1600, and 2000 are leap years, but 1983, 2002, 1700, and 1900 are not.

```
public String toString()
```

Returns a `String` representation of this date in year/month/day order, such as `"2006/07/22"`.

3. Write a class named `GroceryList` that represents a list of items to buy from the market, and another class named `GroceryItemOrder` that represents a request to purchase a particular item in a given quantity (example: four boxes of cookies). The `GroceryList` class should use an array field to store the grocery items and to keep track of its size (number of items in the list so far). Assume that a grocery list will have no more than 10 items. A `GroceryList` object should have the following methods:

```
public GroceryList()
```

Constructs a new empty grocery list.

```
public void add(GroceryItemOrder item)
```

Adds the given item order to this list if the list has fewer than 10 items.

```
public double getTotalCost()
```


Returns the total sum cost of all grocery item orders in this list.

The `GroceryItemOrder` class should store an item quantity and a price per unit. A `GroceryItemOrder` object should have the following methods:

```
public GroceryItemOrder(String name, int quantity, double pricePerUnit)
```

Constructs an item order to purchase the item with the given name, in the given quantity, which costs the given price per unit.

```
public double getCost()
```

Returns the total cost of this item in its given quantity. For example, four boxes of cookies that cost 2.30 per unit have a total cost of 9.20.

```
public void setQuantity(int quantity)
```

Sets this grocery item's quantity to be the given value.