CHAPTER 10

Object-Oriented Programming

In this chapter, you will learn about:

- The basic principles of object-oriented programming
- Classes
- Public and private access
- Different ways to organize classes
- Instance methods
- Static methods
- Using objects

Principles of Object-Oriented Programming

Object-oriented programming (**OOP**) is a style of programming that focuses on an application's data and the methods you need to manipulate that data. OOP uses all of the concepts you are familiar with from modular procedural programming, such as variables, methods, and passing values to methods. Methods in object-oriented programs continue to use sequence, selection, and looping structures and make use of arrays. However, object-oriented programming adds several new concepts to programming and involves a different way of thinking. A considerable amount of new vocabulary is involved as well. First, you will read about object-oriented programming concepts in general, and then you will learn the specific terminology.

Throughout this book, the terms *module* and *method* have been used interchangeably. In Chapter 9, you learned that object-oriented programmers prefer the term *method*, so it will be used in this chapter.

Five important features of object-oriented languages are:

- Classes
- Objects
- Polymorphism
- Inheritance
- Encapsulation

Classes and Objects

In object-oriented terminology, a **class** is a term that describes a group or collection of objects with common attributes. An **object** is one **instance** of a class. For example, your `redChevroletAutomobile WithTheDent` is an instance of the class that is made up of all automobiles, and your `GoldenRetrieverDogNamedGinger` is an instance of the class that is made up of all dogs. A class is like a blueprint from which many houses might be constructed, or like a recipe from which many meals can be prepared. One house and one meal are each an instance of their class.

Objects both in the real world and in object-oriented programming are made up of attributes and methods. **Attributes** are the characteristics that define an object as part of a class. For example, some of your automobile's attributes are its make, model, year, and purchase price. Other attributes include whether the automobile is currently running, its gear, its speed, and whether it is dirty. All automobiles possess the same attributes, but not, of course, the same values for those attributes. Similarly, your dog has the attributes of its breed, name, age, and whether its shots are current. Methods are the actions that alter, use, or retrieve the attributes. For example, an automobile

has methods for changing and discovering its speed, and a dog has methods for setting and finding out its shot status.

Thinking of items as instances of a class allows you to apply your general knowledge of the class to its individual members. A particular instance of an object takes its attributes from the general category. If your friend purchases an `Automobile`, you know it has a model name, and if your friend gets a `Dog`, you know the dog has a breed. You might not know the current state of your friend's `Automobile`, such as its current speed, or the status of her `Dog`'s shots, but you do know what attributes exist for the `Automobile` and `Dog` classes, which allows you to imagine these objects reasonably well before you see them. You know enough to ask the `Automobile`'s model, and not its breed; you know enough to ask the `Dog`'s name and not its engine size. As another example, when you use a new application on your computer, you expect each component to have specific, consistent attributes, such as a button being clickable or a window being closeable, because each component gains these attributes as a member of the general class of GUI (graphical user interface) components.

Much of your understanding of the world comes from your ability to categorize objects and events into classes. As a young child, you learned the concept of "animal" long before you knew the word. Your first encounter with an animal might have been with the family dog, a neighbor's cat, or a goat at a petting zoo. As you developed speech, you might have used the same term for all of these creatures, gleefully shouting "Doggie!" as your parents pointed out cows, horses, and sheep in picture books or along the roadside on drives in the country. As you grew more sophisticated, you learned to distinguish dogs from cows; still later, you learned to distinguish breeds. Your understanding of the class "animal" helps you see the similarities between dogs and cows, and your understanding of the class "dog" helps you see the similarities between a Great Dane and a Chihuahua. Understanding classes gives you a framework for categorizing new experiences. You might not know the term "okapi," but when you learn it's an animal, you begin to develop a concept of what an okapi might be like.

When you think in an object-oriented manner, everything is an object, and every object is a member of a class. You can think of any inanimate physical item as an object—your desk, your computer, and your house are all called "objects" in everyday conversation. You can think of living things as objects, too—your houseplant, your pet fish, and your sister are objects. Events are also objects—the stock purchase you made, the mortgage closing you attended, and your graduation party are all objects.

Most programmers who use more modern languages employ the format in which class names begin with an uppercase letter and multiple-word identifiers are run together, such as `SavingsAccount` or `TemporaryWorker`. Each new word within the identifier starts with an uppercase letter. In Chapter 2, you learned that this format is known as *Pascal casing*.

Everything is an object, and every object is a member of a more general class. Your desk is a member of the class that includes all desks, and your pet fish is a member of the class that contains all fish. An object-oriented programmer would say that the desk in your office is an instance, or one tangible example, of the `Desk` class and your fish is an instance of the `Fish` class. These statements represent **is-a relationships** because you can say, "My oak desk with the scratch on top *is a* `Desk` and my goldfish named Moby *is a* `Fish`." Your goldfish, my guppy, and the zoo's shark each constitute one instance of the `Fish` class.

Object-oriented programmers also use the term *is-a* when describing inheritance. You will learn about inheritance later in this chapter and in Chapter 11.

The concept of a class is useful because of its reusability. For example, if you invite me to a graduation party, I automatically know many things about the object (the party). I assume there will be attributes such as a starting time, a number of guests, some quantity of food, and some nature of gifts. I understand parties because of my previous knowledge of the `Party` class, of which all parties are members. I don't know the number of guests or the date or time of this particular party, but I understand that because all parties have a date and time, then this one must as well. Similarly, even though every stock purchase is unique, each must have a dollar amount and a number of shares. All objects have predictable attributes because they are members of certain classes.

Object-oriented programmers sometimes say an object is one **instantiation** of a class; this is just another form of *instance*.

The data components of a class that belong to every instantiated object are the class's **instance variables**. Also, object attributes are often called **fields** to help distinguish them from other variables you might use. The set of all the values or contents of a class object's instance variables is known as its **state**. For example, the current state of a particular party might be 8 p.m. and Friday; the state of a particular stock purchase might be $10 and five shares.

In addition to their attributes, class objects have methods associated with them, and every object that is an instance of a class possesses the same methods. For example, at some point you might want to issue invitations for a party. You might name the method `issueInvitations()`, and it might display some text as well as the values of the party's date and time fields. Your graduation party, then, might possess the identifier `myGraduationParty`. As a member of the `Party` class, it might have data members for the date and time, like all parties, and it might have a method to issue invitations. When you use the method, you might want to be able to send an argument to `issueInvitations()` that indicates how many copies to print. When you think of an object and its methods, it's as though you can send a message to the object to direct it to accomplish a particular task—you can tell the party object named `myGraduationParty` to print the number of invitations you request. Even though `yourAnniversaryParty` is also a member of the `Party`

In grammar, a noun is equivalent to an object and the values of a class's attributes are adjectives—they describe the characteristics of the objects. An object can also have methods, which are equivalent to verbs.

class, and even though it also has an `issueInvitations()` method, you will send a different argument value to `yourAnniversaryParty`'s `issueInvitations()` method than I send to `myGraduationParty`'s corresponding method. Within any object-oriented program, you continuously make requests to objects' methods, often including arguments as part of those requests.

When you program in object-oriented languages, you frequently create classes from which objects will be instantiated. You also write applications to use the objects, along with their data and methods. Often, you will write programs that use classes created by others; other times, you might create a class that other programmers will use to instantiate objects within their own programs. A program or class that instantiates objects of another prewritten class is a **class client** or **class user**. For example, your organization might already have written a class named `Customer` that contains attributes such as `name`, `address`, and `phoneNumber`, and you might create clients that include arrays of thousands of `Customers`. Similarly, in a GUI operating environment, you might write applications that include prewritten components that are members of classes with names like `Window` and `Button`. You expect each component on a GUI screen to have specific, consistent attributes because each component gains these attributes as a member of its general class.

Polymorphism

The real world is full of objects. Consider a door. A door needs to be opened and closed. You open a door with an easy-to-use interface known as a doorknob. Object-oriented programmers would say you are "passing a message" to the door when you "tell" it to open by turning its knob. The same message (turning a knob) has a different result when applied to your radio than when applied to a door. The procedure you use to open something—call it the "open" procedure—works differently on a door to a room than it does on a desk drawer, a bank account, a computer file, or your eyes. However, even though these procedures operate differently using the different objects, you can call each of these procedures "open." In object-oriented programming, procedures are called methods.

With object-oriented programming, you focus on the objects that will be manipulated by the program—for example, a customer invoice, a loan application, or a menu from which the user will select an option. You define the characteristics of those objects and the methods each of the objects will use; you also define the information that must be passed to those methods.

You can create multiple methods with the same name, which will act differently and appropriately when used with different types of objects. In Chapter 9, you learned that this concept is called *polymorphism*, and you learned to overload methods. For example, you might use a method named `print()` to print a customer invoice, loan application, or envelope. Because you use the same method name, `print()`, to describe the different actions needed to print these diverse objects, you can write statements in object-oriented programming languages that are more like English; you can use the same method name to describe the same type of action, no matter what type of object is being acted upon. Using the method name `print()` is easier than remembering `printInvoice()`, `printLoanApplication()`, and so on. In English, you understand the difference between "running a race," "running a business," and "running a computer program." Object-oriented languages understand verbs in context, just as people do.

As another example of the advantages to using one name for a variety of objects, consider a screen you might design for a user to enter data into an application you are writing. Suppose the screen contains a variety of objects—some forms, buttons, scroll bars, dialog boxes, and so on. Suppose also that you decide to make all the objects blue. Instead of having to memorize the method names that these objects use to change color—perhaps `changeFormColor()`, `changeButtonColor()`, and so on—your job would be easier if the creators of all those objects had developed a `setColor()` method that works appropriately with each type of object.

Purists find a subtle difference between overloading and polymorphism. Some reserve the term *polymorphism* (or **pure polymorphism**) for situations in which one method body is used with a variety of arguments. For example, a single method that can be used with any type of object is polymorphic. The term *overloading* is applied to situations in which you define multiple functions with a single name—for example, three functions, all named `display()`, that display a number, an employee, and a student, respectively. Certainly, the two terms are related; both refer to the ability to use a single name to communicate multiple meanings. For now, think of overloading as a primitive type of polymorphism.

Inheritance

Another important concept in object-oriented programming is **inheritance**, which is the process of acquiring the traits of one's predecessors. In the real world, a new door with a stained glass window inherits most of its traits from a standard door. It has the same purpose, it opens and closes in the same way, and it has the

Watch the video *An Introduction to Object-Oriented Programming*.

same knob and hinges. The door with the stained glass window simply has one additional trait—its window. Even if you have never seen a door with a stained glass window, when you encounter one you know what it is and how to use it because you understand the characteristics of all doors. With object-oriented programming, once you create an object, you can develop new objects that possess all the traits of the original object plus any new traits you desire. If you develop a `CustomerBill` class of objects, there is no need to develop an `OverdueCustomerBill` class from scratch. You can create the new class to contain all the characteristics of the already developed one, and simply add necessary new characteristics. This not only reduces the work involved in creating new objects, it makes them easier to understand because they possess most of the characteristics of already developed objects.

Encapsulation

Real-world objects often employ encapsulation and information hiding. **Encapsulation** is the process of combining all of an object's attributes and methods into a single package. **Information hiding** is the concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege. Outside classes should only be allowed to make a request that an attribute be altered; then it is up to the class's methods to determine whether the request is appropriate. When using a door, you usually are unconcerned with the latch or hinge construction features, and you don't have access to the interior workings of the knob or know what color of paint might have been used on the inside of the door panel. You care only about the functionality and the interface, the user-friendly boundary between the user and internal mechanisms of the device. Similarly, the detailed workings of objects you create within object-oriented programs can be hidden from outside programs and modules if you want them to be. When the details are hidden, programmers can focus on the functionality and the interface, as people do with real-life objects.

Outside classes make requests to alter an attribute by using a set method. You will learn more about set methods later in this chapter.

Information hiding is also called **data hiding**.

In summary, understanding object-oriented programming means that you must consider five of its integral components: classes, objects, polymorphism, inheritance, and encapsulation.

TWO TRUTHS & A LIE

Principles of Object-Oriented Programming

1. Learning about object-oriented programming is difficult because it does not use the concepts you already know, such as declaring variables and using modules.
2. In object-oriented terminology, a class is a term that describes a group or collection of objects with common attributes; an instance of a class is an existing object of a class.
3. A program or class that instantiates objects of another prewritten class is a class client or class user.

The false statement is #1. Object-oriented programming makes use of many of the features of procedural programming, including declaring variables and using modules.

Defining Classes and Creating Class Diagrams

A class is a category of things; an object is a specific instance of a class. A **class definition** is a set of program statements that tell you the characteristics of the class's objects and the methods that can be applied to its objects.

A class definition can contain three parts:

- Every class has a name.
- Most classes contain data, although this is not required.
- Most classes contain methods, although this is not required.

For example, you can create a class named `Employee`. Each `Employee` object will represent one employee who works for an organization. Data members, or attributes of the `Employee` class, include fields such as `lastName`, `hourlyWage`, and `weeklyPay`.

The methods of a class include all actions you want to perform with the class. Appropriate methods for an `Employee` class might include `setHourlyWage()`, `getHourlyWage()`, and `calculateWeeklyPay()`. The job of `setHourlyWage()` is to provide values for an `Employee`'s wage data field, the purpose of `getHourlyWage()` is to retrieve the wage value, and the purpose of `calculateWeeklyPay()` is to multiply the `Employee`'s `hourlyWage` by the number of hours in a workweek to

calculate a weekly salary. With object-oriented languages, you think of the class name, data, and methods as a single encapsulated unit.

Declaring a class does not create any actual objects. A class is just an abstract description of what an object will be like if any objects are ever actually instantiated. Just as you might understand all the characteristics of an item you intend to manufacture long before the first item rolls off the assembly line, you can create a class with fields and methods long before you instantiate any objects that are members of that class. After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. When you declare a simple variable that is a built-in data type, you write a statement such as one of the following:

```
num money
string name
```

When you write a program that declares an object that is a class data type, you write a statement such as the following:

```
Employee myAssistant
```

In some object-oriented programming languages, you need to add more to the declaration statement to actually create an `Employee` object. For example, in Java, you would write:

```
Employee myAssistant = new Employee();
```

You will understand more about the format of this statement when you learn about constructors in Chapter 11.

When you declare the `myAssistant` object, it contains all the data fields and has access to all the methods contained within the class. In other words, a larger section of memory is set aside than when you declare a simple variable, because an `Employee` contains several fields. You can use any of an `Employee`'s methods with the `myAssistant` object. The usual syntax is to provide an object name, a dot (period), and a method name. For example, you can write a program that contains statements such as the ones shown in the pseudocode in Figure 10-1.

Of course, the program segment in Figure 10-1 is very short. In a more useful real-life program, you might read employee data from a data file before assigning it to the object's fields, each `Employee` might contain dozens of fields, and your application might create hundreds or thousands of objects.

```
start
   Declarations
      Employee myAssistant
   myAssistant.setLastName("Reynolds")
   myAssistant.setHourlyWage(16.75)
   output "My assistant makes ",
      myAssistant.getHourlyWage(), " per hour"
stop
```

Figure 10-1 Application that declares and uses an `Employee` object

When you write a statement such as `myAssistant.setHourlyWage(16.75)`, you are making a call to a method that is contained within the `Employee` class. Because `myAssistant` is an `Employee` object, it is allowed to use the `setHourlyWage()` method that is part of its class.

When you write the application in Figure 10-1, you do not need to know what statements are written within the `Employee` class methods, although you could make an educated guess based on the methods' names. Before you could execute the application in Figure 10-1, someone would have to write appropriate statements within the `Employee` class methods. If you wrote the methods, of course you would know their contents, but if another programmer has already written the methods, you could use the application without knowing the details contained in the methods. In Chapter 9, you learned that the ability to use methods without knowing the details of their contents (called a "black box") is a feature of encapsulation. The real world is full of many black box devices. For example, you can use your television and microwave oven without knowing how they work internally—all you need to understand is the interface. Similarly, with well-written methods that belong to classes you use, you need not understand how they work internally to be able to use them; you need only understand the ultimate result when you use them.

In the client program segment in Figure 10-1, the focus is on the object—the `Employee` named `myAssistant`—and the methods you can use with that object. This is the essence of object-oriented programming.

In older object-oriented programming languages, simple numbers and characters are said to be **primitive data types**; this distinguishes them from objects that are class types. In the newest programming languages, every item you name, even one that is a numeric or string type, is an object that is a member of a class that contains both data and methods.

Creating Class Diagrams

Programmers often use a class diagram to illustrate class features or to help plan them. A **class diagram** consists of a rectangle divided into three sections, as shown in Figure 10-2. The top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods. This generic class diagram shows two attributes and three methods, but a given class might have any number of either, including none. Figure 10-3 shows the class diagram for the `Employee` class.

Besides referring to `Employee` as a class, many programmers would refer to it as a **user-defined type**; a more accurate term is **programmer-defined type**. Object-oriented programmers typically refer to a class like `Employee` as an **abstract data type** (**ADT**); this term implies that the type's data can be accessed only through methods.

Some programmers write only client programs, never creating nonclient classes themselves, but using only classes that others have created.

When you instantiate objects, the data fields of each are stored at separate memory locations. However, all members of the same class share one copy of the class's methods. You will learn more about this concept later in this chapter.

Class diagrams are a type of Unified Modeling Language (UML) diagram. Chapter 13 covers the UML.

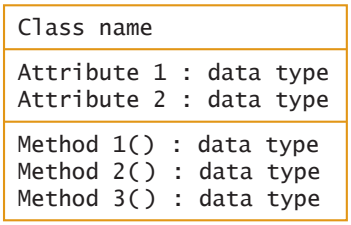

Figure 10-2 Generic class diagram

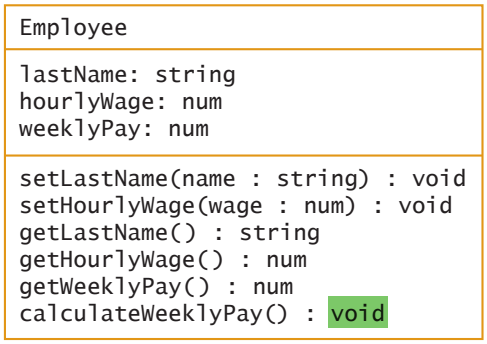

Figure 10-3 `Employee` class diagram

By convention, a class diagram lists the names of the data items first. Each name is followed by a colon and the data type. Similarly, method names are followed by their data types. Listing the names first emphasizes the purposes of the fields and methods more than their types.

Class diagrams are useful for communicating about a class's contents with nonprogrammers.

Figures 10-2 and 10-3 both show that a class diagram is intended to be only an overview of class attributes and methods. A class diagram shows *what* data items and methods the class will use, not the details of the methods nor *when* they will be used. It is a design tool that helps you see the big picture in terms of class requirements. Figure 10-3 shows the `Employee` class that contains three data fields that represent an employee's name, hourly pay rate, and weekly pay amount. Every `Employee` object created in any program that uses this class will contain these three data fields. In other words, when you declare an `Employee` object, you declare three fields with one statement and reserve enough memory to hold all three fields.

Figure 10-3 also shows that the `Employee` class contains six methods. For example, the first method is defined as follows:

```
setLastName(name : string) : void
```

This notation means that the method name is `setLastName()`, that it takes a single `string` parameter named `name`, and that it returns nothing.

Various books, Web sites, and organizations will use class diagrams that list the method name only, as in the following:

```
setLastName()
```

Some developers choose to list the method name and return type only, as in:

```
setLastName() : void
```

Some developers list the method name and return type as well as the parameter type, as in the following:

```
setLastName(string) : void
```

Still other developers list an appropriate identifier for the parameter, but not its type, as in the following:

```
setLastName(name) : void
```

This book will take the approach of being as complete as possible, so the class diagrams you see here will contain each method's identifier, parameter list with types, and return type. You should use the format your instructor prefers. When you are on the job, use the format your supervisor and coworkers understand.

The `Employee` class diagram shows that two of the six methods take parameters (`setLastName()` and `setHourlyWage()`). The diagram also shows the return type for each method—three void methods, two numeric methods, and one string method. The class diagram does not tell you what takes place inside the method (although you might be able to make an educated guess). Later, when you write the code that actually creates the `Employee` class, you include method implementation details. For example, Figure 10-4 shows some pseudocode you can use to show the details for the methods contained within the `Employee` class.

```
class Employee
   Declarations
      string lastName
      num hourlyWage
      num weeklyPay

   void setLastName(string name)
      lastName = name
   return

   void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   string getLastName()
   return lastName

   num getHourlyWage()
   return hourlyWage

   num getWeeklyPay()
   return weeklyPay

   void calculateWeeklyPay()
      Declarations
         num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

Figure 10-4 Pseudocode for `Employee` class described in the class diagram in Figure 10-3

In Figure 10-4, the `Employee` class attributes or fields are identified with a data type and a field name. In addition to listing the data fields required, Figure 10-4 shows the complete methods for the `Employee` class. The purposes of the methods can be divided into three categories:

- Two of the methods accept values from the outside world; these methods, by convention, start with the prefix *set*. These methods are used to set the data fields in the class.
- Three of the methods send data to the outside world; these methods, by convention, start with the prefix *get*. These methods return field values to a client program.
- One method performs work within the class; this method is named `calculateWeeklyPay()`. This method does not communicate with the outside; its purpose is to multiply `hourlyWage` by the number of hours in a week.

The Set Methods

In Figure 10-4, two of the methods begin with the word *set*; they are `setLastName()` and `setHourlyWage()`. They are known as **set methods** because their purpose is to set the values of data fields within the class. Each accepts data from the outside and assigns it to a field within the class. There is no requirement that such methods start with the prefix *set*; the prefix is merely conventional and clarifies the intention of the methods. The method `setLastName()` is implemented as follows:

```
void setLastName(string name)
   lastName = name
return
```

In this method, a string `name` is passed in as a parameter and assigned to the field `lastName`. Because `lastName` is contained in the same class as this method, the method has access to the field and can alter it.

Similarly, the method `setHourlyWage()` accepts a numeric parameter and assigns it to the class field `hourlyWage`. This method also calls the `calculateWeeklyPay()` method, which sets `weeklyPay` based on `hourlyWage`. By writing the `setHourlyWage()` method to call the `calculateWeeklyPay()` method automatically, you guarantee that the `weeklyPay` field is updated any time `hourlyWage` changes.

Methods that set values are called **mutator methods**.

When you create an `Employee` object with a statement such as `Employee mySecretary`, you can use statements such as the following:

```
mySecretary.setLastName("Johnson")
mySecretary.setHourlyWage(15.00)
```

Similarly, you could pass variables or named constants to the methods as long as they were the correct data type. For example, if you write a program in which you make the following declaration, then the assignment in the next statement is valid.

```
num PAY_RATE_TO_START = 8.00
mySecretary.setHourlyWage(PAY_RATE_TO_START)
```

In some languages—for example, Visual Basic and C#—you can create a **property** instead of creating a set method. Using a property provides a way to set a field value using a simpler syntax. By convention, if a class field is `hourlyWage`, its property would be `HourlyWage`, and in a program you could make a statement similar to `mySecretary.HourlyWage = PAY_RATE_TO_START`. The implementation of the property `HourlyWage` (with an uppercase initial letter) would be written in a format very similar to that of the `setHourlyWage()` method.

Just like any other methods, the methods that manipulate fields within a class can contain any statements you need. For example, a more complicated `setHourlyWage()` method might be written as in Figure 10-5. In this version, the wage passed to the method is tested against minimum and maximum values, and is assigned to the class field `hourlyWage` only if it falls within the prescribed limits. If the wage is too low, the `MINWAGE` value is substituted, and if the wage is too high, the `MAXWAGE` value is substituted.

```
void setHourlyWage(num wage)
   Declarations
      num MINWAGE = 6.00
      num MAXWAGE = 70.00
   if wage < MINWAGE then
      hourlyWage = MINWAGE
   else
      if wage > MAXWAGE then
         hourlyWage = MAXWAGE
      else
         hourlyWage = wage
      endif
   endif
   calculateWeeklyPay()
return
```

Figure 10-5 More complex `setHourlyWage()` method

Similarly, if the set methods in a class required them, the methods could contain output statements, loops, array declarations, or any other legal programming statements. However, if the main purpose of a method is not to set a field value, then the method should not be named with the set prefix.

The Get Methods

In the `Employee` class in Figure 10-4, three of the methods begin with the prefix *get*: `getLastName()`, `getHourlyWage()`, and `getWeeklyPay()`. The purpose of a **get method** is to return a value to the world outside the class. The methods are implemented as follows:

```
string getLastName()
return lastName

num getHourlyWage()
return hourlyWage

num getWeeklyPay()
return weeklyPay
```

Each of these methods simply returns the value in the field implied by the method name. Like set methods, any of these get methods could also contain more complicated statements as needed. For example, in a more complicated class, you might want to return the hourly wage of an employee only if the user had also passed an appropriate access code to the method, or you might want to return the weekly pay value as a string with a dollar sign attached instead of as a numeric value.

Methods that get values from class fields are known as **accessor methods**.

When you declare an `Employee` object such as `Employee mySecretary`, you can then make statements in a program similar to the following:

```
Declarations
   string employeeName
employeeName = mySecretary.getLastName()
output "Wage is ", mySecretary.getHourlyWage()
output "Pay for half a week is ", mySecretary.
   getWeeklyPay() * 0.5
```

In other words, the value returned from a get method can be used as any other variable of its type would be used. You can assign the value to another variable, display it, perform arithmetic with it, or make any other statement that works correctly with the returned data type.

In some languages—for example, Visual Basic and C#—instead of creating a get method, you can add statements to the property to return a value using simpler syntax. For example, if you create an `HourlyWage` property, you could write a program that contains the statement `output mySecretary.HourlyWage`.

Work Methods

The `Employee` class in Figure 10-4 contains one method that is neither a get nor a set method. This method, `calculateWeeklyPay()`, is a **work method** within the class. It contains a locally named constant that

represents the hours in a standard workweek, and it computes the `weeklyPay` field value by multiplying `hourlyWage` by the named constant. The method is written as follows:

```
void calculateWeeklyPay()
   Declarations
      num WORK_WEEK_HOURS = 40
   weeklyPay = hourlyWage * WORK_WEEK_HOURS
return
```

No values need to be passed into this method, and no value is returned from it because this method does not communicate with the outside world. Instead, this method is called only from within another method in the same class (the `setHourlyWage()` method), and that method is called from the outside world. Any time a program uses the `setHourlyWage()` method to alter an `Employee`'s `hourlyWage` field, `calculateWeeklyPay()` is called to recalculate the `weeklyPay` field.

No `setWeeklyPay()` method is included in this `Employee` class because the intention is that `weeklyPay` is set only each time the `setHourlyWage()` method is used. If you wanted programs to be able to set the `weeklyPay` field directly, you would have to write a method to allow it.

For example, Figure 10-6 shows a program that declares an `Employee` object and sets the hourly wage value. The program displays the `weeklyPay` value. Then a new value is assigned to `hourlyWage` and `weeklyPay` is displayed again. As you can see from the output in Figure 10-7, the `weeklyPay` value has been recalculated even though it was never set directly by the client program.

```
start
   Declarations
      num LOW = 9.00
      num HIGH = 14.65
      Employee myGardener
   myGardener.setLastName("Greene")
   myGardener.setHourlyWage(LOW)
   output "My gardener makes ",
     myGardener.getWeeklyPay(), " per week"
   myGardener.setHourlyWage(HIGH)
   output "My gardener makes ",
     myGardener.getWeeklyPay(), " per week"
stop
```

Figure 10-6 Program that sets and displays `Employee` data two times

Some programmers call work methods **help methods** or **facilitators**.

Programmers who are new to class creation often are tempted to pass the `hourlyWage` value into the `setWeeklyPay()` method so it can use the value in its calculation. Although this technique would work, it is not required. The `setWeeklyPay()` method has direct access to the `hourlyWage` field by virtue of being a member of the same class.

Figure 10-7 Execution of program in Figure 10-6

TWO TRUTHS & A LIE

Defining Classes and Creating Class Diagrams

1. Every class has a name, data, and methods.
2. After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call.
3. A class diagram consists of a rectangle divided into three sections; the top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods.

The false statement is #1. Most classes contain data and methods, although neither is required.

Understanding Public and Private Access

When you buy a product with a warranty, one of the common conditions of the warranty is that the manufacturer must perform all repair work. For example, if your computer has a warranty and something goes wrong with its operation, you cannot open the system unit yourself, remove and replace parts, and then expect to get your money back for a device that does not work properly. Instead, when something goes wrong with your computer, you must take the device to a technician approved by the manufacturer. The manufacturer guarantees that your machine will work properly only if the manufacturer can control how the internal mechanisms of the machine are modified.

Similarly, in object-oriented design, usually you do not want any outside programs or methods to alter your class's data fields unless you have control over the process. For example, you might design a class that performs a complicated statistical analysis on some data and stores the result. You would not want others to be able to alter your carefully crafted result. As another example, you might design a class from which others can create an innovative and useful GUI

screen object. In this case you would not want others altering the dimensions of your artistic design. To prevent outsiders from changing your data fields in ways you do not endorse, you force other programs and methods to use a method that is part of the class, such as `setLastName()` and `setHourlyWage()`, to alter data. (Earlier in this chapter, you learned that the principle of keeping data private and inaccessible to outside classes is known as information hiding or data hiding.) Object-oriented programmers usually specify that their data fields will have **private access**—that is, the data cannot be accessed by any method that is not part of the class. The methods themselves, like `setHourlyWage()`, support **public access**—other programs and methods may use the methods that control access to the private data. Figure 10-8 shows a complete `Employee` class to which access specifiers have been added to describe each attribute and method. An **access specifier** (or **access modifier**) is the adjective defining the type of access (`public` or `private`) that outside classes will have to the attribute or method. In the figure, each access specifier is shaded.

```
class Employee
   Declarations
      private string lastName
      private num hourlyWage
      private num weeklyPay

   public void setLastName(string name)
      lastName = name
   return

   public void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   public string getLastName()
   return lastName

   public num getHourlyWage()
   return hourlyWage

   public num getWeeklyPay()
   return weeklyPay

   private void calculateWeeklyPay()
      Declarations
         num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

Figure 10-8 `Employee` class including `public` and `private` access specifiers

In many object-oriented programming languages, if you do not declare an access specifier for a data field or method, then it is private by default. This book will follow the convention of explicitly specifying access for every class member.

In Figure 10-8, each of the data fields is private; that means each field is inaccessible to an object declared in a program. In other words, if a program declares an `Employee` object, such as `Employee myAssistant`, then the following statement is illegal:

```
myAssistant.hourlyWage = 15.00
```

Don't Do It
You cannot assign a value to a private variable using a statement in another class.

Instead, `hourlyWage` can be assigned only through a public method as follows:

```
myAssistant.setHourlyWage(15.00)
```

If you made `hourlyWage` public instead of private, then a direct assignment statement would work, but you would violate an important principle of OOP—that of data hiding using encapsulation. Data fields should usually be private and a client application should be able to access them only through the public interfaces; that is, through the class's public methods. That way, if you have restrictions on the value of `hourlyWage`, those restrictions will be enforced by the public method that acts as an interface to the private data field. Similarly, a public get method might control how a private value is retrieved. Perhaps you do not want clients to have access to an `Employee`'s `hourlyWage` if it is more than a specific value, or perhaps you always want to return it to the client as a string with a dollar sign attached. Even when a field has no data value requirements or restrictions, making data private and providing public set and get methods establishes a framework that makes such modifications easier in the future.

Watch the video *Creating a Class*.

In some object-oriented programming languages, such as C++, you can label a set of data fields or methods as public or private using the access specifier name just once, then following it with a list of the items in that category. In other languages, such as Java, you use the specifier public or private with each field or method. For clarity, this book will label each field and method as public or private.

In the `Employee` class in Figure 10-8, only one method is not public; the `calculateWeeklyPay()` method is private. That means if you write a program and declare an `Employee` object such as `Employee myAssistant`, then the following statement is not permitted:

```
myAssistant.calculateWeeklyPay()
```

Don't Do It
The `calculateWeeklyPay()` method is not accessible outside the class.

Because it is private, the only way to call the `calculateWeeklyPay()` method is from within another method that already belongs to the class. In this example, it is called from the `setHourlyWage()` method. This prevents any client program from setting `hourlyWage` to one value while setting `weeklyPay` to some incompatible value. By making the `calculateWeeklyPay()` method private, you ensure that the class retains full control over when and how it is used. Classes most often contain private data and public methods, but as you have just seen, they can contain private methods. Classes can contain public data items as well. For example, an `Employee` class might contain a

public constant data field named MINIMUM_WAGE; outside programs then would be able to access that value without using a method. Public data fields are not required to be named constants, but they frequently are.

Many programmers like to specify in their class diagrams whether each component in a class is public or private. Figure 10-9 shows the conventions that are typically used. A minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.

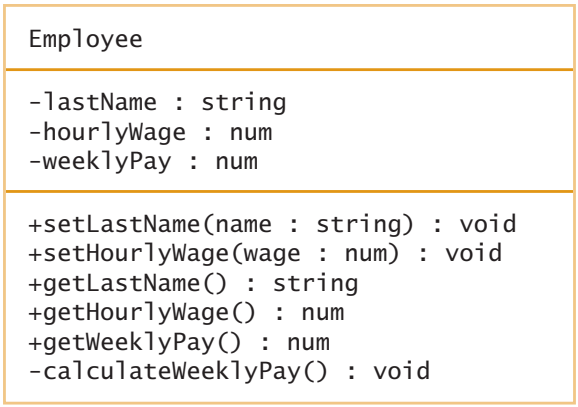

Figure 10-9 Employee class diagram with public and private access specifiers

When you learn more about inheritance in Chapter 11, you will learn about an additional access specifier—the protected access specifier. You use an octothorpe, also called a pound sign or number sign (#), to indicate protected access.

In object-oriented programming languages, the main program is most often written as a method named main() or Main(), and that method is virtually always defined as public.

TWO TRUTHS & A LIE

Understanding Public and Private Access

1. Object-oriented programmers usually specify that their data fields will have private access.
2. Object-oriented programmers usually specify that their methods will have private access.
3. In a class diagram, a minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.

The false statement is #2. Object-oriented programmers usually specify that their methods will have public access.

Organizing Classes

The `Employee` class in Figure 10-9 contains just three data fields and six methods; most classes you create for professional applications will have many more. For example, in addition to requiring a last name and pay information, real employees require an employee number, a first name, address, phone number, hire date, and so on, as well as methods to set and get those fields. As classes grow in complexity, deciding how to organize them becomes increasingly important.

A unique identifier is one that should have no duplicates within an application. For example, an organization might have many employees with the last name Johnson or an hourly wage of $10.00, but only one employee will have employee number 12438.

Although there is no requirement to do so, most programmers place data fields in some logical order at the beginning of a class. For example, an ID number is most likely used as a unique identifier for each employee (what database users often call a **primary key**), so it makes sense to list the employee ID number first in the class. An employee's last name and first name "go together," so it makes sense to store these two `Employee` components adjacently. Despite these common-sense rules, you have a lot of flexibility in how you position your data fields within any class. For example, depending on the class, you might choose to store the data fields alphabetically, or you might choose to group together all the fields that are the same data type. Alternatively, you might choose to store all public data items first, followed by private ones, or vice versa.

In some languages you can organize a class's data fields and methods in any order within a class. For example, you could place all the methods first, followed by all the data fields, or you could organize the class so that several data fields are followed by methods that use them, and then several more data fields might be followed by the methods that use them. This book will follow the convention of placing all data fields first so that you can see their names and data types before reading the methods that use them. This format also echoes the way data and methods appear in standard class diagrams.

For ease in locating a class's methods, many programmers store them in alphabetical order. Other programmers arrange them in pairs of get and set methods, in the same order as the data fields are defined. Another option is to list all accessor (get) methods together and all mutator (set) methods together. Depending on the class, there might be other orders that result in logically functional groupings. Of course, if your company distributes guidelines for organizing class components, you must follow those rules.

TWO TRUTHS & A LIE

Organizing Classes

1. As classes grow in complexity, deciding how to organize them becomes increasingly important.
2. You have a lot of flexibility in how you position your data fields within any class.
3. In a class, methods must be stored in the order in which they are used.

The false statement is #3. Methods can be stored in alphabetical order, in pairs of get and set methods, in the same order as the data fields are defined, or in any other logically functional groupings.

Understanding Instance Methods

Class objects have data and methods associated with them, and every object that is an instance of a class is assumed to possess the same data and have access to the same methods. For example, Figure 10-10 shows a class diagram for a simple `Student` class containing just one private data field that holds a student's grade point average. The class also contains get and set methods for the field. Figure 10-11 shows the pseudocode for the `Student` class. This class becomes the model for a new data type named `Student`; when `Student` objects eventually are created, each will have its own `gradePointAverage` field and have access to methods to get and set it.

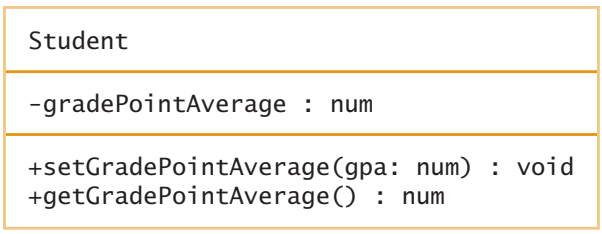

Figure 10-10 Class diagram for `Student` class

```
class Student
   Declarations
      private num gradePointAverage

   public void setGradePointAverage(num gpa)
      gradePointAverage = gpa
   return

   public num getGradePointAverage()
   return gradePointAverage
endClass
```

Figure 10-11 Pseudocode for the `Student` class

In the `Student` class in Figure 10-11, the method `setGradePointAverage()` takes one argument—a value for the `Student`'s grade point average. The identifier `gpa` is local to the `setGradePointAverage()` method, and holds a value that will come into the method from the outside. Within the method, the value in `gpa` is assigned to `gradePointAverage`, which is a field within the class. The `setGradePointAverage()` method assigns a value to the `gradePointAverage` field for each separate `Student` object you create. Therefore, a method such as `setGradePointAverage()` is called an **instance method** because it operates correctly yet differently (using different values) for each separate instance of the `Student` class. In other words, if you create 100 `Students` and assign grade point averages to each of them, you need 100 storage locations in computer memory to store each unique grade point average.

Figure 10-12 shows a program that creates three `Student` objects and assigns values to their `gradePointAverage` fields. It also shows how the `Student` objects look in memory after the values have been assigned.

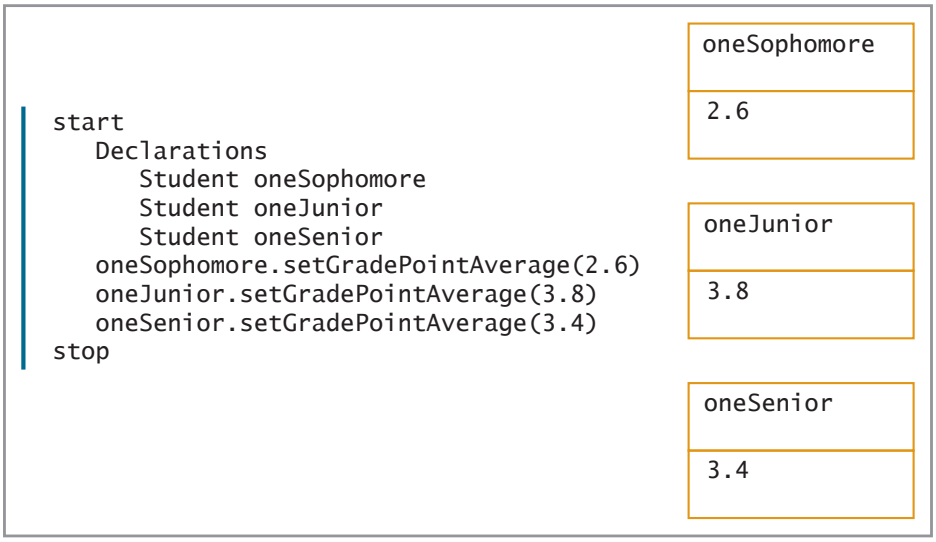

Figure 10-12 `StudentDemo` program and how `Student` objects look in memory

It makes sense for each `Student` object in Figure 10-12 to have its own `gradePointAverage` field, but it does not make sense for each `Student` to have its own copy of the methods that get and set `gradePointAverage`. Any method might have dozens of instructions in it, and to make 100 copies of identical methods would be inefficient. Instead, even though every `Student` has its own `gradePointAverage` field, only one copy of each of the methods `getGradePointAverage()` and `setGradePointAverage()` is stored in memory, but any instantiated object of the class can use the single copy.

Because only one copy of each instance method is stored, the computer needs a way to determine whose `gradePointAverage` is being set or retrieved when one of the methods is called. The mechanism that handles this problem is illustrated in Figure 10-13. When a method call such as `oneSophomore.setGradePointAverage(2.6)` is made, the true method call, which is invisible and automatically constructed, includes the memory address of the `oneSophomore` object. (These method calls are represented by the three narrow boxes in the center of Figure 10-13.)

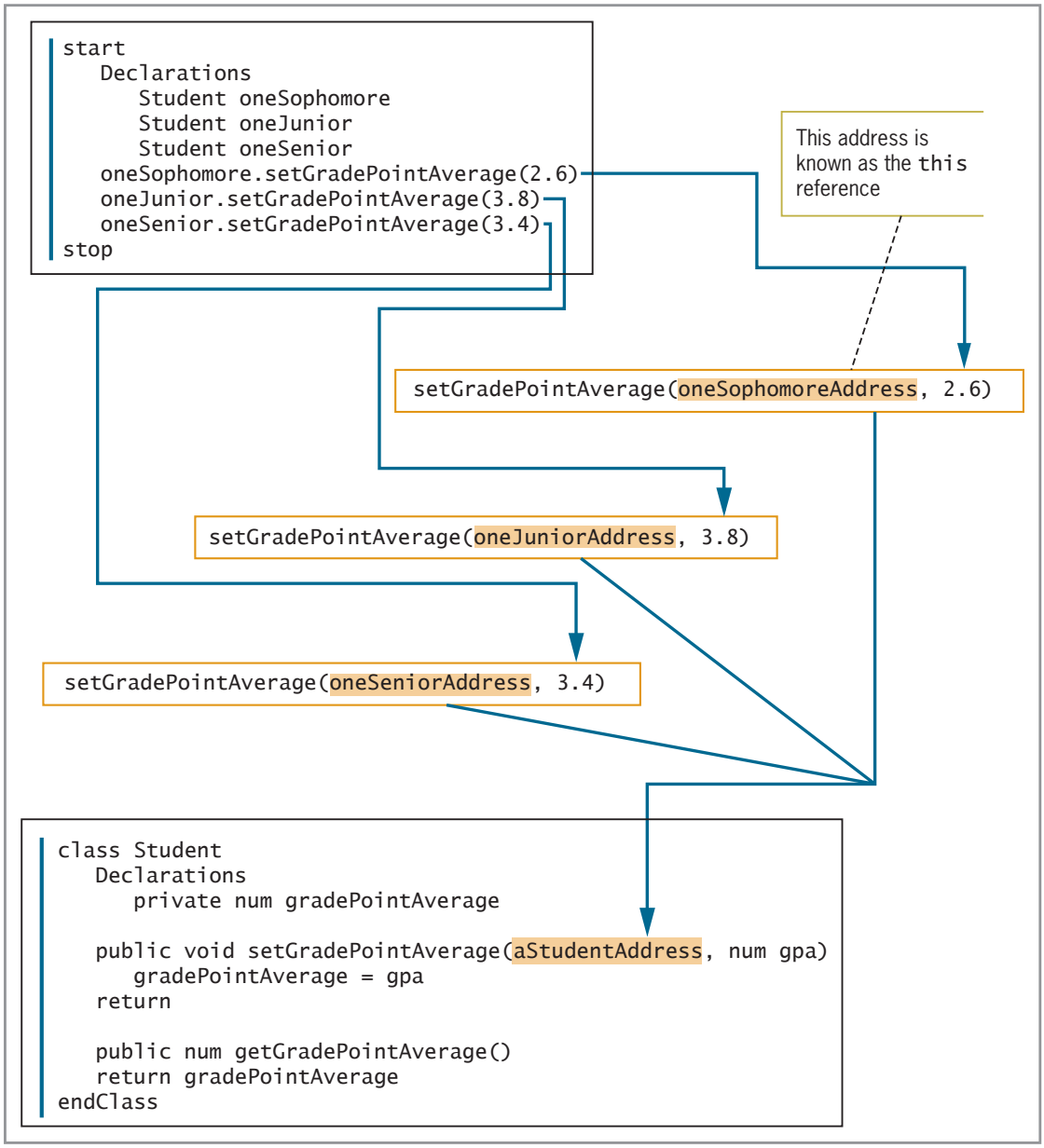

Figure 10-13 How `Student` addresses are passed from an application to an instance method of the `Student` class

Within the `setGradePointAverage()` method in the `Student` class, an invisible and automatically created parameter is added to the list. (For illustration purposes, this parameter is named `aStudentAddress` and is shaded in the `Student` class definition in Figure 10-13. In fact, no parameter is created with that name.) This parameter accepts the address of a `Student` object because the instance

method belongs to the `Student` class; if this method belonged to another class—`Employee`, for example—then the method would accept an address for that type of object. The shaded addresses in Figure 10-13 are not written as code in any program—they are "secretly" sent and received behind the scenes. The address variable in Figure 10-13 is a `this` reference. A **`this` reference** is an automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called. It is called a `this` reference because it refers to "this particular object" that is using the method at the moment. In the application in Figure 10-13, when `oneSophomore` uses the `setGradePointAverage()` method, the address of the `oneSophomore` object is contained in the `this` reference. Later in the program, when the `oneJunior` object uses the `setGradePointAverage()` method, the `this` reference will hold the address of that `Student` object.

Watch the video *The `this` Reference*.

Figure 10-13 shows each place the `this` reference is used in the `Student` class. It is implicitly passed as a parameter to each instance method. You never explicitly refer to the `this` reference when you write the method header for an instance method; Figure 10-13 just shows where it implicitly exists. Within each instance method, the `this` reference is implied any time you refer to one of the class data fields. For example, when you call `setGradePointAverage()` using a `oneSophomore` object, the `gradePointAverage` that is assigned within the method is the "*`this`* `gradePointAverage`", or the one that belongs to the `oneSophomore` object. The phrase "this `gradePointAverage`" usually is written as `this`, followed by a dot, followed by the field name—`this.gradePointAverage`.

The `this` reference exists throughout any instance method. You can explicitly use the `this` reference with data fields, as shown in the methods in the `Student` class in Figure 10-14, but you are not required to do so. Figure 10-14 shows where the `this` reference can be used implicitly, but where you can (but do not have to) use it explicitly. When you write an instance method in a class, the following two identifiers within the method always mean exactly the same thing:

- any field name
- `this`, followed by a dot, followed by the same field name

For example, within the `setGradePointAverage()` method, `gradePointAverage` and `this.gradePointAverage` refer to exactly the same memory location.

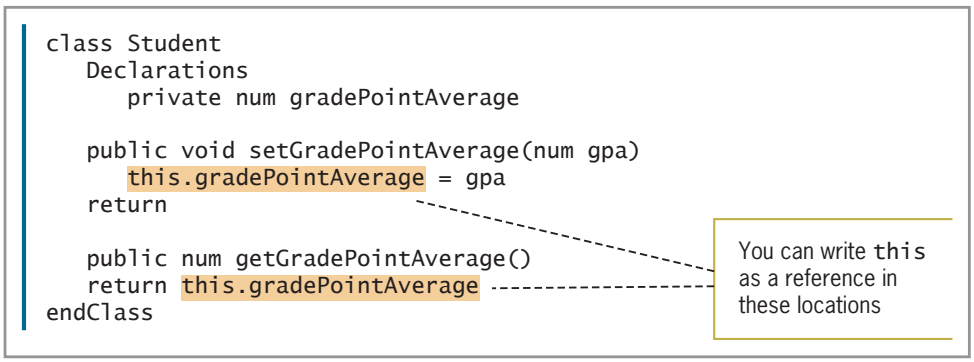

Figure 10-14 Explicitly using `this` in the `Student` class

In a class method, the `this` reference can be used only with identifiers that are field names. For example, in Figure 10-14 you could not refer to `this.gpa` because `gpa` is not a class field—it is only a local variable.

Your organization might prefer that you explicitly use the `this` reference for clarity even though it is not required to create a workable program. It is the programmer's responsibility to follow the conventions established at work or by clients.

The syntax for using `this` differs among programming languages. For example, within a class in C++, you can refer to the `Student` class `gradePointAverage` value as `this->gradePointAverage` or `(*this).gradePointAverage`, but in Java you refer to it as `this.gradePointAverage`. In Visual Basic, the `this` reference is named `Me`, so the variable would be `Me.gradePointAverage`.

Usually you neither want nor need to use the `this` reference explicitly within the methods you write, but the `this` reference is always there, working behind the scenes, so that the data field for the correct object can be accessed.

As an example of an occasion when you might use the `this` reference explicitly, consider the following `setGradePointAverage()` method and compare it to the version in the `Student` class in Figure 10-14.

```
public void setGradePointAverage(num gradePointAverage)
   this.gradePointAverage = gradePointAverage
return
```

In this version of the method, the programmer has chosen to use the variable name `gradePointAverage` as the parameter to the method as well as the instance field within the class. This means that `gradePointAverage` is the name of a local variable within the method whose value is received by passing; it is also the name of a class field. To differentiate the two, you explicitly use the `this` reference with the copy of `gradePointAverage` that is a member of the class. Omitting the `this` reference in this case would result in the local parameter `gradePointAverage` being assigned to itself. The class's instance variable would not be set.

Any time a local variable in a method has the same identifier as a class field, the class field is hidden. This applies whether the local variable is a passed parameter or simply one that is declared within the method. In these cases, you must use a `this` reference to refer to the class field.

TWO TRUTHS & A LIE

Understanding Instance Methods

1. An instance method operates correctly yet differently for each separate instance of a class.
2. A `this` reference is a variable you must explicitly declare with each class you create.
3. When you write an instance method in a class, the following two identifiers within the method always mean exactly the same thing: any field name or `this` followed by a dot, followed by the same field name.

The false statement is #2. A `this` reference is an automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called. You do not declare it explicitly.

Understanding Static Methods

Some methods do not require a `this` reference; that is, the `this` reference makes no sense for them either implicitly or explicitly. For example, the `displayStudentMotto()` method in the `Student` class in Figure 10-15 does not use any data fields from the class, so it does not matter which `Student` object calls it. If you write a program in which you declare 100 `Student` objects, the `displayStudentMotto()` method executes in exactly the same way for each of them; it does not need to know whose motto is displayed and it does not need to access any specific object addresses. As a matter of fact, you might want to display the `Student` motto without instantiating any `Student` objects. Therefore, the `displayStudentMotto()` method can be written as a **class method** instead of an instance method.

```
public static void displayStudentMotto()
   output "Every student is an individual"
   output "in the pursuit of knowledge."
   output "Every student strives to be"
   output "a literate, responsible citizen."
return
```

Figure 10-15 `Student` class `displayStudentMotto()` method

In everyday language, the word *static* means "stationary"; it is the opposite of *dynamic*, which means "changing." In other words, static methods are always the same for the class, whereas nonstatic methods act differently depending on the object used to call them.

All the methods you have worked with in earlier chapters of this book, such as those that performed a calculation or produced some output, were static methods. That is, you did not create objects to call them.

When you write a class, you can indicate two types of methods:

- **Static methods** are those for which no object needs to exist, like the `displayStudentMotto()` method in Figure 10-15. Static methods do not receive a `this` reference as an implicit parameter. Typically, `static` methods include the word static in the method header, as shown shaded in Figure 10-15.
- **Nonstatic methods** are methods that exist to be used with an object created from a class. These instance methods receive a `this` reference to a specific object. In most programming languages, you use the word `static` when you want to declare a static class member and do not use any special word when you want a class member to be nonstatic. In other words, methods in a class are nonstatic instance methods by default.

In most programming languages, you use a static method with the class name, as in the following:

```
Student.displayStudentMotto()
```

In other words, no object is necessary with a static method.

In some languages, notably C++, besides using a static method with the class name, you are also allowed to use a static method with any object of the class, as in `oneSophomore.displayStudentMotto()`.

TWO TRUTHS & A LIE

Understanding Static Methods

1. Class methods do not receive a `this` reference.
2. Static methods do not receive a `this` reference.
3. Nonstatic methods do not receive a `this` reference.

The false statement is #3. Nonstatic methods receive a `this` reference automatically.

Using Objects

After you create a class from which you want to instantiate objects, you can use the objects like you would use any other simpler data type. For example, consider the `InventoryItem` class in Figure 10-16. The class represents items a company manufactures and holds in inventory. Each item has a number, description, and price. The class contains a get and set method for each of the three fields.


```
class InventoryItem
   Declarations
      private string inventoryNumber
      private string description
      private num price

   public void setInventoryNumber(string number)
      inventoryNumber = number
   return

   public void setDescription(string description)
      this.description = description
   return

   public void setPrice(num price)
      if(price < 0)
         this.price = 0
      else
         this.price = price
   return

   public string getInventoryNumber()
   return inventoryNumber

   public string getDescription()
   return description

   public num getPrice()
   return price

endClass
```

Notice the uses of the `this` reference to differentiate between the method parameter and the class field.

Figure 10-16 `InventoryItem` class

Once you declare an `InventoryItem` object, you can use it in many of the ways you would use a simple numeric or string variable. For example, you could pass an `InventoryItem` object to a method or return one from a method. Figure 10-17 shows a program that declares an `InventoryItem` object and passes it to a method for display. The `InventoryItem` is declared in the main program and assigned values. Then the completed item is passed to a method where it is displayed. Figure 10-18 shows the execution of the program.

```
start
   Declarations
      InventoryItem oneItem
   oneItem.setInventoryNumber("1276")
   oneItem.setDescription("Mahogany chest")
   oneItem.setPrice(450.00)
   displayItem(oneItem)
stop

public static void displayItem(InventoryItem item)
   Declarations
      num TAX_RATE = 0.06
      num tax
      num pr
      num total
   output "Item #", item.getInventoryNumber()
   output item.getDescription()
   pr = item.getPrice()
   tax = pr * TAX_RATE
   total = pr + tax
   output "Price is $", pr, " plus $", tax, " tax"
   output "Total is $", total
return
```

Figure 10-17 Application that declares and uses an `InventoryItem` object

Figure 10-18 Execution of application in Figure 10-17

The `InventoryItem` declared in the main program in Figure 10-17 is passed to the `displayItem()` method in much the same way a numeric or string variable would be. The method receives a copy of the `InventoryItem` that is known locally by the identifier item. Within the method, the field values of the local item can be retrieved, displayed, and used in arithmetic statements in the same way they could have been in the main program where the `InventoryItem` was originally declared.

Figure 10-19 shows a more realistic application that uses `InventoryItem` objects. In the main program, an `InventoryItem` is declared and the user is prompted for a number. As long as the user does not enter the `QUIT` value, a loop is executed in which the entered inventory item number is passed to the `getItemValues()` method. Within that method, a local `InventoryItem` object is declared. This local object is used to gather and hold the user's input values. The user

is prompted for a description and price; then the passed item number, as well as the newly obtained description and price, are assigned to the local `InventoryItem` object via its set methods. The completed object is returned to the program where it is assigned to the `InventoryItem` object. That item is then passed to the `displayItem()` method. As in the previous example, the method calculates tax and displays results. Figure 10-20 shows a typical execution.

```
start
   Declarations
      InventoryItem oneItem
      string itemNum
      string QUIT = "0"
   output "Enter item number or ", QUIT, " to quit… "
   input itemNum
   while itemNum <> "0"
      oneItem = getItemValues(itemNum)
      displayItem(oneItem)
      output "Enter next item number or ", QUIT, " to quit… "
      input itemNum
   endwhile
stop

public static InventoryItem getItemValues(string number)
   Declarations
      InventoryItem inItem
      string desc
      num price
   output "Enter description… "
   input desc
   output "Enter price… "
   input price
   inItem.setInventoryNumber(number)
   inItem.setDescription(desc)
   inItem.setPrice(price)
return inItem

public static void displayItem(InventoryItem item)
   Declarations
      num TAX_RATE = 0.06
      num tax
      num pr
      num total
   output "Item #", item.getInventoryNumber()
   output item.getDescription()
   pr = item.getPrice()
   tax = pr * TAX_RATE
   total = pr + tax
   output "Price is $", pr, " plus $", tax, " tax"
   output "Total is $", total
return
```

Figure 10-19 Application that uses `InventoryItem` objects

In Figure 10-19, notice that the return type for the `getItemValues()` method is `InventoryItem`. A method can return only a single value. Therefore, it is convenient that the `getItemValues()` method can encapsulate two strings and a number in a single `InventoryItem` object that it returns to the main program.

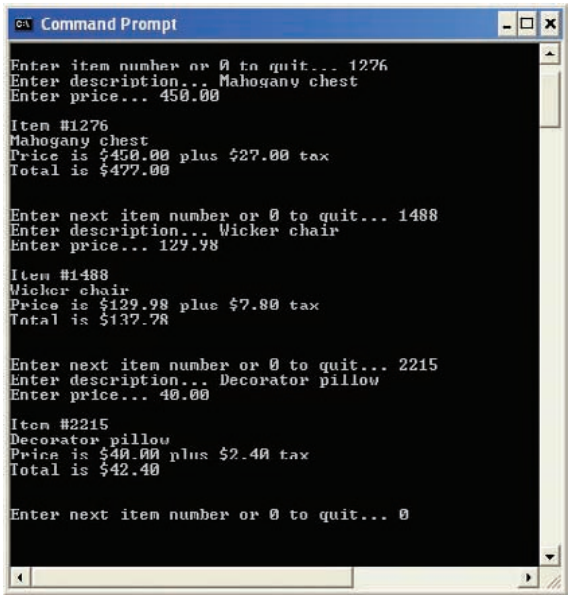

Figure 10-20 Typical execution of program in Figure 10-19

TWO TRUTHS & A LIE

Using Objects

1. You can pass an object to a method.
2. Because only one value can be returned from a method, you cannot return an object that holds more than one field.
3. You can declare an object locally within a method.

The false statement is #2. An object can be returned from a method.

Chapter Summary

- Classes are the basic building blocks of object-oriented programming. When you think in an object-oriented manner, everything is an object, and every object is an instance of a class. A class's fields, or instance variables, hold its data, and every object that is an instance of a class possesses the same methods. A program or class that instantiates objects of another prewritten class is a class client or class user. In addition to classes and objects, three important features of object-oriented languages are polymorphism, inheritance, and encapsulation.
- A class definition is a set of program statements that tell you the characteristics of the class's objects and the methods that can be applied to its objects. A class definition can contain a name, data, and methods. Programmers often use a class diagram to illustrate class features. The purposes of many methods contained in a class can be divided into three categories: set methods, get methods, and work methods.
- Object-oriented programmers usually specify that their data fields will have private access—that is, the data cannot be accessed by any method that is not part of the class. The methods frequently support public access, which means that other programs and methods may use the methods that control access to the private data. In a class diagram, a minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.
- As classes grow in complexity, deciding how to organize them becomes increasingly important. Depending on the class, you might choose to store the data fields by listing a key field first, listing fields alphabetically, or by data type or accessibility. Methods might be stored in alphabetical order or in pairs of get and set methods.
- An instance method operates correctly yet differently for every object instantiated from a class. When an instance method is called, a `this` reference that holds the object's memory address is automatically passed to the method.
- Some methods do not require a `this` reference. When you write a class, you can indicate two types of methods: static methods, which are also known as class methods and do not receive a `this` reference as an implicit parameter; and nonstatic methods, which are instance methods and do receive a `this` reference.
- After you create a class from which you want to instantiate objects, you can use the objects as you would use any other simpler data type.

Key Terms

Object-oriented programming (**OOP**) is a style of programming that focuses on an application's data and the methods you need to manipulate that data.

A **class** describes a group or collection of objects with common attributes.

An **object** is one tangible example of a class; it is an instance of a class.

An **instance** is one tangible example of a class; it is an object.

Attributes are the characteristics that define an object as part of a class.

An **is-a relationship** exists between an object and its class.

An **instantiation** of a class is an instance.

A class's **instance variables** are the data components that belong to every instantiated object.

Fields are object attributes or data.

The **state** of an object is the set of all the values or contents of its instance variables.

A **class client** or **class user** is a program or class that instantiates objects of another prewritten class.

Pure polymorphism describes situations in which one method body is used with a variety of arguments.

Inheritance is the process of acquiring the traits of one's predecessors.

Encapsulation is the process of combining all of an object's attributes and methods into a single package.

Information hiding (or **data hiding**) is the concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege.

A **class definition** is a set of program statements that tell you the characteristics of the class's objects and the methods that can be applied to its objects.

A **user-defined type**, or **programmer-defined type**, is a type that is not built into a language, but is created by the programmer.

An **abstract data type** (**ADT**) is a programmer-defined type such as a class.

Primitive data types are simple numbers and characters that are not class types.

A **class diagram** consists of a rectangle divided into three sections that show the name, data, and methods of a class.

A **set method** sets the values of a data field within a class.

Mutator methods are ones that set values in a class.

A **property** provides methods that allow you to get and set a class field value using a simple syntax.

A **get method** returns a value from a class.

Accessor methods get values from class fields.

Work methods perform tasks within a class.

Help methods and **facilitators** are other names for work methods.

Private access, as applied to a class's data or methods, specifies that the data or method cannot be used by any method that is not part of the same class.

Public access, as applied to a class's data or methods, specifies that other programs and methods may use the specified data or methods.

An **access specifier** (or **access modifier**) is the adjective that defines the type of access outside classes will have to the attribute or method.

A **primary key** is a unique identifier for each object in a database.

An **instance method** operates correctly yet differently for each class object. An instance method is nonstatic and receives a `this` reference.

A **`this` reference** is an automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called.

A **class method** is a static method. Class methods are not instance methods and do not receive a `this` reference.

Static methods are those for which no object needs to exist. Static methods are not instance methods and do not receive a `this` reference.

Nonstatic methods are methods that exist to be used with an object created from a class; they are instance methods and receive a `this` reference.

Review Questions

1. Which of the following means the same as *object*?
   a. class
   b. field
   c. instance
   d. category

2. Which of the following means the same as *instance variable*?
   a. field
   b. instance
   c. category
   d. class

3. A program that instantiates objects of another prewritten class is a(n) _____.
   a. object
   b. client
   c. instance
   d. GUI

4. The relationship between an instance and a class is a(n) _____ relationship.
   a. has-a
   b. is-a
   c. polymorphic
   d. hostile

5. Which of these does not belong with the others?
   a. instance variable
   b. attribute
   c. object
   d. field

6. The process of acquiring the traits of one's predecessors is _____.

   a. inheritance

   b. encapsulation

   c. polymorphism

   d. orientation

7. When discussing classes and objects, *encapsulation* means that _____.

   a. all the fields belong to the same object

   b. all the fields are private

   c. all the fields and methods are grouped together

   d. all the methods are public

8. Every class definition must contain _____.

   a. a name

   b. data

   c. methods

   d. all of the above

9. Assume a working program contains the following statement:

   ```
   myDog.setName("Bowser")
   ```

   Which of the following do you know?

   a. `setName()` is a public method

   b. `setName()` accepts a string parameter

   c. both of these

   d. none of these

10. Assume a working program contains the following statement:

    ```
    name = myDog.getName()
    ```

    Which of the following do you know?

    a. `getName()` returns a string

    b. `getName()` returns a value that is the same data type as `name`

    c. both of these

    d. none of these

11. A class diagram _____.
    a. provides an overview of a class's data and methods
    b. provides method implementation details
    c. is never used by nonprogrammers because it is too technical
    d. all of the above

12. Which of the following is the most likely scenario for a specific class?
    a. Its data is private and its methods are public.
    b. Its data is public and its methods are private.
    c. Its data and methods are both public.
    d. Its data and methods are both private.

13. An instance method _____.
    a. is static
    b. receives a `this` reference
    c. both of these
    d. none of these

14. Assume you have created a class named `Dog` that contains a data field named `weight` and an instance method named `setWeight()`. Further assume the `setWeight()` method accepts a numeric parameter named `weight`. Which of the following statements correctly sets a `Dog`'s weight within the `setWeight()` method?
    a. `weight = weight`
    b. `this.weight = this.weight`
    c. `weight = this.weight`
    d. `this.weight = weight`

15. A static method is also known as a(n) _____ method.
    a. instance
    b. public
    c. private
    d. class

16. By default, methods contained in a class are _____ methods.

    a. static

    b. nonstatic

    c. class

    d. public

17. Assume you have created a class named `MyClass`, and that a working program contains the following statement:

    ```
    output MyClass.number
    ```

    Which of the following do you know?

    a. `number` is a numeric field

    b. `number` is a static field

    c. `number` is an instance variable

    d. all of the above

18. Assume you have created an object named `myObject` and that a working program contains the following statement:

    ```
    output myObject.getSize()
    ```

    Which of the following do you know?

    a. `size` is a private numeric field

    b. `size` is a static field

    c. `size` is a public instance variable

    d. all of the above

19. Assume you have created a class named `MyClass` and that it contains a private field named `myField` and a nonstatic public method named `myMethod()`. Which of the following is true?

    a. `myMethod()` has access to and can use `myField`

    b. `myMethod()` does not have access to and cannot use `myField`

    c. `myMethod()` can use `myField` but cannot pass it to other methods

    d. `myMethod()` can use `myField` only if `myField` is passed to `myMethod()` as a parameter

20. An object can be _____.
    a. stored in an array
    b. passed to a method
    c. returned from a method
    d. all of the above

Exercises

1. Identify three objects that might belong to each of the following classes:
    a. `Automobile`
    b. `NovelAuthor`
    c. `CollegeCourse`

2. Identify three different classes that might contain each of these objects:
    a. Wolfgang Amadeus Mozart
    b. My pet cat named Socks
    c. Apartment 14 at 101 Main Street

3. Design a class named `CustomerRecord` that holds a customer number, name, and address. Include methods to set the values for each data field and display the values for each data field. Create the class diagram and write the pseudocode that defines the class.

4. Design a class named `House` that holds the street address, price, number of bedrooms, and number of baths in a house. Include methods to set the values for each data field, and include a method that displays all the values for a `House`. Create the class diagram and write the pseudocode that defines the class.

5. Design a class named `Loan` that holds an account number, name of account holder, amount borrowed, term, and interest rate. Include methods to set values for each data field and a method that displays all the loan information. Create the class diagram and write the pseudocode that defines the class.

6. Complete the following tasks:

   a. Design a class named `Book` that holds a stock number, author, title, price, and number of pages for a book. Include methods to set and get the values for each data field. Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares two `Book` objects and sets and displays their values.

   c. Design an application that declares an array of 10 `Books`. Prompt the user for data for each of the `Books`, then display all the values.

7. Complete the following tasks:

   a. Design a class named `Pizza`. Data fields include a string field for toppings (such as pepperoni), numeric fields for diameter in inches (such as 12), and price (such as 13.99). Include methods to get and set values for each of these fields. Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares two `Pizza` objects and sets and displays their values.

   c. Design an application that declares an array of 10 `Pizzas`. Prompt the user for data for each of the `Pizzas`, then display all the values.

8. Complete the following tasks:

   a. Design a class named `HousePlant`. A `HousePlant` has fields for a name (for example, "Philodendron"), a price (for example, 29.99), and a field that indicates whether the plant has been fed in the last month (for example, "Yes"). Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares two `HousePlant` objects and sets and displays their values.

   c. Design an application that declares an array of 10 `HousePlants`. Prompt the user for data for each of the `HousePlants`, then display all the values.

Find the Bugs

9. Your student disk contains files named DEBUG10-01.txt, DEBUG10-02.txt, and DEBUG10-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

10. a. Playing cards are used in many computer games, including versions of such classics as Solitaire, Hearts, and Poker. Design a `Card` class that contains a string data field to hold a suit (spades, hearts, diamonds, or clubs) and a numeric data field for a value from 1 to 13. Include get and set methods for each field. Write an application that randomly selects two playing cards and displays their values.

    b. Using two `Card` objects, design an application that plays a simple version of the card game War. Deal two `Cards`—one for the computer and one for the player. Determine the higher card, then display a message indicating whether the cards are equal, the computer won, or the player won. (Playing cards are considered equal when they have the same value, no matter what their suit is.) For this game, assume the Ace (value 1) is low. Make sure that the two `Cards` dealt are not the same `Card`. For example, a deck cannot contain more than one Queen of Spades.

Up for Discussion

11. In this chapter, you learned that instance data and methods belong to objects (which are class members), but that static data and methods belong to a class as a whole. Consider the real-life class named `StateInTheUnitedStates`. Name some real-life attributes of this class that are static attributes and instance attributes. Create another example of a real-life class and discuss what its static and instance members might be.

12. Some programmers use a system called Hungarian notation when naming their variables and class fields. What is Hungarian notation and why do many object-oriented programmers feel it is not a valuable style to use?