13 OBJECT-ORIENTED PROGRAMMING

After studying Chapter 13, you should be able to:

- Understand the basic principles of object-oriented programming
- Define classes and create class diagrams
- Understand public and private access
- Instantiate and use objects
- Understand inheritance
- Understand polymorphism
- Understand protected access
- Understand the role of the `this` reference
- Use constructors and destructors
- Describe GUI classes as an example of built-in classes
- Understand the advantages of object-oriented programming

AN OVERVIEW 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. Object-oriented programming uses all of the concepts you are familiar with from modular procedural programming, such as variables, modules, and passing values to modules. Modules 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; then you will learn the specific terminology.

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 his or her shots are current.

TIP

In grammar, a noun is similar to an object in object-oriented programs, and the values of an object's attributes are like adjectives—they describe the characteristics of the objects. Programmers also call the values of an object's attributes the **properties** of the object. The **state of an object** is the collective value of all its attributes at any point in time. Later in this chapter, you will learn about the methods in a class, which are equivalent to verbs.

In object-oriented terminology, a **class** is a term that describes a group or collection of objects with common properties. An **instance** of a class is an existing object of a class. Therefore, your `red Chevrolet Automobile` with the dent can be considered an instance of the class that is made up of all automobiles, and your `Golden Retriever Dog` named Ginger is an instance of the class that is made up of all dogs. Thinking of items as instances of a class allows you to apply your general knowledge of the class to individual members of the class. 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, and this allows you to imagine these objects reasonably well before you see them. When you visit your friend and see the `Automobile` or `Dog` for the first time, you probably will recognize it as the new acquisition. 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.

When you approach a new programming assignment using object-oriented programming techniques:

- You analyze the objects you are working with and the tasks that need to be performed with, and on, those objects. Then you design classes that encapsulate the attributes and functionality of those objects.

- You pass messages to objects, requesting the objects to take action. The same message works differently (and appropriately) when applied to different objects. This means that, if well-designed, you can use a single module or procedure name to work appropriately with different types of data it receives.
- Objects can share or inherit traits of objects that have already been created, reducing the time it takes to create new objects.
- Encapsulation and information hiding are emphasized.

OBJECTS AND CLASSES

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, but, even though these procedures operate differently using the different objects, you can call all 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.

METHODS

You can create multiple methods with the same name, which will act differently and appropriately when used with different types of objects. This concept is **polymorphism**, which literally means "many forms"—a method can have many configurations that each work appropriately based on the context in which they are used. In most object-oriented programming languages, method names are followed by a set of parentheses; this helps you distinguish method names from variable names. You have been using this style throughout this book. For example, a method named `display()` might be usable to display the characteristics of an `Automobile`, `Dog`, or `CustomerInvoice`. Because you can use the same method name, `display()`, to describe the different actions needed to display 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 `display()` is easier than remembering `displayAutomobile()`, `displayDog()`, 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. In object-oriented programs, when you create multiple methods with the same name but different argument lists, you **overload the method**.

TIP

Purists find a subtle difference between overloading and polymorphism. Some reserve the term "polymorphism" (or **pure polymorphism**) for situations in which one function body is used with a variety of arguments. For example, a single function 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.

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 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 and in the same way with each type of object.

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 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. Similarly, you understand the traits of a `Convertible` because it inherits almost all of its features from an `Automobile` and you understand most of the characteristics of a `Poodle` if you know it is a `Dog`. 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` object, there is no need to develop an `overdueCustomerBill` object from scratch. You can create the new type of object to contain all the characteristics of the already developed object, 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 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 the 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.

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

In summary, to understand object-oriented programming, you must consider five concepts that are integral components of all object-oriented programming languages:

- Classes
- Objects
- Inheritance
- Polymorphism
- Encapsulation

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.

For example, `Dish` is a class. When you know an object is a `Dish`, you know it can be held in your hand and you can eat from it. The specific object `myBlueDinnerPlateWithTheChipOnTheEdge` is an instance of the `Dish` class; so is `auntJanesAntiquePunchBowl` and `myCatsFoodBowl`. You can use the phrase **is-a** to test whether an object is an instance of a class. Because you can say, "My plate *is a* `Dish`," you can discern the object-class relationship. On the other hand, you cannot say, "A Dish is my plate," because many dishes are *not* my plate. Each button on the toolbar of a word-processing program is an instance of a `Button` class. In a program used to manage a hotel, `thePentHouse`, `theBridalSuite`, `room201`, and `room202` all are instances of `HotelRoom`. Although each room is a different object, as members of the same class they share characteristics—each has a maximum number of occupants, a square footage, and a price.

TIP In object-oriented languages such as C++ and Java, by convention, most class names are written with the initial letter of each new word in uppercase, as in *Dish* or *HotelRoom*. Specific objects' names usually are written in lowercase or using camel casing.

TIP Object-oriented programmers also use the term "is-a" to describe class-to-class inheritance relationships.

A class 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 `idNum`, `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 `setFieldValues()`, `calculateWeeklyPay()`, and `printFieldValues()`. The job of `setFieldValues()` is to provide values for an `Employee`'s data fields, the purpose of `calculateWeeklyPay()` is to multiply the `Employee`'s `hourlyWage` by 40 to calculate a weekly salary, and the purpose of `printFieldValues()` is to print the values in the `Employee`'s data fields. With object-oriented languages, you think of the class name, data, and methods as a single encapsulated unit.

Programmers often use a class diagram to illustrate class features. A **class diagram** consists of a rectangle divided into three sections, as shown in Figure 13-1. 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 for a given class there might be any number of either, including none. Figure 13-2 shows the class diagram for the `Employee` class.

FIGURE 13-1: GENERIC CLASS DIAGRAM

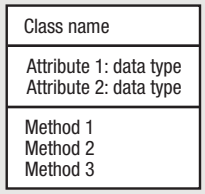
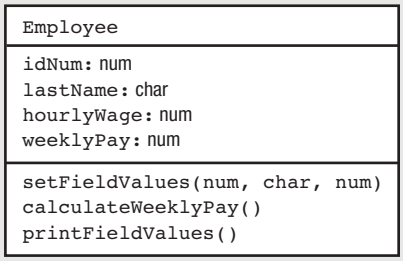


FIGURE 13-2: `Employee` CLASS DIAGRAM



TIP Later in this chapter, you will learn to add access specifiers to your class diagrams.

TIP Some class designers prefer to define any field that never will be used in a computation as a non-numeric data type. For example, in the `Employee` class diagram in Figure 13-2, you might prefer to define `Employee idNum` as a field that can contain characters.

Figures 13-1 and 13-2 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. Later, when you plan the code that actually creates the class, you include method implementation details; at that point, you might draw a flowchart or write pseudocode for each method, as you have been doing throughout this book.

In Figure 13-2 in the `setFieldValues()` method, the class diagram indicates that three items will be sent into the method—a numeric data item, a character data item, and another numeric data item. When you view the class diagram, you don't know how these will be used, but when you write the class definition, their use is defined. For example, Figure 13-3 shows some pseudocode you can use to show the details for the methods contained within the `Employee` class.

FIGURE 13-3: `Employee` CLASS PSEUDOCODE WITHOUT ACCESS SPECIFIERS

```
class Employee
      num idNum
      char lastName
      num hourlyWage
      num weeklyPay

      setFieldValues(num id, char last, num rate)
         const num MAX_RATE = 25.00
         idNum = id
         lastName = last
         if rate <= MAX_RATE then
            hourlyWage = rate
         else
            hourlyWage = MAX_RATE
         endif
      return

      calculateWeeklyPay()
         const num WORK_WEEK = 40
         weeklyPay = hourlyWage * WORK_WEEK
      return

      printFieldValues()
         print idNum, lastName, hourlyWage, weeklyPay
      return
endClass
```

In Figure 13-3, 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 13-3 shows the complete methods for the `Employee` class. The purpose of two of the methods is to communicate with the outside world—the `setFieldValues()` method takes values that come in from the outside and assigns them to the `Employee`'s attributes, and the `printFieldValues()` method displays the `Employee`'s attributes on an output device. The purpose of the `calculateWeeklyPay()` module is to multiply `hourlyWage` by 40. Each method can contain elements with which you are familiar from non-object-oriented programs. For example, the `setFieldValues()` method declares a constant and makes a decision on how to set the `Employee`'s pay rate based on the value of the constant.

UNDERSTANDING PUBLIC AND PRIVATE ACCESS

When you buy a product with a warranty, one of the conditions of the warranty is usually 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 CPU 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 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 product. 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. In the `Employee` class in Figure 13-3, you do not want `hourlyWage` to be initialized to more than $25.00.

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 `setFieldValues()`, to alter data. (You have already learned that the principle of keeping data private and inaccessible to outside classes is called information 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 `setFieldValues()`, allow **public access**, which means that other programs and methods may use the methods. An **access specifier** (or **access modifier**) is an adjective that defines the type of access outside classes will have to the attribute or method (`public` or `private`). Figure 13-4 shows a complete `Employee` class to which shaded access specifiers have been added to describe each attribute and method.

TIP Classes can contain public data and private methods, but it is common for most data to be private and most methods to be public.

TIP 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. 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.

TIP Many object-oriented languages provide more specific access specifiers than just `public` and `private`. Later in this chapter, you learn about the `protected` access specifier.

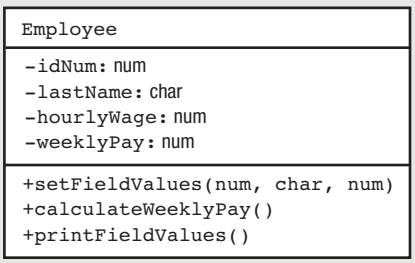
TIP Notice that the last line in the `Employee` class in both Figures 13-3 and 13-4 is an `endClass` statement. Similar to the way this book has used `endif` and `endwhile` to mark the end of `if` and `while` blocks of code, this book will use `endClass` to indicate the end of a class definition.

FIGURE 13-4: `Employee` CLASS USING `private` AND `public` ACCESS SPECIFIERS

```
class Employee
      private num idNum
      private char lastName
      private num hourlyWage
      private num weeklyPay

      public setFieldValues(num id, char last, num rate)
         const num MAX_RATE = 25.00
         idNum = id
         lastName = last
         if rate <= MAX_RATE then
            hourlyWage = rate
         else
            hourlyWage = MAX_RATE
         endif
      return

      public calculateWeeklyPay()
         const num WORK_WEEK = 40
         weeklyPay = hourlyWage * WORK_WEEK
      return

      public printFieldValues()
         print idNum, lastName, hourlyWage, weeklyPay
      return
endClass
```

When creating a class diagram, many programmers like to specify whether each data item and method in a class is public or private. Figure 13-5 shows the conventions that are typically used. A minus sign (–) precedes items that are private; a plus sign (+) precedes those that are public.

FIGURE 13-5: `Employee` CLASS DIAGRAM WITH `public` AND `private` ACCESS SPECIFIERS



TIP When you learn more about inheritance later in this chapter, you will learn about the `protected` access specifier. You use an octothorpe, also called a pound sign or number sign (#), to indicate protected access.

INSTANTIATING AND USING OBJECTS

When you write an object-oriented program, you create objects that are members of a class. You **instantiate** (or create) a class object (or instance) with a statement that includes the type of object and an identifying name. For example, the following statement creates an `Employee` object named `myAssistant`:

```
Employee myAssistant
```

TIP

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();
```

This syntax, using the class name followed by parentheses, will be explained later in this chapter when you learn about constructor methods.

When you declare `myAssistant` as an `Employee` object, the `myAssistant` object contains all of the data fields or attributes defined in the class, and has access to all the class's methods. You can use any of an `Employee`'s methods—`setFieldValues()`, `calculateWeeklyPay()`, and `printFieldValues()`—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 13-6.

FIGURE 13-6: PROGRAM THAT USES AN `Employee` OBJECT

```
start
   declare variables
      Employee myAssistant
   myAssistant.setFieldValues(123, "Tyler", 12.50)
   myAssistant.calculateWeeklyPay()
   myAssistant.printFieldValues()
stop
```

TIP

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**. Programming languages in which you can create your own data types are **extensible**, meaning extendable. A class is also an **abstract data type** (ADT)—a type whose internal form is hidden behind a set of methods you use to access the data.

The following statements contain method calls:

```
myAssistant.setFieldValues(123, "Tyler", 12.50)
myAssistant.calculateWeeklyPay()
myAssistant.printFieldValues()
```

These calls are similar to module or method calls you have seen throughout this book, but in this case the methods themselves are part of the `Employee` class, which is why an `Employee` object can use them. You can think of the

Employee object myAssistant as "owning" or "driving" those methods; when those methods refer to data fields, they refer to the myAssistant object's data fields and not the data fields of any other Employee.

When you write the program in Figure 13-6, you do not need to know what statements are written within the methods of the Employee class, although you could make an educated guess based on the methods' names. Before you could execute the application in Figure 13-6, you would have to write appropriate statements within the Employee class's methods, but if another programmer has already written the methods, then you can use the application in Figure 13-6 without knowing the details contained in the methods. The ability to use methods without knowing the details of their contents is a feature of encapsulation.

TIP Programmers like to say the method details are contained in a black box—a device you can use without knowing how its contents operate. You first learned the term "black box" in Chapter 10.

A program or method that uses a class object is a **client of the class**. Many programmers write only client programs, never creating classes themselves, but using only classes that others have created. In the client program in Figure 13-6, 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.

TIP Of course, the program in Figure 13-6 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, and you might create hundreds of objects in turn.

TIP 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, such as C#, every item you name, even one that is a num or char type, really is an object that is an instance of a class that contains both data and methods.

TIP 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 methods.

UNDERSTANDING INHERITANCE

The concept of class is useful because of its reusability; you can create new classes that are descendents of existing classes. The **descendent classes** (or **child classes**) can inherit all of the attributes of the **original class** (or **parent class**), or the descendent class can override those attributes that are inappropriate. For example, if you have created a class named BankLoan, it probably contains fields such as the account number, the name, address, and phone number of the loan recipient, the amount of the loan, and the interest rate. The class probably also contains methods that set, display, and manipulate these values. When you need a more specific class for a CarLoan that contains data about the car, or HomeImprovementLoan that contains data about the home improvement, you do not want to have to start from scratch. It makes sense to inherit existing features from the BankLoan class, adding only the new features that the more specific loans require.

TIP You can call a parent class a **base class** or **superclass**. You can refer to a child class as a **derived class** or **subclass**.

As another example, to accommodate part-time workers in your personnel programs, you might want to create a child class from the `Employee` class. Part-time workers need an ID, name, and hourly wage, just as regular employees do, but the regular `Employee` pay calculation assumes a 40-hour workweek. You might want to create a `PartTimeEmployee` class that inherits all the data fields contained in `Employee`, but adds a new one—`hoursWorked`. In addition, you want to create a modified `setFieldValues()` method that includes assigning a value to `hoursWorked`, and a new `calculateWeeklyPay()` method that operates correctly for `PartTimeEmployee` objects. This new method multiplies `hourlyWage` by `hoursWorked` instead of by 40. The `printFieldValues()` module that already exists within the `Employee` class works appropriately for both the `Employee` and the `PartTimeEmployee` classes, so there is no need to include a new version of this module within the `PartTimeEmployee` class; `PartTimeEmployee` objects can simply use their parent's existing method.

TIP You can think of a child class as being more specific than a parent class. For example, `PartTimeEmployee` is a specific type of `Employee`.

TIP A child class contains all the data fields and methods of its parent, plus any new ones you define. A parent class does not gain any child class members.

When you create a child class, you can show its relationship to the parent with a class diagram like the one for `PartTimeEmployee` in Figure 13-7. The complete `PartTimeEmployee` class appears in Figure 13-8.

FIGURE 13-7: `PartTimeEmployee` CLASS DIAGRAM

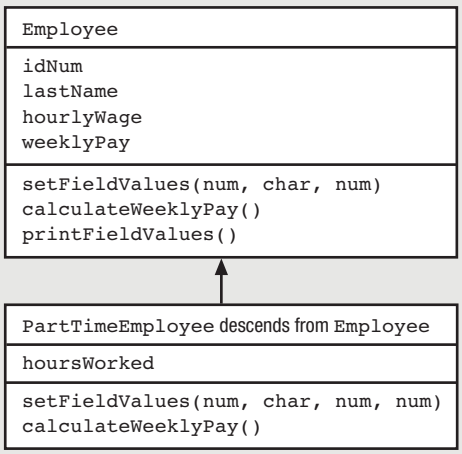


FIGURE 13-8: THE `PartTimeEmployee` CLASS

```
class PartTimeEmployee descends from Employee
   private num hoursWorked
   public void setFieldValues(num id, char last, num rate, num hours)
      Employee class version: setFieldValues(id, last, rate)
      hoursWorked = hours
   return
   public void calculateWeeklyPay()
      weeklyPay = hourlyWage * hoursWorked
   return
endClass
```

TIP The class in Figure 13-8 uses the phrase "descends from" to indicate inheritance. Each programming language uses its own syntax. For example, using Java, you would write "extends", in Visual Basic .NET you would write "inherits", and in C++ and C# you would use a colon between the class name and its parent.

The `PartTimeEmployee` class shown in Figure 13-8 contains five data fields—all the fields that `Employee` contains plus one new one, `hoursWorked`. The `PartTimeEmployee` class also contains three methods. Two of the methods, `setFieldValues()` and `calculateWeeklyPay()`, have been rewritten for the `PartTimeEmployee` child class, because they will operate differently when used with `PartTimeEmployee` than when used with `Employee`. The other method, `printFieldValues()`, is not rewritten because the parent class version is a usable version for the child class.

In Figure 13-8, the `PartTimeEmployee` class `setFieldValues()` method takes four arguments. Three are passed to the parent class `setFieldValues()` method, where they can be assigned to the class fields. Because the parent class method already provides statements that set the values of three of the class fields, the `PartTimeEmployee` class can take advantage of the fact that part of the work has been done. Being able to reuse code is an advantage of inheritance. In Figure 13-8, calling the parent class method is indicated by the phrase "Employee class version:". The actual syntax you use when writing code varies among programming languages.

The fourth argument to the `PartTimeEmployee` class `setFieldValues()` method, `hours`, is assigned to `hoursWorked` in the child class method because the parent class does not contain that field.

The `calculateWeeklyPay()` method in the `PartTimeEmployee` class uses the variable `hoursWorked` instead of the constant 40 to calculate weekly pay. The methods in the child class that have the same name and argument list as those in the parent class are said to **override**, or take precedence over, the parent class methods.

TIP A child class method overrides a parent's method when it has the same name and argument list. It overloads a parent's method just as any method is overloaded—when it has the same name as another, but a different argument list.

TIP Before the `PartTimeEmployee` child class can use the `hourlyWage` and `weeklyPay` fields, object-oriented programming languages require one additional modification to the `Employee` parent class. You will learn about this modification, making the parent class fields protected, later in this chapter.

The `PartTimeEmployee` class also contains the `printFieldValues()` method, which it inherits unchanged from its parent. You do not see a copy of the `printFieldValues()` method in the `PartTimeEmployee` class in Figure 13-8, because the phrase `descends from Employee` in the first line of the class means that all `Employee` class members automatically are included in the child class unless they have been overridden. When you write an application such as the one shown in Figure 13-9, declaring `Employee` as well as `PartTimeEmployee` objects, different `setFieldValues()` and `calculateWeeklyPay()` methods containing different statements are called for each object, but the same `printFieldValues()` method is called in each case.

FIGURE 13-9: APPLICATION THAT USES `Employee` AND `PartTimeEmployee` OBJECTS

```
start
   declare variables
      Employee myAssistant
      PartTimeEmployee myDriver
   myAssistant.setFieldValues(123, "Tyler", 12.50)
   myDriver.setFieldValues(234, "Mitchell", 15.00, 20)
   myAssistant.calculateWeeklyPay()
   myDriver.calculateWeeklyPay()
   myAssistant.printFieldValues()
   myDriver.printFieldValues()
stop
```

In the program in Figure 13-9, two objects are declared. The `myAssistant` object is a "plain" `Employee`; the `myDriver` object is a more specific `PartTimeEmployee`. The statement `myDriver.setFieldValues()` calls a different method than `myAssistant.setFieldValues()`; the two methods have the same name, but belong to different classes. The compiler knows which method to call based on the type of object, but the programmer can use one easy-to-remember method name in both cases. The method name `setFieldValues()` can be used with either type of object, and it works appropriately with either type.

In Figure 13-9, the two calls to `calculateWeeklyPay()` cause two different method executions; the compiler knows which version to use because the objects associated with the calls belong to different classes.

The final two statements before the `stop` statement in Figure 13-9 call the `printFieldValues()` method with each of the two objects. In these statements, the same method is called each time. Naturally, the `myAssistant` object uses the `printFieldValues()` method contained in the `Employee` class. The `myDriver` object also uses the `printFieldValues()` method from the `Employee` class because of the following reasoning:

- `myDriver` is a `PartTimeEmployee`.
- The `PartTimeEmployee` class does not contain its own version of the `printFieldValues()` method.

- The `PartTimeEmployee` class is a child class of `Employee`.
- The `Employee` class contains a `printFieldValues()` method that the `myDriver` object can use.

A child class will use its parent class methods unless the child class has its own version that either overrides or overloads the parent's version.

TIP A good way to determine whether a class is a parent or a child is to use the "is-a" test. A child "is an" example of its parent. For example, it is always true that a `PartTimeEmployee` "is an" `Employee`. However, it is not necessarily true that an `Employee` "is a" `PartTimeEmployee`.

TIP When you create a class that is meant only to be a parent class and not to have objects of its own, you create an **abstract class**. For example, suppose you create an `Employee` class and two child classes, `PartTimeEmployee` and `FullTimeEmployee`. If your intention is that every object belongs to one of the two child classes and that there are no "plain" `Employee` objects, then `Employee` is an abstract class.

TIP In some programming languages, such as C# and Java, every class you create is a child of one ultimate base class, often called the `Object` class. The `Object` class usually provides you with some basic functionality that all the classes you create inherit—for example, the ability to show its memory location and name.

TIP Some, but not all, programming languages allow **multiple inheritance**, in which classes you create can have many parents, inheriting all the attributes and methods of each.

UNDERSTANDING POLYMORPHISM

Object-oriented programs use a feature called polymorphism to allow the same request—that is, the same method call—to be carried out differently, depending on the context; this is seldom allowed in non-object-oriented languages. With the `Employee` and `PartTimeEmployee` classes, you need a different `calculateWeeklyPay()` method, depending on the type of object you use. Without polymorphism, you must write a different module with a unique name for each method because two methods with the same name cannot coexist in a program. Just as your blender can produce juice whether you insert a fruit or a vegetable, with polymorphism a `calculateWeeklyPay()` method produces a correct result whether it operates on an `Employee` or a `PartTimeEmployee`. Similarly, you may want a `computeGradePointAverage()` method to operate differently for a pass-fail course than it does for a graded one, or you might want a word-processing program to produce different results when you press Delete with one word highlighted in a document than when you press Delete with a file name highlighted.

When you write a polymorphic method in an object-oriented programming language, you must write each version of the method, and that can entail a lot of work. The benefits of polymorphism do not seem obvious while you are writing the methods, but the benefits are realized when you can use the methods in all sorts of applications. When you can use a single, simple, easy-to-understand method name such as `printFieldValues()` with all sorts of objects, such

as `Employee`s, `PartTimeEmployee`s, `InventoryItem`s, and `BankTransaction`s, then your objects behave more like their real-world counterparts and your programs are easier to understand.

UNDERSTANDING PROTECTED ACCESS

Making data private is an important object-oriented programming concept. By making data fields private, and allowing access to them only through a class's methods, you protect the ways in which data can be altered.

When a data field within a class is private, no outside class can use it—including a child class. It can be inconvenient when a child class's methods cannot directly access its own inherited data. However, the principle of data hiding would be lost if you had to make a class's data public (and therefore available for use by anyone) just so a child class could access its inherited fields. Therefore, object-oriented programming languages allow a medium-security access specifier that is more restrictive than `public` but less restrictive than `private`. The **protected access** modifier is used when you want no outside classes to be able to use a data field directly, except classes that are children of the original class. Figure 13-10 shows a rewritten `Employee` class that uses the `protected` access modifier on its data fields (see highlighting). When this modified class is used as a base class for another class such as `PartTimeEmployee`, the child class's methods will be able to access each of the protected fields originally defined in the parent class.

FIGURE 13-10: `Employee` CLASS USING `protected` AND `public` ACCESS SPECIFIERS

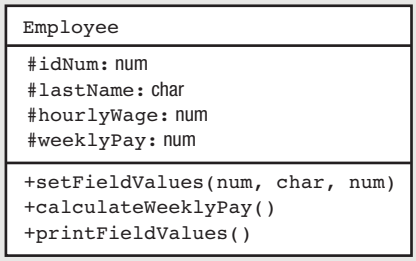
```
class Employee
      protected num idNum
      protected char lastName
      protected num hourlyWage
      protected num weeklyPay

      public setFieldValues(num id, char last, num rate)
         const num MAX_RATE = 25.00
         idNum = id
         lastName = last
         if rate <= MAX_RATE then
            hourlyWage = rate
         else
            hourlyWage = MAX_RATE
         endif
      return

      public calculateWeeklyPay()
         const num WORK_WEEK = 40
         weeklyPay = hourlyWage * WORK_WEEK
      return

      public printFieldValues()
         print idNum, lastName, hourlyWage, weeklyPay
      return
endClass
```

TIP

Although a child class's methods can access nonprivate data fields originally defined in the parent class, a parent class's methods have no special privileges regarding any of its child's data fields. That is, unless the child class's data fields are public, a parent, just like any other unrelated class, cannot access them.

Figure 13-11 contains the class diagram for the version of the `Employee` class shown in Figure 13-10. Notice the octothorpe (#) is used to indicate protected class members.

FIGURE 13-11: `Employee` CLASS DIAGRAM WITH `protected` AND `public` ACCESS SPECIFIERS



TIP

Instead of creating the parent class fields to be protected, you might choose to keep them private and provide protected or public methods that each return a field value. For example, the `Employee` class could contain a method such as the following:

```
public num getID()
return idNum
```

The child class would then use the public method to access `idNum`, just as any other method would. Using this technique, the parent class data would remain private, satisfying those who feel that all data within classes should be private.

UNDERSTANDING THE ROLE OF THE `this` REFERENCE

After you create a class such as the `Employee` class in Figure 13-10, any number of `Employee` objects might eventually be instantiated from it. Each `Employee` will have its own `idNum`, `lastName`, and other values, and enough computer memory must be set aside to hold all the attributes needed for each individual `Employee`. Each `Employee` object also will have access to each method within the class, but because each `Employee` uses the same set of methods, it would be a waste of memory resources to store a separate copy of each method for each `Employee`. Luckily, in OOP languages, just one copy of each method in a class is stored, and all instantiated objects can use that copy.

When you use an instance method with an object, you use the object name, a dot, and the method name—for example, `clerk.setFieldValues()`. When you execute the `clerk.setFieldValues()` method, you are running the general, shared `Employee` class `setFieldValues()` method; the `clerk` object has access to the method because it is a member of the `Employee` class. However, within the `setFieldValues()` method, when you access the `idNum` *field*, you access the `clerk`'s private, individual copy of the field. Because many

`Employee` objects might exist, but just one copy of the method exists no matter how many `Employee`s there are, when you call `clerk.setFieldValues()`, the compiler must determine *whose* copy of the `idNum` value should be set by the single `setFieldValues()` method.

The compiler accesses the correct object's field because when you make the function calls, you implicitly (automatically) pass the memory address of `clerk` to the `setFieldValues()` method. Depending on the language you use to write your programs, an object's memory address is called a **reference** or is said to be held in a **reference variable** or **pointer variable**. Therefore, the memory address of an object that is passed to any instance method of the same class is called the **`this` reference** or the **`this` pointer**. The word `this` is a reserved word in most OOP languages, and the syntax you employ to use it is a little different in each language. However, you can write pseudocode like that shown in Figure 13-12 to explicitly use the `this` reference. The two `setFieldValues()` methods shown in Figure 13-12 perform identically. The first method simply uses the `this` reference without you being aware of it; the second method uses the `this` reference explicitly. In Figure 13-12, you can interpret `this.idNum` to mean the ID number of "this current instance of the class"—that is, the specific instance of the `Employee` class that was used to call the `setFieldValues()` method. Similarly, `this.lastName` and `this.hourlyWage` refer to the data fields for the current object.

FIGURE 13-12: TWO VERSIONS OF THE `setFieldValues()` METHOD, WITH AND WITHOUT AN EXPLICIT `this` REFERENCE

```
public setFieldValues(num id, char last, num rate)
   const num MAX_RATE = 25.00
   idNum = id
   lastName = last
   if rate <= MAX_RATE then
       hourlyWage = rate
   else
       hourlyWage = MAX_RATE
   endif
return
```

```
public setFieldValues(num id, char last, num rate)
   const num MAX_RATE = 25.00
   this.idNum = id
   this.lastName = last
   if rate <= MAX_RATE then
       this.hourlyWage = rate
   else
       this.hourlyWage = MAX_RATE
   endif
return
```

Frequently, you neither want nor need to refer to the `this` reference 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.

TIP

In most object-oriented programming languages, you can create class methods that do not receive a `this` reference and do not require an object to execute. Such methods are called **static methods**.

USING CONSTRUCTORS AND DESTRUCTORS

When you create a class such as `Employee`, and instantiate an object with a statement such as `Employee chauffeur`, you are actually calling a method named `Employee()` that is provided by default by the compiler of the object-oriented language in which you are working. A constructor method, or more simply, a **constructor**, is a method that establishes an object. A constructor has the same name as its class.

When the automatically supplied, prewritten constructor method for the `Employee` class is called (the constructor is the method named `Employee()`), it establishes one `Employee` object with the identifier provided—for example, `chauffeur`. Depending on the programming language, a constructor might automatically provide initial values for the object's data fields. If you do not want an object's fields to hold these default values, or if you want to perform additional tasks when you create an instance of a class, then you can write your own constructor. Any constructor you write must have the same name as the class it constructs, and constructor methods cannot have a return type. Normally, you declare constructors to be public so that other classes can instantiate objects that belong to the class.

For example, if you want every `Employee` object to have a starting salary of $300.00 per week, then you could write the constructor method for the `Employee` class that appears in Figure 13-13. Any `Employee` object instantiated will have a `salary` field value equal to 300.00, and the other `Employee` data fields will contain the default values.

FIGURE 13-13: AN `Employee` CLASS CONSTRUCTOR

```
public Employee()
     salary = 300.00
return
```

TIP

You can create a method with a name like `setDataFields()` to assign values to individual `Employee` objects after construction, but a constructor method assigns the values at the time of creation.

Alternatively, you might choose to create `Employee` objects with initial `idNum` values that differ for each `Employee`. To accomplish this when the object is instantiated, you can pass an employee number to the constructor; that is, you can write constructor methods that receive arguments. A **default constructor** is one that requires no arguments; a **nondefault constructor** requires arguments.

TIP

The automatically supplied constructor for a class is a default constructor. For any class you write, you can create your own default constructors, your own nondefault constructors, or both.

Figure 13-14 shows an `Employee` class containing a constructor that receives an argument. With this constructor (shaded in the figure), an argument is passed using a statement, such as `Employee chauffeur(881)`. When the constructor executes, the numeric value within the method call is passed to `Employee()` as the argument `id`, which is assigned to `idNum` within the constructor.

FIGURE 13-14: `Employee` CLASS WITH CONSTRUCTOR THAT ACCEPTS A VALUE

```
class Employee
     private num idNum
     // other data fields can be defined here
     public Employee(num id)
          idNum = id
     return
     // other methods can be defined here
endClass
```

When you create an `Employee` class with a constructor such as the one shown in Figure 13-14, then you must create every `Employee` object using a numeric argument (which can be a constant such as 881 or a variable). In other words, with this new version of the class, the declaration statement `Employee chauffeur` no longer works. Once you write a constructor for a class, you no longer receive the automatically written default constructor. If a class's only constructor requires an argument, then you must provide an argument for every object of that class that you create. However, you can create multiple constructors for a class as long as every constructor has a different argument list. So, if it suited your purposes, the `Employee` class could contain one default constructor, one that accepted a single numeric argument, and one that accepted three arguments.

Object-oriented programming languages also provide an automatically called method that executes when an object is destroyed. The method is a **destructor**. Like constructors, you can write your own destructors, although only one version can exist for a class. Usually you write your own destructor if you need to complete cleanup tasks when an object is destroyed, such as closing open files. Although you can purposely destroy an object, most often an object is destroyed when the method in which it is declared ends.

TIP A destructor has the same name as its class constructor (and therefore the same name as the class). In Java, C++, and C#, a destructor name is preceded by a tilde (~).

ONE EXAMPLE OF USING PREDEFINED CLASSES: CREATING GUI OBJECTS

When you purchase or download an object-oriented programming language compiler, it comes packaged with myriad predefined, built-in classes. The classes are stored in **libraries**—collections of classes that serve related purposes. Some of the most useful are the classes you can use to create GUI objects such as frames, buttons, labels, and text boxes. You place these GUI components within interactive programs so that users can manipulate them using input devices, most frequently a keyboard and a mouse. For example, using a language that supports GUI applications, if you want to place a clickable button on the screen, you instantiate an object that belongs to the already created class named `Button`. The `Button` class is already created and contains private data fields such as `text` and `height` and public methods such as `setText()` and `setHeight()` that allow you to place instructions on your `Button` object and to change its vertical size, respectively.

TIP In some languages, such as Java, libraries are also called **packages**.

TIP Languages that contain prewritten GUI object classes frequently refer to the class attributes as **properties**.

If no predefined GUI object classes existed, you could create your own. However, there would be several disadvantages to doing this:

- It would be a lot of work. Creating graphical objects requires a lot of code, and at least a modicum of artistic talent.
- It would be repetitious work. Almost all GUI programs require standard components such as buttons and labels. If each programmer created the classes that represent these components from scratch, a lot of work would be unnecessarily repeated.
- The components would look different in various applications. If each programmer created his or her own component classes, then objects like buttons would vary in appearance and operation in different applications. Users like standardization in their components—title bars on windows that are a uniform height, buttons that appear to be pressed when clicked, frames and windows that contain maximize and minimize buttons in predictable locations, and so on. By using standard component classes, programmers are assured that the GUI components in their programs have the same look and feel as those in other programs.

In programming languages that provide existing GUI classes, you often are provided with a **visual development environment** in which you can create programs by dragging components such as buttons and labels onto a screen and arranging them visually. Then you write programming statements to control the actions that take place when a user manipulates the controls by clicking them using a mouse, for example. Many programmers never create any classes of their own from which they will instantiate objects, but only write classes that are applications that use built-in GUI component classes. Some languages, particularly Visual Basic, lend themselves very well to this type of programming.

UNDERSTANDING THE ADVANTAGES OF OBJECT-ORIENTED PROGRAMMING

Using the features of object-oriented programming languages provides you with many benefits as you develop your programs. Whether you use classes you have created or use those created by others, when you instantiate objects in programs, you save development time because each object automatically includes appropriate, reliable methods and attributes. When using inheritance, you can develop new classes more quickly by extending classes that already exist and work; you need to concentrate only on new features that the new class adds. When using existing objects, you need to concentrate only on the interface to those objects, not on the internal instructions that make them work. By using polymorphism, you can use reasonable, easy-to-remember names for methods and concentrate on their purpose rather than on memorizing different method names.

CHAPTER SUMMARY

- Object-oriented programming is a style of programming that focuses on an application's data and the methods you need to manipulate that data. Objects both in the real world and in object-oriented programming are made up of attributes and methods. In object-oriented terminology, a class is a term that describes a group or collection of objects with common properties. An instance of a class is an existing object of a class. In object-oriented programming, procedures are called methods. Inheritance and polymorphism are important object-oriented programming concepts.
- 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 contains three parts: a name, optional data, and optional methods. A class diagram consists of a rectangle divided into three sections containing the name, attributes, and methods.
- Data hiding is the principle of keeping data private and inaccessible to outside classes. 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 support public access, which means that other programs and methods may use the methods that control access to the private data.
- When you write an object-oriented program, you create objects that are members of a class. You instantiate (or create) a class object (or instance) with a statement that includes the type of object and an identifying name. A program that uses a class object is a client of the class.
- The concept of class is useful because of its reusability; you can create new classes that are descendents of existing classes. The descendent classes (or child classes) can inherit all of the attributes of the original class (or parent class), or the descendent class can override those attributes that are inappropriate. Object-oriented programs use a feature called polymorphism to allow the same request—that is, the same method call—to be carried out differently, depending on the context.
- Object-oriented programming languages allow a medium-security access specifier that is more restrictive than `public` but less restrictive than `private`. The `protected` access modifier is used when you want no outside classes to be able to use a data field directly, except classes that are children of the original class.
- The compiler accesses the correct object's field because when you make the function calls, you implicitly (automatically) pass the memory address of the object to its class method. The memory address of an object that is passed to any object's instance method is called the `this` reference or the `this` pointer.
- When you create a class and instantiate an object with a statement, you are actually calling a constructor that is provided by default by the compiler of the object-oriented language in which you are working. A constructor is a method that establishes an object. Object-oriented programming languages also provide an automatically called destructor that executes when an object is destroyed. You can write your own constructors and destructors.

- When you purchase or download an object-oriented programming language compiler, it comes packaged with myriad predefined, built-in classes. The classes are stored in libraries—collections of classes that serve related purposes. Some of the most useful are the classes you can use to create graphical user interface (GUI) objects such as frames, buttons, labels, and text boxes.
- Using the features of object-oriented programming languages provides you with many benefits as you develop your programs. Whether you use classes you have created or use those created by others, when you instantiate objects in programs you save development time.

KEY TERMS

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

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

The **properties** of an object are the values of its attributes.

The **state of an object** is the collective value of all its attributes at any point in time.

A **class** is a term that describes a group or collection of objects with common properties.

An **instance** of a class is an existing object of a class.

In object-oriented programming, procedures are called **methods**.

Polymorphism is the object-oriented feature that allows you to create multiple methods with the same name, which will act differently and appropriately when used with different types of objects.

In object-oriented programs, when you create multiple methods with the same name but different argument lists, you **overload the method**.

Pure polymorphism occurs when one function body can be 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 is the concept that other classes should not alter an object's attributes—outside classes should only be allowed to make a request that an attribute be altered; then it is up to the class methods to determine whether the request is appropriate.

The **interface** is the user-friendly boundary between the user and the internal mechanisms of the device.

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

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.

Is-a is a phrase you can use to test whether an object is an instance of a class.

A **field** is a data item within, or attribute of, an object.

A **class diagram** is a tool used to describe a class; it consists of a rectangle divided into three sections.

Object-oriented programmers usually specify that their data fields will have **private access**, which means that the data cannot be accessed by any method that is not part of the class.

Object-oriented programmers usually specify that their methods will have **public access**, which means that other programs and methods may use the methods that control access to the private data.

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

To create a class object is to **instantiate** it.

A class is a **user-defined type**.

A class is a **programmer-defined type**.

A programming language is **extensible** when you can create new data types.

An **abstract data type** (ADT) is a type whose internal form is hidden behind a set of methods you use to access the data.

A **client of a class** is a program or method that uses a class object.

A **primitive data type** is a simple data type, as opposed to a class type.

A **descendent class**, also called a **child class**, **derived class**, or **subclass**, inherits the attributes of another class.

An **original class**, also called a **parent class**, **base class**, or **superclass**, is one that has descendents. In other words, it is a class from which other classes are derived.

A child class method with the same name and argument list as a parent class method **overrides**, or takes precedence over, the parent class version.

An **abstract class** is one that is created only to be a parent class and not to have objects of its own.

Some programming languages support **multiple inheritance**, in which a class can inherit from more than one parent.

The **protected access** modifier is used when you want no outside classes to be able to use a data field directly, except classes that are children of the original class.

A **reference** is a memory address.

A **reference variable** holds a memory address.

A **pointer variable** holds a memory address.

The **`this` reference** or the **`this` pointer** holds an object's memory address within a method of the object's class.

A **static method** is a class method that does not receive a `this` reference and does not require an object to execute.

A **constructor** is a method that establishes an object.

A **default constructor** is one that requires no arguments.

A **nondefault constructor** is one that requires arguments.

A **destructor** is a method that destroys an object.

Libraries, or **packages**, are collections of classes that serve related purposes.

Properties are the attributes of prewritten GUI classes.

A **visual development environment** is one in which you can create programs by dragging components such as buttons and labels onto a screen and arranging them visually.

REVIEW QUESTIONS

1. **Which of the following is *not* a feature of object-oriented programming?**
   a. You pass messages to objects.
   b. Programming objects mimic real-world objects.
   c. Encapsulation is avoided.
   d. Classes can inherit features of other classes.

2. **With object-oriented programming, the same message _____.**
   a. works the same way with every object
   b. works differently and appropriately when applied to different objects
   c. can never be used more than once
   d. all of the above

3. **In object-oriented programming, the process of acquiring the traits of one's predecessors is known as _____.**
   a. inheritance
   b. polymorphism
   c. data redundancy
   d. legacy programming

4. **`Class` is to object as `Dog` is to _____.**
   a. animal
   b. mammal
   c. poodle
   d. my dog Murphy

5. **To programmers, another word for object is _____.**
   a. class
   b. instance
   c. structure
   d. item

6. **The object-class relationship can be tested using the phrase _____.**
   a. can-do
   b. open-close
   c. is-a
   d. can-be

7. **Which of the following is least likely to be a feature contained within most classes?**
   a. a name
   b. private data
   c. public data
   d. public methods

8. **Another term used for class data fields is _____.**
   a. attributes
   b. components
   c. points
   d. paths

9. **A class diagram consists of a rectangle divided into _____.**
   a. two sections: data and methods
   b. three sections: name, data, and methods
   c. four sections: name, data, methods, and purpose
   d. five sections: name, numeric data, text data, methods, and purpose

10. **The principle of keeping data private and inaccessible to outside classes is called _____.**
    a. information overloading
    b. attribute secrecy
    c. polymorphism
    d. data hiding

11. **Object-oriented programmers usually specify that their data fields will have _____ access.**
    a. `public`
    b. `private`
    c. `protected`
    d. `personal`

12. **Creating an object is called _____ the object.**
    a. morphing
    b. declaring
    c. instantiating
    d. formatting

13. **A method is often like a black box, meaning it contains some _____.**
    a. elements you can use, but cannot see
    b. elements you can see, but cannot use
    c. elements you can neither see nor use
    d. none of the above; it contains nothing

14. **One name for a class from which others inherit is a _____ class.**
    a. benefactor
    b. child
    c. descendent
    d. parent

15. Suppose you have a class named `Horse` containing such fields as `name` and `age`. When you create a child class named `RaceHorse`, _____.

a. every `RaceHorse` object has an `age` field
b. some `RaceHorse` objects have an `age` field
c. no `RaceHorse` objects have an `age` field
d. `Horse` objects no longer have an `age` field

16. Suppose you have a class named `Horse` containing such fields as `name` and `age`. When you create a child class named `RaceHorse` and add a new field named `winnings`, _____.

a. every `RaceHorse` object has a `winnings` field
b. some `RaceHorse` objects have a `winnings` field
c. every `Horse` object has a `winnings` field
d. every `Horse` object and every `RaceHorse` object has a `winnings` field

17. The feature of object-oriented programming languages that allows the same method call to be carried out differently, depending on the context, is _____.

a. inheritance
b. ambiguity
c. polymorphism
d. overriding

18. The access specifier that is more liberal than `private`, but not as liberal as `public`, is _____.

a. `semiprivate`
b. `sheltered`
c. `protected`
d. `constrained`

19. Collections of classes that serve related purposes are called _____.

a. archives
b. anthologies
c. compendiums
d. libraries

20. Which of the following is *not* a benefit provided by object-oriented programming?

a. You save development time because each object automatically includes appropriate, reliable methods and attributes.
b. When using inheritance, you can develop new classes more quickly by extending classes that already exist and work; you need to concentrate only on new features that the new class adds.
c. When using existing objects, you need to concentrate only on the interface to those objects, not on the internal instructions that make them work.
d. By using method overloading and polymorphism, you can use more precise and unique names for each operation you want to perform using different objects.

FIND THE BUGS

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **The `Date` class contains a month, day, and year, and methods to set and display the values. The month cannot be set to more than 12, and the day of the month cannot be set to more than 31. The demonstration program instantiates four `Date`s and purposely assigns invalid values to some of the arguments; the class methods will correct the invalid values.**

```
class Date
   private num month
   private num day
   private num year
   public setMonth(num)
   public setDay()
   public setYear(num)
   public showDate()
return

setDate(num m, num d)
   const num HIGH_MONTH = 12
   const num HIGH_DAY = 31
   if m < HIGH_MONTH then
      month = HIGH_MONTH
   else
      m = month
   endif
   if d > HIGH_DAY then
      day = HIGH_DAY
   else
      day = day
   endif
   y = year
return

showDate()
  print "Date: ", month, "/", day, "/", year
return
```

```
start
     Date birthday, anniversary, graduation, party
     birthday.setDate(6, 24, 1982)
     anniversary.setDate(10, 15, 2007)
     graduation.setDate(14, 19, 2008)
     party.setDate(7, 35, 2006)
     print "Birthday "
     birthday.showDate()
     print "Anniversary "
     anniversary.showDate()
     print "Graduation "
     graduation.showDate()
     print "Party "
     party.showDate()
stop
```

2. The `GroceryItem` class sets fields for an item for sale in a grocery store. The `dataEntry()` function ensures that the stock number is within legal range (1000 through 9999) and that the quantity and price are non-negative. The demonstration program declares a grocery object, sets its fields, and displays the object's data.

```
class GroceryItem
   private num stockNum
   private num priceEach
   private num quantity
   private num totalValue
   public dataEntry()
   public displayGroceryItem()
   setStockNum()
      const num LOW = 1000
      const num HIGH = 9999
      print "Enter stock number - use 4 digits "
      input stock
      while num < LOW OR stockNum < HIGH
         print "Use 4 digits please "
         input stock
      endwhile
      print "Enter price each "
      input price
      while priceEach = 0
         print "Price must be non-negative "
         input PriceEach
      endwhile
      print "Enter quantity in stock "
      input quantity
```

```
            if quantity < 0
                print "Quantity must be non-negative "
                input quantity
             endwhile
             totalValue = quantity * price
         return

         displayGroceryItem()
            print "ID #", stockNum, " Price:$", priceEach
            print "Quantity in stock ", quan
            print "Value $", total
         return
      endClass

      start
           GroceryItem oneItem
           dataEntry()
           displayGroceryItem
      stop
```

EXERCISES

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

2. **For each of the following objects, identify three different classes that might contain it:**
   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 print 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 prints 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 a method that sets all the data fields and another that prints the values for each data field. Create the class diagram and write the pseudocode that defines the class.
   b. Design a class named `TextBook` that is a child class of `Book`. Include a new data field for the grade level of the book. Override the `Book` class methods that set and print the data so that you accommodate the new grade-level field. Create the class diagram and write the pseudocode that defines the class.

7. **Complete the following tasks:**
   a. Design a class named `Player` that holds a player number and name for a sports team participant. Include a method that sets the values for each data field and another that prints the values for each data field. Create the class diagram and write the pseudocode that defines the class.
   b. Design two classes named `BaseballPlayer` and `BasketballPlayer` that are child classes of `Player`. Include a new data field in each class for the player's position. Include an additional field in the `BaseballPlayer` class for batting average. Include a new field in the `BasketballPlayer` class for free-throw percentage. Override the `Player` class methods that set and print the data so that you accommodate the new fields. Create the class diagram and write the pseudocode that defines the class.

8. **Complete the following tasks:**
   a. Design a class named `PlayingCard`. Its attributes include `suit` ("Clubs", "Diamonds", "Hearts", or "Spades"), `value` (a number 1 through 13), and `valueName`. If a `PlayingCard`'s value is between 2 and 10 inclusive, the `valueName` is blank; however, if the value is 1, the `valueName` is "Ace", and if it is 11, 12, or 13, the `valueName` is "Jack", "Queen", or "King", respectively. Create two overloaded constructors for the class. One takes no arguments and uses a built-in method that returns a randomly generated number. This method's signature is `num rand(num high)`, where `high` represents the highest value the method might return—the method returns a random number between 0 and this high value inclusive. Use the random-number-generating method to select both the suit and the value for any `PlayingCard` that uses the default constructor. The second constructor method assigns values to the `PlayingCard` attributes based on passed arguments. This constructor verifies that the passed arguments are within range (that is, only one of the four allowed suits and only one of the 13 allowed values); if the values are out of range, force the `PlayingCard` to be the Ace of Spades. This nondefault constructor also sets the `valueName` for cards valued at 11 through 13. Also create a `showCard()` method for the class that displays a `PlayingCard`'s value.
   b. Design the logic for a program that instantiates two `PlayingCard` objects. Allow one object's values to be randomly generated, but use user input values for the second object.
   c. Add any additional class methods you need so that you can create a game in which the user tries to guess the value of a randomly generated `PlayingCard`. Give the user 1 point for guessing the suit correctly, 2 points for guessing the value correctly, and 10 points for guessing both values of the `PlayingCard` correctly.

DETECTIVE WORK

1. **Many programmers think object-oriented programming is a superior approach to procedural programming. Others think it adds a level of complexity that is not needed in many scenarios. Find and summarize arguments on both sides.**
2. **When and why was the Java programming language created?**

UP FOR DISCUSSION

1. **Do you think all class data should be private? Is protected class data justified when the class will serve as a base class, or does it violate the principles of data hiding? Should any class data ever be public?**
2. **Many object-oriented programmers are opposed to using multiple inheritance. Find out why and decide whether you agree with this stance.**