Objects in the Everyday World

The chapter you're about to read is about objects, and you're probably wondering what on earth that means. The answer is simple: Anything that has properties and a function (or functions) is an object. Objects are all around us—the chair you're sitting on is an object, this book is an object, and so is the washing machine you use to clean your clothes. Even you are an object.

Consider the washing machine. It certainly has properties—it's made of metal and has a tub, motor, and gearbox, has certain dimensions, and weighs a few hundred pounds. After writing a long list of its properties, we may know what a washing machine looks like (which is fine), but we don't yet know enough to define it. We also have to talk about its functions, the processes it carries out: The machine turns on, fills with water, agitates, empties, rinses, spins, and turns off. Finally, we need to know what our object “works on”; in this case, clothes. Put together all these pieces—properties, functions, and something to work on—and we can completely describe a useful object.

Here are a couple of other important aspects of a washing machine or, for that matter, any useful object:

- You don't have to know how it works internally to be able to use it.
- If someone has already built a suitable one, and it's available for purchase (or, better yet, for free), you don't have to build it yourself.

As another example, consider the chair mentioned above. Here's how we could describe one of these objects:

- Properties: Made of wood, has four legs, has a seat, has a back, is light brown
- Functions: Holds up a weight applied to the seat and back
- Works on: A person

In programming, objects containing properties (data) and functions (processes) provide “packaged solutions” to help us solve our programming problems. Defining and creating objects may initially seem unnecessarily complicated, but their use leads to elegant and efficient ways to handle complex problems. Moreover, by their very nature, objects ultimately simplify the programming process and ensure that we don't have to re-invent the wheel (or washing machine or chair).
Throughout this book, we have used a single approach in developing our more complicated programs: top-down, modular design. In this chapter, we will discuss two other approaches to program design—object-oriented and event-driven programming. In Section 2.5, we provided a brief introduction to these topics; here, we will explore them in more depth. We will first discuss the basic concepts that underlie object-oriented programming (OOP) and then apply this material to the objects that make up a typical graphical user interface (GUI). To be more specific, you will learn

1. The basic terminology used in object-oriented programming [Section 8.1].
2. How to define classes and create objects [Section 8.1].
3. About the inheritance and polymorphism features of OOP [Section 8.2].
4. About object-oriented program design [Section 8.2].
5. About the objects used in creating a graphical user interface [Section 8.3].
6. How to handle events in programming for a GUI [Section 8.4].
7. About event-driven program design [Sections 8.4, 8.5].
8.1 Classes and Objects

As you may recall from Section 2.5, an object is a structure comprised of data (or attributes) and processes (or methods) that perform operations on that data. Object-oriented programming (or OOP, for short) refers to an approach to program design and coding that places an emphasis on the objects needed to solve a given problem and the relationships among them. In this section, we will discuss some of the basic concepts underlying object-oriented programming.

Some Terminology

When learning a new subject, coming to grips with its special terminology is not always easy. In the case of object-oriented programming, we have the additional complication that there are often several terms that describe the same concept. On the other hand, as you will see, many of the new terms and concepts are analogous to ones with which you are already familiar.

The fundamental entity in object-oriented programming is the class. A class is a data type that allows us to create objects; it provides the definition for a collection of objects by describing its attributes (data) and specifying the methods (operations) that may be applied to that data. For example, consider the following definition of the class “television”:

- Its attributes include brand name, model number, dimensions, screen size, and cabinet color.
- Its methods include turn it on, change channels, change volume, and turn it off.

This “television class” just describes what a television is and what can be done with it; a “television object,” on the other hand, is a particular example of a television, such as a Sony XBR32S.

So, to put it simply, the purpose of defining a class is to allow us to create objects; an object is just a particular instance of its class. The relationship between a class and its objects is analogous to the relationship between a data type and variables of that type. For example, when we write

```
Declare Number As Integer
```

the type Integer states what kind of data we are dealing with and what operations (+, -, etc.) can be performed on it. The variable Number is a particular instance of the type Integer. It can be assigned a specific integer value to be used within the program.

Let us now take a closer look at the objects themselves. As you know, objects are made up of two components: data and operations on that data. (We say that an object encapsulates—packages together—data and operations.) The operations are specified in the class definition; the data are specific to the particular object under consideration (although the type
of data is also specified in the class definition). Be aware that there are several alternate names for the two components that make up an object:

- An object’s data are known as its attributes, properties, or state.
- An object’s operations are called methods, behaviors, services, procedures, or functions.

In this book, we will usually use the terms attributes (for data) and methods (for operations).

**Defining Classes and Creating Objects**

If you want to use objects in a program, the first step is to define a class for each kind of object. The class definition provides the structure of the objects in it—the attributes they possess and the methods that may be applied to them. The following example illustrates the kind of pseudocode we will use to define a class.

**EXAMPLE 1**

Recall that a *cube* is a box-shaped solid in which all sides are of equal length and whose volume is obtained by taking the third power of the length of a side, \( V = s^3 \). Suppose that we want to define a class called Cube that will have

- **Attributes**: the length of a side (Side) and the volume (Volume) of the cube
- **Methods** that:
  - assign a value to the side (SetSide)
  - compute the volume of the cube (ComputeVolume)
  - return the value of the side to the program (GetSide)
  - return the volume of the cube to the program (GetVolume)

(We will discuss the specifics of these methods later in this section.)

To define the class Cube, we use the following pseudocode:

```
Class Cube
    Side As Real
    Volume As Real
    Subprogram SetSide(NewSide)
        Set Side = NewSide
    End Subprogram
    Subprogram ComputeVolume()
        Set Volume = Side ^ 3
    End Subprogram
    Function GetVolume() As Real
        Set GetVolume = Volume
    End Function
    Function GetSide() As Real
        Set GetSide = Side
    End Function
End Class
```
The Function and Subprogram notation we have used in this pseudocode was introduced in Chapter 7. Recall that the variables within parentheses in the subprogram header (for example, NewSide in the subprogram SetSide) are called parameters. If a particular subprogram has no parameters, notice that we still write the parentheses, ()—see, for example, the subprogram ComputeVolume.

In Example 1, the methods SetSide, GetSide, and GetVolume, are called access methods. They provide the rest of the program with access to the object’s attributes. SetSide imports a value of the attribute Side from the main program. GetSide and GetVolume allow the main program to make use of the values of Side and Volume. This raises the question: “Why not just pass the values of the variables Side and Volume back and forth to the program as parameters? The answer to this question is one of the keys to understanding OOP: In object-oriented programming, we normally want to keep the class variables completely “hidden” from the rest of the program. This practice of data hiding has a two-fold purpose:

1. It enhances the security of the object’s data—the data cannot be altered except by the means intended, namely, by using one of the object’s methods.
2. It helps shield the inner workings of the object from the programmer.

In OOP, objects work like “black boxes.” Although their interface with the outside programming world (their methods) is made public, the way in which a method gets its job done and the variables with which it works is kept private.

To explicitly state which members (attributes and/or methods) of a class are public (available to code outside an object of that class) and which are private to the class (not available outside the class), programming languages use the keywords Public and Private. The relevant keyword, placed in front of the variable or method name, specifies the status of that class member. For instance, in Example 1, to declare all variables as private and all methods as public, we would rewrite the pseudocode as:

```
Class Cube
  Private Side As Real
  Private Volume As Real
  Public Subprogram SetSide(NewSide)
    Set Side = NewSide
  End Subprogram
  Public Subprogram ComputeVolume()
    Set Volume = Side ^ 3
  End Subprogram
  Public Function GetVolume() As Real
    Set GetVolume = Volume
  End Function
```

PROGRAMMING POINTER

To explicitly state which members (attributes and/or methods) of a class are public (available to code outside an object of that class) and which are private to the class (not available outside the class), programming languages use the keywords Public and Private. The relevant keyword, placed in front of the variable or method name, specifies the status of that class member. For instance, in Example 1, to declare all variables as private and all methods as public, we would rewrite the pseudocode as:

```
Class Cube
  Private Side As Real
  Private Volume As Real
  Public Subprogram SetSide(NewSide)
    Set Side = NewSide
  End Subprogram
  Public Subprogram ComputeVolume()
    Set Volume = Side ^ 3
  End Subprogram
  Public Function GetVolume() As Real
    Set GetVolume = Volume
  End Function
```
Public Function GetSide() As Real
    Set GetSide = Side
End Function
End Class

Attributes are normally declared to be Private (to protect their integrity). Methods are declared as Public if they are part of the interface between the object and program, and as Private if they are only used internally—within the class itself.

Creating objects

Remember: Defining a class is analogous to creating a data type, and just as a data type (like Integer) cannot be referenced within the program, neither can the name of the class. So, once we have defined a class, we need to create one or more objects of that class, which can be referenced within the program. (In OOP language, we need to create an instance of the class; that is, we must perform an instantiation operation.) This is typically done by means of a declaration statement placed in the main program. For example, in this book, we use the statement

Declare Cube1, Cube2 As Cube

to create two objects, named Cube1 and Cube2, of the class Cube.

Once they are created, we can make use of the objects Cube1 and Cube2 in our program, but we need a notation for doing so. In this book, we will use a “dot notation” that allows us to refer, using a single expression, to both the object and method or attribute under consideration. For example, to assign a value of 10 to the Side attribute of Cube1 (see Example 1), we will use the statement:

Call Cube1.SetSide(10)

This statement calls the subprogram (method) SetSide, assigning the value 10 to its argument, NewSide, in the process. To ensure that this method is setting the Side of the object Cube1 (not that of Cube2) equal to 10, we place Cube1 in front of the subprogram name, separated from it by a dot (period). As another example, to display the Volume attribute of Cube2, we use the statement:

Write Cube2.GetVolume

In general, to refer to a public member (attribute or method) called MemberName, of an object called ObjectName, we use the notation:

ObjectName.MemberName

The next example further illustrates this notation.

EXAMPLE 2

The following program makes use of the Cube1 object in the class Cube (defined in Example 1). This program inputs a number from the user that represents the length of the side of a cube and displays the volume of that cube.
Main
    Declare Cube1 As Cube
    Write “Enter the length of the side of a cube:”
    Input Side1
    Call Cube1.SetSide(Side1)
    Call Cube1.ComputeVolume()
    Write “The volume of a cube of side”, Cube1.GetSide
    Write “is”, Cube1.GetVolume
End Program

Notice that there are four calls in this pseudocode to the methods of the object Cube1; two subprogram calls—to SetSide and ComputeVolume—and two function calls—to GetSide and GetVolume. (In OOP language, each of these calls is referred to as a message to the appropriate method.)

☞ The first subprogram call, Call Cube1.SetSide(Side1), assigns the number input to the subprogram variable Side.
☞ The next subprogram call, Call Cube1.ComputeVolume(), sets Volume = Side^3.
☞ The statement Write “is”, Cube1.GetVolume displays the value of Volume.

In Example 2, a program error will result if the main program calls the function ComputeVolume before its Side attribute is given a value. To prevent this (and for other reasons, as well), object-oriented programming languages supply an easy way, through the use of a constructor, to initialize an object’s attributes. A constructor is a special method included in the class definition that automatically performs specified setup tasks (such as initializing the object’s attributes) when an object is created.

Reading Check 8.1
1. What are the two major components of an object?
2. What is the relationship between a class and objects in that class?
3. What is the difference between public and private class members?
4. Define a class called InAndOut that has one attribute named Value (of type Real) and two methods:
   ◆ SetValue imports a value of the attribute Value from the main program.
   ◆ GetValue returns the value of the attribute Value to the main program.
5. What is a constructor?
More on Object-oriented Programming

In this section, we will continue our discussion of object-oriented programming (OOP). We will describe additional features that provide OOP with versatility and power, and also discuss the process of object-oriented program design.

Inheritance and Polymorphism

To contrast it to OOP, the approach to programming that makes use of top-down modular design is known as procedural programming. Most early programming languages, such as FORTRAN and BASIC, did not support the use of classes and objects, and are known as procedural languages. Most modern languages, on the other hand, allow the programmer to make use of objects, in addition to providing the tools found in a procedural language. However, to take full advantage of the power of OOP—to truly support object-oriented programming—a language must include the following features:

- **Encapsulation**—The incorporation of data and operations on that data into a single unit in such a way that the data can only be accessed through these operations. This is the fundamental idea behind classes and objects.

- **Inheritance**—The ability to create new classes that are based on existing ones. The methods (operations) and attributes (data) of the original class are incorporated into the new class, together with methods and attributes specific to the latter.

- **Polymorphism**—The ability to create methods that perform a general function, which automatically adapts itself to work with objects of different classes.

Some Benefits of Object-oriented Languages

Object-oriented programming tools have been in use for several decades, but only started to gain great popularity in the late 1980s. There are two basic reasons for this development:

- During the 1980s, programs became more complex as demand grew for sophisticated applications such as word processors, graphics programs, and computer games. Programs of this sort had so many options and possible outcomes that keeping track of their subprograms became a nightmare. Due to the self-contained nature of objects and the properties of inheritance and polymorphism, OOP is better equipped than procedural programming to deal with these extremely complex programs.

- The graphical user interface (GUI), popularized by the Apple Macintosh in the mid-1980s, slowly became almost universal for computers. A GUI is comprised of objects (windows, boxes, buttons, etc.), so OOP became the natural way of programming for these interfaces.
The concept of encapsulation was discussed in Section 8.1; inheritance and polymorphism will be described below.

**Inheritance** In everyday life, we often classify things. For example, trucks and cars are kinds of vehicles, and station wagons, convertibles, and sedans are all types of cars. In a more mathematical way, we can say that the set of cars is a subset of the set of vehicles and the set of convertibles is a subset of the set of cars. To picture the relationships among these sets, we can use a kind of hierarchy chart, as shown in Figure 1.

Classifying objects can help explain what they are and how they operate. For example, if we know what a car is, then in describing a convertible, we don’t have to explain the attributes and functions that it has in common with (that it has *inherited from*) a car. We just state that a convertible is a car, and present the special features of a convertible that distinguish it from a car.

This concept of classification and inheritance also works in object-oriented programming. Object-oriented languages allow us to create a subclass of an existing class. In this case, the existing class is called the **base class** (or parent class) and the subclass is called the **derived class** (or child class). In creating a subclass, the attributes and methods of the base class automatically become members of the derived class, together with any additional attributes and methods defined specifically for the latter.

In OOP, the derived class may or may not be a special type (a subset) of the base class. Rather, a derived class is created to take advantage of the methods that have already been defined in an existing class. As an example, recall the “Cube” class of Section 8.1. Its members are:

- **Attributes**—Side, Volume
- **Methods**—SetSide, ComputeVolume, GetSide, GetVolume
We may consider a cube to be a special type of box—one in which all sides are equal. If we want to define a class that models another kind of box, one with a square base but whose height is not the same as the sides of its base, we need not start from scratch; we can define this new class to be a subclass of Cube. The next example demonstrates how to do this.

**EXAMPLE 3**

The following pseudocode gives the definitions of the class Cube (from Section 8.1) and its subclass, SquareBox. The SquareBox class makes use of all attributes and methods of the Cube class (although it changes the definition of the ComputeVolume method) and adds an attribute and two methods of its own.

```plaintext
Class Cube
    Side As Real
    Volume As Real
    Subprogram SetSide(NewSide)
        Set Side = NewSide
    End Subprogram
    Subprogram ComputeVolume()
        Set Volume = Side ^ 3
    End Subprogram
    Function GetVolume() As Real
        Set GetVolume = Volume
    End Function
    Function GetSide() As Real
        Set GetSide = Side
    End Function
End Class

Class SquareBox As Cube
    Height As Real
    Subprogram SetHeight(NewHeight)
        Set Height = NewHeight
    End Subprogram
    Function GetHeight() As Real
        Set GetHeight = Height
    End Function
    Subprogram ComputeVolume()
        Set Volume = Side ^ 2 * Height
    End Subprogram
End Class
```

In this pseudocode, notice that we specify SquareBox to be a subclass of Cube with the statement:

```plaintext
Class SquareBox As Cube
```

8.2 More on Object-oriented Programming
(Of course, each programming language has its own way of defining subclasses.) The derived class SquareBox has the following members:

- Its attributes are Side and Volume, both inherited from Cube, and Height.
- Its methods are SetSide and GetSide, inherited from Cube, and SetHeight, GetHeight, and ComputeVolume.

A statement (elsewhere in the program) like

```
Declare Box As SquareBox
```

creates an object called Box of the class SquareBox that can take advantage of all these attributes and methods. For example, the following statements assign the values 10 and 20 to the Side and Height attributes, respectively:

```
Call Box.SetSide(10)
Call Box.SetHeight(20)
```

In Example 3, notice that a ComputeVolume method appears in both the base class Cube and the derived class SquareBox. In such a case, for an object in the derived class, the definition in the derived class overrides (is used instead of) the one in the base class. The next example clarifies this notion.

**EXAMPLE 4**

With the definition of the SquareBox class as given in Example 3, the following pseudocode inputs values for the side and height of a box from the user and computes and displays its volume.

```
Main
   Declare Box As SquareBox
   Write “For a box with a square base and arbitrary height,”
   Write “enter the length of the sides of its base:”
   Input BoxSide
   Call Box.SetSide(BoxSide)
   Write “Enter the height of the box:”
   Input BoxHeight
   Call Box.SetHeight(BoxHeight)
   Call Box.ComputeVolume()
   Write “The volume of the box is ”, Box.GetVolume
End Program
```

When the message

```
Call Box.SetSide(BoxSide)
```

is sent, the computer "looks in" the class definition SquareBox (the class of the object Box) for the method SetSide. Not finding it there, the computer looks in the parent class Cube for this method and applies it to the
object Box, setting Box.Side equal to the value input. On the other hand, when the subprogram SetHeight is called, this method is found in the definition of SquareBox, and invoked. Similarly, when the message

Call Box.ComputeVolume

is sent, it is received by the method ComputeVolume defined in SquareBox. Thus, the correct formula, Volume = Side ^ 2 * Height, is applied at this point.

**Polymorphism** In a hierarchy of classes, it is often the case that some methods are common to several classes, but that their definitions may differ from class to class. For example, the ComputeVolume method of Example 3 calculates the volume of a cube in the base class Cube and the volume of a box with a square base in the derived class SquareBox. More generally, a module might contain definitions of many three-dimensional objects—cubes, boxes, spheres, cylinders, etc.—each of which has a volume given by a different formula. It would be convenient if a programmer were able to use a single method, ComputeVolume, to calculate the volume of any of these objects. Polymorphism allows for this kind of flexibility.

The word *polymorphism* means “many shapes.” In programming, it allows a method to take on many definitions when applied to objects in a hierarchy of classes. Although this OOP feature is implemented differently in various object-oriented languages, the next example shows the general way in which it works.

**EXAMPLE 5** The following pseudocode shows how a method can take on different forms when applied to different objects. It makes use of the class definitions given in Example 3 (earlier in this section).

```
Main
    Declare Box1 As Cube
    Declare Box2 As SquareBox
    Write "Enter the length of the side of a cube:"
    Input Side1
    Call Box1.SetSide(Side1)
    Call Box1.ComputeVolume()
    Write "The volume of this cube is ", Box1.GetVolume
    Write "For a box with a square base and arbitrary height,"
    Write "enter the length of the sides of its base:"
    Input Side2
    Call Box2.SetSide(Side2)
    Write "Enter the height of the box:"
    Input Height2
    Call Box2.SetHeight(Height2)
    Call Box2.ComputeVolume()
    Write "The volume of this box is ", Box2.GetVolume
End Program
```
The Declare statements create the objects Box1 and Box2 as instances of the classes Cube and SquareBox, respectively. Thus, for all statements that refer to Box1, the Cube class definition is referenced. In particular, when the first message is sent to the method ComputeVolume, the formula used is Volume = Side^3. On the other hand, for statements that refer to Box2, the SquareBox class definition is referenced; the parent Cube definition is used only when an attribute or method definition is not found in SquareBox. Thus, when the second message is sent to ComputeVolume (this time, preceded by “Box2.”), the definition in SquareBox is used, providing the formula Volume = Side^2 * Height, as desired.

**Object-oriented Program Design**

As you know, the top-down modular approach to program design (procedural programming) places an emphasis on determining the subprograms (procedures) that are needed in a program.

Object-oriented program design, on the other hand, places an emphasis on determining the objects needed to solve the given problem. In fact, in most cases in which OOP is better-suited to solve a given problem (such as in programming for a graphical user interface), there is no strong hierarchy of program modules.

Viewed in terms of the program development cycle—problem analysis, program design, program coding, and program testing—it is the analysis phase that differs the most between the two approaches. In developing an OOP program, the analysis phase entails:

1. Identifying the classes to be used in the program.
2. Determining the attributes needed for these classes.
3. Determining the methods needed for these classes.
4. Determining the relationships among the classes.

The manner in which these steps are carried out is not so different from that of the top-down approach. The program designer works with the program specifications, imagining what kinds of situations the program will encounter—what types of specific cases it must handle—and determines the needed objects, attributes, and methods from these scenarios.

In carrying out these steps, there is a natural transition into the design phase of the program development cycle. Here, as in the top-down technique, the methods (subprograms) must be defined. Although this is easy to say, in real-life programs, there may be hundreds or even thousands of methods. Encapsulating them in objects generally makes it easier to manage them. Finally, although the coding and testing phases proceed as in procedural programming, here too, a benefit of OOP emerges. It is likely that some pre-existing objects can be recycled in this new program; making use of this reusable code speeds up both coding and testing.

The next example illustrates the analysis phase of an object-oriented design.
EXAMPLE 6

Suppose we want to create a grade management program (like the one
developed in Section 7.5) to input and store student names and test scores,
and compute test averages for each student. An object-oriented solution to
this problem would begin as follows:

1. In analyzing the problem, we decide to use two classes of objects:
   - A Student class, whose objects are the students enrolled in the course.
   - A Test class, whose objects are the test scores and averages for
each student.
2. The attributes of the Student class are Name and ID number. (There
could be others, such as Major and Rank—Freshman, Sophomore, etc.).
The attributes of the Test class are an array of Scores and AverageScore.
3. The methods for the Student class allow access to its attributes; they
   are SetName, GetName, SetID, and GetID. The methods for the Test
   class are those that allow access to its attributes together with a sub-
   program, ComputeAverage, that calculates the average test score.
4. The classes Student and Test are related; each test score and aver-
age refers to a particular student. Thus, we take Test to be a class
derived from the parent class Student, so that the former inherits the
attributes and methods of the latter.

Now that we have determined the classes, attributes, and methods
needed to solve this problem, we must design, code, and test the corre-
sponding subprograms, as well as the program itself. (Section 8.5 presents
a GUI solution to the grade management problem.)

Reading Check 8.2

1. Fill in the blank: In contrast to object-oriented programming, the
top-down modular approach to programming is referred to as
   __________ programming.
2. Identify each of the following OOP features using a single word:
   a. The incorporation of data and operations on that data into a
      single unit in such a way that the data can only be accessed by
      making use of these operations.
   b. The ability to create new classes that are based on existing
      ones and to make use of their data and attributes.
   c. The ability to create methods that perform a general function
      which automatically adapts to work with certain related classes.
3. Fill in the blanks: When we create a subclass of an existing class,
   the new class is called the __________ class; the original is the
   __________ class.
4. List the fundamental steps taken in analyzing a problem to be
   solved using object-oriented program design.
5. What is one advantage of OOP over procedural programming?
8.3 ~ Graphical User Interfaces Revisited

One of the important applications of object-oriented programming (OOP) is to create programs that employ a graphical user interface. A graphical user interface (or GUI) is comprised of windows that contain components such as menus, buttons, boxes, etc., which allow users to make choices and issue commands in a simple, intuitive way. In Section 2.5, we introduced some basic GUI concepts. In this section and the next, we will explore this topic in greater detail.

Window Components

Figure 2 displays a typical dialog box window (the Print dialog box for the Microsoft Windows GUI). Its buttons, boxes, and other elements are collectively known as window components (or controls). From a programming point of view, both windows and window components are objects. The names and functions of common window components are described below; their attributes and methods are discussed later in this section and in Section 8.4.

~ Command buttons, like the Properties, OK, and Cancel buttons in Figure 2, are represented by labeled rectangles. When you click a command button (when you position the mouse pointer over it and tap the left mouse button), it initiates an action. For example, clicking the command button labeled OK in Figure 2 closes the dialog box (removes it from the screen) and sends part or all of the current document to the printer. The Cancel button closes the dialog box without performing any action and returns all items within it to the values they had when the box was opened.
A **check box**, like the “Print to file” check box in Figure 2, is represented by a small square, which may or may not contain a check mark. When the check mark is present (indicating that the specified option is in effect), we say that the check box is **selected**; when the check mark does not appear (indicating that this option is not in effect), the check box is said to be **deselected** or **cleared**. For example, the deselected check box in Figure 2 indicates that the document will not “Print to file.” To select a check box, the user clicks on it; to clear it, the user clicks on it again.

A **text box**, like the “from” and “to” text boxes in Figure 2, allows the user to input text or a number into the dialog box. A text box is symbolized by a rectangle which is either blank or contains the **current value** for that object—the text or number that is used if it is not changed. (In Figure 2, the current value for the “from” box is 1.) To enter data into a text box, the user clicks inside it, deletes the current value (if any), and then types the desired letters or digits.

**Option buttons**, which are represented by small circles, always appear in groups (like the “Print range” group in Figure 2). These kinds of buttons are sometimes called **radio buttons** because they work like the station selection push buttons on a radio—when the user clicks on one of them, that button is **selected** (turned on) and the others are automatically **deselected** (turned off). For this reason, in a group of option buttons, only one of the corresponding options can be selected at any given time. The currently selected option button is indicated by a **bullet** within the corresponding circle. For example, in Figure 2, the “All” option is selected; it contains the bullet. If the user clicks on the “Pages” option button, it will acquire the bullet and become the current selection.

A **drop-down list box**, like the one labeled Name in Figure 2, is used to display a list of items. The current value of this object (“HP OfficeJet R Series” in Figure 2) is the only item initially visible. To display the rest of the list, the user clicks on the downward-facing triangle at the right end of the box. Then, to select an item from the list, the user clicks on that item—the list will close and the new item will become the current value.

A **list box** (not shown in Figure 2) is like a drop-down list box, but displays the entire list of items, with the current value at the top. Again, to select an item from the list so that it becomes the current value, the user clicks on that item.

A **label** is text in the dialog box that is not part of another window component. It is often located near a component to help identify it. For example, in Figure 2, Printer, Name, Status, and Ready are all labels.
Creating GUI Objects in a Program

The major OOP languages have built-in classes (class libraries) that supply the objects needed to write programs for a graphical user interface. Thus, there is no need to define windows, command buttons, etc. The way that these objects are created within a program depends on the language used:

- In some programming languages, GUI objects are created in the same way as any other objects, as instances of the class to which they belong.
- In other programming languages, GUI objects are selected from a menu or toolbar and “drawn” on the screen.

For example, in the Java programming language, we add a command button to a window by using a New statement to create the object and an Add statement to place it in the window. In Visual Basic, on the other hand, we click on the command button icon on a toolbar and then “draw” the button (using the mouse) in the window; these actions create code that adds the button to the window. For simplicity’s sake, in this book we will assume that windows and window components can be created in the Visual Basic way, without explicitly writing any code.

Each window and window component (command button, text box, etc.) in a GUI has a set of attributes, or properties, such as name, position, and size. Here are some examples of GUI object properties.

- The properties of a window include:
  - Name (by which it’s known in the program)
  - Height and Width, (usually in pixels, the tiny dots that make up the screen image)
  - Title (“Area Calculator” in Figure 3)
  - Title Font—the typeface and size of the window title
  - Visible (True or False)—whether or not the window appears on the screen.

- Command button properties include:
  - Name (by which it’s known in the program)
Caption ("Done" or "Calculate" in Figure 3)
Position—distance (in pixels) from the left and top edges of the window
Enabled (True or False)—whether the button is active or inactive (the Calculate button is inactive, or "grayed out," and will not respond to a mouse click; the Done button is active)

☞ Text boxes (like the two in Figure 3) have properties similar to those of command buttons, but text boxes have a Text property (holding the character string that currently appears inside the box) instead of a Caption property.

☞ Labels (like the two in Figure 3) have properties that are very similar to those of text boxes.

☞ Option button properties include:
   Name (by which it’s known in the program)
   Caption ("All," "Pages," or "Selection" in Figure 2)
   Enabled (True or False)—whether the button is active or inactive (grayed out)
   Value (True or False)—whether or not the option button has a bullet

Since windows and window components are objects, their properties can be assigned values, just like the attributes of any object. Most GUI properties have default values—those that are automatically used unless new ones are assigned. To change these default values, we can make use of assignment statements. For example, in the Area Calculator window of Figure 3:

☞ If the name of the window is MainWindow, then the statements
   Set MainWindow.Title = "Area Calculator"
   Set MainWindow.Height = 100
specify that the title of the window is “Area Calculator” and that the height of the window is 100 pixels.

☞ If the name of the right command button is QuitButton, then the statements
   Set QuitButton.Caption = "Done"
   Set QuitButton.Enabled = False
label it as “Done” and cause it to be grayed out when the window is opened.

Remembering the names of all the possible properties of all window components isn’t easy. To simplify matters, some languages allow you to specify properties by selecting items from menus or entering values into dialog boxes.
As objects, windows and their components also have methods (procedures) associated with them. We will discuss this aspect of GUI programming in Section 8.4.

**Reading Check 8.3**

1. Identify each of the following window components as either a command button, an option button, a check box, or a text box.
   a. It can receive data input by the user.
   b. When clicked, it initiates an action.
   c. It allows the user to select one choice from a group of related choices.

2. What is the difference between a list box and a drop-down list box?

3. Give two properties of each of the following objects:
   a. A window
   b. A text box
   c. An option button

4. A command button has the name OKbutton. Write assignment statements that set its Caption property equal to “OK” and its Enabled property equal to False.

---

### 8.4 Event-driven Programming

Thus far in this chapter we have discussed object-oriented programming (OOP) and some of the objects used in creating a graphical user interface (GUI). In this section, we will continue this discussion, returning to the concepts of events and event-driven program design that were touched on briefly in Section 2.5.

#### Handling Events

In a procedural (top-down) program, the flow of execution is determined by the code and the data that it makes use of. Although the user can influence the flow through the input of data, it is the data that make the difference, not the fact that the user entered that data. However, in many programs nowadays, the actions of the user (such as clicking the mouse) or system-related circumstances (such as the state of a timer) determine the flow of execution. These actions are called **events** and the resulting program is said to be **event-driven**. Although event-driven programs need not be written for a GUI or be object-based, the prime examples of event-driven programs are object-oriented and employ a graphical user interface.

To illustrate the concept of events and how they are handled by a program, let us return to the simple window displayed in Figure 3 of Section 8.3. (For the sake of convenience, it is reproduced here in Figure 4.) We will use this window and the components it contains as the interface for an event-driven program.

As you know from Section 8.3, windows and the components contained within them are objects that have various attributes. Also associated
with each object is a set of events and methods, called event-handlers or event procedures, that enable the objects to carry out their functions. For example, the basic function of a command button is to allow the user, by clicking that button, to call for a specified action. Thus, the command button object needs a method to handle this event—to recognize the click and then call the appropriate subprogram. The programming language software ensures that this is done automatically. However, the programmer must design and write the code for this subprogram so that it performs the proper action when called.

Here is a list of some typical methods associated with the objects in a GUI. (As usual, the names we use here will probably differ from those in an actual programming language.)

ณา Methods for a window include:
- Open—opens (displays) the window
- StartUp—programmer-written procedure that executes automatically when the window is opened
- Close—closes the window (removes it from the screen)

ณา Command button methods include:
- Click—executes automatically when the button is clicked

ณา Text box methods include:
- Click—executes automatically when the box is clicked
- Change—executes automatically when the text within the box is changed

ณา Option button methods include:
- Click—executes automatically when the button is clicked

**A Simple GUI Program**

When a GUI program is run, the initial window appears on the screen and a StartUp procedure, if any, for this window is executed. From that point on, the flow of execution is determined by the events that occur. As an example of a simple GUI program and how it works, we will present the pseudocode used to create an “area calculator.”
EXAMPLE 7

To use the Area Calculator window shown in Figure 4, the user types a number (representing the side of a square) into the upper text box. When this is done, the Calculate button becomes active (it is no longer grayed out) so that it can respond to a mouse click. The user then clicks this button to display the area of the square in the lower text box. To calculate another area, the user clicks inside the upper text box, which clears the numbers in both text boxes, grays out the Calculate button, and allows the process to be repeated. When done calculating areas, the user clicks the Done button, which closes the window and ends the program. Here is the pseudocode that defines the methods and sets some of the attributes for the objects in this program (attributes for GUI objects are described in Section 8.3):

Window
   Name = MainWindow
   Title = "Area Calculator"

Upper Label
   Text = "Side of Square"

Lower Label
   Text = "Area of Square"

Upper Text Box
   Name = InputBox
   Text = ""
   Subprogram InputBox.Click()
      Set InputBox.Text = ""
      Set OutputBox.Text = ""
      Set CalculateButton.Enabled = False
   End Subprogram
   Subprogram InputBox.Change()
      Set CalculateButton.Enabled = True
   End Subprogram

Lower Text Box
   Name = OutputBox
   Text = ""

Left Command Button
   Name = CalculateButton
   Caption = "Calculate"
   Enabled = False
   Subprogram CalculateButton.Click()
      Set OutputBox.Text = Val(InputBox.Text) ^ 2
   End Subprogram
When the program is started, the window shown in Figure 4 appears on the screen. (All object properties, including those specified in the pseudocode above, are assigned when the program is compiled.) Since there is no StartUp procedure defined for this window, execution pauses until an event occurs. The program works as described at the beginning of this example; to make sense out of the pseudocode, imagine that you are running the program, and when you “perform an action,” read the corresponding subprogram. Also, note that:

- The Val function converts string data into numeric data; it is described in Section 7.3.
- The End Program statement in the last subprogram halts execution.

Event-driven Program Design

One of the striking differences between top-down programming and event-driven programming is that the latter has no main program, no “master controller.” Execution begins by displaying the program’s initial window and running its StartUp procedure, if any. But, from then on, execution moves from subprogram to subprogram depending on the events that take place. Execution terminates when a statement that ends the program is encountered.

The analysis phase of an event-driven program design is similar to that of an object-oriented program (see Section 8.2). Here are the basic steps involved:

1. Identify the windows needed in the program.
2. Determine the relationships among the windows; for example, which window can open another (so that the latter appears on the screen). Such relationships can be pictured in a flow diagram (see Figure 5). In a flow diagram, an arrow pointing from one window to another (as from Window 6 to Window 2 in Figure 5) means that the first can open or reactivate the second; a “double arrow,” pointing in both directions (as with Windows 2 and 3 in Figure 5) means that either window can open or reactivate the other.
3. For each window:
   - Determine the components (command buttons, text boxes, etc.) needed for that window.
Draw a rough sketch of the resulting window.

Determine the properties and methods needed for the window and each of its components. (The methods need not be “fleshed out” in this phase.)

The third step of this process leads us into the design phase of the program development. Here, the methods for each object (and perhaps additional properties) are defined. We will illustrate the design process in detail in Section 8.5.

Reading Check 8.4

1. Is each of the following statements true or false?
   a. When the user clicks a command button during program execution, that action is considered to be an event.
   b. Event-driven programs must make use of a graphical user interface.

2. Which method would you use to respond to data being typed into a Text box—its Click method or its Change method?

3. How would you modify the Area Calculator program of this section to
   a. Display the number 0 in both text boxes when the program starts?
b. Display the message “Invalid input” in the bottom text box if the user enters a negative number into the top text box and clicks the Calculate button? *(Hint: Use an If-Then-Else statement in the CalculateButton.Click subprogram.)*

4. Which of the following is the first step in developing an event-driven program?
   a. Write the main program code.
   b. Write the “click” subprograms.
   c. Determine which windows are needed.
   d. Determine which buttons are needed.

**8.5 Focus on Problem Solving**

In this section, we will develop an event-driven program for a problem we originally solved in Section 7.5 using a traditional top-down modular approach. Our new program uses the material covered in this chapter, but will be easier to understand if you first review Section 7.5.

**The Grade Management Program Revisited**

One of your instructors, Professor Allknowing, would like to have a program that creates an “electronic grade sheet” so that he can keep track of student scores in his classes. This program should allow the professor to enter the names of his students and their scores on three tests, compute the average test score for each student, and display this information any time he wants.

**Problem Analysis** We first want to determine the windows we will need to handle this problem. To do so, we imagine that we are teaching this class, and think about the kinds of things we will do with this program.

1. We must create a file (GRADES) to hold the grade sheet information, specifically the name, test scores, and test average for each student in the class. In the process of creating the file, we enter the names of students in the class.
2. Once the GRADES file has been created, we need to be able to retrieve the file to display its information, enter test scores, or compute test averages.

Thus, the initial window should allow the user to select from two options: create the grade sheet or retrieve the grade sheet.

Now that we have determined the contents of the initial window, it is clear that we need at least two additional windows:

☞ A Create Grade Sheet window
☞ A Retrieve Grade Sheet window
FIGURE 6
Flow Diagram for the Grade Management Program

Create Grade Sheet

Retrieve Grade Sheet

Display Grade Sheet

Select Test

Compute Test Averages

Enter Test Scores

The latter window would allow users to specify the particular task that they want to perform: display the grade sheet, enter test scores, or compute test averages. This leads to at least three more windows:

- A *Display Grade Sheet* window
- An *Enter Test Scores* window
- A *Compute Test Averages* window

Now, to see if additional windows are needed, we imagine that we were using the program to perform its stated functions. In doing so, we realize that when entering test scores, the first thing the program must know is whether those scores are for test 1, test 2, or test 3. For this reason, we create one more window, a *Select Test* window, to precede the actual entering of test scores.

Figure 6 displays a flow diagram that shows the windows described above and their relationship to one another. Recall that an arrow from one window to another means that execution can move from the first window to the second.

*Program Design*. The program design consists of two stages. For each window indicated in Figure 6, we must
1. Determine the objects (window components) needed to implement the user interface for that window.

2. Write pseudocode that describes the attributes (properties) and methods (subprograms) for that window.

(Due to the space required to present these descriptions, we will not supply detailed pseudocode for every aspect of this program.)

Initial window We will use the following window components for the initial window:
- A label to describe the options that follow.
- Two option buttons to allow selection of the display grade sheet or retrieve grade sheet option.
- A command button (OK) to put the selected option into effect.
- A command button (Quit) to allow the user to end the program.

A sketch of this window (that the program designer might draw) and the actual window are shown in Figures 7 and 8, respectively. Here is the pseudocode for this window’s option buttons and command buttons:

---

**FIGURE 7**
A Sketch of the Initial Window

Do you want to:

- Create the grade sheet?
- Retrieve the grade sheet?

OK    Quit

**FIGURE 8**
The Actual Initial Window

---

8.5 Focus on Problem Solving  

...
Top option button:
   Name = Option1
   Caption = "Create the grade sheet?"
   Value = False [no bullet]
   Subprogram Option1.Click()
      Set Option1.Value = True
      Set Option2.Value = False
   End Subprogram

Bottom option button:
   Name = Option2
   Caption = "Retrieve the grade sheet?"
   Value = True [bullet]
   Subprogram Option2.Click()
      Set Option1.Value = False
      Set Option2.Value = True
   End Subprogram

Left command button:
   Name = OKbutton
   Caption = "OK"
   Subprogram OKbutton.Click()
      Set InitialWindow.Visible = False
      If Option1.Value = True Then
         Call Create.Open
      Else
         Call Retrieve.Open
      End If
   End Subprogram

Right command button:
   Name = QuitButton
   Caption = "Quit"
   Subprogram QuitButton.Click()
      End Program
   End Subprogram

Create Grade Sheet window The window components are shown in Figure 9. The pseudocode needed to implement this window involves:
1. Opening a file (for Output) to hold the required information—name, test scores, and test average—for each student in the class. This file has records of the form:
   student name, test 1 score, test 2 score, test 3 score, average
2. Inputting the student names from the user.
3. Writing the names to the file and, at the same time, initializing all test scores and averages in the file to 0.
The pseudocode for opening the file is housed in this window’s Start-Up method. This method also initializes a counter, StudentCount, that will hold the number of students in the class for use by several other subprograms. If the current window has the Name attribute Create, then the pseudocode for StartUp is:

Subprogram Create.StartUp()
    Open “GRADES” For Output As GradeFile
    Set StudentCount = 0
End Subprogram

To input a student name, the user types it in the text box and clicks the Enter command button. Thus, all the code to input names and write the corresponding file records can be housed in the Enter command button’s Click method. (In this pseudocode, we assume that the Name attribute of the text box is InputBox.) This method also increments the counter StudentCount that was initialized in the window’s StartUp method.

Subprogram Enter.Click()
    Write GradeFile, InputBox.Text, 0, 0, 0, 0
    Set StudentCount = StudentCount + 1
    Set InputBox.Text = “”
End Subprogram

Finally, the file is closed when the user clicks the Done button, which also closes the Create Grade Sheet window and reactivates (makes visible) the initial window.

Subprogram Done.Click()
    Close GradeFile
    Call Create.Close
    Set InitialWindow.Visible = True
End Subprogram

**Retrieve Grade Sheet window**  This window and its methods have several functions:

1. When the window is opened, its StartUp method opens the file GRADES for Input and loads it into three parallel arrays—two one-dimensional arrays to hold the student names (Names) and test averages (Averages)...
and a two-dimensional array (Scores) to hold the scores on the three tests (see Section 7.5).

2. The window allows the user to choose one of three options—display the grade sheet, enter test scores, or compute test averages.

3. The window allows the user to elect to exit the program, and then copies the arrays back to the GRADES file (see Section 7.5), closes this file, and halts execution.

To implement these actions, we use

- A label object at the top displaying the text “Select an option and click OK.”
- Three option buttons with captions “Display the grade sheet,” “Enter test scores,” and “Compute test averages.”
- Two command buttons at the bottom: One labeled OK, which transfers execution to the proper window; the other labeled Exit Program, which closes the GRADES file and halts execution.

We leave the sketch of this window and the pseudocode for its components as an exercise (see Reading Check 8.5).

**Display Grade Sheet window** When opened, this window displays the contents of the grade sheet (the Names, Scores, and Averages arrays) in the window itself. The code to do this is located in the window’s Start-Up method. This window also contains a single command button, labeled OK, that closes this window and reactivates the Retrieve Grade Sheet window.

**Select Test window** The Select Test window (with Name = SelectTest) contains three option buttons, stacked vertically. By clicking the appropriate option button, the user specifies whether he or she wants to enter scores for test 1, test 2, or test 3 into the Scores array. This window also contains OK and Cancel command buttons. Clicking the OK button records the user’s selection, closes the window, and opens the Enter Test Scores window. Clicking the Cancel button closes the window and reactivates the Retrieve Grade Sheet window. The pseudocode for this window is:

```
Top option button:
    Name = Option1
    Caption = “Enter scores for test 1”
    Value = False
    Subprogram Option1.Click()
        Set Option1.Value = True
        Set Option2.Value = False
        Set Option3.Value = False
    End Subprogram
```
**Middle option button:**

Name = Option2  
Caption = “Enter scores for test 2”  
Value = False  
Subprogram Option2.Click()  
  Set Option1.Value = False  
  Set Option2.Value = True  
  Set Option3.Value = False  
End Subprogram

**Bottom option button:**

Name = Option3  
Caption = “Enter scores for test 3”  
Value = False  
Subprogram Option3.Click()  
  Set Option1.Value = False  
  Set Option2.Value = False  
  Set Option3.Value = True  
End Subprogram

**Left command button:**

Name = OKbutton  
Caption = “OK”  
Subprogram OKbutton.Click()  
  If Option1.Value = True Then  
    Set TestNum = 1  
  End If  
  If Option2.Value = True Then  
    Set TestNum = 2  
  End If  
  If Option3.Value = True Then  
    Set TestNum = 3  
  End If  
  Call EnterScores.Open  
  Call SelectTest.Close  
End Subprogram

**Right command button:**

Name = CancelButton  
Caption = “Cancel”  
Subprogram CancelButton.Click()  
  Set Retrieve.Visible = True  
  Call SelectTest.Close  
End Subprogram

**Enter Test Scores window** The Enter Test Scores window contains two text boxes and a command button. The student names are displayed, one by one, in the upper text box. After each name is displayed, the user types the corresponding score in the lower text box and clicks the Enter
When all student scores have been entered, this window closes automatically and the Retrieve Grade Sheet window is displayed. (The Enter Test Scores window is shown in the Programming Problems section at the end of this chapter. In that section, you are asked to create a program that inputs test scores.)

Compute Test Averages window  When the “Compute test averages” option in the Retrieve Grade Sheet window is activated (by clicking the OK button in that window after this option button has been selected), the average score for each student is calculated and written to the Averages array. This pseudocode (which you are asked to provide in Reading Check 8.5) is located in the Compute Test Averages StartUp method. The window itself is very simple—it consists of the message (label) “Average test scores have been computed!” and a single command button labeled OK. When the latter is clicked, the window closes and the Retrieve Grade Sheet window is reactivated.

Program Code  The program code is now written using the design as a guide. Even though this is not a procedural (top-down) program, it still needs documentation (see Section 2.3). Header comments are used to help explain the general purpose of the program, each window, and the more complicated subprograms; step comments are used to explain the purpose of certain variables, properties, and statements. Here are a couple of other points concerning the coding of this program:

- Keep in mind that some languages, although providing built-in class definitions for GUI objects, require you to write code that creates these objects and adds them to the appropriate window. You may also have to, in code, specify an object’s position, size, and many other attributes not mentioned in the pseudocode in this book.
- To display the contents of the GRADES file in a neat, readable fashion, the output produced in the Display Grade Sheet window must be formatted—the names, test scores, and averages should line up in columns. This can be accomplished using the programming language’s special print formatting statements.

Program Test  This program is tested in a way similar to that of its top-down counterpart in Section 7.5: Imagine that you are the instructor of a very small class (say, three students), and use the program to create the grade sheet, enter scores for each of the three tests, and compute the test averages. After performing each of these tasks, display the grade sheet to check that the operation has been executed successively. Moreover, as you perform the testing, examine each window and ask yourself if it could be improved in any way: Could it look better? Could it be made more intuitive? Could it be simplified?
Reading Check 8.5

These exercises refer to the Grade Management Program developed in this section.

1. Draw a sketch of the Retrieve Grade Sheet window.

2. Suppose that the (internal) name for the Retrieve Grade Sheet window is Retrieve, that the names of its option buttons are Option1, Option2, and Option3, and that the name of its “OK” command button is OKbutton.
   a. Write a Click procedure for Option1 that causes it to acquire the bullet (and causes the other two option buttons to be bulletless).
   b. Write a Click procedure for OKbutton that “hides” the Retrieve Grade Sheet window and opens the window named Display, SelectTest, or ComputeAverages, depending on the state of the option buttons.

3. The StartUp procedure for the ComputeAverages window uses the data in the Scores array to calculate the average test score for each student, and writes this data to the Averages array. Write a subprogram called ComputeAverages.StartUp that does this. Recall that
   ◆ The variable StudentCount holds the number of students in the class.
   ◆ The array Scores has entries of the form Scores[testnum, studentnum]; for example, Scores[2, 15] holds the score of student 15 on test 2.

Chapter Review and Exercises

Key Terms

Object
Object-oriented programming
OOP
Class (of objects)
Attributes (of an object)
Methods (for a class)
Public member
Private member
Message (to a method)
Constructor
Procedural programming
Encapsulation
Inheritance
Polymorphism
Base class
Derived class
Graphical user interface
GUI
Window
Window component
Command button
Check box
Text box
Option button
Drop-down list box
List box
Label
Event
Event-driven program
Chapter Summary

In this chapter, we have discussed the following topics:

1. Classes and objects:
   - A class is a data type that defines a structure consisting of attributes (data) and methods (procedures). We use a Class … End Class statement to define a class.
   - An object is an instance of a class. We use a Declare statement to create an object.

2. Characteristics of an object-oriented programming language:
   - Encapsulation—the combining of data and operations on that data into a single package.
   - Inheritance—the ability to create a subclass (derived class) that automatically contains the attributes and methods of its parent (base) class.

STUDY SKILLS

Taking Tests

The most effective test-taking strategy is to conscientiously prepare for the test (see the Study Skills feature in Chapters 6 and 7). Nevertheless, there are many things you can do while taking a test that can help to improve your grade.

- Get to the test on time! You may need to use every minute of the allotted time to complete the test.
- If you are not allowed to make use of the textbook or your notes during the test, and you’re not confident that you’ll remember some of the facts that you’ve memorized, as soon as you receive your test paper, quickly write this information on the back or in the margins.
- Unless the test is a very short one, quickly scan it to see the kinds of problems it contains.
- If the test consists of different types of problems, such as short answer questions and programming problems, it is usually a good idea to get the short answers out of the way first. As a general rule, do the easy (or quick) problems first to build up your confidence, then tackle the harder ones.
- For each problem, read the instructions carefully. Many errors occur (especially in solving programming problems), simply because the student does not answer the given question or solve the stated problem.
- If you don’t know the answer to a question or how to solve a problem, place a mark next to it and move on to another one. Come back to these “skipped” problems at the end of the test.
- If the test contains multiple-choice or true/false questions (and there is no “penalty for guessing”), don’t leave any of these answers blank. If you’re not sure of an answer, take your best guess, place a mark next to the question, and come back to it if you have time at the end of the test.
- When you complete the exam, you will probably want to hand it in immediately. However, if there is still time left, resist this temptation. A much better idea is to use that time to check your work. On a programming exam, it’s very common to make careless mistakes, and by going over your work before handing it in, you may catch most of these errors.
- Polymorphism—the ability of a method to work with objects of different classes.

3. Object-oriented program design; in analyzing the problem:
   - Identify the classes to be used in the program.
   - Determine the attributes needed for these classes.
   - Determine the methods needed for these classes.
   - Determine the relationships among the classes.

4. Objects in a graphical user interface include:
   - Windows—to act as containers for other objects
   - Command buttons—to initiate actions
   - Check boxes—to allow the user to select or deselect a single option
   - Text boxes—to receive input or display output
   - Option buttons—to allow the user to select from among several options
   - List boxes—to allow the user to select an item from a list

5. A GUI object has:
   - Properties (attributes) that include its name, its caption or text, its value, whether or not it is enabled (active or inactive), the font of its text, and so on.
   - Methods that include responding to a click and other events, depending on the particular object.

6. Event-driven programs:
   - Respond to user or system events (actions).
   - Are designed in a manner similar to an OOP program, but put the emphasis on the windows, rather than on the general objects, needed to solve the given problem.

-review exercises

1. Fill in the blank: A(n) ____________ is a data type comprised of attributes and methods.
2. Fill in the blank: A(n) ____________ is an instance of a class.
3. True or false: The attributes of an object include the name of the class to which it belongs.
4. True or false: The methods of an object are the operations that can be performed on that object’s data.
5. True or false: Attributes are also known as subprograms, procedures, or functions.
6. True or false: Methods are also known as services or behaviors.

Exercises 7–10 refer to the following pseudocode, which defines a class called Square.
Class Square
Side As Real
Area As Real
Subprogram SetSide(NewSide)
  Set Side = NewSide
End Subprogram
Subprogram ComputeArea()
  Set Area = Side ^ 2
End Subprogram
Function GetArea() As Real
  Set GetArea = Area
End Function
Function GetSide() As Real
  Set GetSide = Side
End Function
End Class

7. List the attributes of the class Square.
8. Give the names of the methods of the class Square.
9. Fill in the blank: To specify that the variable Side may not be referenced from outside this class, place the keyword ____________ in front of it (in the first line of the class definition).
10. Fill in the blank: To specify that Subprogram SetSide may be called from outside this class, place the keyword ____________ in front of it (in the third line of the class definition).

In Exercises 11–14, write a (main program) statement that uses the class Square (defined above Exercise 7) to perform the indicated operation.

11. Create an object Square1 in the class Square.
12. Set the length of the side of Square1 equal to 20.
13. Compute the area of Square1.
14. Display the area of Square1.
15. Fill in the blank: The property of OOP that refers to the incorporation of data and operations into a single unit is called ____________.
16. Fill in the blank: The property of OOP that allows the creation of new classes based on existing ones is called ____________.
17. Fill in the blank: The property of OOP that allows a method to be applied to objects of different classes is called ____________.
18. Fill in the blank: When a subclass is created based on an existing class, the original class is called the ____________; the new class is called the ____________.

Exercises 19–22 refer to the following pseudocode, which uses the class Square defined above Exercise 7.
Class Rectangle As Square
  Height As Real
  Subprogram SetHeight(NewHeight)
    Set Height = NewHeight
  End Subprogram
  Function GetHeight() As Real
    Set GetHeight = Height
  End Function
  Subprogram ComputeArea()
    Set Area = Side * Height
  End Subprogram
End Class

19. List all attributes of an object in the class Rectangle.
20. List the names of all methods for an object of the class Rectangle.
21. Suppose Square1 is an object in the class Square and the statement
    Call Square1.ComputeArea
    appears in the main program. Which formula for area is used in the
    execution of the subprogram ComputeArea?
    a. Area = Side ^ 2
    b. Area = Side * Height
    c. Neither of these formulas is used.
22. Suppose Rectangle1 is an object in the class Rectangle and the statement
    Call Rectangle1.ComputeArea
    appears in the main program. Which formula for area is used in the
    execution of the subprogram ComputeArea?
    a. Area = Side ^ 2
    b. Area = Side * Height
    c. Neither of these formulas is used.
23. True or false: When contrasted to OOP, the approach to programming that uses top-down,
    modular program design is called procedural programming.
24. True or false: In designing an object-oriented program, the emphasis is placed on the procedures,
    rather than on the objects, needed to solve a given problem.
25. For the window shown in Figure 10, give the captions/labels for its
    a. Command buttons
    b. Option buttons
26. For the window shown in Figure 10, give the text appearing in its
    a. Text boxes
    b. Label objects
27. In the window shown in Figure 10, is the check box selected or clear?
28. In the window shown in Figure 10, which option button is selected?

29. Which of the following GUI objects is most commonly used to allow the user to initiate an action?
   a. Check box  
   b. Command button  
   c. Option button  
   d. Text box

30. Which of the following GUI objects is most commonly used to receive input from the user?
   a. Check box  
   b. Command button  
   c. Option button  
   d. Text box

31. Which of the following GUI objects is most commonly used to allow the user to select or deselect a single option?
   a. Check box  
   b. Command button  
   c. Option button  
   d. Text box

32. Which of the following GUI objects is most commonly used to allow the user to make one choice from a group of choices?
   a. Check box  
   b. Command button  
   c. Option button  
   d. Text box

33. True or false: The attributes of objects that make up a GUI are also called properties.

34. True or false: All window components have the same attributes.

35. List three properties of a window.

36. List three properties common to command buttons, option buttons, and text boxes.

37. True or false: From a programming point of view, clicking a command button is considered an event.

38. True or false: An event-driven program must make use of the mouse.

39. True or false: Command buttons, option buttons, and text boxes all have Click events associated with them.

40. True or false: The programming language supplies the code for the action that takes place when the user clicks a command button.
41. Write a subprogram called Abutton.Click that sets the text in the
text box named Text1 equal to “Hello” and grays out the command
button named Bbutton.

42. Write a subprogram called OKbutton.Click that opens a window
named Window1 or a window named Window2, depending on
whether option buttons named Option1 or Option2, respectively,
contain a bullet.

43. Write a subprogram called Option1.Click that places a bullet in an
option button named Option1 and removes the bullet from an
option button named Option2.

44. Write a subprogram called EnterButton.Click that adds the number
in a text box named NumberBox to a variable Sum and then clears
the text box.

Programming Problems
For Problems 1 and 2, use pseudocode to write an object-oriented program
consisting of a class definition and a main program (like those in Sec-
tions 8.1 and 8.2) to solve the given problem.

1. Write a program that inputs (from the user) the number of hours
worked and hourly pay rate for employees and outputs their total
pay. The program should process an arbitrary number of employ-
ees; the user will terminate input by entering 0 for both hours
worked and rate of pay. Use a class called Worker with
◆ Attributes: Hours, Rate, Total
◆ Methods: ComputeTotal and access methods for each of the
attributes

2. Write a program that displays the income tax due (in a certain
state) on a taxable income entered by the user, according to the
following table:

<table>
<thead>
<tr>
<th>TAXABLE INCOME</th>
<th>TAX DUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
</tr>
<tr>
<td>$0</td>
<td>$50,000</td>
</tr>
<tr>
<td>$50,000 $100,000</td>
<td>$2,500 + 7% of amount over $50,000</td>
</tr>
<tr>
<td>$100,000 . . . . . .</td>
<td>$6,000 + 9% of amount over $100,000</td>
</tr>
</tbody>
</table>

The program should allow the user to enter a series of incomes,
terminated by 0. Use a class called Tax, with attributes Income and
TaxDue and methods that include ComputeTax.

In Problems 3–6, use pseudocode to write an event-driven program (like
those in Sections 8.3–8.5) according to the given specifications. If a pic-
ture of a window is not shown, sketch it before writing the pseudocode.
3. Write a program for an interface that consists of a single window entitled Greeting, containing an unlabeled text box and two command buttons (Respond and Done). When the program starts, the text box contains the word Hello. If the user clicks the Respond button, the word changes to Goodbye. When the user clicks the Done button, the program terminates.

4. Using the window shown in Figure 11, write a temperature conversion program. To use this program, the user selects an option, types a temperature into the appropriate text box, and clicks the Convert command button. The corresponding temperature then appears in the other text box. Clicking on the Done button ends the program. Hint: Use the formulas:

\[
C = \frac{5}{9} (F - 32) \\
F = \frac{9}{5} C + 32
\]

5. Write a program that displays the names of the students in a certain class, one by one, and allows the user to enter a test score for that student. The student names and scores are contained in one-dimensional arrays called Names and Scores, respectively. Assume that both arrays have already been declared and that the Names array contains N entries.

The window for this program is shown in Figure 12. When the program starts, the first student name is displayed in the upper text box. The user types the test score for that student in the lower text box.
box and clicks the Enter button. This action assigns that number to the first element of the array Scores, displays the next name in the upper text box, and empties the lower text box so that the next score can be entered. When all student scores have been entered (and assigned to the corresponding elements of Scores), the window closes automatically and the program terminates.

6. Write a program that searches a sequential file called CUSTOMER, with records of the form

   name (string), phone number (string)

for a specified name and then displays the corresponding phone number. The interface consists of a window entitled “Customer Phone Numbers” that contains two text boxes (an upper one labeled “Customer name:” and a lower one labeled “Phone number:”) and three command buttons (labeled Find, Clear, and Done).

   The CUSTOMER file is opened in the window’s StartUp method. To use this program, the user types the customer name into the upper text box, clicks on the Find command button, and

   ◆ If the name is found in the file, the corresponding phone number appears in the lower text box.

   ◆ If the name is not found, the lower text box displays “Name not found.”

   Clicking the Clear button removes the text in the upper and lower text boxes. Clicking the Done button closes the file and terminates the program.