APPENDIX E

A Brief Introduction to Object-Oriented Programming

What exactly is object-oriented programming? And what is meant by an object-oriented system? To answer this, let's first look at the programming paradigm that preceded it.

Procedural Programming

Until the mid 1990s most computer programs developed were *procedural programs*. These are programs designed as a set of procedures, or functions, for carrying out the various tasks that together provide the functionality of the program. Each procedure is made up of programming language statements that are executed by the computer, one after the other, to handle one of the tasks the program must carry out. The statements might gather input from the user, manipulate data stored in the computer's memory, perform calculations, or do any other operation necessary to complete a task. For example, suppose we want the computer to calculate someone's gross pay. Here is a list of things the computer should do:

1. Display a message on the screen asking "How many hours did you work?"
2. Accept a number input by the user and store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Accept a second number input by the user and store it in memory.
5. Once both numbers are entered, multiply them and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in step 5.

If the algorithm's six steps are performed in order, one after the other, it will succeed in calculating and displaying the user's gross pay.

Procedural programming was the standard back when users interacted with text-based computer terminals. For example, Figure E-1 illustrates the screen of an older MS-DOS computer running a program that performs the pay-calculating algorithm. The numbers the user entered are shown in bold.

Figure E-1

```
How many hours did you work? 10
How much are you paid per hour? 15
You have earned $150.00
C>_
```

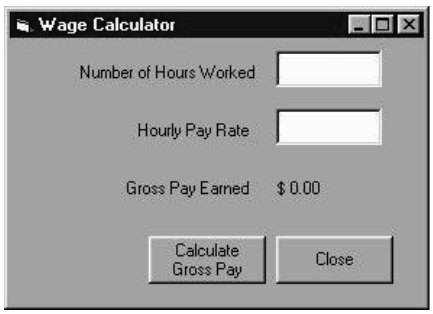
In text-based environments using procedural programs, the user responds to prompts from the program. Modern operating systems, however, such as Windows, use a graphical user interface, or GUI (pronounced "gooey"). Although GUIs have made programs friendlier and easier to use, they have not simplified the task of programming. In fact, in some ways they have made more work for the programmer. GUIs make it necessary for the programmer to create a variety of on-screen elements such as windows, dialog boxes, buttons, menus, and other items that provide an interface through which the user can interact with the program. Furthermore, the programmer must write statements that handle the user's interactions with these on-screen elements, in any order they might occur. Instead of the user responding to the program, the program responds to the user. The need to manage these types of things has helped influence the shift from procedural programming to object-oriented programming.

Object-Oriented Programming

Whereas procedural programming is centered on creating procedures, object-oriented programming is centered on creating objects. An *object* is a programming entity that normally models some real-world entity, such as a student, a bank account, or even a computer screen. The object *knows* certain things about itself and can *do* certain things. The things it knows are called its *attributes* and are "remembered" by storing them as data. The things it can do are called its *methods* and consist of the actions, or behaviors, it can carry out with its functions. The object is, conceptually, a self-contained unit consisting of data (attributes) and functions (methods).

Object-oriented programming (OOP) has revolutionized GUI software development. For instance, in a GUI environment, the pay-calculating program might appear as the window shown in Figure E-2.

Figure E-2



This window can be thought of as an object. It contains other objects as well, such as text input boxes, and command buttons. Each object has attributes that determine its appearance. For example, look at the command buttons. One has the caption "Calculate Gross Pay" and the other reads "Close". These captions, as well as the buttons' sizes and positions, are attributes of the command button objects. Objects can also hold data entered by the user. For example, one of the text input boxes allows the user to enter the number of hours worked. When this data is entered, it is stored as an attribute of the text input box.

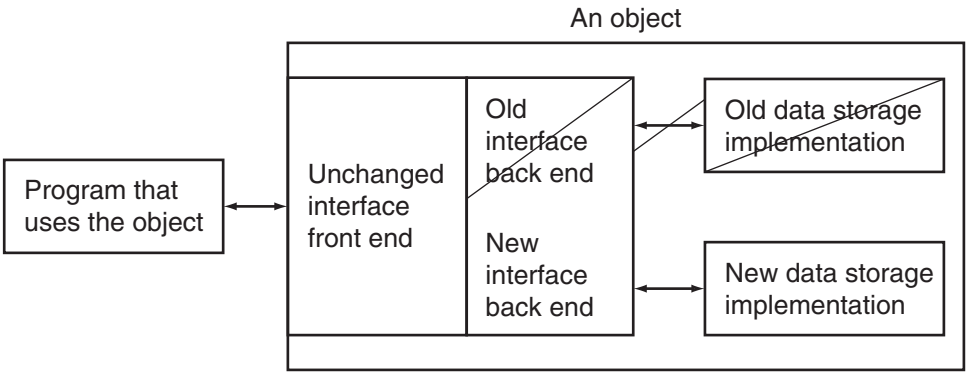
The objects also have actions, or methods. For example, when the user clicks the "Calculate Gross Pay" button with the mouse, the program will display the amount of gross pay. A method, or function, associated with the button object labeled "Calculate Gross Pay" performs this action.

The Benefits of Object-Oriented Programming

The complexity of GUI software development was not the first difficult challenge that procedural programmers faced. Long before Windows and other GUIs, programmers were wrestling with the problems of code/data separation. In procedural programming, there is a distinct separation between data and program code. Data is kept in variables of specific data types, as well as programmer-defined data structures. The program code passes the data to modules designed to receive and manipulate it. But, what happens when a program's specifications change, resulting in redesigned data structures, changed data types, and new variables being added to the program? In a procedural program when the structure of the data changes, the modules that operate on it must also be changed to accept the new format. This results in added work for programmers and creates an opportunity for bugs to appear in the code.

Object-oriented programming (OOP) addresses the problem of code/data separation through encapsulation and data hiding. *Encapsulation* refers to bundling together data and the procedures that work with it into a single object. *Data hiding* refers to an object's ability to hide data storage details from programs that use the object. Code outside the object can only access the data by calling the object's methods. These methods, or procedures, provide an interface through which external programs access the data stored in the object. The programs do not need to know anything about how the data is stored. They only need to know how to interact with the object's methods. If a programmer needs to change the type or structure of an object's internal data, the procedures that provide the interface between the object's data and the external programs using it are changed at the same time. But nothing changes from the external program's point of view; it accesses the data as it did before. This is illustrated in Figure E-3.

Figure E-3



Component Reusability

Another trend in software development that has encouraged the use of OOP is *component reusability*. A component is a software object that performs a specific, well-defined operation or that provides a particular service. The component is not a stand-alone program, but rather an object that can be used by programs that need the component's service. For example, Sharon is a programmer who has developed a component for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her component is coded to perform all the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. To save time and work, he can use Sharon's component to perform the 3D rendering.

An Everyday Example of an Object

Think of a simple digital alarm clock as an object. It has the following attributes:

- *hour* (a value in the range of 1–12)
- *minute* (a value in the range of 0–59)
- *second* (a value in the range of 0–59)
- *day/night indicator* (a.m. or p.m.)
- *alarm set time* (a valid hour, minute, and day/night indicator)
- *alarm status* (off or on)

As you can see, the attributes are merely data values that define the alarm clock's *state*. Table E-1 lists the alarm clock's methods. These are the actions the clock performs.

Table E-1

Method	When Performed	Action
Increment second	Every second	Adds 1 to value of the *second* attribute. If value was 59, value becomes 0.
Increment minute	*second* changes from 59 to 0	Adds 1 to value of the *minute* attribute. If value was 59, value becomes 0. Activates "Check alarm time" method.
Increment hour	*minute* changes from 59 to 0	Adds 1 to value of the *hour* attribute. If value was 12, value becomes 1.
Change am/pm indicator	*hour* changes from 11 to 12	Changes indicator status.
Set current time	User presses **set time button**	Changes *hour*, *minute*, and *am/pm* to values set by the user.
Set alarm time	User presses **set alarm button**	Changes *alarm set time* to values set by the user.
Check alarm time	Activated by "Increment minute" method	Checks if current time = *alarm set time* and *alarm status* is on. If so, activates "Sound alarm" method.
Turn alarm on	User moves **alarm enable switch** to on position	Sets *alarm status* to on.
Sound alarm	Activated by "Check alarm time" method	Sounds the alarm until "Turn alarm off" method is activated.
Turn alarm off	User moves **alarm enable switch** to off position	Sets *alarm status* to off.

The methods described in Table E-1 are part of the alarm clock object's private, internal workings. External entities (such as yourself) do not have direct access to the alarm clock's attributes, but these methods do. The object is designed to execute these methods automatically and hide the details from you, the user. These methods, along with the object's attributes, are part of the alarm clock's *private persona*.

Some of the alarm clock's methods are publicly available to you, however. In particular you can cause the "Set current time", Set alarm time", "Turn alarm on" and "Turn alarm off" methods to execute. You do this by pressing various buttons and moving various switches that have been provided as part of the public *interface* to allow external entities to interact with the object.

Classes and Objects

A class is a type, or category, of object. It specifies the attributes and methods that objects of that class possess. A class is not an object, however. It simply describes what objects of the class will look like when they are created. The objects themselves are instances of the class, and once a class has been defined, multiple instances (objects) can be created from it. Let's use our alarm clock example to explore this idea further.

Before an alarm clock can be produced it must be designed. What will it look like? What kinds of internal components will it have and how will they be controlled? What will the interface consist of? After the design is compete, the actual machinery must be put in place to assemble it. Once this has been done, multiple clocks can be produced with the same equipment based on the same design.

Now let's look at a software example. Jessica is a computer programmer who has a butterfly garden and studies butterflies as a hobby. She decides to create a program to help her catalog information on butterflies.

Before writing the actual program, however, she designs a `Butterfly` class, which specifies the attributes (variables) and methods that will be useful to hold and manipulate data common to all butterflies. After she creates the class, she writes the program. The program creates and uses many different `Butterfly` objects. Each one has a different name, such as `monarch`, `hollyBlue`, `swallowTail`, etc. However, they are all instances of a `Butterfly`.

Object-Oriented Systems

Component reusability and object-oriented programming technology set the stage for large-scale computer applications to become entire systems of unique collaborating entities (components). As you continue through this book you will become more familiar with writing object-oriented programs. You will also learn how to effectively use existing components, how to create new components, and how to make components work together to create entire object-oriented systems.