Chapter 3
Object-Oriented Programming Concepts

In This Chapter

Component
Refers to both the compile-time class definition and the runtime object of that class. May also refer to a class library.

In the first chapter of this guide, it was stated that object-oriented programming (OOP) "naturally lends itself to GUI environments by giving you the capability to develop complex systems through standard, reusable *components*, in a manner that models the real world." In this chapter, we'll explore some of the basic OOP concepts and, in doing so, explain exactly what is meant by this statement.

Why Object-Orientation?

What is the motivation for learning OOP? Furthermore, what possible motivation is there to rewrite existing programs in an object-oriented fashion?

The reasons are simple. First, it's a logical adaptation of the way we already view the world. Our perception of the world is a collection of objects that interact with each other; therefore, it is natural for us to think of software development in the same way.

Secondly, it's a smart business decision—objects are intrinsically modular and therefore encourage both reusability and safe, incremental enhancements.

Finally, object-orientation fits the event-driven nature of GUI programming rather well. We'll examine all these reasons in greater detail in this chapter, but first, let's explore this last point.

The Paradigm Shift

Paradigm
Model of behavior.

People often speak of a "paradigm shift" when programming for GUI environments. This is because Windows applications behave differently from traditional DOS applications.

In applications developed before GUIs became commonplace, the program dominated the conversation with the user. The program asked the user for input and displayed output in return. When filling in an insurance claim form, for example, the program led the user through each item that it required.

In well-designed GUI applications, however, the user is in control. Not only can users flip between applications at will, but they can also choose what to do next in an application (for example, which fields in the form to fill in and which to leave to the imagination of the application). Windows applications, therefore, are said to be *event-driven*, because the events generated by the user dictate what happens in the application. This is contrary to DOS applications, in which the program has control. (For more information about moving from DOS to Windows, see the *Programmer's Guide, Volume I*.)

Event-Driven Programming

Not surprisingly, the new paradigm has profound effects on how you program. Flexible control for the user means changing the way you program.

It is this shift to "the user's in control" that caused developers to select object-orientation as *the* way to tackle GUI development.

This is because event-driven behavior requires very modular programming: the program must execute in tiny atomic units that can start up at any time, do their task quickly, and finish. As we previously stated and as you will soon see, objects are intrinsically modular. Object-oriented programming, therefore, naturally lends itself to GUI environments; traditional procedural programming techniques do not.

Note: This does not mean, however, that you need to throw out all your code. Procedural programming does have its place in OOP; however, instead of directing the application, it is used in smaller units to handle the different actions that the user can perform.

Thinking in an Object-Oriented Way

Thinking in object-oriented terms also involves another shift: changing the way you view software development. While object-orientation helps meet the challenges of programming for event-driven GUI environments, it doesn't mix well with conventional programming training.

Traditionally, programmers are taught to tackle a programming problem by breaking it down into the operations that need to be performed. We were taught to think about the steps involved in solving a problem. We aligned ourselves with the *processes* involved.

Solving a problem using an object-oriented approach, on the other hand, means thinking about the *things* in the system. By breaking down large, complex things into smaller, simpler components, we reach the same goal—software that solves a problem. For example, rather than seeing an inventory system in terms of reducing stock, repricing, or posting to the general ledger, we would look at it in terms of its elements—such as parts, pick orders, and warehouse bins—and their individual properties and associated actions.

It's easy to overcomplicate the difference between object-oriented programming and traditional process-oriented programming. However, in reality, it's just a matter of perspective. The plan and overall objective are the same in both disciplines—we're still creating a solution to a problem by breaking that problem into several simpler ones. The only difference is in the approach we take to solving the problem.

The biggest hurdle, then, is actually learning to think about programming differently. But once you've mastered it, you're likely to find OOP a more intuitive way of doing things, because it more closely resembles your natural thought processes—you already think in an object-oriented way. (And OOP introduces a world of advantages you haven't even seen yet!)

What Is an Object?

You are an object. So is the chair in which you're sitting. And this manual you're reading. In fact, as you look around, you'll notice plenty of objects. Your computer, your office, your coffee mug, your chair. You already think in an object-oriented way.

And what do you notice about objects? In most cases, you'll probably notice two things:

1. The object has certain *properties* that help you figure out what it is and classify it.

   For example, a typical office chair has wheels, a seat, and a back.

2. The object has *actions* associated with it.

   For example, the chair described above can roll across the floor.

In the object-oriented world of software development, this is also true. An object—like a window or a check box—has properties (like a border or a caption) and actions (like displaying itself on the screen, in the case of a window, or toggling the check mark indicator on and off, in the case of a check box). However, instead of actions, they are referred to as *methods* (more on this later).

What Is a Class?

To begin to think of programming in terms of the things in the system (rather than the processes), it is imperative that you understand what it means to classify something.

When we classify a thing, we create an *abstraction* that describes it. For example, the office chair that we described earlier is a thing that has wheels, a seat, and a back, and can roll across the floor. This is, of course, an abstract definition. We don't know what color the chair is, what material it is made of, and so on, but it gives us some guidelines in determining what is a chair and what isn't.

Class = Blueprint

In OOP, a *class* is an abstract definition of something. Classes are useful because they help us categorize and group things. They also make it easier to create *new* things. It's easy to create a thing if you know—according to its definition—exactly what is needed to create it. In many respects, then, a class is like a blueprint.

Consider a house. If you want to build a house, you don't go to the lumberyard, buy a truckload of timber and nails and paint, and start building. Instead, you draw a blueprint of the house: without the blueprint, you wouldn't know how much wood to buy, what kind of nails you need, etc. It would be impossible to construct an entire house without a plan.

The blueprint for a house gives the builder all the necessary information. It fully describes every part of the house—the placement of the windows and doors, the number and kinds of fixtures, the materials used in flooring, the pitch of the roof, etc. It is important for the blueprint to be complete: for example, if no doors are shown in the blueprint, the resulting house won't have any either.

The blueprint is essentially the abstract definition for a house. In OOP terminology, then, the "House" class would be the abstract definition of a house thing (or object).

Instantiate
Create an object from a class.

The blueprint, however, is not a house—you can't live in the blueprint. If you wish to occupy the house described in your abstract definition, you will need to create, or *instantiate*, an actual house. When you instantiate a class, you get an object of that class—for example, instantiating the House class results in a House object.

Class versus Object

Instantiation is important because a class is not very useful by itself. It can't do anything; it merely specifies the characteristics that an object of a particular class would possess and how the object would behave if it existed.

The difference between objects and classes is critical: objects exist in space and time, whereas a class is an abstract definition, a plan you use to construct those objects. (In software terminology, a class exists at compile time, whereas an object does not exist until runtime.)

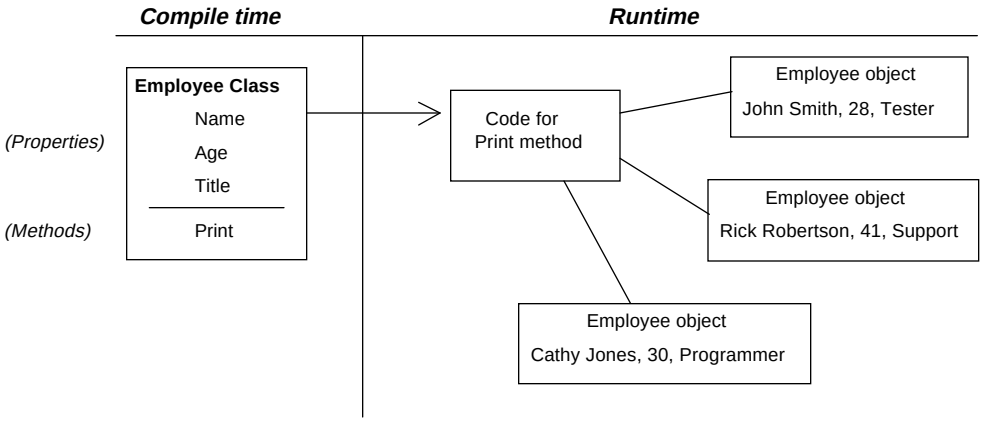
Applying Object-Oriented Thinking to Software

Let's apply what we've just learned to software. Suppose, for example, you are designing an information system for a small business to track employees, customers, inventory, sales transactions, financial records, etc.

In this system, there are certain things that all employees have in common—for example, they all have a name, age, and title. In addition, the system should be able to print out these personal details for each employee. By setting up these basic requirements, we've just described the Employee class.

With this class, you can create a whole set of Employee objects, one for each person in the company. Each Employee has its own data (for example, John Smith, Cathy Jones, Rick Robertson) and the ability to print.

From the one class definition, you can create multiple objects. This relationship is summarized in the following diagram:



Note: Since many objects of the same class can exist simultaneously, you can assign them to variables to uniquely identify one object from another. This is how you distinguish, for example, which Employee object you want to print or which one's name you want to know.

Methods

As you can see in the previous diagram, a class definition sets up two things for its objects: the properties it can have and the actions it can perform.

The code portion of an object—the actions that it can perform—is defined by methods of its class. In the above example, the Employee class has just one method, named Print.

Methods define what a class of objects is capable of doing. They are a lot like functions (they have parameters, declarations, programming statements, and return values), but they're different in that they are defined for a specific class and invoked for a specific object of that class.

Properties

The code inside the class (that of its methods) can see the data of the object that it is acting upon. Code outside the class (often called *external* code) usually sees only methods. This ability of the class to hide its data (or *instance variables*) from external code is called *encapsulation*.

So, how does external code get to the data? There are two ways: through *exported instance variables* or through special methods, called *access* and *assign* methods (also called *virtual variables*). The term *property* refers to either an exported instance variable or a virtual variable (in other words, any data visible to code that is external to the class is a property of that class).

Exported instance variables are sometimes frowned upon because they violate the encapsulation principle by making the object's data directly available to external code.

Virtual variables, on the other hand, allow data to be passed back and forth between an object and external code without violating the encapsulation rule. Access methods deliver data from the inside of an object to the outside, and assign methods deliver data from the outside of an object to the inside.

State

When an object is created at runtime, you can assign values to its properties and thereby change its *state*. Thus, all objects of the same class have the same properties, but the state of one object may be different from that of another. For example, all employees have a name, but one may be "Cathy Jones," while another is "Rick Robertson."

Similarly, all objects of the same class share exactly the same behavior via the methods defined in the class at compile time. The methods, however, do not change from one object to another (for example, the code for printing all objects in a class is identical, regardless of the object's state).

(The principle of encapsulation is discussed further later in this chapter in the Additional Strengths of OOP section.)

Inheritance: Superclasses and Subclasses

Not only are classes useful for creating many instances of the same type of object, but they are also helpful in setting up a *hierarchy* of related classes. In this hierarchy, there is an *inheritance* relationship among the various levels—each new level in the hierarchy "inherits from" the previous, higher level.

Returning to the House Example

To understand this concept in an abstract sense, let's return to the house example for a moment. Imagine the architect who will design the blueprints for all the houses in a development. The houses, aside from small differences, are basically identical. Should the architect draw a set of blueprints for each house from scratch? No, that would be reinventing the wheel unnecessarily—all that is really needed is a single, generic blueprint which contains *only the details that will be the same in all houses*. From the one "master" blueprint, the architect can then design new blueprints to add the details that are unique for each house.

For example, some of the houses are to have a two-car garage, others a one-car garage. The architect, then, would design three blueprints: one master blueprint and two secondary blueprints, both of which inherit from the master but add their own unique details (one for a house with a one-car garage and another for a house with a two-car garage).

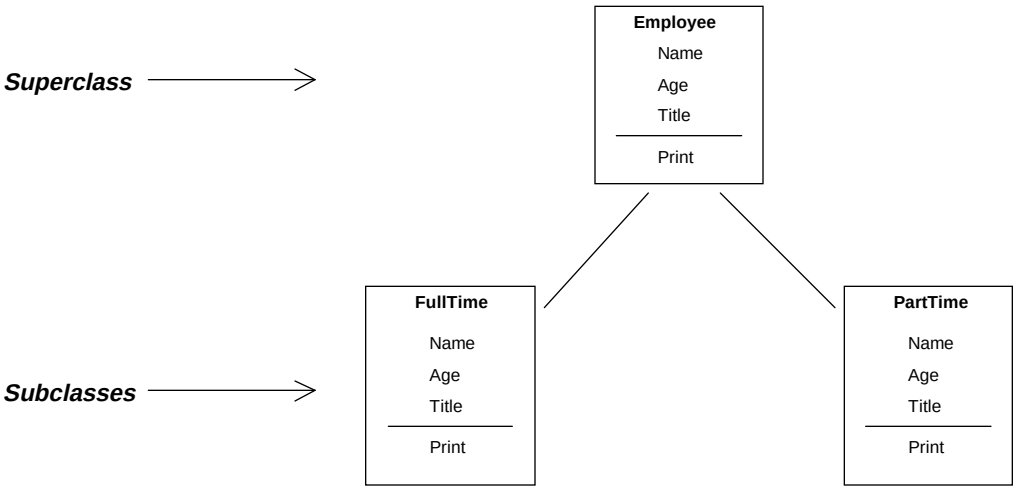
Think of the master blueprint as a starting point, and imagine that it is drawn on a transparency. When new features need to be added to the base design, the architect simply places another transparency on top of the original and draws the additions on it.

When the two transparencies are separated, the first will have only the generic design. This is the *superclass*, or parent. When the two transparencies are together, a house with embellishments is the one designed. This is the *subclass*, or child. It inherits all the characteristics of the parent but defines a more specific kind of house by adding characteristics of its own.

Subclassing the Employee Class

Moving back to our Employee class, suppose you decided to add a property to store the person's salary. If the person is full-time, they are salaried and entitled to benefits; if they are part-time, they are paid hourly, overtime may need to be calculated, and benefits are not granted. Therefore, it's not really just a simple matter of adding a Salary property to the Employee class—there are far too many other issues involved.

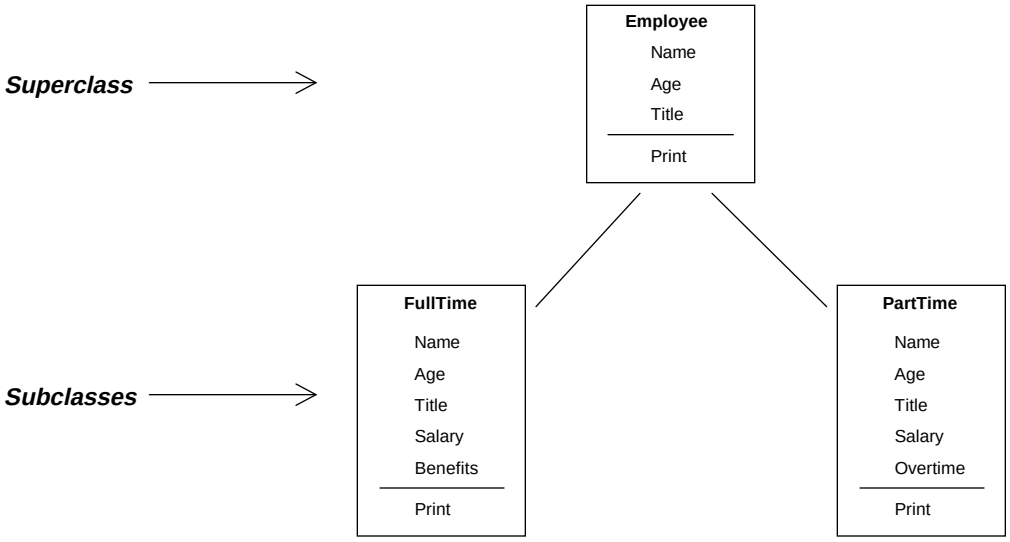
The best solution, then, is to subclass the Employee class to define two new classes, named "FullTime" and "PartTime."
The new *subclasses* (FullTime and PartTime) inherit everything from their *superclass* (Employee)—both their data (name, age, title) and their methods (Print):



The subclass, then, can define its own unique data and methods, as well as modify the behavior of its inherited methods (as opposed to rewriting the originals).

For example, you need to have separate code to handle the salary calculations in the FullTime and PartTime subclasses because they are not computed in the same way. Thus, objects created from the FullTime and PartTime classes would still have all the basic characteristics of an Employee, but each would have a specialized

Salary property, computed differently depending on the object's class. Each would also have properties unique
to it, as shown below:



Note: The interface to these two subclasses is identical. In both cases, you refer to Salary without knowing or caring how the underlying class computes the value.

Inheritance is not necessarily only one level deep—you could potentially go on to create more subclasses and perhaps even subclasses from those subclasses. If you look at inheritance like a tree, a subclass inherits not only from its immediate parent but from *all* of its ancestors.

Inheritance, then, is the programming technique by which you adapt the behavior of a component without changing that component. Because you can give the component new behavior without in any way destabilizing it, you achieve safe, incremental enhancement. You progress from a stable status to an improved stable status.

A Real-World Example

Let's return again to the information system for which we designed the Employee class. What components would you likely need for this system? Well, a database for sure. In fact, you'll probably need several tables to keep track of employees, customers, inventory, sales transactions, financial records, etc.

You might also want a calendar, which will be used to track shipments and billing dates. Finally, you'll probably want some mechanism through which you can generate reports (for example, total sales per month or a customer mailing list).

We're already on the path to creating an object-oriented information system, merely by the fact that we're thinking of the components of the system as classes—tables, a calendar, and a report generator.

The Table

Let's consider a single table first. We need to build the structure of the table by specifying what fields to include, what their data type should be, and other details. Next, we need to implement actions (methods) to make this table operational.

What are the methods we want to apply to a table object? Well, certainly, we'll want to add records and delete records. We'll also want to search for records, and possibly edit them. Thus, we've already determined four basic methods that will be available to every table we create.

Let's pause and take a look at what we've done. We designed a table structure, as well as several methods that we can use to provide access to the information in that table. By doing so, we've created our first class definition. Using this abstract definition of a table as the "master blueprint," we can now continue by creating more specific table designs, adding fields and functionality specific to each table, if necessary.

Tip: In fact, a class that has these methods is supplied with CA-Visual Objects. It is called DBServer and is located in the DBF Classes library.

The Calendar

Once we've designed the table, we can move on to the calendar. To keep the design simple, the calendar will hold information by month and day only. We can represent the calendar as an array of month objects. Each month object has an array of day objects, and we schedule appointments for these days. Each day has an array of appointments.

Now that we have the calendar object's structure defined, what methods will we possibly need? We'll definitely need a method to make an appointment and one to cancel an appointment. Perhaps a method to mark employees' vacation days would be helpful, too.

The Report Generator

You've probably got the idea by now. When we go to create our report generator class, we'll have to create its structure, which might include the page format and the information source. And for action, it would have at least a single method to send output to the printer.

Communicating Between Objects

You may have noticed that the three classes we've imagined have methods that apply only to themselves. For instance, the methods that add, delete, edit, and search records in the table act only upon the table itself, not on the calendar or the report generator.

We could also just as easily have given our objects methods that allow them to interact with each other. For example, the table might have methods to send the name of a person who has a scheduled appointment to the calendar. The calendar would then have a method to receive that name. The calendar might need a method to send information to the report generator to print a daily schedule of appointments. The report generator might need to access information from the table to create a report.

However, these methods would break the encapsulation of the individual classes. The right way to handle this is for a controlling object, such as a window, to manage traffic between the various objects.

Additional Strengths of OOP

We mentioned earlier that there were other benefits to OOP. Let's discuss some of these now.

Encapsulation

Encapsulation refers to the insulation of the inside of a thing from changes you make outside it (and vice versa). It is the hiding or protecting of data.

Client
The code that uses a class, or the person who writes that code.

When you develop classes, it is not necessary for the *client* of those classes to understand the inner workings of the class. For example, how exactly the Employee class goes about implementing its Print method is irrelevant to the client—only the fact that it exists and works is important. Or, to use a more simplistic example, the average person probably knows next to nothing about how a television works, and doesn't care as long as it turns on and off, switches channels on command, and presents a high-quality picture.

From a software perspective, encapsulation plays an important role. When you design a class, you can hide certain data, so that when an object is instantiated from the class, the data is invisible to other objects. This allows *controlled access* to data: the only way that code outside the class can touch protected data is through methods or properties. Any external modification to the data, therefore, is done only with "permission."

If we consider the television example again, a person really should not (and therefore cannot) bypass the volume control and manipulate the volume by touching the television's internal working parts. Encapsulation prevents this. By hiding internal details of objects, and giving access only to things that should be accessed, encapsulation provides a simple and safe framework for working with objects and the data they hold.

Modularity and Reusability

Standard Components = Reusability

When constructing a house, you always use prefabricated components: girders, door frames, sink units. When constructing a computer, you assemble prefabricated integrated circuits, power supplies, and disk drives. Nobody would consider producing such complex artifacts from concrete and raw timber, from silicon and iron ore.

The same should be true for software—yet, when constructing software applications, often equally complex, the tradition of using prefabricated components is not as well established. The proliferation of OOP, however, has started to address this shortcoming.

Object-orientation extends the developers' ability to write modular, reusable code. Objects are essentially packaged code and data. Bolstered by encapsulation and inheritance, objects become powerful application building blocks. Future applications will be easier to create because they will be programmed by simply assembling component parts—imagine constructing a personnel package by just bundling together the payroll system and employee benefits components!

Modularity = Safe, Incremental Enhancements

In the long run, however, software construction is probably going to be more demanding than the construction of a house or computer, because it continuously requires adjustment and elaboration.

For this reason, the standard components must be easy to modify and adapt to new uses and circumstances, and the architecture must allow the continuous rearrangement of components and addition of new components.

The tenets of object-orientation hold that proven code rarely needs to be touched when enhancements or other changes are necessary. Unless code written in the past is found to be incorrect—or incompatible with future interface designs—code need not be modified. Instead, changes are made by creating a subclass through inheritance and coding only what is new or different. Therefore, only one version of any piece of code need ever exist—code is reusable.

In addition, class definitions can be grouped together in user-defined libraries. Over time, these libraries can grow to form powerful

application building blocks. Since new functionality is added via inheritance, source code in the class library never changes (and consequently, user-defined class libraries are easier to maintain than function libraries).

Because objects are intrinsically modular and reusable, programmers automatically achieve several benefits:

- There is less code to write and debug. The development cycle, therefore, is streamlined and more productive.
- The quality of resulting applications is higher, since reused components are more frequently used and therefore better tested.
- Reusability decreases maintenance—it is easier to make changes (both enhancements and corrections) without side effects.
- Because code is of higher quality and is better tested, and because development time is streamlined, programmers can focus more on design, creating applications that are more sophisticated and robust.

Summary

Hopefully, it's becoming clear that object-orientation is a good way to manage GUI applications. OOP is the answer to many of the programming complexities and challenges presented by GUI environments. Windows development is the perfect place to put object-oriented theory into practice.

But it doesn't stop there. Applications in general can be thought of and implemented in object-oriented terms, whether or not they are GUI, whether or not they utilize databases, or even if they perform only rudimentary tasks.

The CA-Visual Objects Libraries

To facilitate object-oriented programming, CA-Visual Objects includes a set of extensive libraries. These libraries provide very powerful building blocks for your applications.

In fact, much of the power of CA-Visual Objects comes from its libraries. They provide an elegant and extensible way of using supporting services. They integrate well, not only with the programming language but also with the IDE (for example, the Class Browser). Libraries also provide an extremely effective way of insulating application code from platform-specific implementation details.

CA-Visual Objects provides the following class and function libraries:

- System Classes
- GUI Classes
- DBF Classes
- SQL Classes
- Report Classes
- System Library
- DBF
- Terminal
- Windows API

System Classes

This library defines classes that are used by the other system class libraries (that is, GUI Classes, DBF Classes, and SQL Classes). Whenever you associate one of these class libraries with your application, you should also associate System Classes with it. To use the code generated by the Menu, DB Server, FieldSpec, SQL Server, and Window Editors, you *must* associate this library with your application.

GUI Classes

This library contains over a hundred classes that allow you to create the objects required for a full-featured GUI. It includes facilities for creating windows, menus, push buttons, scroll bars, status bars, list boxes, and so on, and also includes simple shapes and other abstractions associated with a GUI.

Using the GUI Classes library also gives you access to the Standard Program (which you will explore thoroughly in "Learning the Basics" later in this guide), robust error and exception handling, and a wide range of stock objects, such as useful icons, bitmaps, and colors.

You must associate this library with your applications if you plan to use code generated by the Window Editor and/or the Menu Editor.

DBF Classes

This library provides an OOP interface to Xbase .DBF files using classes and methods instead of traditional commands and functions. It must be associated with your applications if you plan to use code generated by the DB Server Editor.

SQL Classes

This library provides an OOP interface to SQL tables using classes and methods instead of traditional SQL statements, and must be associated with your applications if you plan to use code generated by the SQL Editor.

As described earlier, SQL database access is accomplished using the ODBC protocol. Therefore, for your convenience, this library also contains ODBC API function definitions that can be used to program directly to the ODBC API.

Note: The ODBC API functions are not specific to CA-Visual Objects and, therefore, are not included in our documentation. Refer to the standard ODBC documentation provided by your ODBC vendor for details about these functions.

Report Classes

This library provides an OOP interface to CA-RET, and must be associated with your applications if you plan to use code generated by the Report Editor.

System Library	This library provides basic system function support. It also provides a small subset of Windows API functions and constants that are used by the system. It is automatically associated with every CA-Visual Objects application.
DBF	This library provides support for traditional Xbase database commands and functions like SKIP and EOF(). It must be associated with your applications if you plan to use these techniques.
Terminal	This library contains compatibility commands and functions for traditional Xbase screen I/O techniques (for example, @...SAY...GET). You should associate this library with your applications only if you plan to use the terminal emulation layer.
Windows API	This library contains Windows API function, constant, and structure definitions. You should associate this library with your applications only if you plan to exploit low-level, system programming. **Note:** The Windows API functions are not specific to CA-Visual Objects and, therefore, are not included in our documentation. Refer to your *Microsoft Windows Software Development Kit Programmer's Reference, Volume II: Functions* for details about these functions.

What's Next

To gain a better understanding of the features available to you, the next chapter points out some of the various features of the CA-Visual Objects development environment.