1

Introduction to Object-Oriented Concepts

Although many programmers don't realize it, object-oriented (OO) software development has been around since the early 1960s. It wasn't until the mid to late 1990s that the object-oriented paradigm started to gain momentum, despite the fact that popular object-oriented programming languages such as Smalltalk and C++ were already widely used.

The rise of OO methodologies coincides with the emergence of the Internet as a business and entertainment platform. In short, objects work well over a network. And after it became obvious that the Internet was here to stay, object-oriented technologies were already well positioned to develop the new web-based technologies.

It is important to note that the title of this first chapter is "Introduction to Object-Oriented Concepts." The operative word here is "concepts" and not "technologies." Technologies change very quickly in the software industry, whereas concepts evolve. I use the term "evolve" because, although they remain relatively stable, they do change. And this is what is really cool about focusing on the concepts. Despite their consistency, they are always undergoing reinterpretations, and this allows for some very interesting discussions.

This evolution can be easily traced over the past 20 years or so as we follow the progression of the various industry technologies from the first primitive browsers of the mid to late 1990s to the mobile/phone/web applications that dominate today. As always, new developments are just around the corner as we explore hybrid apps and more. Throughout this journey, OO concepts have been there every step of the way. That is why the topics of this chapter are so important. These concepts are just as relevant today as they were 20 years ago.

The Fundamental Concepts

The primary point of this book is to get you thinking about how the concepts are used in designing object-oriented systems. Historically, object-oriented languages are defined by the following: *encapsulation*, *inheritance*, and *polymorphism*. Thus, if a language does not implement

all of these, it is generally not considered completely object-oriented. Along with these three terms, I always include composition in the mix; thus, my list of object-oriented concepts looks like this:

- Encapsulation
- Inheritance
- Polymorphism
- Composition

We will discuss all these in detail as we proceed through the rest of the book.

One of the issues that I have struggled with right from the first edition of this book is how these concepts relate directly to current design practices, which are always changing. For example, there has always been debate about using inheritance in an OO design. Does inheritance actually break encapsulation? (This topic will be covered in later chapters.) Even now, many developers try to avoid inheritance as much as possible.

My approach is, as always, to stick to concepts. Whether or not you use inheritance, you at least need to understand what inheritance is, thus enabling you to make an educated design choice. As mentioned in the introduction, the intended audience is those who want *a general introduction to fundamental OO concepts*. With this statement in mind, in this chapter I present the fundamental object-oriented concepts with the hope that the reader will then gain a solid foundation for making important design decisions. The concepts covered here touch on most, if not all, of the topics covered in subsequent chapters, which explore these issues in much greater detail.

Objects and Legacy Systems

As OO moved into the mainstream, one of the issues facing developers was the integration of new OO technologies with existing systems. At the time, lines were being drawn between OO and structured (or procedural) programming, which was the dominant development paradigm at the time. I always found this odd because, in my mind, object-oriented and structured programming do not compete with each other. They are complementary because objects integrate well with structured code. Even now, I often hear this question: Are you a structured programmer or an object-oriented programmer? Without hesitation, I would answer: both.

In the same vein, object-oriented code is not meant to replace structured code. Many non-OO *legacy systems* (that is, older systems that are already in place) are doing the job quite well, so why risk potential disaster by changing or replacing them? In most cases, you should not change them, at least not for the sake of change. There is nothing inherently wrong with systems written in non–OO code. However, brand-new development definitely warrants the consideration of using OO technologies (in some cases, there is no choice but to do so).

Although there has been a steady and significant growth in OO development in the past 20 years, the global community's dependence on networks such as the Internet and mobile

infrastructures has helped catapult it even further into the mainstream. The literal explosion of transactions performed on browsers and mobile apps has opened up brand-new markets, where much of the software development is new and mostly unencumbered by legacy concerns. Even when there are legacy concerns, there is a trend to wrap the legacy systems in object wrappers.

Object Wrappers

Object wrappers are object-oriented code that includes other code inside. For example, you can take structured code (such as loops and conditions) and *wrap* it inside an object to make it look like an object. You can also use object wrappers to *wrap* functionality such as security features, nonportable hardware features, and so on. Wrapping structured code is covered in detail in Chapter 6, "Designing with Objects."

Today, one of the most interesting areas of software development is the integration of legacy code with mobile- and web-based systems. In many cases, a mobile web front-end ultimately connects to data that resides on a mainframe. Developers who can combine the skills of mainframe and mobile web development are in demand.

You probably experience objects in your daily life without even realizing it. These experiences can take place in your car, when you're talking on your cell phone, using your home entertainment system, playing computer games, and many other situations. The electronic highway has, in essence, become an object-based highway. As businesses gravitate toward the mobile web, they are gravitating toward objects because the technologies used for electronic commerce are mostly OO in nature.

Mobile Web

No doubt, the emergence of the Internet provided a major impetus for the shift to object-oriented technologies. This is because objects are well suited for use on networks. Although the Internet was at the forefront of this paradigm shift, mobile networks have now joined the mix in a major way. In this book, the term *mobile web* will be used in the context of concepts that pertain to both mobile app development and web development. The term *hybrid* app is sometimes used to refer to applications that render in browser on both web and mobile devices.

Procedural Versus OO Programming

Before we delve deeper into the advantages of OO development, let's consider a more fundamental question: What exactly is an object? This is both a complex and a simple question. It is complex because learning any method of software development is not trivial. It is simple because people already think in terms of objects.

For example, when you look at a person, you see the person as an object. And an object is defined by two components: attributes and behaviors. A person has attributes, such as eye color, age, height, and so on. A person also has behaviors, such as walking, talking, breathing, and so on. In its basic definition, an *object* is an entity that contains *both* data and behavior.

The word *both* is the key difference between OO programming and other programming methodologies. In procedural programming, for example, code is placed into totally distinct functions or procedures. Ideally, as shown in Figure 1.1, these procedures then become "black boxes," where inputs go in and outputs come out. Data is placed into separate structures and is manipulated by these functions or procedures.

Difference Between OO and Procedural

In OO design, the attributes and behaviors are contained within a single object, whereas in procedural, or structured, design, the attributes and behaviors are normally separated.

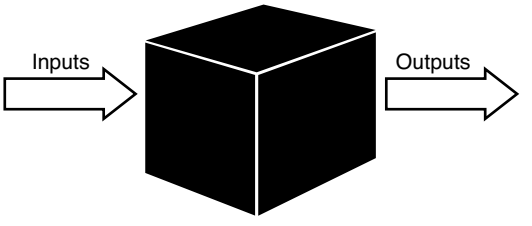

Figure 1.1 Black boxes.

As OO design grew in popularity, one of the realities that initially slowed its acceptance was that there were a lot of non-OO systems in place that worked perfectly fine. Thus, it did not make any business sense to change the systems for the sake of change. Anyone who is familiar with any computer system knows that any change can spell disaster—even if the change is perceived to be slight.

This situation came into play with the lack of acceptance of OO databases. At one point in the emergence of OO development, it seemed somewhat likely that OO databases would replace relational databases. However, this never happened. Businesses have a lot of money invested in relational databases, and one overriding factor discouraged conversion—they worked. When all the costs and risks of converting systems from relational to OO databases became apparent, there was no compelling reason to switch.

In fact, the business forces have now found a happy middle ground. Much of the software development practices today have flavors of several development methodologies, such as OO and structured.

As illustrated in Figure 1.2, in structured programming the data is often separated from the procedures, and often the data is global, so it is easy to modify data that is outside the scope of your code. This means that access to data is uncontrolled and unpredictable (that is, multiple functions may have access to the global data). Second, because you have no control over who has access to the data, testing and debugging are much more difficult. Objects address these problems by combining data and behavior into a nice, complete package.

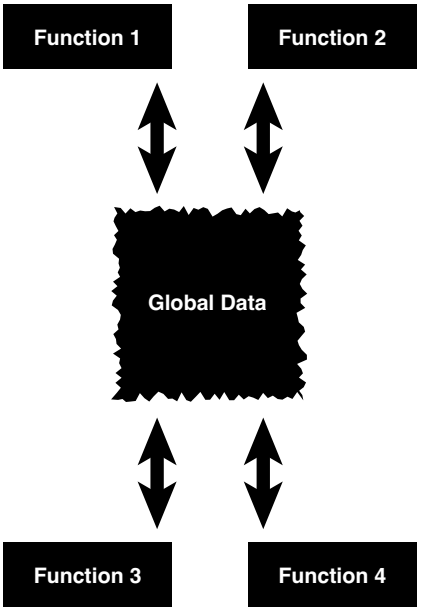

Figure 1.2 Using global data.

Proper Design

We can state that when properly designed, there is no such thing as global data in an OO model. This fact provides a high amount of data integrity in OO systems.

Rather than replacing other software development paradigms, objects are an evolutionary response. Structured programs have complex data structures, such as arrays, and so on. C++ has structures, which have many of the characteristics of objects (classes).

However, objects are much more than data structures and primitive data types, such as integers and strings. Although objects do contain entities such as integers and strings, which are used to represent attributes, they also contain methods, which represent behaviors. In an object, methods are used to perform operations on the data as well as other actions. Perhaps more important, you can control access to members of an object (both attributes and methods). This means that some members, both attributes and methods, can be hidden from other objects. For instance, an object called `Math` might contain two integers, called `myInt1` and `myInt2`. Most likely, the `Math` object also contains the necessary methods to set and retrieve the values of `myInt1` and `myInt2`. It might also contain a method called `sum()` to add the two integers together.

Data Hiding

In OO terminology, data are referred to as *attributes*, and behaviors are referred to as *methods*. Restricting access to certain attributes and/or methods is called *data hiding*.

By combining the attributes and methods in the same entity, which in OO parlance is called *encapsulation*, we can control access to the data in the `Math` object. By defining these integers as off-limits, another logically unconnected function cannot manipulate the integers `myInt1` and `myInt2`—only the `Math` object can do that.

Sound Class Design Guidelines

Keep in mind that it is possible to create poorly designed OO classes that do not restrict access to class attributes. The bottom line is that you can design bad code just as efficiently with OO design as with any other programming methodology. Simply take care to adhere to sound class design guidelines (see Chapter 5, "Class Design Guidelines").

What happens when another object—for example, `myObject` wants to gain access to the sum of `myInt1` and `myInt2`? It asks the `Math` object: `myObject` sends a message to the `Math` object. Figure 1.3 shows how the two objects communicate with each other via their methods. The message is really a call to the `Math` object's `sum` method. The `sum` method then returns the value to `myObject`. The beauty of this is that `myObject` does not need to know how the sum is calculated (although I'm sure it can guess). With this design methodology in place, you can change how the `Math` object calculates the sum without making a change to `myObject` (as long as the means to retrieve the sum do not change). All you want is the sum—you *don't care* how it is calculated.

Using a simple calculator example illustrates this concept. When determining a sum with a calculator, all you use is the calculator's interface—the keypad and LED display. The calculator has a sum method that is invoked when you press the correct key sequence. You may get the correct answer back; however, you have no idea how the result was obtained—either electronically or algorithmically.

Calculating the sum is not the responsibility of `myObject`—it's the `Math` object's responsibility. As long as `myObject` has access to the `Math` object, it can send the appropriate messages and obtain the proper result. In general, objects should not manipulate the internal data of other objects (that is, `myObject` should not directly change the value of `myInt1` and `myInt2`). And, for reasons we will explore later, it is normally better to build small objects with specific tasks rather than build large objects that perform many.

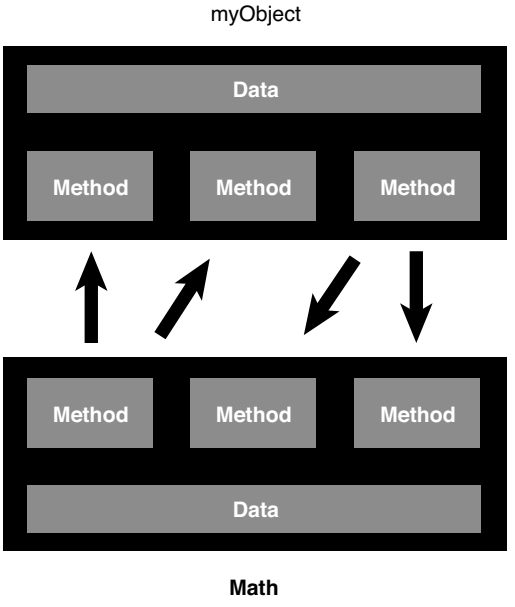

Figure 1.3 Object-to-object communication.

Moving from Procedural to Object-Oriented Development

Now that we have a general understanding about some of the differences about procedural and object-oriented technologies, let's delve a bit deeper into both.

Procedural Programming

Procedural programming normally separates the data of a system from the operations that manipulate the data. For example, if you want to send information across a network, only the relevant data is sent (see Figure 1.4), with the expectation that the program at the other end of the network pipe knows what to do with it. In other words, some sort of handshaking agreement must be in place between the client and server to transmit the data. In this model, it is quite possible that no code is actually sent over the wire.

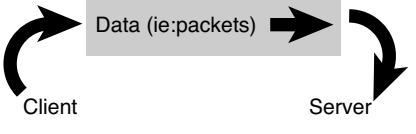

Figure 1.4 Data transmitted over a wire.

OO Programming

The fundamental advantage of OO programming is that the data and the operations that manipulate the data (the code) are both encapsulated in the object. For example, when an object is transported across a network, the entire object, including the data and behavior, goes with it.

> **A Single Entity**
>
> Although thinking in terms of a single entity is great in theory, in many cases, the behaviors themselves may not be sent because both sides have copies of the code. However, it is important to think in terms of the entire object being sent across the network as a single entity.

In Figure 1.5, the `Employee` object is sent over the network.

> **Proper Design**
>
> A good example of this concept is an object that is loaded by a browser. Often, the browser has no idea of what the object will do ahead of time because the code is not there previously. When the object is loaded, the browser executes the code within the object and uses the data contained within the object.

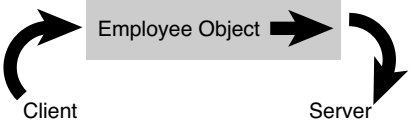

Figure 1.5 Objects transmitted over a wire.

What Exactly Is an Object?

Objects are the building blocks of an OO program. A program that uses OO technology is basically a collection of objects. To illustrate, let's consider that a corporate system contains objects that represent employees of that company. Each of these objects is made up of the data and behavior described in the following sections.

Object Data

The data stored within an object represents the state of the object. In OO programming terminology, this data is called *attributes*. In our example, as shown in Figure 1.6, employee attributes could be Social Security numbers, date of birth, gender, phone number, and so on. The attributes contain the information that differentiates between the various objects, in this case the employees. Attributes are covered in more detail later in this chapter in the discussion on classes.

Figure 1.6 Employee attributes.

Object Behaviors

The *behavior* of an object represents what the object can do. In procedural languages, the behavior is defined by procedures, functions, and subroutines. In OO programming terminology, these behaviors are contained in *methods*, and you invoke a method by sending a message to it. In our employee example, consider that one of the behaviors required of an employee object is to set and return the values of the various attributes. Thus, each attribute would have corresponding methods, such as `setGender()` and `getGender()`. In this case, when another object needs this information, it can send a message to an employee object and ask it what its gender is.

Not surprisingly, the application of getters and setters, as with much of object-oriented technology, has evolved since the first edition of this book was published. This is especially true when in comes to data. As we will see in Chapter 11, "Objects and Portable Data: XML and JSON," and Chapter 12, "Persistent Objects: Serialization, Marshalling, and Relational Databases," data is now constructed in an object-oriented manner. Remember that one of the most interesting, not to mention powerful, advantages of using objects is that the data is part of the package—it is not separated from the code.

The emergence of XML has not only focused attention on presenting data in a portable manner; it also has facilitated alternative ways for the code to access the data. In .NET techniques, the getters and setters are considered properties of the data itself.

For example, consider an attribute called Name, using Java, that looks like the following:

```
public String Name;
```

The corresponding getter and setter would look like this:

```
public void setName (String n) {name = n;};
public String getName() {return name;};
```

Now, when creating an XML attribute called `Name`, the definition in C# .NET may look something like this:

```
Private string strName;
public String Name
{
    get
    {
        return this.strName;
    }
    set
    {
        if (value == null) return;
        this.strName = value;
    }
}
```

In this approach, the getters and setters are actually *properties* of the attributes—in this case, `Name`.

Regardless of the approach, the purpose is the same—controlled access to the attribute. For this chapter, I want to first concentrate on the conceptual nature of accessor methods; we will get more into properties when we cover object-oriented data in Chapter 11 and beyond.

> **Getters and Setters**
>
> The concept of getters and setters supports the concept of data hiding. Because other objects should not directly manipulate data within another object, the getters and setters provide controlled access to an object's data. Getters and setters are sometimes called accessor methods and mutator methods, respectively.

Note that we are showing only the interface of the methods, and not the implementation. The following information is all the user needs to know to effectively use the methods:

- The name of the method
- The parameters passed to the method
- The return type of the method

To illustrate behaviors, consider Figure 1.7.

In Figure 1.7, the `Payroll` object contains a method called `CalculatePay()` that calculates the pay for a specific employee. Among other information, the `Payroll` object must obtain the Social Security number of this employee. To get this information, the payroll object must send a message to the `Employee` object (in this case, the `getSocialSecurityNumber()` method). Basically, this means that the `Payroll` object calls the `getSocialSecurityNumber()` method of the `Employee` object. The employee object recognizes the message and returns the requested information.

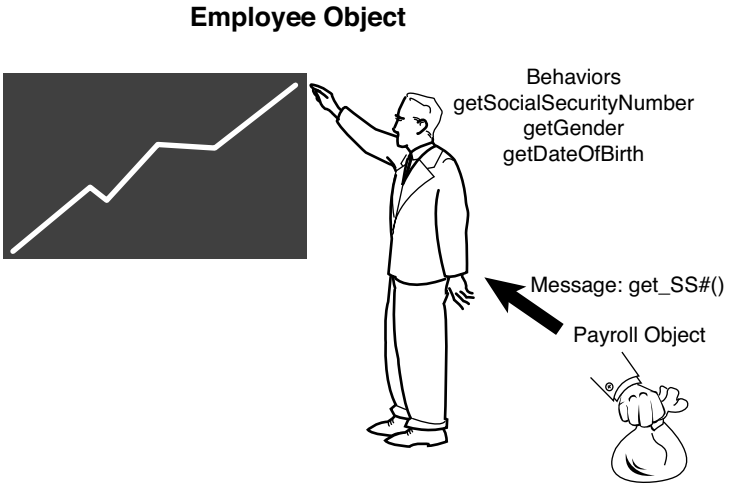

Figure 1.7 Employee behaviors.

To illustrate further, Figure 1.8 is a class diagram representing the `Employee/Payroll` system we have been talking about.

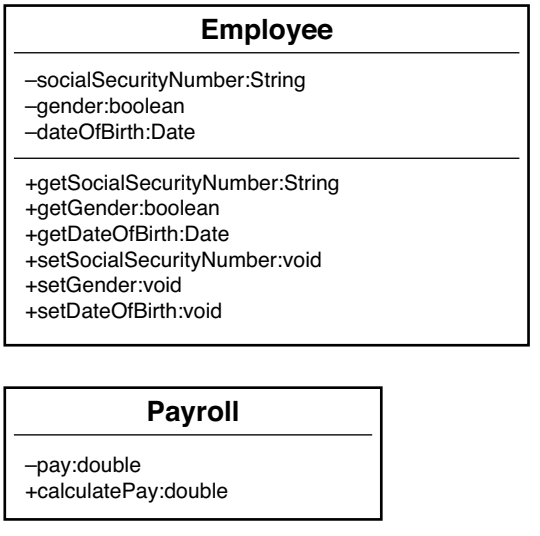

Figure 1.8 Employee and payroll class diagrams.

UML Class Diagrams

Because this is the first class diagram we have seen, it is very basic and lacks some of the constructs (such as constructors) that a proper class should contain. Fear not—we will discuss class diagrams and constructors in more detail in Chapter 3, "Advanced Object-Oriented Concepts."

Each class diagram is defined by three separate sections: the name itself, the data (attributes), and the behaviors (methods). In Figure 1.8, the `Employee` class diagram's attribute section contains `SocialSecurityNumber`, `Gender`, and `DateofBirth`, whereas the method section contains the methods that operate on these attributes. You can use UML modeling tools to create and maintain class diagrams that correspond to real code.

Modeling Tools

Visual modeling tools provide a mechanism to create and manipulate class diagrams using the Unified Modeling Language (UML). Class diagrams are discussed throughout this book, and you can find a description of this notation in Chapter 10, "Creating Object Models." UML class diagrams are used as a tool to help visualize classes and their relationships to other classes. The use of UML in this book is limited to class diagrams.

We will get into the relationships between classes and objects later in this chapter, but for now you can think of a class as a template from which objects are made. When an object is created, we say that the objects are instantiated. Thus, if we create three employees, we are actually creating three totally distinct instances of an `Employee` class. Each object contains its own copy of the attributes and methods. For example, consider Figure 1.9. An employee object called `John` (John is its identity) has its own copy of all the attributes and methods defined in the `Employee` class. An employee object called `Mary` has its own copy of attributes and methods. They both have a separate copy of the `DateOfBirth` attribute and the `getDateOfBirth` method.

An Implementation Issue

Be aware that there is not necessarily a physical copy of each method for each object. Rather, each object points to the same implementation. However, this is an issue left up to the compiler/operating platform. From a conceptual level, you can think of objects as being wholly independent and having their own attributes and methods.

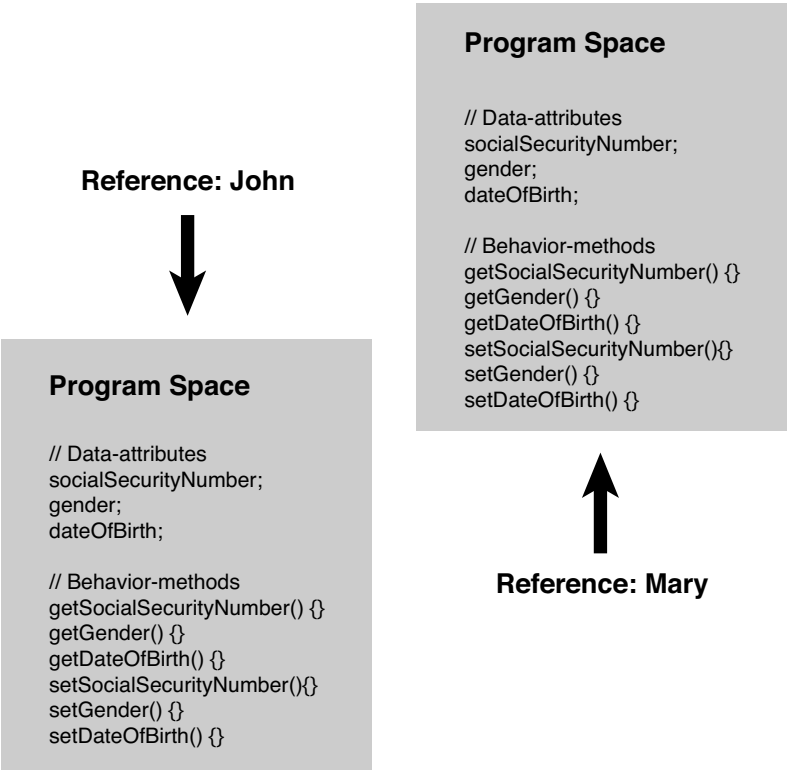

Figure 1.9 Program spaces.

What Exactly Is a Class?

In short, a class is a blueprint for an object. When you instantiate an object, you use a class as the basis for how the object is built. In fact, trying to explain classes and objects is really a chicken-and-egg dilemma. It is difficult to describe a class without using the term *object* and visa versa. For example, a specific individual bike is an object. However, someone had to have created the blueprints (that is, the class) to build the bike. In OO software, unlike the chicken-and-egg dilemma, we do know what comes first—the class. An object cannot be instantiated without a class. Thus, many of the concepts in this section are similar to those presented earlier in the chapter, especially when we talk about attributes and methods.

To explain classes and methods, it's helpful to use an example from the relational database world. In a database table, the definition of the table itself (fields, description, and data types used) would be a class (metadata), and the objects would be the rows of the table (data).

This book focuses on the concepts of OO software and not on a specific implementation (such as Java, C#, Visual Basic .NET, Objective C, or C++), but it is often helpful to use code examples

to explain some concepts, so Java code fragments are used throughout the book to help explain some concepts when appropriate. However, when appropriate, the end of each chapter contains the chapter's example code in C#. Much of the code presented in the book is available electronically on the publisher's website. For many chapters, the code examples are provided electronically in Java, C# .Net, VB .NET, and Objective C.

The following sections describe some of the fundamental concepts of classes and how they interact.

Creating Objects

Classes can be thought of as the templates, or cookie cutters, for objects, as seen in Figure 1.10. A class is used to create an object.

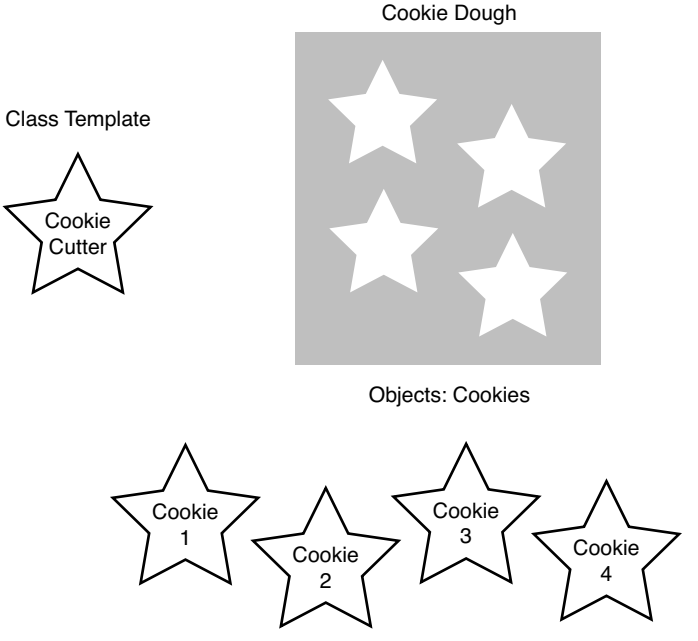

Figure 1.10 Class template.

A class can be thought of as a sort of higher-level data type. For example, just as you create an integer or a float:

```
int x;
float y;
```

you can also create an object by using a predefined class:

```
myClass myObject;
```

In this example, the names themselves make it obvious that `myClass` is the class and `myObject` is the object.

Remember that each object has its own attributes (data) and behaviors (functions or routines). A class defines the attributes and behaviors that all objects created with this class will possess. Classes are pieces of code. Objects instantiated from classes can be distributed individually or as part of a library. Because objects are created from classes, it follows that classes must define the basic building blocks of objects (attributes, behavior, and messages). In short, you must design a class before you can create an object.

For example, here is a definition of a `Person` class:

```
public class Person{

    //Attributes
    private String name;
    private String address;

    //Methods
    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }

    public String getAddress(){
        return address;
    }
    public void setAddress(String adr){
        address = adr;
    }

}
```

Attributes

As you already saw, the data of a class is represented by attributes. Each class must define the attributes that will store the state of each object instantiated from that class. In the `Person` class example in the previous section, the `Person` class defines attributes for `name` and `address`.

> ### Access Designations
>
> When a data type or method is defined as `public`, other objects can directly access it. When a data type or method is defined as `private`, only that specific object can access it. Another access modifier, `protected`, allows access by related objects, which you'll learn about in Chapter 3.

Methods

As you learned earlier in the chapter, methods implement the required behavior of a class. Every object instantiated from this class has the methods as defined by the class. Methods may implement behaviors that are called from other objects (messages) or provide the fundamental, internal behavior of the class. Internal behaviors are private methods that are not accessible by other objects. In the `Person` class, the behaviors are `getName()`, `setName()`, `getAddress()`, and `setAddress()`. These methods allow other objects to inspect and change the values of the object's attributes. This is common technique in OO systems. In all cases, access to attributes within an object should be controlled by the object itself—no other object should directly change an attribute of another.

Messages

Messages are the communication mechanism between objects. For example, when Object A invokes a method of Object B, Object A is sending a message to Object B. Object B's response is defined by its return value. Only the public methods, not the private methods, of an object can be invoked by another object. The following code illustrates this concept:

```
public class Payroll{

    String name;

    Person p = new Person();

    String = p.setName("Joe");

    ... code

    String = p.getName();

}
```

In this example (assuming that a `Payroll` object is instantiated), the `Payroll` object is sending a message to a `Person` object, with the purpose of retrieving the name via the `getName()` method. Again, don't worry too much about the actual code, because we are really interested in the concepts. We address the code in detail as we progress through the book.

Using Class Diagrams as a Visual Tool

Over the years, many tools and modeling methodologies have been developed to assist in designing software systems. Right from the start, I have used UML class diagrams to assist in the educational process. Although it is beyond the scope of this book to describe UML in any detail, we will use UML class diagrams to illustrate the classes that we build. In fact, we have already used class diagrams in this chapter. Figure 1.11 shows the `Person` class diagram we discussed earlier in the chapter.

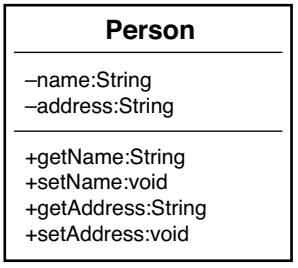

Figure 1.11 The `Person` class diagram.

As we saw previously, notice that the attributes and methods are separated (the attributes on the top, and the methods on the bottom). As we delve more deeply into OO design, these class diagrams will get much more sophisticated and convey much more information on how the different classes interact with each other.

Encapsulation and Data Hiding

One of the primary advantages of using objects is that the object need not reveal all its attributes and behaviors. In good OO design (at least what is generally accepted as good), an object should reveal only the interfaces that other objects must have to interact with it. Details not pertinent to the use of the object should be hidden from all other objects.

Encapsulation is defined by the fact that objects contain both the attributes and behaviors. Data hiding is a major part of encapsulation.

For example, an object that calculates the square of a number must provide an interface to obtain the result. However, the internal attributes and algorithms used to calculate the square need not be made available to the requesting object. Robust classes are designed with encapsulation in mind. In the next sections, we cover the concepts of interface and implementation, which are the basis of encapsulation.

Interfaces

We have seen that the interface defines the fundamental means of communication between objects. Each class design specifies the interfaces for the proper instantiation and operation of objects. Any behavior that the object provides must be invoked by a message sent using one of the provided interfaces. The interface should completely describe how users of the class interact with the class. In most OO languages, the methods that are part of the interface are designated as `public`.

> **Private Data**
>
> For data hiding to work, all attributes should be declared as `private`. Thus, attributes are never part of the interface. Only the `public` methods are part of the class interface. Declaring an attribute as `public` breaks the concept of data hiding.

Let's look at the example just mentioned: calculating the square of a number. In this example, the interface would consist of two pieces:

- How to instantiate a `Square` object
- How to send a value to the object and get the square of that value in return

As discussed earlier in the chapter, if a user needs access to an attribute, a method is created to return the value of the attribute (a getter). If a user then wants to obtain the value of an attribute, a method is called to return its value. In this way, the object that contains the attribute controls access to it. This is of vital importance, especially in security, testing, and maintenance. If you control the access to the attribute, when a problem arises, you do not have to worry about tracking down every piece of code that might have changed the attribute—it can be changed in only one place (the setter).

From a security perspective, you don't want uncontrolled code to change or retrieve data such as passwords and personal information.

> **Signatures—Interfaces Versus Interfaces**
>
> It is important to note that there are interfaces to the classes as well as the methods; don't confuse the two. The interfaces to the classes are the public methods. You invoke these methods by using their signature, which primarily consists of the method name and its parameter list. This concept is covered in more detail later.

Implementations

Only the public attributes and methods are considered the interface. The user should not see any part of the internal implementation—interacting with an object solely through class interfaces. Thus, anything defined as private is inaccessible to the user and considered part of the class's internal implementation.

In the previous example, for instance the `Employee` class, only the attributes were hidden. In many cases, there will be methods that also should be hidden and thus not part of the interface. Continuing the example of the square root from the previous section, the user does not care how the square root is calculated—as long as it is the correct answer. Thus, the implementation can change, and it will not affect the user's code. For example, the company that produces the calculator can change the algorithm (perhaps because it is more efficient) without affecting the result.

A Real-World Example of the Interface/Implementation Paradigm

Figure 1.12 illustrates the interface/implementation paradigm using real-world objects rather than code. The toaster requires electricity. To get this electricity, the cord from the toaster must be plugged into the electrical outlet, which is the interface. All the toaster needs to do to obtain the required electricity is to implement a cord that complies with the electrical outlet specifications; this is the interface between the toaster and the power company (actually the power industry). That the actual implementation is a coal-powered electric plant is not the concern of the toaster. In fact, for all the toaster cares, the implementation could be a nuclear power plant or a local power generator. With this model, any appliance can get electricity, as long as it conforms to the interface specification, as shown in Figure 1.12.

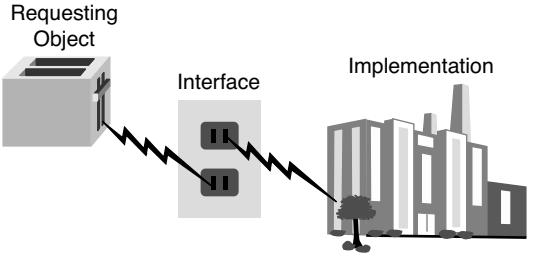

Figure 1.12 Power plant example.

A Model of the Interface/Implementation Paradigm

Let's explore the `Square` class further. Assume that you are writing a class that calculates the squares of integers. You must provide a separate interface and implementation. That is, you must specify a way for the user to invoke and obtain the square value. You must also provide the implementation that calculates the square; however, the user should not know anything about the specific implementation. Figure 1.13 shows one way to do this. Note that in the class diagram, the plus sign (+) designates public and the minus sign (-) designates private. Thus, you can identify the interface by the methods, prefaced with plus signs.

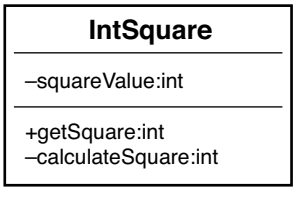

Figure 1.13 The square class.

This class diagram corresponds to the following code:

```
public class IntSquare {

    // private attribute
    private int squareValue;

    // public interface
    public int getSquare (int value) {

        SquareValue =calculateSquare(value);

        return squareValue;

    }

    // private implementation
    private int calculateSquare (int value) {

        return value*value;

    }
}
```

Note that the only part of the class that the user has access to is the public method `getSquare`, which is the interface. The implementation of the square algorithm is in the method `calculateSquare`, which is private. Also notice that the attribute `SquareValue` is private because users do not need to know that this attribute exists. Therefore, we have hidden the part of the implementation: The object reveals only the interfaces the user needs to interact with it, and details that are not pertinent to the use of the object are hidden from other objects.

If the implementation were to change—suppose you wanted to use the language's built-in square function—you would not need to change the interface. Here the code uses the Java library method Math.pow, which performs the same function, but note that the interface is still `calculateSquare`:

```
// private implementation
private int calculateSquare (int value) {

    return = Math.pow(value,2);

}
```

The user would get the same functionality using the same interface, but the implementation would have changed. This is very important when you're writing code that deals with data; for example, you can move data from a file to a database without forcing the user to change any application code.

Inheritance

One of the most powerful features of OO programming is, perhaps, code reuse. Structured design provides code reuse to a certain extent—you can write a procedure and then use it as many times as you want. However, OO design goes an important step further, allowing you to define relationships between classes that facilitate not only code reuse, but also better overall design, by organizing classes and factoring in commonalties of various classes. *Inheritance* is a primary means of providing this functionality.

Inheritance allows a class to inherit the attributes and methods of another class. This allows creation of brand-new classes by abstracting out common attributes and behaviors.

One of the major design issues in OO programming is to factor out commonality of the various classes. For example, suppose you have a `Dog` class and a `Cat` class, and each will have an attribute for eye color. In a procedural model, the code for `Dog` and `Cat` would each contain this attribute. In an OO design, the color attribute could be moved up to a class called `Mammal`—along with any other common attributes and methods. In this case, both `Dog` and `Cat` inherit from the `Mammal` class, as shown in Figure 1.14.

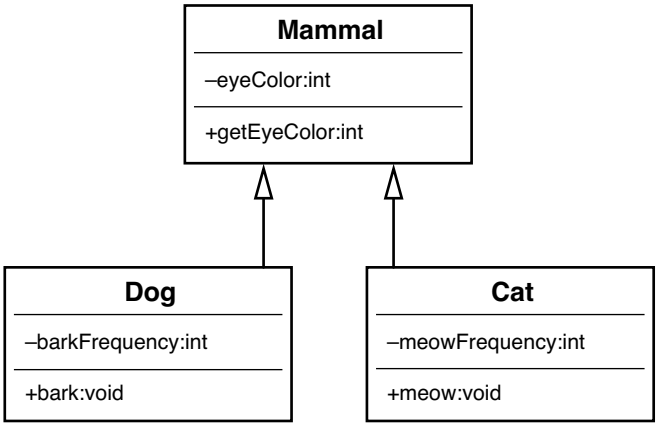

Figure 1.14 Mammal hierarchy.

The `Dog` and `Cat` classes both inherit from `Mammal`. This means that a `Dog` class has the following attributes:

```
eyeColor         // inherited from Mammal
barkFrequency    // defined only for Dogs
```

In the same vein, `Dog` object has the following methods:

```
getEyeColor    // inherited from Mammal
bark         // defined only for Dogs
```

When the Dog or the Cat object is instantiated, it contains everything in its own class, as well as everything from the parent class. Thus, Dog has all the properties of its class definition, as well as the properties inherited from the Mammal class.

Superclasses and Subclasses

The superclass, or parent class (sometimes called base class), contains all the attributes and behaviors that are common to classes that inherit from it. For example, in the case of the Mammal class, all mammals have similar attributes, such as eyeColor and hairColor, as well as behaviors, such as generateInternalHeat and growHair. All mammals have these attributes and behaviors, so it is not necessary to duplicate them down the inheritance tree for each type of mammal. Duplication requires a lot more work, and perhaps more worrisome, it can introduce errors and inconsistencies.

The subclass, or child class (sometimes called derived class), is an extension of the superclass. Thus, the Dog and Cat classes inherit all those common attributes and behaviors from the Mammal class. The Mammal class is considered the superclass of the Dog and the Cat subclasses, or child classes.

Inheritance provides a rich set of design advantages. When you're designing a Cat class, the Mammal class provides much of the functionality needed. By inheriting from the Mammal object, Cat already has all the attributes and behaviors that make it a true mammal. To make it more specifically a cat type of mammal, the Cat class must include any attributes or behaviors that pertain solely to a cat.

Abstraction

An inheritance tree can grow quite large. When the Mammal and Cat classes are complete, other mammals, such as dogs (or lions, tigers, and bears), can be added quite easily. The Cat class can also be a superclass to other classes. For example, it might be necessary to abstract the Cat class further, to provide classes for Persian cats, Siamese cats, and so on. Just as with Cat, the Dog class can be the parent for GermanShepherd and Poodle (see Figure 1.15). The power of inheritance lies in its abstraction and organization techniques.

In most recent OO languages (such as Java, .NET, and Objective C), a class can have only a single parent class; however, a class can have many child classes. Some languages, such as C++, can have multiple parents. The former case is called *single-inheritance*, and the latter is called *multiple-inheritance*.

Note that the classes GermanShepherd and Poodle both inherit from Dog—each contains only a single method. However, because they inherit from Dog, they also inherit from Mammal. Thus, the GermanShepherd and Poodle classes contain all the attributes and methods included in Dog and Mammal, as well as their own (see Figure 1.16).

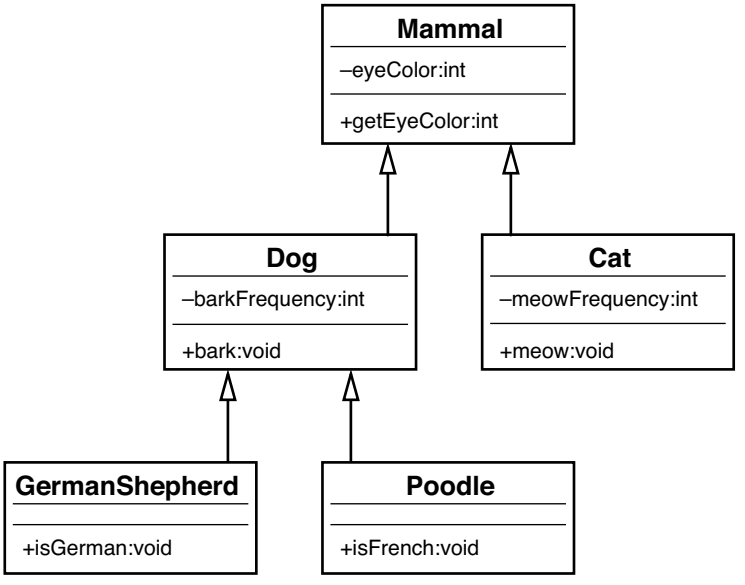

Figure 1.15 Mammal UML diagram.

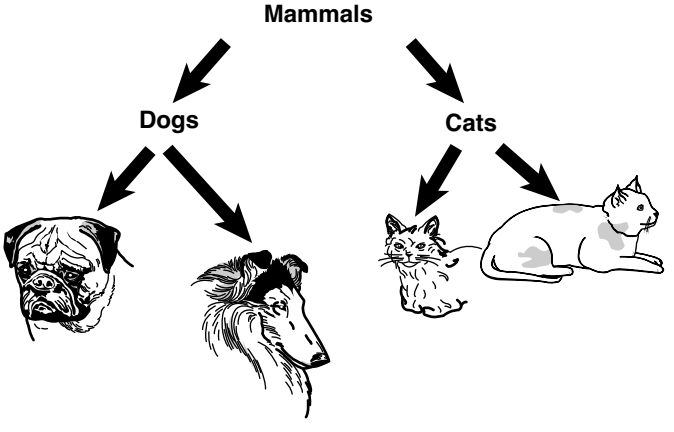

Figure 1.16 Mammal hierarchy.

Is-a Relationships

Consider a `Shape` example where `Circle`, `Square`, and `Star` all inherit directly from `Shape`. This relationship is often referred to as an *is-a relationship* because a circle is a shape, and a

square is a shape. When a subclass inherits from a superclass, it can do anything that the superclass can do. Thus, `Circle`, `Square`, and `Star` are all extensions of `Shape`.

In Figure 1.17, the name on each of the objects represents the `Draw` method for the `Circle`, `Star`, and `Square` objects, respectively. When we design this `Shape` system, it would be very helpful to standardize how we use the various shapes. Thus, we could decide that if we want to draw a shape, no matter what shape, we will invoke a method called `draw`. If we adhere to this decision, whenever we want to draw a shape, only the `Draw` method needs to be called, regardless of what the shape is. Here lies the fundamental concept of polymorphism—it is the individual object's responsibility, be it a `Circle`, `Star`, or `Square`, to draw itself. This is a common concept in many current software applications, such as drawing and word processing applications.

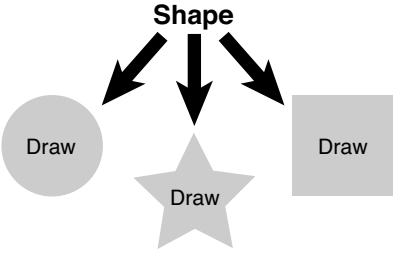

Figure 1.17 The shape hierarchy.

Polymorphism

Polymorphism is a Greek word that literally means many shapes. Although polymorphism is tightly coupled to inheritance, it is often cited separately as one of the most powerful advantages to object-oriented technologies. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message.

For example, consider the `Shape` class and the behavior called `Draw`. When you tell somebody to draw a shape, the first question asked is, "What shape?" No one can draw a shape, because it is an abstract concept (in fact, the `Draw()` method in the `Shape` code following contains no implementation). You must specify a concrete shape. To do this, you provide the actual implementation in `Circle`. Even though `Shape` has a `Draw` method, `Circle` overrides this method and provides its own `Draw()` method. Overriding basically means replacing an implementation of a parent with one from a child.

For example, suppose you have an array of three shapes—`Circle`, `Square`, and `Star`. Even though you treat them all as `Shape` objects, and send a `Draw` message to each `Shape` object, the end result is different for each because `Circle`, `Square`, and `Star` provide the actual

implementations. In short, each class is able to respond differently to the same `Draw` method and draw itself. This is what is meant by polymorphism.

Consider the following `Shape` class:

```
public abstract class Shape{

    private double area;

    public abstract double getArea();

}
```

The `Shape` class has an attribute called `area` that holds the value for the area of the shape. The method `getArea()` includes an identifier called `abstract`. When a method is defined as `abstract`, a subclass must provide the implementation for this method; in this case, `Shape` is requiring subclasses to provide a `getArea()` implementation. Now let's create a class called `Circle` that inherits from `Shape` (the `extends` keyword specifies that `Circle` inherits from `Shape`):

```
public class Circle extends Shape{

    double radius;

    public Circle(double r) {

        radius = r;

    }

    public double getArea() {

        area = 3.14*(radius*radius);
        return (area);

    }
}
```

We introduce a new concept here called a *constructor*. The `Circle` class has a method with the same name, `Circle`. When a method name is the same as the class and no return type is provided, the method is a special method, called a constructor. Consider a constructor as the entry point for the class, where the object is built; the constructor is a good place to perform initializations and start-up tasks.

The `Circle` constructor accepts a single parameter, representing the radius, and assigns it to the `radius` attribute of the `Circle` class.

The `Circle` class also provides the implementation for the `getArea` method, originally defined as `abstract` in the `Shape` class.

We can create a similar class, called `Rectangle`:

```
public class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(double l, double w){
        length = l;
        width = w;
    }

    public double getArea() {
        area = length*width;
        return (area);
    }

}
```

Now we can create any number of rectangles, circles, and so on and invoke their `getArea()` method. This is because we know that all rectangles and circles inherit from `Shape`, and all `Shape` classes have a `getArea()` method. If a subclass inherits an abstract method from a superclass, it must provide a concrete implementation of that method, or else it will be an abstract class itself (see Figure 1.18 for a UML diagram). This approach also provides the mechanism to create other, new classes quite easily.

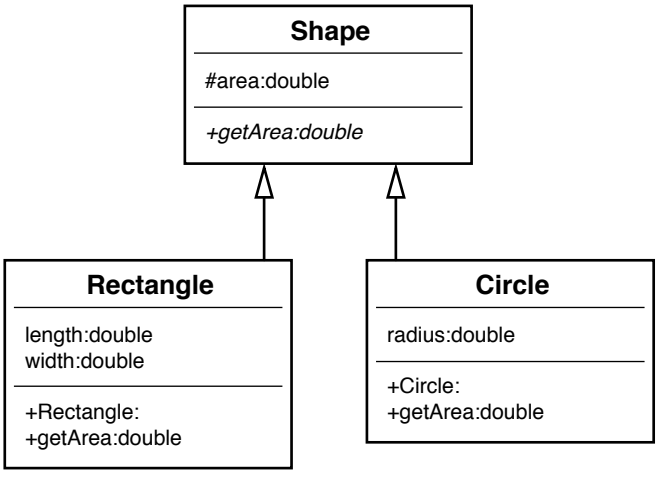

Figure 1.18 Shape UML diagram.

Thus, we can instantiate the `Shape` classes in this way:

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4,5);
```

Then, using a construct such as a stack, we can add these `Shape` classes to the stack:

```
stack.push(circle);
stack.push(rectangle);
```

> **What Is a Stack?**
>
> A stack is a data structure that is a last-in, first-out system. It is like a coin changer, where you insert coins at the top of the cylinder and, when you need a coin, you take one off the top, which is the last one you inserted. Pushing an item onto the stack means that you are adding an item to the top (like inserting another coin into the changer). Popping an item off the stack means that you are taking the last item off the stack (like taking the coin off the top).

Now comes the fun part. We can empty the stack, and we do not have to worry about what kind of `Shape` classes are in it (we just know they are shapes):

```
while (  !stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println ("Area = " + shape.getArea());
}
```

In reality, we are sending the same message to all the shapes:

```
shape.getArea()
```

However, the actual behavior that takes place depends on the type of shape. For example, `Circle` calculates the area for a circle, and `Rectangle` calculates the area of a rectangle. In effect (and here is the key concept), we are sending a message to the `Shape` classes and experiencing different behavior depending on what subclass of `Shape` is being used.

This approach is meant to provide standardization the interface across classes, as well as applications. Consider an office suite application that includes a word processing and a spreadsheet application. Let's assume that both have a class called Office which contains an interface called `print()`. This `print()` interface is required for all classes that are part of the office suite. The interesting thing here is that although both the word processor and spreadsheet invoke the print() interface, they do different things—one prints a word processing document and the other a spreadsheet document.

Composition

It is natural to think of objects as containing other objects. A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives. Although the computer can be considered an object unto itself, the drive is also considered a valid object.

In fact, you could open up the computer and remove the drive and hold it in your hand. Both the computer and the drive are considered objects. It is just that the computer contains other objects—such as drives.

In this way, objects are often built, or composed, from other objects: This is composition.

Abstraction

Just as with inheritance, composition provides a mechanism for building objects. In fact, I would argue that there are only two ways to build classes from other classes: *inheritance* and *composition*. As we have seen, inheritance allows one class to inherit from another class. We can thus abstract out attributes and behaviors for common classes. For example, dogs and cats are both mammals because a dog *is-a* mammal and a cat *is-a* mammal. With composition, we can also build classes by embedding classes in other classes.

Consider the relationship between a car and an engine. The benefits of separating the engine from the car are evident. By building the engine separately, we can use the engine in various cars—not to mention other advantages. But we can't say that an engine *is-a* car. This just doesn't sound right when it rolls off the tongue (and because we are modeling real-world systems, this is the effect we want). Rather, we use the term *has-a* to describe composition relationships. A car *has-a(n)* engine.

Has-a Relationships

Although an inheritance relationship is considered an *is-a* relationship for reasons already discussed, a composition relationship is termed a *has-a relationship*. Using the example in the previous section, a television *has-a* tuner and *has-a* video display. A television is obviously not a tuner, so there is no inheritance relationship. In the same vein, a computer *has*-a video card, *has-a* keyboard, and *has-a* disk drive. The topics of inheritance, composition, and how they relate to each other are covered in great detail in Chapter 7, "Mastering Inheritance and Composition."

Conclusion

There is a lot to cover when discussing OO technologies. However, you should leave this chapter with a good understanding of the following topics:

- **Encapsulation**—Encapsulating the data and behavior into a single object is of primary importance in OO development. A single object contains both its data and behaviors and can hide what it wants from other objects.
- **Inheritance**—A class can inherit from another class and take advantage of the attributes and methods defined by the superclass.
- **Polymorphism**—Polymorphism means that similar objects can respond to the same message in different ways. For example, you might have a system with many shapes.

However, a circle, a square, and a star are each drawn differently. Using polymorphism, you can send each of these shapes the same message (for example, `Draw`), and each shape is responsible for drawing itself.

- **Composition**—Composition means that an object is built from other objects.

This chapter covers the fundamental OO concepts, of which by now you should have a good grasp.

Example Code Used in This Chapter

The following code is presented in C# .NET. Code for other languages, such as VB .NET and Objective-C, are available electronically on the publisher's web site. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestPerson` Example: C# .NET

```
using System;

namespace ConsoleApplication1
{
    class TestPerson
    {
        static void Main(string[] args)
        {

            Person joe = new Person();

            joe.Name = "joe";

            Console.WriteLine(joe.Name);

            Console.ReadLine();
        }
    }

    public class Person
    {

        //Attributes
        private String strName;
        private String strAddress;

        //Methods
        public String Name
```

```
        {
            get { return strName; }
            set { strName = value; }
        }

        public String Address
        {
            get { return strAddress; }
            set { strAddress = value; }
        }

    }
}
```

The `TestShape` Example: C# .NET

```
using System;

namespace TestShape
{
    class TestShape
    {
        public static void Main()
        {

            Circle circle = new Circle(5);
            Console.WriteLine(circle.calcArea());

            Rectangle rectangle = new Rectangle(4, 5);
            Console.WriteLine(rectangle.calcArea());

            Console.ReadLine();

        }
    }
    public abstract class Shape
    {

        protected double area;

        public abstract double calcArea();

    }
    public class Circle : Shape
    {
```

```
        private double radius;

        public Circle(double r)
        {

            radius = r;

        }

        public override double calcArea()
        {

            area = 3.14 * (radius * radius);
            return (area);

        }
    }
    public class Rectangle : Shape
    {

        private double length;
        private double width;

        public Rectangle(double l, double w)
        {
            length = l;
            width = w;
        }

        public override double calcArea()
        {
            area = length * width;
            return (area);
        }
    }
}
```