CHAPTER 3

Objects and Classes

OBJECTS ARE THE FUNDAMENTAL building blocks of an object-oriented (OO) system. Just as you learned in Chapter 2 that abstraction involves producing a model of the real world, you'll see in this chapter that objects are "mini abstractions" of the various real-world components that comprise such a model.

In this chapter, you'll learn

- What makes up a software object
- How we use classes to specify an object's data and behavior
- How we create objects based on a class definition
- How objects keep track of one another

What Is an Object?

Before we talk about software objects, let's talk about real-world objects in general. According to Merriam-Webster's Collegiate Dictionary, an object is

> *(1) Something material that may be perceived by the senses; (2) something mental or physical toward which thought, feeling, or action is directed.*

The first part of this definition refers to objects as we typically think of them: as physical "things" that we can see and touch, and which occupy space. Because we intend to use the Student Registration System (SRS) case study as the basis for learning about objects throughout this book, let's think of some examples of **physical objects** that make sense in the general context of an academic setting, namely

- The ***students*** who attend classes
- The ***professors*** who teach them
- The ***classrooms*** in which class meetings take place

- The ***furniture*** in these classrooms
- The ***buildings*** in which the classrooms are located
- The ***textbooks*** students use

and on and on. Of course, while all of these types of objects are commonly found on a typical college campus, not all of them are relevant to registering students for courses, nor are they all necessarily called out by the SRS case study, but we won't worry about that for the time being. In Part Two of this book, you'll learn a technique for using a requirements specification as the basis for identifying which types of objects are relevant to a particular abstraction.

Now, let's focus on the second half of the definition, particularly on the phrase ***"something mental . . . toward which thought, feeling, or action is directed."*** There are a great many **conceptual objects** that play important roles in an academic setting; some of these are

- The ***courses*** that students attend
- The ***departments*** that faculty work for
- The ***degrees*** that students receive

and, of course, many others. Even though we can't see, hear, touch, taste, or smell them, conceptual objects are every bit as important as physical objects are in describing an abstraction.

Let's now get a bit more formal, and define a **software object:**

- A (**software**) **object** is a software construct that bundles together **state** (**data**) and **behavior** (**operations**) that, taken together, represent an abstraction of a "real-world" (physical or conceptual) object.

Let's explore the two sides of objects—their **state** and **behavior**—separately, in more depth.

State/Attributes/Data

If we wish to record information about a student, what data might we require? Some examples might be

- The student's name
- His or her student ID number

- The student's birthdate
- His or her address
- The student's designated major field of study, if the student has declared one yet
- His or her cumulative grade point average (GPA)
- Who the student's faculty advisor is
- A list of the courses that the student is currently enrolled in this semester (i.e., the student's current course load)
- A history of all of the courses that the student has taken to date, the semester/year in which each was taken, and the grade that was earned for each: in other words, the student's transcript

and so on. Now, how about for an academic course? Perhaps we'd wish to record

- The course number (e.g., "ART 101")
- The course name (e.g., "Introductory Basketweaving")
- A list of all of the courses that must have been successfully completed by a student prior to allowing that student to register for ***this*** course (i.e., the course's prerequisites)
- The number of credit hours that the course is worth
- A list of the professors who have been approved to teach this course

and so on. In object nomenclature, the data elements used to describe an object are referred to as the object's **attributes.**

Use of the term "attribute" in this fashion is a language-neutral object modeling and programming convention. But, all .NET languages (including C#) have a specific **programming construct** *called an attribute, which has a more complex purpose than simply referring to a data element of an object. It's important not to confuse the two uses of the term "attribute" when talking specifically about one of the .NET languages. (.NET languages instead prefer to use the term "field" to refer to an object's data elements/attributes in the generic sense of the word.)*

We'll explain what an attribute in the C# (.NET) specific sense is all about in Chapter 13. For the time being, however (i.e., throughout the remainder of Parts One and Two of the book), whenever we use the term "attribute," we're using it in the generic OO sense.

An object's attribute values, when taken collectively, are said to define the **state,** or condition, of the object. For example, if we wanted to determine whether or not a student is "eligible to graduate" (a state), we might look at a combination of

- The student's transcript (an attribute), and
- The list of courses he or she is currently enrolled in (a second attribute)

to see if the student is indeed expected to have satisfied the course requirements for their chosen major field of study (a third attribute) by the end of the current academic year.

A given attribute may be simple—for example, "GPA", which can be represented as a simple floating point number—or complex—for example, "transcript", which represents a rather extensive collection of information with no simple representation (at least as far as C# simple types are concerned).

Behavior/Operations/Methods

Now, let's revisit the same two types of object—a student and a course—and talk about these objects' respective behaviors. A student's behaviors (relevant to academic matters, that is!) might include

- Enrolling in a course
- Dropping a course
- Choosing a major field of study
- Selecting a faculty advisor

- Telling you his or her GPA
- Telling you whether or not he or she has taken a particular course, and if so, when the course was taken, which professor taught it, and what grade the student received

It's a bit harder to think of an inanimate, conceptual object like a course as having behaviors, but if we were to imagine a course to be a living thing, then we can imagine that a course's behaviors might include

- Permitting a student to register
- Determining whether or not a given student is ***already*** registered
- Telling you how many students have registered so far, or conversely, how many seats remain before the course is full
- Telling you what its prerequisite courses are
- Telling you how many credit hours it's worth
- Telling you which professor is assigned to teach the course this semester

and so on.

When we talk about software objects specifically, we define an object's behaviors, also known as its **operations,** as both the things that an object does to **access** its attributes (data), and the things that an object does to **modify/maintain** its attribute values (data).

If we take a moment to reflect back on the behaviors we expect of a student as listed previously, we see that each operation involves one or more of the student's attributes. For example:

- Telling you his or her GPA involves ***accessing*** the value of the student's "GPA" attribute.
- Choosing a major field of study involves ***modifying*** the value of the student's "major" attribute.
- Enrolling in a course involves ***modifying*** the value of the student's "course load" attribute.

Since we recently learned that the collective set of attribute values for an object defines its state, we now can see that operations are capable of ***changing an object's state.*** Let's say that we define the state of a student who hasn't yet selected a major field of study as an "undeclared" student. Asking such a student

object to perform its "choosing a major field of study" method will cause that object to update the value of its "major field of study" attribute to reflect the newly selected major field. This, then, changes the student's state from "undeclared" to "declared".

Yet another way to think of an object's operations are as **services** that can be requested of the object. For example, one service that we might call upon a course object to perform is to provide us with a list of all of the students who are currently registered for the course (i.e., a student roster).

When we actually get around to programming an object in a language like C#, we refer to the programming language representation of an operation as a **method,** whereas, strictly speaking, the term "operation" is typically used to refer to a behavior conceptually.

Classes

A **class** is an abstraction describing the common features of all objects in a group of similar objects. For example, a class called "Student" could be created to describe all student objects recognized by the Student Registration System.

A class defines the following:

- The data structure (names and types of attributes) needed to define an object belonging to that class
- The operations to be performed by such objects: specifically, what these operations are, how an object belonging to that class is formally called upon to perform them, and what "behind the scenes" things an object has to do to actually carry them out

For example, the `Student` class might be defined to have the nine attributes described in Table 3-1.

Table 3-1. Proposed Attributes of the Student Class

Attribute	Type
`name`	`string`
`studentId`	`string`
`birthdate`	`DateTime`
`address`	`string`

Table 3-1. Proposed Attributes of the Student Class (continued)

Attribute	Type
major	string
gpa	double
advisor	???
courseLoad	???
transcript	???

This means each and every `Student` object will have these ***same*** nine attributes. Note that many of the attributes can be represented by predefined C# types (e.g., `string`, `double`, and `DateTime`) but that a few of the attributes—`advisor`, `courseLoad`, and `transcript`—are too complex for predefined types to handle; you'll learn how to tackle such attributes a bit later on.

In terms of operations, the `Student` class might define five methods as follows:

- `RegisterForCourse`
- `DropCourse`
- `ChooseMajor`
- `ChangeAdvisor`
- `PrintTranscript`

Note that an object can only do those things for which methods have been defined by the object's class. In that respect, an object is like an appliance: it can do whatever it was designed to do (a DVD player provides buttons to play, pause, stop, and seek a particular movie scene), and nothing more (you can't ask a DVD to toast a bagel—at least not with much chance of success!). So, an important aspect of successfully designing an object is making sure to anticipate all of the behaviors it will need to be able to perform in order to carry out its "mission" within the system. We'll see how to determine what an object's mission, data structure, and behaviors should be, based on the requirements for a system, in Part Two of the book.

*The terms **feature** and **member** are used interchangeably to refer to both attributes and methods of a class. That is, a class definition that includes three attribute declarations and five method declarations is said to have eight features/members. "Feature" is the generic OO term, "member" the C#-specific term. We'll use generic OO terminology in Parts One and Two of the book, switching to C#-specific terminology in Part Three.*

Features are the building blocks of a class: virtually everything found within a class definition is either an attribute or a method of the class.

In the C# language, several other types of things are included within the boundaries—i.e., are features—of a class definition, but we won't worry about these until we get to Part Three of the book. Conceptually, it's sufficient to think of an object as consisting only of attributes and methods at this point in time.

A Note Regarding Naming Conventions

It's recommended practice to name classes starting with an ***uppercase*** letter, but to use mixed case for the name overall: `Student`, `Course`, `Professor`, and so on. When the name of a class would ideally be stated as a multiword phrase, such as "course catalog", start each word with a capital letter, and concatenate the words without using spaces, dashes, or underscores to separate them: for example, `CourseCatalog`. This style is known as **Pascal casing**.

For method names, the C# convention is to use Pascal casing as well. Typical method names might thus be `Main`, `GetName`, or `RegisterForCourse`.

In contrast, the C# convention for attribute names is to start with a ***lowercase*** letter, but to capitalize the first letter of any subsequent words in the attribute name. Typical attribute names might thus be `name`, `studentId`, or `courseLoad`. This style is known as **Camel casing.**

In subsequent chapters, we'll refine the rules for how Pascal and Camel casing are to be applied.

Instantiation

A class definition may be thought of as a template for creating software objects—a "pattern" used to

- Stamp out a prescribed data area in memory to house the attributes of a new object.

- Associate a certain set of behaviors with that object.

The term **instantiation** is used to refer to the process by which an object is created (constructed) based upon a class definition. From a single class definition—for example, `Student`—we can create many objects, in the same way that we use a single cookie cutter to make many cookies. Another way to refer to an object, then, is as an **instance** of a particular class—e.g., a `Student` object is an instance of the `Student` class. We'll talk about the physical process of **instantiating** objects as it occurs in C# in a bit more detail later in this chapter.

Classes may be differentiated from objects, then, as follows:

- A ***class*** defines the features—attributes, methods, etc.—that all objects belonging to the class possess, and can be thought of as serving as an object ***template***, as illustrated in Figure 3-1.

- An ***object***, on the other hand, is a unique instance of a ***filled-in*** template for which attribute values have been provided, and on which methods may be called, as illustrated in Figure 3-2.

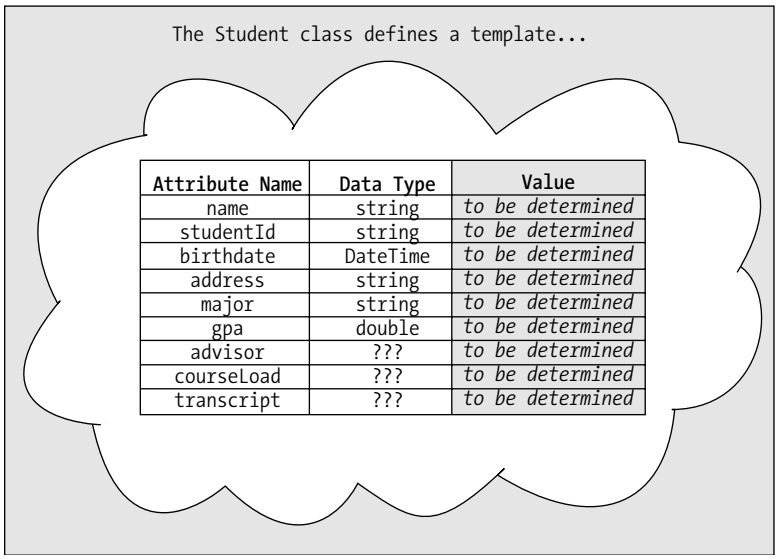

Figure 3-1. A class defines attribute names and types.

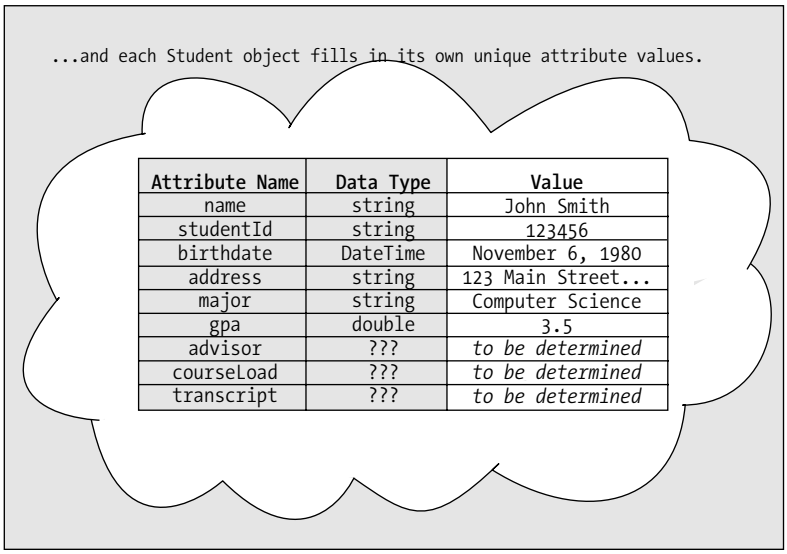

Figure 3-2. An object provides attribute values.

Encapsulation

Encapsulation is a formal term referring to the mechanism that bundles together the state and behavior of an object into a single logical unit. Everything that we need to know about a given student is, in theory, contained within the "walls" of the student object, either directly as a field of that object or indirectly as a method that can answer a question or make a determination about the object's state.

Encapsulation isn't unique to OO languages, but in some senses is perfected by them. For those of you familiar with C, you know that a C `struct(ure)` *encapsulates data:*

```
struct employee {
    char name[30];
    int age;
}
```

and a C function encapsulates logic—data is passed in, operated on, and an answer is optionally returned:

```
float average(float x, float y) {
    return (x + y)/2.0;
}
```

But only with OO programming languages is the notion of encapsulating data and behavior in a single construct, to represent an abstraction of a real-world entity, truly embraced.

User-Defined Types and Reference Variables

In a non-OO programming language such as C, the statement

```
int x;
```

is a **declaration** that variable x is an int(eger), one of several simple, **predefined types** defined to be part of the C language.

What does this ***really*** mean? It means that

- x is a symbolic name that represents an integer value.
- The "thing" that we've named x understands how to respond to a number of different operations, such as addition (+), subtraction (–), multiplication (*), division (/), logical comparisons (>, <, =), and so on that have been defined for the int type.
- Whenever we want to operate on this particular integer value in our program, we refer to it via its symbolic name x:

```
if (x > 17) x = x + 5;
```

In an object-oriented language like C#, we can define a class such as Student, and then declare a variable as follows:

```
Student y;
```

What does this mean? It means that

- y is a symbolic name that refers to a Student object (an instance of the Student class).
- The "thing" that we have named y understands how to respond to a number of different service requests—how to register for a course, drop a course, and so on—that have been defined by the Student class.
- Whenever we want to operate on this particular object, we refer to y:

```
// Pseudocode.
  if (y hasn't chosen an advisor yet) Console.WriteLine ("Uh oh ...");
```

Note the parallels between y as a Student in the preceding example and x as an int earlier. Just as int is a predefined type (in both C and C#), the Student

class is a **user-defined type**. And, because y in the preceding example is a variable that ***refers to*** an instance (object) of class Student, y is informally known as a **reference variable.**

Names for (nonattribute) reference variables follow the same convention as method and attribute names: i.e., they use Pascal casing. Some sample reference variable declarations are as follows:

```
Student x;
Student aStudent;
Course prerequisiteOfThisCourse;
Professor myAdvisor;
```

Instantiating Objects: A Closer Look

Different OO languages differ in terms of when an object is actually instantiated (created). In C#, when we declare a variable to be of a user-defined type, like

```
Student y;
```

we haven't actually created an object in memory yet. Rather, we've simply declared a reference variable of type Student named y. This reference variable has the ***potential*** to refer to a Student object, but it doesn't refer to one just yet; rather, it's said to have the value null, which as we saw in Chapter 1 is the C# keyword used to represent a nonexistent object.

We have to take the distinct step of using a special C# operator, the new operator, to actually carve out a brand-new Student object in memory, and to then associate the new object with the reference variable y, as follows:

```
y = new Student();
```

> *Behind the scenes, what we're actually doing is associating the value of the* ***physical memory address*** *at which this object was created—known as a* ***reference****—to variable* y.
>
> *Don't worry about the parentheses at the end of the preceding statement; we'll talk about their significance in Chapter 4, when we discuss the notion of* ***constructors.***

Think of the newly created object as a helium balloon, as shown in Figure 3-3, and a reference variable as the hand that holds a string tied to the balloon so that we may access the object whenever we'd like.

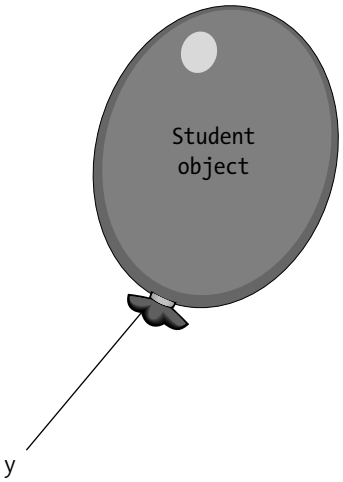

Figure 3-3. Using a reference variable to keep track of an object in memory

Because a reference variable is sometimes informally said to "hold onto" an object, we often use the informal term **handle,** as in the expression "reference variable `y` maintains a handle on a `Student` object."

We can also create a new object without immediately assigning it to a reference variable, as in the following line of code:

```
new Student();
```

but such an object would be like a helium balloon without a string tied to it: it would indeed exist, but we'd never be able to access this object in our program. It would, in essence, "float away" from us in memory.

Note that we can combine the two steps—declaring a reference variable and actually instantiating an object for that variable to refer to—into a single line of code:

```
Student y = new Student();
```

Another way to initialize a reference variable is to hand it a ***preexisting*** object: that is, an object ("helium balloon") that is already being referenced ("held onto") by a ***different*** reference variable ("hand"). Let's look at an example:

```
// We declare a reference variable, and instantiate our first Student object.
Student x = new Student();

// We declare a second reference variable, but do not instantiate a
// second object.
Student y;
```

```
// We pass y a "handle" on the same object that x is holding onto
// (x continues to hold onto it, too).  We now, in essence,
// have two "strings" tied to the same "balloon".
y = x;
```

The conceptual outcome of the preceding code is illustrated in Figure 3-4: two "strings", being held by two different "hands", tied to the same "balloon"—that is, two ***different*** reference variables referring to the ***same*** physical object in memory.

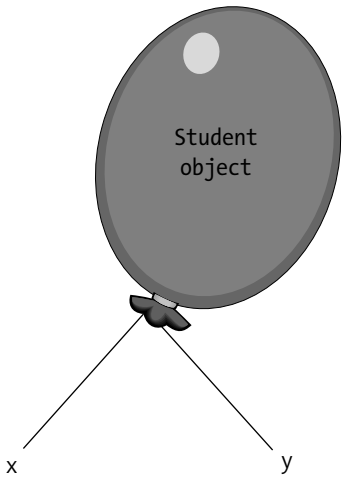

Figure 3-4. Maintaining multiple handles on the same object

We therefore see that the same object can have many reference variables simultaneously referring to it; but, as it turns out, any ***one*** reference variable can only hold onto/refer to ***one*** object at a time. To grab onto a new object handle means that a reference variable must let go of the object handle that it was previously holding onto, if any.

If there comes a time when ***all*** handles for a particular object have been let go of, then as we discussed earlier the object is no longer accessible to our program, like a helium balloon that has been let loose. Continuing with our previous example (note highlighted code below and Figures 3-5, 3-6, and 3-7):

```
// We instantiate our first Student object.
Student x = new Student();

// We declare a second reference variable, but do not instantiate a
// second object.
Student y;
```

```
// We pass y a "handle" on the same object that x is holding onto
// (x continues to hold onto it, too).  We now, in essence,
// have TWO "strings" tied to the same "balloon".
y = x;

// We now declare a third reference variable and instantiate a second
// Student object.
Student z = new Student();
```

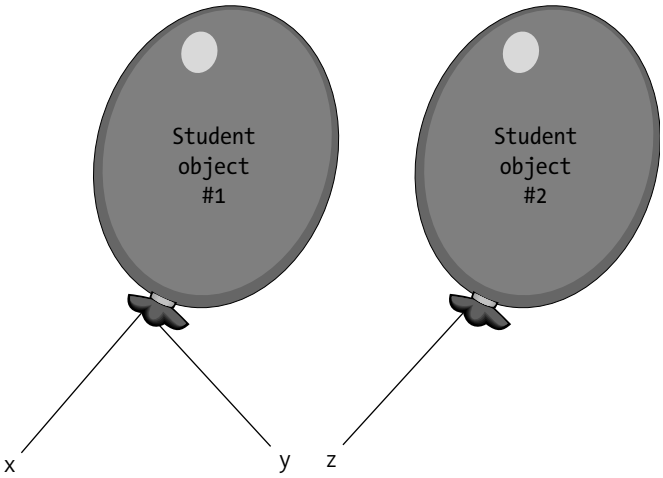

Figure 3-5. A second object comes into existence.

```
// y now lets go of the first Student object and grabs onto the second.
y = z;
```

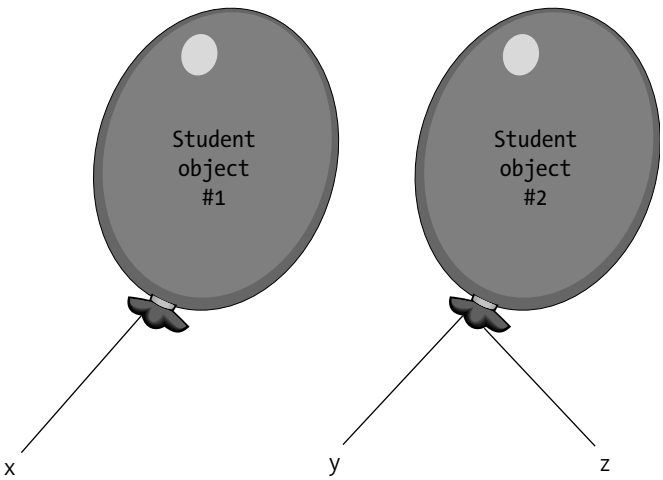

Figure 3-6. Transferring object handles

```
// Finally, x lets go of the first Student object, and grabs onto
// the second, as well; the first Student object is now lost to
// the program because we no longer have any reference variables
// maintaining a "handle" on it!
x = z;
```

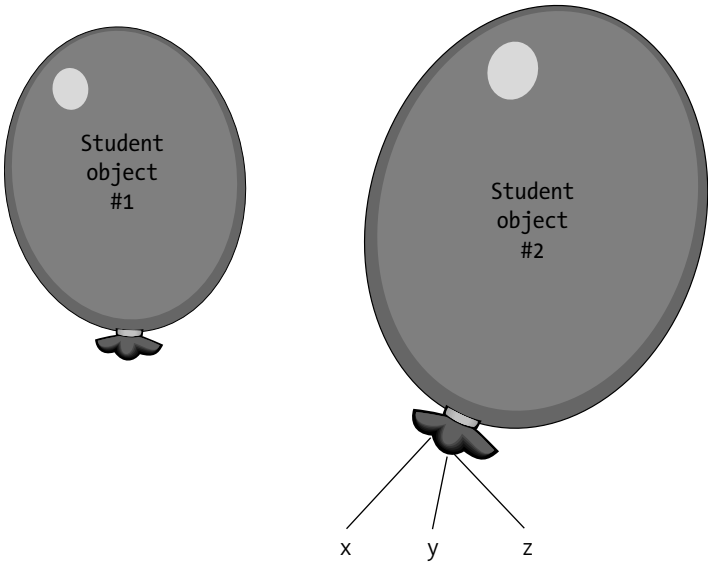

Figure 3-7. The first object is now lost to our program.

As it turns out, if all of an object's handles are lost, it might seem as though the memory that the object occupies is permanently wasted. (In a language like C++, this is indeed the case, and programmers have to explicitly take care to "reclaim" the memory of an object that is no longer needed before all of its handles are dropped. Failure to do so is a chronic source of problems in C++ programs.) In C# (and all other .NET languages) the **common language runtime (CLR)** periodically performs **garbage collection,** a process that automatically reclaims the memory of "lost" objects for us. We'll revisit this topic in Chapter 13.

Objects As Attributes

When we first discussed the attributes and methods associated with the `Student` class, we stated that some of the attributes could be represented by predefined types provided by the C# language, whereas the types of a few others (`advisor`, `courseLoad`, and `transcript`) were left undefined. Let's now put what we've learned about user-defined types to good use.

Rather than declaring the `Student` class's `advisor` attribute as simply a `string` representing the advisor's name, we'll declare it to be of user-defined type—namely, type `Professor`, another class that we've invented (see Table 3-2).

Table 3-2. Student Class Attributes, Revisited

Attribute	Type
name	string
studentID	string
birthdate	DateTime
address	string
major	string
gpa	double
advisor	**Professor**
courseLoad	???
transcript	???

By having declared the `advisor` attribute to be of type `Professor`—i.e., by making the `advisor` attribute a reference variable—we've just enabled a `Student` object to maintain a handle on the actual `Professor` object that is advising the student. We'll still leave the `courseLoad` and `transcript` types unspecified for the time being; we'll see how to handle these a bit later.

The `Professor` class, in turn, might be defined to have attributes as listed in Table 3-3.

Table 3-3. Student Class Attributes

Attribute	Type
name	string
employeeID	string
birthdate	DateTime
address	string
worksFor	string (or Department)
studentAdvisee	**Student**
teachingAssignments	???

Again, by having declared the `studentAdvisee` attribute of `Professor` to be of type `Student`—i.e., by making the `studentAdvisee` attribute a reference variable—we've just given a `Professor` object a way to hold onto/refer to the actual `Student` object that the professor is advising. We'll leave the type of `teachingAssignments` undefined for the time being.

The methods of the `Professor` class might be as follows:

- `TransferToDepartment`
- `AdviseStudent`
- `AgreeToTeachCourse`
- `AssignGrades`

A few noteworthy points about the `Professor` class:

- It's likely that a professor will be advising several students simultaneously, so having an attribute like `studentAdvisee` that can only reference a single `Student` object is not terribly useful. We'll discuss techniques for handling this in Chapter 6, when we talk about **collections,** which we'll also see as being useful for defining the `teachingAssignments` attribute of `Professor` and the `courseLoad` and `transcript` attributes of `Student`.

- The `worksFor` attribute represents the department to which a professor is assigned. We can choose to represent this as either a simple `string` representing the department name—for example, "MATH"—or as a reference variable that maintains a handle on a `Department` object—specifically, the `Department` object representing the "real-world" Math Department. Of course, to do so would require us to define the attributes and methods for a new class called `Department`. As we'll see in Part Two of this book, the decision of whether or not we need to invent a new user-defined type/class to represent a particular real-world concept/abstraction isn't always clear-cut.

Composition

Whenever we create a class, such as `Student` or `Professor`, in which one or more of the attributes are themselves handles on other objects, we are employing an OO technique known as **composition**. The number of levels to which objects can be conceptually bundled inside one another is endless, and so composition enables us to model very sophisticated real-world concepts. As it turns out, most "interesting" classes employ composition.

With composition, it may seem as though we're nesting objects one inside the other, as depicted in Figure 3-8.

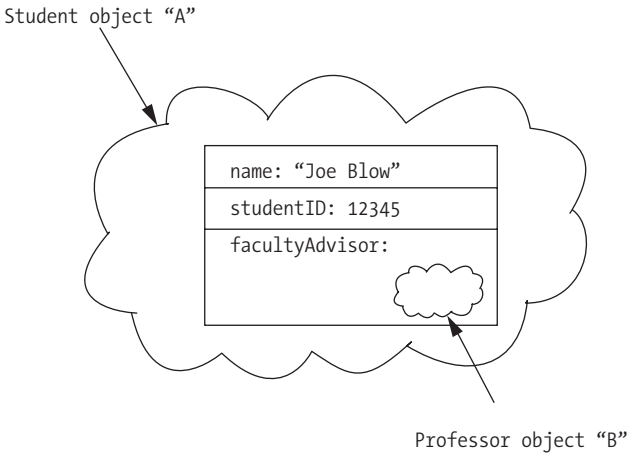

Figure 3-8. Conceptual object "nesting"

Actual object nesting (i.e., declaring one class inside of another) is possible in some OO programming languages, and does indeed sometimes make sense: namely, if an object **A** doesn't need to have a life of its own from the standpoint of an OO application, and only exists for the purpose of serving enclosing object **B.**

- Think of your brain, for example, as an object that exists only within the context of your body (another object).

- As an example of object nesting relevant to the SRS, let's consider a grade book used to track student performance in a particular course. If we were to define a `GradeBook` class, and then create `GradeBook` objects as attributes—one per `Course` object—then it might be reasonable for each `GradeBook` object to exist wholly within the context of its associated `Course` object. No other objects would need to communicate with the `GradeBook` directly; if a `Student` object wished to ask a `Course` object what grade the `Student` has earned, the `Course` object might internally consult its embedded `GradeBook` object, and simply hand a letter grade back to the `Student`.

However, we often encounter the situation—as with the sample `Student` and `Professor` classes—in which an object **A** needs to refer to an object **B**, object **B** needs to refer back to **A**, and ***both*** objects need to be able to respond to requests independently of each other as made by the application as a whole. In such a case, handles come to the rescue! In reality, we are ***not*** storing whole objects as attributes inside of other objects; rather, we are storing ***references*** to objects. When an attribute of an object **A** is defined in terms of an object reference **B,** the two objects exist separately in memory, and simply have a convenient way of

finding one another whenever it's necessary for them to interact. Think of yourself as an object, and your cellular phone number as your reference. Other people—"objects"—can reach you to speak with you whenever they need to, even though they don't know where you're physically located, using your cell phone number.

Memory allocation using handles might look something like Figure 3-9 conceptually.

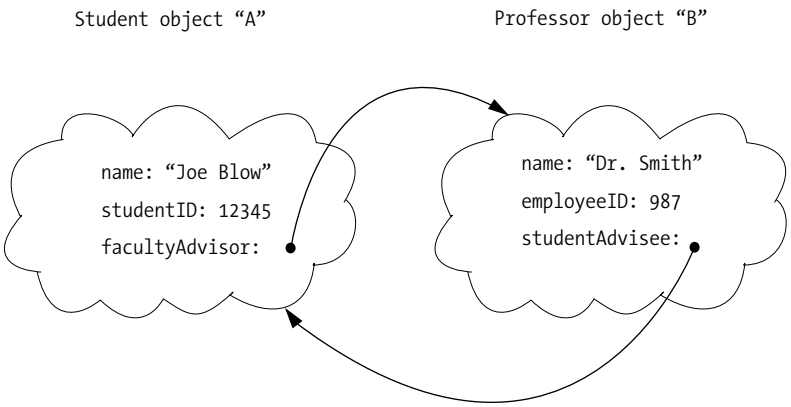

Figure 3-9. Objects exist separately in memory and maintain handles on one another.

With this approach, each object is allocated in memory only once; the `Student` object knows how to find and communicate with its advisor (`Professor`) object whenever it needs to through its handle/reference, and vice versa.

What do we gain by defining the `Student`'s `advisor` attribute as a reference to a `Professor` object, instead of merely storing the name of the advisor as a `string` attribute of the `Student` object?

For one thing, we can ask the `Professor` object its name whenever we need it (through a technique that we'll discuss in Chapter 4). Why is this important? ***To avoid data redundancy and the potential for loss of data integrity.***

- If the `Professor` object's name changes for some reason, the name will only be stored in one place: encapsulated as an attribute within the `Professor` object that "owns" the name, which is precisely where it belongs.

- If we instead were to redundantly store the `Professor`'s name both as a `string` attribute of the `Professor` object and as a `string` attribute of the `Student` object, we'd have to remember to update the name in two places any time the name changed (or three, or four, or however many places this `Professor`'s name is referenced as an advisor of countless `Students`). If we were to forget to do so, then the name of the `Professor` would be "out of synch" from one instance to another.

Just as importantly, by maintaining a handle on the `Professor` object via the `advisor` attribute of `Student`, the `Student` object can also ***request other services*** of this `Professor` object via whatever methods are defined for the `Professor` class. A `Student` object may, for example, ask its advisor (`Professor`) object where the `Professor`'s office is located, or what classes the `Professor` is teaching so that the `Student` can sign up for one of them.

Another advantage of using object handles from an implementation standpoint is that they also reduce memory overhead. Storing a reference to (aka memory address of) an object only requires 4 bytes (on 32-bit machines) or 8 bytes (on 64-bit machines) of memory, instead of however many bytes of storage the referenced object as a whole occupies in memory. If we were to have to make a copy of an entire object every place we needed to refer to it in our application, we could quickly exhaust the total memory available to our application.

Three Distinguishing Features of an Object-Oriented Programming Language

In order to be considered truly object oriented, a programming language must provide support for three key mechanisms:

- (Programmer creation of) User-defined types
- Inheritance
- Polymorphism

We've just learned about the first of these mechanisms, and will discuss the other two in chapters to follow.

Summary

In this chapter, you've learned that

- An object is a software abstraction of a physical or conceptual real-world object.
- A class serves as a template for defining objects: specifically, a class defines the following:
  - What data the object will house, known as an object's attributes
  - What behaviors an object will be able to perform, known as an object's operations (methods)

- An object may then be thought of as a filled-in template.
- Just as we can declare variables to be of simple predefined types such as `int`, `double`, and `bool`, we can also declare variables to be of user-defined types such as `Student` and `Professor`.
- When we create a new object (a process known as instantiation), we typically store a reference to that object in a reference variable. We can then use that "handle" to communicate with the object.
- We can define attributes of a class A to serve as handles on objects belonging to another class B. In doing so, we allow each object to encapsulate the information that rightfully belongs to that object, but enable objects to share information by contacting one another whenever necessary.

Exercises

1. From the perspective of an academic setting (but not necessarily the SRS case study specifically), think about what the appropriate attributes and methods of the following classes might be:

   - `Classroom`
   - `Department`
   - `Degree`

   Which of the attributes of each of these classes should be declared using predefined C# types, and which should be declared using user-defined types? Explain your rationale.

2. For the problem area whose requirements you defined for exercise 3 in Chapter 2, list the classes that you might need to create in order to model it properly.

3. List the classes that you might need to create in order to model the Prescription Tracking System discussed in Appendix B.

4. Would `Color` be a good candidate for a user-defined type/class? Why or why not?