

Using Classes and Objects

In this chapter you will:

- ◎ Learn about class concepts
- ◎ Create classes from which objects can be instantiated
- ◎ Create objects
- ◎ Create properties, including auto-implemented properties
- ◎ Learn more about using `public` and `private` access modifiers
- ◎ Learn about the `this` reference
- ◎ Write and use constructors
- ◎ Use object initializers
- ◎ Overload operators
- ◎ Declare object arrays and use methods with them
- ◎ Write destructors
- ◎ Understand GUI application objects

Much of your understanding of the world comes from your ability to categorize objects and events into classes. As a young child, you learned the concept of *animal* long before you knew the word. Your first encounter with an animal might have been with the family dog, a neighbor's cat, or a goat at a petting zoo. As you developed speech, you might have used the same term for all of these creatures, gleefully shouting, "Doggie!" as your parents pointed out cows, horses, and sheep in picture books or along the roadside on drives in the country. As you grew more sophisticated, you learned to distinguish dogs from cows; still later, you learned to distinguish breeds. Your understanding of the class *animal* helps you see the similarities between dogs and cows, and your understanding of the class *dog* helps you see the similarities between a Great Dane and a Chihuahua. Understanding classes gives you a framework for categorizing new experiences. You might not know the term *okapi*, but when you learn it's an animal rather than a food or piece of clothing, you begin to develop a concept of what an okapi might be like.

Classes are also the basic building blocks of object-oriented programming. You already understand that differences exist among the `Double`, `Int32`, and `Float` classes, yet you also know that items belonging to these classes possess similarities—they are all data types, you can perform arithmetic with all of them, they all can be converted to strings, and so on. Understanding classes enables you to see similarities in objects and increases your understanding of the programming process. In this chapter, you will discover how `C#` handles classes, learn to create your own classes, and learn to construct objects that are members of those classes.

Understanding Class Concepts

When you write programs in `C#`, you create two types of classes:

- *Classes that are only application programs with a `Main()` method.* These classes can contain other methods that the `Main()` method calls. You have been creating these classes throughout this book.
- *Classes from which you instantiate objects.* To **instantiate** an object is to create a tangible example. An object is an **instantiation** of a class or an **instance** of a class. A class that is created to be a model for object instantiations can contain a `Main()` method, but it is not required. This chapter explores the nuances of such classes.

When you think in an object-oriented manner, things are objects, and every object is an instance or example of a class. You can think of any inanimate physical item as an object—your desk, your computer, and your house are all called *objects* in everyday conversation. You can think of living things as objects, too—your houseplant, your pet fish, and your sister are objects. Events also are objects—for example, the stock purchase you made, the mortgage closing you attended, or a graduation party in your honor.

Every object is an instance of a more general class. Your desk is a specific instance of the class that includes all desks, and your pet fish is an instance of the class that contains all fish. In part, the concept of a class is useful because it provides you with knowledge about objects.

Objects receive their attributes from classes; so, for example, if you invite me to a graduation party, I automatically know many things about the object. I assume that there will be a starting time, a number of guests, and some refreshments. I understand your party because of my previous knowledge of the `Party` class. I don't necessarily know the number of guests or the date or time of this particular party, but I understand that because all parties have a date and time, then this one must as well. Similarly, even though every stock purchase is unique, each must have a dollar amount and a number of shares. All objects have predictable attributes because they are instances of certain classes.

The data components of a class that differ for each object are stored in **instance variables**. Instance variables also are called **fields** to help distinguish them from other variables you might use. In Chapter 1, you learned that the data field values of an object are also called its *attributes* and that the set of attributes is known as an object's *state*. For example, the current state of a particular party is 8 p.m. and Friday; the state of a particular stock purchase is \$10 and five shares.

In addition to their attributes, objects have methods associated with them, and every object that is an instance of the same class possesses the same methods. Methods associated with objects are **instance methods**. For example, two `Party` objects might possess the identifiers `myGraduationParty` and `yourAnniversaryParty` that both have access to an `IssueInvitations()` method. The method might operate in the same way for both `Party` objects, but use data appropriate to the specific object. When you program in C#, you frequently create classes from which objects will be instantiated (or other programmers create them for you). You also write applications to use the objects, along with their data and methods. Often, you will write programs that use classes created by others; similarly, you might create a class that other programmers will use to instantiate objects within their own programs. A program or class that instantiates objects of another prewritten class is a **class client** or **class user**.

TWO TRUTHS & A LIE

Understanding Class Concepts

1. C# classes always contain a `Main()` method.
2. An object is an instantiation of a class.
3. The data components of a class often are its instance variables.

The false statement is #1. C# applications always contain a `Main()` method, but some classes do not if they are not meant to be run as programs.

Creating a Class from Which Objects Can Be Instantiated

When you create a class, you must assign a name to it and determine what data and methods will be part of the class. For example, suppose that you decide to create a class named `Employee`. One instance variable of `Employee` might be an employee number, and one necessary method might display a welcome message to new employees. To begin, you create a **class header** or **class definition** that starts the class. It contains three parts:

1. An optional access modifier
2. The keyword `class`
3. Any legal identifier you choose for the name of your class; because each class represents a type of object, class names are usually singular nouns.

You will learn about other optional components that can be added to a class definition as you continue to study *C#*.

For example, one usable header for an `Employee` class is `internal class Employee`. The keyword `internal` is an example of a **class access modifier**. You can declare a class using any of the modifiers in Table 9-1.

Class Access Modifier	Description
<code>public</code>	Access to the class is not limited.
<code>protected</code>	Access to the class is limited to the class and to any classes derived from the class. (You will learn about deriving classes in the chapter <i>Introduction to Inheritance</i> .)
<code>internal</code>	Access is limited to the assembly to which the class belongs. (An assembly is a group of code modules compiled together to create an executable program. The <code>.exe</code> files you create after compiling a <i>C#</i> program are assemblies.)
<code>private</code>	Access is limited to another class to which the class belongs. In other words, a class should be <code>private</code> if it is contained within another class, and the containing class is the only one that should have access to it.

Table 9-1 Class access modifiers

If you do not explicitly include an access specifier, class access is `internal` by default. Because most classes you create will have `internal` access, typing an access specifier is often unnecessary. (When you declare a class using a namespace, you only can declare it to be `public` or `internal`.)

In addition to the class header, classes you create must have a class body enclosed between curly braces. Figure 9-1 shows a shell for an `Employee` class.

```
class Employee
{
    // Instance variables and methods go here
}
```

Figure 9-1 Employee class shell

Creating Instance Variables and Methods

When you create a class, you define both its attributes and its methods. You declare the class's instance variables within the curly braces using the same syntax you use to declare other variables—you provide a type and an identifier. For example, within the `Employee` class, you can declare an integer ID number, and when you create `Employee` objects, each will have its own `idNumber`. You can define the ID number simply as `int idNumber;`. However, programmers frequently include an access modifier for each field in a class and declare the `idNumber` as `private int idNumber;`; Figure 9-2 shows an `Employee` class that contains the private `idNumber` field.

```
class Employee
{
    private int idNumber;
}
```

Figure 9-2 Employee class containing `idNumber` field

The allowable field modifiers are `new`, `public`, `protected`, `internal`, `private`, `static`, `readonly`, and `volatile`. If you do not provide an access specifier for a field, its access is `private` and nonstatic by default, which means that no other class can access the field's values, and only nonstatic methods of the same class will be allowed to set, get, or otherwise use the private field directly. Using private fields within classes is an example of **information hiding**, a feature found in all object-oriented languages.

You see cases of information hiding in real-life objects every day. For instance, you cannot see into your automobile's gas tank to determine how full it is. Instead, you use a gauge on the dashboard to provide you with the necessary information. Similarly, data fields are frequently `private` in object-oriented programming, but their contents are accessed through `public` methods. The `private` data of a class should be changed or manipulated only by its own methods, not by methods that belong to other classes. A benefit of making data items `private` is the ability to control their values. A method that sets a variable's value can use decisions to ensure that the value falls within a specified range. For example, perhaps an `Employee`'s salary should not be below the federal minimum wage, or perhaps a department number should not be negative or greater than 10.

In contrast to a class's private data fields, most instance methods are not usually `private`; they are `public`. The resulting `private` data/`public` method arrangement provides a means to

control outside access to your data—only a class’s nonprivate methods can be used to access a class’s `private` data. The situation is similar to having a “public” receptionist who controls the messages passed in and out of your private office. The way in which the nonprivate methods are written controls how you will use the `private` data.

For example, an `Employee` class that contains an `idNumber` might need a method to display the employee’s welcoming message to company clients. A reasonable name for this method is `WelcomeMessage()`, and its declaration is `public void WelcomeMessage()`, because it will have `public` access and return nothing. Figure 9-3 shows the `Employee` class with the addition of the `WelcomeMessage()` method.

```
class Employee
{
    private int idNumber;
    public void WelcomeMessage()
    {
        WriteLine("Welcome from Employee #{0}", idNumber);
        WriteLine("How can I help you?");
    }
}
```

Figure 9-3 Employee class with `idNumber` field and `WelcomeMessage()` method

Notice that the `WelcomeMessage()` method does not employ the `static` modifier, unlike many other methods you have created. The keyword `static` is used for classwide methods but not for instance methods that “belong” to objects. If you are creating a program with a `Main()` method that you will execute to perform some task, then many of your methods will be `static`. You call `static` methods from `Main()` without creating an object. (If the method is in a different class, you use the class name, a dot, and the method call.) However, if you are creating a class from which objects will be instantiated, most methods will probably be nonstatic, as you will be associating the methods with individual objects and their data. (You use an object name, a dot, and the method call.) Each time the `WelcomeMessage()` instance method is used in the class in Figure 9-3, it will display an `idNumber` for a specific object. In other words, the method will work appropriately for each object instance, so it is not static.



In Chapter 7, you learned that when you create a `Form`, you sometimes include nonstatic variable declarations outside of any methods—for example, if two `Click()` methods must access the same data. Now you can understand that such variables are nonstatic because they are associated with an object—the `Form` that is being defined.

The `Employee` class in Figure 9-3 is not a program that will run; it contains no `Main()` method. Rather, it simply describes what `Employee` objects will have (an `idNumber`) and be able to do (display a greeting) when you write a program that contains one or more `Employee` objects.



A class can contain fields that are objects of other classes. For example, you might create a class named `Date` that contains a month, day, and year and add two `Date` fields to an `Employee` class to hold the `Employee`'s birth date and hire date. You might also create a `BusinessOffice` class and declare `Employee` objects within it. Using an object within another object is known as **composition**. The relationship created is also called a **has-a relationship** because one object “has an” instance of another.



Watch the video *Creating a Class*.

TWO TRUTHS & A LIE

Creating a Class from Which Objects Can Be Instantiated

1. A class header always contains the keyword `class`.
2. When you create a class, you define both its attributes and its methods.
3. Most fields and methods defined in a class are `private`.

The false statement is #3. Most fields in a class are `private`, but most methods are `public`.

Creating Objects

Declaring a class does not create any actual objects. A class is just an abstract description of what an object will be like if any objects are ever actually instantiated. Just as you might understand all the characteristics of an item you intend to manufacture long before the first item rolls off the assembly line, you can create a class with fields and methods before you instantiate any objects that are occurrences of that class. You can think of a class declaration as similar to a blueprint for building a new house or a recipe for baking a cake. In other words, it is a plan that exists before any objects are created.

Defining an object is similar to defining a simple variable that is a built-in data type; in each case, you use a data type and an identifier. For example, you might define an integer as `int someValue;` and you might define an `Employee` as `Employee myAssistant;`, where `myAssistant` could be any legal identifier you choose to represent an `Employee`.

Every object name is a reference—that is, it holds a computer memory location where the fields for the object reside. The memory location holds the values for the object.

When you declare an integer as `int myInteger;`, you notify the compiler that an integer named `myInteger` will exist, and computer memory automatically is reserved for it at the same time—the exact amount of computer memory depends on the declared data type. When you declare

the `myAssistant` instance of the `Employee` class, you are notifying the compiler that you will use the identifier `myAssistant`. However, you are not yet setting aside computer memory in which the `Employee` named `myAssistant` can be stored—that is done only for the built-in, predefined types. To allocate the needed memory for the object, you must use the `new` operator. You used the `new` operator earlier in this book when you learned to set aside memory for an array.

Defining an `Employee` object named `myAssistant` requires two steps—you must declare a reference to the object and then you must use the statement that actually sets aside enough memory to hold `myAssistant`:

```
Employee myAssistant;
myAssistant = new Employee();
```

You also can declare and reserve memory for `myAssistant` in one statement, as in the following:

```
Employee myAssistant = new Employee();
```

In this statement, `Employee` is the object's type (as well as its class), and `myAssistant` is the name of the object. The equal sign is the assignment operator, so a value is being assigned to `myAssistant`. The `new` operator is allocating a new, unused portion of computer memory for `myAssistant`. The value being assigned to `myAssistant` is a memory address at which the object created by the `Employee()` constructor will be located. You need not be concerned with the actual memory address—when you refer to `myAssistant`, the compiler will locate it at the appropriate address for you.



Because the identifiers for objects are references to their memory addresses, you can call any class a **reference type**—in other words, a type that refers to a specific memory location. A reference type is a type that holds an address, as opposed to the predefined types such as `int`, `double`, and `char`, which are **value types** and hold a value as opposed to its address. (In Chapter 2, you learned about enumerations; they are value types too.) In programming circles, you might hear debate about whether everything in C# is an object. Although some disagree, the general consensus is that value types like `int` and `double` are not objects; only reference types are. Further confusing the issue, a value type can easily be converted to an `Object` with a statement such as `Object myObject = myInteger;`. The action that takes place in such an implicit conversion is known as *boxing*.



In C#, you can create a `struct` which is similar to a class in that it can contain field and methods but is a value type rather than a reference type. See the C# documentation for further details.



You also can use the `new` operator for simple data types. For example, to declare an integer variable `x`, you can write the following:

```
int x = new int();
```

However, programmers usually use the simpler form:

```
int x;
```

With the first form, `x` is initialized to 0. With the second form, `x` holds no usable starting value.

In the statement `Employee myAssistant = new Employee();`, the last portion of the statement after the `new` operator, `Employee()`, looks suspiciously like a method name with its parentheses. In fact, it is the name of a method that constructs an `Employee` object. `Employee()` is a *constructor*. You will write your own constructors later in this chapter. For now, note that when you don't write a constructor for a class, `C#` writes one for you, and the name of the constructor is always the same as the name of the class whose objects it constructs.



Later in this chapter you will learn that the automatically generated constructor for a class is a *default constructor*, which means it takes no parameters. Programmers sometimes erroneously equate the terms *automatically generated* and *default*. The automatically created constructor for a class is an *example* of a default constructor, but you also can explicitly create your own default constructor.

After an object has been instantiated, its nonstatic `public` members can be accessed using the object's identifier, a dot, and the nonstatic public member's identifier. For example, if you declare an `Employee` named `myAssistant`, you can access `myAssistant`'s `WelcomeMessage()` method with the following statement:

```
myAssistant.WelcomeMessage();
```

An object such as `myAssistant` that uses an instance method is an **invoking object**. The statement `myAssistant.WelcomeMessage();` would be illegal if `WelcomeMessage()` was a `static` method. The method can be used appropriately with each unique `Employee` object only because it is nonstatic.

A program that declares `myAssistant` cannot access `myAssistant`'s `idNumber` directly; the only way a program can access the `private` data is by using one of the object's `public` methods. Because the `WelcomeMessage()` method is part of the same class as `idNumber`, and because `WelcomeMessage()` is `public`, a client method (for example, a `Main()` method in a program) can use the method. Figure 9-4 shows a class named `CreateEmployee` whose `Main()` method declares an `Employee` and displays the `Employee`'s welcome message. Figure 9-5 shows the execution of the program.

```
using static System;
class CreateEmployee
{
    static void Main()
    {
        Employee myAssistant = new Employee();
        myAssistant.WelcomeMessage();
    }
}
```

Figure 9-4 The `CreateEmployee` program

```
C:\C#\Chapter.09>CreateEmployee
Welcome from Employee #0
How can I help you?

C:\C#\Chapter.09>
```

Figure 9-5 Output of the CreateEmployee program

In the output in Figure 9-5, the Employee's ID number is 0. By default, all unassigned numeric fields in an object are initialized to 0. When you compile the program in Figure 9-4, you receive a warning message:

```
Field 'Employee.idNumber' is never assigned to, and will always have its default value 0.
```

Of course, usually you want to provide a different value for each Employee's `idNumber` field. One way to provide values for object fields is to create *properties*, which you will do later in this chapter.

Passing Objects to Methods

A class represents a data type, and you can pass objects to methods just as you can pass variables that are simple data types. For example, the program in Figure 9-6 shows how string and Employee objects can be passed to a `DisplayEmployeeData()` method. Each Employee is assigned to the `emp` parameter within the method. Figure 9-7 shows the program's output.

```
using static System.Console;
class CreateTwoEmployees
{
    static void Main()
    {
        Employee aWorker = new Employee();
        Employee anotherWorker = new Employee();
        DisplayEmployeeData("First", aWorker);
        DisplayEmployeeData("Second", anotherWorker);
    }
    static void DisplayEmployeeData(string order, Employee emp)
    {
        WriteLine("\n{0} employee's message:", order);
        emp.WelcomeMessage();
    }
}
```

Figure 9-6 The CreateTwoEmployees program

```

C:\C#\Chapter.09>CreateTwoEmployees

First employee's message:
Welcome from Employee #0
How can I help you?

Second employee's message:
Welcome from Employee #0
How can I help you?

C:\C#\Chapter.09>

```

Figure 9-7 Output of the `CreateTwoEmployees` program

When you pass an object to a method, you pass a reference. Therefore, any change made to an object parameter in a method also affects the object used as an argument in the calling method.

In Figure 9-6, the `DisplayEmployeeData()` method is `internal` by default. Because the method accepts an `Employee` parameter, it must be no more accessible than the `Employee` class, which is also `internal`. This restriction preserves the security provided for the `Employee` class's nonpublic members. The program would also work if you made both the `DisplayEmployeeData()` method and the `Employee` class `public`.

TWO TRUTHS & A LIE

Creating Objects

1. Declaring a class creates one object of a new data type.
2. After you declare a class, you must use the `new` operator to allocate memory for an object of that class and to instantiate it.
3. After an object has been instantiated, its public members can be accessed using the object's identifier, a dot, and a method call.

The false statement is #1. Declaring a class does not create any actual objects; the declaration describes only what an object of that class will be.

Creating Properties

Frequently, methods you call with an object are used to alter the states of its fields. For example, you might want to set or change the date or time of a party. In the `Employee` class, you could write a method such as `SetIdNumber()` to set an employee's `idNumber` field as follows:

```

public void SetIdNumber(int number)
{
    idNumber = number;
}

```

Then, after you instantiate an `Employee` object named `myAssistant`, you could call the method with a statement like the following:

```
myAssistant.SetIdNumber(785);
```

364

Although this technique would work, and might be used in other programming languages, C# programmers more often create properties to perform similar tasks. A **property** is a member of a class that provides access to a field of a class; properties define how fields will be set and retrieved. C# programmers refer to properties as “smart fields” because they provide a combination of the best features of `private` fields and `public` methods:

- Like public methods, they protect `private` data from outside access.
- Like fields, their names are used in the same way simple variable names are used. When you create properties, the syntax in your client programs becomes more natural and easier to understand.

Properties have **accessors** that specify the statements that execute when a class’s fields are accessed. Specifically, properties contain **set accessors** for setting an object’s fields and **get accessors** for retrieving the stored values. Figure 9-8 shows an `Employee` class in which a property named `IdNumber` has been defined in the shaded area.

```
class Employee
{
    private int idNumber;
    public int IdNumber
    {
        get
        {
            return idNumber;
        }
        set
        {
            idNumber = value;
        }
    }
    public void WelcomeMessage()
    {
        WriteLine("Welcome from Employee #{0}", IdNumber);
        WriteLine("How can I help you?");
    }
}
```

Figure 9-8 Employee class with defined property

A property declaration resembles a variable declaration; it contains an access modifier, a data type, and an identifier. It also resembles a method in that it is followed by curly braces that contain statements. By convention, a property identifier is the same as the field it manipulates, except the first letter is capitalized. The field that supports a property is its **backing field**. Be careful with capitalization of properties. For example, within a **get** accessor for `IdNumber`, if you return the property (`IdNumber`) instead of the backing field (`idNumber`), you initiate an infinite loop—the property continuously accesses itself.

The `IdNumber` property in Figure 9-8 contains both **get** and **set** accessors defined between curly braces; a property declaration can contain a **get** accessor, a **set** accessor, or both. When a property has a **set** accessor, programmers say the property can be “written to.” When it has a **get** accessor, programmers say the property can be “read from.” When a property has only a **get** accessor (and not a **set** accessor), it is a **readonly property**. In C#, the **get** and **set** accessors often are called the **getter** and the **setter**, respectively.



In the `WelcomeMessage()` method in Figure 9-8, the `IdNumber` property is displayed. Alternatively, this method could continue to use the `idNumber` field (as in Figure 9-3) because the method is a member of the same class as the field. Programmers are divided on whether a method of a class should use a field or a property to access its own methods. One popular position is that if **get** and **set** accessors are well designed, they should be used everywhere, even within the class. Sometimes, you want a field to be **readonly**, so you do not create a **set** accessor. In such a case, you can use the field (with the lowercase initial by convention) within class methods.



Throughout this book you have seen keywords displayed in boldface in the program figures. The words `get`, `set`, and `value` are not C# keywords—for example, you could declare a variable named `get` within a C# program. However, within a property, `get`, `set`, and `value` have special meanings and are not allowed to be declared as identifiers there. In the Visual Studio Integrated Development Environment, these three words appear in blue within properties, but in black elsewhere. Identifiers that act like keywords in specific circumstances are **contextual keywords**. C# has six contextual keywords: `get`, `set`, `value`, `partial`, `where`, and `yield`.

Each accessor in a property looks like a method, except no parentheses are included in the identifier. A **set** accessor acts like a method that accepts a parameter and assigns it to a variable. However, it is not a method and you do not use parentheses with it. A **get** accessor returns the value of the field associated with the property, but you do not code a return type; the return type of a **get** accessor is implicitly the type of the property in which it is contained.

When you use **set** and **get** accessors in a method, you do not use the words *set* or *get*. Instead, to set a value, you use the assignment operator (`=`), and to get a value, you simply use the property name. For example, if you declare an `Employee` named `myChef`, you can assign an `IdNumber` as simply as you would a variable, as in the following:

```
Employee myChef = new Employee();
myChef.IdNumber = 2345;
```

In the second statement, the `IdNumber` property is set to 2345. The value to the right of the equal sign is sent to the `set` accessor as an implicit parameter named `value`. (An **implicit parameter** is one that is undeclared and that gets its value automatically.) In the statement `myChef.IdNumber = 2345;`, the constant 2345 is sent to the `set` accessor, where it becomes the value of `value`. Within the `set` accessor, `value` is assigned to the field `idNumber`. The `idNumber` field could not have been set directly from `Main()` because it is private; however, the `IdNumber` property can be set through its `set` accessor because the property is `public`.

Writing a `get` accessor allows you to use a property like you would a simple variable. For example, a declared `Employee`'s ID number can be displayed with the following:

```
WriteLine("ID number is {0}", myChef.IdNumber);
```

The expression `myChef.idNumber` (using the field name that starts with a lowercase `i`) would not be allowed in a client program because `idNumber` is `private`; however, the `public` `get` accessor of the property allows `myChef.IdNumber` to be displayed. Figure 9-9 shows a complete application that uses the `Employee` class with accessors defined in Figure 9-8. Figure 9-10 shows the output.

```
using static System.Console;
class CreateEmployee2
{
    static void Main()
    {
        Employee myChef = new Employee();
        myChef.IdNumber = 2345;
        WriteLine("ID number is {0}", myChef.IdNumber);
        myChef.WelcomeMessage();
    }
}
```

Figure 9-9 The `CreateEmployee2` application that uses the `Employee` class containing a property

```
C:\C#\Chapter.09>CreateEmployee2
ID number is 2345
Welcome from Employee #2345
How can I help you?
C:\C#\Chapter.09>
```

Figure 9-10 Output of the `CreateEmployee2` application

At this point, declaring `get` and `set` accessors that do nothing except retrieve a value from a field or assign a value to one might seem like a lot of work for very little payoff. After all, if a field was `public` instead of `private`, you would just use it directly and avoid the work of creating the property. However, it is conventional (and consistent with object-oriented principles) to make a class's fields `private` and allow accessors to manipulate them only as you deem appropriate. Keeping data hidden is an important feature of object-oriented programming, as is controlling how data values are set and used. Additionally, you can customize accessors to suit the restrictions you want to impose on how some fields are retrieved and accessed. For example, you could write a `set` accessor that restricts ID numbers within the `Employee` class as follows:

```
set
{
    if(value < 500)
        idNumber = value;
    else
        idNumber = 500;
}
```

This `set` accessor would ensure that an `Employee idNumber` would never be greater than 500. If clients had direct access to a `private idNumber` field, you could not control what values could be assigned there, but when you write a custom `set` accessor for your class, you gain full control over the allowed data values.

A minor inconvenience when using properties is that a property cannot be passed to a method as a `ref` or `out` parameter. For example, if `IdNumber` is an `Employee` property and `myChef` is an `Employee` object, the following statement does not compile:

```
int.TryParse(ReadLine(), out myChef.IdNumber);
```

Instead, to use `TryParse()` to interactively get an `IdNumber` value, you must use a temporary variable, as in the following example:

```
int temp;
int.TryParse(ReadLine(), out temp);
myChef.IdNumber = temp;
```

Using Auto-Implemented Properties

Although you can include any number of custom statements within the `get` and `set` accessors in a property, the most frequent scenario is that a `set` accessor simply assigns a value to the appropriate field, and the `get` accessor simply returns the field value. Because the code in `get` and `set` accessors frequently is standard as well as brief, programmers sometimes take one of several shorthand approaches to writing properties.

For example, instead of writing an `IdNumber` property using 11 code lines as in Figure 9-8, a programmer might write the property on five lines, as follows:

```
public int IdNumber
{
    get{return idNumber;}
    set{idNumber = value;}
}
```

368

This format does not eliminate any of the characters in the original version of the property; it eliminates only some of the white space, placing each accessor on a single line.

Other programmers choose an even more condensed form and write the entire property on one line as:

```
public int IdNumber {get{return idNumber;} set{idNumber = value;}}
```

An even more concise format was introduced in C# 3.0. In this and newer versions of C#, you can write a property as follows:

```
public int IdNumber {get; set;}
```

A property written in this format is an **auto-implemented property**. The property's implementation (its set of working statements) is created for you automatically with the assumptions that the `set` accessor should simply assign a value to the appropriate field, and the `get` accessor should simply return the field. You cannot use an auto-implemented property if you need to include customized statements within one of your accessors (such as placing restrictions on an assigned value), and you can declare an auto-implemented property only when you use both `get` and `set`. Auto-implemented properties are sometimes called **automatic properties**.

When you create an auto-implemented property, you should not declare the backing field that corresponds to the property. The corresponding backing field is generated by the compiler and has an internal name that would not match the field's name if you had coded the field yourself. If you also declare a field for the property and do not refer to it explicitly, you will receive a warning that your field is never used because your field and the property's field refer to separate memory addresses. You can avoid the warning by using the declared field, but that can lead to potential confusion and errors. For example, Figure 9-11 shows an `Employee` class in which no specialized code is needed for the properties associated with the ID number and salary. In this class, the fields are not declared explicitly; only auto-implemented properties are declared. The figure also contains a short program that uses the class, and Figure 9-12 shows the output. Auto-implemented properties provide a convenient shortcut when you need both `get` and `set` abilities without specialized statements.


```
using static System.Console;
class CreateEmployee3
{
    static void Main()
    {
        Employee aWorker = new Employee();
        aWorker.IdNumber = 3872;
        aWorker.Salary = 22.11;
        WriteLine("Employee #{0} makes {1}",
            aWorker.IdNumber, aWorker.Salary.ToString("C"));
    }
}
class Employee
{
    public int IdNumber {get; set;}
    public double Salary {get; set;}
}
```

Figure 9-11 An Employee class with no declared fields and auto-implemented properties, and a program that uses them

```
C:\C#\Chapter.09>CreateEmployee3
Employee #3872 makes $22.11
C:\C#\Chapter.09>
```

Figure 9-12 Output of the CreateEmployee3 application

If you want to create a read-only property using auto-implemented accessors, you can make the set accessor private, as in the following:

```
public int IdNumber {get; private set;}
```

Using this technique, the `IdNumber` property cannot be set by any statement in another class. However, it could be assigned a value by methods within its own class. You also might want to create a property that outside classes could set but not retrieve, as follows:

```
public int IdNumber {private get; set;}
```



Watch the video *Creating Properties*.

TWO TRUTHS & A LIE

Creating Properties

1. A property is a member of a class that defines how fields will be set and retrieved.
2. Properties contain `set` accessors for retrieving an object's fields and `get` accessors for setting the stored values.
3. You can create auto-implemented properties when you want a field's `set` accessor simply to assign a value to the appropriate field, and a field's `get` accessor simply to return the field value.

The false statement is #2. Properties contain `set` accessors for setting an object's fields and `get` accessors for retrieving the stored values.



You Do It

Creating a Class and Objects

In this section, you create a `Student` class and instantiate objects from it. This class contains an ID number, last name, and grade point average for the `Student`. It also contains properties that get and set each of these fields. You also pass each `Student` object to a method.

1. Start a new project by declaring a `Student` class. Insert an opening curly brace, and declare three `private` fields that will hold an ID number, last name, and grade point average, as follows:

```
class Student  
{  
    private int idNumber;  
    private string lastName;  
    private double gradePointAverage;
```

2. Add two constants that represent the highest and lowest possible values for a grade point average.

```
public const double HIGHEST_GPA = 4.0;  
public const double LOWEST_GPA = 0.0;
```

(continues)

(continued)

3. Add two properties that get and set `idNumber` and `lastName`. By convention, properties have an identifier that is the same as the field they service, except they start with a capital letter.

```
public int IdNumber
{
    get
    {
        return idNumber;
    }
    set
    {
        idNumber = value;
    }
}
public string LastName
{
    get
    {
        return lastName;
    }
    set
    {
        lastName = value;
    }
}
```

4. Add the following `set` accessor in the property for the `gradePointAverage` field. It sets limits on the value assigned, assigning 0 if the value is out of range.

```
public double GradePointAverage
{
    get
    {
        return gradePointAverage;
    }
    set
    {
        if(value >= LOWEST_GPA && value <= HIGHEST_GPA)
            gradePointAverage = value;
        else
            gradePointAverage = LOWEST_GPA;
    }
}
```

(continues)

(continued)

5. Add a closing curly brace for the class. Save the file as **CreateStudents.cs**.

6. At the top of the file, begin a program that creates two `Student` objects, assigns some values, and displays the `Students`.

```
using System;
using static System.Console;
class CreateStudents
{
```

7. Add a `Main()` method that declares two `Students`. Assign field values, including one “illegal” value—a grade point average that is too high.

```
static void Main()
{
    Student first = new Student();
    Student second = new Student();
    first.IdNumber = 123;
    first.LastName = "Anderson";
    first.GradePointAverage = 3.5;
    second.IdNumber = 789;
    second.LastName = "Daniels";
    second.GradePointAverage = 4.1;
```

8. Instead of creating similar `writeLine()` statements to display the two `Students`, call a method with each `Student`. You will create the method to accept a `Student` argument in the next step. Add a closing curly brace for the `Main()` method.

```
    Display(first);
    Display(second);
}
```

9. Write the `Display()` method so that the passed-in `Student`'s `IdNumber`, `LastName`, and `GradePointAverage` are displayed and aligned. Add a closing curly brace for the class.

```
static void Display(Student stu)
{
    WriteLine("{0, 5} {1, -10}{2, 6}", stu.IdNumber,
              stu.LastName, stu.GradePointAverage.ToString("F1"));
}
}
```



Recall that field contents are left-aligned when you use a minus sign before the field size. Also recall that the “F1” argument to the `ToString()` method causes the value to be displayed to one decimal place.

(continues)

(continued)

10. Save the file, and then compile and execute the program. Figure 9-13 shows the output. Each `Student` has unique data values and uses the same `Display()` method. Notice how the second `Student`'s grade point average was forced to 0 by the `set` accessor in the property for the field.

```
C:\C#\Chapter.09>CreateStudents
123 Anderson 3.5
789 Daniels 0.0
C:\C#\Chapter.09>_
```

Figure 9-13 Output of the `CreateStudents` program

Using Auto-Implemented Properties

When a property's `get` accessor simply returns the corresponding field's value, and its `set` accessor simply assigns a value to the appropriate field, you can reduce the code in your classes by using auto-implemented properties. In the `Student` class, both `IdNumber` and `LastName` are candidates for this shortcut, so you can replace the full versions of these properties with their auto-implemented versions. The `GradePointAverage` property cannot take advantage of auto-implementation because additional code is required for the property to fulfill its intended function.

1. Open the file that contains the `Student` class if it is not still open on your screen. Remove the properties for `IdNumber` and `LastName` and replace them with these auto-implemented versions:

```
public int IdNumber {get; set;}
public string LastName {get; set;}
```
2. Remove the declarations for the fields `idNumber` and `lastName`.
3. Save the file, and then recompile and execute it. The output is the same as when the program used the original version of the `Student` class in Figure 9-13.

More About `public` and `private` Access Modifiers

Most of the time, a class's data fields are `private` and its methods are `public`. This technique ensures that data will be used and changed only in the ways provided in your accessors. Novice programmers might make a data field `public` to avoid having to create a property containing `get` and `set` accessors. However, doing so violates a basic principle of object-oriented programming. Data should be hidden when at all possible, and access to it should be controlled by well-designed accessors.

Although `private` fields and `public` methods and accessors are the norm, occasionally you need to create `public` fields or `private` methods. Consider the `Carpet` class shown in Figure 9-14. Although it contains several `private` data fields, this class also contains one `public` constant field (shaded). Following the three `public` property declarations, one `private` method is defined (also shaded).

374

```
class Carpet
{
    public const string MOTTO = "Our carpets are quality-made";
    private int length;
    private int width;
    private int area;
    public int Length
    {
        get
        {
            return length;
        }
        set
        {
            length = value;
            CalcArea();
        }
    }
    public int Width
    {
        get
        {
            return width;
        }
        set
        {
            width = value;
            CalcArea();
        }
    }
    public int Area
    {
        get
        {
            return area;
        }
    }
    private void CalcArea()
    {
        area = Length * Width;
    }
}
```

Figure 9-14 The Carpet class

When you create Carpet objects from the class in Figure 9-14, each Carpet will have its own Length, width, and area, but all Carpet objects will have the same MOTTO. The field MOTTO is preceded by the keyword `const`, meaning MOTTO is constant. That is, no program can change its value. Making a constant `public` does not violate information hiding in the way that making an object's data fields `public` would, because programs that use this class cannot change the constant's value.

When you define a named constant within a class, it is always `static`, even though you cannot use the keyword `static` as a modifier. In other words, the field belongs to the entire class, not to any particular instance of the class. When you create a `static` field, only one copy is stored for the entire class, no matter how many objects you instantiate. When you use a constant field in a client class, you use the class name and a dot rather than an object name, as in `Carpet.MOTTO`. Some built-in C# classes contain useful named constants, such as `Math.PI`, which contains the value of pi. You do not create a `Math` object to use `PI`; therefore, you know it is `static`.



Throughout this book, you have been using `static` to describe the `Main()` method of a class. You do not need to create an object of any class that contains a `Main()` method to be able to use `Main()`.

Figure 9-15 shows a program that instantiates and uses a Carpet object, and Figure 9-16 shows the results when the program executes. Notice that, although the output statements require an object to use the `Width`, `Length`, and `Area` properties, `MOTTO` is referenced using the class name only.

```
using static System.Console;
class TestCarpet
{
    static void Main()
    {
        Carpet aRug = new Carpet();
        aRug.Width = 12;
        aRug.Length = 14;
        Write("The {0} X {1} carpet ", aRug.Width, aRug.Length);
        WriteLine("has an area of {0}", aRug.Area);
        WriteLine("Our motto is: {0}", Carpet.MOTTO);
    }
}
```

Figure 9-15 The TestCarpet class

```
C:\C#\Chapter.09>TestCarpet
The 12 X 14 carpet has an area of 168
Our motto is: Our carpets are quality-made
C:\C#\Chapter.09>
```

376

Figure 9-16 Output of the TestCarpet program

The Carpet class contains one private method named `CalcArea()`. As you examine the code in the `TestCarpet` class in Figure 9-15, notice that `Width` and `Length` are set using an assignment operator but `Area` is not. The `TestCarpet` class can make assignments to `Width` and `Length` because these properties are `public`. However, you would not want a client program to assign a value to `Area` because the assigned value might not agree with the `Width` and `Length` values. Therefore, the `Area` property is a read-only property—it does not contain a `set` accessor, and no assignments by clients are allowed. Instead, whenever the `Width` or `Length` properties are changed, the private `CalcArea()` method is called from the accessor. The `CalcArea()` method is defined as `private` because there is no reason for a client class like `TestCarpet` to call `CalcArea()`. Programmers probably create `private` methods more frequently than they create `public` data fields. Some programmers feel that the best style is to use `public` methods that are nothing but a list of method calls with descriptive names. Then, the methods that actually do the work are all `private`.

TWO TRUTHS & A LIE

More About `public` and `private` Access Modifiers

1. Good object-oriented techniques require that data should usually be hidden and access to it should be controlled by well-designed accessors.
2. Although `private` fields, methods, and accessors are the norm, occasionally you need to create `public` versions of them.
3. When you define a named constant within a class, it is always `static`; that is, the field belongs to the entire class, not to any particular instance of the class.

The false statement is #2. Although `private` fields and `public` methods and accessors are the norm, occasionally you need to create `public` fields or `private` methods.

Understanding the this Reference

When you create a class, only one copy of the class code is stored in computer memory. However, you might eventually create thousands of objects from the class. When you create each object, you provide storage for each of the object's instance variables. For example, Figure 9-17 shows part of a `Book` class that contains three fields, a property for the title field, and an advertising message method. When you declare several `Book` objects, as in the following statements, each `Book` object requires separate memory locations for its `title`, `numPages`, and `price`:

```
Book myBook = new Book();
Book yourBook = new Book();
```

```
class Book
{
    private string title;
    private int numPages;
    private double price;
    public string Title
    {
        get
        {
            return title;
        }
        set
        {
            title = value;
        }
    }
    public void AdvertisingMessage()
    {
        WriteLine("Buy it now: {0}", Title);
    }
}
```

Figure 9-17 Partially developed `Book` class



When you compile the `Book` class, you receive warnings that the `numPages` and `price` fields are never used. The omission was purposeful for this demonstration program.

In this example, storing a single `Book` object requires allocating storage space for three separate fields; the storage requirements for `Book` objects used by a library or retail bookstore would be far more considerable, but necessary—each `Book` must be able to “hold” its own data, including publisher, date published, author, ISBN, and so on. Each nonstatic field is called an *instance variable* precisely because separate values are stored for each instance. In addition to its fields,

if each `Book` object also required its own copy of each property and method contained in the class, the storage requirements would multiply. It makes sense that each `Book` needs space to store its unique title and other data, but because every `Book` uses the same methods, storing multiple copies of methods is wasteful and unnecessary.

Fortunately, each `Book` object does not need to store its own copy of each property and method. Whether you make the method call `myBook.AdvertisingMessage()` or `yourBook.AdvertisingMessage()`, you access the same `AdvertisingMessage()` method. However, there must be a difference between the two method calls, because each displays a different title in its message. The difference lies in an implicit, or invisible, reference that is passed to every instance method and property accessor. The implicitly passed reference is the **this reference**, which is the address of the invoking object. When you call the method `myBook.AdvertisingMessage()`, you automatically pass the `this` reference to the method so that it knows which instance of `Book` to use. Only nonstatic methods receive a `this` reference. The `this` reference is aptly named. When you execute a nonstatic method such as `myBook.AdvertisingMessage()`, you might ask yourself which object's data will be used in the method. The answer is "*This* object's (*myBook's*) data."

You can explicitly refer to the `this` reference within an instance method or property, as shown in Figure 9-18. When you refer to `Title` (or `title`) within a `Book` class method or accessor, you are referring to the `title` field of "this" `Book`—the `Book` whose name you used in the method call, perhaps `myBook` or `yourBook`. Using the shaded keywords in Figure 9-18 is not required; the version of the methods shown in Figure 9-17 (where `this` was implied but not written explicitly) works just as well. Figure 9-19 shows an application that uses the `Book` class, and Figure 9-20 shows the output.

```
class Book
{
    private string title;
    private int numPages;
    private double price;
    public string Title
    {
        get
        {
            return this.title;
        }
        set
        {
            this.title = value;
        }
    }
    public void AdvertisingMessage()
    {
        WriteLine("Buy it now: {0}", this.Title);
    }
}
```

Figure 9-18 The `Book` class with methods explicitly using `this` references

```
using static System;
class CreateTwoBooks
{
    static void Main()
    {
        Book myBook = new Book();
        Book yourBook = new Book();
        myBook.Title = "Silas Marner";
        yourBook.Title = "The Time Traveler's Wife";
        myBook.AdvertisingMessage();
        yourBook.AdvertisingMessage();
    }
}
```

Figure 9-19 Program that declares two Book objects

```
C:\C#\Chapter.09>CreateTwoBooks
Buy it now: Silas Marner
Buy it now: The Time Traveler's Wife
C:\C#\Chapter.09>
```

Figure 9-20 Output of the CreateTwoBooks program

The `Book` class in Figure 9-18 worked without adding the explicit references to `this`. However, you should be aware that the `this` reference is always there, working behind the scenes, even if you do not code it. Sometimes, you may want to include the `this` reference within a method for clarity, so the reader has no doubt when you are referring to an instance variable.

On occasion, you might need to explicitly code the `this` reference. For example, consider the abbreviated `Book` class in Figure 9-21. The programmer has chosen the same identifier for a field and a parameter to a method that uses the field. To distinguish between the two, the field must be referenced as `this.price` within the method that has a parameter with the same name. All references to `price` (without using `this`) in the `SetPriceAndTax()` method in Figure 9-21 are references to the local parameter. If the method included the statement `price = price;`, the parameter's value would be assigned to itself and the class's field would never be set.

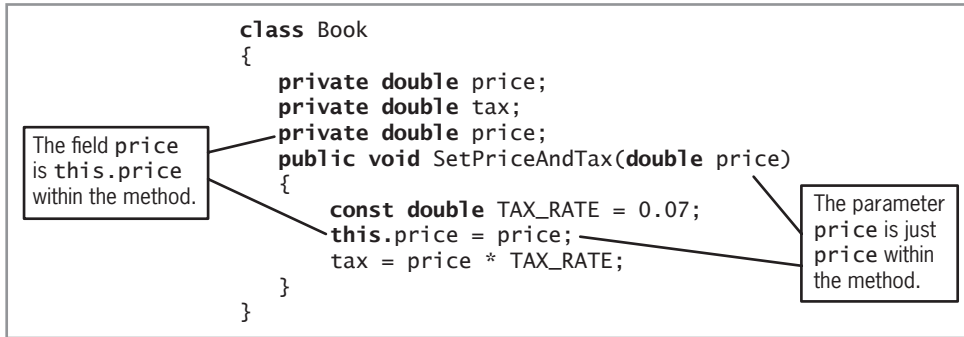


Figure 9-21 Book class that must explicitly use the `this` reference



Watch the video Understanding the *this* Reference.

TWO TRUTHS & A LIE

Understanding the `this` Reference

1. An implicit, or invisible, `this` reference is passed to every instance method and property accessor in a class; instance methods and properties are nonstatic.
2. You can explicitly refer to the `this` reference within an instance method or property.
3. Although the `this` reference exists in every instance method, it is invisible, so you can never refer to it within a method.

The false statement is #3. Sometimes, you may want to include the `this` reference within a method for clarity, so the reader has no doubt when you are referring to an instance variable.

Understanding Constructors

When you create a class such as `Employee` and instantiate an object with a statement such as `Employee aWorker = new Employee();`, you are actually calling a method named `Employee()` that is provided by `C#`. A **constructor** is a method that instantiates (creates an instance of) an object. If you do not write a constructor for a class, then each class you create is automatically supplied with a `public` constructor with no parameters. A constructor without

parameters is a class's **default constructor**. (The term *default constructor* is not just used for a class's automatically supplied constructor; it is used for any constructor that takes no parameters. You will learn to create constructors with parameters in the next section.)

The automatically created constructor named `Employee()` establishes one `Employee` with the identifier `aWorker`, and provides the following initial values to the `Employee`'s data fields:

- Numeric fields are set to 0 (zero).
- Character fields are set to `'\0'`.
- Boolean fields are set to `false`.
- References, such as `string` fields or any other object fields, are set to `null` (or empty).

The value of an object initialized with a default constructor is known as the **default value of the object**. If you do not want all of an `Employee`'s fields to hold default values, or if you want to perform additional tasks when you create an `Employee`, you can write your own constructor to replace the automatically supplied version. Any constructor you write must have the same name as its class, and constructors cannot have a return type. For example, if you create an `Employee` class that contains a `Salary` property, and you want every new `Employee` object to have a `salary` of 300.00, you could write the constructor for the `Employee` class as follows:

```
Employee()
{
    Salary = 300.00;
}
```

Assuming a `Salary` property has been defined for the `salary` field, any instantiated `Employee` will have a default `salary` value of 300.00.

You can write any statement in a constructor. For example, you can perform arithmetic or display a message. However, the most common constructor task is to initialize fields. If you create a class in which one or more fields are never assigned a value, the compiler will issue a warning that the fields in question will hold default values.

Passing Parameters to Constructors

You can create a constructor to ensure that all objects of a class are initialized with the same values in their data fields. Alternatively, you might create objects initialized with unique field values by writing constructors to which you pass one or more parameters. You then can use the parameter values to set properties or fields for individual object instantiations.

For example, consider an `Employee` class with two data fields, a constructor, and an auto-implemented property, as shown in Figure 9-22. Its constructor assigns 9.99 to each potentially instantiated `Employee`'s `PayRate`. Any time an `Employee` object is created using a statement such as `Employee partTimeWorker = new Employee();`, even if no other data-assigning methods are ever used, you are ensured that the `Employee`'s `PayRate` holds a default value.

```
class Employee
{
    private int idNumber;
    private string name;
    public Employee()
    {
        PayRate = 9.99;
    }
    public double PayRate {get; set;}
    // Other class members can go here
}
```

Figure 9-22 Employee class with a parameterless constructor

The constructor in Figure 9-22 is a **parameterless constructor**—one that takes no arguments. In other words, it is a default constructor. As an alternative, you might choose to create `Employee`s with a different initial pay rate value for each `Employee`. To accomplish this task, you can pass a pay rate to the constructor. Figure 9-23 shows an `Employee` constructor that receives a parameter. A value is passed to the constructor using a statement such as the following:

```
Employee partTimeWorker = new Employee(12.50);
```

When the constructor executes, the double used as the actual parameter within the method call is passed to `Employee()` and assigned to the `Employee`'s `PayRate`.

```
public Employee(double rate)
{
    PayRate = rate;
}
```

Figure 9-23 Employee constructor with parameter

Overloading Constructors

C# automatically provides a default constructor for your classes. As soon as you create your own constructor, whether it has parameters or not, you no longer have access to the automatically created version. However, if you want a class to have both parameter and parameterless versions of a constructor, you can create them. Like any other C# methods, constructors can be overloaded. You can write as many constructors for a class as you want, as long as their parameter

lists do not cause ambiguity. A class can contain only one parameterless constructor: the default constructor, which can be the automatically provided one or one that you write. If you wrote multiple parameterless constructors, they would be ambiguous. If you create any number of constructors but do not create a parameterless version, then the class does not have a default constructor.

For example, the `Employee` class in Figure 9-24 contains four constructors. The `Main()` method within the `CreateSomeEmployees` class in Figure 9-25 shows how different types of `Employees` might be instantiated. Notice that one version of the `Employee` constructor—the one that supports a character parameter—doesn't even use the parameter; sometimes you might create a constructor with a specific parameter type simply to force that constructor to be the version that executes. The output of the `CreateSomeEmployees` program is shown in Figure 9-26.

```

class Employee
{
    public int IdNumber {get; set;}

    public double Salary {get; set;}
    public Employee()
    {
        IdNumber = 999;
        Salary = 0;
    }
    public Employee(int empId)
    {
        IdNumber = empId;
        Salary = 0;
    }
    public Employee(int empId, double sal)
    {
        IdNumber = empId;
        Salary = sal;
    }
    public Employee(char code)
    {
        IdNumber = 111;
        Salary = 100000;
    }
}

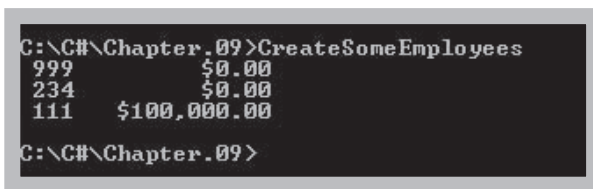
```

This parameterless constructor is the class's default constructor.

Figure 9-24 Employee class with four constructors

```
using static System.Console;
class CreateSomeEmployees
{
    static void Main()
    {
        Employee aWorker = new Employee();
        Employee anotherWorker = new Employee(234);
        Employee theBoss = new Employee('A');
        WriteLine("{0, 4}{1, 14}", aWorker.IdNumber,
            aWorker.Salary.ToString("C"));
        WriteLine("{0, 4}{1, 14}",
            anotherWorker.IdNumber,
            anotherWorker.Salary.ToString("C"));
        WriteLine("{0, 4}{1, 14}", theBoss.IdNumber,
            theBoss.Salary.ToString("C"));
    }
}
```

Figure 9-25 The CreateSomeEmployees program



```
C:\C#\Chapter.09>CreateSomeEmployees
 999          $0.00
 234          $0.00
 111    $100,000.00
C:\C#\Chapter.09>
```

Figure 9-26 Output of the CreateSomeEmployees program

Most likely, a single application would not use all four constructors of the `Employee` class. More likely, each application that uses the class would use only one or two constructors. You create a class with multiple constructors to provide flexibility for your clients. For example, some clients might choose to construct `Employee` objects with just ID numbers, and others might prefer to construct them with ID numbers and salaries.

Using Constructor Initializers

The `Employee` class in Figure 9-24 contains four constructors, and each constructor initializes the same two fields. In a fully developed class used by a company, many more fields would be initialized, creating a lot of duplicated code. Besides the original extra work of writing the repetitive statements in these constructors, even more work will be required when the class is modified in the future. For example, if your organization institutes a new employee ID number format that requires a specific number of digits, then each constructor will have to be modified, requiring extra work. In addition, one or more of the constructor versions that must be modified might be overlooked, introducing errors into the programs that are clients of the class.

As an alternative to repeating code in the constructors, you can use a constructor initializer. A **constructor initializer** is a clause that indicates another instance of a class constructor should be executed before any statements in the current constructor body. Figure 9-27 shows a new version of the `Employee` class using constructor initializers in three of the four overloaded constructor versions.

```
class Employee
{
    public int IdNumber {get; set;}
    public double Salary {get; set;}
    public Employee() : this(999, 0)
    {
    }
    public Employee(int empId) : this(empId, 0)
    {
    }
    public Employee(int empId, double sal)
    {
        IdNumber = empId;
        Salary = sal;
    }
    public Employee(char code) : this(111, 100000)
    {
    }
}
```

Figure 9-27 Employee class with constructor initializers

In the three shaded clauses in Figure 9-27, the `this` reference is used to mean “the constructor for this object being constructed.” For example, when a client calls the parameterless `Employee` constructor, 999 and 0 are passed to the two-parameter constructor. There, they become `empId` and `sal`, parameters that are assigned to the `IdNumber` and `Salary` properties. If statements were written within the parameterless constructor, they would then execute; however, in this class, additional statements are not necessary. Similarly, if a client uses the constructor version that accepts only an ID number, that parameter and a 0 for salary are passed to the two-parameter constructor. The only time just one version of the constructor executes is when a client uses both an ID number and a salary as constructor arguments. In the future, if additional statements need to be added to the class (for example, a decision that ensures an ID number was at least five digits at construction), the decision would be added only to the two-parameter version of the constructor, and all the other versions could use it.

Using the readonly Modifier in a Constructor

You have already learned that when you use the `const` modifier with a variable, you must assign a value at declaration and the value cannot be changed. The `readonly` modifier is similar to `const` in that it is used to assign a constant value that cannot be changed, but when you use `readonly` with a field you can assign a value either at declaration or within a constructor. This means that different constructor versions can assign different values to a `readonly` field. A `const` field must be assigned its value when a program is compiled, but a `readonly` field can be assigned a value when initialized or at runtime. For example, a `readonly` field's value might be retrieved from user input or from the operating system.

Figure 9-28 shows a program with an `Employee` class with an `idNumber` field that has been declared `readonly`. (See the first shaded statement.) The `Main()` program can send different values to the constructor to assign to `idNumber`, but after construction, `idNumber` cannot be changed. The second shaded statement causes a compiler error.

```
class EmployeesReadOnlyDemo
{
    static void Main()
    {
        Employee myAssistant = new Employee(1234);
        Employee myDriver = new Employee(2345);
        myAssistant.IdNumber = 3456;
    }
}
class Employee
{
    private readonly int idNumber;
    public Employee(int id)
    {
        idNumber = id;
    }
    public int IdNumber
    {
        get
        {
            return idNumber;
        }
        set
        {
            idNumber = value;
        }
    }
}
```

Don't Do It

The `idNumber` field cannot be assigned a value after construction. This statement generates an error.

Figure 9-28 Employee class with a `readonly` field

TWO TRUTHS & A LIE

Understanding Constructors

1. Every class you create is automatically supplied with a public constructor with no parameters.
2. If you write a constructor for a class, you do not have a default constructor for the class.
3. Any constructor you write must have the same name as its class, and constructors cannot have a return type.

The false statement is #2. If you write a parameterless constructor for a class, it becomes the default constructor, and you lose the automatically supplied version. If you write only constructors that require parameters, then the class no longer contains a default constructor.



You Do It

Adding Overloaded Constructors to a Class

In these steps, you add overloaded constructors to the `Student` class.

1. Open the file that contains the `Student` class if it is not still open on your screen. Just before the closing curly brace for the `Student` class, add the following constructor. It takes three parameters and assigns them to the appropriate fields:

```
public Student(int id, string name, double gpa)
{
    IdNumber = id;
    LastName = name;
    GradePointAverage = gpa;
}
```

2. Add a second parameterless constructor. It calls the first constructor, passing 0 for the ID number, "XXX" for the name, and 0.0 for the grade point average. Its body is empty.

```
public Student() : this(0, "XXX", 0.0)
{
}
```

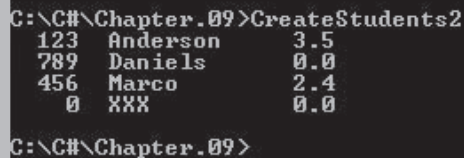
(continues)

(continued)

3. Change the name of the `CreateStudents` class to **`CreateStudents2`**.
4. After the existing declarations of the `Student` objects, add two more declarations. With one, use three arguments, but with the other, do not use any.

```
Student third = new Student(456, "Marco", 2.4);  
Student fourth = new Student();
```
5. At the end of the `Main()` method, just after the two existing calls to the `Display()` method, add two more calls using the new objects:

```
Display(third);  
Display(fourth);
```
6. Save the file, and then compile and execute it. The output looks like Figure 9-29. All four objects are displayed. The first two have had values assigned to them after declaration, but the third and fourth ones obtained their values from their constructors.



```
C:\C#\Chapter.09>CreateStudents2  
123 Anderson 3.5  
789 Daniels 0.0  
456 Marco 2.4  
0 XXX 0.0  
C:\C#\Chapter.09>
```

Figure 9-29 Output of the `CreateStudents2` program

Using Object Initializers

An **object initializer** allows you to assign values to any accessible members or properties of a class at the time of instantiation without calling a constructor with parameters. For example, assuming an `Employee` class has been created with a `public IdNumber` property and a parameterless constructor, you can write an object initializer in either of the following ways, with or without parentheses:

```
Employee aWorker = new Employee{IdNumber = 101};  
Employee aWorker = new Employee(){IdNumber = 101};
```

In either of these statements, 101 is assigned to the `aWorker` object's `IdNumber` property. The assignment is made within a pair of curly braces; using parentheses with the constructor name is optional. When either of these statements execute, the parameterless, default constructor for the class is executed first, and then the object initializer assignment is made.

For example, Figure 9-30 shows an `Employee` class that contains properties for `IdNumber` and `Salary` and a default constructor that assigns a value to `Salary`. For demonstration purposes, the constructor displays the current object's ID number and salary. The figure also shows a program that instantiates one `Employee` object and displays its value, and Figure 9-31 shows the output. When the object is created in the shaded statement, the constructor executes, assigns 99.99 to `Salary`, and displays the first line of output, showing `IdNumber` is still 0. After the object is constructed in the `Main()` method, the next output line is displayed, showing that the assignment of the ID number occurred after construction.

```
using static System.Console;
class DemoObjectInitializer
{
    static void Main()
    {
        Employee aWorker = new Employee {IdNumber = 101};
        WriteLine("Employee #{0} exists. Salary is {1}.",
            aWorker.IdNumber, aWorker.Salary);
    }
}
class Employee
{
    public int IdNumber {get; set;}
    public double Salary {get; set;}
    public Employee()
    {
        Salary = 99.99;
        WriteLine("Employee #{0} created. Salary is {1}.",
            IdNumber, Salary);
    }
}
```

Figure 9-30 The `DemoObjectInitializer` program

```
C:\C#\Chapter.09>DemoObjectInitializer
Employee #0 created. Salary is 99.99.
Employee #101 exists. Salary is 99.99.
C:\C#\Chapter.09>
```

Figure 9-31 Output of the `DemoObjectInitializer` program

To use object initializers, a class must have a default constructor. That is, you either must not create any constructors or you must create one that requires no parameters.

Multiple assignments can be made with an object initializer by separating them with commas, as in the following:

390

```
Employee myAssistant = new Employee {IdNumber = 202, Salary = 25.00};
```

This single code line has the same results as the following three statements:

```
Employee myAssistant = new Employee();  
myAssistant.IdNumber = 202;  
myAssistant.Salary = 25.00;
```

Using object initializers allows you to create multiple objects with different initial assignments without having to provide multiple constructors to cover every possible situation.

Additionally, using object initializers allows you to create objects with different starting values for different properties of the same data type. For example, consider a class like the `Box` class in Figure 9-32 that contains multiple properties of the same type. The constructor sets the `Height`, `Width`, and `Depth` properties to 1. You could write a constructor that accepts a single integer parameter to be assigned to `Height` (using the default value 1 for the other dimensions), but then you could not write an additional overloaded constructor that accepts a single integer parameter to be assigned to `Width` because the constructors would be ambiguous. However, by using object initializers, you can create objects to which you assign the properties you want. Figure 9-33 shows a program that declares three `Box` objects, each with a different assigned dimension, and Figure 9-34 shows the output, which demonstrates that each property was assigned appropriately.

```
class Box  
{  
    public int Height {get; set;}  
    public int Width {get; set;}  
    public int Depth {get; set;}  
    public Box()  
    {  
        Height = 1;  
        Width = 1;  
        Depth = 1;  
    }  
}
```

Figure 9-32 The `Box` class

```

using static System.Console;
class DemoObjectInitializer2
{
    static void Main()
    {
        Box box1 = new Box {Height = 3};
        Box box2 = new Box {Width = 15};
        Box box3 = new Box {Depth = 268};
        DisplayDimensions(1, box1);
        DisplayDimensions(2, box2);
        DisplayDimensions(3, box3);
    }
    static void DisplayDimensions(int num, Box box)
    {
        WriteLine("Box {0}: Height: {1} Width: {2} Depth: {3}",
            num, box.Height, box.Width, box.Depth);
    }
}

```

Figure 9-33 The DemoObjectInitializer2 program

```

C:\C#\Chapter.09>DemoObjectInitializer2
Box 1: Height: 3 Width: 1 Depth: 1
Box 2: Height: 1 Width: 15 Depth: 1
Box 3: Height: 1 Width: 1 Depth: 268
C:\C#\Chapter.09>

```

Figure 9-34 Output of the DemoObjectInitializer2 program

TWO TRUTHS & A LIE

Using Object Initializers

Assume that a working program contains the following object initializer:

```
Invoice oneBill = new Invoice {Amount = 0};
```

1. You know that the Invoice class contains a default constructor.
2. You know that an Invoice object could be constructed without using a parameter.
3. You know that the Invoice class contains a single property.

The false statement is #3. The Invoice class might have any number of properties. However, only one is being initialized in this statement.

Overloading Operators

C# operators are the symbols you use to perform operations on objects. You have used many operators, including arithmetic operators (such as + and -) and logical operators (such as == and <). Separate actions can result from what seems to be the same operation or command. This occurs frequently in all computer programming languages, not just object-oriented languages. For example, in most programming languages and applications such as spreadsheets and databases, the + operator has a variety of meanings. A few of them include:

- Alone before a value (called *unary form*), + indicates a positive value, as in the expression +7.
- Between two integers (called *binary form*), + indicates integer addition, as in the expression 5 + 9.
- Between two floating-point numbers (also called binary form), + indicates floating-point addition, as in the expression 6.4 + 2.1.
- Between two strings, + indicates concatenation, as in the expression "Hello," + "there".

Expressing a value as positive is a different operation from using the + operator to perform arithmetic or concatenate strings, so + is overloaded several times in that it can take one or two arguments and have a different meaning in each case. It also can take different operand types—you use a + to add two `ints`, two `doubles`, an `int` and a `double`, and a variety of other combinations. Each use results in different actions behind the scenes.

Just as it is convenient to use a + between both integers and `doubles` to add them, it also can be convenient to use a + between objects, such as `Employees` or `Books`, to add them. To be able to use arithmetic symbols with your own objects, you must overload the symbols.

C# operators are classified as unary or binary, depending on whether they take one or two arguments, respectively. The rules for overloading are shown in the following list.

- The overloadable unary operators are:
 - + - ! ~ ++ -- true false
- The overloadable binary operators are:
 - + - * / % & | ^ == != > < >= <=
- You cannot overload the following operators:
 - = && || ?? ?: checked unchecked new typeof as is
- You cannot overload an operator for a built-in data type. For example, you cannot change the meaning of + between two `ints`.

- When a binary operator is overloaded and it has a corresponding assignment operator, it is also overloaded. For example, if you overload `+`, then `+=` is automatically overloaded too.
- Some operators must be overloaded in pairs. For example, when you overload `==`, you also must overload `!=`, and when you overload `>`, you also must overload `<`.



When you overload `==`, you also receive warnings about methods in the `Object` class. You will learn about this class in the chapter “Introduction to Inheritance”; you should not attempt to overload `==` until you have studied that chapter.

You have used many, but not all, of the operators listed above. If you want to include an overloaded operator in a class, you must decide what the operator will mean in your class. When you do, you write statements in a method to carry out your meaning. The method has a return type and arguments just like other methods, but its identifier is required to be named operator followed by the operator being overloaded—for example, `operator+()` or `operator*()`.

For example, suppose that you create a `Book` class in which each object has a title, number of pages, and a price. Further assume that, as a publisher, you have decided to “add” `Books` together. That is, you want to take two existing `Books` and combine them into one. Assume that you want the new `Book` to have the following characteristics:

- The new title is a combination of the old titles, joined by the word “and.”
- The number of pages in the new book is equal to the sum of the pages in the original `Books`.
- Instead of charging twice as much for a new `Book`, you have decided to charge the price of the more expensive of the two original `Books`, plus \$10.

A different publisher might have decided that “adding `Books`” means something different—for example, an added `Book` might have a fixed new price of \$29.99. The statements you write in your `operator+()` method depend on how you define adding for your class. You could write an ordinary method to perform these tasks, but you could also overload the `+` operator to mean “add two `Books`.” Figure 9-35 shows a `Book` class that has properties for each field and a shaded `operator+()` method.

```
class Book
{
    public Book(string title, int pages, double price)
    {
        Title = title;
        NumPages = pages;
        Price = price;
    }
    public static Book operator+(Book first, Book second)
    {
        const double EXTRA = 10.00;
        string newTitle = first.Title + " and " + second.Title;
        int newPages = first.NumPages + second.NumPages;
        double newPrice;
        if(first.Price > second.Price)
            newPrice = first.Price + EXTRA;
        else
            newPrice = second.Price + EXTRA;
        return(new Book(newTitle, newPages, newPrice));
    }
    public string Title {get; set;}
    public int NumPages {get; set;}
    public double Price {get; set;}
}
```

Figure 9-35 Book class with overloaded + operator

The `operator+()` method in Figure 9-35 is declared to be `public` (so that class clients can use it) and `static` (which is required). The method is `static` because it does not receive a `this` reference to any object; instead, the two objects to be added are both parameters to the method.

The return type is `Book` because the addition of two `Books` is defined to be a new `Book` with different values from either of the originals. You could overload the `+` operator so that when two `Books` are added they return some other type, but it is most common to make the addition of two objects result in an “answer” of the same type.

The two parameters in the `operator+()` method in the `Book` class are both `Books`. Therefore, when you eventually call this method, the data types on both sides of the `+` sign will be `Books`. For example, you could write other methods that add a `Book` and an `Employee`, or a `Book` and a `double`.

Within the `operator+()` method, the statements perform the following tasks:

- A constant is declared to hold the extra price used in creating a new `Book` from two existing ones.
- A new string is created and assigned the first parameter `Book`'s title, plus the string “ and ”, plus the second parameter `Book`'s title.

- A new integer is declared and assigned the sum of the number of pages in each of the parameter `Books`.
- A new `double` is declared and assigned the value of the more expensive original `Book` plus \$10.00.
- Within the `return` statement, a new anonymous `Book` is created (an anonymous object is one without an identifier) using the new title, page number, and price and is returned to the calling method. (Instead of an anonymous `Book`, it would have been perfectly acceptable to use two statements—the first one creating a named `Book` with the same arguments, and the second one returning the named `Book`.)

Figure 9-36 shows a client program that can use the `+` operator in the `Book` class. It first declares three `Books`; then, in the shaded statement, it adds two `Books` together and assigns the result to the third. When `book1` and `book2` are added, the `operator+()` method is called. The returned `Book` is assigned to `book3`, which is then displayed. Figure 9-37 shows the results.

```
using static System.Console;
class AddBooks
{
    static void Main()
    {
        Book book1 = new Book("Silas Marner", 350, 15.95);
        Book book2 = new Book("Moby Dick", 250, 16.00);
        Book book3;
        book3 = book1 + book2;
        WriteLine("The new book is \"{0}\"", book3.Title);
        WriteLine("It has {0} pages and costs {1}",
            book3.NumPages, book3.Price.ToString("C"));
    }
}
```

Figure 9-36 The `AddBooks` program

```
C:\C#\Chapter.09>AddBooks
The new book is "Silas Marner and Moby Dick"
It has 600 pages and costs $26.00
C:\C#\Chapter.09>
```

Figure 9-37 Output of the `AddBooks` program

In the `Book` class, it took many statements to overload the operator; however, in the client class, just typing a `+` between objects allows a programmer to use the objects and operator intuitively. You could write any statements you wanted within the operator method. However, for clarity, you should write statements that intuitively have the same meaning as the common use of

the operator. For example, although you could overload the `operator*()` method to display a `Book`'s title and price instead of performing multiplication, it would be a bad programming technique.

Because each addition operation returns a `Book`, it is possible to chain addition in a statement such as `collection = book1 + book2 + book3;`, assuming that all the variables have been declared to be `Book` objects. In this example, `book1` and `book2` would be added, returning a temporary `Book`. Then the temporary `Book` and `book3` would be added, returning a different temporary `Book` that would be assigned to `collection`.

When you overload an operator, at least one argument to the method must be a member of the containing class. In other words, within the `Book` class, you can overload `operator*()` to multiply a `Book` by an integer, but you cannot overload `operator*()` to multiply a `double` by an integer.

When you overload a unary operator, the method takes a single argument. For example, suppose that you want to use a negative sign with a `Book` to make its price negative. You could use the operator in a statement such as the following:

```
returnedBook = -purchasedBook;
```

The `operator-()` method header might appear as follows:

```
public static Book operator-(Book aBook)
```

In this example, the minus sign is understood to cause unary negation instead of binary subtraction because the method accepts only one parameter, which is a `Book`. The method also returns a `Book`, which is an altered version of the parameter. Figure 9-38 shows a method that overloads the minus sign for a `Book` object. The method accepts a `Book` parameter, changes the price to its negative value, and returns the altered `Book` object to the calling method.

```
public static Book operator-(Book aBook)
{
    aBook.Price = -aBook.Price;
    return aBook;
}
```

Figure 9-38 An `operator-()` method for a `Book`



In some other languages, notably C++, you can write two methods to overload `++` and `--` differently as prefix and postfix operators. However, when you overload either the `++` or `--` operator in C#, the operator can be used either before or after the object, but the same method executes either way.



Watch the video *Overloading Operators*.

TWO TRUTHS & A LIE

Overloading Operators

1. All C# operators can be overloaded.
2. You cannot overload an operator for a built-in data type.
3. Some operators must be overloaded in pairs.

397

The false statement is #1. You cannot overload the following operators: = && ||
?? ? : checked unchecked new typeof as is

Declaring an Array of Objects

Just as you can declare arrays of integers or `doubles`, you can declare arrays that hold elements of any type, including objects. Remember that object names are references. When you create an array from a value type, such as `int` or `char`, the array holds the actual values. When you create an array from a reference type, such as a class you create, the array holds the memory addresses of the objects. In other words, the array “refers to” the objects instead of containing the objects.

You can create an array of references to seven `Employee` objects as follows:

```
Employee[] empArray = new Employee[7];
```

This statement reserves enough computer memory for the references to seven `Employee` objects named `empArray[0]` through `empArray[6]`. It does not actually construct those `Employees`; all the references are initialized to `null`. To create objects to fill the array, you must call the `Employee()` constructor seven times.

If the `Employee` class contains a default constructor, you can use the following loop to call the constructor seven times:

```
for(int x = 0; x < empArray.Length; ++x)
    empArray[x] = new Employee();
```

As `x` varies from 0 through 6, each of the seven `empArray` objects is constructed.

If you want to use a nondefault constructor, you must provide appropriate arguments. For example, if the `Employee` class contains a constructor with an `int` parameter, you can write the following to fill three of the seven array elements:

```
Employee[] empArray = {new Employee(123),
    new Employee(234), new Employee(345)};
```

To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot-method. For example, to set all seven `Employee IdNumber` properties to 999, you can write the following:

```
for(int x = 0; x < empArray.Length; ++x)
    empArray[x].IdNumber = 999;
```

398

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects

In the chapter “Using Arrays,” you learned about using the `System.Array` class’s built-in `BinarySearch()` and `Sort()` methods with simple data types such as `int`, `double`, and `string`. The `Sort()` method accepts an array parameter and arranges its elements in descending order. The `BinarySearch()` method accepts a sorted array and a value that it attempts to match in the array.

A complication arises when you consider searching or sorting arrays of objects you create. When you create and sort an array of simple data items, there is only one type of value to consider, and the order is based on the Unicode value of that item. The classes that support simple data items each contain a method named `CompareTo()`, which provides the details of how the basic data types compare to each other. In other words, they define comparisons such as “2 is more than 1” and “B is more than A.” The `Sort()` and `BinarySearch()` methods use the `CompareTo()` method for the current type of data being sorted. In other words, `Sort()` uses the `Int32` version of `CompareTo()` when sorting integers and the `Char` version of `CompareTo()` when sorting characters.



You learned about the built-in data type class names in Chapter 2; they are summarized in Table 2-1.



You have been using the `String` class (and its `string` alias) throughout this book. The class also contains a `CompareTo()` method that you first used in Chapter 2.

When you create a class in which comparisons will be made between objects, you must tell the compiler which field to use when making those comparisons. For example, you logically might sort an organization’s `Employee` objects by ID number, salary, department number, last name, hire date, or any field contained in the class. To tell C# which field to use for placing `Employee` objects in order, you must create an interface.

An **abstract method** has no method statements; an interface contains abstract methods. In the next chapter, you learn more about abstract methods. For now, understand that an **interface** is

a data type, like a class, except it is a collection of empty, abstract methods (and perhaps other members) that can be used by any class, as long as the class provides definitions to override the interface's do-nothing method definitions. Unlike a class, an interface cannot contain instance fields, and you cannot create objects from it.

A class that implements an interface must override the interface's methods. When a method **overrides** another, it takes precedence over the method, and is used instead of the original version. In other words, the methods in an interface are empty, and any class that uses them must contain a new version that provides the details and is used instead. Interfaces define named behaviors that classes must implement, so that all classes can use the same method names but use them appropriately for the class. In this way, interfaces provide for polymorphism—the ability of different objects to use the same method names but act appropriately based on the context.



When a method overrides another, it has the same signature as the method it overrides. When methods are overloaded, they have different signatures. You will learn more about overriding methods and abstract methods and classes in the next chapter.

C# supports many interfaces. You can identify an interface name by its initial letter *I*. For example, C# contains an `IComparable` interface, which contains the definition for the `CompareTo()` method that compares one object to another and returns an integer. Figure 9-39 shows the definition of `IComparable`. The `CompareTo()` method accepts a parameter `Object` but does not contain any statements; you must provide an implementation for this method in classes you create if you want the objects to be comparable. That is, the `CompareTo()` method is abstract and you must determine the meanings of *greater than*, *less than*, and *equal to* for the class.

```
interface IComparable
{
    int CompareTo(Object o);
}
```

Figure 9-39 The `IComparable` interface

When you create a class whose instances will likely be compared by clients:

- You must include a single colon and the interface name `IComparable` after the class name.
- You must write a method that contains the following header:

```
int IComparable.CompareTo(Object o)
```



`Object` is a class—the most generic of all classes. Every `Employee` object you create is not only an `Employee` but also an `Object`. By using the type `Object` as a parameter, the `CompareTo()` method can accept anything. You will learn more about the `Object` class in the next chapter.

To work correctly in methods such as `BinarySearch()` and `Sort()`, the `CompareTo()` method you create for your class must return an integer value. Table 9-2 shows the return values that every version of `CompareTo()` should provide.

Return Value	Meaning
Negative	This instance is less than the compared object.
Zero	This instance is equal to the compared object.
Positive	This instance is greater than the compared object.

Table 9-2 Return values of `IComparable.CompareTo()` method

When you create a class that contains an `IComparable.CompareTo()` method, the method is an instance method and receives a `this` reference to the object used to call it. A second object is passed to the method; within the method, you first must convert, or cast, the passed object to the same type as the calling object's class and then compare the corresponding fields you want from the `this` object and the passed object. For example, Figure 9-40 shows an `Employee` class that contains a shaded `CompareTo()` method and compares `Employee` objects based on the contents of their `idNumber` fields.

```
class Employee : IComparable
{
    public int IdNumber {get; set;}
    public double Salary {get; set;}
    int IComparable.CompareTo(Object o)
    {
        int returnVal;
        Employee temp = (Employee)o;
        if(this.IdNumber > temp.IdNumber)
            returnVal = 1;
        else
            if(this.IdNumber < temp.IdNumber)
                returnVal = -1;
            else
                returnVal = 0;
        return returnVal;
    }
}
```

Figure 9-40 `Employee` class using `IComparable` interface

The `Employee` class in Figure 9-40 uses a colon and `IComparable` in its class header to indicate an interface. The shaded method is an instance method; that is, it “belongs” to an `Employee` object.

When another `Employee` is passed in as `Object o`, it is cast as an `Employee` and stored in the `temp` variable. The `idNumber` values of the `this Employee` and the passed `Employee` are compared, and one of three integer values is returned.

For example, if you declare two `Employee` objects named `worker1` and `worker2`, you can use the following statement:

```
int answer = worker1.CompareTo(worker2);
```

Within the `CompareTo()` method in the `Employee` class, `worker1` would be “this” `Employee`—the controlling `Employee` in the method (the invoking `Employee` object). The `temp Employee` would be `worker2`. If, for example, `worker1` had a higher ID number than `worker2`, the value of `answer` would be 1.

Figure 9-41 shows a program that uses the `Employee` class. The program declares an array of five `Employee` objects with different ID numbers and all with salaries of 0; the ID numbers are purposely out of order to demonstrate that the `Sort()` method works correctly. The program also declares a `seekEmp` object with an ID number of 222. The program sorts the array, displays the sorted elements, then finds the array element that matches the `seekEmp` object. Figure 9-42 shows the program execution.

```
using System;
using static System.Console;
class ComparableEmployeeArray
{
    static void Main()
    {
        Employee[] empArray = new Employee[5];
        int x;
        for(x = 0; x < empArray.Length; ++x)
            empArray[x] = new Employee();
        empArray[0].IdNumber = 333;
        empArray[1].IdNumber = 444;
        empArray[2].IdNumber = 555;
        empArray[3].IdNumber = 111;
        empArray[4].IdNumber = 222;
        Employee seekEmp = new Employee();
        seekEmp.IdNumber = 222;
        Array.Sort(empArray);
        WriteLine("Sorted employees:");
        for(x = 0; x < empArray.Length; ++x)
            WriteLine("Employee #{0}: {1} {2}", x,
                empArray[x].IdNumber, empArray[x].Salary.ToString("C"));
        x = Array.BinarySearch(empArray, seekEmp);
        WriteLine("Employee #{0} was found at position {1}",
            seekEmp.IdNumber, x);
    }
}
```

Figure 9-41 The `ComparableEmployeeArray` program

```
C:\C#\Chapter.09>ComparableEmployeeArray
Sorted employees:
Employee #0: 111 $0.00
Employee #1: 222 $0.00
Employee #2: 333 $0.00
Employee #3: 444 $0.00
Employee #4: 555 $0.00
Employee #222 was found at position 1
C:\C#\Chapter.09>
```

Figure 9-42 Output of the `ComparableEmployeeArray` program

Notice that the `seekEmp` object matches the `Employee` in the second array position based on the `idNumber` only—not the `salary`—because the `CompareTo()` method in the `Employee` class uses only `idNumber` values and not salaries to make comparisons. You *could* have written code that requires both the `idNumber` and `salary` values to match before returning a positive number.

TWO TRUTHS & A LIE

Declaring an Array of Objects

Assume that a working program contains the following array declaration:

```
BankAccount[] acctArray = new BankAccount[500];
```

1. This statement reserves enough computer memory for 500 `BankAccount` objects.
2. This statement constructs 500 `BankAccount` objects.
3. The valid subscripts for `acctArray` are 0 through 499.

The false statement is #2. This statement reserves space for 500 `BankAccount` objects but does not actually construct those objects; to do so, you must call the constructor 500 times.



You Do It

Creating an Array of Objects

In the next steps, you create an array of `Student` objects. You prompt the user for data to fill the array, and you sort the array by student ID number before displaying all the data.

1. Open the **CreateStudents2** program and immediately save it as **CreateStudents3**. Change the class name to **CreateStudents3**.
2. Delete all the existing statements in the `Main()` method, leaving the opening and closing curly braces. Between the braces, declare an array of eight `Student` objects. Also declare a variable to use as an array subscript and declare three variables that will temporarily hold a user's input data before `Student` objects are constructed.

```
Student[] students = new Student[8];
int x;
int id;
string name;
double gpa;
```

3. In a loop, call a `GetData()` method (which you will write shortly). Send the method out arguments so that you can retrieve values for variables that will hold an ID number, name, and grade point average. Then, in turn, send these three values to the `Student` constructor for each of the eight `Student` objects.

```
for(x = 0; x < students.Length; ++x)
{
    GetData(out id, out name, out gpa);
    students[x] = new Student(id, name, gpa);
}
```

4. Call the `Array.Sort()` method, sending it the student array. Then, one object at a time in a loop, call the `Display()` method that you wrote in the last set of steps.

```
Array.Sort(students);
WriteLine("Sorted List:");
for(x = 0; x < students.Length; ++x)
    Display(students[x]);
```

(continues)

(continued)

- Write the `GetData()` method. Its parameters are `out` parameters so that their values will be known to the calling method. The method simply prompts the user for each data item, reads it, and converts it to the appropriate type, if necessary.

```
static void GetData(out int id, out string name,
    out double gpa)
{
    string inString;
    Write("Please enter student ID number >> ");
    inString = ReadLine();
    int.TryParse(inString, out id);
    Write("Please enter last name for student {0} >> ",
        id);
    name = ReadLine();
    Write("Please enter grade point average >> ");
    inString = ReadLine();
    double.TryParse(inString, out gpa);
}
```

- After the header for the `Student` class, add a colon and `IComparable` so that objects of the class can be sorted:

```
class Student : IComparable
```

- Just before the closing curly brace for the `Student` class, add the `IComparable.CompareTo()` method that is required for the objects of the class to be sortable. The method will sort `Student` objects based on their ID numbers, so it returns 1, -1, or 0 based on `IdNumber` property comparisons. The method accepts an object that is cast to a `Student` object. If the `IdNumber` of the controlling `Student` object is greater than the argument's `IdNumber`, then the return value is set to 1. If the `IdNumber` of the controlling `Student` object is less than the argument's `IdNumber`, then the return value is -1. Otherwise, the return value is 0.

```
int IComparable.CompareTo(Object o)
{
    int returnVal;
    Student temp = (Student)o;
    if(this.IdNumber > temp.IdNumber)
        returnVal = 1;
    else
        if(this.IdNumber < temp.IdNumber)
            returnVal = -1;
        else
            returnVal = 0;
    return returnVal;
}
```

(continues)

(continued)

8. Save the file, and then compile and execute it. When prompted, enter any student IDs, names, and grade point averages you choose. The objects will be sorted and displayed in `iDNumber` order. Figure 9-43 shows a typical execution.

```

C:\C#\Chapter.09>CreateStudents3
Please enter student ID number >> 912
Please enter last name for student 912 >> Harris
Please enter grade point average >> 3.4
Please enter student ID number >> 634
Please enter last name for student 634 >> Lewis
Please enter grade point average >> 2.2
Please enter student ID number >> 610
Please enter last name for student 610 >> Johnson
Please enter grade point average >> 2.6
Please enter student ID number >> 377
Please enter last name for student 377 >> Lee
Please enter grade point average >> 4.0
Please enter student ID number >> 391
Please enter last name for student 391 >> Young
Please enter grade point average >> 3.9
Please enter student ID number >> 215
Please enter last name for student 215 >> Jenks
Please enter grade point average >> 3.8
Please enter student ID number >> 185
Please enter last name for student 185 >> Bower
Please enter grade point average >> 2.3
Please enter student ID number >> 478
Please enter last name for student 478 >> Cooper
Please enter grade point average >> 2.6
Sorted List:
  185 Bower      2.3
  215 Jenks     3.8
  377 Lee       4.0
  391 Young     3.9
  478 Cooper    2.6
  610 Johnson   2.6
  634 Lewis     2.2
  912 Harris    3.4

C:\C#\Chapter.09>_

```

Figure 9-43 Typical execution of the `CreateStudents3` program

Understanding Destructors

A **destructor** contains the actions you require when an instance of a class is destroyed. Most often, an instance of a class is destroyed when it goes out of scope. As with constructors, if you do not explicitly create a destructor for a class, C# automatically provides one.

To explicitly declare a destructor, you use an identifier that consists of a tilde (`~`) followed by the class name. You cannot provide any arguments when you call a destructor; it must have an empty parameter list. As a consequence, destructors cannot be overloaded; a class can have one destructor at most. Like a constructor, a destructor has no return type.

Figure 9-44 shows an `Employee` class that contains only one field (`idNumber`), a property, a constructor, and a (shaded) destructor. When you execute the `Main()` method in the `DemoEmployeeDestructor` class in Figure 9-45, you instantiate two `Employee` objects, each with its own `idNumber` value. When the `Main()` method ends, the two `Employee` objects go out of scope, and the destructor for each object is called. Figure 9-46 shows the output.

```
class Employee
{
    public int idNumber {get; set;}
    public Employee(int empID)
    {
        IdNumber = empID;
        WriteLine("Employee object {0} created", IdNumber);
    }
    ~Employee()
    {
        WriteLine("Employee object {0} destroyed!", IdNumber);
    }
}
```

Figure 9-44 Employee class with destructor

```
using static System.Console;
class DemoEmployeeDestructor
{
    static void Main()
    {
        Employee aWorker = new Employee(101);
        Employee anotherWorker = new Employee(202);
    }
}
```

Figure 9-45 The `DemoEmployeeDestructor` program

```
C:\C#\Chapter.09>DemoEmployeeDestructor
Employee object 101 created
Employee object 202 created
Employee object 202 destroyed!
Employee object 101 destroyed!

C:\C#\Chapter.09>
```

Figure 9-46 Output of `DemoEmployeeDestructor` program

The program in Figure 9-44 never explicitly calls the `Employee` class destructor, yet you can see from the output that the destructor executes twice. Destructors are invoked automatically; you cannot explicitly call one. Interestingly, the last object created is the first object destroyed; the same relationship would hold true no matter how many objects the program instantiated.



An instance of a class becomes eligible for destruction when it is no longer possible for any code to use it—that is, when it goes out of scope. The actual execution of an object's destructor might occur at any time after the object becomes eligible for destruction.

For now, you have little reason to create a destructor except to demonstrate how it is called automatically. Later, when you write more sophisticated C# programs that work with files, databases, or large quantities of computer memory, you might want to perform specific cleanup or close-down tasks when an object goes out of scope. Then you will place appropriate instructions within a destructor.

TWO TRUTHS & A LIE

Understanding Destructors

1. To explicitly declare a destructor, you use an identifier that consists of a tilde (~) followed by the class name.
2. You cannot provide any parameters to a destructor; it must have an empty argument list.
3. The return type for a destructor is always `void`.

The false statement is #3. Like a constructor, a destructor, has no return type.

Understanding GUI Application Objects

The objects you have been using in GUI applications, such as `Forms`, `Buttons`, and `Labels`, are objects just like others you have studied in this chapter. That is, they encapsulate properties and methods. When you start a Windows Forms application in the IDE and drag a `Button` onto a `Form`, a statement is automatically created to instantiate a `Button` object.

Figure 9-47 shows a new GUI application that a programmer started simply by dragging one Button onto a Form.

408

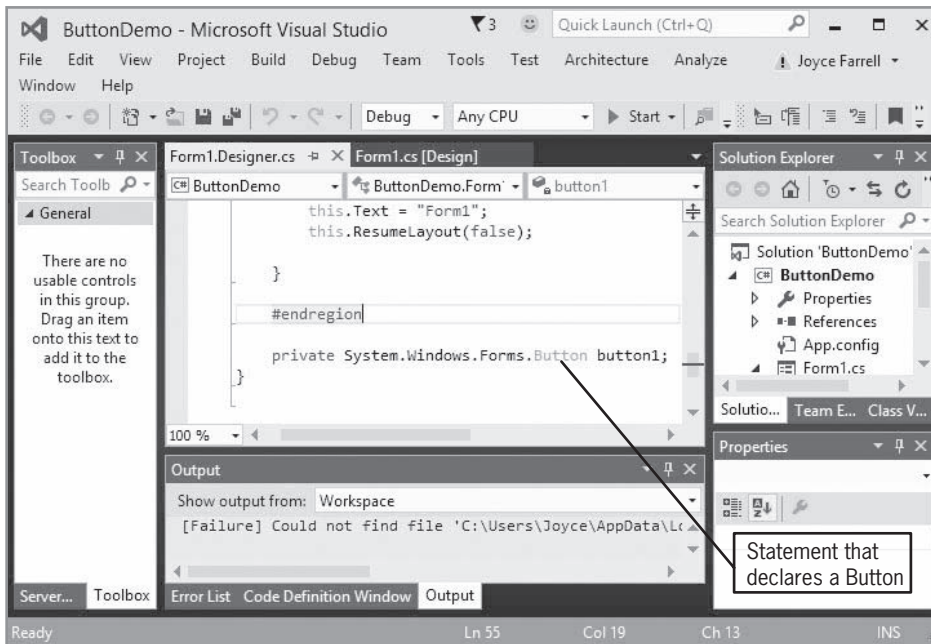


Figure 9-47 A Button on a Form

If you click on `Form1.Designer.cs` in the Solution Explorer and scroll down, you can display the code where the `Button` is declared. See Figure 9-48.

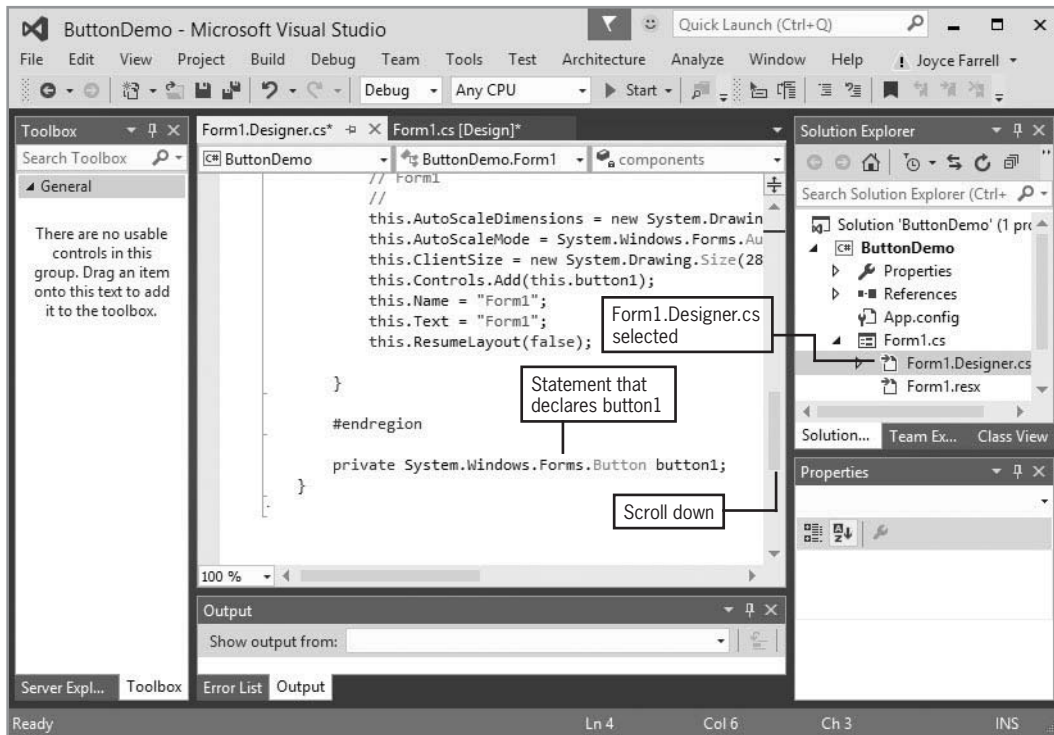


Figure 9-48 The automatically generated statement that declares `button1`

If you right-click `button1` in the code and select `Find All References` from the menu that appears, a list of every reference to `button1` in the project is displayed in the `Find Symbol Results` area at the bottom of the IDE. (You might need to resize the window to see the references clearly.) You can double-click each entry in the list in turn to locate all the places where code has been generated for `button1`. For example, several instances of `button1` are in a method named `InitializeComponent()`, as shown in Figure 9-49. The first reference to `button1` calls its constructor. You can also see how the `Button`'s `Name`, `Location`, `Size`, and other properties are set. You could have written these statements yourself, especially now that you know more about objects and how they are constructed, but it is easier to develop an application visually by dragging a `Button` onto a `Form` and allowing the IDE to create these statements for you.

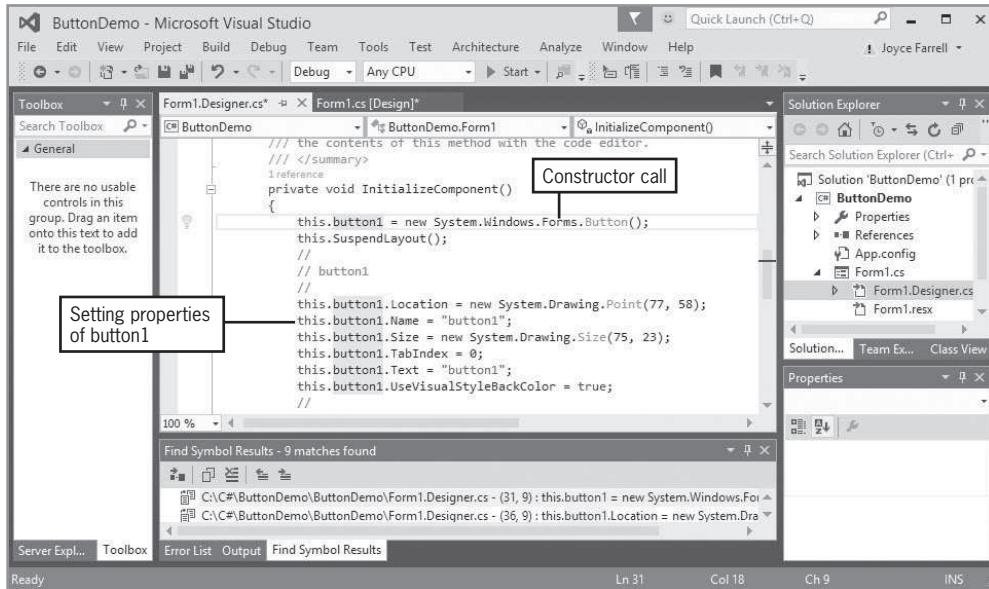


Figure 9-49 Some automatically generated `button1` references in the IDE

In Figure 9-49, the `this` reference preceding each occurrence of `button1` refers to the class (`Form1`) that contains these statements.



Now that you understand how to create properties in your own classes, you can understand the properties of GUI objects and how they were created for your use.

Chapter Summary

- Application classes contain a `Main()` method. You also can create classes from which you instantiate objects; these classes need not have a `Main()` method. The data components of a class are its instance variables (or fields) and methods. A program or class that instantiates objects of another prewritten class is a class client or class user.
- When you create a class, you must assign a name to it and determine what data and methods will be part of the class. A class header or class definition contains an optional access modifier, the keyword `class`, and an identifier. A class body is enclosed between curly braces. When you create a class, you usually declare instance variables to be `private` and instance methods to be `public`.

- When you create an object that is an instance of a class, you supply a type and an identifier, and you allocate computer memory for that object using the **new** operator. After an object has been instantiated, its **public** methods can be accessed using the object's identifier, a dot, and a method call. You can pass objects to methods just as you can pass simple data types.
- A property is a member of a class that provides access to a field. Properties have **set** accessors for setting an object's fields and **get** accessors for retrieving the stored values. As a shortcut, you can create an auto-implemented property when a field's **set** accessor should simply assign a value to the appropriate field, and when its **get** accessor should simply return the field.
- In most classes, fields are **private** and methods are **public**. This technique ensures that data will be used and changed only in the ways provided in the class's accessors. Occasionally, however, you need to create **public** fields or **private** methods.
- The **this** reference is passed to every instance method and property accessor in a class. You can explicitly refer to the **this** reference within an instance method or property, but usually you are not required to do so.
- A constructor is a method that instantiates an object. If you do not write a constructor, then each class is automatically supplied with a **public** constructor with no parameters. You can write your own constructor to replace the automatically supplied version. Any constructor you write must have the same name as its class, and constructors cannot have a return type. You can overload constructors and pass arguments to a constructor.
- An object initializer allows you to assign values to any accessible members or properties of a class at the time of instantiation without calling a constructor with parameters. Using object initializers allows you to create multiple objects with different initial assignments without having to provide multiple constructors to cover every possible situation. Additionally, using object initializers allows you to create objects with different starting values for different properties of the same data type.
- You can overload an operator by writing a method whose identifier is **operator**, followed by the operator being overloaded—for example, **operator+()** or **operator*()**. When you overload an operator, you should write statements that intuitively have the same meaning as the common use of the operator.
- You can declare arrays of references to objects. After doing so, you must call a constructor to instantiate each object. To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot-method. The **IComparable** interface contains the definition for the **CompareTo()** method; you override this method to tell the compiler how to compare objects.
- A destructor contains the actions you require when an instance of a class is destroyed. If you do not explicitly create a destructor for a class, C# automatically provides one. To explicitly declare a destructor, you use an identifier that consists of a tilde (**~**) followed by the class name. You cannot provide any parameters to a destructor; a class can have one destructor at most.

- GUI objects such as **Forms**, **Buttons**, and **Labels** are typical C# objects that encapsulate properties and methods. When you start a Windows Forms application and drag a component onto a **Form**, statements are automatically created by the IDE to instantiate the object and assign values to some of its properties.

Key Terms

To **instantiate** an object is to create it.

An **instantiation** of a class is a created object.

An **instance** of a class is one object, or one instantiation.

The **instance variables** of a class are the data components that exist separately for each instantiation.

Fields are instance variables within a class.

Instance methods are methods that are used with object instantiations.

A **class client** or **class user** is a program or class that instantiates objects of another prewritten class.

A **class header** or **class definition** describes a class; it contains an optional access modifier, the keyword `class`, and any legal identifier for the name of the class.

A **class access modifier** describes access to a class.

The **public** class access modifier means access to the class is not limited.

The **protected** class access modifier means access to the class is limited to the class and to any classes derived from the class.

The **internal** class access modifier means access is limited to the assembly to which the class belongs.

The **private** class access modifier means access is limited to another class to which the class belongs. In other words, a class can be **private** if it is contained within another class, and only the containing class should have access to the **private** class.

An **assembly** is a group of code modules compiled together to create an executable program.

Information hiding is a feature found in all object-oriented languages, in which a class's data is private and changed or manipulated only by its own methods.

Composition is the technique of using an object within another object.

A **has-a relationship** is the relationship created using composition, so-called because one class "has an" instance of another.

A **reference type** is a type that holds a memory address.

Value types hold a value; they include predefined types such as `int`, `double`, and `char`.

An **invoking object** is an object that calls an instance method.

A **property** is a member of a class that provides access to a field of a class; properties define how fields will be set and retrieved.

Accessors in properties specify how a class's fields are accessed.

The **set accessors** of a class allow assignment of values to fields using a property name.

The **get accessors** of a class allow retrieval of a field value by using a property name.

A **backing field** is a field that has a property coded for it.

A **read-only property** has only a **get** accessor, and not a **set** accessor.

The **getter** is another term for a class property's **get** accessor.

The **setter** is another term for a class property's **set** accessor.

Contextual keywords are identifiers that act like keywords in specific circumstances.

An **implicit parameter** is undeclared and gets its value automatically.

An **auto-implemented property** is one in which the code within the accessors is created automatically. The only action in the **set** accessor is to assign a value to the associated field, and the only action in the **get** accessor is to return the associated field value.

Automatic properties are auto-implemented properties.

The **this reference** is the reference to an object that is implicitly passed to an instance method of its class.

A **constructor** is a method that instantiates (creates an instance of) an object.

A **default constructor** is a parameterless constructor.

The **default value of an object** is the value initialized with a default constructor.

A **parameterless constructor** is one that takes no arguments.

A **constructor initializer** is a clause that indicates another instance of a class constructor should be executed before any statements in the current constructor body.

An **object initializer** allows you to assign values to any accessible members or properties of a class at the time of instantiation without calling a constructor with parameters.

An **abstract method** has no method statements.

An **interface** is a collection of abstract methods (and perhaps other members) that can be used by any class, as long as the class provides a definition to override the interface's do-nothing, or abstract, method definitions.

To **override** a method is to take precedence over another method version.

A **destructor** contains the actions required when an instance of a class is destroyed.

Review Questions

- An object is a(n) _____ of a class.
 - child
 - institution
 - instantiation
 - relative
- A class header or class definition can contain all of the following *except* _____.
 - an optional access modifier
 - the keyword `class`
 - an identifier
 - initial field values
- Most fields in a class are created with the _____ modifier.
 - `public`
 - `protected`
 - `new`
 - `private`
- Most methods in a class are created with the _____ modifier.
 - `public`
 - `protected`
 - `new`
 - `private`
- Instance methods that belong to individual objects are _____ `static` methods.
 - always
 - usually
 - occasionally
 - never
- To allocate memory for an object instantiation, you must use the _____ operator.
 - `mem`
 - `alloc`
 - `new`
 - `instant`
- Assume that you have created a class named `MyClass`. The header of the `MyClass` constructor can be _____.
 - `public void MyClass()`
 - `public MyClassConstructor()`
 - Either of these can be the constructor header.
 - Neither of these can be the constructor header.

8. Assume that you have created a class named `MyClass`. The header of the `MyClass` constructor can be _____.
- `public MyClass()`
 - `public MyClass (double d)`
 - Either of these can be the constructor header.
 - Neither of these can be the constructor header.
9. Assume that you have created a class named `DemoCar`. Within the `Main()` method of this class, you instantiate a `Car` object named `myCar` and the following statement executes correctly:
- ```
WriteLine("The Car gets {0} miles per gallon",
 myCar.ComputeMpg());
```
- Within the `Car` class, the `ComputeMpg()` method can be \_\_\_\_\_.
- public** and **static**
  - public** and nonstatic
  - private** and **static**
  - private** and nonstatic
10. Assume that you have created a class named `TermPaper` that contains a character field named `letterGrade`. You also have created a property for the field. Which of the following cannot be true?
- The property name is `letterGrade`.
  - The property is read-only.
  - The property contains a **set** accessor that does not allow a grade lower than 'C'.
  - The property does not contain a **get** accessor.
11. A **this** reference is \_\_\_\_\_.
- implicitly passed to nonstatic methods
  - implicitly passed to **static** methods
  - explicitly passed to nonstatic methods
  - explicitly passed to **static** methods
12. When you use an instance variable within a class's nonstatic methods, you \_\_\_\_\_ explicitly refer to the method's **this** reference.
- must
  - can
  - cannot
  - should (even though it is not required)

13. A class's default constructor \_\_\_\_\_.
- sets numeric fields to 0
  - is parameterless
  - both of these
  - none of these
14. Assume that you have created a class named `Chair` with a constructor defined as `Chair(int height)`. Which of the following overloaded constructors could coexist with the `Chair` constructor without ambiguity?
- `Chair(int legs)`
  - `Chair(int height, int legs)`
  - both of these
  - none of these
15. Which of the following statements correctly instantiates a `House` object if the `House` class contains a single constructor with the declaration `House(int bedrooms, double price)`?
- `House myHouse = new House();`
  - `House myHouse = new House(3, 125000.00);`
  - `House myHouse = House(4, 200000.00);`
  - two of these
16. You explicitly call a destructor \_\_\_\_\_.
- when you are finished using an object
  - when an object goes out of scope
  - when a class is destroyed
  - You cannot explicitly call a destructor.
17. In a program that creates five object instances of a class, the constructor executes \_\_\_\_\_ time(s) and the destructor executes \_\_\_\_\_ time(s).
- one; one
  - one; five
  - five; one
  - five; five



18. Suppose that you declare a class named `Furniture` that contains a `string` field named `woodType` and a conventionally named property with a `get` accessor. When you declare an array of 200 `Furniture` objects named `myChairs`, which of the following accesses the last `Furniture` object's wood type?
- `Furniture.Get(woodType[199])`
  - `myChairs[199].WoodType()`
  - `myChairs.WoodType[199]`
  - `myChairs[199].WoodType`
19. A collection of methods that can be used by any class, as long as the class provides a definition to override the collection's abstract definitions, is \_\_\_\_\_.
- a superclass
  - a polymorph
  - a perimeter
  - an interface
20. When you create a class and want to include the capability to compare its objects so they can use the `Array.Sort()` or `Array.BinarySearch()` method, you must \_\_\_\_\_.
- include at least one numeric field within the class
  - write a `CompareTo()` method for the class
  - be careful not to override the existing `IComparable.CompareTo()` method
  - Two of these are true.

## Exercises



### Programming Exercises

For each of the following programming exercises, you may choose to write a console-based or GUI application, or both.

- Create an application named `TestHockeyPlayer` that instantiates and displays a `HockeyPlayer` object. The `HockeyPlayer` class contains fields for a player's name (a string), jersey number (an integer), and goals scored (an integer).

2. Create an application named **TestClassifiedAd** that instantiates and displays a **ClassifiedAd** object. A **ClassifiedAd** has fields for a category (for example, *Used Cars*), a number of words, and a price. Include properties that contain **get** and **set** accessors for the category and number of words, but only a **get** accessor for the price. The price is calculated at nine cents per word.
3. Create an application named **SalesTransactionDemo** that declares several **SalesTransaction** objects and displays their values and their sum. The **SalesTransaction** class contains fields for a salesperson name, sales amount, and commission and a **readonly** field that stores the commission rate. Include three constructors for the class. One constructor accepts values for the name, sales amount, and rate, and when the sales value is set, the constructor computes the commission as sales value times commission rate. The second constructor accepts a name and sales amount, but sets the commission rate to 0. The third constructor accepts a name and sets all the other fields to 0. An overloaded **+** operator adds the sales values for two **SalesTransaction** objects.
4. Create a program named **TilingDemo** that instantiates an array of 10 **Room** objects and demonstrates its methods. The **Room** constructor requires parameters for length and width fields; use a variety of values when constructing the objects. The **Room** class also contains a field for floor area of the **Room** and number of boxes of tile needed to tile the room. Both of these values are computed by calling **private** methods. A room requires one box of tile for every 12 full square feet plus a box for any partial square footage over 12, plus one extra box for waste from irregular cuts. In other words, a 122 square foot room requires 12 boxes—10 boxes for the first 120 square feet, 1 box for the 2 extra square feet over 120, and 1 box for waste. Include read-only properties to get a **Room**'s values.
5. Create an application named **CarDemo** that declares at least two **Car** objects and demonstrates how they can be incremented using an overloaded **++** operator. Create a **Car** class that contains a model and a value for miles per gallon. Include two overloaded constructors. One accepts parameters for the model and miles per gallon; the other accepts a model and sets the miles per gallon to 20. Overload a **++** operator that increases the miles per gallon value by 1. The **CarDemo** application creates at least one **Car** using each constructor and displays the **Car** values both before and after incrementation.
6. a. Create a program named **TaxPayerDemo** that declares an array of 10 **Taxpayer** objects. Prompt the user for data for each object and display the 10 objects. Data fields for **Taxpayer** objects include the Social Security number (use a string for the type, but do not use dashes within the Social Security number), the yearly gross income, and the income tax owed. Include a property with **get** and **set** accessors for the first two data fields, but make the tax owed a read-only property.

The tax should be calculated whenever the income is set. Assume that the tax rate is 15 percent for incomes under \$30,000 and 28 percent for incomes that are \$30,000 or higher.

- b. Create a program named **TaxpayerDemo2** so that after the 10 **Taxpayer** objects are displayed, they are sorted in order by the amount of tax owed and displayed again. Modify the **Taxpayer** class so its objects are comparable to each other based on tax owed.
7. Create an application named **ShirtDemo** that declares several **Shirt** objects and includes a display method to which you can pass different numbers of **Shirt** objects in successive method calls. The **Shirt** class contains auto-implemented properties for a material, color, and size.
8. Create a program named **ConferencesDemo** for a hotel that hosts business conferences. Allows a user to enter data about five **Conference** objects and then displays them in order of attendance from smallest to largest. The **Conference** class contains fields for the **Conference** group name, starting date (as a string), and number of attendees. Include properties for each field. Also, include an **IComparable.CompareTo()** method so that **Conference** objects can be sorted.
9.
  - a. Create a program named **RelativesList** that declares an array of at least 12 **Relative** objects and prompts the user to enter data about them. The **Relative** class includes auto-implemented properties for the **Relative**'s name, relationship to you (for example, *aunt*), and three integers that together represent the **Relative**'s birthday—month, day, and year. Display the **Relative** objects in alphabetical order by first name.
  - b. Create a **RelativesBirthday** program that modifies the **RelativesList** program so that after the alphabetical list is displayed, the program prompts the user for a specific **Relative**'s name and the program returns the **Relative**'s relationship and birthday. Display an appropriate message if the relative requested by the user is not found.
10.
  - a. Write a program named **DemoJobs** for Harold's Home Services. The program should instantiate several **Job** objects and demonstrate their methods. The **Job** class contains four data fields—description (for example, "wash windows"), time in hours to complete (for example, 3.5), per-hour rate charged (for example, \$25.00), and total fee (hourly rate times hours). Include properties to get and set each field except the total fee—that field will be read-only, and its value is calculated each time either the hourly fee or the number of hours is set. Overload the + operator so that two **Jobs** can be added. The sum of two **Jobs** is a new **Job** containing the descriptions of both original **Jobs** (joined by *and*), the sum of the time in hours for the original **Jobs**, and the average of the hourly rate for the original **Jobs**.

b. Harold has realized that his method for computing the fee for combined jobs is not fair. For example, consider the following:

- His fee for painting a house is \$100 per hour. If a job takes 10 hours, he earns \$1000.
- His fee for dog walking is \$10 per hour. If a job takes 1 hour, he earns \$10.
- If he combines the two jobs and works a total of 11 hours, he earns only the average rate of \$55 per hour, or \$605.

Devise an improved, weighted method for calculating Harold's fees for combined **Jobs** and include it in the overloaded **operator+(C)** method. Write a program named **DemoJobs2** that demonstrates all the methods in the class work correctly.

11. a. Write a **FractionDemo** program that instantiates several **Fraction** objects and demonstrates that their methods work correctly. Create a **Fraction** class with fields that hold a whole number, a numerator, and a denominator. In addition:
- Create properties for each field. The **set** accessor for the denominator should not allow a 0 value; the value defaults to 1.
  - Add three constructors. One takes three parameters for a whole number, numerator, and denominator. Another accepts two parameters for the numerator and denominator; when this constructor is used, the whole number value is 0. The last constructor is parameterless; it sets the whole number and numerator to 0 and the denominator to 1. (After construction, **Fractions** do not have to be reduced to proper form. For example, even though 3/9 could be reduced to 1/3, your constructors do not have to perform this task.)
  - Add a **Reduce()** method that reduces a **Fraction** if it is in improper form. For example, 2/4 should be reduced to 1/2.
  - Add an **operator+(C)** method that adds two **Fractions**. To add two fractions, first eliminate any whole number part of the value. For example, 2 1/4 becomes 9/4 and 1 3/5 becomes 8/5. Find a common denominator and convert the fractions to it. For example, when adding 9/4 and 8/5, you can convert them to 45/20 and 32/20. Then you can add the numerators, giving 77/20. Finally, call the **Reduce()** method to reduce the result, restoring any whole number value so the fractional part of the number is less than 1. For example, 77/20 becomes 3 17/20.
  - Include a function that returns a **string** that contains a **Fraction** in the usual display format—the whole number, a space, the numerator, a slash (/), and a denominator. When the whole number is 0, just the **Fraction** part of the value should be displayed (for example, 1/2 instead of 0 1/2). If the numerator is 0, just the whole number should be displayed (for example, 2 instead of 2 0/3).

- b. Add an **operator\***(`)` method to the **Fraction** class created in Exercise 11a so that it correctly multiplies two **Fractions**. The result should be in proper, reduced format. Demonstrate that the method works correctly in a program named **FractionDemo2**.
- c. Write a program named **FractionDemo3** that includes an array of four **Fractions**. Prompt the user for values for each. Display every possible combination of addition results and every possible combination of multiplication results for each **Fraction** pair (that is, each type will have 16 results).



## Debugging Exercises

1. Each of the following files in the `Chapter.09` folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, `DebugNine1.cs` will become `FixedDebugNine1.cs`.
  - a. `DebugNine1.cs`
  - b. `DebugNine2.cs`
  - c. `DebugNine3.cs`
  - d. `DebugNine4.cs`



## Case Problems

1. In previous chapters, you have created programs for the Greenville Idol competition. Now create a **Contestant** class with the following characteristics:
  - The **Contestant** class contains public static arrays that hold talent codes and descriptions. Recall that the talent categories are *Singing*, *Dancing*, *Musical instrument*, and *Other*.
  - The class contains an auto-implemented property that holds a contestant's name.
  - The class contains fields for a talent code and description. The **set** accessor for the code assigns a code only if it is valid. Otherwise, it assigns *I* for *Invalid*. The talent description is a read-only property that is assigned a value when the code is set.

Modify the **GreenvilleRevenue** program so that it uses the **Contestant** class and performs the following tasks:

- The program prompts the user for the number of contestants in this year's competition; the number must be between 0 and 30. The program continues to prompt the user until a valid value is entered.
  - The expected revenue is calculated and displayed. The revenue is \$25 per contestant.
  - The program prompts the user for names and talent codes for each contestant entered. Along with the prompt for a talent code, display a list of the valid categories.
  - After data entry is complete, the program displays the valid talent categories and then continuously prompts the user for talent codes and displays the names of all contestants in the category. Appropriate messages are displayed if the entered code is not a character or a valid code.
2. In previous chapters, you have created programs for Marshall's Murals. Now create a **Mural** class with the following characteristics:

- The **Mural** class contains public static arrays that hold mural codes and descriptions. Recall that the mural categories are *Landscape*, *Seascape*, *Abstract*, *Children's*, and *Other*.
- The class contains an auto-implemented property that holds a mural customer's name.
- The class contains fields for a mural code and description. The **set** accessor for the code assigns a code only if it is valid. Otherwise, it assigns *I* for *Invalid*. The mural description is a read-only property that is assigned a value when the code is set.

Modify the **Marshall'sRevenue** program so that it uses the **Mural** class and performs the following tasks:

- The program prompts the user for the month, the number of interior murals scheduled, and the number of exterior murals scheduled. In each case, the program continues to prompt the user until valid entries are made.
- The expected revenue is calculated and displayed. Recall that exterior murals cannot be painted in December through February. Also recall that exterior murals are \$750 in all months except April, May, September, and October, when they are \$699. Interior murals are \$500 except during July and August, when they are \$450.
- The program prompts the user for names and mural codes for interior and exterior murals. Along with the prompt for a mural code, display a list of the valid categories.
- After data entry is complete, the program displays the valid mural categories and then continuously prompts the user for codes and displays the names of all customers ordering murals in the category. Appropriate messages are displayed if the entered code is not a character or a valid code.