Chapter 10

Methods, Properties, and Operators

Methods, Properties, and Operators

Objectives

After completing this unit you will be able to:

- **Explain how methods are defined and used, how parameters are passed to and from methods, and how the same method name can be overloaded, with different versions having different parameter lists.**
- **Implement methods in C# that take a variable number of parameters.**
- **Use the C# get/set (property syntax) methods for accessing data**
- **Overload operators in C#, which makes invoking certain methods more natural and intuitive.**

Static and Instance Methods

- **We have seen that classes can have different kinds of members, including fields, constants, and *methods*.**
  - A method implements behavior that can be performed by an object or a class.
  - Ordinary methods, sometimes called **instance methods**, are invoked through an object instance.

```
Account acc = new Account();
acc.Deposit(25);
```

  - Static methods are invoked through a class, and do not depend upon the existence of any instances.

```
int sum = SimpleMath.Add(5, 7);
```

Method Parameters

- **Methods have a list of parameters, which may be empty.**
  - Methods either return a value or have a **void** return.
  - Multiple methods may have the same name, so long as they have different signatures, a feature known as **method overloading**.
  - Methods have the same signature if they have the same number of parameters, and these parameters have the same types and modifiers (such as **ref** or **out**).
- **The return type does not contribute to defining the signature of a method. By default, parameters are value parameters, which means a copy is made of the parameter.**
  - The keyword **ref** designates a **reference** parameter, in which case the parameter inside the method and the corresponding actual argument refer to the same object.
  - The keyword **out** refers to an **output** parameter, which is the same as a reference parameter, except that on the calling side, the parameter need not be assigned prior to the call.
  - We will study parameter passing and method overloading in more detail later in this chapter.

No “Freestanding” Functions in C#

- **In C# *all* functions are methods and so are associated with a class.**
  - There is no such thing as a freestanding function, as in C and C++.
  - “All functions are methods” is rather similar to “everything is an object” and reflects the fact that C# is a pure object-oriented language.
  - The advantage of all functions being methods is that classes become a natural organizing principle. Methods are nicely grouped together.

Classes with All Static Methods

- **Sometimes part of the functionality of your system may not be tied to any data but may be purely functional in nature.**
- **In C# you would organize such functions into classes that have all static methods and no fields.**
- **The program *TestSimpleMath/Step1* provides an elementary example.**

```
// SimpleMath.cs

public class SimpleMath
{
   public static int Add(int x, int y)
   {
      return x + y;
   }
   public static int Multiply(int x, int y)
   {
      return x * y;
   }
}
```

Parameter Passing

- **Programming languages have different mechanisms for passing parameters.**
- **In the C family of languages the standard is "call by value."**
  - This means that the actual data values themselves are passed to the method.
  - Typically, these values are pushed onto the stack, and the called function obtains its own independent copy of the values.
  - Any changes made to these values will not be propagated back to the calling program. C# provides this mechanism of parameter passing as the default, but C# also supports "reference" parameters and "output" parameters.
  - In this section we will examine all three of these mechanisms, and we will also look at the ramifications of passing class and struct data types.

Parameter Terminology

- **Storage is allocated on the stack for method parameters.**
  - This storage area is known as the **activation record**.
  - It is popped when the method is no longer active.
  - The **formal parameters** of a method are the parameters as seen within the method.
  - They are provided storage in the activation record.
  - The **actual parameters** of a method are the expressions between commas in the parameter list of the method call.

```
int sum = SimpleMath.Add(5, 7);
                          // actual parameters are
                          // 5 and 7
...

public static int Add(int x, int y)
{                         // formal parameters are
                          // x and y
   ...
}
```

Value Parameters

- **Parameter passing is the process of initializing the storage of the formal parameters by the actual parameters.**
- **The default method of parameter passing in C# is *call-by-value*, in which the values of the actual parameters are copied into the storage of the formal parameters.**
  - Call-by-value is "safe," because the method never directly accesses the actual parameters, only its own local copies.
- **But there are drawbacks to call-by-value:**
  - There is no direct way to modify the value of an argument. You may use the return type of the method, but that only allows you to pass one value back to the calling program.
  - There is overhead in copying a large object.
- **The overhead in copying a large object is borne when you pass a *struct* instance.**
  - If you pass a class instance, or an instance of any other reference type, you are passing only a reference and not the actual data itself.
  - This may sound like "call-by-reference," but what you are actually doing is passing a reference by value.
  - Later in this section we will discuss the ramifications of passing *struct* and *class* instances.

Reference Parameters

- **Consider a situation in which you want to pass more than one value back to the calling program.**
- **C# provides a clean solution through *reference parameters*.**
  - You declare a reference parameter with the **ref** keyword, which is placed before both the formal parameter and the actual parameter.
  - A reference parameter does not result in any copying of a value.
  - Instead, the formal parameter and the actual parameter refer to the same storage location.
  - Thus changing the formal parameter will result in the actual parameter changing, as both are referring to exactly the same storage location.

Reference Parameters (Cont'd)

- **The program *ReferenceMath* illustrates using *ref* parameters.**
  - The two methods **Add** and **Multiply** are replaced by a single method **Calculate**, which passes back two values as reference parameters.

```
// ReferenceMath.cs

public class ReferenceMath
{
   public static void Calculate(int x, int y,
                          ref int sum, ref int prod)
   {
      sum = x + y;
      prod = x * y;
   }
}
```

Reference Parameters (Cont'd)

- **Notice the use of the *ref* keyword in front of the third and fourth parameters. Here is the test program:**

```
// TestReferenceMath.cs

using System;

public class TestReferenceMath
{
   public static void Main(string[] args)
   {
      int sum = 0, product = 0;
      MultipleMath.Calculate(5, 7, ref sum,
         ref product);
      Console.WriteLine("sum = {0}", sum);
      Console.WriteLine("product = {0}", product);
   }
}
```

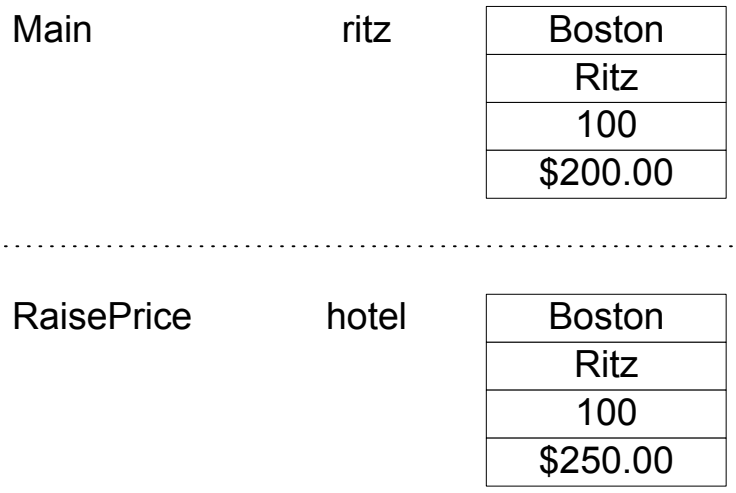
- **The *ref* keyword is used in front of the parameters.**
- **Variables must be initialized before they are used as reference parameters.**

Output Parameters

- **A reference parameter is used for two-way communication between the calling program and the called program, both passing data in and getting data out.**
- **Thus reference parameters must be initialized before use.**
  - In **TestReferenceMath.cs** (previous slide), we are only obtaining output, so initializing the variables only to assign new values is rather pointless.
  - C# provides for this case with **output parameters**.
  - Use the keyword **out** wherever you would use the keyword **ref**.
  - Then you do not have to initialize the variable before use.
  - Naturally, you could not use an **out** parameter inside the method; you can only assign it.
- **The program *OutputMath* illustrates the use of output parameters.**

Structure Parameters

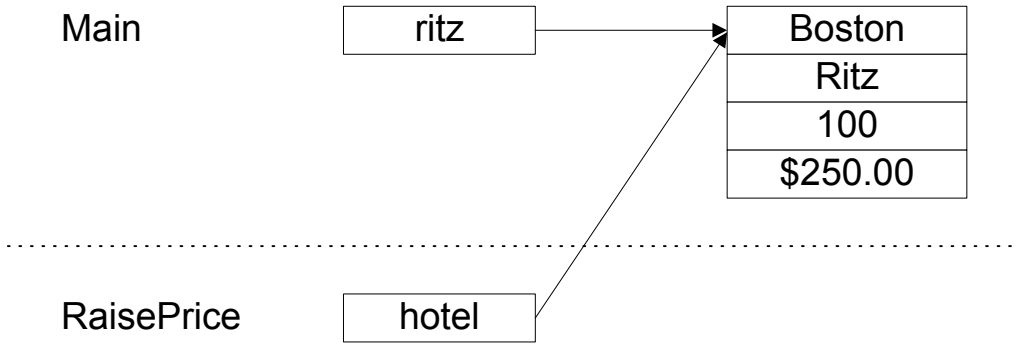
- **A struct is a value type, so if you pass a struct as a value parameter, the struct instance in the called method will be an independent copy of the struct in the calling method.**
- **The program *HotelStruct* illustrates passing an instance of a *Hotel* struct by value.**
- **The object *hotel* in the *RaisePrice* method is an independent copy of the object *ritz* in the Main method.**
  - This figure shows the values in both structures after the price has been raised for **hotel**.
  - Thus the change in price does not propagate back to **Main**.



  - The program **HotelStructRef** has the same struct definition, but the test program passes the **Hotel** instance by reference.
  - Now the change does propagate, as you would expect.

Class Parameters

- **A class is a reference type, so if you pass a class instance as a value parameter, the class instance in the called method will refer to the same object as the reference in the calling method.**
- **The program *HotelClass/Step1* illustrates passing an instance of a *Hotel* class by value.**
  - This figure illustrates how the **hotel** reference in the **RaisePrice** method refers to the same object as the **ritz** reference in **Main**.



  - Thus when you change the price in the **RaisePrice** method, the object in **Main** is the same object and shows the new price.

Method Overloading

- **In a traditional programming language such as C, you need to create unique names for all your methods.**
- **If methods do basically the same thing but only apply to different data types, it becomes tedious to create unique names.**
  - For example, suppose you have a **FindMax** method that can find the maximum of two **int** or two **long** or two **string**.
  - If we need to come up with a unique name for each method, we would have to create method names such as **FindMaxInt**, **FindMaxLong**, and **FindMaxString**.
- **In C#, as in other object-oriented languages such as C++ and Java, you may *overload* method names.**
  - That is, different methods can have the same name, if they have different **signatures**.
  - Two methods have the same signature if they have the same number of parameters, the parameters have the same data types, and the parameters have the same modifiers (none, **ref**, or **out**).
  - The return type does not contribute to defining the signature of a method.
  - So, in order to have two functions with the same name, there must be a difference in the number and/or types and/or modifiers of the parameters.

Method Overloading (Cont'd)

- **At runtime the compiler will resolve a given invocation of the method by trying to match up the actual parameters with formal parameters.**
  - A match occurs if the parameters match exactly or if they can match through an implicit conversion.
  - For the exact matching rules, consult the **C# Language Specification**.
- **The program *OverloadDemo* illustrates method overloading.**
  - The method **FindMax** is overloaded to take either **long** or **string** parameters.
  - The method is invoked three times, for **int**, **long**, and **string** parameters.
  - There is an exact match for the case of **long** and **string**.
  - The call with **int** actual parameters can resolve to the **long** version, because there is an implicit conversion of **int** into **long**.
  - You may wish to review the discussion of conversions of data types at the end of Chapter 4.
- **We will cover the *string* data type and the *Compare* method in Chapter 11.**

Modifiers as Part of the Signature

- **It is important to understand that if methods have identical types for their formal parameters, but differ in a modifier (none, *ref*, or *out*), then the methods have different signatures.**
- **The program *OverloadHotel* provides an illustration.**
  - We have two **RaisePrice** methods.
  - In the first method, the hotel is passed as a value parameter.
  - In the second version, the hotel is passed as a reference parameter.
  - These methods have different signatures.

Variable Length Parameter Lists

- **Our *FindMax* methods in the previous section were very specific with respect to the number of parameters—there were always exactly two parameters.**
- **Sometimes you may want to be able to work with a variable number of parameters, for example, to find the maximum of two, three, four, or more numbers.**
- **C# provides the *params* keyword, which you can use to indicate that an array of parameters is provided.**
  - Sometimes you may want to provide both a general version of your method that takes a variable number of parameters and also one or more special versions that take an exact number of parameters.
  - The special version will be called in preference, if there is an exact match. The special versions are more efficient.
- **The program *VariableMax* illustrates a general *FindMax* method that takes a variable number of parameters.**
  - There is also a special version that takes two parameters.
  - Each method prints out a line identifying itself, so you can see which method takes precedence.

Properties

- **The encapsulation principle leads us to typically store data in private fields and to provide access to this data through public accessor methods that allow us to set and get values.**
  - For example, in the **Account** class we used as an illustration in Chapter 8, we provided a method **GetBalance** to access the private field **balance**.
  - You don't need any special syntax; you can simply provide methods and call these methods what you want, typically **GetXXX** and **SetXXX**.
- **C# provides a special property syntax that simplifies user code.**
- **Rather than using methods, you can simply use an object reference followed by a dot followed by a property name.**
  - Here are some hypothetical examples of a **Balance** property (that we assume for the sake of argument is both read/write) of a hypothetical **Account** class.
  - We show in comments the corresponding method code.

Properties Example

```
Account acc = new Account();
decimal bal;
bal = acc.Balance;        // bal =
acc.GetBalance();
acc.Balance = 100m;       // acc.SetBalance(100m);
acc.Balance += 1m; //
acc.SetBalance(acc.GetBalance() + 1m);
```

- **As you can see, the syntax using the property is a little more concise.**
- **Properties were popularized in Visual Basic, and are now part of .NET and available in selected other .NET languages, such as C#.**
  - The program **AccountProperty** illustrates implementing and using several properties, **Balance**, **Id**, and **Owner**.
  - The first two properties are read-only (only **get** defined), and the third property is read/write (both **get** and **set**).
  - It is also possible to have a write-only property (only **set** defined).
- **The next page shows the code for the *Account* class, where the properties are defined.**
  - Notice the syntax and the special C# keyword **value**.

Properties Example (Cont'd)

```
// Account.cs

public class Account
{
   private int id;
   private static int nextid = 1;
   private decimal balance;
   private string owner;
   public Account(decimal balance, string owner)
   {
      this.id = nextid++;
      this.balance = balance;
      this.owner = owner;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
   }
   public void Withdraw(decimal amount)
   {
      balance -= amount;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
   public int Id
   {
      get
      {
         return id;
      }
   }
```

Properties Example (Cont'd)

```
   public string Owner
   {
      get
      {
         return owner;
      }
      set
      {
         owner = value;
      }
   }
}
```

Operator Overloading

- **Another kind of syntactic simplification that can be provided in C# is *operator overloading*.**
- **The idea is that certain method invocations can be implemented more concisely using operators rather than method calls.**
  - Suppose we have a class **Matrix** that has static methods to add and multiply matrices.
  - Using methods, we could write a matrix expression like this:

```
Matrix a, b, c, d;
// code to initialize the object references
d = Matrix.Multiply(a, (Matrix.Add(b, c));
```

  - If we overload the operators + and *, we can write this code more succinctly:

```
d = a * (b + c);
```

Operator Overloading (Cont'd)

- **You cannot create a brand new operator, but you can overload many of the existing C# operators to be an alias for a static method.**
  - For example, given the static method **Add** in the **Matrix** class ...

```
class Matrix
{
...
   public static Matrix Add(Matrix x, Matrix y)
   {
```

  - ... you could write instead:

```
   public static Matrix operator+(Matrix x,
                                  Matrix y)
```

- **All of the rest of the class implementation code stays the same, and you can then use operator notation in client code. Operator declarations, such as *operator*+ shown above, must obey the following rules:**
  - Operators must be **public** and **static**, and may not have any other modifiers.
  - Operators take only value parameters, and not reference or output parameters.
  - Operators must have a signature that differs from the signatures of all other operators in the class.

Operator Overloading (Cont'd)

- **There are three categories of operators that can be overloaded.**
  - The table shows the unary and binary operators that can be overloaded.
  - A third category of operators is user-defined conversions, which will be discussed in Chapter 15.

Type	Operators
Unary	+ - ! ~ ++ -- true false
Binary	+ - * / % & | ^ << >> == != > < >= <=

  - If you overload a binary operator **op**, the corresponding compound assignment operator **op=** will be overloaded for you by the compiler. For example, if you overload + you will automatically have an overload of +=.
- **The relational operators must be overloaded in pairs:**
  - operator== and operator!=
  - operator> and operator<
  - operator>= and operator<=.

Sample Program

- **As an illustration of operator overloading, consider the program *TestClock*, which has a class *Clock* that does "clock arithmetic."**
  - The legal values of **Clock** are integers between 1 and 12 inclusive.
  - Addition is performed modulo 12. Thus 9 + 7 is 16 modulo 12, or 4.
  - We overload the plus operator to do this special kind of addition operation.
  - We have two different versions of the plus operator. One adds two **Clock** values, and the other adds a **Clock** and an **int**.
  - In the test program note that we are able to use += even though we have not explicitly provided such an overload. The compiler automatically furnishes this overload for us by virtue of our overloading +.

Operator Overloading in the Class Library

- **Although you may rarely have occasion to overload operators in your own classes, you will find that a number of classes in the .NET Framework Class Library make use of operator overloading.**
- **In Chapter 11 you will see how + is used for concatenation of strings.**
- **In Chapter 19 you will see how += is used for adding an event handler to an event.**

Summary

- **In this chapter we examined a number of features of methods.**
- **In C# there is no such thing as a freestanding function.**
- **All functions are tied to classes and are called methods.**
- **If you do not care about class instances, you can implement a class that has only static methods.**
- **By default, parameters are passed by value, but C# also supports reference parameters and out-put parameters.**
- **A method name can be overloaded, with different versions having different parameter lists.**
- **You can also implement methods in C# that take a variable number of parameters.**
- **C# provides a special property syntax for concisely invoking get/set methods for accessing data.**
- **You can overload operators in C#, a feature which makes the C# language inherently more extensible without requiring special coding in the compiler.**