Chapter 8

Classes

Classes

Objectives

After completing this unit you will be able to:

- **State the important role of classes in object-oriented programming.**
- **Use classes in C# for representing structured data, distinguish between objects and classes.**
- **Explain how C# classes support encapsulation through fields and methods in conjunction with the public and private access specifiers.**
- **Use the new operator to instantiate objects from classes.**
- **Describe the use of references in C# and explain the role of garbage collection.**
- **Use C# constructors to initialize objects.**
- **Understand the use of static members.**
- **Use const and readonly to specify constants in C# programs.**

Classes As Structured Data

- **C# defines primitive data types that are built in to the language, as discussed in Chapter 4.**
  - Data types such as **int**, **decimal**, and **bool** can be used to represent simple data.
  - C# provides the class mechanism to represent more complex forms of data.
  - Through a class, you can build up structured data out of simpler elements, which are called data members, or fields.
  - (See TestAccount\Step1.)

```
// Account.cs

public class Account
{
   public int Id;
   public decimal Balance;
}
```

  - **Account** is now a new data type. An account has an **Id** (e.g., 1) and a **Balance** (e.g., 100.00).

Classes and Objects

- **A class represents a "kind of," or type of, data.**
  - It is analogous to the built-in types like int and decimal.
  - A class can be thought of as a template from which individual instances can be created.
  - An instance of a class is called an object. Just as you can have several individual integers that are instances of int, you can have several accounts that are instances of Account.
  - The fields, such as Id and Balance in our example, are sometimes also called instance variables.

References

- **There is a fundamental distinction between the primitive data types and the extended data types that can be created using classes.**
  - When you declare a variable of a primitive data, you are allocating memory and creating the instance.

```
int x;     // 4 bytes of memory have been allocated
```

- **When you declare a variable of a class type (an object reference), you are only obtaining memory for a reference to an object of the class type.**
  - No memory is allocated for the object itself, which may be quite large.

```
Account acc;      // acc is a reference to an
                  // Account object
             // The object itself does not yet
             // exist
```

Instantiating and Using an Object

- **You instantiate an object by the *new* operator.**

```
acc = new Account();    // Account object now
                        // exists and acc is a
                        // reference to it
```

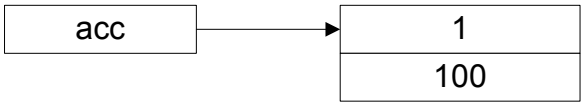
- **Once an object exists, you work with it, including accessing its fields and methods.**
  - Our simple **Account** class at this point has no methods, only two fields.
  - You access fields and methods using a dot.

```
acc.Id = 1;
acc.Balance = 100;      // Fields have now been
                        // assigned
Console.WriteLine("Account id {0} has balance {1}",
acc.Id, acc.Balance);
```

- **Example program**
  - TestAccount\Step1

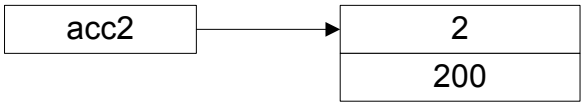
Assigning Object References

- **The figure shows the object reference *acc* and the data it refers to after the assignment:**



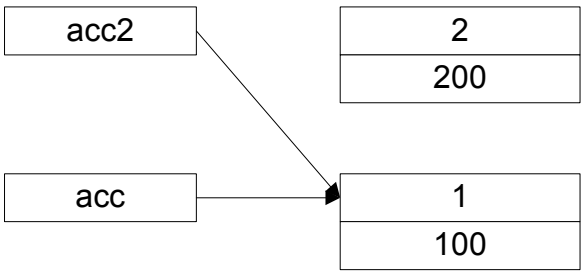

```
acc.Id = 1;
acc.Balance = 100;     // Fields have now been
                       // assigned
```

- **Now consider a second object variable referencing a second object, as illustrated in the figure, after the assignment:**




```
Account acc2 = new Account();
acc2.Id = 2;
acc2.Balance = 200;
```

- **When you assign an object variable, you are only assigning the reference; *there is no copying of data*. The figure shows both object references and their data after the assignment:**




```
acc2 = acc;  // acc2 now refers to same object acc
             // does
```

- **This is also known as "Shallow Copy".**

Garbage Collection

- **Through the assignment of a reference, an object may become orphaned.**
  - Such an orphan object (or "garbage") takes up memory in the computer, which can now never be referenced.
  - In the preceding figure the account with **Id** of 2 is now garbage.
  - The Common Language Runtime automatically reclaims the memory of unreferenced objects.
  - This process is known as *garbage collection*.
  - Garbage collection takes up some execution time, but it is a great convenience for programmers, helping to avoid a common program error known as a *memory leak*.

Sample Program

- **An illustration of assigning object references is provided in *TestAccount\Step2*.**
  - We print out the contents of **acc** and **acc2** before and after assigning **acc2** to refer to the same object as **acc**.
  - Once these two references refer to the same object, changing the value of a field will affect them both.
- **See *TestAccount\Step2*.**

Methods

- **Typically, a class will specify *behavior* as well as *data*. A class encapsulates data and behavior in a single entity. A method consists of**
  - An access specifier, typically **public** or **private**
  - A return type (can be **void** if the method does not return data)
  - A method name, which can be any legal C# identifier
  - A parameter list, enclosed by parentheses, which specifies data that is passed to the method (can be empty if no data is passed)
  - A method body, enclosed by curly braces, which contains the C# code that the method will execute

Method Syntax Example

```
public void Deposit(decimal amount)
{
   balance += amount;
}
```

- **In this example:**
  - The access of the member function is **public**,
  - The return type is **void** (no data is passed back),
  - The method name is **Deposit**,
  - The parameter list consists of a single parameter of type **decimal**, and
  - The body contains one line of code that adds the value that is passed in the parameter **amount** to the member variable **balance**.

Public and Private

- **Fields and methods of a C# class can be specified as *public* or *private*.**
  - The access qualifier is placed before each declaration.
  - Normally, you declare fields as **private**.
  - A private field can only be accessed from within the class, not from outside.

```
public class Account
{
   private int id;
   private decimal balance;
   // ...other members
}
```

Public and Private (Cont'd)

- **Methods may be declared as either *public* or *private*.**
  - Public methods are called from outside the class and are used to perform calculations and to manipulate the private data.
  - You may also provide public "accessor" methods to provide access to private fields. (Later, we will see another way to do this using *properties*.)

```
public decimal GetBalance()
{
   return balance;
}
public int GetId()
{
   return id;
}
```

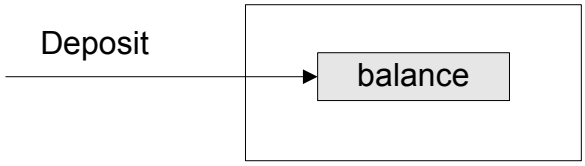
- **With this approach, all instances of *Account* will start out with the same initial values.**
  - We would like to find a way for the class to initialize its instance data appropriately, in a way that can be specified when the object is created.
  - This capability is provided in C# by constructors.
- **You may also have private methods, which can be thought of as "helper functions" for use within the class.**
  - Rather than duplicating code in several places, you may create a private method, which will be called wherever it is needed.

Abstraction

- **An abstraction captures the essential features of an entity, suppressing unnecessary details.**
  - There are many possible features of an account, but for our purposes, the only essential things are its id, its balance, and the operations of making a deposit and a withdrawal.
  - All instances of an abstraction share these common features.
  - Abstraction helps us deal with complexity.
- **Abstraction motivation:**
  - Consider the modeling of several different automobiles, e.g., sedan, pickup, and SUV.
  - All of the automobiles have elements of their interfaces in common, such as steering and brakes (although some of those elements may have different implementations).
  - It would save considerable coding (and later, maintenance effort) if the common elements could be gathered into one place, and the resulting abstraction used in each implementation.

Encapsulation

- **The implementation of an abstraction should be hidden from the rest of the system, or *encapsulated*.**
  - Objects have a public and a private side.
  - The public side is what the rest of the system knows, while the private side implements the public side.
  - The figure illustrates the public method **Deposit**, which operates on the private field **balance**.



- **Data itself is private and can only be accessed through methods with a public interface. Such encapsulation provides two kinds of protection:**
  - Internal data is protected from corruption.
  - Users of the object are protected from changes in the representation.
- **Done properly, encapsulation will reduce future maintenance effort, because implementation details may be changed without breaking existing client code.**

Initialization

- **Another important issue for classes is *initialization*.**
  - When an object is created, what initial values are assigned to the instance data?
  - A classical problem in programming is uninitialized variables.
  - When you run the program, you may get unpredictable results, depending on what happened to be in memory at the time you ran the program.
  - C# helps prevent such unpredictable behavior by performing a default initialization. Variables of numerical data types are initialized to 0.
- **In general you will want to perform your own initialization. There are two approaches.**
- **One is for the user of the class to perform the initialization.**
  - That is the approach followed in Steps 1 and 2 of the **TestAccount** program.
  - That approach was feasible because we made the fields public, and so the class user could initialize them.

Initialization with Constructors

- **A second approach is to provide initialization code in the class itself.**
  - You could assign instance data in the class definition, as illustrated in ***InitialAccount\Step1***.
- **With this approach, all instances of *Account* will start out with the same initial values.**
  - We would like to find a way for the class to initialize its instance data appropriately, in a way that can be specified when the object is created.
  - This capability is provided in C# by constructors.

Constructors (Cont'd)

- **Through a constructor, you can initialize individual objects in any way you wish. Besides initializing instance data, you can perform other appropriate initializations (e.g., open a file).**
- **A constructor is like a special method that is automatically called when an object is created via *new*. A constructor**
  - Has no return type
  - Has the same name as the class
  - Should usually have **public** access
  - May take parameters, which are passed when invoking **new**

```
public Account(int i, decimal bal)
{
   id = i;
   balance = bal;
}
```

- **In the calling program, you use *new* to instantiate object instances, and you pass desired values as parameters.**
- **This example is illustrated in *InitialAccount\Step2*.**

Default Constructor

- **If you do not define a constructor in your class, C# will implicitly create one for you, called the *default constructor*, which takes no arguments.**
  - The default constructor will assign instance data, using any assignments in the class definition.
- ***InitialAccount\Step1* provides an illustration.**
  - The default constructor is called when an object instance is created with new and no parameters. Add the following code to the **Step2** test program shown previously:

```
Account acc3 = new Account();
Console.WriteLine("balance of {0} is {1}",
acc3.GetId(), acc3.GetBalance());
```

- **While this worked fine in *Step1* when there was no explicit constructor at all, we now get a compiler error:**

```
error CS1501: No overload for method 'Account'
takes '0' arguments
```

- **In C# you may overload methods, including constructors, in which you have several methods with the same name but different argument lists.**
  - We can fix this problem by defining a second constructor with no arguments.
  - We may leave the body empty.
- ***InitialAccount\Step*3 illustrates this.**

this

- **Sometimes it is necessary within code for a method to be able to access the current object reference.**
- **C# defines a keyword *this*, which is a special variable that always refers to the current object instance.**
  - With **this** you can then refer to instance variables.
  - If you examine the code for the constructor above, you will see that we made a point to use different names for the parameters than for the instance variables.
  - We can make use of the same names and avoid ambiguity by using the **this** variable.
  - Here is alternate code for the constructor:

```
public Account(int id, decimal balance)
   {
      this.id = id;
      this.balance = balance;
   }
```

  - A better way to avoid ambiguity, however, is to adopt a naming convention that distinguishes between parameters and member names.

    A common naming convention in C# is the use of a trailing underscore for private data member names.

TestAccount Sample Program

- **The program *TestAccount\Step3* illustrates all the features we have discussed so far.**

```
// Account.cs - Step3

public class Account
{
   private int id;
   private decimal balance;
   public Account()
   {
   }
   public Account(int id, decimal balance)
   {
      this.id = id;
      this.balance = balance;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
   }
   public void Withdraw(decimal amount)
   {
      balance -= amount;
   }
   public decimal GetBalance()
   {
      return balance;
   }
   public int GetId()
   {
      return id;
   }
}
```

TestAccount (Cont'd)

- **Here is the driver program:**

```
// TestAccount.cs - Step3

using System;

public class TestAccount
{
   public static void Main(string[] args)
   {
      Account acc;
      acc = new Account(1, 100);
      Console.WriteLine("balance of {0} is {1}",
         acc.GetId(), acc.GetBalance());
      acc.Deposit(25);
      acc.Withdraw(50);
      Console.WriteLine("balance of {0} is {1}",
         acc.GetId(), acc.GetBalance());
      acc = new Account();
      Console.WriteLine("balance of {0} is {1}",
         acc.GetId(), acc.GetBalance());
   }
}
```

- **Here is the output:**

```
balance of 1 is 100
balance of 1 is 75
balance of 0 is 0
```

Static Fields And Methods

- **In C# a field normally is assigned on a *per-instance* basis, with a unique value for each object instance of the class.**
  - Sometimes it is useful to have a single value associated with the entire class.
  - Such a field is called a static field.
  - Like instance data members, static data members can be either **public** or **private**.
  - To access a public static member, you use the dot notation, but in place of an object reference before the dot, you use the name of the class.

Static Methods

- **A method may also be declared *static*.**
  - A static method can be called without instantiating the class.
  - You use the dot notation, with the class name in front of the dot.
  - Because you can call a static method without an instance, a static method can only use static data members and not instance data members.
- **Static methods may be declared *public* or *private*.**
  - A private static method, like other private methods, may be used as a helper function within a class, but not called from outside.
- **Static methods have no *this* reference, because they are not associated with a specific object instance.**
  - Trying to use the *this* keyword in a static method will result in a compiler error.

Sample Program

- **Our previous *Account* classes relied on the user of the class to assign an id for the account.**
  - A better approach is to encapsulate assigning an id within the class itself, so that a unique id will be automatically generated every time an **Account** object is created.
  - It is easy to implement such a scheme by using a static field **nextid**, which is used to assign an id.
  - Every time an id is assigned, **nextid** is incremented.
- **The program *StaticAccount* demonstrates this solution, and also illustrates use of private static helper functions.**
- **Note that the static method *GetNextId* is accessed through the class *Account* and not through an object reference such as *acc3*.**
  - This program also illustrates the fact that **Main** is a static method and is invoked by the runtime without an instance of the **StaticAccount** class being created.
- **Since there is no instance, any method called from within *Main* must also be declared *static*, as illustrated by the method *WriteAccount( )*.**

Static Constructor

- **Besides having static fields and static methods, a class may also have a static constructor.**
  - A static constructor is called only once, before any object instances have been created.
  - A static constructor is defined by prefixing the constructor with **static**.
  - A static constructor can take no parameters and has no access modifier. However, it is always implicitly public.
  - The static constructor may initialize only static data members.
- **The program *StaticConstructor* illustrates a static constructor for the *Account* class.**
  - All of the constructors in this example program write a line of text when they are called, so you can see when the various constructors are called.

Constant And Readonly Fields

- **If you want to make sure that a variable always has the same value, you can assign the value via an initializer and use the *const* modifier.**
  - Such a constant is automatically static, and you will access it from outside the class through the class name.
  - Another situation may call for a one-time initialization at runtime, and after that the value cannot be changed.
  - You can achieve this effect through a **readonly** field.
  - Such a field may be either an instance member or a static member.
  - In the case of an instance member, it will be assigned in an ordinary constructor.
  - In the case of a static member, it will be assigned in a static constructor.
- **The program *ConstantAccount* illustrates the use of both *const* and *readonly*.**
  - In both cases, you will get a compiler error if you try to modify the value.

Summary

- **A class can be thought of as a template from which individual instances (called *objects*) can be created.**
- **Classes support encapsulation through *fields* and *methods*.**
  - Typically, fields are private and methods are public.
- **The *new* operator is used to instantiate objects from classes.**
  - When you declare a variable of a primitive data, you are allocating memory and creating the instance.
  - When you declare a variable of a class type (an "object reference"), you are only obtaining memory for a *reference* to an object of the class type.
  - No memory is allocated for the object itself until *new* is invoked.
- **The Common Language Runtime automatically reclaims the memory of orphaned or unreferenced objects in a process called *garbage collection*.**
- **Initialization of objects can be performed in constructors.**
- **Static members apply to the entire class rather than to a particular instance.**
- ***const* and *readonly* can be used to specify constants in C# programs.**