# Chapter 12

# Arrays and Indexers

# Arrays and Indexers

## Objectives

*After completing this unit you will be able to:*

- **Use arrays and indexers in C#.**
- **Use the Random class to generate pseudo-random sequences.**
- **Start creating more interesting and less trivial programs.**

# Arrays

- **An array is a collection of elements with the following characteristics.**
  - All array elements must be of the same type. The element type of an array can be any type, including an array type. An array of arrays is often referred to as a **jagged** array.
  - An array may have one or more dimensions. For example, a two dimensional array can be visualized as a table of values. The number of dimensions is known as the array's **rank**.
  - Array elements are accessed using one or more computed integer values, each of which is known as an **index**. A one-dimensional array has one index.
  - In C# an array index starts at 0, as in other C family languages.
  - The elements of an array are created when the array object is created. The elements are automatically destroyed when there are no longer any references to the array object.

# One Dimensional Arrays

- **An array is declared using square brackets [] after the type, not after the variable.**

```
int [] a;              // declares an array of int
```

  - Note that the size of the array is not part of its type. The variable declared is a reference to the array.

- **You create the array elements and establish the size of the array using the new operator.**

```
a = new int[10]; // creates 10 array elements
```

  - The new array elements start out with the appropriate default values for the type (0 for **int**).

- **You may both declare and initialize array elements using curly brackets, as in C/C++.**

```
int a[] = {2, 3, 5, 7, 11};
```

  - You can indicate you are done with the array elements by assigning the array reference to **null**.

```
a = null;
```

  - The garbage collector is now free to deallocate the elements.

# System.Array

- **Arrays are objects. *System.Array* is the abstract base class for all array types.**
  - Accordingly, you can use the properties and methods of **System.Array** for any array.
- **Here are some examples:**
  - **Length** is a property that returns the number of elements currently in the array.
  - **Sort** is a static method that will sort the elements of an array.
  - **BinarySearch** is a static method that will search for an element in a sorted array, using a binary search algorithm.

```
int [] a = {5, 2, 11, 7, 3};
Array.Sort(a);        // sorts the array
for (int i = 0; i < a.Length; i++)
    Console.Write("{0} ", a[i]);
Console.WriteLine();
int target = 5;
int index = Array.BinarySearch(a, target);
if (index < 0)
    Console.WriteLine("{0} not found", target);
else
    Console.WriteLine("{0} found at {1}", target,
index);
```

- **A complete program containing the code shown above can be found in *ArrayMethods*.**
  - Here is the output:

```
2 3 5 7 11
5 found at 2
```

# Sample Program

- **The program *ArrayDemo* is an interactive test program for arrays.**
  - A small array is created initially, and you can create new arrays.
  - You can populate an array either with a sequence of square numbers or with random numbers.
  - You can sort the array, reverse the array, and perform a binary search (which assumes that the array is sorted in ascending order).
  - You can destroy the array by assigning the array reference to null.

# Interfaces for System.Array

- **If you look at the documentation for methods of *System.Array*, you will see many references to various interfaces, such as *IComparable*.**
- **By using such interfaces you can control the behavior of methods of *System.Array*.**
  - For example, if you want to sort an array of objects of a class that you define, you must implement the interface **IComparable** in your class so that the **Sort** method knows how to compare elements to carry out the sort.
  - The .NET Framework provides an implementation of **IComparable** for all the primitive types. We will come back to this point after we discuss interfaces in Chapter 17.

# Random Number Generation

- **The *ArrayDemo* program contains the following code for populating an array with random integers between 0 and 100.**

```
Random rand = new Random();
for (int i = 0; i < size; i++)
{
   array[i] = rand.Next(100);
}
```

- **The .NET Framework provides a useful class, *Random*, in the *System* namespace that can be used for generating pseudo-random numbers for simulations.**

# Constructors for Random

- **There are two constructors:**

```
Random();              // uses default seed
Random(int seed);// seed is specified
```

- **The default seed is based on date and time, resulting in a different stream of random numbers each time.**
  - By specifying a seed, you can produce a deterministic stream.

# Next Methods

- **There are three overloaded *Next* methods that return a random *int*.**

```
int Next();
int Next(int maxValue);
int Next(int minValue, int maxValue);
```

  - The first method returns an integer greater than or equal to zero and less than **Int32.MaxValue**.
  - The second method returns an integer greater than or equal to zero and less than **maxValue**.
  - The third method returns an integer greater than or equal to **minValue** and less than or equal to **maxValue**.

- **The *NextDouble* method produces a random *double* between 0 and 1.**

```
double NextDouble();
```

  - The return value **r** is in the range: 0 <= r < 1.

# Jagged Arrays

- **You can declare an array of arrays, or a "jagged" array.**
  - Each row can have a different number of elements.

```
int [][] binomial;
```

- **You then create the array of rows, specifying how many rows there are (each row is itself an array).**

```
binomial = new int [rows][];
```

- **Next you create the individual rows:**

```
binomial[i] = new int [i+1];
```

- **Finally you can assign individual array elements.**

```
binomial[0][0] = 1;
```

- **The example program *Pascal* creates and prints Pascal's triangle.**
- **Higher dimensional jagged arrays can be created following the same principles.**

# Rectangular Arrays

- **C# also permits you to define rectangular arrays, where all rows have the same number of elements. First you declare the array:**

```
int [,] MultTable;
```

- **Then you create all the array elements, specifying the number of rows and columns:**

```
MultTable = new int[rows, columns];
```

- **Finally you can assign individual array elements.**

```
MultTable[i,j] = i * j;
```

- **The example program *RectangularArray* creates and prints out a multiplication table.**
  - Note that the columns do not quite line up. We will discuss formatting in Chapter 15.
  - Higher dimensional rectangular arrays can be created following the same principles.

```
int [,,] Mult3DimTable; // 3 dimensions
```

# Arrays As Collections

- **The class *System.Array* supports the *IEnumerable* interface.**
  - Hence arrays can be treated as collections, a topic we will discuss in Chapter 18.
  - This means that a **foreach** loop can be used to iterate through the elements of an array.
- **The *Pascal* example code contains nested *foreach* loops to display the jagged array.**
  - The outer loop iterates through all the rows, and the inner loop iterates through all the elements within a row.

```
Console.WriteLine(
   "Pascal triangle via nested foreach loop");
foreach (int[] row in binomial)
{
   foreach (int x in row)
   {
      Console.Write("{0} ", x);
   }
   Console.WriteLine();
}
```

# Arrays As Collections (Cont'd)

- **In the *RectangularArray* example there is only one collection.**
  - The **foreach** loop prints out all the array elements on one line, which represents the order in which the array elements are stored in memory.
  - You can see that C# uses "row major" order for storing rectangular arrays: All the elements of the first row are stored, then all the elements of the next row, and so on.
  - Here is the code:

```
// RectangularArray.cs
...

foreach (int x in MultTable)
{
   Console.Write("{0} ", x);
}
Console.WriteLine();
```

  - Here is the output:

```
0 0 0 0 0 0 1 2 3 4 0 2 4 6 8 0 3 6 9 12 0 4 8 12
16
```

# Bank Case Study: Step 1

- **We have covered enough C# that we can begin implementing some interesting programs.**
  - Over the next several chapters, the bulk of our examples will be focused on a case study of a banking system.
- **The code for the case study will be in the *CaseStudy* directory of each chapter.**
- **The examples we have looked at so far in this chapter have all been arrays of integers.**
  - Arrays can be constructed using any data type, including user-defined classes.

# Bank Case Study: Step 1 (Cont'd)

- **In this section we will extend our bank account sample program to implement a *Bank* class, which will contain an array of *Account* objects.**
- **Our case study at this point consists of five classes, each in its own file.**
  - **InputWrapper**. This class simplifies prompting for input and reading in the data. It is identical to the class by this name that we have used previously.
  - **Account**. This class encapsulates a single bank account, consisting of an Id, an Owner, and a Balance. Operations are **Deposit** and **Withdraw**.
  - **Bank**. This class represents a bank, which has several accounts. Methods are provided to add an account, delete an account, and get a list of accounts.
  - **TestBank**. This class provides an interactive test program for exercising the Bank class. Commands are provided to open an account, close an account, show all the accounts, and start an ATM to perform transactions on a particular account.
  - **Atm**. This class provides a user interface for the ATM, which allows a user to perform transactions on a particular account. The operations supported are deposit, withdraw, change owner name, and show account information.

# Account Class

- **The *Account* class is based on the *Account* class in the *AccountProperty* program in Chapter 10.**
  - This version of **Account** adds a **Transactions** property, which keeps a count of the number of transactions (deposits and withdrawals) that have been performed.
  - A new method, **GetStatement**, returns a string showing the owner, the ID, the number of transactions, and the balance.
  - The class no longer assigns an ID internally; instead, an ID is assigned by the **Bank** class.

```
// Account.cs

public class Account
{
   private int id;
   private decimal balance;
   private string owner;
   private int numXact = 0;
                          // number of transactions
   public Account(decimal balance, string owner,
      int id)
   {
      this.balance = balance;
      this.owner = owner;
      this.id = id;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
      numXact++;
   }
```

# Account Class (Cont'd)

```
public void Withdraw(decimal amount)
{
   balance -= amount;
   numXact++;
}
public decimal Balance
{
   get
   {
      return balance;
   }
}
public int Id
{
   get
   {
      return id;
   }
}
public string Owner
{
   get
   {
      return owner;
   }
   set
   {
      owner = value;
   }
}
public int Transactions
{
   get
   {
      return numXact;
   }
}
```

# Account Class (Cont'd)

```
   public string GetStatement()
   {
      string s = "Statement for " + this.Owner
         + " id = " + Id + "\n"
         + this.Transactions
         + " transactions, balance = " + balance;
      return s;
   }
}
```

# Bank Class

- **The class *Bank* maintains an array of *Account* objects.**
  - Separate counters are maintained for the next ID and for the number of open accounts.
  - Methods are provided to add an account, delete an account, get a list of accounts (in the form of an array of strings), and find an account, given the account ID.
  - The constructor creates an array that can store up to 10 accounts and then adds three accounts as initial test data.

```
// Bank.cs

using System;

public class Bank
{
   private Account[] accounts;
   private int nextid = 1;
   private int count = 0;
   public Bank()
   {
      accounts = new Account[10];
      AddAccount(100, "Bob");
      AddAccount(200, "Mary");
      AddAccount(300, "Charlie");
   }
```

# Bank Class (Cont'd)

```
public int AddAccount(decimal bal, string owner)
{
   Account acc;
   int id = nextid++;
   acc = new Account(bal, owner, id);
   accounts[count++] = acc;
   return id;
}
public string[] GetAccounts()
{
   string[] array = new string[count];
   for (int i = 0; i < count; i++)
   {
      string owner = accounts[i].Owner;
      string sid = accounts[i].Id.ToString();
      string sbal =
         accounts[i].Balance.ToString();
      string str = sid + "\t" + owner
         + "\t" + sbal;
      array[i] = str;
   }
   return array;
}
public void DeleteAccount(int id)
{
   int index = FindIndex(id);
   if (index != -1)
   {
      // move accounts down
      for (int i = index; i < count; i++)
      {
         accounts[i] = accounts[i+1];
      }
      count--;
   }
}
```

# Bank Class (Cont'd)

```
   private int FindIndex(int id)
   {
      for (int i = 0; i < count; i++)
      {
         if (accounts[i].Id == id)
            return i;
      }
      return -1;
   }
   public Account FindAccount(int id)
   {
      for (int i = 0; i < count; i++)
      {
         if (accounts[i].Id == id)
            return accounts[i];
      }
      return null;
   }
}
```

# TestBank Class

- **The *TestBank* class provides a user interface in the *Main* method to open an account, close an account, and show all the accounts.**
  - The command "account" brings up an ATM user interface to allow the user to perform transactions on a particular account.

```
// TestBank.cs

using System;

public class TestBank
{
   public static void Main()
   {
      Bank bank = new Bank();
      InputWrapper iw = new InputWrapper();
      string cmd;
      Console.WriteLine(
         "Enter command, quit to exit");
      cmd = iw.getString("> ");
      while (! cmd.Equals("quit"))
      {
         if (cmd.Equals("open"))
         {
            decimal bal = iw.getDecimal(
               "starting balance: ");
            string owner = iw.getString("owner: ");
            int id = bank.AddAccount(bal, owner);
            Console.WriteLine(
               "Account opened, id = {0}", id);
         }
```

# TestBank Class (Cont'd)

```
            else if (cmd.Equals("close"))
            {
               int id = iw.getInt("account id: ");
               bank.DeleteAccount(id);
            }
            else if (cmd.Equals("show"))
               ShowArray(bank.GetAccounts());
            else if (cmd.Equals("account"))
            {
               int id = iw.getInt("account id: ");
               Account acc = bank.FindAccount(id);
               Atm.ProcessAccount(acc);
            }
            else
               help();
            cmd = iw.getString("> ");
         }
      }
      private static void ShowArray(string[] array)
      {
         foreach (string str in array)
            Console.WriteLine(str);
      }
      private static void help()
      {
         Console.WriteLine(
            "The following commands are available:");
         Console.WriteLine(
            "\topen     -- open an account");
         Console.WriteLine(
            "\tclose    -- close an account");
         ...
      }
}
```

# Atm Class

- **The *Atm* class has a single static method, *ProcessAccount*, which provides a user interface for performing transactions on an account.**
  - The operations supported are deposit, withdraw, change the owner name, and obtain a current statement for the account.

```
// Atm.cs

using System;

public class Atm
{
   public static void ProcessAccount(Account acc)
   {
      Console.WriteLine("balance = {0}",
         acc.Balance);
      string cmd;
      InputWrapper iw = new InputWrapper();
      Console.WriteLine(
         "Enter command, quit to exit");
      cmd = iw.getString(">> ");
      while (! cmd.Equals("quit"))
      {
         if (cmd.Equals("deposit"))
         {
            decimal amount =
               iw.getDecimal("amount: ");
            acc.Deposit(amount);
            Console.WriteLine("balance = {0}",
               acc.Balance);
         }
```

# Atm Class (Cont'd)

```
            else if (cmd.Equals("withdraw"))
            {
               decimal amount = iw.getDecimal(
                  "amount: ");
               acc.Withdraw(amount);
               Console.WriteLine("balance = {0}",
                  acc.Balance);
            }
            else if (cmd.Equals("owner"))
            {
               string owner = iw.getString(
                  "new owner name: ");
               acc.Owner = owner;
               show(acc);
            }
            else if (cmd.Equals("show"))
               show(acc);
            else
               accountHelp();
            cmd = iw.getString(">> ");
         }
      }
      private static void show(Account acc)
      {
         Console.WriteLine(acc.GetStatement());
      }
      private static void accountHelp()
      {
         Console.WriteLine(
            "The following commands are available:");
         Console.WriteLine(
            "\tdeposit  -- make a deposit");
         ...
      }
}
```

# Running the Case Study

- **You should become thoroughly familiar with this case study, as we will use it extensively in the next several chapters.**
  - You should both study the code and run it.
- **Again, the program is located in the *CaseStudy* directory for this chapter.**
  - The following is a transcript of a sample run, in which an account is added, the accounts are shown, a deposit is made to the new account, an account is deleted, and the accounts are shown again.

# Running the Case Study (Cont'd)

```
C:\OIC\CSharp\chap12\CaseStudy>TestBank
Enter command, quit to exit
> ?
The following commands are available:
     open     -- open an account
     close    -- close an account
     show     -- show all accounts
     account  -- perform transactions on an account
     quit     -- exit the program
> open
starting balance: 100
owner: Howard
Account opened, id = 4
> account
account id: 4
balance = 100
Enter command, quit to exit
>> deposit
amount: 1000
balance = 1100
>> quit
> show
1       Bob     100
2       Mary    200
3       Charlie 300
4       Howard  1100
> close
account id: 3
> show
1       Bob     100
2       Mary    200
4       Howard  1100
> quit

C:\OIC\CSharp\chap12\CaseStudy>
```

# Indexers

- **C# provides various ways to help the user of a class access encapsulated data.**
  - In Chapter 10 we saw how **properties** can provide access to a single piece of data associated with a class, making it appear like a public field.
- **In this section we will see how *indexers* provide a similar capability for accessing a group of data items, using an array index notation.**
  - Indexers can be provided when there is a private array or other collection.
  - An indexer can also be provided even if there is nothing like an array within the class.

# Indexers (Cont'd)

- **The program *ColorIndex* provides an illustration.**
  - There are three **byte** private variables, **red**, **green**, and **blue**, that hold a color intensity value between 0 and 255 for the three primary colors.
  - We want to provide a way of accessing these values through a "color index" that will be 0 for red, 1 for green, and 2 for blue.
- **The notation is somewhat similar to that used for properties, with *set* and *get* functions.**
  - But there is no "name" for the indexer, as the indexer is accessed through a variable of type **ColorIndex** and an index.
  - So where a property name would be present, we use **this** for an indexer, and there is an index. In making an assignment, the keyword **value** is used, just as for properties.

# ColorIndex Example Program

```
// ColorIndex.cs

public class ColorIndex
{
   private byte red = 255;
   private byte green = 127;
   private byte blue = 0;
   public byte this[int index]
   {
      get
      {
         if (index == 0)
            return red;
         else if (index == 1)
            return green;
         else
            return blue;
      }
      set
      {
         if (index == 0)
            red = value;
         else if (index == 1)
            green = value;
         else
            blue = value;
      }
   }
   public string Color
   {
      get
      {
         return red + ":" + green + ":" + blue;
      }
   }
}
```

# Using the Indexer

- **The indexer is used via array notation.**

```
// TestColorIndex.cs

using System;

public class TestColorIndex
{
   public static void Main(string[] args)
   {
      ColorIndex ci = new ColorIndex();
      Console.WriteLine(ci.Color);
      Console.WriteLine("red = {0}", ci[0]);
      Console.WriteLine("green = {0}", ci[1]);
      Console.WriteLine("blue = {0}", ci[2]);
      ci[0] = 77;
      ci[1] = 133;
      ci[2] = 199;
      Console.WriteLine(ci.Color);
   }
}
```

  - Here is the output of the program:

```
255:127:0
red = 255
green = 127
blue = 0
77:133:199
```

# Summary

- **In C# arrays are objects of a reference data type and are based on the class *System.Array*.**
- **Arrays have methods to perform operations such as sorting and searching.**
- **The *Random* class provides a convenient way to populate arrays with test data.**
- **There are two varieties of higher dimensional arrays: jagged, and rectangular.**
- **A jagged array is an array of arrays, and each row can have a different number of elements. In rectangular arrays, all rows have the same number of elements.**
- **Arrays are a special kind of collection, which means that the *foreach* loop can be used in C# for iterating through array elements.**
- **Indexers provide a way to access encapsulated data in a class with an array notation.**