Session: 12

Events, Delegates, and Collections

Objectives

- Explain delegates
- Explain events
- Define and describe collections

- Following are the features of delegates:

In the .NET Framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.

Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names.

Using delegates, you can call any method, which is identified only at run-time.

A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time.

In C#, invoking a delegate will execute the referenced method at run-time.

To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

Delegates in C#

- Consider two methods, `Add`() and `Subtract`(). The method `Add`() takes two parameters of type integer and returns their sum as an integer value. Similarly, the method `Subtract`() takes two parameters of type integer and returns their difference as an integer value.
- Since both methods have the same parameter and return types, a delegate, `Calculation`, can be created to be used to refer to `Add`() or `Subtract`(). However, when the delegate is called while pointing to `Add`(), the parameters will be added. Similarly, if the delegate is called while pointing to `Subtract`(), the parameters will be subtracted.
- Following are the features of delegates in C# that distinguish them from normal methods:
  - Methods can be passed as parameters to a delegate. In addition, a delegate can accept a block of code as a parameter. Such blocks are referred to as anonymous methods because they have no method name.
  - A delegate can invoke multiple methods simultaneously. This is known as multicasting.
  - A delegate can encapsulate static methods.
  - Delegates ensure type-safety as the return and parameter types of the delegate are the same as that of the referenced method. This ensures secured reliable data to be passed to the invoked method.

Declaring Delegates

- Delegates in C# are declared using the `delegate` keyword followed by the return type and the parameters of the referenced method.
- Declaring a delegate is quite similar to declaring a method except that there is no implementation. Thus, the declaration statement must end with a semi-colon.
- The following figure displays an example of declaring delegates:

- The following syntax is used to declare a delegate:

Syntax

```
<access_modifier> delegate <return_type> DelegateName([list_of_parameters]);
```

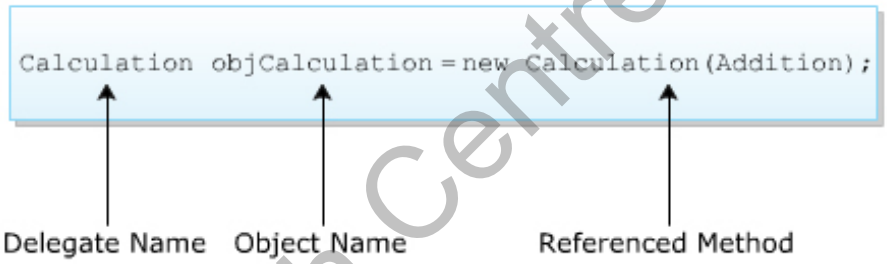
- where,
  - `access_modifier:` Specifies the scope of access for the delegate. If declared outside the class, the scope will always be public.
  - `return_type:` Specifies the data type of the value that is returned by the method.
  - `DelegateName:` Specifies the name of the delegate.
  - `list_of_parameters:` Specifies the data types and names of parameters to be passed to the method.
- The following code declares the delegate **`Calculation`** with the return type and the parameter types as integer:

Snippet

```
public delegate int Calculation(int numOne, int numTwo);
```

Instantiating Delegates 1-2

- The next step after declaring the delegate is to instantiate the delegate and associate it with the required method by creating an object of the delegate.
- Like all other objects, an object of a delegate is created using the `new` keyword.
- This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate.
- The created object is used to invoke the associated method at run-time.
- The following figure displays an example of instantiating delegates:



- The following syntax is used to instantiate a delegate:

Syntax

```
<DelegateName><objName> = new <DelegateName>(<MethodName>);
```

- where,
  - `DelegateName`: Specifies the name of the delegate .
  - `objName`: Specifies the name of the delegate object.
  - `MethodName`: Specifies the name of the method to be referenced by the delegate object.

Instantiating Delegates 2-2

- The following code declares a delegate **Calculation** outside the class **Mathematics** and instantiates it in the class:

Snippet

```
public delegate int Calculation (int numOne, int numTwo);
class Mathematics
{
    static int Addition(int numOne, int numTwo)
    {
        return (numOne + numTwo);
    }
    static int Subtraction(int numOne, int numTwo)
    {
        return (numOne - numTwo);
    }
    static void Main(string[] args)
    {
        int valOne = 5;
        int valTwo = 23;
        Calculation objCalculation = new Calculation(Addition);
        Console.WriteLine (valOne + " + " + valTwo + " = " +
        objCalculation (valOne, valTwo));
    }
}
```

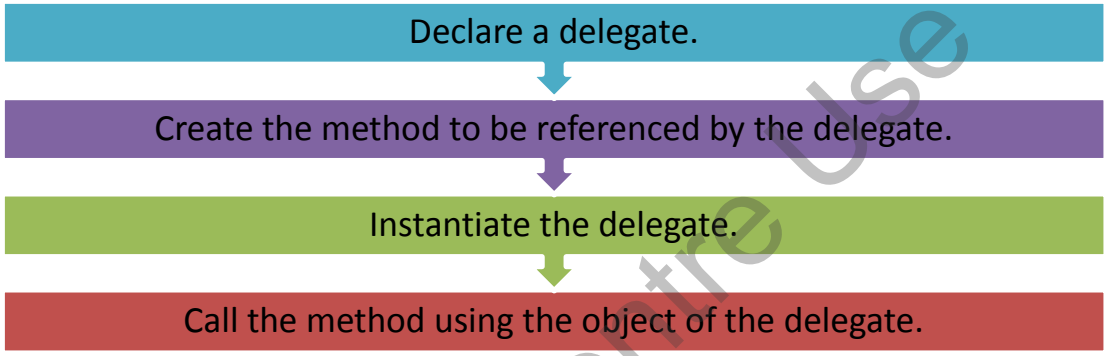
- In the code:
  - The delegate called **Calculation** is declared outside the class **Mathematics**.
  - In the `Main()` method, an object of the delegate is created that takes the **Addition()** method as the parameter. The parameter type of the method and that of the delegate is the same, which is type `int`.

Output

5 + 23= 28

Using Delegates 1-2

- A delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.
- The following are the four steps to implement delegates in C#:



- Each of these step is demonstrated with an example shown in the following figure:

```
class DelegatesDemo
{
  public delegate double Temperature(double temp);

  public static double FahrenheitToCelsius(double temp)
     {
       return ((temp-32) / 9)*5;
     }
 public static void Main()
 {
  temperature tempConversion = new temperature(FahrenhietToCelsius);

  double tempF = 96;

  double tempC = tempConversion(tempF);

  Console.WriteLine("Temperature in Fahrenhiet = {0:F}".tempF);

  Console.WriteLine("Temperature in Celsius = {0:F)".tempC);
 }
}
```

Using Delegates 2-2

- An anonymous method is an inline block of code that can be passed as a delegate parameter that helps to avoid creating named methods.
- The following figure displays an example of using anonymous methods:

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
            {
                Console.Write("Testing… ");
                Console.WriteLine("Threads.");
            });
    objThread.Start();
}
```

Anonymous Method

Delegate-Event Model

- The delegate-event model is a programming model that enables a user to interact with a computer and computer-controlled devices using graphical user interfaces. This model consists of:
  - An event source, which is the console window in case of console-based applications.
  - Listeners that receive the events from the event source.
  - A medium that gives the necessary protocol by which every event is communicated.
- In this model, every listener must implement a medium for the event that it wants to listen to by using the medium, every time the source generates an event, the event is notified to the registered listeners.

Example

- Consider a guest ringing a doorbell at the doorstep of a home. The host at home listens to the bell and responds to the ringing action by opening the door.
- Here, the ringing of the bell is an event that resulted in the reaction of opening the door. Similarly, in C#, an event is a generated action that triggers its reaction.
- For example, pressing `Ctrl+Break` on a console-based server window is an event that will cause the server to terminate.
- This event results in storing the information in the database, which is the triggered reaction. Here, the listener is the object that invokes the required method to store the information in the database.
- Delegates can be used to handle events as they take methods that need to be invoked when events occur which are referred to as the event handlers.

Multiple Delegates 1-2

- In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate, the appropriate delegate is invoked.
- The following code demonstrates the use of multiple delegates by creating two delegates **`CalculateArea`** and **`CalculateVolume`** that have their return types and parameter types as double:

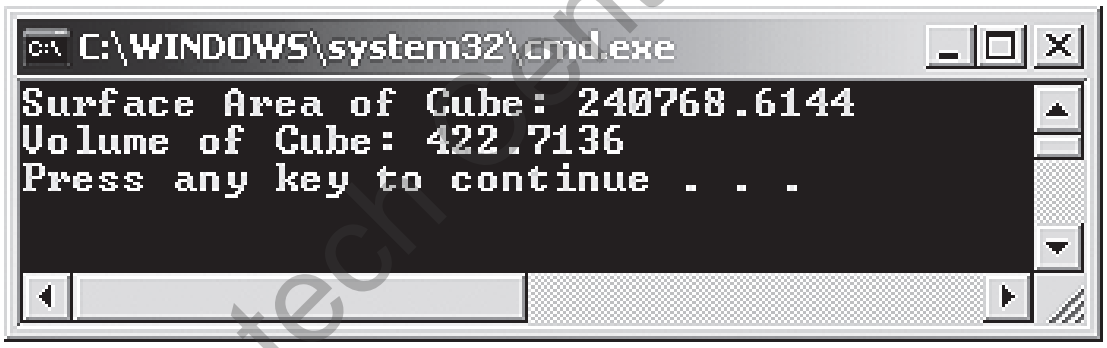
Snippet

```
using System;
public delegate double CalculateArea(double val);
public delegate double CalculateVolume(double val);

class Cube
{
      static double Area(double val)
      {
        return 6 * (val * val);
      }
      static double Volume(double val)
      {
        return (val * val);
      }
      static void Main(string[] args)
      {
          CalculateArea objCalculateArea = new CalculateArea(Area);
          CalculateVolume objCalculateVolume = new
          CalculateVolume(Volume);
          Console.WriteLine ("Surface Area of Cube: " +
          objCalculateArea(200.32));
          Console.WriteLine("Volume of Cube: " +
          objCalculateVolume(20.56));
      }
}
```

Multiple Delegates 2-2

- In the code:
  - When the delegates **`CalculateArea`** and **`CalculateVolume`** are instantiated in the `Main`() method, the references of the methods **`Area`** and **`Volume`** are passed as parameters to the delegates **`CalculateArea`** and **`CalculateVolume`** respectively.
  - The values are passed to the instances of appropriate delegates, which in turn invoke the respective methods.
- The following figure shows the use of multiple delegates:



Multicast Delegates 1-3

- A single delegate can encapsulate the references of multiple methods at a time to hold a number of method references.
- Such delegates are termed as 'Multicast Delegates' that maintain a list of methods (invocation list) that will be automatically called when the delegate is invoked.
- Multicast delegates in C# are sub-types of the `System.MulticastDelegate class`. Multicast delegates are defined in the same way as simple delegates, however, the return type of multicast delegates can only be `void`.
- If any other return type is specified, a run-time exception will occur because if the delegate returns a value, the return value of the last method in the invocation list of the delegate will become the return type of the delegate resulting in inappropriate results. Hence, the return type is always `void`.
- To add methods into the invocation list of a multicast delegate, the user can use the '+' or the '+=' assignment operator. Similarly, to remove a method from the delegate's invocation list, the user can use the '-' or the '-=' operator. When a multicast delegate is invoked, all the methods in the list are invoked sequentially in the same order in which they have been added.

- The following code creates a multicast delegate **Maths**. This delegate encapsulates the reference to the methods **Addition, Subtraction, Multiplication**, and **Division:**

Snippet

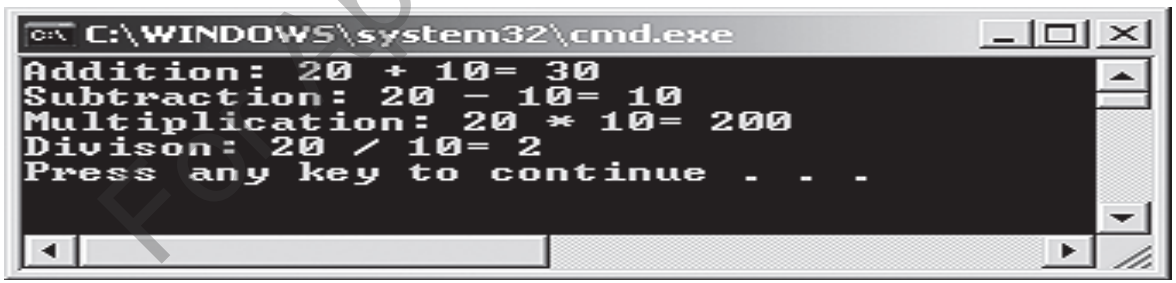
```
using System;
public delegate void Maths (int valOne, int valTwo);

class MathsDemo
{
      static void Addition(int valOne, int valTwo)
      {
          int result = valOne + valTwo;
          Console.WriteLine("Addition: " + valOne + " + " +
          valTwo + "= " + result);
      }
      static void Subtraction(int valOne, int valTwo)
      {
          int result = valOne - valTwo;
          Console.WriteLine("Subtraction: " + valOne + " - " +
          valTwo + "= " + result);
      }
      static void Multiplication(int valOne, int valTwo)
      {
          int result = valOne * valTwo;
          Console.WriteLine("Multiplication: " + valOne + " * "
          + valTwo + "= " + result);
}
```

Multicast Delegates 3-3

```
        static void Division(int valOne, int valTwo)
        {
            int result = valOne / valTwo;
            Console.WriteLine("Division: " + valOne + " / " +
            valTwo + "= " + result);
        }
        static void Main(string[] args)
        {
            Maths objMaths = new Maths(Addition);
            objMaths += new Maths(Subtraction);
            objMaths += new Maths(Multiplication);
            objMaths += new Maths(Division);
            if (objMaths != null)
            {
            objMaths(20, 10);
            }
        }
}
```

- In the code:
  - The delegate **`Maths`** is instantiated in the `Main()` method. Once the object is created, methods are added to it using the '+=' assignment operator, which makes the delegate a multicast delegate.
- The following figure shows the creation of a multicast delegate:



System.Delegate Class 1-5

- The `Delegate` class of the `System` namespace is a built-in class defined to create delegates in C#.
- All delegates in C# implicitly inherit from the `Delegate` class. This is because the delegate keyword indicates to the compiler that the defined `delegate` in a program is to be derived from the Delegate class. The `Delegate` class provides various constructors, methods, and properties to create, manipulate, and retrieve `delegates` defined in a program.
- The following table lists the constructors defined in the `Delegate` class:

Constructor	Description
`Delegate(object, string)`	Calls a method referenced by the object of the class given as the parameter
`Delegate(type, string)`	Calls a static method of the class given as the parameter

- The following table lists the properties defined in the `Delegate` class:

Property	Description
`Method`	Retrieves the referenced method
`Target`	Retrieves the object of the class in which the delegate invokes the referenced method

System.Delegate Class 2-5

- The following table lists some of the methods defined in the `Delegate` class:

Method	Description
`Clone`	Makes a copy of the current delegate
`Combine`	Merges the invocation lists of the multicast delegates
`CreateDelegate`	Declares and initializes a delegate
`DynamicInvoke`	Calls the referenced method at run-time
`GetInvocationList`	Retrieves the invocation list of the current delegate

- The following code demonstrates the use of some of the properties and methods of the built-in `Delegate` class:

Snippet

```
using System;
public delegate void Messenger(int value);
class CompositeDelegates
{
        static void EvenNumbers(int value)
        {
          Console.Write("Even Numbers: ");
          for (int i = 2; i <= value; i+=2)
          {
          Console.Write(i + " ");
          }
        }
        void OddNumbers(int value)
{
```

System.Delegate Class 3-5

```
        Console.WriteLine();
        Console.Write("Odd Numbers: ");
        for (int i = 1; i <= value; i += 2)
        {
        Console.Write (i + " ");
        }
    }
    static void Start(int number)
    {
        CompositeDelegates objComposite = new CompositeDelegates();
        Messenger objDisplayOne = new Messenger(EvenNumbers);
        Messenger objDisplayTwo = new Messenger
        (objComposite.OddNumbers);
        Messenger objDisplayComposite =
        (Messenger)Delegate.Combine
        (objDisplayOne, objDisplayTwo);
        objDisplayComposite(number);
        Console.WriteLine();
        Object obj = objDisplayComposite.Method.ToString();
        if (obj != null)
        {
        Console.WriteLine ("The delegate invokes an instance
        method: " + obj);
        }
        else
        {
          Console.WriteLine ("The delegate invokes only
          static methods");
        }
```

System.Delegate Class 4-5

```
        }
        static void Main(string[] args)
        {
            int value = 0;
            Console.WriteLine("Enter the values till which you want
            to display even and odd numbers");
            try
            {
              value = Convert.ToInt32(Console.ReadLine());
            }
            catch (FormatException objFormat)
            {
              Console.WriteLine("Error: " + objFormat);
            }
            Start(value);
            }
}
```

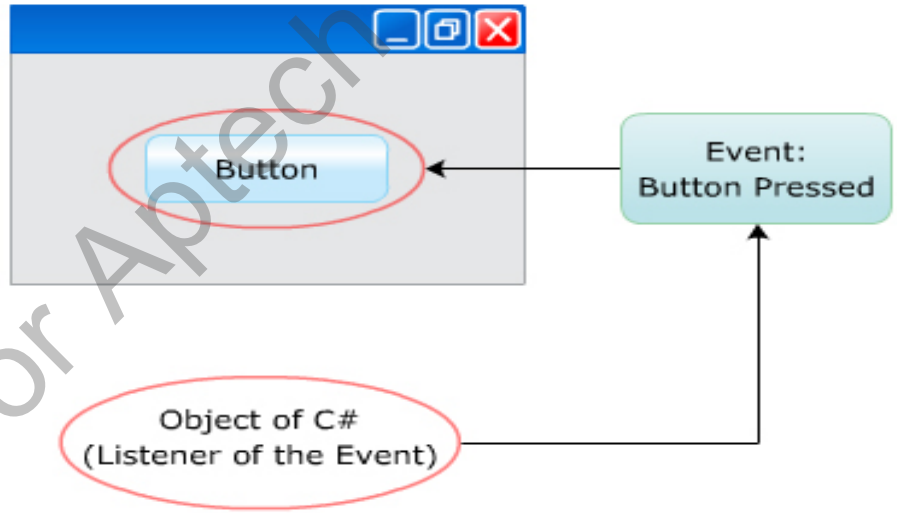
- In the code:
  - The delegate **`Messenger`** is instantiated in the **`Start()`** method.
  - An instance of the delegate, **`objDisplayOne`**, takes the static method, **`EvenNumbers()`**, as a parameter, and another instance of the delegate, **`objDisplayTwo`**, takes the non-static method, **`OddNumbers()`**, as a parameter by using the instance of the class.
  - The `Combine()` method merges the delegates provided in the list within the parentheses.

System.Delegate Class 5-5

- The `Method` property checks whether the program contains instance methods or static methods. If the program contains only static methods, then the Method property returns a null value.
- The `Main()` method allows the user to enter a value. The **`Start()`** method is called by passing this value as a parameter. This value is again passed to the instance of the class **`CompositeDelegates`** as a parameter, which in turn invokes both the delegates.
- The code displays even and odd numbers within the specified range by invoking the appropriate methods.
- The following figure shows the use of some of the properties and methods of the `Delegate` class:

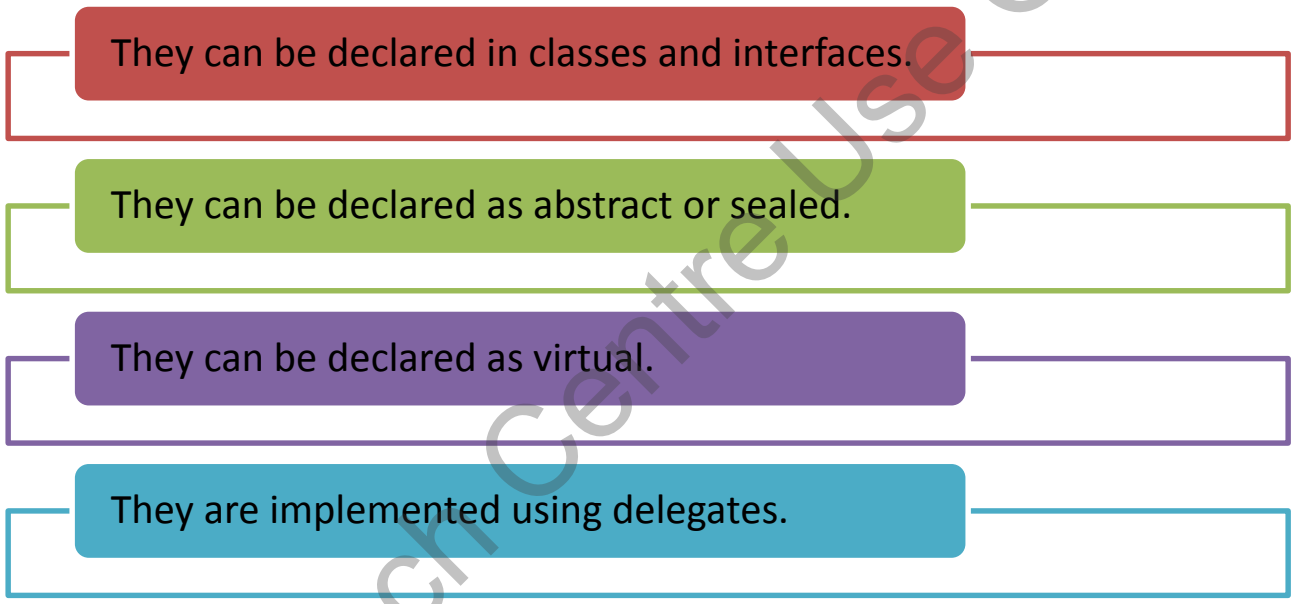
Events

- Consider a group of people at a party playing Bingo. When a number is called, the participants check if the number is on their cards whereas the non-participants go about their business, enjoying other activities.
- If this situation is analyzed from a programmer's perspective, the calling of the number corresponds to the occurrence of an event.
- The notification about the event is given by the announcer.
- Here, the people playing the game are paying attention (subscribing) to what the announcer (the source of the event) has to say (notify).
- Similarly, in C#, events allow an object (source of the event) to notify other objects (subscribers) about the event (a change having occurred).
- The following figure depicts the concept of events:



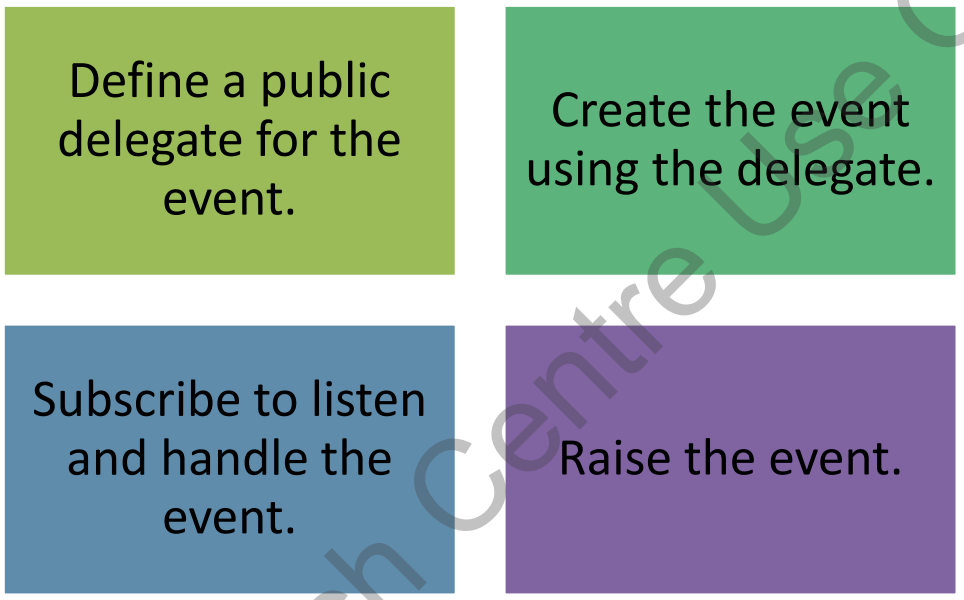
Features

- An event is a user-generated or system-generated action that enables the required objects to notify other objects or classes to handle the event. Events in C# have the following features:



- Events can be used to perform customized actions that are not already supported by C#.
- Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked.

- Following are the four steps for implementing events in C#:



- Events use delegates to call methods in objects that have subscribed to the event.
- When an event containing a number of subscribers is raised, many delegates will be invoked.

Declaring Events 1-2

- An event declaration consists of two steps, creating a delegate and creating the event. A delegate is `declared` using the delegate keyword.
- The delegate passes the parameters of the appropriate method to be invoked when an event is generated.
- This method is known as the event handler.
- The event is then declared using the event keyword followed by the name of the delegate and the name of the event.
- This declaration associates the event with the delegate.
- The following figure displays the syntax for declaring delegates and events:

Declaring a Delegate:

```
<access_modifier> delegate <return type> <Identifier> (parameters);
```

Declaring an Event:

```
<access_modifier> event <DelegateName> <EventName>;
```

- An object can subscribe to an event only if the event exists. To subscribe to the event, the object adds a delegate that calls a method when the event is raised.
- This is done by associating the event handler to the created event, using the **+= addition assignment** operator which is known as subscribing to an event.

Declaring Events 2-2

- To unsubscribe from an event, use the **-= subtraction assignment** operator.
- The following syntax is used to create a method in the receiver class:

Syntax

```
<access_modifier> <return_type> <MethodName> (parameters);
```

- where,
  - `objectName:` Is the object of the class in which the event handler is defined.
- The following code associates the event handler to the declared event:

```
using System;
public delegate void PrintDetails();

class TestEvent
{

      event PrintDetails Print;

      void Show()
      {
          Console.WriteLine("This program illustrate how to subscribe objects
          to an event");
          Console.WriteLine("This method will not execute since the event has
          not been raised");
      }
      static void Main(string[] args)
      {
          TestEvent objTestEvent = new TestEvent();
          objTestEvent.Print += new PrintDetails(objEvents.Show);
      }
}
```

- In the code:
  - The delegate called **`PrintDetails()`** is declared without any parameters. In the class **`TestEvent`**, the event **`Print`** is created that is associated with the delegate. In the `Main()` method, object of the class **`TestEvent`** is used to subscribe the event handler called **`Show()`** to the event **`Print.`**

Raising Events 1-2

- An event is raised to notify all the objects that have subscribed to the event. Events are either raised by the user or the system.
- Once an event is generated, all the associated event handlers are executed. The delegate calls all the handlers that have been added to the event.
- However, before raising an event, it is important for you to create handlers and thus, make sure that the event is associated to the appropriate event handlers.
- If the event is not associated to any event handler, the declared event is considered to be `null`.
- The following figure displays the raising events:

```
public delegate void Display();

class Events
{
        event Display Print;

        void Show()
        {
        Console.WriteLine("This is an event driven program");
        }

        static void Main(string[] args)
        {
        Events objEvents = new Events();
        objEvents.Print += new Display(objEvents.Show);
        objEvents.Print();
        }
}

Output:

This is an event driven program
```

Invoking the Event Handler through the Created Event

Raising Events 2-2

- The following code can be used to check a particular condition before raising the event:

Snippet

```
if(condition)
{
   eventMe();
}
```

- In the code:
  - If the checked condition is satisfied, the event **eventMe** is raised.
- The syntax for raising an event is similar to the syntax for calling a method. When **eventMe** is raised, it will invoke all the delegates of the objects that have subscribed to it. If no objects have subscribed to the event and the event has been raised, an exception is thrown.

Events and Inheritance 1-2

- Events in C# can only be invoked in the class in which they are declared and defined. Therefore, events cannot be directly invoked by the derived classes. However, events can be invoked indirectly in C# by creating a protected method in the base class that will, in turn, invoke the event defined in the base class. The following code illustrates how an event can be indirectly invoked:

Snippet

```
using System;
public delegate void Display(string msg);

public class Parent
{
      event Display Print;
      protected void InvokeMethod()
      {
         Print += new Display(PrintMessage);
         Check();
      }
      void Check()
      {
         if (Print != null)
         {
         PrintMessage("Welcome to C#");
         }
      }
      void PrintMessage(string msg)
      {
         Console.WriteLine(msg);
      }
}
class Child : Parent
{
      static void Main(string[] args)
      {
         Child objChild = new Child();
         objChild.InvokeMethod();
         }
}
```

- In the code:
  - The class **Child** is inherited from the class **Parent**. An event named **Print** is created in the class **Parent** and is associated with the delegate **Display**. The protected method **InvokeMethod()** associates the event with the delegate and passes the method **PrintMessage()** as a parameter to the delegate. The **Check()** method checks whether any method is subscribing to the event. Since the **PrintMessage()** method is subscribing to the **Print** event, this method is called. The `Main()` method creates an instance of the derived class **Child**. This instance invokes the **InvokeMethod()** method, which allows the derived class **Child** access to the event **Print** declared in the base class **Parent**.
- The following figure shows the outcome of invoking the event:

Collections

- A collection is a set of related data that may not necessarily belong to the same data type that can be set or modified dynamically at run-time.
- Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.
- The following table lists the differences between arrays and collections:

Array	Collection
Cannot be resized at run-time.	Can be resized at run-time.
The individual elements are of the same data type.	The individual elements can be of different data types.
Do not contain any methods for operations on elements.	Contain methods for operations on elements.

System.Collections Namespace 1-3

- The System.Collections namespace in C# allows you to construct and manipulate a collection of objects that includes elements of different data types. The System.Collections namespace defines various collections such as dynamic arrays, lists, and dictionaries.
- The System.Collections namespace consists of classes and interfaces that define the different collections.
- The following table lists the commonly used classes and interfaces in the System.Collections namespace:

Class/Interface	Description
ArrayList Class	Provides a collection that is similar to an array except that the items can be dynamically added and retrieved from the list and it can contain values of different types
Stack Class	Provides a collection that follows the Last-In-First-Out (LIFO) principle, which means the last item inserted in the collection, will be removed first
Hashtable Class	Provides a collection of key and value pairs that are arranged, based on the hash code of the key
SortedList Class	Provides a collection of key and value pairs where the items are sorted, based on the keys
IDictionary Interface	Represents a collection consisting of key/value pairs
IDictionaryEnumerator Interface	Lists the dictionary elements
IEnumerable Interface	Defines an enumerator to perform iteration over a collection
ICollection Interface	Specifies the size and synchronization methods for all collections
IEnumerator Interface	Supports iteration over the elements of the collection
IList Interface	Represents a collection of items that can be accessed by their index number

System.Collections Namespace 2-3

- The following code demonstrates the use of the commonly used classes and interfaces of the `System.Collections` namespace:

Snippet

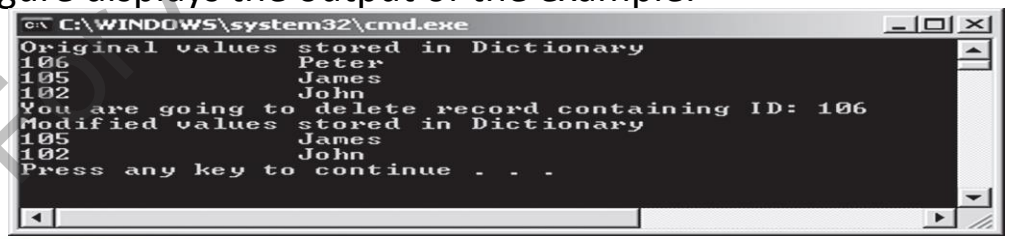
```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

class Employee : DictionaryBase
{
        public void Add(int id, string name)
        {
           Dictionary.Add(id, name);
        }
        public void OnRemove(int id)
        {
        Console.WriteLine("You are going to delete record
        containing ID: " + id);
        Dictionary.Remove(id);
        }
        public void GetDetails()
        {
        IDictionaryEnumerator objEnumerate =
        Dictionary.GetEnumerator();
            while (objEnumerate.MoveNext())
            {
                Console.WriteLine(objEnumerate.Key.ToString() +
                "\t\t" +
            objEnumerate.Value);
            }
        }
        static void Main(string[] args)
        {
```

System.Collections Namespace 3-3

```
            Employee objEmployee = new Employee();
            objEmployee.Add(102, "John");
            objEmployee.Add(105, "James");
            objEmployee.Add(106, "Peter");
            Console.WriteLine("Original values stored in
            Dictionary");
            objEmployee.GetDetails();
            objEmployee.OnRemove(106);
            Console.WriteLine("Modified values stored in
            Dictionary");
            objEmployee.GetDetails();
            }
}
```

- In the code:
  - The class **Employee** is inherited from the DictionaryBase class.
  - The DictionaryBase class is an abstract class. The details of the employees are inserted using the methods present in the DictionaryBase class.
  - The user-defined **Add()** method takes two parameters, namely **id** and **name**. These parameters are passed onto the Add() method of the Dictionary class.
  - The Dictionary class stores these values as a key/value pair.
  - The OnRemove() method of DictionaryBase is overridden. It takes a parameter specifying the key whose key/value pair is to be removed from the Dictionary class.
  - This method then prints a warning statement on the console before deleting the record from the Dictionary class.
  - The Dictionary.Remove() method is used to remove the key/value pair.
  - The GetEnumerator() method returns an IDictionaryEnumerator, which is used to traverse through the list.
- The following figure displays the output of the example:



System.Collections.Generic Namespace

Example

- Consider an online application form used by students to register for an examination conducted by a university.
- The application form can be used to apply for examination of any course offered by the university.
- Similarly, in C#, generics allow you to define data structures that consist of functionalities which can be implemented for any data type.
- Thus, generics allow you to reuse a code for different data types.
- To create generics, you should use the built-in classes of the `System.Collections.Generic` namespace. These classes ensure type-checking.
- To create generics, you should use the built-in classes of the `System.Collections.Generic` namespace.
- These classes ensure type-safety, which is a feature of C# that ensures a value is treated as the type with which it is declared.

- The `System.Collections.Generic` namespace consists of classes and interfaces that define the different generic collections.
  - **Classes:**
    - The `System.Collections.Generic` namespace consists of classes that allow you to create type-safe collections.
    - The following table lists the commonly used classes in the `System.Collections.Generic` namespace:

Class	Description
`List<T>`	Provides a generic collection of items that can be dynamically resized
`Stack<T>`	Provides a generic collection that follows the LIFO principle, which means that the last item inserted in the collection will be removed first
`Queue<T>`	Provides a generic collection that follows the FIFO principle, which means that the first item inserted in the collection will be removed first
`Dictionary<K, V>`	Provides a generic collection of keys and values
`SortedDictionary<K, V>`	Provides a generic collection of sorted key and value pairs that consist of items sorted according to their key
`LinkedList<T>`	Implements the doubly linked list by storing elements in it

- **Interfaces and Structures**
  - The `System.Collections.Generic` namespace consists of interfaces and structures that can be implemented to create type-safe collections.
  - The following table lists some of the commonly used ones:

Interface	Description
`Icollection` Interface	Defines methods to control the different generic collections
`Ienumerable` Interface	Is an interface that defines an enumerator to perform an iteration of a collection of a specific type
`IComparer` Interface	Is an interface that defines a method to compare two objects
`IDictionary` Interface	Represents a generic collection consisting of the key and value pairs
`IEnumerator` Interface	Supports simple iteration over elements of a generic collection
`IList` Interface	Represents a generic collection of items that can be accessed using the index position
`Dictionary.Enumerator` Structure	Lists the elements of a Dictionary
`Dictionary.KeyCollection.Enumerator` Structure	Lists the elements of a `Dictionary.KeyCollection`
`Dictionary.ValueCollection.Enumerator` Structure	Lists the elements of a `Dictionary.ValueCollection`
`Key/ValuePair` Structure	Defines a key/value pair

- The following code demonstrates the use of the commonly used classes, interfaces, and structures of the `System.Collection.Generic` namespace:

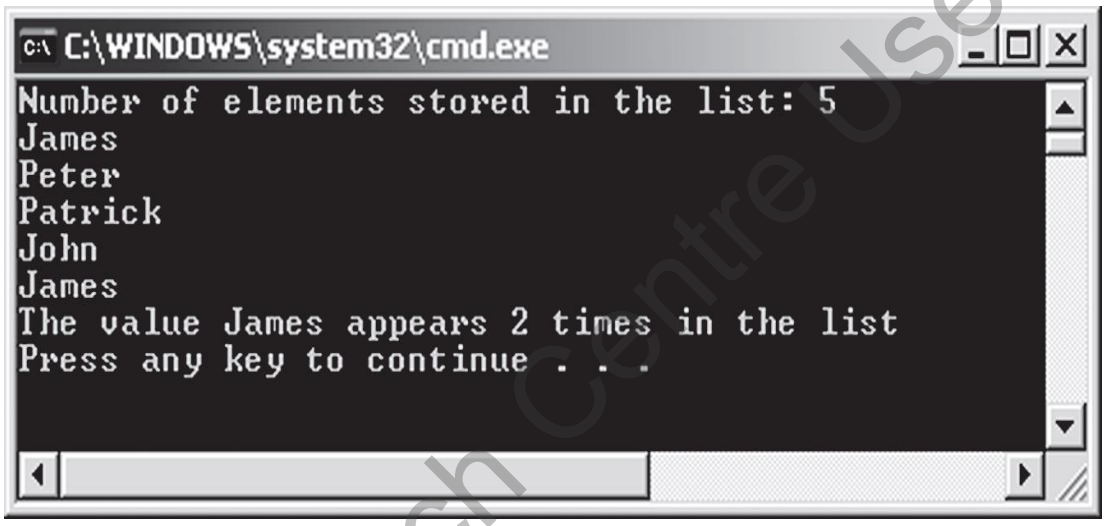
Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;
class Student : IEnumerable
{
    LinkedList<string> objList = new LinkedList<string>();
    public void StudentDetails()
    {
        objList.AddFirst("James");
        objList.AddFirst("John");
        objList.AddFirst("Patrick");
        objList.AddFirst("Peter");
        objList.AddFirst("James");
        Console.WriteLine("Number of elements stored in the list: "
        + objList.Count);
    }
    public void Display(string name)
    {
        LinkedListNode<string> objNode;
        int count = 0;
        for (objNode = objList.First; objNode != null; objNode =
        objNode.Next)
        {
            if (objNode.Value.Equals(name))
            {
               count++;
            }
    }
        Console.WriteLine("The value " + name + " appears " + count
        +" times in the list");
```

```
    }
    public IEnumerator GetEnumerator()
    {
       return objList.GetEnumerator();
    }
    static void Main(string[] args)
    {
       Student objStudent = new Student();
       objStudent.StudentDetails();
       foreach (string str in objStudent)
       {
           Console.WriteLine(str);
       }
       objStudent.Display("James");
    }
}
```

- In the code:
  - The **`Student`** class implements the `IEnumerable` interface. A doubly-linked list of `string` type is created.
  - The **`StudentDetails()`** method is defined to insert values in the linked list.
  - The `AddFirst()` method of the `LinkedList` class is used to insert values in the linked list.
  - The **`Display()`** method accepts a single `string` argument that is used to search for a particular value. A `LinkedListNode` class reference of string type is created inside the **`Display()`** method.
  - This reference is used to traverse through the linked list.
  - Whenever a match is found for the string argument accepted in the **`Display()`** method, a counter is incremented.
  - This counter is then used to display the number of times the specified string has occurred in the linked list.
  - The `GetEnumerator()` method is implemented, which returns an `IEnumerator`.
  - The `IEnumerator` is used to traverse through the list and display all the values stored in the linked list.

- The following figure displays the `System.Collection.Generic` namespace example:



ArrayList Class 1-5

- Following are the features of `ArrayList` class:

The `ArrayList` class is a variable-length array, that can dynamically increase or decrease in size. Unlike the `Array` class, this class can store elements of different data types.

The `ArrayList` class allows you to specify the size of the collection, during program execution and also allows you to define the capacity that specifies the number of elements an array list can contain.

However, the default capacity of an `ArrayList` class is 16. If the number of elements in the list reaches the specified capacity, the capacity of the list gets doubled automatically. It can accept null values and can also include duplicate elements.

The `ArrayList` class allows you to add, modify, and delete any type of element in the list even at run-time.

The elements in the `ArrayList` can be accessed by using the index position. While working with the `ArrayList` class, you need not bother about freeing up the memory.

The `ArrayList` class consists of different methods and properties that are used to add and manipulate the elements of the list.

- **Methods**
  - The methods of the `ArrayList` class allow you to perform actions such as adding, removing, and copying elements in the list.
  - The following table displays the commonly used methods of the `ArrayList` class:

Method	Description
Add	Adds an element at the end of the list
Remove	Removes the specified element that has occurred for the first time in the list
RemoveAt	Removes the element present at the specified index position in the list
Insert	Inserts an element into the list at the specified index position
Contains	Determines the existence of a particular element in the list
IndexOf	Returns the index position of an element occurring for the first time in the list
Reverse	Reverses the values stored in the `ArrayList`
Sort	Rearranges the elements in an ascending order

- **Properties**
  - The properties of the `ArrayList` class allow you to count or retrieve the elements in the list. The following table displays the commonly used properties of the `ArrayList` class:

Property	Description
`Capacity`	Specifies the number of elements the list can contain
`Count`	Determines the number of elements present in the list
`Item`	Retrieves or sets value at the specified position

ArrayList Class 3-5

- The following code demonstrates the use of the methods and properties of the `ArrayList` class:

Snippet

```
using System;
using System.Collections;

class ArrayCollection
{
     static void Main(string[] args)
     {
          ArrayList objArray = new ArrayList();
          objArray.Add("John");
          objArray.Add("James");
          objArray.Add("Peter");
          objArray.RemoveAt(2);
          objArray.Insert(2, "Williams");
          Console.WriteLine("Capacity: " + objArray.Capacity);
          Console.WriteLine("Count: " + objArray.Count);
          Console.WriteLine();
          Console.WriteLine("Elements of the ArrayList");
          foreach (string str in objArray)
          {
              Console.WriteLine(str);
          }
     }
}
```

- In the code:
  - The `Add()` method inserts values into the instance of the class at different index positions.
  - The `RemoveAt()` method removes the value `James` from the index position 2 and the `Insert()` method inserts the value `Williams` at the index position 2.
  - The `WriteLine()` method is used to display the number of elements the list can contain and the number of elements present in the list using the Capacity and Count properties respectively.

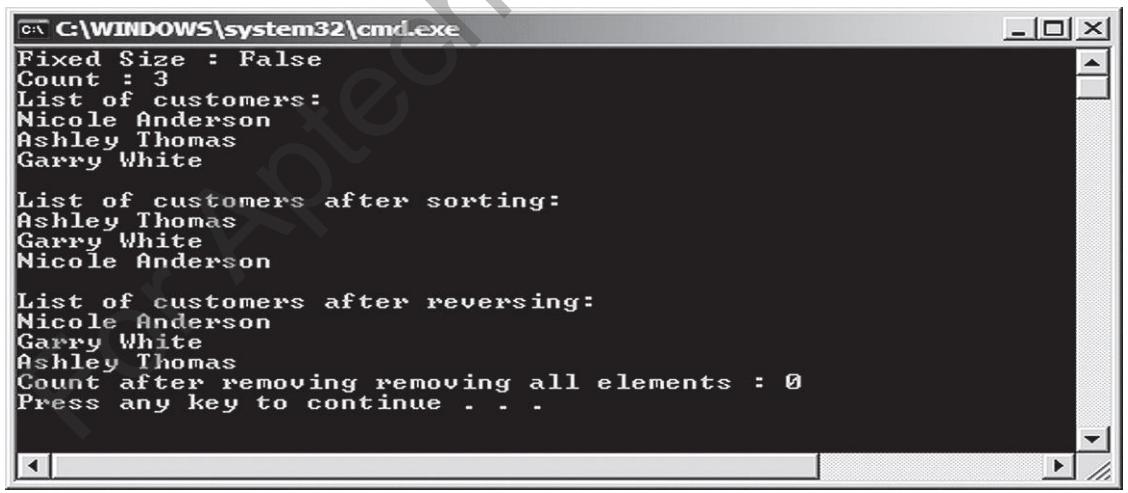
Output

```
Capacity: 4
Count: 3
Elements of the ArrayList
John
James
Williams
```

```
using System;
using System.Collections;
class Customers
{
      static void Main(string[] args)
      {
         ArrayList objCustomers = new ArrayList();
         objCustomers.Add("Nicole Anderson");
         objCustomers.Add("Ashley Thomas");
         objCustomers.Add("Garry White");
         Console.WriteLine("Fixed Size : " +
         objCustomers.IsFixedSize);
         Console.WriteLine("Count : " + objCustomers.Count);
         Console.WriteLine("List of customers:");
         foreach (string names in objCustomers)
         {
             Console.WriteLine("{0}",names);
         }
         objCustomers.Sort();
         Console.WriteLine("\nList of customers after
         sorting:");
         foreach (string names in objCustomers)
         {
            Console.WriteLine("{0}", names);
         }
      objCustomers.Reverse();
      Console.WriteLine("\nList of customers after
      reversing:");
      foreach (string names in objCustomers)
      {
```

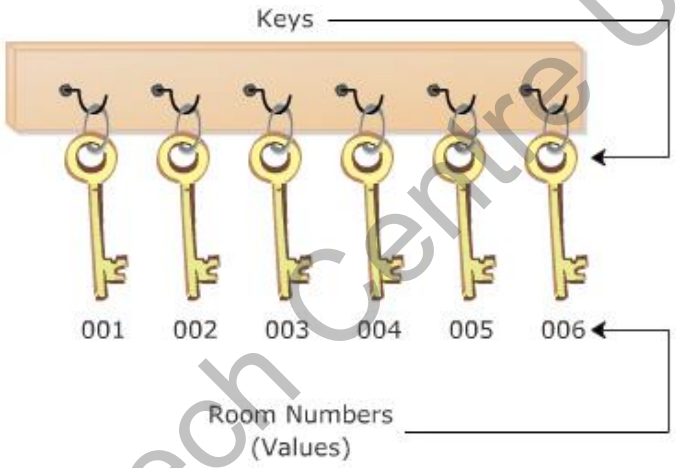
```
            Console.WriteLine("{0}", names);
        }
        objCustomers.Clear();
        Console.WriteLine("Count after removing all elements
        : " + objCustomers.Count);
        }
}
```

- In the code:
  - The `Add()` method inserts value at the end of the `ArrayList`. The values inserted in the array are displayed in the same order before the `Sort()` method is used.
  - The `Sort()` method then displays the values in the sorted order. The `FixedSize()` property checks whether the array is of a fixed size.
  - When the `Reverse()` method is called, it displays the values in the reverse order.
  - The `Clear()` method deletes all the values from the `ArrayList` class.
- The following figure displays the use of methods of the `ArrayList` class:



Hashtable Class 1-7

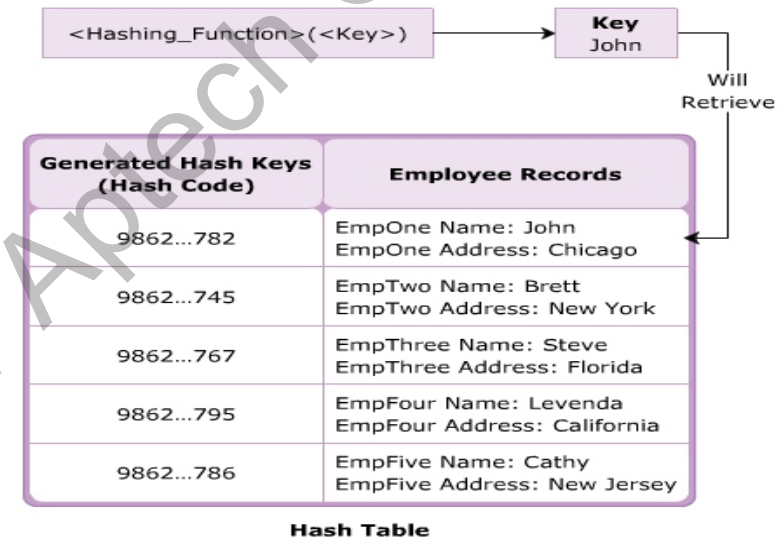
- Consider the reception area of a hotel where you find the keyholder storing a bunch of keys.
- Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.
- The following figure demonstrates a real-world example of unique keys:



- Similar to the keyholder, the `Hashtable` class in C# allows you to create collections in the form of keys and values.
- It generates a `hashtable` which associates keys with their corresponding values.
- The `Hashtable` class uses the `hashtable` to retrieve values associated with their unique key.

Hashtable Class 2-7

- The `hashtable` generated by the `Hashtable` class uses the hashing technique to retrieve the corresponding value of a key.
- Hashing is a process of generating the hash code for the key and the code is used to identify the corresponding value of the key.
- The `Hashtable` object takes the key to search the value, performs a hashing function and generates a hash code for that key.
- When you search for a particular value using the key, the hash code is used as an index to locate the desired record.
- For example, a student name can be used as a key to retrieve the student id and the corresponding residential address. The following figure represents the `Hashtable`:



Hash Table

Hashtable Class 3-7

- The `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the `hashtable`.
- The methods of the `Hashtable` class allow you to perform certain actions on the data in the `hashtable`.
- The following table displays the commonly used methods of the `Hashtable` class:

Method	Description
`Add`	Adds an element with the specified key and value
`Remove`	Removes the element having the specified key
`CopyTo`	Copies elements of the hashtable to an array at the specified index
`ContainsKey`	Checks whether the hashtable contains the specified key
`ContainsValue`	Checks whether the hashtable contains the specified value
`GetEnumerator`	Returns an `IDictionaryEnumerator` that traverses through the `Hashtable`

- **Properties**
  - The properties of the `Hashtable` class allow you to access and modify the data in the `hashtable`.
  - The following figure displays the commonly used properties of the `Hashtable` class:

Property	Description
`Count`	Specifies the number of key and value pairs in the hashtable
`Item`	Specifies the value, adds a new value or modifies the existing value for the specified key
`Keys`	Provides an `ICollection` consisting of keys in the hashtable
`Values`	Provides an `ICollection` consisting of values in the hashtable
`IsReadOnly`	Checks whether the `Hashtable` is read-only

- The following code demonstrates the use of the methods and properties of the `Hashtable` class:

Snippet

```
using System;
using System.Collections;

class HashCollection
{
       static void Main(string[] args)
       {
            Hashtable objTable = new Hashtable();
            objTable.Add(001, "John");
            objTable.Add(002, "Peter");
            objTable.Add(003, "James");
            objTable.Add(004, "Joe");
            Console.WriteLine("Number of elements in the hash table: " +
            objTable.Count);
            ICollection objCollection = objTable.Keys;
            Console.WriteLine("Original values stored in hashtable are:
            ");
            foreach (int i in objCollection)
            {
                Console.WriteLine (i + " : " + objTable[i]);
            }
            if (objTable.ContainsKey(002))
            {
                objTable[002] = "Patrick";
            }
            Console.WriteLine("Values stored in the hashtable after
            removing values");
            foreach (int i in objCollection)
            {
               Console.WriteLine(i + " : " + objTable[i]);
            }
       }
}
```

Output

```
Number of elements in the hashtable: 4
Original values stored in hashtable are:
4 : Joe
3 : James
2 : Peter
1 : John
Values stored in the hashtable after removing values
4 : Joe
3 : James
2 : Patrick
1 : John
```

- In the Code:
  - The `Add()` method inserts the keys and their corresponding values into the instance. The `Count` property displays the number of elements in the `hashtable`.
  - The `Keys` property provides the number of keys to the instance of the `ICollection` interface.
  - The `ContainsKey()` method checks whether the `hashtable` contains the specified key. If the `hashtable` contains the specified key, `002`, the default `Item` property that is invoked using the square bracket notation ([]) replaces the value `Peter` to the value `Patrick`.
  - This output is in the descending order of the key. However, the output may not always be displayed in this order.
  - It could be either in ascending or random orders depending on the hash code.

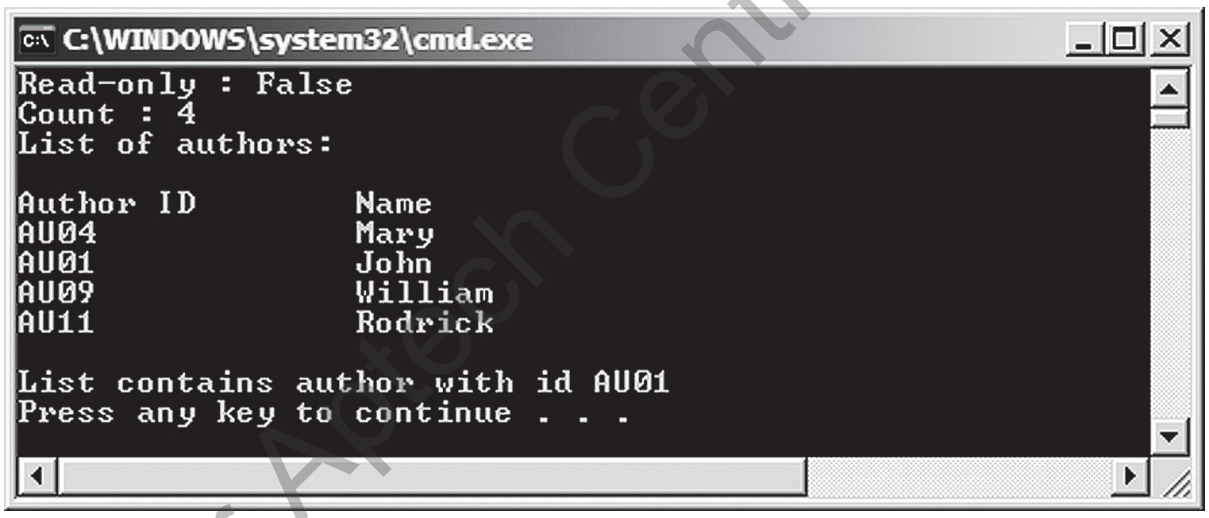
Hashtable Class 6-7

- The Code Snippet demonstrates the use of methods and properties of the `Hashtable` class.

Snippet

```
using System;
using System;
using System.Collections;
class Authors
{
      static void Main(string[] args)
      {
            Hashtable objAuthors = new Hashtable();
            objAuthors.Add("AU01","John");
            objAuthors.Add("AU04", "Mary");
            objAuthors.Add("AU09", "William");
            objAuthors.Add("AU11", "Rodrick");
            Console.WriteLine("Read-only : " +
            objAuthors.IsReadOnly);
            Console.WriteLine("Count : " + objAuthors.Count);
            IDictionaryEnumerator objCollection =
            objAuthors.GetEnumerator();
            Console.WriteLine("List of authors:\n");
            Console.WriteLine("Author ID \t Name");
            while(objCollection.MoveNext())
            {
               Console.WriteLine(objCollection.Key + "\t\t " +
               objCollection.Value);
            }
            if(objAuthors.Contains("AU01"))
            {
                Console.WriteLine("\nList contains author with id
                AU01");
            }
            else
            {
               Console.WriteLine("\nList does not contain author
               with id AU01");
            }
      }
}
```

- In the code:
  - The `Add()` method inserts values in the `Hashtable`.
  - The `IsReadOnly()` method checks whether the values in the array can be modified or not.
  - The `Contains()` method checks whether the value `AU01` is present in the list.
- The following figure displays the `Hashtable` example:



SortedList Class 1-6

- The `SortedList` class represents a collection of key and value pairs where elements are sorted according to the key.
- By default, the `SortedList` class sorts the elements in ascending order, however, this can be changed if an `IComparable` object is passed to the constructor of the `SortedList` class.
- These elements are either accessed using the corresponding keys or the index numbers.
- If you access elements using their keys, the `SortedList` class behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.
- The `SortedList` class consists of different methods and properties that are used to add and manipulate the data in the sorted list.
  - **Methods**
    - The methods of the `SortedList` class allow you to perform certain actions on the data in the sorted list.

- The following table displays the commonly used methods of the `SortedList` class:

Method	Description
`Add`	Adds an element to the sorted list with the specified key and value
`Remove`	Removes the element having the specified key from the sorted list
`GetKey`	Returns the key at the specified index position
`GetByIndex`	Returns the value at the specified index position
`ContainsKey`	Checks whether the instance of the `SortedList` class contains the specified key
`ContainsValue`	Checks whether the instance of the `SortedList` class contains the specified value
`RemoveAt`	Deletes the element at the specified index

- **Properties**
  - The properties of the `SortedList` class allow you to access and modify the data in the sorted list.

Property	Description
`Capacity`	Specifies the number of elements the sorted list can contain
`Count`	Specifies the number of elements in the sorted list
`Item`	Returns the value, adds a new value or modifies the existing value for the specified key
`Keys`	Returns the keys in the sorted list
`Values`	Returns the values in the sorted list

- The following code demonstrates the use of methods and properties of the `SortedList` class:

Snippet

```
using System;
using System.Collections;

class SortedCollection
{
      static void Main(string[] args)
      {
      SortedList objSortList = new SortedList();
      objSortList.Add("John", "Administration");
      objSortList.Add("Jack", "Human Resources");
      objSortList.Add("Peter", "Finance");
      objSortList.Add("Joel", "Marketing");
      Console.WriteLine("Original values stored in the
      sorted list");
      Console.WriteLine("Key \t\t Values");
      for (int i=0; i<objSortList.Count; i++)
      {
          Console.WriteLine(objSortList.GetKey(i) + " \t\t " +
          objSortList.GetByIndex(i));
      }
      if (!objSortList.ContainsKey("Jerry"))
      {
         objSortList.Add("Jerry", "Construction");
      }
      objSortList["Peter"] = "Engineering";
      objSortList["Jerry"] = "Information Technology";
      Console.WriteLine();
      Console.WriteLine("Updated values stored in hashtable");
      Console.WriteLine("Key \t\t Values");
      for (int i = 0; i < objSortList.Count; i++)
      {
          Console.WriteLine(objSortList.GetKey(i) + " \t\t " +
          objSortList.GetByIndex(i));
      }
      }
}
```

- In the code:
  - The `Add()` method inserts keys and their corresponding values into the instance and the `Count` property counts the number of elements in the sorted list.
  - The `GetKey()` method returns the keys in the sorted order from the sorted list while the `GetByIndex()` method returns the values at the specified index position.
  - If the sorted list does not contain the specified key, Jerry, then the `Add()` method adds the key, `Jerry` with its corresponding value.
- The default `Item` property that is invoked using the square bracket notation ([]) replaces the values associated with the specified keys, `Peter` and `Jerry`.

Output

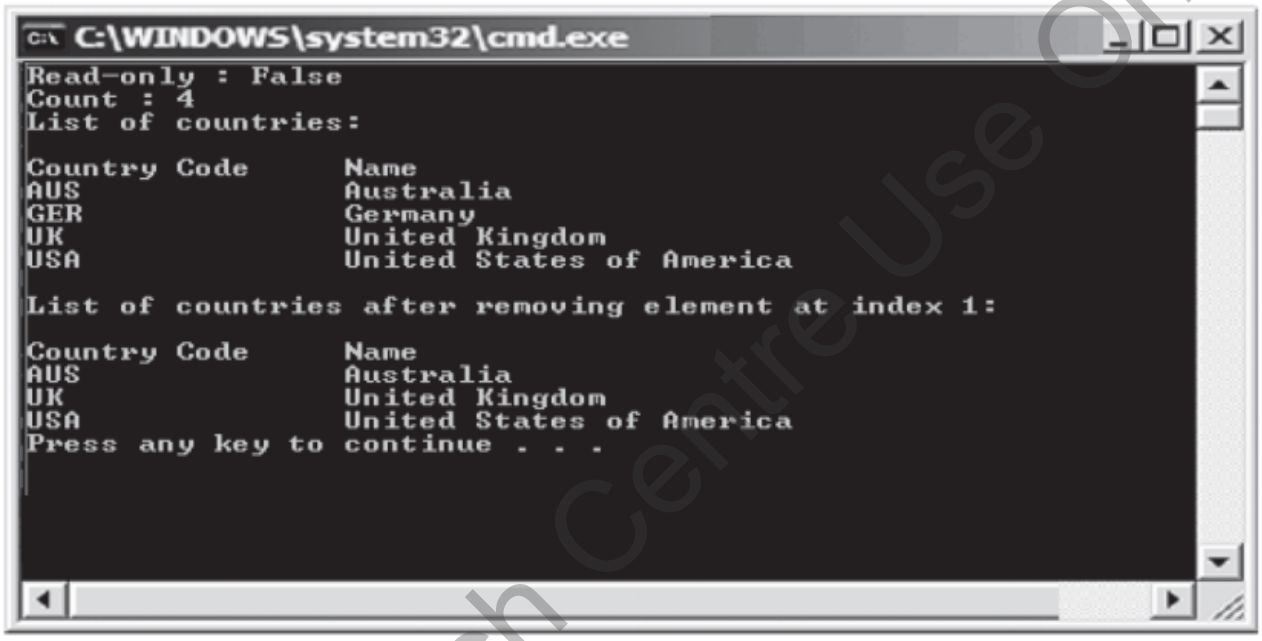
```
Original values stored in the sorted list
Key Values
Jack Human Resources
Joel Marketing
John Administration
Peter Finance
Updated values stored in hashtable
Key Values
Jack Human Resources
Jerry Information Technology
Joel Marketing
John Administration
Peter Engineering
```

- The following code demonstrates the use of methods in the `SortedList` class:

Snippet

```
using System;
using System.Collections;
class Countries
{
       static void Main(string[] args)
       {
          SortedList objCountries = new SortedList();
          objCountries.Add("UK", "United Kingdom");
          objCountries.Add("GER", "Germany");
          objCountries.Add("USA", "United States of America");
          objCountries.Add("AUS", "Australia");
          Console.WriteLine("Read-only : " +
          objCountries.IsReadOnly);
          Console.WriteLine("Count : " + objCountries.Count);
          Console.WriteLine("List of countries:\n");
          Console.WriteLine("Country Code \t Name");
          for ( int i = 0; i < objCountries.Count; i++ )
          {
             Console.WriteLine(objCountries.GetKey(i) + "\t\t
             " + objCountries.GetByIndex(i));
          }
          objCountries.RemoveAt(1);
          Console.WriteLine("\nList of countries after removing
          element at index 1:\n");
          Console.WriteLine("Country Code \t Name");
          for (int i = 0; i < objCountries.Count; i++)
          {
             Console.WriteLine(objCountries.GetKey(i) + "\t\t
             " + objCountries.GetByIndex(i));
          }
       }
}
```

- The following figure displays the `SortedList` class example:



- In the code:
  - The `Add()` method inserts values in the list.
  - The `IsReadOnly()` method checks whether the values in the list can be modified or not. The `GetByIndex()` method returns the value at the specified index.
  - The `RemoveAt()` method removes the value at the specified index.

DictionaryGeneric Class 1-6

- The `System.Collections.Generic` namespace contains a vast number of generic collections.
- One of the most commonly used among these is the `Dictionary` generic class that consists of a generic collection of elements organized in key and value pairs and maps the keys to their corresponding values.
- Unlike other collections in the `System.Collections` namespace, it is used to create a collection of a single data type at a time.
- Every element that you add to the dictionary consists of a value, which is associated with its key and can retrieve a value from the dictionary by using its key.
- The following syntax declares a `Dictionary` generic class:

Syntax

```
Dictionary<TKey, TValue>
```

- where,
  - `TKey`: Is the type parameter of the keys to be stored in the instance of the `Dictionary` class.
  - `TValue`: Is the type parameter of the values to be stored in the instance of the `Dictionary` class.

DictionaryGeneric Class 2-6

- The `Dictionary` generic class consists of different methods and properties that are used to add and manipulate elements in a collection.
  - **Methods**
    - The methods of the `Dictionary` generic class allow you to perform certain actions on the data in the collection.
    - The following table displays the commonly used methods of the `Dictionary` generic class:

Method	Description
`Add`	Adds the specified key and value in the collection
`Remove`	Removes the value associated with the specified key
`ContainsKey`	Checks whether the collection contains the specified key
`ContainsValue`	Checks whether the collection contains the specified value
`GetEnumerator`	Returns an enumerator that traverses through the Dictionary
`GetType`	Retrieves the Type of the current instance

  - **Properties**
    - The properties of the `Dictionary` generic class allow you to modify the data in the collection.
    - The following table displays the commonly used properties of the `Dictionary` generic class:

Property	Description
`Count`	Determines the number of key and value pairs in the collection
`Item`	Returns the value, adds a new value or modifies the existing value for the specified key
`Keys`	Returns the collection containing the keys
`Values`	Returns the collection containing the values

- The following code demonstrates the use of the methods and properties of the `Dictionary` class:

Snippet

```
using System;
using System.Collections;
class DictionaryCollection{
     static void Main(string[] args) {
          Dictionary<int, string> objDictionary = new Dictionary<int,
          string>();
          objDictionary.Add(25, "Hard Disk");
          objDictionary.Add(30, "Processor");
          objDictionary.Add(15, "MotherBoard");
          objDictionary.Add(65, "Memory");
          ICollection objCollect = objDictionary.Keys;
          Console.WriteLine("Original values stored in the collection");
          Console.WriteLine("Keys \t Values");
          Console.WriteLine("----------------");
          foreach (int i in objCollect){
             Console.WriteLine(i + " \t " + objDictionary[i]);
          }
          objDictionary.Remove(65);
          Console.WriteLine();
          if (objDictionary.ContainsValue("Memory")) {
              Console.WriteLine("Value Memory could not be deleted");
          }
          else {
              Console.WriteLine("Value Memory deleted successfully");
          }
          Console.WriteLine();
          Console.WriteLine("Values stored after removing element");
          Console.WriteLine("Keys \t Values");
          Console.WriteLine("----------------");
          foreach (int i in objCollect) {
              Console.WriteLine(i + " \t " + objDictionary[i]);
          }
     }
}
```

- In the code:
  - The `Dictionary` class is instantiated by specifying the `int` and `string` data types as the two parameters.
  - The `int` data type indicates the keys and the `string` data type indicates values.
  - The `Add()` method inserts keys and values into the instance of the `Dictionary` class.
  - The `Keys` property provides the number of keys to the instance of the `ICollection` interface.
  - The `ICollection` interface defines the size and synchronization methods to manipulate the specified generic collection.
  - The `Remove()` method removes the value `Memory` by specifying the key associated with it, which is `65`.
  - The `ContainsValue()` method checks whether the value `Memory` is present in the collection and displays the appropriate message.

Output

```
Original values stored in the collection
Keys Values
----------------
25 Hard Disk
30 Processor
15 MotherBoard
65 Memory
Value Memory deleted successfully
Values stored after removing element
Keys Values
----------------
25 Hard Disk
30 Processor
15 MotherBoard
```

DictionaryGeneric Class 5-6

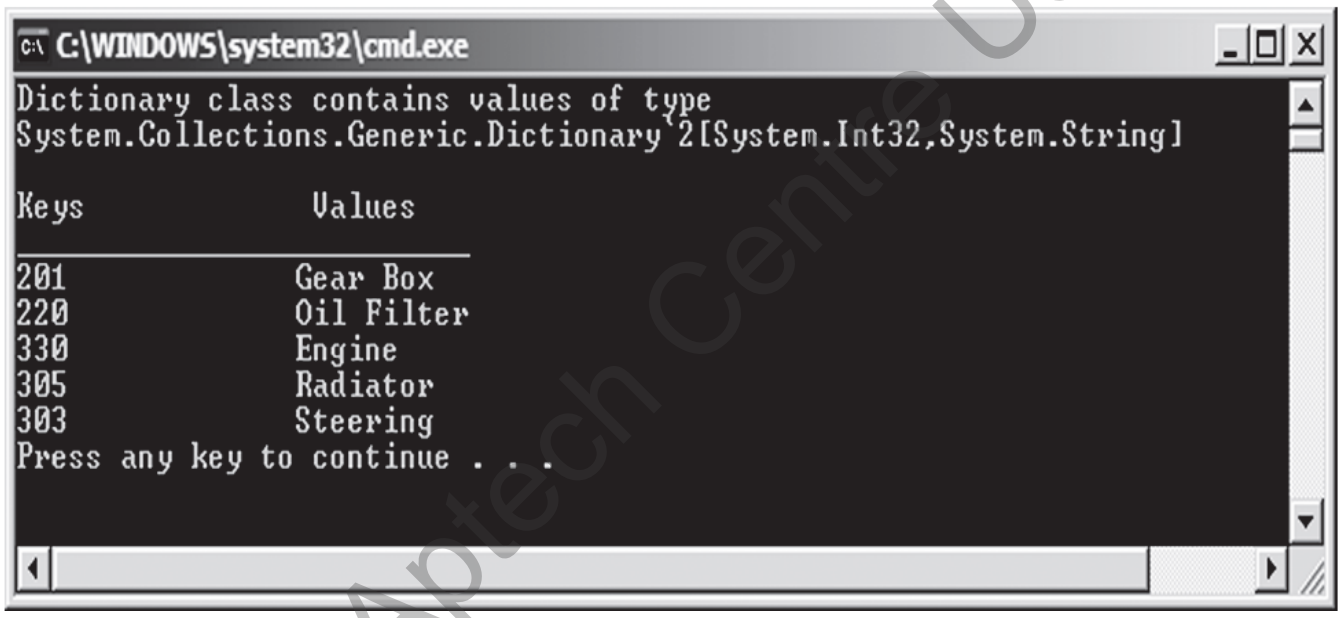
- The following code demonstrates the use of methods in `Dictionary` generic class:

Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;
class Car
{
        static void Main(string[] args)
        {
            Dictionary<int, string> objDictionary = new
            Dictionary<int,string>();
            objDictionary.Add(201, "Gear Box");
            objDictionary.Add(220, "Oil Filter");
            objDictionary.Add(330, "Engine");
            objDictionary.Add(305, "Radiator");
            objDictionary.Add(303, "Steering");
            Console.WriteLine("Dictionary class contains values
            of type");
            Console.WriteLine(objDictionary.GetType());
            Console.WriteLine("Keys \t\t Values");
            Console.WriteLine("_____________________________");
            IDictionaryEnumerator objDictionayEnumerator =
            objDictionary.GetEnumerator();
            while (objDictionayEnumerator.MoveNext ())
            {
            Console.WriteLine(objDictionayEnumerator.Key.ToString()
            + "\t\t"+ objDictionayEnumerator.Value);
            }
        }
}
```

DictionaryGeneric Class 6-6

- In the code:
  - The `Add()` method inserts values into the list.
  - The `GetType()` method returns the type of the object.
- The following figure displays the use of `Dictionary` generic class:



Collection Initializers 1-2

- Collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implements the `Ienumerable` interface using element initializers.
- The element initializers can be a simple value, an expression, or an object initializer.
- When a programmer uses collection initializer, the programmer is not required to provide multiple `Add()` methods to add elements to the collection, making the code concise. It is the responsibility of the compiler to provide the `Add()` methods when the program is compiled.
- The following code uses a collection initializer to initialize an `ArrayList` with integers:

Snippet

```
using System;
using System.Collections;

class Car
{
    static void Main (string [] args)
   {
       ArrayList nums=new ArrayList{1,2,3*6,4,5};
       foreach (int num in nums)
       {
           Console.WriteLine("{0}", num);
       }
 }
}
```

Output

```
1
2
18
4
5
```

Collection Initializers 2-2

- In the code:
  - The `Main()` method uses collection initializer to create an `ArrayList` object initialized with integer values and expressions that evaluates to integer values.
  - As collection often contains objects, collection initializers accept object initializers to initialize a collection.
  - The following code shows a collection initializer that initializes a generic `Dictionary` object with an integer keys and `Employee` objects:

Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;

class Employee{
        public String Name { get; set; }
        public String Designation { get; set; }
}
class CollectionInitializerDemo{
      static void Main(string[] args)
      {
      Dictionary<int, Employee> dict = new Dictionary<int,
Employee>(){
        { 1, new Employee {Name="Andy Parker",
Designation="Sales Person"}}, { 2, new Employee {Name="Patrick Elvis",
Designation="Marketing Manager"}}
        };
}
}
```

- In the code:
  - The `Employee` class consists of two public properties: `Name` and `Designation`.
  - The `Main()` method creates a `Dictionary<int, Employee>` object and encloses the collection initializers within a pair of braces.
  - For each element, added to the collection, the innermost pair of braces encloses the object initializer of the `Employee` class.

Summary

- A delegate in C# is used to refer to a method in a safe manner.
- An event is a data member that enables an object to provide notifications to other objects about a particular action.
- The `System.Collections.Generic` namespace consists of generic collections that allow reusability of code and provide better type-safety.
- The `ArrayList` class allows you to increase or decrease the size of the collection during program `executionHashtable` class.
- The `Hashtable` class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the `hashtable` is uniquely identified by its key.
- The `SortedList` class allows you to store elements as key and value pairs where the data is sorted based on the key.
- The `Dictionary` generic class represents a collection of elements organized in key and value pairs.