Session 10: Events, Delegates, and Classes

Objectives

- Explain delegates
- Explain events
- Define and describe collections

Delegates

- 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 must 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.

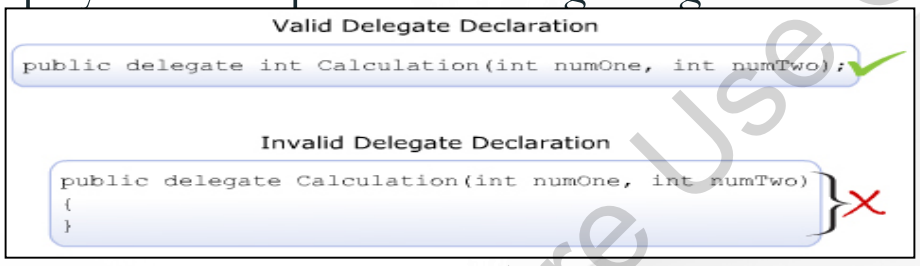
Delegates in C#

- 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.

- 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.

Declaring Delegates

- Delegates in C# are declared using the delegate keyword followed by the return type and the parameters of the referenced method.
- Following figure displays an example of declaring delegates:



- Following syntax is used to declare a delegate:

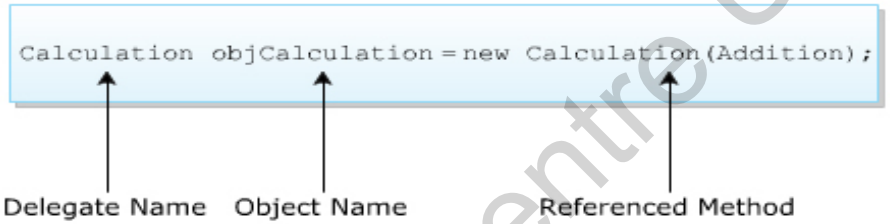
Syntax

```
<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.

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.
- The created object is used to invoke the associated method at run-time.
- Following figure displays an example of instantiating delegates:



- 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

- 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));
    }
}
```

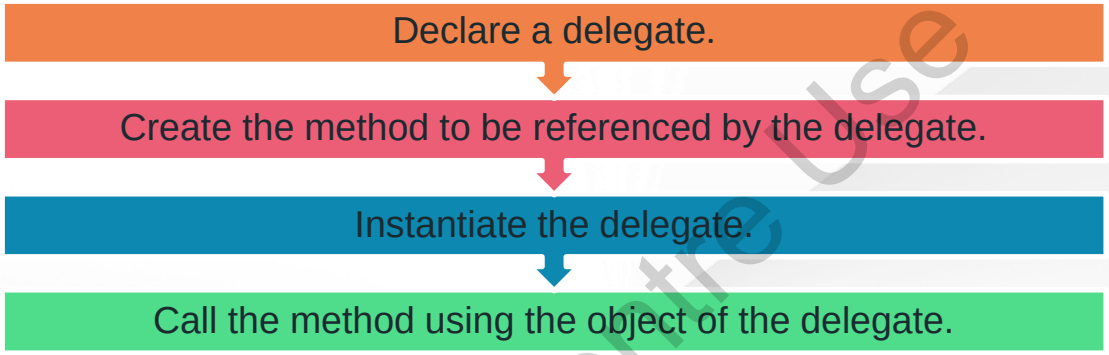
- In the code:
  - The delegate called **Calculation** is declared outside the class **Mathematics.**
  - In the Main() method, an object of the delegate is created that takes the **Addition()** method as the parameter. The parameter type of the method and that of the delegate is the same, which is type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

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.

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.
- 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:

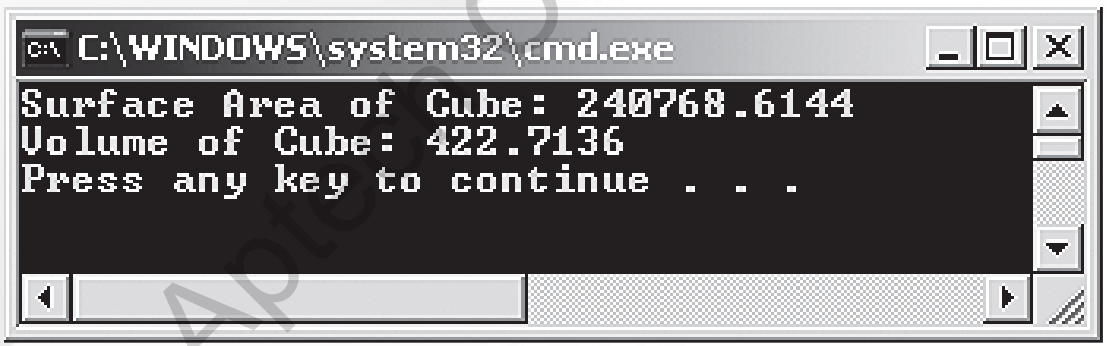
Snippet

```
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.
- Following figure shows the use of multiple delegates:



Multicast Delegates

- 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.
- 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.
- 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.

System.Delegate Class

- 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.
- Following table lists the constructors defined in the Delegate class:

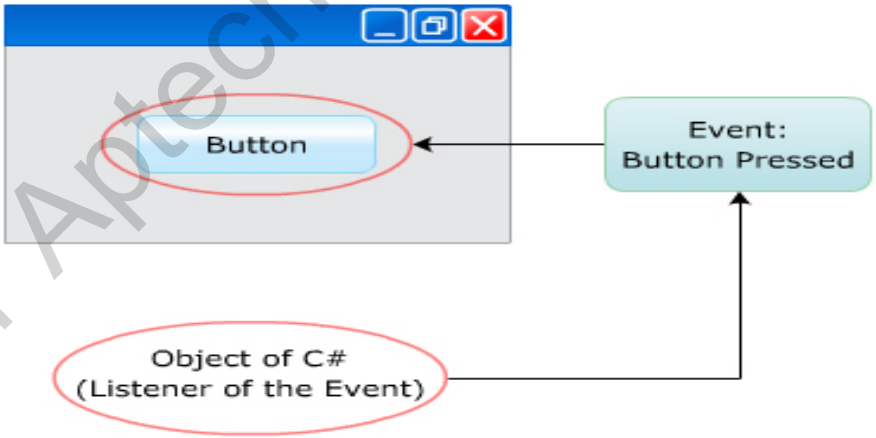
Constructor	Description
Delegate(object, string)	Calls a method referenced by the object of the class given as the parameter
Delegate(type, string)	Calls a static method of the class given as the parameter

- 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

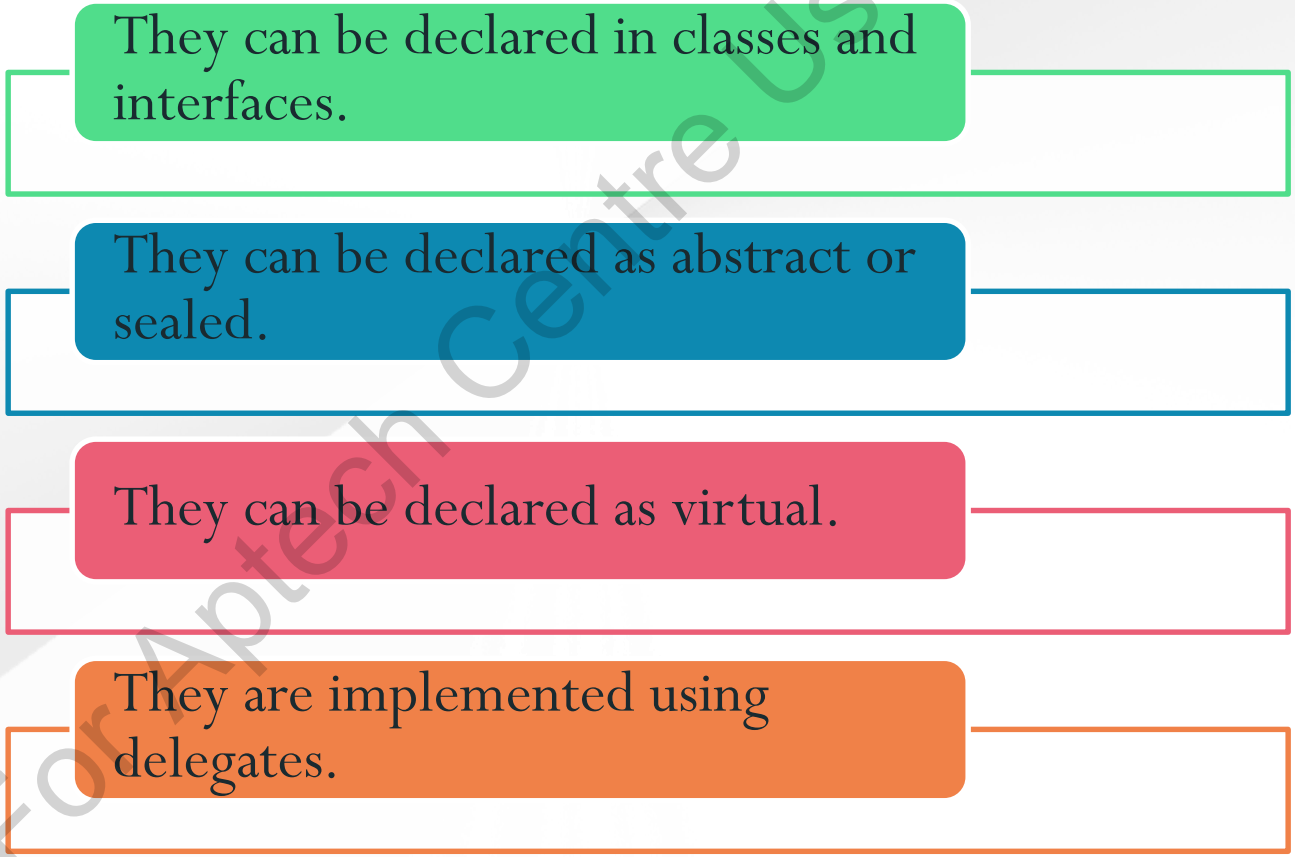
Events

- 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).
- 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:



Creating and using Events

- 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

- 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.
- 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.

Raising Events

- 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`.
- 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

Events and Inheritance 1-2

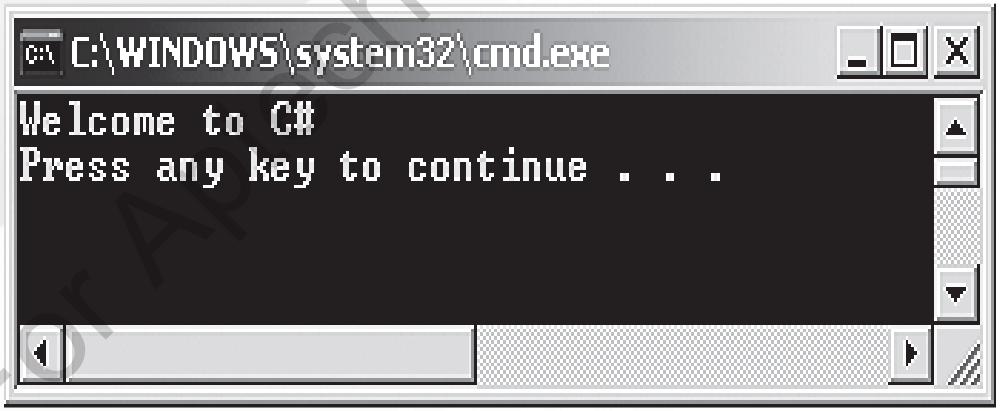
- Events in C# can only be invoked in the class in which they are declared and defined. Therefore, events cannot be directly invoked by the derived classes. Following code illustrates how an event can be indirectly invoked:

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();
        }
}
```

Events and Inheritance 2-2

- 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**.
- 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#.
- 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 1-2

- 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 consists of classes and interfaces that define the different collections.
- 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

System.Collections Namespace 2-2

Class/Interface	Description
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.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.

Classes and Interfaces 1-2

- 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.
    - 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

Classes and Interfaces 2-2

- **Interfaces and Structures**
  - The `System.Collections.Generic` namespace consists of interfaces and structures that can be implemented to create type-safe collections.
  - 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

ArrayList Class 1-2

- 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 do not have to 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.

ArrayList Class 2-2

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

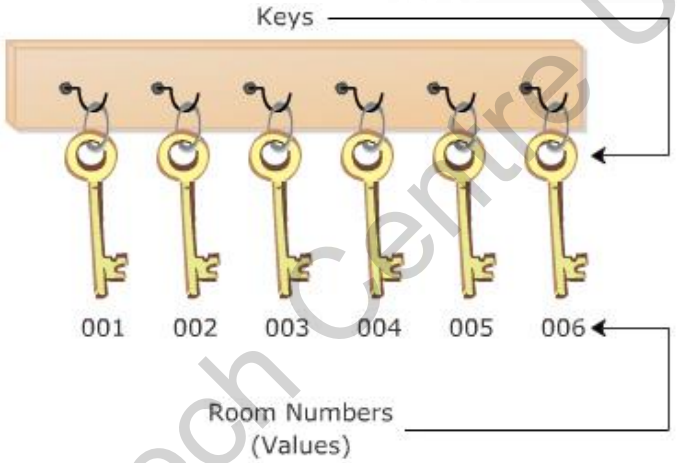
Method	Description
`Add`	Adds an element at the end of the list
`Remove`	Removes the specified element that has occurred for the first time in the list
`RemoveAt`	Removes the element present at the specified index position in the list
`Insert`	Inserts an element into the list at the specified index position
`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. 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

Hashtable Class 1-5

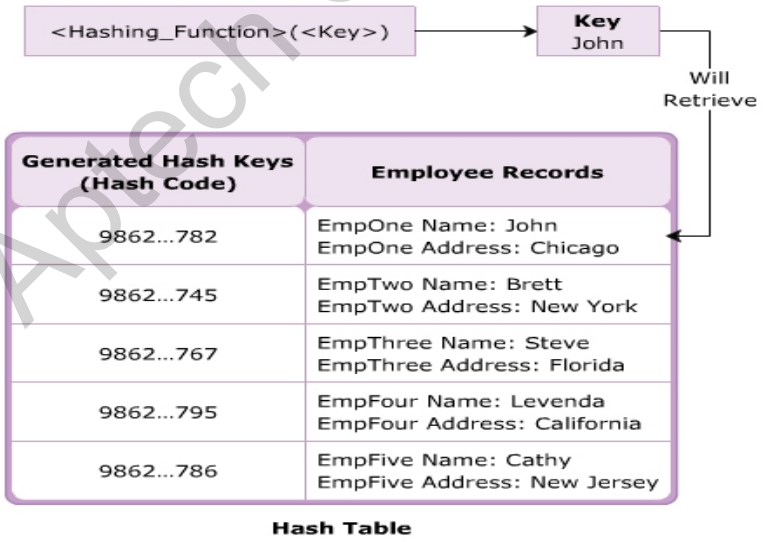
- Consider the reception area of a hotel where you find the keyholder storing a bunch of keys.
- Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.
- Following figure demonstrates a real-world example of unique keys:



- 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 2-5

- 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. Following figure represents the `Hashtable`:



Hash Table

Hashtable Class 3-5

- 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.
- 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.
  - 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

Hashtable Class 4-5

- 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]);
            }
       }
}
```

Hashtable Class 5-5

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.

SortedList Class 1-2

- 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.
- 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.

SortedList Class 2-2

- 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

Dictionary Generic Class 1-2

- 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.
- 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.
- 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.

Dictionary Generic Class 2-2

- 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.
    - 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.
    - 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

Collection Initializers

- 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.
- 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
```

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.
- 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.