UNIT 5

Collections:

In C#, collection represents group of objects. By the help of collections, we can perform various operations on objects Collection types implement the following common functionality:

- Adding and inserting items to a collection
- Removing items from a collection
- Finding, sorting, searching items
- Replacing, copy items
- Capacity and Count properties to find the capacity of the collection and number of items in the collection

.NET supports two types of collections:

- Generic collections
- Non-generic collections.

1. Generic Collections

Generic Collections work on the specific type that is specified in the program. Some characteristics of generic collection are:

1. Works on specific type
2. Array Size is not fixed
3. Elements can be added / removed at runtime.
4. Examples of Generic collections are List, Dictionary etc.

LIST

The List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the generic version of the ArrayList that comes under System.Collections.Generic namespace.

List<T> Characteristics

- List<T> equivalent of the ArrayList, which implements IList<T>.
- It comes under System.Collections.Generic namespace.
- List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
- Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
- List<T> performs faster and less error-prone than the ArrayList.

In other word, a list is an object which holds variables in a specific order. The type of variable that the list can store is defined using the generic syntax. Here is an example of defining a list called numbers which holds integers.

```
List<int> numbers = new List<int>();
```

The difference between a list and an array is that lists are dynamic sized, while arrays have a fixed size. When you do not know the amount of variables your array should hold, use a list instead.

List<T> Class Properties and Methods

The following table lists the important properties and methods of List<T> class:

Property	Usage
Items	Gets or sets the element at the specified index
Count	Returns the total number of elements exists in the List<T>
Method	**Usage**
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the specified element exists or not in a List<T>.
Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
InsertRange	Inserts elements of another collection at the specified index.
Remove	Removes the first occurrence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.
TrueForAll	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

```
using System;
using System.Collections.Generic;
class Program {
  static void Main() {
    List<string> sports = new List<string>();
    sports.Add("Football"); // add method
    sports.Add("Tennis");
    sports.Add("Soccer");
    Console.WriteLine("Old List...");
    Console.WriteLine("Capacity Is: " + sports.Capacity);

      // Printing the Count of firstlist
      Console.WriteLine("Count Is: " + sports.Count);
    foreach (string s in sports) {
      Console.WriteLine(s);
    }
     Console.WriteLine("New List...");
    sports.Remove("Tennis"); // remove method
     sports.RemoveAt(0);
    foreach (string s in sports) {
      Console.WriteLine(s);}}}
```

Old List...
Capacity Is: 4
Count Is: 3
Football
Tennis
Soccer
New List...
Soccer

Dictionary:

The Dictionary<TKey, TValue> is a generic collection that stores key-value pairs in no particular order.

Dictionary Characteristics

- Dictionary<TKey, TValue> stores key-value pairs.
- Comes under System.Collections.Generic namespace.
- Implements IDictionary<TKey, TValue> interface.
- Keys must be unique and cannot be null.
- Values can be null or duplicate.
- Values can be accessed by passing associated key in the indexer e.g. myDictionary[key]
- Elements are stored as KeyValuePair<TKey, TValue> objects.

In other words, Dictionaries are special lists, whereas every value in the list has a key which is also a variable. A good example for a dictionary is a phone book.

```
Dictionary<string, long> phonebook = new Dictionary<string, long>();
```

Properties

Comparer	Gets the IEqualityComparer<T> that is used to determine equality of keys for the dictionary.
Count	Gets the number of key/value pairs contained in the Dictionary<TKey,TValue>.
Item[TKey]	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the Dictionary<TKey,TValue>.
Values	Gets a collection containing the values in the Dictionary<TKey,TValue>.

Methods

Add(TKey, TValue)	Adds the specified key and value to the dictionary.
Clear()	Removes all keys and values from the Dictionary<TKey,TValue>.
ContainsKey(TKey)	Determines whether the Dictionary<TKey,TValue> contains the specified key.
ContainsValue(TValue)	Determines whether the Dictionary<TKey,TValue> contains a specific value.

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Dictionary<TKey,TValue>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize the Dictionary<TKey,TValue> instance.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(TKey)	Removes the value with the specified key from the Dictionary<TKey,TValue>.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

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

public class Demo {
  public static void Main() {
    Dictionary<int, int> kv = new Dictionary<int, int>();
    kv.Add(1,97);
    kv.Add(2,89);
    kv.Add(3,77);
    kv.Add(4,88);
     foreach (KeyValuePair<int, int> d in kv)
      {
         Console.WriteLine(d.Key+" "+d.Value);
      }
    // Dictionary elements
    Console.WriteLine("Dictionaly elements: "+kv.Count);
     kv.Remove(1);
     Console.WriteLine("After remove operation");

      foreach (KeyValuePair<int, int> d in kv)
      {
         Console.WriteLine(d.Key+" "+d.Value);
      }
  }
}
1 97
2 89
3 77
4 88
Dictionaly elements: 4
```

After remove operation
2 89
3 77
4 88

Difference between List and Dictionary

List

- List in C# is used for storing a collection of elements of same type.
- List is an ordered collection. Elements are stored in the order they were added.
- List allows duplicate values.
- You must iterate through the list to find a specific element. This can be slow for large lists.
- List uses less memory because it only stores elements.
- Index Out of Range: Accessing an index beyond the List's size results in an exception. Searching can be slow for large Lists.

Dictionary

- Dictionary in C# is used for storing key-value pairs where each key is unique.
- Dictionary stores Key-Value pairs where elements are accessed by a unique key.
- Dictionary requires unique keys. Adding a duplicate key will overwrite the existing value.
- Dictionary Offers constant-time lookup for values based on keys. It's very-fast for retrieval.
- Dictionary uses more memory because it stores both keys and values.
- Key Not Found: Trying to access a key that doesn't exist in a Dictionary throws an exception.

2. Non-Generic

In non-generic collections, each element can represent a value of a different type. The collection size is not fixed. Items from the collection can be added or removed at runtime.

1. ArrayList

In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same as Array except that its size increases dynamically. An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

ArrayList Properties

Properties	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

ArrayList Methods

Methods	Description
Add()/AddRange()	Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList.
Insert()/InsertRange()	Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList.

Methods	Description
Remove()/RemoveRange()	Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList.
RemoveAt()	Removes the element at the specified index from the ArrayList.
Sort()	Sorts entire elements of the ArrayList.
Reverse()	Reverses the order of the elements in the entire ArrayList.
Contains	Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false.
Clear	Removes all the elements in ArrayList.
CopyTo	Copies all the elements or range of elements to compitible Array.
GetRange	Returns specified number of elements from specified index from ArrayList.
IndexOf	Search specified element and returns zero based index if found. Returns -1 if element not found.
ToArray	Returns compitible array from an ArrayList.

Create an ArrayList

The ArrayList class included in the System.Collections namespace. Create an object of the ArrayList using the new keyword.

Example: Create an ArrayList

```
using System.Collections;
ArrayList arlist = new ArrayList();
// or
var arlist = new ArrayList(); // recommended
using System;
using System.Collections;

  class GFG {

      public static void Main()
    {

      // Creating an ArrayList
      ArrayList myList = new ArrayList(10);

      // Adding elements to ArrayList
      myList.Add(2);
      myList.Add(3);
      myList.Add(4);
      myList.Add(5);
      myList.Add(6);
      myList.Add(7);
       myList.Remove(7);
      myList.RemoveAt(1);
      // Displaying the elements in ArrayList
      Console.WriteLine("The elements in ArrayList are :");
```

```
        foreach(int i in myList)
            Console.WriteLine(i);
    }
}
```

The elements in ArrayList are :

2 4 5 6

2. **HashTable**

The Hashtable is a non-generic collection that stores key-value pairs, similar to generic Dictionary<TKey, TValue> collection. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

Hashtable Characteristics

- Hashtable stores key-value pairs.
- Comes under System.Collections namespace.
- Implements IDictionary interface.
- Keys must be unique and cannot be null.
- Values can be null or duplicate.
- Values can be accessed by passing associated key in the indexer e.g. myHashtable[key]
- Elements are stored as DictionaryEntry objects.

Create a Hashtable

To create Hashtable in C#, we need to use the System.Collections namespace. Here is how we can create a Hashtable in C#.

```
// create a hashtable
        Hashtable myTable = new Hashtable();
        using System;
        using System.Collections; //must
        namespace ConsoleApplication
        {
            class Program
            {
                static void Main(string[] args)
                {
```

```
            Hashtable HT = new Hashtable();
            HT.Add(1,"s");
            HT.Add(3, "n");
            HT.Add(4, "j");
            HT.Add(2, "a");
            HT.Add(5, "u");

            foreach (DictionaryEntry di in HT)
              Console.WriteLine("keys={0} values={1}", di.Key, di.Value);

                }
    }
}
keys=5 values=u
keys=4 values=j
keys=3 values=n
keys=2 values=a
keys=1 values=s
```

Basic Operations on Hashtable

.

Add(Object, Object)
Adds an element with the specified key and value into the Hashtable.
Clear()
Removes all elements from the Hashtable.
ContainsKey(Object)
Determines whether the Hashtable contains a specific key.
ContainsValue(Object)
Determines whether the Hashtable contains a specific value.
Remove(Object)
This method is used to remove the element with the specified key from the hashtable.

Generics

In C#, generic means not specific to a particular data type. C# allows you to define generic classes, interfaces, abstract classes, fields, methods, static methods, properties, events, delegates, and

operators using the type parameter and without the specific data type. A type parameter is a placeholder for a particular type specified when creating an instance of the generic type.

A generic type is declared by specifying a type parameter in an angle brackets after a type name, e.g. TypeName<T> where T is a type parameter.

Generic Class

Generic classes are defined using a type parameter in an angle brackets after the class name. The following defines a generic class.

```
using System;

public class Student<T>
{
   // define a variable of type T
   public T data;

   // define a constructor of the Student class
   public Student(T data)
   {
      this.data = data;
      Console.WriteLine("Data passed: " + this.data);
   }
}

class Program
{
   static void Main()
   {
      // create an instance with data type string
      Student<string> studentName = new Student<string>("Avicii");

      // create an instance with data type int
      Student<int> studentId = new Student<int>(23);
   }
}
```

Output

```
Data passed: Avicii
Data passed: 23thod.
```

Advantages of Generics

1. Generics increase the reusability of the code. You don't need to write code to handle different data types.
2. Generics are type-safe. You get compile-time errors if you try to use a different data type than the one specified in the definition.
3. Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

Comparing and Sorting

The .NET Framework includes a standard set of interfaces that are used for comparing and sorting objects. Although implementing these interfaces is optional, any class that does implement them can interact with other .NET Framework classes to achieve greater functionality.

Creating Comparable Types with the IComparable Interface

The IComparable interface can be implemented by types to provide a standard way of comparing multiple objects. By implementing IComparable, types can maintain reference equality semantics while providing a standard method for value comparison.

The IComparable interface is declared as follows:

```
interface IComparable
{
   int CompareTo(object obj);
}
```

The CompareTo method returns one of the following three possible values:

- Less than 0 if the current object compares as less than obj
- 0 if the current object compares as equal to obj
- Greater than 0 if the current object compares as greater than obj

Using the Built-In Comparison Classes

The .NET Framework includes two classes that implement the IComparer interface and can be leveraged in your own classes to simplify comparisons.

- Comparer
- CaseInsensitiveComparer

The Comparer and CaseInsensitiveComparer classes provide basic implementations of IComparer that are suitable for many comparison classes. You never directly create an instance of the Comparer class using the new operator. Instead, you call the Default method, which returns a properly initialized Comparer object, as shown here:

```
Comparer theComparer = Comparer.Default;
```

The CaseInsensitiveComparer class works much like the Comparer class, comparing any two objects through the IComparable interface, except that it performs a case-insensitive comparison on strings that it encounters. Like the Comparer class, the static Default method is used to obtain a reference to a properly initialized CaseInsensitiveComparer object, as shown here:

```
CaseInsensitiveComparer aComparer = CaseInsensitiveComparer.Default;
```

Sorting:
In C#, we can do sorting using the built-in Sort/OrderBy methods with the Comparison delegate, the IComparer, and IComparable interfaces.

C# List Sort method

The Sort method sorts the elements or a portion of the elements in the list.

The method has four overloads:

- Sort(Comparison<T>) - Sorts the elements in the entire List<T> using the specified Comparison<T>.
- Sort(Int32, Int32, IComparer<T>) - Sorts the elements in a range of elements in List<T> using the specified comparer.
- Sort() - Sorts the elements in the entire List<T> using the default comparer.
- Sort(IComparer<T>) - Sorts the elements in the entire List<T> using the specified comparer.

The sorting algorithms are already built into the standard library of the language. If the data is not sorted naturally, we need to provide a comparison method (either a class method or a lambda expression) which tells the underlying sorting algorithm how to sort the data. What attributes to sort and in what way.

WinForms: Introduction

Windows Forms is a Graphical User Interface(GUI) class library which is bundled in . Net Framework. Its main purpose is to provide an easier interface to develop the applications for desktop, tablet, PCs. It is also termed as the WinForms.WinForms applications can contain the different type of controls like labels, list boxes, tooltip etc.

First, open the Visual Studio then Go to **File -> New -> Project** to create a new project and then select the language as *Visual C#* from the left menu. Click on *Windows Forms App(.NET Framework)* in the middle of current window. After that give the project name and Click **OK**.

Control

Most forms are designed by adding controls to the surface of the form to define a user interface (UI). A *control* is a component on a form used to display information or accept user input.

- Controls can be added to the Windows forms via the Toolbox in Visual Studio. Controlssuch as labels, checkboxes, radio buttons, etc. can be added to the form via the toolbox.
- One can also use advanced controls like the tree view control and the PictureBox control.
- Event handlers are used to respond to events generated from controls. The most commonone is the one added for the button clicked event.

Control	Description
Button	Fires an event when a mouse click occurs or the Enter or Esc key is pressed. Represents a button on a form. Its text property determines the caption displayed on the button's surface.
CheckBox	Permits a user to select one or more options.Consists of a check box with text or an image beside it. The check box can also be represented as a button by setting: checkBox1.Appearance = Appearance.Button.
CheckedListBox	Displays list of items. ListBox with checkbox preceding each item in list.
ComboBox	Provides TextBox and ListBox functionality. Hybrid control that consists of a textbox and a drop-down list. It combines properties from both the TextBox and the ListBox.
GridView DataGridView	Manipulates data in a grid format. The DataGridView is the foremost control to represent relational data. It supports binding to a database. The DataGridView was introduced in .NET 2.0 and supersedes the DataGrid.
GroupBox	Groups controls. Use primarily to group radio buttons; it places a border around the controls it contains.
ImageList	Manages a collection of images. Container control that holds a collection of images used by other controls such as the ToolStrip, ListView, and TreeView.
Label	Adds descriptive information to a form. Text that describes the contents of a control or instructions for using a control or form.
ListBox	Displays a list of items—one or more of which may be selected. May contain simple text or objects. Its methods, properties, and events allow items to be selected, modified, added, and sorted.
ListView	Displays items and subitems. May take a grid format where each row represents a different item and sub-items. It also permits items to be displayed as icons.
RadioButton	Permits user to make one choice among a group of options. Represents a Windows radio button.
TextBox	Accepts user input. Can be designed to accept single- or multi-line input. Properties allow it to mask input for passwords, scroll, set letter casing automatically, and limit contents to read-only.

Menus

- An imperative part of the user interface in a Windows-based application is the menu.
- A menu on a form is created with a Menu object, which is a collection of MenuItem objects.
- You can add menus to Windows Forms at design time by adding the Menu control and then adding menu items to it using the Menu Designer.
- Menus can also be added programmatically by creating one or more Menu controlsobjects in to a form and adding MenuItem objects to the collection.
- Different types of menus available in C# are
  - MenuStrip –used to create normal horizontal or vertical menus.
  - ContextMenu- used to create popup menus. Menus appears when the mouse right clicked and disappears once selection is made.
  - ToolStrip- used to create menus when there is less space and these are called as dropdown menus.
  - StatusStrip- used to create status bar where status of form controls can be displayed.

Context menus

- The ContextMenu class represents shortcut menus that can be displayed when the userclicks the right mouse button over a control or area of the form. Shortcut menus are typically used to combine different menu items from a MainMenu of a form that are useful for the user given the context of the application.
- They are called context menus because the menu is usually specific to the object overwhich the mouse was clicked.

- Context menus are also sometimes referred to as Popup menus or Shortcut menus.
- Context menus are added to a form in C# using the ContextMenuStrip object andassociating it with a particular control.
- For example, you can use a shortcut menu assigned to a TextBox control to provide menuitems for changing the font of the text.
- Visible controls and Form have a ContextMenu property that binds the ContextMenu class to the control that displays the shortcut menu. More than one control can use a ContextMenu.

Type of items in ContextMenuStrip

1. **MenuItem (ToolStripMenuItem):** It is used to give a simple menu item like "Exit" in the above example.
2. **ComboBox(ToolStripComboBox):** It is used to insert a ComboBox in the context menu where the user can select or type an item in the ComboBox.
3. **Separator (ToolStripSeparator):** It is used to put a horizontal line (ruler) between menu items.
4. **TextBox (ToolStripTextBox):** It is used to put a TextBox in the context menu where the user can enter an item.

MenuStrip

- MenuStrip adds a menu bar to your Windows Forms program. With this control, we add a menuarea and then add the default menus or create custom menus directly in Visual Studio.
- We can create a MenuStrip control using a Forms designer at design-time or using the MenuStripclass in code at run-time or dynamically.
- To create a MenuStrip control at design-time, you simply drag and drop a MenuStrip control from Toolbox to a Form in Visual Studio. After you drag and drop a MenuStrip on a Form, theMenuStrip1 is added to the Form
- Once a MenuStrip is on the Form, you can add menu items and set its properties and events.

- Creating a MenuStrip control at run-time is merely a work of creating an instance of MenuStripclass, setting its properties and adding MenuStrip class to the Form controls.
- The following are the ToolStripItems which can be added as menu items to the ContextMenuStripEx control.
    - MenuItem
    - TextBox
    - ComboBox
    - Separator

Type of items in MenuStrip

- **MenuItem (ToolStripMenuItem):** It is used to give a simple menu item like "Exit" in the above example.
- **ComboBox(ToolStripComboBox):** It is used to insert a ComboBox in the context menu where the user can select or type an item in the ComboBox.
- **Separator (ToolStripSeparator):** It is used to put a horizontal line (ruler) between menu items.
- **TextBox (ToolStripTextBox):** It is used to put a TextBox in the context menu where the user can enter an item.

ToolbarStrip

- ToolStrip is the base class for MenuStrip, StatusStrip, and ContextMenuStrip.
- Use ToolStrip and its associated classes in new Windows Forms applications to create toolbarsthat can have a Windows XP, Office, Internet Explorer, or custom appearance
- The ToolStrip class provides many members that manage painting, mouse and keyboard input,and drag-and-drop functionality.
- Use the ToolStripRenderer class with the ToolStripManager class to gain even more control and customizability over the painting and layout style of all ToolStrip controls on a Windows Form.

How to use ToolStrip Control

Drag and drop ToolStrip Control from toolbox on the window Form.Add ToolStrip Item which you want to show. Add one of the items in your ToolStrip that derives from ToolStrip Item. Example Button is added. Create event handlers for the above menu items

The following items are available by default at design time for the ToolStrip control:

- ToolStripButton
- ToolStripSeparator
- ToolStripLabel
- ToolStripDropDownButton
- ToolStripSplitButton
- ToolStripTextBox
- ToolStripComboBox

Graphics and GDI

- The common language runtime uses an advanced implementation of the Windows GraphicsDevice Interface (GDI) called GDI+.
- With GDI+ you can create graphics, draw text, and manipulate graphical images as objects. GDI+ is designed to offer performance and ease of use.
- You can use GDI+ to render graphical images on Windows Forms and controls.

- System.Drawing namespace provides access to GDI+ basic graphics functionality.
- More advanced functionality is provided in the System.Drawing.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text namespaces.

Before drawing any object (for example circle, or rectangle) we have to create a surface usingGraphics class. Generally we use Paint event of a Form to get the reference of the graphics.

Another way is to override **OnPaint** method.

```
private void form1_Paint(object sender, PaintEventArgs e)
        {
           Graphics g = e.Graphics;
        }
```

OR:

```
        protected override void OnPaint(PaintEventArgs e)
        {
           Graphics g = e.Graphics;
        }
```

MDI

- Stands for multiple document interface.
- MDIs start with a single container window that represents the entire application. Inside thecontainer window are multiple child windows.
- Depending on the type of application, these child windows might represent different documents the user is editing at the same time or different views of the same data.
- Visual Studio is an MDI application.
- Switch focus to specific document can be easily handled in MDI. For maximizing all documents parent window is maximized by MDI

SDI

- Stands (single document interface).
- SDIs can open only a single document at a time.
- Notepad is an example of an SDI application—if you want to open two text files at once, youneed to fire up two instances of Notepad.
- SDI is independent and thus is a standalone window.
- For grouping SDI uses special window managers

Differences between SDI and MDI

SDI	*MDI*
Stands for "Single Document Interface".	Stands for "Multiple Document Interface"
One document per window is enforced in SDI	Child windows per document are allowed in MDI.
SDI is not container control	MDI is a container control
SDI contains one window only at a time	MDI contain multiple document at a time appeared as child window

SDI supports one interface means you can handle only one application at a time.	MDI supports many interfaces means we can handle many applications at a time according to user's requirement.
SDI uses Task Manager for switching between documents	For switching between documents MDI uses special interface inside the parent window
SDI grouping is possible through special window managers.	MDI grouping is implemented naturally
In SDI it is implemented through special code or window manager.	For maximizing all documents, parent window is maximized by MDI
It is difficult to implement in SDI.	Switch focus to specific document can be easily handled

Dialogbox (Modal and Modeless)

A dialog box in C# is a type of window, which is used to enable common communication ordialog between a computer and its user.

Dialog boxes are of two types, which are given below.

1. Modal dialog box
2. Modeless dialog box

There are two classes, Modal and Modeless for each dialog box. Both are derived from the Form class.

Modal dialog box

A dialog box that temporarily halts the application and the user cannot continue until the dialog has been closed is called modal dialog box. The application may require some additional information before it can continue or may simply wish to confirm that the user wants to proceed with a potentially dangerous course of action. The application continues to execute only after thedialog box is closed; until then the application halts. For example, when saving a file, the user gives a name of an existing file; a warning is shown that a file with the same name exists, whether it should be overwritten or be saved with different name. The file will not be saved unless the user selects "OK" or "Cancel". Model dialog is displayed, using ShowDialog() method.

```
class Modal : Form
```

```
{
//Implementation here
}
```

Modeless dialog box

It is used when the requested information is not essential to continue, so the Window can be leftopen, while work continues somewhere else. For example, when working in a text editor, the user wants to find and replace a particular word.

This can be done, using a dialog box, which asks for the word to be found and replaced. The usercan continue to work, even if this box is open. Modeless dialog boxes are displayed, using Show() method.

```
class Modeless : Form
{
//Implementation here
}
```

Form Inheritance

Form inheritance, a new feature of .NET that lets you create a base form that becomes the basisfor creating more advanced forms. The new "derived" forms automatically inherit all the functionality contained in the base form. This design paradigm makes it easy to group common functionality and, in the process, reduce maintenance costs. When the base form is modified, the"derived" classes automatically follow suit and adopt the changes.

Visual inheritance allows you to see the controls on the base form and to add new controls

Inherit a form programmatically

1. In your class, add a reference to the namespace containing the form you wish to inheritfrom.
2. In the class definition, add a reference to the form to inherit from. The reference should include the namespace that contains the form, followed by a period, then the name of thebase form itself.

   ```
   public class Form2 : Namespace1.Form1
   ```

Inherit Forms Using the Inheritance Picker

The easiest way to inherit a form or other object is to use the **Inheritance Picker** dialog box. With it, you can take advantage of code or user interfaces (UI) you have already created in other solutions.

1. In Visual Studio, from the **Project** menu, choose **Add Windows Form**. The **Add New Item** dialog box opens.
2. Search the **Inherited Form** template by clicking on the **Windows Forms** category, selectit, and name it in the **Name** box. Click the **Add** button to proceed.
3. The Inheritance Picker dialog box opens. To inherit from a form in another assembly,click the **Browse** button.

4. Within the **Select a file which contains a component to inherit from** dialog box, Clickthe name of the .exe or .dll file to select it and click the **Open** button.

Developing Custom
Custom Controls are nothing but just graphics. It is used to improve performance and appearanceof your created application. It looks better than just simple controls in .NET or any language.

Custom control is a control that is not included in the . NET framework library and is insteadcreated by a third-party software vendor or a user. Custom control is a concept used while building both Windows Forms client and ASP.NET Web applications

Inherit from the Control class if:

- You want to provide a custom graphical representation of your control.
- You need to implement custom functionality that is not available through standardcontrols.
- To implement a custom control, you must write code for the OnPaint event of the control,as well as any feature-specific code you need.
- A different shaped buttons like Rectangular,RoundRectangular,Circular etc.

Composite and Extended Controls.

Composite controls are controls that combine multiple controls together to form a new reusable control. For example, a simple composite control could consist of both a Label control and a TextBox control.

Composite controls provide a means by which custom graphical interfaces can be created and reused. A composite control is essentially a component with a visual representation. As such, it might consist of one or more Windows Forms controls, components, or blocks of code that can extend functionality by validating user input, modifying display properties,. Composite controls can be placed on Windows Forms in the same manner as other controls.

Extended control

If you want to extend the functionality of an existing control, you can create a control derived from an existing control through inheritance. When inheriting from an existing control, you inherit all of the functionality and visual properties of that control. For example, if you were creating a control that inherited from Button, your new control would look and act exactly like a standard Button control. You could then extend or modify the functionality of your new control through the implementation of custom methods and properties. In some controls, you can also change the visual appearance of your inherited control by overriding its OnPaint method.