CBCS SCHEME

USN 1CR19MCA59 18MCA51

Fifth Semester MCA Degree Examination, Jan./Feb. 2021
Programming Using C#.NET

Time: 3 hrs. Max. Marks: 100

Note: ***Answer any FIVE full questions, choosing ONE full question from each module.***

Module-1

1 a. Explain the components of .NET framework with the help of architecture diagram. **(10 Marks)**

b. Explain the different data types in C#. **(10 Marks)**

OR

2 a. Write short notes on :
i) Windows work flow foundation ii) Windows card space. **(10 Marks)**

b. Bring out the difference between value types and reference types. Also explain boxing and unboxing with the help of an simple program. **(10 Marks)**

Module-2

3 a. Explain with example the method of implementing encapsulation using general method and using class properties. **(10 Marks)**

b. Distinguish between compile time polymorphism and runtime polymorphism with the help of example. **(10 Marks)**

OR

4 a. Explain the following with example :
i) Static class and static members ii) Abstract class. **(10 Marks)**

b. Demonstrate interface inheritance and implementation of interfaces, with the help of code example. **(10 Marks)**

Module-3

5 a. What are Delegates? Explain with the help of example the steps in creating and using delegates and also the concept of multicasting with delegates. **(10 Marks)**

b. Explain the architecture of ADO.NET with a neat diagram and also list the ADO.NET providers. **(10 Marks)**

OR

6 a. Write a C# program using try, catch and finally to explain integral type exception handling using checked and unchecked statements. **(10 Marks)**

b. Write and explain the steps to create a connection to database and run queries using SQL server database. **(10 Marks)**

Module-4

7 a. Explain WPF architecture. With neat diagram. **(10 Marks)**

b. Explain the following :
i) Event driven GUI ii) MDI windows. **(10 Marks)**

OR

8 a. List and explain the control class properties and methods. **(10 Marks)**

b. Explain the following :

i) XAML elements.

ii) Markup extension classes in XAML. **(10 Marks)**

Module-5

9 a. Explain in detail multitier application architecture. **(10 Marks)**

b. Explain different validation controls with suitable example supported by ASP.NET. **(10 Marks)**

OR

10 a. What are Cookies? Explain session management in ASP.NET using cookies. **(10 Marks)**

b. Explain the following AJAX server controls :

i) Script Manager Control.

ii) Update Panel Control **(10 Marks)**

* * * * *

1a. **Module 1**

Components of .NET Framework 4.0:

The .NET Framework provides all the necessary components to develop and run an application. The components of .NET Framework 4.0 architecture are as follows:

- Common Language Runtime (CLR)
- Common Type System (CTS)
- Metadata and Assemblies
- .NET Framework class library
- Windows Forms
- ASP.NET and ASP.NET AJAX
- ADO.NET
- Windows Workflow Foundation
- Windows Presentation Foundation
- Windows Communication Foundation
- Windows CardSpace
- LINQ

Let's now discuss about each of them in detail.

CLR[Common Language Runtime]:

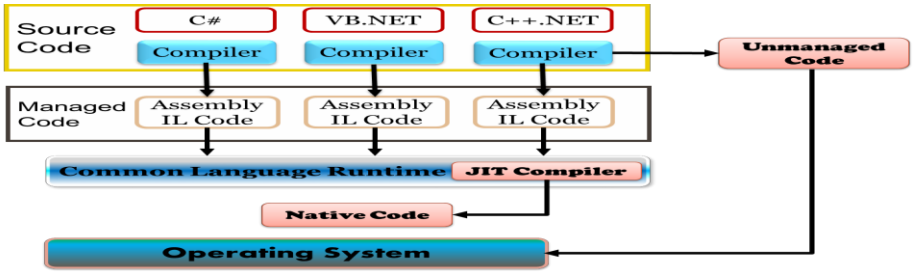
"CLR *is an* ***Execution Engine for .NET Framework applications*****".**

CLR is a heart of the.NET Framework. It provides a run-time environment to run the code and various services to develop the application easily.

The services provided by CLR are –

- Memory Management
- Exception Handling
- Debugging
- Security
- Thread execution
- Code execution
- Language Integration
- Code safety
- Verification
- Compilation

The following figure shows the **process** of compilation and execution of the code by the JIT Compiler:



i. After verifying, a **JIT** [*Just-In-Time*] compiler extracts the metadata from the file to translate that verified IL code into ***CPU-specific code*** or ***native code***. These type of IL Code is called as **managed code**.

ii. The source code which is directly compiles to the machine code and runs on the machine where it has been compiled such a code called as **unmanaged code**. It does not have any services of CLR.

iii. Automatic garbage collection, exception handling, and memory management are also the responsibility of the CLR.

Managed Code: Managed code is the code that is executed directly by the CLR. The application that are created using managed code automatically have CLR services, such as type checking, security, and automatic garbage collection.

The process of executing a piece of managed code is as follows:

- Selecting a language compiler
- Compiling the code to IL[This intermediate language is called managed code]
- Compiling IL to native code Executing the code

Unmanaged Code: Unmanaged Code directly compiles to the machine code and runs on the machine where it has been compiled. It does not have services, such as security or memory management, which are provided by the runtime. If your code is not security-prone, it can be directly interpreted by any user, which can prove harmful.

Automatic Memory Management: CLR calls various predefined functions of .NET framework to allocate and de-allocate memory of .NET objects. So that, developers need not to write code to explicitly allocate and de-allocate memory.

CTS [Common Type Specifications]:

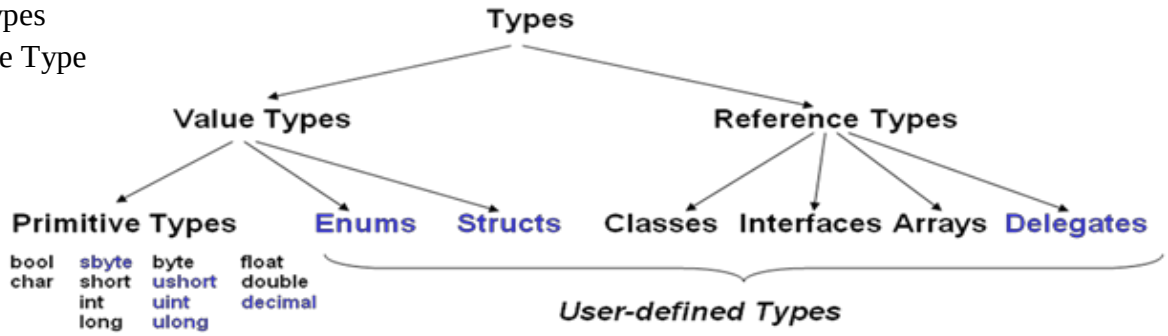
The CTS defines the rules for declaring, using, and managing types at runtime. It is an integral part of the runtime for supporting cross-language communication.

The common type system performs the following functions:

- Enables cross-language integration, type safety, and high-performance code execution.
- Provides an object-oriented model for implementation of many programming languages.
- Defines rules that every language must follow which runs under .NET framework like C#, VB.NET, F# etc. can interact with each other.

The CTS can be classified into two data types, are

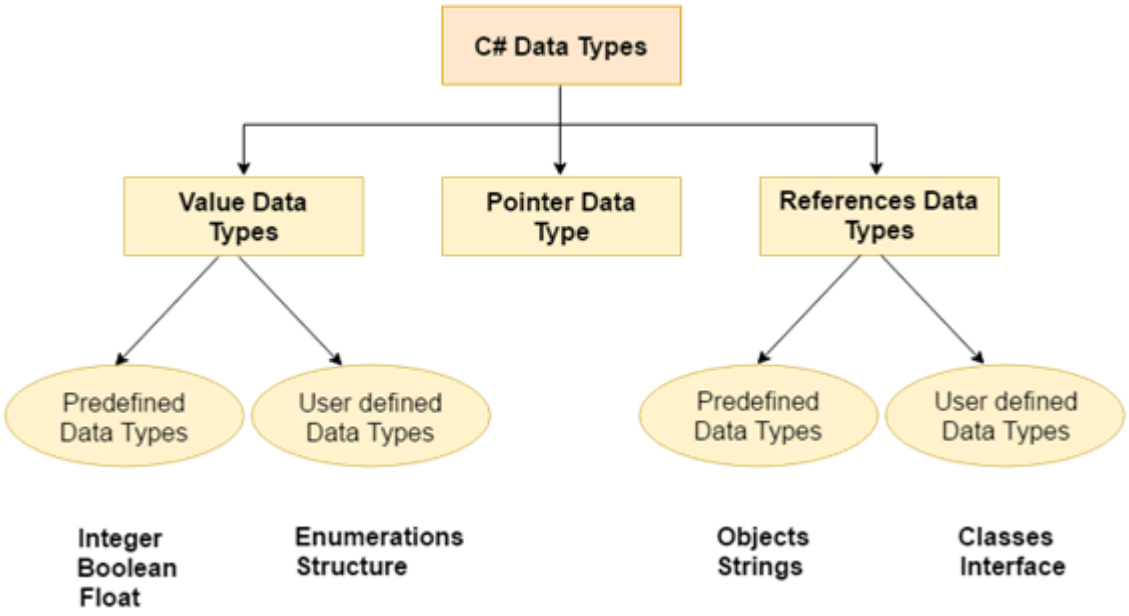
iv. Value Types
v. Reference Type



1b. **C# DataTypes**

Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface

Pointer Data Type	Pointers



Value Data Type

The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.

There are 2 types of value data type in C# language.

1) Predefined Data Types - such as Integer, Boolean, Float, etc.

2) User defined Data Types - such as Structure, Enumerations, etc.

The memory size of data types may change according to 32 or 64 bit operating system.

Let's see the value data types. It size is given according to 32 bit OS.

Data Types	Memory Size	Range

char	2 byte	-128 to 127
signed char	2 byte	-128 to 127
unsigned char	2 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,647
signed int	4 byte	-2,147,483,648 to -2,147,483,647
unsigned int	4 byte	0 to 4,294,967,295
long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	1.5 * 10-45 - 3.4 * 1038, 7-digit precision
double	8 byte	5.0 * 10-324 - 1.7 * 10308, 15-digit precision
decimal	16 byte	at least -7.9 * 10?28 - 7.9 * 1028, with at

		least 28-digit precision

Reference Data Type

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables. In other words they refer to the memory location.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

Object Type: The Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
Object ob1;
ob1 = 100; // This is boxing
```

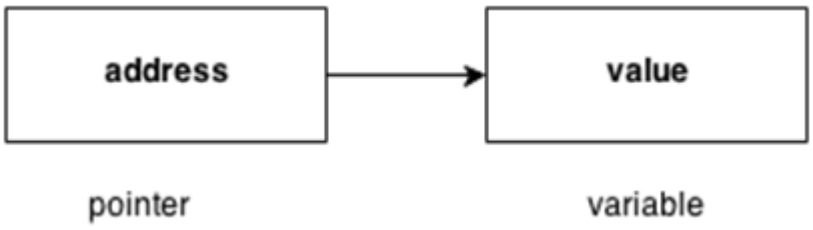
Dynamic Type : You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

```
dynamic variablename = value;
dynamic d = 10;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Example:

```
int *a;

char *b;
```

2a.

Windows Workflow Foundation (WF)

Windows Workflow Foundation is the programming model, engine, and tools for quickly building workflow-enabled applications on Windows. It consists of a namespace, an in-process workflow engine, and designers for Visual Studio 2005. Windows Workflow Foundation can be developed and run on Windows Vista, Windows XP, and the Windows Server 2003 family. Windows Workflow Foundation includes support for both system workflow and human workflow across a variety of scenarios, including workflow within line-of-business applications, user interface page-flow, document-centric workflow, human workflow, composite workflow for service-oriented applications, business rule-driven workflow, and workflow for systems management.

Windows Workflow Foundation provides a consistent and familiar development experience with other .NET Framework 3.0 technologies, such as Windows Communication Foundation and Windows Presentation Foundation. It provides full support for Visual Basic .NET and C#, debugging, a graphical workflow designer, and developing your workflow completely in code or in markup. Windows Workflow Foundation also provides an extensible model and designer to build custom activities that encapsulate workflow functionality for end users or for reuse across multiple projects.

Windows CardSpace

The Internet continues to be increasingly valuable, and yet also faces significant challenges. Online identity theft, fraud, and privacy concerns are rising. Users must track a growing number of accounts and passwords. This burden results in "password fatigue," and that results in insecure practices, such as reusing the same account names and passwords at many sites. Many of these problems are rooted in the lack of a widely adopted identity solution for the Internet.

CardSpace is Microsoft's implementation of an Identity Metasystem that enables users to choose from a portfolio of identities that belong to them and use them in contexts where they are accepted, independent of the underlying identity systems where the identities originate and are used.

It is the code name for an identity management component in Microsoft's upcoming WinFX product suite. With InfoCard, end users create a digital file on their own computer that includes user-specific information, like a work phone number, e-mail address or snail mail address. Users are able to save several different profiles and use whichever profile is appropriate for a certain situation. CardSpace is a potential successor to the Microsoft Passport information management system, which stores user information on a database instead of individual machines.

Customer security is becoming an increasingly important part of Web application development. One key to security, particularly in credit-card and other financial transactions, is safer customer identity management.

There are two types of CardSpace information cards -- personal cards, which users create themselves, and managed cards, which are issued by identity providers. The card itself does not actually contain any personal data. Rather, the card indicates which identity provider must be contacted to obtain the claims to this data. An application requests these claims by issuing a security token; once this happens, the entire transaction is locked down, and no code at all will run.

2b

1. Value types:
   Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations.

2. Reference types
   Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and delegates.

Differences between value type and Reference type:

- Value types are stored in **stack,** Reference types are stored in **heap**
- When we assign one value type to another value type, it is cloned and the two instances operate independently.
  For eg, **a=b**; A new memory location is allocated for **a** & it is hold the value individually. Changing the value of **b** does not affect **a.**

- When reference type is assigned to another reference type, the two references share the same instance and change made by the one instance affects the other.
  For eg, **a=b;** a reference (pointer) is created for **a** and both **a** and **b** now points to same address. Any alteration made to **b** will affect **a.**

- Value types cannot be set to null, Reference types can be set to null
- Converting value type to reference type is called **boxing,** converting reference type to value type to called **unboxing/undoing.**
- Value types are by default passed by value to other methods, Reference types are by default **passed by reference** to other methods.

The **stack** holds value type variables and **heap** hold reference type variables

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing.**

```
Object ob1;

ob1 = 100; // This is boxing

dynamic variablename = value;

dynamic d = 10;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

Boxing

```
using System;
class GFG {

    // Main Method
    static public void Main()
    {

        // assigned int value
        // 2020 to num
        int num = 2020;

        // boxing
        object obj = num;

        // value of num to be change
        num = 100;

        System.Console.WriteLine
        ("Value - type value of num is : {0}", num);
        System.Console.WriteLine
        ("Object - type value of obj is : {0}", obj);
    }
}
```

Unboxing

```
using System;
class GFG {

    // Main Method
    static public void Main()
    {
```

```
        // assigned int value
        // 23 to num
        int num = 23;

        // boxing
        object obj = num;

        // unboxing
        int i = (int)obj;

        // Display result
        Console.WriteLine("Value of ob object is : " + obj);
        Console.WriteLine("Value of i is : " + i);
    }
}
```

Module 2

3a.

Generics allow you to define the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

Generic Methods
we can declare a generic method with a type parameter.

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

namespace GenericMethodAppl {
   class Program {
      static void Swap<T>(ref T lhs, ref T rhs) {
         T temp;
         temp = lhs;
         lhs = rhs;
         rhs = temp;
      }
      static void Main(string[] args) {
         int a, b;
         char c, d;
         a = 10;
         b = 20;
         c = 'I';
         d = 'V';

         //display values before swap:
         Console.WriteLine("Int values before calling swap:");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values before calling swap:");
         Console.WriteLine("c = {0}, d = {1}", c, d);

         //call swap
         Swap<int>(ref a, ref b);
         Swap<char>(ref c, ref d);

         //display values after swap:
         Console.WriteLine("Int values after calling swap:");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values after calling swap:");
         Console.WriteLine("c = {0}, d = {1}", c, d);
```

```
         Console.ReadKey();
      }
   }
}
```

Class Properties

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

```
using System;
namespace tutorialspoint {
   class Student {
      private string code = "N.A";
      private string name = "not known";
      private int age = 0;

      // Declare a Code property of type string:
      public string Code {
         get {
            return code;
         }
         set {
            code = value;
         }
      }

      // Declare a Name property of type string:
      public string Name {
         get {
            return name;
         }
         set {
            name = value;
         }
      }

      // Declare a Age property of type int:
      public int Age {
         get {
            return age;
         }
         set {
            age = value;
         }
      }
      public override string ToString() {
         return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
      }
   }

   class ExampleDemo {
      public static void Main() {

         // Create a new Student object:
         Student s = new Student();

         // Setting code, name and the age of the student
         s.Code = "001";
         s.Name = "Zara";
         s.Age = 9;
         Console.WriteLine("Student Info: {0}", s);
```

```
         //let us increase age
         s.Age += 1;
         Console.WriteLine("Student Info: {0}", s);
         Console.ReadKey();
      }
   }
}
```

3b.

Polymorphism can be static or dynamic. In static polymorphism, the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.

The linking of a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are Function overloading and Operator overloading.

In function overloading you can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

```
using System;

namespace PolymorphismApplication {
   class Printdata {
      void print(int i) {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f) {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s) {
         Console.WriteLine("Printing string: {0}", s);
      }

      static void Main(string[] args) {
         Printdata p = new Printdata();

         // Call print to print integer
```

```
            p.print(5);

            // Call print to print float
            p.print(500.263);

            // Call print to print string
            p.print("Hello C#");
            Console.ReadKey();
        }
    }
}
```

Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

```
using System;
public class Shape{

   public virtual void draw(){
      Console.WriteLine("drawing...");
   }
}
public class Rectangle: Shape
{
   public override void draw()
   {
      Console.WriteLine("drawing rectangle...");
   }

}


public class Circle : Shape  {
   public override void draw()
   {
      Console.WriteLine("drawing circle...");
   }

}
public class TestPolymorphism
    {
       public static void Main()
       {
```

```
        Shape s;
        s = new Shape();
        s.draw();
        s = new Rectangle();
        s.draw();
       s = new Circle();
        s.draw();

    }
}
```

4a.

Static Class

The C# static class is like the normal class but it cannot be instantiated. It can have only static members. The advantage of static class is that it provides you guarantee that instance of static class cannot be created.

- C# static class contains only static members.
- C# static class cannot be instantiated.
- C# static class is sealed.
- C# static class cannot contain instance constructors.

```
using System;
  public static class MyMath
   {
      public static float PI=3.14f;
      public static int cube(int n){return n*n*n;}
   }
  class TestMyMath{
     public static void Main(string[] args)
      {
         Console.WriteLine("Value of PI is: "+MyMath.PI);
         Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));
      }
   }
```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it's explicitly passed in a method parameter.

It is more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field cannot be declared as `static const`, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, `const` fields can be accessed by using the same `ClassName.MemberName` notation that's used for static fields. No object instance is required.

C# does not support static local variables (that is, variables that are declared in method scope).

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

4.

C# Abstract class

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

```
using System;
public abstract class Shape
{
public abstract void draw();
}
public class Rectangle : Shape
{
public override void draw()
{
Console.WriteLine("drawing rectangle...");
}
}
public class Circle : Shape
{
```

```
public override void draw()
{
Console.WriteLine("drawing circle...");
}
}
public class TestAbstract
{
public static void Main()
{
Shape s;
s = new Rectangle();
s.draw();
s = new Circle();
s.draw();
}
}
```

4b. Interface Inheritance

C# allows the user to inherit one interface into another interface. When a class implements the inherited interface then it must provide the implementation of all the members that are defined within the interface inheritance chain.

Important Points:

- If a class implements an interface, then it is necessary to implement all the method that defined by that interface including the base interface methods. Otherwise, the compiler throws an error.
- If both derived interface and base interface declares the same member then the base interface member name is hidden by the derived interface member name.

```
// C# program to illustrate the concept

// of inheritance in interface

using System;



// declaring an interface

public interface A {



        // method of interface

        void mymethod1();

        void mymethod2();

}
```

```
// The methods of interface A
// is inherited into interface B
public interface B : A {

        // method of interface B
        void mymethod3();
}



// Below class is inherting
// only interface B
// This class must
// implement both interfaces
class Geeks : B
{

        // implementing the method
        // of interface A
        public void mymethod1()
        {
                Console.WriteLine("Implement method 1");
        }

        // Implement the method
        // of interface A
        public void mymethod2()
        {
                Console.WriteLine("Implement method 2");
        }
```

```
        // Implement the method
        // of interface B
        public void mymethod3()
        {
                Console.WriteLine("Implement method 3");
        }
}

// Driver Class
class GFG {

        // Main method
        static void Main(String []args)
        {

                // creating the object
                // class of Geeks
                Geeks obj = new Geeks();

                // calling the method
                // using object 'obj'
                obj.mymethod1();
                obj.mymethod2();
                obj.mymethod3();
        }
}
```

5a.

Delegates

A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods. Delegates in C# are similar to the function pointer in C/C++. It provides a way which tells which method is to be called when an event is triggered.

For example, if you click an Button on a form (Windows Form application), the program would call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

A Delegate can be defined as a delegate type. Its definition must be similar to the function signature. A delegate can be defined in a namespace and within a class.
A delegate cannot be used as a data member of a class or local variable within a method.

Delegate declarations look almost exactly like abstract method declarations, you just replace the abstract keyword with the delegate keyword.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the System. Delegate class.

In C#, delegate is a *reference to the method*. It works like *function pointer* in C and C++. But it is objected-oriented, secured and type-safe than function pointer.

For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both.

The best use of delegate is to use as event.

Internally a delegate declaration defines a class which is the derived class of **System.Delegate**.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

delegate <return type> <delegate-name> <parameter list>

Instantiating Delegates

Once a delegate type is declared, a delegate object must be created with the new keyword and be associated with a particular method. When creating a delegate, the argument passed to the new expression is written similar to a method call, but without the arguments to the method

Delegate Program 1

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace delegatesex
{
    // Declaring the delegate named delefunc which returns an integer
    public delegate int delefunc(int x, int y);

    class Program
    {
      // function add is passed as argument to the delegate delefunc
        static int add(int a, int b)
        {
            return a + b;
        }

        static void Main(string[] args)
        {
            // create an instance of delegate
            delefunc d1 = new delefunc(add);

            // pass the values and print output
            Console.WriteLine("Addition of numbers = {0}", d1(20, 30));
            Console.ReadKey();
        }
    }

}
```

Delegates can be of two categories mainly

1. Singlecast Delegate
2. Multicast Delegates
3. Generic Delegates

Singlecast Delegate

This is a kind of delegate that can refer to single method at one time. SingleCast Delegates refer to a single method with matching signature. SingleCast Delegates derive from the System.Delegate class.

Single cast delegate program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace delegatefunction
{
    // Delegate definition
    public delegate int delefunc(int x, int y);

    class Program
    {
        static int add(int a, int b)
        {
            return a + b;
```

```
        }
        public static void Main(string[] s)
        {
            // instantiate the delegate
            // delegatename obj = new delegatename(classname.methodname)
            delefunc d1 = new delefunc(Program.add);
            // pass the values and print output

            Console.WriteLine("Addition of numbers = {0}", d1(20, 30));
            Console.ReadKey();
        }
    }
}
```

Multicast Delegate

A delegate that holds a reference to more than one method is called multicasting delegate. A **Multicast Delegate** is a delegate that holds the references of more than one function. When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked. If you want to call multiple methods using a delegate then all the method signature should be the same.

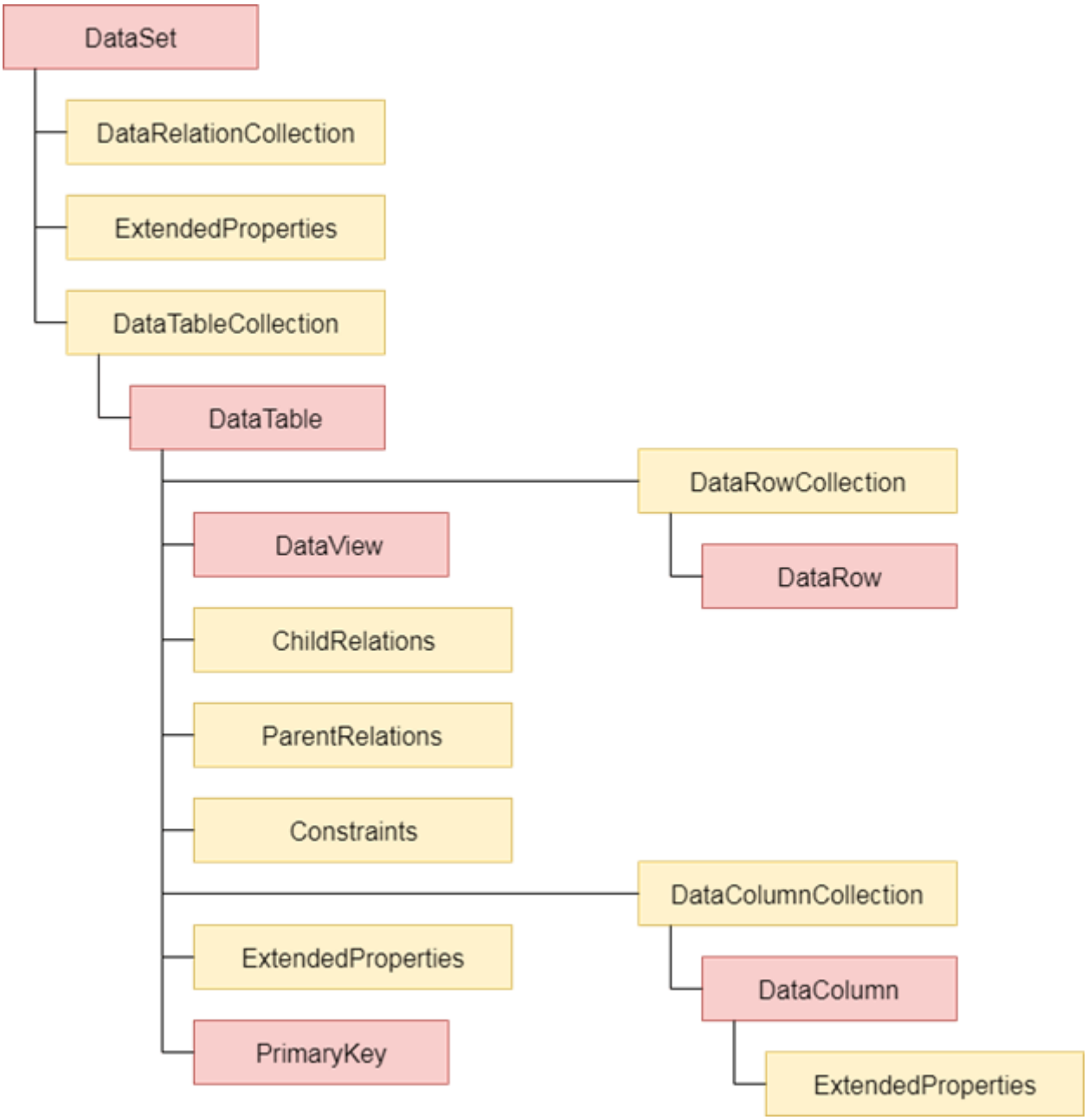
Multicast Delegate Program 1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace multicastedelexamole
{
    // declaring the delegate
    public delegate void MyDel(int num1, int num2);
    class Sample
    {
        // Method Add is the first method called by the delegate MyDel
         static void Add(int num1, int num2)
        {
            Console.WriteLine("\tAddition: " + (num1 + num2));
        }
         // Method Sub is the second method called by the delegate MyDel
        static void Sub(int num1, int num2)
        {
            Console.WriteLine("\tSubtraction: " + (num1 - num2));
        }
         // Method Mul is the third method called by the delegate MyDel
        static void Mul(int num1, int num2)
        {
            Console.WriteLine("\tMultiplication: " + (num1 * num2));
        }

        static void Main()
        {
            int num1 = 0;
            int num2 = 0;
```

```
            // instantiating the delegate with first method as parameter
            MyDel del = new MyDel(Add);
            // input the values to be passed as arguments
            Console.Write("Enter the value of num1: ");
            num1 = int.Parse(Console.ReadLine());

            Console.Write("Enter the value of num2: ");
            num2 = int.Parse(Console.ReadLine());
            // second method is appended to the delegate object
            del += new MyDel(Sub);
            // third method is appended to the delegate object
            del += new MyDel(Mul);

            Console.WriteLine("Call 1:");
// the methods will be executed one after the other and output will be displayed.
            del(num1, num2);

            Console.ReadKey();
        }
    }
}
```

5b.

ADO.NET uses a multilayer architecture that has components such as the *Connection, Reader, Command, Adapter* and *DataSet* objects. ADO.NET introduced *data providers* that are a set of special classes to access a specific database, execute SQL commands and retrieve data. Data providers are extensible; developers can create their own providers for proprietary data source Some examples of data providers include SQL server providers, OLE DB and Oracle providers.



The ADO.NET Framework comes with the following providers:

- **OLEDB:** The OLEDB provider, expressed through the *System.Data.OleDb* namespace. You can use this provider to access SQL Server 6.5 and earlier, SyBase, DB2/400, and Microsoft Access.
- **ODBC:** The ODBC provider, expressed through the *System.Data.Odbc* namespace. This provider is typically used when no newer provider is available.
- **SQL Server:** The Microsoft SQL Server provider, expressed through the *System.Data.SqlClient* namespace. It Contains classes that provide functionality similar to the generic *OleDb* provider. The difference is that these classes are tuned for SQL Server 7 and later data access.

6a.

```
using System;
namespace CSharpProgram
{
   class Program
   {
      static void Main(string[] args)
      {
         checked
         {
            int val = int.MaxValue;
            Console.WriteLine(val + 2);
         }
      }
   }
}
```

```
using System;

namespace CSharpProgram

{

class Program

{

static void Main(string[] args)

{

    unchecked

    {

    int val = int.MaxValue;

    Console.WriteLine(val + 2);

    }

}

}

}
```

6b.

Simple Steps:

//To establish a connection:

Using System.Data.OLEDB;

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();
MessageBox.Show("Database connected");
con.Close();
```

//To insert values into DB:

Using System.Data.OLEDB;

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = con;
cmd.CommandText = "insert into t1 values(" + textBox1.Text + ",'" +
textBox2.Text + "')";
cmd.ExecuteNonQuery();
MessageBox.Show("Value Inserted");
con.Close();
```

//To Retrieve all values and place in datagrid;
//using Datagrid/ Data Adapter/ Datatable

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/Samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = con;
cmd.CommandText = "select *from t1";
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
DataTable dt = new DataTable();
da.Fill(dt);
dataGridView1.DataSource = dt;
con.Close();
```

//To Retrieve values and place in datagrid for a particular condition;

```
private void button1_Click(object sender, EventArgs e)
{
    string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
    Source=C:/Users/MCA/Desktop/Samp1.accdb";
    OleDbConnection con = new OleDbConnection(cs);
    con.Open();
    OleDbCommand cmd = new OleDbCommand();
    cmd.Connection = con;
    cmd.CommandText = "select *from t1 where ID=" + textBox1.Text;
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataTable dt = new DataTable();
    da.Fill(dt);
    dataGridView1.DataSource = dt;
    con.Close();
```

```
}
```

// How to populate ListBox using Dataset, Datatable, Data Adapter

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = con;
cmd.CommandText = "select ID from t1";
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
DataSet ds = new DataSet();
da.Fill(ds);
DataTable dt = new DataTable();
dt = ds.Tables[0];
listBox1.DataSource = dt;
listBox1.DisplayMember = "ID";
con.Close();
```

// How to populate Text using Data Reader

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "select *from t1 where ID= 2";
cmd.Connection = con;
OleDbDataReader dr = cmd.ExecuteReader();
while (dr.Read())
{
    textBox1.Text = dr.GetValue(0).ToString();
    textBox2.Text = dr.GetValue(1).ToString();

}
con.Close();
```

// Adding multiple table to a dataset

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
OleDbConnection con = new OleDbConnection(cs);
con.Open();

OleDbDataAdapter d1 = new OleDbDataAdapter("select *from t1",con);
DataSet ds = new DataSet();
d1.Fill(ds,"[t1]");
dataGridView1.DataSource = ds.Tables["[t1]"];
```

```
OleDbDataAdapter d2 = new OleDbDataAdapter("select *from t2",con);
d2.Fill(ds,"[t2]");
dataGridView2.DataSource = ds.Tables["[t2]"];
  con.Close();
```

// DataView

```
string cs = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:/Users/MCA/Desktop/samp1.accdb";
 OleDbConnection con = new OleDbConnection(cs);
 con.Open();
 OleDbDataAdapter d1 = new OleDbDataAdapter("select *from t2", con);
 DataSet ds = new DataSet();
 d1.Fill(ds, "[t2]");
 DataView dv = new DataView(ds.Tables[0]);
 dv.Sort = " phone desc";
 dataGridView1.DataSource = dv;
 con.Close();
```

Module 4

7a.

WPF stands for Windows Presentation Foundation. It is a powerful framework for building Windows applications. This tutorial explains the features that you need to understand to build WPF applications and how it brings a fundamental change in Windows applications.

WPF was first introduces in .NET framework 3.0 version, and then so many other features were added in the subsequent .NET framework versions.

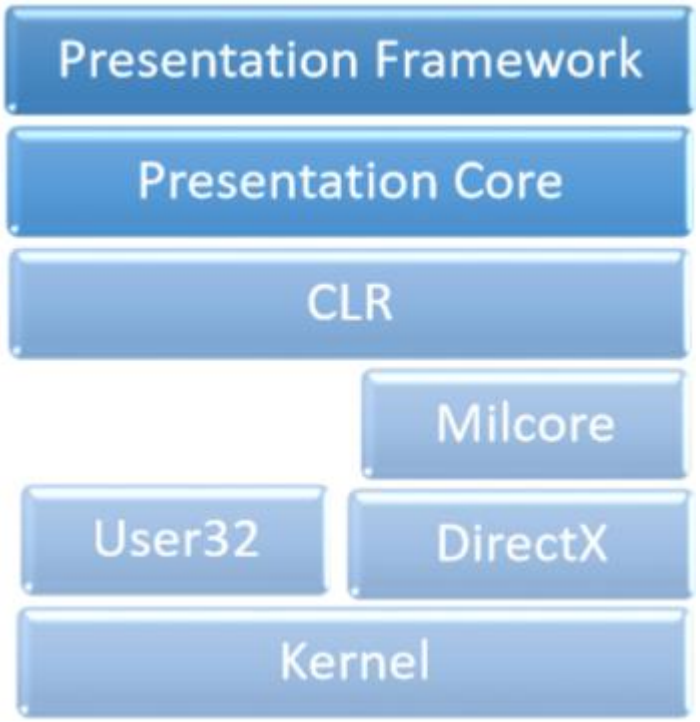
WPF Architecture

Before WPF, the other user interface frameworks offered by Microsoft such as MFC and Windows forms, were just wrappers around User32 and GDI32 DLLs, but WPF makes only minimal use of User32. So,

- WPF is more than just a wrapper.
- It is a part of the .NET framework.
- It contains a mixture of managed and unmanaged code.

The major components of WPF architecture are as shown in the figure below. The most important code part of WPF are −

- Presentation Framework
- Presentation Core
- Milcore



The **presentation framework** and the **presentation core** have been written in managed code. **Milcore** is a part of unmanaged code which allows tight integration with DirectX (responsible for display and rendering). **CLR** makes the development process more productive by offering many features such as memory management, error handling, etc.

7b.

Event Driven GUI

Event-driven programming is the dominant paradigm used in **graphical** user interfaces and other applications (e.g., JavaScript web applications) that are centered on performing certain actions in response to user input. Event-driven programming: A style of coding where a program's overall flow of execution is dictated by events. • The program loads, then waits for user input events. • As each event occurs, the program runs particular code to respond. • The overall flow of what code is executed is determined by the series of events that occur • Contrast with application- or algorithm-driven control where program expects input data in a pre-determined order and timing – Typical of large non-GUI applications like web crawling, payroll, batch simulation

• event: An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.

• listener: An object that waits for events and responds to them. – To handle an event, attach a listener to a component. – The listener will be notified when the event occurs (e.g. button click).

Mouse move/drag/click, mouse button press/release

• Keyboard: key press/release, sometimes with modifiers like shift/control/alt/meta/cokebottle

• Touchscreen finger tap/drag

• Joystick, drawing tablet, other device inputs

• Window resize/minimize/restore/close

• Network activity or file I/O (start, done, error)

• Timer interrupt (including animations)

MDI windows

A multiple-document interface (MDI) is a graphical user interface in which multiple windows reside under a single parent window. Such systems often allow child windows to embed other windows inside them as well, creating complex nested hierarchies. This contrasts with single-document interfaces (SDI) where all windows are independent of each other. MDI can be confusing if it has a lack of information about the currently opened windows. In MDI applications, the application developer must provide a way to switch between documents or view a list of open windows, and the user might have to use an application-specific menu ("window list" or something similar) to switch between open documents. This is different from SDI applications where the window manager's task bar or task manager displays the currently opened windows. In recent years it has become increasingly common for MDI applications to use "tabs" to display the currently opened windows. An interface in which tabs are used to manage open documents is referred to as a "tabbed document interface" (TDI). Another option is "tiled" panes or windows, which make it easier to prevent content from overlapping.

8b.

XAML is a new descriptive programming language developed by Microsoft to write user interfaces for next-generation managed applications. XAML is the language to build user interfaces for Windows and Mobile applications that use Windows Presentation Foundation (WPF), UWP, and Xamarin Forms.

The purpose of XAML is simple, to create user interfaces using a markup language that looks like XML. Most of the time, you will be using a designer to create your XAML but you're free to directly manipulate XAML by hand.

XAML uses the XML format for elements and attributes. Each element in XAML represents an object which is an instance of a type. The scope of a type (class, enumeration etc.) is defined as a namespace that physically resides in an assembly (DLL) of the .NET Framework library.

Similar to XML, a XAML element syntax always starts with an open angle bracket (<) and ends with a close angle bracket (>). Each element tag also has a start tag and an end tag. For example, a Button object is represented by the <Button> object element. The following code snippet represents a Button object element.

```
<Button></Button>
```

Alternatively, you can use a self-closing format to close the bracket.

```
<Button />
```

An object element in XAML represents a type. A type can be a control, a class or other objects defined in the framework library.

The Root Elements

Each XAML document must have a root element. The root element usually works as a container and defines the namespaces and basic properties of the element. Three most common root elements are <Windows />, <Page />, and <UserControl >. The <ResourceDirectory /> and <Application /> are other two root elements that can be used in a XAML file. The Window element represents a Window container. The following code snippet shows a Window element with its Height, Width, Title and x:Name attributes. The x:Name attribute of an element represents the ID of an element used to

access the element in the code-behind. The code snippet also sets xmlns and xmlns:x attributes that represent the namespaces used in the code. The x:Class attribute represents the code-behind class name.

```
<Window x:Class="HelloXAML.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
</Window>
```

The Page element represents a page container. The following code snippet creates a page container. The code also sets the FlowDirection attribute that represents the flow direct of the contents of the page.

```
<Page
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="WPFApp.Page1"
x:Name="Page"
WindowTitle="Page"
FlowDirection="LeftToRight"
Width="640" Height="480"
WindowWidth="640" WindowHeight="480">
</Page>
```

XAML Namespaces

The part of the root element of each XAML are two or more attributes pre-fixed with xmlns and xmlns:x.

The xmlns attribute indicates the default XAML namespace so the object elements in used in XAML can be specified without a prefix. Additionally, the x: prefix is used with more than just the namespace. Here are some common x:prefix syntaxes that are used in XAML.

1. x:Key: Sets a unique key for each resource in a ResourceDictionary.
2. x:Class: Class name provides code-behind for a XAML page.
3. x:Name: Unique run-time object name for the instance that exists in run-time code after an object element is processed.
4. x:Static: Enables a reference that returns a static value that is not otherwise a XAML-compatible property.
5. x:Type: Constructs a Type reference based on a type name.

Elements and Attributes

A type in WPF or Windows RT is represented by an XAML element. The <Page> and <Button> elements represent a page and a button control respectively. The XAML Button element listed in the following code represents a button control.

<Button />

Each of the elements such as <Page> or <Button> have attributes that can be set within the element itself. An attribute of an element represents a property of the type. For example, a Button has Height, Width, Background and Foreground

properties that represent the height, width, foreground color and background color of the button respectively. The Content property of the Button represents the text of a button control. The x:Name property represents the unique ID of a control that may be used to access a control in the code behind.

Content Property

Each XAML object element is capable of displaying different content types. XAML provides a special property called Content that works to display the content of the element depending on the element capabilities. For example, a Content property of a Button can be a set to a string, an object, a UIElement, or even and container. However, the Content property of a ListBox is set using the Items property.

Note: Some XAML object elements may not have the Content property available directly. It must be set through a property.

```
1. <Button Height="50" Margin="10,10,350,310" Content="Hello XAML" />
2. Here is an alternative way to set the Content property of a Button.
3. <Button Height="50" Margin="10,10,350,310">Hello XAML</Button>
```

ii)

In XAML applications, markup extensions are a method/technique to gain a value that is neither a specific XAML object nor a primitive type. Markup extensions can be defined by opening and closing curly braces and inside that curly braces, the scope of the markup extension is defined.

Data binding and static resources are markup extensions. There are some predefined XAML markup extensions in **System.xaml** which can be used.

```
<Window x:Class = "XAMLMarkupExtension.MainWindow"
   xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:my = "clr-namespace:XAMLMarkupExtension"
   Title = "MainWindow" Height = "350" Width = "525">

   <Grid>
      <Button Content = "{my:MyMarkupExtension FirstStr = Markup, SecondStr = Extension}"
         Width = "200" Height = "20" />
   </Grid>

</Window>
```

In the above XAML code, a button is created with some properties and for the content value, a custom markup extension **(my:MyMarkupExtension)** has been used with two values “Markup” and “Extension” which are assigned to FirstStr and SecondStr respectively.

Actually, `MyMarkupExtension` is a class which is derived from `MarkupExtension` as shown below in the C# implementation. This class contains two string variables, FirstStr and SecondStr, which are concatenated and return that string from the ProvideValue method to the Content of a button.

Module 5

9a. A **multi-tier application** is any **application** developed and distributed among more than one **layer**. It logically separates the different **application**-specific, operational layers. The number of layers varies by business and **application** requirements, but three-**tier** is the most commonly used **architecture.** Any application that depends on or uses a middleware application is known as a multi-tier application. A multi-tier application is also known as a multitiered application or n-tier application. A multi-tier application is used to divide an enterprise application into two or more components that may be separately developed and executed. In general, the tiers in a multi-tier application include the

following: **Presentation tier: Provides basic user interface and application access services Application processing tier: Possesses the core business or application logic Data access tier: Provides the mechanism used to access and process data Data tier: Holds and manages data that is at rest This division allows each component/tier to be separately developed, tested, executed and reused.**

9b. ASP.NET validation controls validate the user input data to ensure that useless, unauthenticated, or contradictory data don't get stored.

ASP.NET provides the following validation controls:

- RequiredFieldValidator
- RangeValidator
- CompareValidator
- RegularExpressionValidator
- CustomValidator
- ValidationSummary

BaseValidator Class

The validation control classes are inherited from the BaseValidator class hence they inherit its properties and methods. Therefore, it would help to take a look at the properties and the methods of this base class, which are common for all the validation controls:

Members	Description
ControlToValidate	Indicates the input control to validate.
Display	Indicates how the error message is shown.
EnableClientScript	Indicates whether client side validation will take.
Enabled	Enables or disables the validator.
ErrorMessage	Indicates error string.
Text	Error text to be shown if validation fails.
IsValid	Indicates whether the value of the control is valid.
SetFocusOnError	It indicates whether in case of an invalid control, the focus should switch to the related input control.
ValidationGroup	The logical group of multiple validators, where this control belongs.
Validate()	This method revalidates the control and updates the IsValid property.

RequiredFieldValidator Control

The RequiredFieldValidator control ensures that the required field is not empty. It is generally tied to a text box to force input into the text box.

The syntax of the control is as given:

```
<asp:RequiredFieldValidator ID="rfvcandidate"
   runat="server" ControlToValidate ="ddlcandidate"
   ErrorMessage="Please choose a candidate"
   InitialValue="Please choose a candidate">

</asp:RequiredFieldValidator>
```

RangeValidator Control

The RangeValidator control verifies that the input value falls within a predetermined range.

It has three specific properties:

Properties	Description
Type	It defines the type of the data. The available values are: Currency, Date, Double, Integer, ar String.
MinimumValue	It specifies the minimum value of the range.
MaximumValue	It specifies the maximum value of the range.

The syntax of the control is as given:

```
<asp:RangeValidator ID="rvclass" runat="server" ControlToValidate="txtclass"
   ErrorMessage="Enter your class (6 - 12)" MaximumValue="12"
   MinimumValue="6" Type="Integer">

</asp:RangeValidator>
```

CompareValidator Control

The CompareValidator control compares a value in one control with a fixed value or a value in another control.

It has the following specific properties:

Properties	Description
Type	It specifies the data type.
ControlToCompare	It specifies the value of the input control to compare with.
ValueToCompare	It specifies the constant value to compare with.
Operator	It specifies the comparison operator, the available values are: Equal, NotEqual, GreaterTha GreaterThanEqual, LessThan, LessThanEqual, and DataTypeCheck.

The basic syntax of the control is as follows:

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
   ErrorMessage="CompareValidator">

</asp:CompareValidator>
```

RegularExpressionValidator

The RegularExpressionValidator allows validating the input text by matching against a pattern of a regular expression. The regular expression is set in the ValidationExpression property.

10a.

ASP.NET Cookie is a small bit of text that is used to store user-specific information. This information can be read by the web application whenever user visits the site.

When a user requests for a web page, web server sends not just a page, but also a cookie containing the date and time. This cookie stores in a folder on the user's hard disk.

When the user requests for the web page again, browser looks on the hard drive for the cookie associated with the web page. Browser stores separate cookie for each different sites user visited.

There are two ways to store cookies in ASP.NET application.

- Cookies collection
- HttpCookie

We can add Cookie either to Cookies collection or by creating instance of HttpCookie class. both work same except that HttpCookie require Cookie name as part of the constructor.

With the **Session** object, you can create only an in-memory cookie. For the **Session** object to work correctly, you need to determine when a user's visit to the site begins and ends. IIS does this by using a cookie that stores an ASP Session ID, which is used to maintain a set of information about a user. If an ASP Session ID is not present, the server considers the current request to be the start of a visit. The visit ends when there have been no user requests for ASP files for the default time period of 20 minutes.

In this lesson, you will create the following:

- **Global.asa**: Global.asa is a file that allows you to perform generic actions at the beginning of the application and at the beginning of each user's session. An application starts the first time the first user ever requests a page and ends when the application is unloaded or when the server is taken offline. A unique session starts once for each user and ends 20 minutes after that user has requested their last page. Generic actions you can perform in Global.asa include setting application or session variables, authenticating a user, logging the date and time that a user connected, instantiating COM objects that remain active for an entire application or session, and so forth.
- **VisitCount.asp**: This ASP script uses the **Session** object to create an in-memory cookie.

When an application or session begins or ends, it is considered an event. Using the Global.asa file, you can use the predefined event procedures that run in response to the event.

10b.

Script manager control

The ScriptManager Control:

The ScriptManager control is the most important control and must be

present on the page for other controls to work.

Syntax:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
</asp:ScriptManager>
```

If you create an 'Ajax Enabled site' or add an 'AJAX Web Form' from the 'Add Item' dialog box, the web form automatically contains the script manager control.

The **ScriptManager** control takes care of the client-side script for all the server side controls.

The UpdatePanel Control:

The UpdatePanel control is a container control and derives from the Control class. It acts as a container for the child controls within it and does not have its own interface.

When a control inside it triggers a post back, the UpdatePanel interferes to initiate the post asynchronously and update just that portion of the page.

Example:

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
                <asp:Button ID="btnpartial" runat="server"
                        onclick="btnpartial_Click" Text="Partial PostBack"/>
                <br />
                <br />
                <asp:Label ID="lblpartial" runat="server"></asp:Label>
         </ContentTemplate>
</asp:UpdatePanel>
```

Properties of the UpdatePanel Control

The following table shows the properties of the update panel control:

Properties	Description
ChildrenAsTriggers	This property indicates whether the post backs are coming from the child controls, which cause the update panel to refresh.
ContentTemplate	It is the content template and defines what appears in the update panel when it is rendered.
UpdateMode	Gets or sets the rendering mode by determining some conditions.
Triggers	Defines the collection trigger objects each corresponding to an event causing the panel to refresh automatically.

Methods of the UpdatePanel Control

The following table shows the methods of the update panel control:

Methods	Description
CreateContentTemplateContainer	Creates a Control object that acts as a container for child controls that define the UpdatePanel control's content.
CreateControlCollection	Returns the collection of all controls that are contained in the UpdatePanel control.
Initialize	Initializes the UpdatePanel control trigger collection if partial-page rendering is enabled.
Update	Causes an update of the content of an UpdatePanel control.

The behavior of the update panel depends upon the values of the UpdateMode property and ChildrenAsTriggers property.

UpdateMode	ChildrenAsTriggers	Effect
Conditional	False	UpdatePanel refreshes if whole page refreshes or a triggering control outside it initiates a refresh.
Conditional	True	UpdatePanel refreshes if whole page refreshes or a child control on it posts back or a triggering control outside it initiates a refresh.