.NET FRAMEWORK AND C# (MCA-137)

UNIT 2

C# Inheritance

C# Inheritance

- In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

- In C#, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C# Inheritance

- Code reusability:

  Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C# Single Level Inheritance Example: Inheriting Fields

- When one class inherits another class, it is known as single level inheritance.

```
using System;
  public class Employee  {
      public float salary = 40000;
}
  public class Programmer: Employee  {
      public float bonus = 10000;
}
  class TestInheritance{
      public static void Main(string[] args)  {
          Programmer p1 = new Programmer();
            Console.WriteLine("Salary: " + p1.salary);
          Console.WriteLine("Bonus: " + p1.bonus);
       }
  }
```

Output:

Salary: 40000
Bonus: 10000

C# Single Level Inheritance Example: Inheriting Methods

```
using System;
  public class Animal
   {
      public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {
      public void bark() { Console.WriteLine("Barking..."); }
  }
  class TestInheritance2{
      public static void Main(string[] args)
       {
          Dog d1 = new Dog();
          d1.eat();
          d1.bark();
       }
  }
```

Output:

```
Eating...
Barking...
```

C# Multi Level Inheritance

- When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#.

- Inheritance is transitive so the last derived class acquires all the members of all its base classes.

C# Multi Level Inheritance Example

```
using System;
  public class Animal   {
     public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal  {
     public void bark() { Console.WriteLine("Barking..."); }
  }
  public class BabyDog : Dog  {
     public void weep() { Console.WriteLine("Weeping..."); }
  }
  class TestInheritance2{
     public static void Main(string[] args)  {
         BabyDog d1 = new BabyDog();
         d1.eat();
         d1.bark();
         d1.weep();
      }
  }
```

Output:

Eating...
Barking...
Weeping...

C# Multiple Inheritance

- C# does not support multiple inheritance.
- However, you can use interfaces to implement multiple inheritance.

C# Multiple Inheritance

Output:
Total area: 35
Total paint cost: $2450

```
using System;
namespace InheritanceApplication {
  class Shape {
    public void setWidth(int w) {
      width = w;     }
    public void setHeight(int h) {
      height = h;     }
    protected int width;
    protected int height;   }
  // Base class PaintCost
  public interface PaintCost {
    int getCost(int area);   }
  // Derived class
  class Rectangle : Shape, PaintCost {
    public int getArea() {
      return (width * height);     }
    public int getCost(int area) {
      return area * 70;   }   }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle Rect = new Rectangle();
      int area;
       Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();
      // Print the area of the object.
      Console.WriteLine("Total area: {0}",
Rect.getArea());
      Console.WriteLine("Total paint cost:
${0}" , Rect.getCost(area));
      Console.ReadKey();
    }   } }
```

C# Polymorphism

C# Polymorphism

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.
- There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

C# Runtime Polymorphism Example

```
using System;
public class Animal{
    public virtual void eat(){
        Console.WriteLine("eating...");
    }
}
public class Dog: Animal  {
    public override void eat()  {
        Console.WriteLine("eating bread...");
    }
}
public class TestPolymorphism  {
    public static void Main()     {
        Animal a= new Dog();
        a.eat();
    }
}
```

Output:

```
eating bread...
```

C# Runtime Polymorphism Example 2

```
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();
   }
}
```

Output:

drawing...
drawing rectangle...
drawing circle...

Runtime Polymorphism with Data Members

- Runtime Polymorphism can't be achieved by data members in C#.

```
using System;
public class Animal{
   public string color = "white";
 }
public class Dog: Animal  {
   public string color = "black";
}
public class TestSealed  {
   public static void Main()
   {
      Animal d = new Dog();
      Console.WriteLine(d.color);

   }
}
```

Output:

white

C# Member Overloading

- If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:
  - methods,
  - constructors, and
  - indexed properties

- It is because these members have parameters only.

C# Member Overloading Example: By changing data type of arguments

```
using System;
public class Cal{
    public static int add(int a, int b){
        return a + b;
    }
    public static float add(float a, float b)
    {
        return a + b;
    }
}
```

```
public class TestMemberOverloading
{
    public static void Main()
    {
        Console.WriteLine(Cal.add(12, 23));
        Console.WriteLine(Cal.add(12.4f,21.
3f));
    }
}
```

Output:

35
33.7

C# Method Overloading

- Having two or more methods with same name but different in parameters, is known as method overloading in C#.
- The advantage of method overloading is that it increases the readability of the program because you don't need to use different names for same action.
- You can perform method overloading in C# by two ways:
  - By changing number of arguments
  - By changing data type of the arguments

C# Method Overloading Example: By changing no. of arguments

```
using System;
public class Cal{
    public static int add(int a,int b){
        return a + b;
    }
    public static int add(int a, int b, int c)  {
        return a + b + c;
    }
}
public class TestMemberOverloading  {
    public static void Main()      {
        Console.WriteLine(Cal.add(12, 23));
        Console.WriteLine(Cal.add(12, 23, 25));
    }
}
```

Output:

35
60

C# Method Overloading

- Having two or more methods with same name but different in parameters, is known as method overloading in C#.
- The advantage of method overloading is that it increases the readability of the program because you don't need to use different names for same action.
- You can perform method overloading in C# by two ways:
  - By changing number of arguments
  - By changing data type of the arguments

C# Method Overriding

- If derived class defines same method as defined in its base class, it is known as method overriding in C#. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

- To perform method overriding in C#, you need to use virtual keyword with base class method and override keyword with derived class method.

C# Method Overriding Example

- Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

```
using System;
public class Animal{
   public virtual void eat(){
      Console.WriteLine("Eating...");
   }
}
public class Dog: Animal
{
   public override void eat()
   {
      Console.WriteLine("Eating bread...");
   }
}
```

```
public class TestOverriding
{
   public static void Main()
   {
      Dog d = new Dog();
      d.eat();
   }
}
```

Output:

Eating bread...

C# Operator Overloading

Operator Overloading

- The concept of overloading a function can also be applied to operators.
- Operator overloading gives the ability to use the same operator to do various operations.
- It provides additional capabilities to C# operators when they are applied to user-defined data types.
- It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.

Operator Overloading

- Only the predefined set of C# operators can be overloaded.
- To make operations on a user-defined data type is not as simple as the operations on a built-in data type.
- To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement.
- An operator can be overloaded by defining a function to it. The function of the operator is declared by using the operator keyword.

Operator Overloading

- Syntax :

```
access specifier  className  operator Operator_symbol (parameters)
{
   // Code
}
```

Note : Operator overloading is basically the mechanism of providing a special meaning to an ideal C# operator w.r.t. a user-defined data type such as structures or classes

Operator Overloading

- OPERATORS DESCRIPTION
- +, -, !, ~, ++, – –unary operators take one operand and can be overloaded.
- +, -, *, /, % Binary operators take two operands and can be overloaded.
- ==, !=, = Comparison operators can be overloaded.
- &&, || Conditional logical operators cannot be overloaded directly
- +=, -+, *=, /=, %=, = Assignment operators cannot be overloaded.

Overloading Unary Operators

- The return type can be of any type except void for unary operators like !, ~, + and dot (.) but the return type must be the type of ‘Type’ for – and ++ operators and must be a bool type for true as well as false operators.
- But do remember that the true and false operators can be overloaded as pairs only. The compilation error arises if a class declares one of these operators without declaring the other.

Overloading Unary Operators

- The return type can be of any type except void for unary operators like !, ~, + and dot (.) but the return type must be the type of ‘Type’ for – and ++ operators and must be a bool type for true as well as false operators.
- But do remember that the true and false operators can be overloaded as pairs only. The compilation error arises if a class declares one of these operators without declaring the other.

Overloading Unary Operators

- The return type can be of any type except void for unary operators like !, ~, + and dot (.) but the return type must be the type of ‘Type’ for – and ++ operators and must be a bool type for true as well as false operators.
- But do remember that the true and false operators can be overloaded as pairs only. The compilation error arises if a class declares one of these operators without declaring the other.

Overloading Unary Operators

```
using System;
namespace Calculator {

class Calculator {

    public int number1 , number2;
    public Calculator(int num1 , int num2)
    {
        number1 = num1;
        number2 = num2;
    }

// Function to perform operation
// By changing sign of integers
public static Calculator operator -(Calculator c1)
{
    c1.number1 = -c1.number1;
    c1.number2 = -c1.number2;
    return c1;
}

// Function to print the numbers
public void Print()
{
    Console.WriteLine ("Number1 = " + number1);
    Console.WriteLine ("Number2 = " + number2);
}
}
```

```
class EntryPoint
{

    // Driver Code
    static void Main(String []args)
    {

        // using overloaded - operator
        // with the class object
        Calculator calc = new Calculator(15, -25);

        calc = -calc;

        // To display the result
        calc.Print();
    }
}
}
```

Output :

Number1 = -15
Number2 = 25

Overloading Unary Operators Example

```
using System;
namespace Calculator {

class Calculator {

    public int number1 , number2;
    public Calculator(int num1 , int num2)
    {
        number1 = num1;
        number2 = num2;
    }

// Function to perform operation
// By changing sign of integers
public static Calculator operator -(Calculator c1)
{
    c1.number1 = -c1.number1;
    c1.number2 = -c1.number2;
    return c1;
}

// Function to print the numbers
public void Print()
{
    Console.WriteLine ("Number1 = " + number1);
    Console.WriteLine ("Number2 = " + number2);
}
}
```

```
class EntryPoint
{

    // Driver Code
    static void Main(String []args)
    {

        // using overloaded - operator
        // with the class object
        Calculator calc = new Calculator(15, -25);

        calc = -calc;

        // To display the result
        calc.Print();
    }
}
}
```

Output :

Number1 = -15
Number2 = 25

Overloading Binary Operators

- Binary Operators will work with two Operands.

- Examples of binary operators include the Arithmetic Operators (+, -, *, /, %), Arithmetic Assignment operators (+=, -+, *=, /+, %=) and Relational Operators etc. Overloading a binary operator is similar to overloading a unary operator, except that a binary operator requires an additional parameter.

Overloading Binary Operators

```
// C# program to illustrate the
// Binary Operator Overloading
using System;
namespace BinaryOverload {

class Calculator {

    public int number = 0;

    // no-argument constructor
    public Calculator() {}


    // parameterized constructor
    public Calculator(int n)
    {
        number = n;
    }

    // Overloading of Binary "+" operator
    public static Calculator operator + (Calculator Calc1,
                                         Calculator Calc2)
    {
        Calculator Calc3 = new Calculator(0);
        Calc3.number = Calc2.number + Calc1.number;
        return Calc3;
    }

    // function to display result
    public void display()
    {
        Console.WriteLine("{0}", number);
    }
}


class CalNum {

    // Driver Code
    static void Main(string[] args)
    {

        Calculator num1 = new Calculator(200);
        Calculator num2 = new Calculator(40);
        Calculator num3 = new Calculator();


        num3 = num1 + num2;

        num1.display(); // Displays 200

        num2.display(); // Displays 40

        num3.display(); // Displays 240

    }
}
}
```

C# Delegates

C# Delegates

- 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.

C# Delegate Example

```
//example of delegate in C# which calls add() and mul() methods.
using System;
delegate int Calculator(int n);//declaring delegate
public class DelegateExample {
    static int number = 100;
    public static int add(int n){
        number = number + n;
        return number;
        }
    public static int mul(int n){
        number = number * n;
        return number;
       }
    public static int getNumber() {
        return number;
        }
```

```
    public static void Main(string[] args)  {
        Calculator c1 = new Calculator(add);
//instantiating delegate
        Calculator c2 = new Calculator(mul);
        c1(20);//calling method using delegat
e
        Console.WriteLine("After c1 delegate,
Number is: " + getNumber());
        c2(3);
        Console.WriteLine("After c2 delegate,
Number is: " + getNumber());

    }
}
```

Output:
After c1 delegate, Number is: 120
After c2 delegate, Number is: 360

C# Events

Events in C#

- The Event is something special that is going to happen. Here we will take an example of an event, where Microsoft launches the events for the developer.
- In this Event, Microsoft wants to aware the developer about the feature of the existing or new products.
- For this, Microsoft will use Email or other advertisement options to aware the developer about the Event. So, in this case, Microsoft will work as a publisher who raises the Event and notifies the developers about it.
- Developers will work as the subscriber of the Event who handles the Event.

Events in C#

- Similarly, in C#, Events follow the same concept. In C#, Event can be subscriber, publisher, notification, and a handler. Generally, the User Interface uses the events. Here we will take an example of Button control in Windows.
- Button performs multiple events such as click, mouseover, etc. The custom class contains the Event through which we will notify the other subscriber class about the other things which is going to happen. So, in this case, we will define the Event and inform the other classes about the Event, which contains the event handler.

Events in C#

- The event is an encapsulated delegate. C# and .NET both support the events with the delegates. When the state of the application changes, events and delegates give the notification to the client application.
- Delegates and Events both are tightly coupled for dispatching the events, and event handling require the implementation of the delegates.
- The sending event class is known as the publisher, and the receiver class or handling the Event is known as a subscriber.

Key Points about the Events

- In C#, event handler will take the two parameters as input and return the void.
- The first parameter of the Event is also known as the source, which will publish the object.
- The publisher will decide when we have to raise the Event, and the subscriber will determine what response we have to give.
- Event can contain many subscribers.
- Generally, we used the Event for the single user action like clicking on the button.
- If the Event includes the multiple subscribers, then synchronously event handler invoked.

Events in C# (Contd.)

- Declaration of the Event
  - Syntax
  - public event EventHandler CellEvent;

- Steps for implementing the Event
  - For the declaration of the Event in the class, firstly, the event type of the delegate must be declared.

  - public delegate void CellEventHandler(object sender, EventArgs e);

Events in C# (Contd.)

- Declaration of the Event
  - public event CellEventHandler CellEvent;

- Invokation of the Event
  - if (CellEvent != null) CellEvent(this, e);
  - We can invoke the Event only from within the class where we declared the Event.

- Hooking up the Event
  - OurEventClass.OurEvent += new ChangedEventHandler(OurEventChanged);

Events in C# (Contd.)

- Detach the Event
  - OurEventClass.OurEvent -= new ChangedEventHandler(OurEventChanged);
  - Delegates work as pointer to a function. It is a reference data type and it holds the reference of the method. System.Delegate class implicitly derived all the delegates.
  - Delegate can be declared using the delegate keyword which is followed by the signature
- Syntax of Delegates
  - <access modifier> delegate <return type> <delegate_name>(<parameters>)

Example of Delegates

- public delegate void PrintWord(int value);
- The above PrintWord delegate can be used to point any method which has the same return type and declared parameters with PrintWord.
- Here we will take an example that declares and uses the PrintWord delegates.

Example of Delegates(Contd.)

```
class Program1  {
   // declare delegate
   public delegate void PrintWord(int value);
   static void Main(string[] args)  {
      // Print delegate points to PrintNum
      PrintWord printDel = PrintNum;
      // or
      // Print printDel = new Print(PrintNumber);
      printDel(100000);
      printDel(200);
      // Print delegate points to PrintMoney
      printDel = PrintMoney;
      printDel(10000);
      printDel(200);  }
   public static void PrintNum(int num)  {
      Console.WriteLine("Number: {0,-12:N0}",num);  }
   public static void PrintMoney(int money) {
      Console.WriteLine("Money: {0:C}", money);  }  }
```

Output:

Number: 100,000
Number: 200
Money: $10,000.00
Money: $200.00

Example of Delegates (Contd.)

- Also, we can create the delegate object by using the new operator and specify the name of the method, as shown below:
  - PrintWord printDel = new PrintWord(PrintNum);
- Delegates can be declared, as shown below:
  - public delegate void someEvent();
  - public organize event

Add/Remove Operation in Events

```
public class MyTest  {
    public event EventHandler MyEvent  {
        add  {
            Console.WriteLine("add operation");
        }
        remove  {
            Console.WriteLine("remove operation");
        }
    }
}
public class Test {
    public void TestEvent()  {
        MyTest myTest = new MyTest();
        myTest.MyEvent += myTest_MyEvent;
        myTest.MyEvent -= myTest_MyEvent;
    }
    public void myTest_MyEvent(object sender, EventArgs e)
    {   }
}
```

```
static void Main(string[] args)
{
    Test test = new Test();
    test.TestEvent();
    Console.ReadKey();
}
```

Output

add operation
remove operation

Event - Basic Conceptual Implementation

```
public delegate void EventHandler(object sender, EventArgs e);
public class Publisher: ArrayList {
   public event EventHandler ProdcutAddedInfo;
   protected virtual void OnChanged(EventArgs e)      {
      if (ProdcutAddedInfo != null) ProdcutAddedInfo(this, e);  }
   public override int Add(Object product)  {
      int added = base.Add(product);
      OnChanged(EventArgs.Empty);
      return added;  }
   public override void Clear()  {
      base.Clear();
      OnChanged(EventArgs.Empty);  }
   public override object this[int index]  {
      set  {
         base[index] = value;
         OnChanged(EventArgs.Empty);
      }  }  }
```

Event - Basic Conceptual Implementation

```
public class Subscriber  {
    private Publisher publishers;
    public Subscriber(Publisher publisher)  {
        this.publishers = publisher;
        publishers.ProdcutAddedInfo += publishers_ProdcutAddedInfo;
    }
    void publishers_ProdcutAddedInfo(object sender, EventArgs e)  {
        if (sender == null)  {
            Console.WriteLine("No New Product Added.");
            return;
        }
        Console.WriteLine("A New Prodct Added.");
    }
    public void UnSubscribeEvent()   {
        publishers.ProdcutAddedInfo -= publishers_ProdcutAddedInfo;
    }
}
```

Event - Basic Conceptual Implementation

```
class Program  {
   static void Main(string[] args)  {
      Publisher publisher = new Publisher();
      Subscriber subscriber = new Subscriber(publisher);
      publisher.Add(new Product
      {
         ProductName = "Complan", Price = 20
      });
      publisher.Add(new Product
      {
         ProductName = "Horlics", Price = 120
      });
      publisher.Add(new Product
      {
         ProductName = "Boost", Price = 200
      });
      subscriber.UnSubscribeEvent();
      Console.ReadKey();
   }
}
```

Output

```
A New Product Added.
A New Product Added.
A New Product Added.
```

C# File IO

Outline

- C# FileStream
- C# StreamWriter
- C# StreamReader
- C# TextWriter
- C# TextReader
- C# BinaryWriter
- C# BinaryReader
- C# StringWriter
- C# StringReader
- C# FileInfo
- C# DirectoryInfo
- C# Serialization
- C# Deserialization
- C# System.IO

C# FileStream

- C# FileStream class provides a stream for file operation. It can be used to perform synchronous and asynchronous read and write operations. By the help of FileStream class, we can easily read and write data into file.

C# FileStream example: writing single byte into file

```
using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);
//creating file stream
        f.WriteByte(65);//writing byte into stream
        f.Close();//closing stream
    }
}
```

Output:
A

C# FileStream example: writing multiple bytes into file

```
using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);
        for (int i = 65; i <= 90; i++)
        {
            f.WriteByte((byte)i);
        }
        f.Close();
    }
}
```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

C# FileStream example: reading all bytes from file

```
using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);
        int i = 0;
        while ((i = f.ReadByte()) != -1)
        {
            Console.Write((char)i);
        }
        f.Close();
    }
}
```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

C# StreamWriter

- C# StreamWriter class is used to write characters to a stream in specific encoding. It inherits TextWriter class. It provides overloaded write() and writeln() methods to write data into file.

C# StreamWriter example

```
using System;
using System.IO;
public class StreamWriterExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\output.txt", FileMode.Create);
        StreamWriter s = new StreamWriter(f);

        s.WriteLine("hello c#");
        s.Close();
        f.Close();
     Console.WriteLine("File created successfully...");
    }
}
```

Output:
File created successfully...

C# StreamReader

- C# StreamReader class is used to read string from the stream. It inherits TextReader class. It provides Read() and ReadLine() methods to read data from the stream.

C# StreamReader example to read one line

```
using System;
using System.IO;
public class StreamReaderExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\output.txt", FileMode.OpenOrCrea
te);
        StreamReader s = new StreamReader(f);

        string line=s.ReadLine();
        Console.WriteLine(line);

        s.Close();
        f.Close();
    }
}
```

Output:
Hello C#

C# TextWriter

- C# TextWriter class is an abstract class. It is used to write text or sequential series of characters into file. It is found in System.IO namespace.

C# TextWriter Example

```
using System;
using System.IO;
namespace TextWriterExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TextWriter writer = File.CreateText("e:\\f.txt"))
            {
                writer.WriteLine("Hello C#");
                writer.WriteLine("C# File Handling by Programmer");
            }
            Console.WriteLine("Data written successfully...");
        }
    }
}
```

Output:
Data written successfully...
f.txt:
Hello C# C# File Handling by Programmer

C# TextReader

- C# TextReader class is found in System.IO namespace. It represents a reader that can be used to read text or sequential series of characters.

C# TextReader Example: Read All Data

```
using System;
using System.IO;
namespace TextReaderExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TextReader tr = File.OpenText("e:\\f.txt"))
            {
                Console.WriteLine(tr.ReadToEnd());
            }
        }
    }
}
```

Output:
Hello C#
C# File Handling by Programmer

C# TextReader Example: Read One Line

```
using System;
using System.IO;
namespace TextReaderExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TextReader tr = File.OpenText("e:\\f.txt"))
            {
                Console.WriteLine(tr.ReadLine());
            }
        }
    }
}
```

Output:

Hello C#

C# BinaryWriter

- C# BinaryWriter class is used to write binary information into stream. It is found in System.IO namespace. It also supports writing string in specific encoding.

C# BinaryWriter Example

```
using System;
using System.IO;
namespace BinaryWriterExample{
    class Program  {
        static void Main(string[] args)  {
            string fileName = "e:\\binaryfile.dat";
            using (BinaryWriter writer = new BinaryWriter(File.Open(fileName,
FileMode.Create)))
            {
                writer.Write(12.5);
                writer.Write("this is string data");
                writer.Write(true);
            }
            Console.WriteLine("Data written successfully...");
        }
    }
}
```

Output:
Data written successfully...

C# BinaryReader

- C# BinaryReader class is used to read binary information from stream. It is found in System.IO namespace. It also supports reading string in specific encoding.

C# BinaryReader Example

```
using System;
using System.IO;
namespace BinaryWriterExample{
   class Program{
      static void Main(string[] args){
         WriteBinaryFile();
         ReadBinaryFile();
         Console.ReadKey();  }
      static void WriteBinaryFile(){
         using (BinaryWriter writer = new BinaryWriter(File.Open("e:\\binaryfile.
dat", FileMode.Create))){
            writer.Write(12.5);
            writer.Write("this is string data");
            writer.Write(true);  }}
      static void ReadBinaryFile(){
         using (BinaryReader reader = new BinaryReader(File.Open("e:\\bina
ryfile.dat", FileMode.Open))) {
            Console.WriteLine("Double Value : " + reader.ReadDouble());
            Console.WriteLine("String Value : " + reader.ReadString());
            Console.WriteLine("Boolean Value : " + reader.ReadBoolean());
         } } } }
```

Output:
Double Value : 12.5
String Value : this is string data
Boolean Value : true

C# StringWriter Class

- This class is used to write and deal with string data rather than files. It is derived class of TextWriter class. The string data written by StringWriter class is stored into StringBuilder.

- The purpose of this class is to manipulate string and save result into the StringBuilder.

StringWriter Class Signature

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class StringWriter : TextWriter
```

C# StringWriter Constructors

Constructors	Description
StringWriter()	*It is used to initialize a new instance of the StringWriter class.*
StringWriter(IFormatProvider)	*It is used to initialize a new instance of the StringWriter class with the specified format control.*
StringWriter(StringBuilder)	*It is used to initialize a new instance of the StringWriter class that writes to the specified StringBuilder.*
StringWriter(StringBuilder, ?IFormatProvider)	*It is used to initialize a new instance of the StringWriter class that writes to the specified StringBuilder and has the specified format provider.*

C# StringWriter Properties

Property	Description
Encoding	*It is used to get the Encoding in which the output is written.*
FormatProvider	*It is used to get an object that controls formatting.*
NewLine	*It is used to get or set the line terminator string used by the current* **TextWriter.**

C# StringWriter Methods

Methods	Description
Close()	*It is used to close the current StringWriter and the underlying stream.*
Dispose()	*It is used to release all resources used by the TextWriter object.*
Equals(Object)	*It is used to determine whether the specified object is equal to the current object or not.*
Finalize()	*It allows an object to try to free resources and perform other cleanup operations.*
GetHashCode()	*It is used to serve as the default hash function.*
GetStringBuilder()	*It returns the underlying StringBuilder.*
ToString()	*It returns a string containing the characters written to the current StringWriter.*
WriteAsync(String)	*It is used to write a string to the current string asynchronously.*
Write(Boolean)	*It is used to write the text representation of a Boolean value to the string.*
Write(String)	*It is used to write a string to the current string.*
WriteLine(String)	*It is used to write a string followed by a line terminator to the string or stream.*
WriteLineAsync(String)	*Writes a string followed by a line terminator asynchronously to the current string.(Overrides TextWriter.WriteLineAsync(String).)*

C# StringWriter Example

```
using System;
using System.IO;
using System.Text;
namespace CSharpProgram{
   class Program{
      static void Main(string[] args){
         string text = "Hello, Welcome
           to the Programmer \n" +
            "It is nice platform. \n" +
            "It provides technical Stuff";
         // Creating StringBuilder instance
         StringBuilder sb = new StringBuilder();
        / Passing StringBuilder instance into StringWriter
         StringWriter writer = new StringWriter(sb);
         // Writing data using StringWriter
         writer.WriteLine(text);
         writer.Flush();
 // Closing writer connection
         writer.Close();
         // Creating StringReader instanc
e and passing StringBuilder
         StringReader reader = new String
Reader(sb.ToString());
         // Reading data
         while (reader.Peek() > -1)
         {
         Console.WriteLine(reader.
         ReadLine());   } } } }
```

Output:
Hello, Welcome to the Programmer
It is nice platform.
It provides technical Stuff

C# StringReader Class

StringReader class is used to read data written by the StringWriter class. It is subclass of TextReader class. It enables us to read a string synchronously or asynchronously. It provides constructors and methods to perform read operations.

C# StringReader Signature

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class StringReader : TextReader
```

C# StringReader Constructors

StringReader has the following constructors.

Constructors	Description
StringReader(String)	*Initializes a new instance of the StringReader class that reads from the specified string.*

C# StringReader Methods

Method	Description
Close()	*It is used to close the StringReader.*
Dispose()	*It is used to release all resources used by the TextReader object.*
Equals(Object)	*It determines whether the specified object is equal to the current object or not.*
Finalize()	*It allows an object to try to free resources and perform other cleanup operations.*
GetHashCode()	*It serves as the default hash function.*
GetType()	*It is used to get the type of the current instance.*
Peek()	*It is used to return the next available character but does not consume it.*
Read()	*It is used to read the next character from the input string.*
ReadLine()	*It is used to read a line of characters from the current string.*
ReadLineAsync()	*It is used to read a line of characters asynchronously from the current string.*
ReadToEnd()	*It is used to read all the characters from the current position to the end of the string.*
ReadToEndAsync()	*It is used to read all the characters from the current position to the end of the string asynchronously.*
ToString()	*It is used to return a string that represents the current object.*

C# StringReader Example

```
using System;
using System.IO;
namespace CSharpProgram {
    class Program {
        static void Main(string[] args) {
            StringWriter str = new StringWriter();
            str.WriteLine("Hello, this message is read by StringReader class");
            str.Close();
            // Creating StringReader instance and passing StringWriter
            StringReader reader = new StringReader(str.ToString());
            // Reading data
            while (reader.Peek() > -1)
            {
                Console.WriteLine(reader.ReadLine());
            }
        }
    }
}
```

Output:

Hello, this message is read by StringReader class

C# FileInfo Class

- The FileInfo class is used to deal with file and its operations in C#. It provides properties and methods that are used to create, delete and read file. It uses StreamWriter class to write data to the file. It is a part of System.IO namespace.

C# FileInfo Class Signature

- [SerializableAttribute]
- [ComVisibleAttribute(true)]
- public sealed class FileInfo : FileSystemInfo

C# FileInfo Constructors

Constructor	Description
FileInfo(String)	*It is used to initialize a new instance of the FileInfo class which acts as a wrapper for a file path*

C# FileInfo Properties

Properties	Description
Attributes	*It is used to get or set the attributes for the current file or directory.*
CreationTime	*It is used to get or set the creation time of the current file or directory.*
Directory	*It is used to get an instance of the parent directory.*
DirectoryName	*It is used to get a string representing the directory's full path.*
Exists	*It is used to get a value indicating whether a file exists.*
FullName	*It is used to get the full path of the directory or file.*
IsReadOnly	*It is used to get or set a value that determines if the current file is read only.*
LastAccessTime	*It is used to get or set the time from current file or directory was last accessed.*
Length	*It is used to get the size in bytes of the current file.*
Name	*It is used to get the name of the file.*

C# FileInfo Methods

Method	Description
AppendText()	*It is used to create a StreamWriter that appends text to the file represented by this instance of the FileInfo.*
CopyTo(String)	*It is used to copy an existing file to a new file.*
Create()	*It is used to create a file.*
CreateText()	*It is used to create a StreamWriter that writes a new text file.*
Decrypt()	*It is used to decrypt a file that was encrypted by the current account using the Encrypt method.*
Delete()	*It is used to permanently delete a file.*
Encrypt()	*It is used to encrypt a file so that only the account used to encrypt the file can decrypt it.*
GetAccessControl()	*It is used to get a FileSecurity object that encapsulates the access control list (ACL) entries.*

C# FileInfo Methods

Method	Description
MoveTo(String)	*It is used to move a specified file to a new specified location.*
Open(FileMode)	*It is used to open a file in the specified mode.*
OpenRead()	*It is used to create a read-only FileStream.*
OpenText()	*It is used to create a StreamReader with UTF8 encoding that reads from an existing text file.*
OpenWrite()	*It is used to create a write-only FileStream.*
Refresh()	*It is used to refresh the state of the object.*
Replace(String ,String)	*It is used to replace the contents of a specified file with the file described by the current FileInfo object.*
ToString()	*It is used to return the path as a string.*

C# FileInfo Example: Creating a File

```
using System;
using System.IO;
namespace CSharpProgram  {
   class Program  {
      static void Main(string[] args)  {
         try  {
            // Specifying file location
            string loc = "F:\\abc.txt";
            // Creating FileInfo instance
            FileInfo file = new FileInfo(loc);
            // Creating an empty file
            file.Create();
            Console.WriteLine("File is created Successfuly");
         }catch(IOException e)
         {
            Console.WriteLine("Something went wrong: "+e);
         } } } }
```

Output:

File is created Successfully

C# FileInfo Example: writing to the file

```
using System;
using System.IO;
namespace CSharpProgram{
   class Program {
      static void Main(string[] args) {
         try  {
            // Specifying file location
            string loc = "F:\\abc.txt";
            // Creating FileInfo instance
            FileInfo file = new FileInfo(loc);
            // Creating an file instance to write
            StreamWriter sw = file.CreateText();
            // Writing to the file
            sw.WriteLine("This text is written to the file by using StreamWriter cl
ass.");
            sw.Close();
         }
catch(IOException e)
         {
            Console.WriteLine("Something
 went wrong: "+e);
         } } } }
```

C# FileInfo Example: Reading text from the file

```
using System;
using System.IO;
namespace CSharpProgram {
   class Program  {
      static void Main(string[] args)  {
         try  {
            // Specifying file to read
            string loc = "F:\\abc.txt";
            // Creating FileInfo instance
            FileInfo file = new FileInfo(loc);
            // Opening file to read
            StreamReader sr = file.OpenText();
            string data = "";
            while ((data = sr.ReadLine()) != null){
                Console.WriteLine(data);  } }
         catch (IOException e) {
            Console.WriteLine("Something went wrong: " + e);
         }  }  }}
```

Output:
This text is written to the file by using StreamWriter class

C# DirectoryInfo Class

- DirectoryInfo class is a part of System.IO namespace. It is used to create, delete and move directory. It provides methods to perform operations related to directory and subdirectory. It is a sealed class so, we cannot inherit it.

C# DirectoryInfo Syntax

- [SerializableAttribute]
- [ComVisibleAttribute(true)]
- public sealed class DirectoryInfo : FileSystemInfo

C# DirectoryInfo Constructors

Constructor	Description
DirectoryInfo(String)	*It is used to initialize a new instance of the DirectoryInfo class on the specified path.*

C# DirectoryInfo Properties

Property	Description
Attributes	*It is used to get or set the attributes for the current file or directory.*
CreationTime	*It is used to get or set the creation time of the current file or directory.*
CreationTimeUtc	*It is used to get or set creation time, in coordinated universal time (UTC).*
Exists	*It is used to get a value indicating whether the directory exists.*
Extension	*It is used to get the string representing the extension part of the file.*
FullName	*It is used to get the full path of the directory.*
LastAccessTime	*It is used to get or set the time the current file or directory was last accessed.*
LastAccessTimeUtc	*It is used to get or set the time, in coordinated universal time (UTC) that the current file or directory was last accessed.*
LastWriteTime	*It is used to get or set the time when the current file or directory was last written.*

C# DirectoryInfo Properties

Property	Description
LastWriteTimeUtc	*It is used to get or set the time, in coordinated universal time (UTC), when the current file or directory was last written.*
Name	*It is used to get the name of this DirectoryInfo instance.*
Parent	*It is used to get the parent directory of a specified subdirectory.*
Root	*It is used to get the root portion of the directory.*

C# DirectoryInfo Methods

Method	Description
Create()	*It is used to create a directory.*
Create(DirectorySecurity)	*It is used to create a directory using a DirectorySecurity object.*
CreateObjRef(Type)	*It is used to create an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.*
CreateSubdirectory(String)	*It is used to create a subdirectory or subdirectories on the specified path.*
CreateSubdirectory(String,DirectorySecurity)	*It is used to create a subdirectory or subdirectories on the specified path with the specified security.*
Delete()	*It is used to delete this DirectoryInfo if it is empty.*
Delete(Boolean)	*It is used to delete this instance of a DirectoryInfo, specifying whether to delete subdirectories and files.*

C# DirectoryInfo Methods

Method	Description
EnumerateDirectories()	*It returns an enumerable collection of directory information in the current directory.*
EnumerateFiles()	*It returns an enumerable collection of file information in the current directory.*
GetAccessControl()	*It is used to get a DirectorySecurity object that encapsulates the access control list (ACL) entries for the directory.*
GetDirectories()	*It returns the subdirectories of the current directory.*
GetFiles()	*It returns a file list from the current directory.*
GetType()	*It is used to get the Type of the current instance.*
MoveTo(String)	*It is used to move a DirectoryInfo instance and its contents to a new path.*
Refresh()	*It is used to refresh the state of the object.*
SetAccessControl(DirectorySecurity)	*It is used to set access control list (ACL) entries described by a DirectorySecurity object.*
ToString()	*It returns the original path that was passed by the user.*

C# DirectoryInfo Example

```
using System;
using System.IO;
namespace CSharpProgram{
   class Program  {
      static void Main(string[] args)  {
         // Provide directory name with complete location.
         DirectoryInfo directory = new DirectoryInfo(@"F:\MCA");
         try {
            // Check, directory exist or not.
            if (directory.Exists)  {
               Console.WriteLine("Directory already exist.");
               return;  }
            // Creating a new directory.
            directory.Create();
            Console.WriteLine("The directory is created successfully.");  }
         catch (Exception e) {
            Console.WriteLine("Directory not created: {0}", e.ToString());
         }  }  }}
```

C# DirectoryInfo Example: Deleting Directory

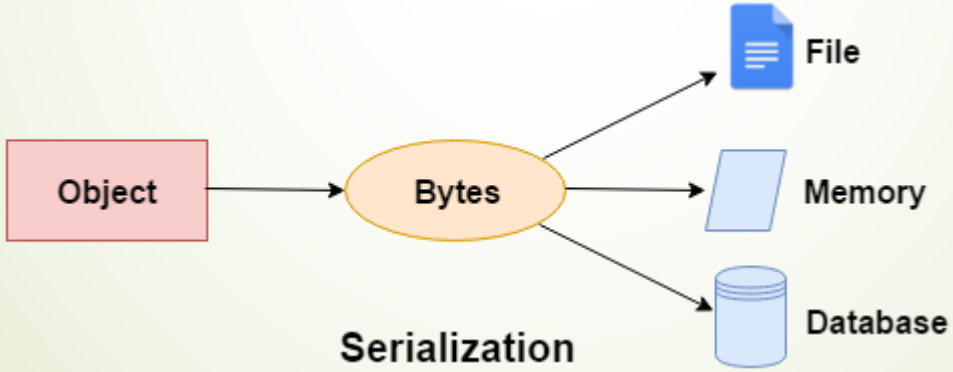
```
using System;
using System.IO;
namespace CSharpProgram {
    class Program {
        static void Main(string[] args) {
            // Providing directory name with complete location.
            DirectoryInfo directory = new DirectoryInfo(@"F:\MCA");
            try {
                // Deleting directory
                directory.Delete();
                Console.WriteLine("The directory is deleted successfully.");
            }  catch (Exception e)
            {
                Console.WriteLine("Something went wrong: {0}", e.ToString());
            }
        }
    }
}
```

Output:

The directory is deleted successfully.

C# Serialization

In C#, serialization is the process of converting object into byte stream so that it can be saved to memory, file or database. The reverse process of serialization is called deserialization. Serialization is internally used in remote applications.



C# SerializableAttribute

To serialize the object, you need to apply SerializableAttribute attribute to the type. If you don't apply SerializableAttribute attribute to the type, SerializationException exception is thrown at runtime.

C# Serialization example

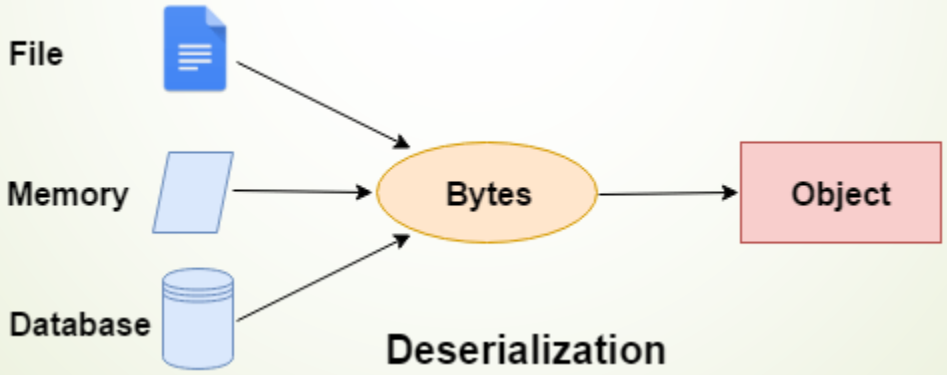
```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Student {
   int rollno;
   string name;
   public Student(int rollno, string name) {
      this.rollno = rollno;
      this.name = name;  }  }
public class SerializeExample  {
   public static void Main(string[] args)  {
      FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCr
eate);
      BinaryFormatter formatter=new BinaryFormatter();
      Student s = new Student(101, "sonoo");
      formatter.Serialize(stream, s);
      stream.Close();
   }  }
```

sss.txt:

JConsoleApplication1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null Student rollnoname e sonoo

C# Deserialization

In C# programming, deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use BinaryFormatter.Deserialize(stream) method to deserialize the stream.



C# Deserialization Example

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Student {
   public int rollno;
   public string name;
   public Student(int rollno, string name)  {
      this.rollno = rollno;
      this.name = name;  }  }
public class DeserializeExample  {
   public static void Main(string[] args)   {
      FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCr
eate);
      BinaryFormatter formatter=new BinaryFormatter();
      Student s=(Student)formatter.Deserialize(stream);
      Console.WriteLine("Rollno: " + s.rollno);
      Console.WriteLine("Name: " + s.name);
       stream.Close();
   }  }
```

Output:
Rollno: 101
Name: sonoo

C# System.IO Namespace

The System.IO namespace consists of IO related classes, structures, delegates and enumerations. These classes can be used to reads and write data to files or data streams. It also contains classes for file and directory support.

C# System.IO Namespace Classes

Class	Description
BinaryReader	*It is used to read primitive data types as binary values in a specific encoding.*
BinaryWriter	*It is used to write primitive types in binary to a stream.*
BufferedStream	*It is used to add a buffering layer to read and write operations on another stream. It is a sealed class.*
Directory	*It is used to expose static methods for creating, moving and enumerating through directories and subdirectories. It is a sealed class.*
DirectoryInfo	*It is used to expose instance methods for creating, moving and enumerating through directories and subdirectories. It is a sealed class.*
DirectoryNotFoundException	*It is used to handle exception related to the file or directory cannot be found.*
DriveInfo	*It is used to access the information on a drive.*
DriveNotFoundException	*It is used to handle drive not found exception.*

C# System.IO Namespace Classes

Class	Description
EndOfStreamException	*It is used to handle end of stream exception.*
ErrorEventArgs	*It provides data for the FileSystemWatcher.Error event.*
File	*This class provides static methods for the creation, copying, deletion, moving and opening of a single file.*
FileFormatException	*It is used to handle file format exception.*
FileInfo	*It is used to provide properties and instance methods for the creation, copying, deletion, moving and opening of files.*
FileLoadException	*It is used to handle file load exception.*
FileNotFoundException	*It is used to handle file load exception.*
FileNotFoundException	*It is used to handle file not found exception.*
FileStream	*It provides a Stream for a file, supporting both synchronous and asynchronous read and write operations.*
FileSystemEventArgs	*It provides data for the directory events.*
FileSystemInfo	*It provides the base class for both FileInfo and DirectoryInfo objects.*
FileSystemWatcher	*It listens to the file system change notifications and raises events when a directory or file in a directory, changes.*
InternalBufferOverflow Exception	*This class is used to handle internal buffer overflow exception.*
InvalidDataException	*It is used to handle invalid data exception.*
IODescriptionAttribute	*It sets the description visual designers can display when referencing an event, extender or property.*

C# System.IO Namespace Classes

Class	Description
IOException	*It is an exception class that handles I/O errors.*
MemoryStream	*It is used to create a stream whose backing store is memory.*
Path	*It performs operations on String instances that contain file or directory path information.*
PathTooLongExcept ion	*It is an exception class and used to handle path too long exception.*
PipeException	*This exception class is used to handle pipe related exception.*
RenamedEventArgs	*It is used to provide data for the Renamed event.*
Stream	*It is used to provide a generic view of a sequence of bytes. It is an abstract class.*
StreamReader	*It is used to implement a TextReader that reads characters from a byte stream.*
StringReader	*It is used to implement a TextReader that reads from a string.*
StringWriter	*It is used to implement a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.*

C# System.IO Namespace Classes

Class	Description
TextReader	*This class is used to represent a reader that can read a sequential series of characters.*
TextWriter	*This class is used to represent a writer that can write a sequential series of characters.*
UnmanagedMemoryAccessor	*It is used to provide random access to unmanaged blocks of memory from managed code.*
UnmanagedMemoryStream	*It is used to get access to unmanaged blocks of memory from managed code.*

System.IO Namespace Structures

Structure	Description
WaitForChangedResult	*It contains information on the change that occurred.*

System.IO Namespace Delegates

Delegates	Description
ErrorEventHandler	*It represents the method that will handle the Error event of a FileSystemWatcher object.*
FileSystemEventHandler	*It represents the method that will handle the Changed, Created or Deleted event of a FileSystemWatcher class.*
RenamedEventHandler	*It represents the method that will handle the renamed event of a FileSystemWatcher class.*

System.IO Namespace Enumerations

Enumeration	Description
DriveType	*It is used to define constants for drive types including CDRom, Fixed, Network etc.*
FileAccess	*It is used to define constants for read, write or read/write access to a file.*
FileAttributes	*It is used to provide attributes for files and directories.*
FileMode	*It is used to specify how the operating system should open a file.*
FileOptions	*It is used to represents advanced options for creating a FileStream object.*
FileShare	*It is used to contain constants for controlling the kind of access other FileStream objects can have to the same file.*

System.IO Namespace Enumerations

Enumeration	Description
HandleInheritability	*It specifies whether the underlying handle is inheritable by child processes.*
NotifyFilters	*It is used to specify changes to watch for in a file or folder.*
SearchOption	*It is used to specify whether to search the current directory or the current directory and all subdirectories.*
SeekOrigin	*It is used to specify the position in a stream to use for seeking.*
WatcherChangeTypes	*It changes that might occur to a file or directory.*

C# Multithreading

C# Multithreading

- Multithreading in C# is a process in which multiple threads work simultaneously.
- It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time.
- To create multithreaded application in C#, we need to use System.Threading namespace.

System.Threading Namespace

- The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:
  - Thread
  - Mutex
  - Timer
  - Monitor
  - Semaphore
  - ThreadLocal
  - ThreadPool
  - Volatile etc.

Process and Thread

- A process represents an application whereas a thread represents a module of the application.
- Process is heavyweight component whereas thread is lightweight.
- A thread can be termed as lightweight subprocess because it is executed inside a process.
- Whenever you create a process, a separate memory area is occupied. But threads share a common memory area.

C# Thread Life Cycle

- In C#, each thread has a life cycle. The life cycle of a thread is started when instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its life cycle is ended.
- There are following states in the life cycle of a Thread in C#.
  - Unstarted
  - Runnable (Ready to run)
  - Running
  - Not Runnable
  - Dead (Terminated)

C# Thread Life Cycle (Contd.)

- Unstarted State
  - When the instance of Thread class is created, it is in unstarted state by default. Runnable State
  - When start() method on the thread is called, it is in runnable or ready to run state.
- Running State
  - Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.
- Not Runnable State
  - The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.
- Dead State
  - After completing the task, thread enters into dead or terminated state.

C# Thread class

- C# Thread class provides properties and methods to create and control threads.

- It is found in System.Threading namespace.

C# Thread Properties

Property	Description
CurrentThread	*returns the instance of currently running thread.*
IsAlive	*checks whether the current thread is alive or not. It is used to find the execution status of the thread.*
IsBackground	*is used to get or set value whether current thread is in background or not.*
ManagedThreadId	*is used to get unique id for the current managed thread.*
Name	*is used to get or set the name of the current thread.*
Priority	*is used to get or set the priority of the current thread.*
ThreadState	*is used to return a value representing the thread state.*

C# Thread Methods

Method	Description
Abort()	*is used to terminate the thread. It raises ThreadAbortException.*
Interrupt()	*is used to interrupt a thread which is in WaitSleepJoin state.*
Join()	*is used to block all the calling threads until this thread terminates.*
ResetAbort()	*is used to cancel the Abort request for the current thread.*
Resume()	*is used to resume the suspended thread. It is obselete.*
Sleep(Int32)	*is used to suspend the current thread for the specified milliseconds.*
Start()	*changes the current state of the thread to Runnable.*
Suspend()	*suspends the current thread if it is not suspended. It is obselete.*
Yield()	*is used to yield the execution of current thread to another thread.*

C# Main Thread Example

- The first thread which is created inside a process is called Main thread. It starts first and ends at last.

```
using System;
using System.Threading;
public class ThreadExample
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "MainThread";
        Console.WriteLine(t.Name);
    }
}
```

C# Threading Example: static method

```
using System;
using System.Threading;
public class MyThread{
    public static void Thread1() {
        for (int i = 0; i < 10; i++)  {
            Console.WriteLine(i);
        }
    }
}
public class ThreadExample {
    public static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
        Thread t2 = new Thread(new ThreadStart(MyThread.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

The output of this program can be anything because there is context switching between the threads.

C# Threading Example: Sleep() method

- The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new Thread
Start(mt.Thread1));
        Thread t2 = new Thread(new Thread
Start(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

C# Threading Example: Sleep() method

- The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new Thread
Start(mt.Thread1));
        Thread t2 = new Thread(new Thread
Start(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
0
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
```

C# Threading Example: Abort() method

```
using System;
using System.Threading;
public class MyThread  {
   public void Thread1()   {
      for (int i = 0; i < 10; i++)  {
         Console.WriteLine(i);
         Thread.Sleep(200);
      } } }
public class ThreadExample {
   public static void Main()  {
      Console.WriteLine("Start of Main");
      MyThread mt = new MyThread();
      Thread t1 = new Thread(new ThreadStart(mt.Thread1));
      Thread t2 = new Thread(new ThreadStart(mt.Thread1));
      t1.Start();
      t2.Start();
      try  {
         t1.Abort();
         t2.Abort();
      }
      catch (ThreadAbortException tae){
         Console.WriteLine(tae.ToString());
      }
      Console.WriteLine("End of Main");
   }
}
```

Output:

Output is unpredictable because thread may be in running state.

C# Threading Example: Join() method

```
using System;
using System.Threading;
public class MyThread  {
   public void Thread1()  {
      for (int i = 0; i < 5; i++)  {
         Console.WriteLine(i);
         Thread.Sleep(200);
      }  }  }
public class ThreadExample {
   public static void Main()  {
      MyThread mt = new MyThread();
      Thread t1 = new Thread(new ThreadStart(mt.Thread1));
      Thread t2 = new Thread(new ThreadStart(mt.Thread1));
      Thread t3 = new Thread(new ThreadStart(mt.Thread1));
      t1.Start();
      t1.Join();
      t2.Start();
      t3.Start();
   }  }
```

Output:

```
0
1
2
3
4
0
0
1
1
2
2
3
3
4
4
```

C# Threading Example: Join() method

```
using System;
using System.Threading;
public class MyThread  {
   public void Thread1()  {
      for (int i = 0; i < 5; i++)  {
         Console.WriteLine(i);
         Thread.Sleep(200);
      }  }  }
public class ThreadExample {
   public static void Main()  {
      MyThread mt = new MyThread();
      Thread t1 = new Thread(new ThreadStart(mt.Thread1));
      Thread t2 = new Thread(new ThreadStart(mt.Thread1));
      Thread t3 = new Thread(new ThreadStart(mt.Thread1));
      t1.Start();
      t1.Join();
      t2.Start();
      t3.Start();
   }  }
```

Output:

```
0
1
2
3
4
0
0
1
1
2
2
3
3
4
4
```

C# Threading Example: ThreadPriority

```
using System;
using System.Threading;
public class MyThread {
   public void Thread1()     {
      Thread t = Thread.CurrentThread;
      Console.WriteLine(t.Name+" is running");  } }
public class ThreadExample  {
   public static void Main()  {
      MyThread mt = new MyThread();
      Thread t1 = new Thread(new ThreadStart(mt.Thread1));
      Thread t2 = new Thread(new ThreadStart(mt.Thread1));
      Thread t3 = new Thread(new ThreadStart(mt.Thread1));
      t1.Name = "Player1";
      t2.Name = "Player2";
      t3.Name = "Player3";
      t3.Priority = ThreadPriority.Highest;
      t2.Priority = ThreadPriority.Normal;
      t1.Priority = ThreadPriority.Lowest;
      t1.Start();
      t2.Start();
      t3.Start();
   }
}
```

Output
Player1 is running
Player3 is running
Player2 is running

Output:
The output is unpredictable because threads are highly system dependent. It may follow any algorithm preemptive or non-preemptive.

C# Thread Synchronization

- The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution Synchronization is a technique that allows only one thread to access the resource for the particular time. No other thread can interrupt until the assigned thread finishes its task.
- In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system. We deal with it by making threads synchronized.
- It is mainly used in case of transactions like deposit, withdraw etc.

Advantage of Thread Synchronization

- Consistency Maintain
- No Thread Interference

C# Lock

- We can use C# lock keyword to execute program synchronously. It is used to get lock for the current thread, execute the task and then release the lock. It ensures that other thread does not interrupt the execution until the execution finish.

- Here, we are creating two examples that executes asynchronously and synchronously.

C# Example: Without Synchronization

```
using System;
using System.Threading;
class Printer {
   public void PrintTable()  {
      for (int i = 1; i <= 10; i++)  {
         Thread.Sleep(100);
         Console.WriteLine(i);
      }  }  }
class Program {
   public static void Main(string[] args)     {
      Printer p = new Printer();
      Thread t1 = new Thread(new ThreadStart(p.PrintTable));
      Thread t2 = new Thread(new ThreadStart(p.PrintTable));
      t1.Start();
      t2.Start();
   }  }
```

Output:

```
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
```

C# Thread Synchronization Example

```
using System;
using System.Threading;
class Printer  {
    public void PrintTable() {
        lock (this)  {
            for (int i = 1; i <= 10; i++)  {
                Thread.Sleep(100);
                Console.WriteLine(i);
            } } } }
class Program {
    public static void Main(string[] args)  {
        Printer p = new Printer();
        Thread t1 = new Thread(new ThreadStart(p.PrintTable));
        Thread t2 = new Thread(new ThreadStart(p.PrintTable));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10
```