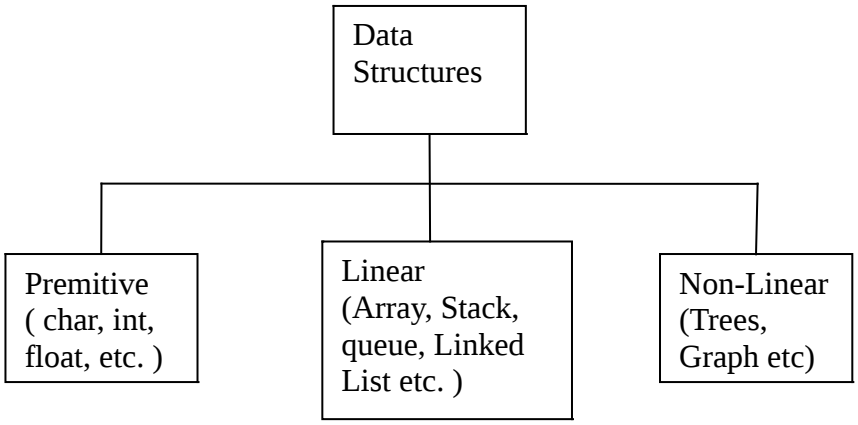
(C# and .Net Framework)
Important questions 2

(1) What are Data Structures?

Logical organization of **data** and **the operations** that are allowed are called Data Structures.



Operation on Data Structures
Insertion – Deletion – Traverse – reversing – Searching – Sorting – Copying – Merging

ARRAY
Linear Array – Organized such that elements are placed in consequitive memory locations

Linked Lists – Collection of data items called Nodes,where the linear order is given by means of logical links.

Stacks and Queues
Stacks
Insertion and Deletion Operations are performed at top of the stack.

Queues
Insertion operation is performed only at the end.
Deletion operation is performed at the other end.

GRAPH

A Graph G consisists of a set of Vertices and a set of Edges. It is represented by :
G(V,E)
where V is the Vertex Set and E is the Edge Set.
A Graph G is said to be a connected Graph,if any two of its vertices can be connected by a path.

(2) How a linked list is implemented in C# ?

A linked list is created in C# ,first creating individual nodes and then linking them together To form a linked list.

Creation of a Node

A node is created using the following code :

```
Class Node
{
Public object Data;
Public node Next;
}
Class List
{
Public static void Main()
{
Node n = new Node();
n.Data = 10;
n.Next = null;
--
}
```

Creation of nodes

```
Node n1 = new Node();
Node n2 = new Node();
N1.Data = 100; n2.Data = 200;
N1.Next = null;  n2.Next = null;
```

Forming a link between two nodes

```
n1.Next = n2;
```

(3) Write program to implement doubly linked list

Doubly-Linked-List Implementation in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 namespace CSTest
{
    class DLinkedList
   {    private int data;
        private DLinkedList next;
        private DLinkedList prev;
        public DLinkedList()
      {  data = 0;
```

```
    next = null;
    prev = null;
  }
  public DLinkedList(int value)
  {   data = value;
    next = null;
    prev = null;
  }
  public DLinkedList InsertNext(int value)
  {     DLinkedList node = new DLinkedList(value);
      if(this.next == null)
      {      // Easy to handle
          node.prev = this;
          node.next = null; // already set in constructor
          this.next = node;
      }
      else
      {      // Insert in the middle
          DLinkedList temp = this.next;
          node.prev = this;
          node.next = temp;
          this.next = node;
          temp.prev = node;
          // temp.next does not have to be changed
      }
      return node;
  }
  public DLinkedList InsertPrev(int value)
  {           DLinkedList node = new DLinkedList(value);
      if(this.prev == null)
      {
          node.prev = null; // already set on constructor
          node.next = this;
          this.prev = node;
      }
      else
      {   // Insert in the middle
          DLinkedList temp = this.prev;
          node.prev = temp;
          node.next = this;
          this.prev = node;
          temp.next = node;
          // temp.prev does not have to be changed
      }
      return node;
  }
  public void TraverseFront()
  {
    TraverseFront(this);
  }
```

```
    public void TraverseFront(DLinkedList node)
    {
        if(node == null)
            node = this;
        System.Console.WriteLine("\n\nTraversing in Forward Direction\n\n");
         while(node != null)
        {
            System.Console.WriteLine(node.data);
            node = node.next;
        }
    }
    public void TraverseBack()
    {
      TraverseBack(this);
    }
    public void TraverseBack(DLinkedList node)
    {
        if(node == null)
            node = this;
        System.Console.WriteLine("\n\nTraversing in Backward Direction\n\n");
        while(node != null)
        {
            System.Console.WriteLine(node.data);
            node = node.prev;
        }
    }
    static void Main(string[] args)
    {     DLinkedList node1 = new DLinkedList(1);
        DLinkedList node3 = node1.InsertNext(3);
        DLinkedList node2 = node3.InsertPrev(2);
        DLinkedList node5 = node3.InsertNext(5);
        DLinkedList node4 = node5.InsertPrev(4);
         node1.TraverseFront();
        node5.TraverseBack();
    }
  }
}
```

Output

```
Traversing in Forward Direction
1
2
3
4
5
 Traversing in Backward Direction
5
4
3
2
```

1

(4) Write a program to implement a binary tree

TREE TRAVERSAL

Each node in a binary search tree contains two references *lchild* and *rchild.*
Traversing a tree is an operation by which node in a tree is visited exactly once.

```
Using system;
Class TreeNode
{
Public int data;
Public TreeNode left;
Public TreeNode right;

Public TreeNode()
{}
Public TreeNode(int Data)
{
Data = Data;
Left = right = null;
}

Public TreeNode(int Data,TreeNode l,TreeNode r)
{
Data = Data;
Left = l;
Right = r:
}
}
Class BinaryTree
{
TreeNode root;
Public BinaryTree()
{
Root = null;
}
Public void insert(int Data)
{
Root = insertNode(root,Data);
}
Public TreeNode insertNode(TreeNode tree,int Data)
{
If (tree == null)
{
Tree = new TreeNode(Data);
Return(tree);
```

```
}
Eleif (Data < tree.Data)
Tree.left = insertNode(tree,left,Data);
Elseif (Data > tree.Data)
Tree.right = insertNode(tree.right,Data);
Return(tree);
}
Public void PreOrder()
{
preOrdertraverse(root);
}
Public void PreOrderTraverse(TreeNode tree)
{
If (tree != null )
{
Console.Write("{0}\t,tree.data);
preOrderTraverse(tree.left);
preOrderTraverse(tree.right);
}
}
Public void postOrder()
{
postOrdertraverse(root);
}
Public void postOrderTraverse(TreeNode tree)
{
If (tree != null)
{
postOrderTraverse(tree.left);
postOrderTraverse(tree.right);
Console.Write("{0}\t,tree.data);

}
}

Public void inOrdeer()
{
inOrderTraverse(root);
}
Public void inOrderTraverse(TreeNode(reeNode tree)
{
If (tree != null)
{
inOrderTraverse(tree.left);
Console.Write("{0}\t,tree.data);
inOrderTraverse(tree.right);

}
}
```

(5) Illustrate the concept of stream and its usage in C#.

STREAM

In C#, all I/O operations are handled using streams. A stream is an abstraction of continuous one way flow of data.

Stream is the abstract base class of all streams. A stream is an abstraction of a sequence of bytes, such as a file, an input/output device, an inter-process communication pipe, or a TCP/IP socket.

Streams helps to achieve

- Persistent storage of objects
- Transmitting objects between nodes in a network
- Encrypting message streams
- Data compression

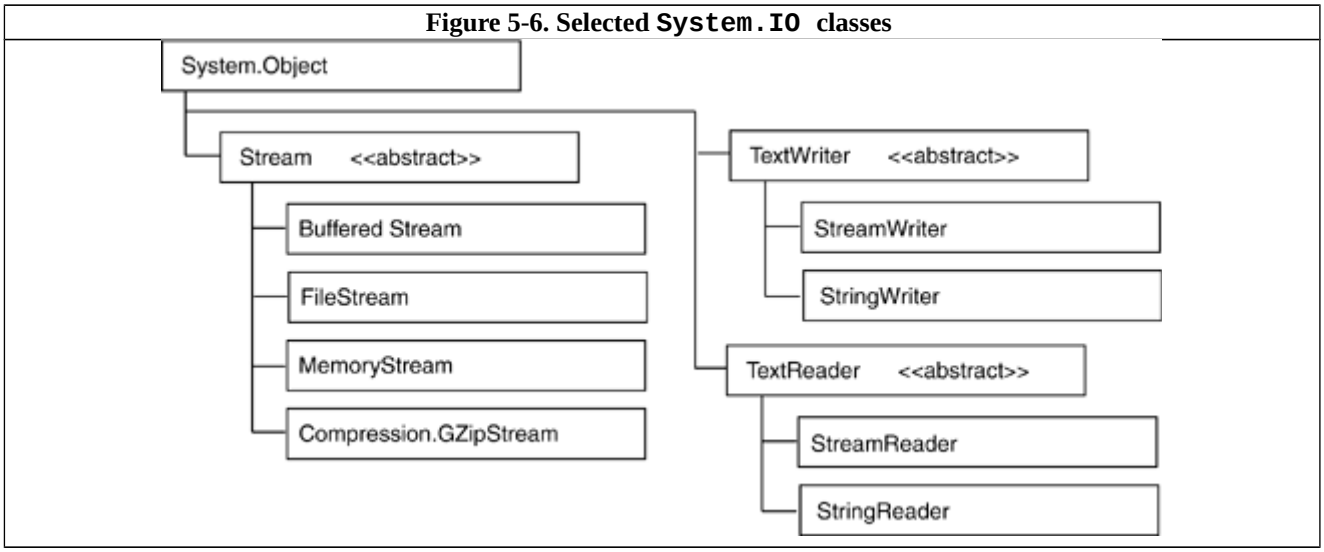
A stream is a path along which data flows. Once the link is established by associating the stream with Source/Destination,Data Flow commences. An analogy can be flow of water in a pipe from a tank to a bucket. The tank is the source of water and the bucket is the destination. The pipe is a path along which water flows. The tap controls the flow of water. Opening the tap commences the flow of water. Closing the tap stops the flow of water.
A similar concept of datqa flow from source to destination is achieved using stgreams in c#.
A **stream** is a path along which data flows.

(6) Explain the classes used in C# to Read and Wrte Streams of data.

System.IO: Classes to Read and Write Streams of Data

The System.IO namespace contains the primary classes used to move and process streams of data. The data source may be in the form of text strings, or raw bytes of data coming from a network or device on an I/O port. Classes derived from the Stream class work with raw bytes; those derived from the TexTReader and TextWriter classes operate with characters and text strings.

Figure 5-6. Selected `System.IO` classes



The Stream Class

This class defines the generic members for working with raw byte streams. Its purpose is to abstract data into a stream of bytes independent of any underlying data devices. This frees the programmer to focus on the data stream rather than device characteristics. The class members support three fundamental areas of operation: reading, writing, and seeking (identifying the current byte position within a stream). Table 5-10 summarizes some of its important members .

Table 5-10. Selected Stream Members

Member	Description
`CanRead`	Indicates whether the stream supports reading, seeking, or writing.
`CanSeek`	
`CanWrite`	
`Length`	Length of stream in bytes; returns long type.
`Position`	Gets or sets the position within the current stream; has long type.
`Close()`	Closes the current stream and releases resources associated with it.
`Flush()`	Flushes data in buffers to the underlying device—for example, a file.
`Read(byte array, offset, count)`	Reads a sequence of bytes from the stream and advances the position within the stream to the number of bytes read. `ReadByte` reads one byte. `Read` returns number of bytes read; `ReadByte` returns -1 if at end of the stream.
`ReadByte()`	
`SetLength()`	Sets the length of the current stream. It can be used to extend or truncate a stream.
`Seek()`	Sets the position within the current stream.
`Write(byte array, offset, count)`	Writes a sequence of bytes (`Write`) or one byte (`WriteByte`) to the current stream. Neither has a return value.
`WriteByte()`	

These methods and properties provide the bulk of the functionality for the `FileStream` , `MemoryStream` , and `BufferedStream` classes.

(7) What are FileStreams? Explain with an example application.

FileStreams

A `FileStream` object is created to process a stream of bytes associated with a backing store —a term used to refer to any storage medium such as disk or memory. The following code segment demonstrates how it is used for reading and writing bytes:

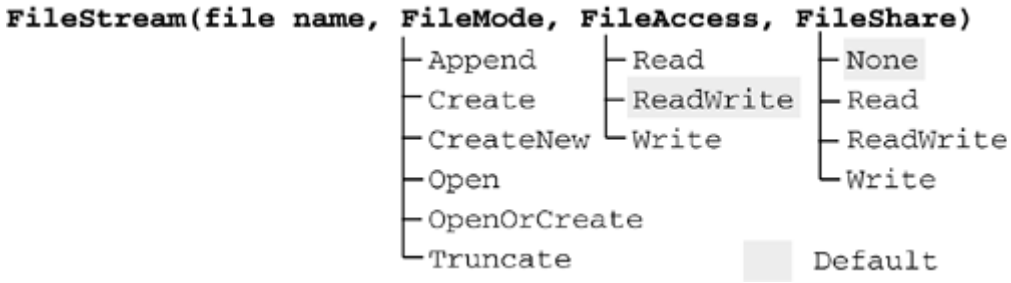
```
try
{
   // Create FileStream object
   FileStream fs = new FileStream(@"c:\artists\log.txt",
         FileMode.OpenOrCreate, FileAccess.ReadWrite);
   byte[] alpha = new byte[6] {65,66,67,68,69,70}; //ABCDEF
   // Write array of bytes to a file
   // Equivalent to: fs.Write(alpha,0, alpha.Length);
   foreach (byte b in alpha) {
      fs.WriteByte(b);}
   // Read bytes from file
   fs.Position = 0;         // Move to beginning of file
   for (int i = 0; i< fs.Length; i++)
      Console.Write((char) fs.ReadByte()); //ABCDEF
   fs.Close();
catch(Exception ex)
{
   Console.Write(ex.Message);
}
```

As this example illustrates, a stream is essentially a byte array with an internal pointer that marks a current location in the stream. The `ReadByte` and `WriteByte` methods process stream bytes in sequence. The `Position` property moves the internal pointer to any position in the stream. By opening the `FileStream` for `ReadWrite` , the program can intermix reading and writing without closing the file.

Creating a FileStream

The `FileStream` class has several constructors. The most useful ones accept the path of the file being associated with the object and optional parameters that define file mode, access rights, and sharing rights. The possible values for these parameters are shown in Figure 5-7.

Figure 5-7. Options for `FileStream` constructors



The `FileMode` enumeration designates how the operating system is to open the file and where to position the file pointer for subsequent reading or writing. Table 5-11 is worth noting because you will see the enumeration used by several classes in the `System.IO` namespace.

Table 5-11. FileMode Enumeration Values

Value	Description
`Append`	Opens an existing file or creates a new one. Writing begins at the end of the file.
`Create`	Creates a new file. An existing file is overwritten.
`CreateNew`	Creates a new file. An exception is thrown if the file already exists.
`Open`	Opens an existing file.
`OpenOrCreate`	Opens a file if it exists; otherwise , creates a new one.
`truncate`	Opens an existing file, removes its contents, and positions the file pointer to the beginning of the file.

The `FileAccess` enumeration defines how the current `FileStream` may access the file; `FileShare` defines how file streams in other processes may access it. For example, `FileShare.Read` permits multiple file streams to be created that can simultaneously read the same file.

(8) What are the file handling classes used in C#?

The following table describes some commonly used classes in the System.IO namespace.

FileStream	It is used to read from and write to any location within a file
BinaryReader	It is used to read primitive data types from a binary stream
BinaryWriter	It is used to write primitive data types in binary format
StreamReader	It is used to read characters from a byte Stream
StreamWriter	It is used to write characters to a stream.
StringReader	It is used to read from a string buffer
StringWriter	It is used to write into a string buffer

DirectoryInfo	It is used to perform operations on directories
FileInfo	It is used to perform operations on files

Writing to a File

```
public class WriteFile
 {
     public WriteFile()
     {
         StreamWriter myFile = File.CreateText("hello.txt");
         myFile.WriteLine("Hello");
         myFile.WriteLine("This is a text file");
         myFile.WriteLine("Goodbye");
         myFile.Close();
     }
 }
```

Reading from a file

Now that you have a file on disc, you will want to open the file and read the data.

```
public class ReadFile
{

    public ReadFile()
    {

        string text;

        if (File.Exists("hello.txt"))
        {

        StreamReader myFile = File.OpenText("hello.txt");
        while ((text = myFile.ReadLine()) != null)
        {
            Console.WriteLine(text);
        }
        myFile.Close();
        }
        else
        {
            Console.WriteLine("File does not exists");
        }
    }

}
```

(9) Define a) FileInfo b) DirectoryInfo

FileInfo and DirectoryInfo are the classes that support file management operations Such as a) creating, b) copying , c) renaming, d) deleting a file / Directory.

FileSystemInfo is an abstract class from which FileInfo and DirectoryInfo classes have been derived.
The methods provided in the classes FileInfo and DirectoryInfo are instance methods.

(10) List some important properties of a) FileInfo b) DirectoryInfo.

Property	Purpose	Applied to
Creation Time	Time file or directory was created	FI / DI
DirectoryName	Full path name of the containing directory	FI
Parent	Full path name of the containing directory	DI
Existing	Whether file or directory exists	FI/DI
LastAccessTime	Time file/directory was last accessed	FI/DI
LastWriteTime	Time file/directory was last modified	FI/DI
Name	Name of the file / Directory	FI/ DI
Root	The root portion of the path	FI / DI
Length	Get the size of the file in bytes	FI

(11) List some important methods of a) FileInfo b) DirectoryInfo.

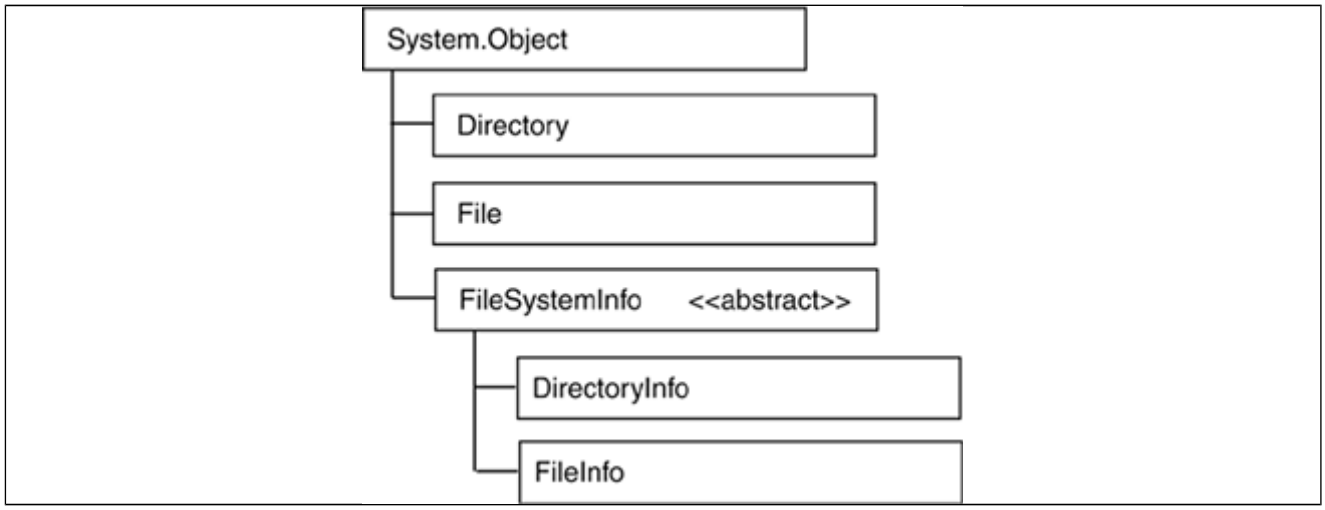
Method	Purpose	Applied to
Create()	Creates a directory or an empty file	FI/DI
Delete()	Deletes a File or Directory. Recursively deletes all directories and files while deleting a directory	FI/DI
MoveTo()	Moves and/or renames the file or directory	FI/DI
GetDirectories()	All directories contained in this directory is returned as an array of objects	DI
GetFiles()	All files contained in this folder is returned	DI

(12) Write a program to illustrate DirectoryInfo and FileInfo classes.

System.IO: Directories and Files

The System.IO namespace includes a set of system- related classes that are used to manage files and directories. Figure 5-9 shows a hierarchy of the most useful classes. Directory and DirectoryInfo contain members to create, delete, and query directories. The only significant difference in the two is that you use Directory with static methods , whereas a DirectoryInfo object must be created to use instance methods. In a parallel manner, File and FileInfo provide static and instance methods for working with files.

Figure 5-9. `Directory` and `File` classes in the `System.IO` namespace




```
// DirectoryInfo
String dir = @”c:\artists”;
DirectoryInfo di = new DirectoryInfo(dir);
di.Refresh();
DateTime IODate = di.CreateTime;
Console.WriteLine(“{0:d}”,IODATE);  // 10/9/2011
//FileInfo


using System;
using System.IO;

class Program
{
    static void Main()
    {
        FileInfo info = new FileInfo("C:\\file.txt");

        DateTime time = info.CreationTime;
        Console.WriteLine(time);

        time = info.LastAccessTime;
        Console.WriteLine(time);

        time = info.LastWriteTime;
        Console.WriteLine(time);
    }
}
```

Output

```
7/17/2010 9:48:48 AM
7/17/2010 9:48:48 AM
8/18/2010 4:48:27 PM
```

(13) Write a program to illustrate StringWriter and StringBuilder

```
//StringWriter usage
StringWriter writer = new StringWriter();
Writer.WriteLine(“Today I have returned,”);
Writer.WriteLine(“after long months “);
Writer.WriteLine(“that seemed like centuries “);
Writer.WriteLine(writer.NewLine);
Writer.close();

//Read string  just written from memory
String myString = writer.ToString();
StringReader reader = new StringReader(myString);
String line = null;
While ( (line = reader.ReadLine() ) != null)
{
Console.WriteLine(line);
}
Reader.close();

//StringBuilder usage
Using system;
Using System.Text;
Public class MyApp
{
Static void Main()
{
//Create comma delimited string with quotes around names
String name1 = “Jan Feb Mar “;
String name2 = “Apr May june”;
StringBuilder sbcsv = new StringBuilder();
Sbcsv.Append(name1).Append(name2);
Sbcsv.Replace(“ “,”’,’”);
//Insert quote at beginning and end of string
Sbcsv.Insert(0,”’”).Append(“’”);
String csv = sbcsv.ToString();
// csv = ‘Jan’,’Feb’,’Mar’,’Apr’,’May’,’Jun’
}
}
```

(14) Write a program to read a file using a byte array

```
string myString = "a test string";
byte[] myByteArray = new byte[myString.Length];
int i = 0;
foreach(char c in InStr.ToCharArray())
{
  myByteArray [i] = (byte)c;
  i++;
}
```

Then some kind soul (thanks Michael) mentioned the Encoding class <slaps forehead>...

```
System.Text.Encoding enc = System.Text.Encoding.ASCII;
byte[] myByteArray = enc.GetBytes("a text string);
string myString = enc.GetString(myByteArray );
```

(15) Write a program to illustrate MemoryStream class

MemoryStreams

This class is used to stream bytes to and from memory as a substitute for a temporary external physical store.

The following example that copies a file first by reading the original file into a memory stream and then writes this to a FileStream using the WriteTo method:

```
FileStream  fsIn = new FileStream(@"c:\manet.bmp",FileMode.Open,FileAcces.Read);
FileStream  fsOut = new
FileStream(@"c:\manetcopy.bmp",FileMode.OpenOrcreate,FileAcces.Write);
memoryStream ms = new MemoryStream();

//Input image byte-by-byte and store in memory stream
Int imgByte;
While ( (imgByte = fsIn.ReadByte() ) != -1)
{
Ms.WriteByte((byte)imgByte);
}
Ms.WriteTo(fsOut);  // Copy image from memory to disk
Byte[] imgArray = ms.ToArray(); // Convert to array of bytes
fsIn.Close();
fsOut.Close();
ms.Close();
```

(16) Define thread.

A thread is a path of execution within a program that can be executed separately. In .NET framework threads run in application domains. A thread is also known as lightweight process.

(17) What are the characteristics or advantages of threads?

- Threads have execution states and may synchronize with another.
- Threads share the same address space.
- Context switching between threads is normally inexpensive.
- Communication between threads is also inexpensive.

(18) What is Multitaking ?

Modern OS hold more than one activity(program) in memory and the processor can switch among all to execute them. Simultaneous execution of many programs on the computer is known as

multitasking.

(19) What is multithreading ?

C# allows the concept of Multithreading , which enables execution of two or more parts of a program concurrently. Each part is known as a thread.

The execution of C# program starts with a single thread known as main thread and that is automatically run by the CLR and the OS. From the main thread, we can create other threads for performing desired tasks. The process of execution of multiple threads is known as multithreading.

(20) What are the characteristics and advantages of multithreading ?

Multithreading allows us to develop efficient programs that could optimize the use of computer resources such as CPU, memory and I/O devices.

System.Threading namespace contains classes and interfaces that are required for developing and running multithreaded programs.

Thread class

The **Thread** class helps us to create and set priorities of a thread.

(21) Enumerate Thread class properties.

PROPERTY	TASK
CurrentThread	Retrieves the name of the thread which is currently running
IsAlive	Indicates current state of thread execution
Name	Specify a name for a thread
Priority	By default the priority is Normal The other value it takes are : Highest,Above Normal,Belowanormal, or Lowest
ThreadState	Indicates the state of a thread. Default value is Unstarted. The other values are Running, Stopped, Suspended, WaitSleepJoin

(22) Enumerate Thread class methods.

The Thread class provides certain methods, that can be used to manage operations such as starting a thread, resuming a suspended thread etc.

Method	TASK
Interrupt	To interrupt the thread which is in the WaitSleepJoin state
Join	To bloc a thread until another thread has terminated
Resume	To resume a thread, which has been suspended earlier.
Sleep	To block the current thread for a specified time period.
SpinWait	To make a thread wait the no. of times specified in Iterations parameter
Start	To start a thread
Suspend	To suspend a thread

(23) How threads are created in a program?

Creation of a Thread in a c# Program

Threading enables your C# program to perform concurrent processing so you can do more than one operation at a time. For example, you can use threading to monitor input from the user, perform background tasks, and handle simultaneous streams of input.

The System.Threading namespace provides classes and interfaces that support multithreaded programming and enable you to easily perform tasks such as creating and starting new threads, synchronizing multiple threads, suspending threads, and aborting threads.

To incorporate threading in your C# code, simply create a function to be executed outside the main thread and point a new Thread object at it. The following code example creates a new thread in a C# application:

```
System.Threading.Thread newThread;
newThread = new System.Threading.Thread(anObject.AMethod);
```

The following code example starts a new thread in a C# application:

```
newThread.Start();
```

Multithreading solves problems with responsiveness and multi-tasking, but can also introduce resource sharing and synchronization issues because threads are interrupted and resumed without warning according to a central thread scheduling mechanism.

(24) Differentiate a process and a thread.

A process is a program in execution. The creation of processes are controlled by the operating system.

A thread is an unit of execution which is a part of a program. Simultaneously many threads can be created and they can be started. It is called multi threading. A thread is under the control of the programmer.

(25) What is synchronization? How it is enforced in C#?

We can easily create multiple thread of execution. It is often necessary for multiple threads to share a resource without control. The behaviour of multi threaded programs sharing a resource yields non-deterministic results.

To provide control over multiple threads under execution, C# allocates methods to co-ordinate activities between threads.

A process which is used to coordinate the activities of two or more threads is called synchronization.

The need for Syncronization arises when two or more threads try to access shared resource that can be used only one thread at a time.

Synchronization enables performance benefits of multithreading as well as maintaining the integrity of the object state and data.

C# uses **lock** keyword to provide data synchronization.

(26) What is critical section?

A method containing code involving shared access/update of data by many threads, is known as a **critical section**.

A critical section can be defined in C# using :

(1) **Monitor** class
(2) **Mutex** class
(3) **Lock** statement

(27) Explain with examples the usage of a) monitor class b) mutex class and c) lock

statement in critical section.

Monitor

The Monitor class can be used in a method, that becomes a critical section.

Example :

```
Public void Updtbalance()
{
Monitor.Enter(this);
----
Monitor.exit(this);
}
```

Explanation:

The method Enter() acquires a Monitor lock and the method Exit() releases the monitor lock.

Mutex

Mutex is a class which is mutually exclusive.

```
Class sample
{
Mutex m = new mutex(false);
Public void updtbalance()
{
m.WaitOne();
..
m.close();
}
..
}
```

Here WaitOne() is the method to acquire mutex.

Lock

The lock statement is used to acquire a mutual exclusion lock.

Example:

```
Class sample
{
Public void updtbalance()
{
Lock(this);
… | code to be locked
… |
}
}
```

(28) Illustrate with an example implementation of synchronization in C#.

Synchronization Example in C# : synchronization.cs

```
Using system;
```

```
Using system.Threading;
Class syncData
{
Int index = 0;
String[] Comment = new String[] { “One”, “Two”, ….   “Ten”};
Public string GetNextComment()
{
//allows only a single thread at a time
Lock(this)
{
If (index < Comment.Length)
{
Return Comment[index++)];
}
Else
{
Return empty; }
}
}
}
Class synchro nization
{
syncData sdat = new syncdata();
static void Main(String[] args)
synchronization sync = nre synchronization((();
Thread t1 = new Thread(new ThreadStart(sync.Comments);
 Thread t2 = new Thread(new ThreadStart(sync.Comments);
Thread t3 = new Thread(new ThreadStart(sync.Comments);
T1.Name = “Thread 1”;
T2.Name = “Thread 2”;
T3.Name = “Thread 3”;

                T1.start((();
T2.start();
T3.start();

Punlic void GetComments()
{
String comment;
Do
{
Comment = sdat.GetNextComment();
Console.WriteLine(“Curent Thread : {0}, comment : {1}” +
Thread.CurrentThread.name,comment);

}  while ( comment != ‘empty’) ;
```

Program Output

Curent Thread : Thread1, Comment : One

Thread 3, comment : two

: Thread 2, comment : ten
: Thread 1 , comment : empty