Listing A Directory

```
class DirTest1{
static void Main(){
System.Console.WriteLine("sub directries in this directory
        "); string[] dirs = Directory.GetDirectories("C:\\");
        int count = dirs.Length;
        for (int i=0; i<count; i++)
        System.Console.WriteLine(dirs[i]);
System.Console.WriteLine("Files in this dierectory are");
        string[] files = Directory.GetFiles("C:\\");
        int count1 = files.Length;
        for (int i=0; i<count1; i++)
        System.Console.WriteLine(files[i]);
}}
```

Formatted Strings in C#

The parameter placeholder {0}, {1} etc in Console.Write is handled by the string formatting. The placeholder has to have the index of the parameter but can also include formatting information. This is used in Console.WriteLine and also string.format hence its inclusion here.

Layout of the PlaceHolder

The three parts are

index, alignment : format

So {0:X} (x means Hexadecimal) or {0,10} meaning output in a width of 10. The alignment value specifies teh width and left or right alignment by using - (left aligned) or + (right aligned) numbers. 10 Means right aligned in a width of 10, -6 means left aligned in a width of 6. Alignment is ignored if the output exceeds the width.
This provides a large number of formatting examples.

List of Numeric Formats

- C or C - For Currency. Uses the cultures currency symbol.
- D or D - Integer types. Add a number for 0 padding eg D5.
- E or e - Scientific notation.
- F or f - Fixed Point.
- G or g - Compact fixed-point/scientific notation.
- N or n - Number. This can be enhanced by a NumberFormatInfo object.
- P or p - Percentage.
- R or r - Round-trip. Keeps exact digits when converted to string and back.
- X or x - Hexadecimal. x - uses abcdef, X use ABCDEF.

Dates can also be specified either using Standard Format strings

- O or o - YYYY-MM-dd:mm:ss:ffffffffzz
- R or r - RFC1123 eg ddd, dd MMM yyyy HH:ss GMT
- s - sortable . yyyy-MM-ddTHH:mm:ss
- u - Universal Sort Date - yyyy-MM-dd HH:mm:ssZ or Format specifiers.

Example

```
using System;
using System.Text;
using System.Globalization;
```

```
namespace ex6
{
   class Program
   {
      static void Main(string[] args)
      {
         // Numeric Formatting Examples
         // integer
   int i = 5678;
   string s = string.Format("{0,10:D}", i); // Into a string right aligned 10 width
   Console.WriteLine("{0,10:D}", i);       // or output direct
   Console.WriteLine("{0,10:D7}", i);       // or output direct with leading 0

         // double and currency in various
  formats double d=47.5;
  double bigd = 19876543.6754;
  Console.WriteLine("{0,15:C2}", d); // In Uk = £47.50 right aligned in 15 width

  Console.WriteLine("{0,15:N10}", bigd); // Number
  Console.WriteLine("{0,15:E3}", d); // Scientific 4.750E+001
  Console.WriteLine("{0,15:F5}", d); // Fixed Point 47.50000
  Console.WriteLine("{0,15:G4}", d); // Compact 47.5
  Console.WriteLine("{0,10:P2}", d/100.0); // %
  Console.WriteLine("{0,15:R}", bigd); // Roundtrip - not a digit lost

         // Hex-a-diddly-decimal
  Console.WriteLine("{0,10:x8}", i); // lowercase 0000162e
  Console.WriteLine("{0,10:X8}", i); // uppercase 0000162E


         // Date formats
  DateTime dt = DateTime.Now;
         // Standards
Console.WriteLine("{0:O}", dt); // O or o yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzz
Console.WriteLine("{0:R}", dt); // R or r ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
Console.WriteLine("{0:s}", dt); // s yyyy'-'MM'-'dd'T'HH':'mm':'ss
Console.WriteLine("{0:u}", dt); // u yyyy'-'MM'-'dd HH':'mm':'ss'Z'

         // Using date/time specifiers
Console.WriteLine("{0:t}", dt); // short time
Console.WriteLine("{0:T}", dt); // long time
Console.WriteLine("{0:d}", dt); // short date
Console.WriteLine("{0:D}", dt); // long date
Console.WriteLine("{0:f}", dt); // long date / short time
Console.WriteLine("{0:F}", dt); // long date / long time
Console.WriteLine("{0:g}", dt); // short date / short time
Console.WriteLine("{0:G}", dt); // short date / long time
Console.WriteLine("{0:o}", dt); // Round Trip

       // roll your own...
```

```
Console.WriteLine("{0:dd/mm/yyyy HH:MM:ss}", dt);//custom - what most people use (UK)!
Console.WriteLine("{0:mm/dd/yyyy HH:MM:ss}", dt);//custom - what most people use (US)!
Console.WriteLine("{0:yyyy/mm/dd HH:MM:ss}", dt);//custom - (Japan) Good for sorting!

Console.WriteLine("{0:dd MMM yyyy HH:MM:ss}", dt); // custom - month (UK)
Console.WriteLine("{0:MMM dd yyyy HH:MM:ss}", dt); // custom - month (US)
Console.WriteLine("{0:yyyy MMM dd HH:MM:ss}", dt); // custom - (Japan)
Console.ReadKey();
      }
   }
}
```

3.3 Multi-Threading

Thread :

- A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
- Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.
Following are the various states in the life cycle of a thread:

- **The Unstarted State**: it is the situation when the instance of the thread is created but the Start method has not been called.
- **The Ready State**: it is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State**: a thread is not runnable, when:
  - Sleep method has been called
  - Wait method has been called
  - Blocked by I/O operations
- **The Dead State**: it is the situation when the thread has completed execution or has been aborted.

The Main Thread
In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.
When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.
The following program demonstrates main thread
execution: using System;

```
using System.Threading;

namespace MultithreadingApplication
{
   class MainThreadProgram
   {
```

```
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}",
            th.Name); Console.ReadKey();
        }
    }
}
```

OUTPUT:
This is MainThread

Creating Threads: Threads are created by creating the object of Thread. The extended Thread class then calls the **Start()**method to begin the child thread execution. The following program demonstrates the concept:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new
            ThreadStart(CallToChildThread); Console.WriteLine("In
            Main: Creating the Child thread"); Thread childThread =
            new Thread(childref); childThread.Start();
            Console.ReadKey();
        }
    }
}
```

OUTPUT:
In Main: Creating the Child
thread Child thread starts

Managing Threads
The Thread class provides various methods for managing threads.
The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication
```

```
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
         // the thread is paused for 5000
         milliseconds int sleepfor = 5000;
         Console.WriteLine("Child Thread Paused for {0}
                     seconds", sleepfor / 1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }

      static void Main(string[] args)
      {
         ThreadStart childref = new
         ThreadStart(CallToChildThread); Console.WriteLine("In
         Main: Creating the Child thread"); Thread childThread =
         new Thread(childref); childThread.Start();
         Console.ReadKey();
      }
   }
}
```

OUTPUT:
In Main: Creating the Child
thread Child thread starts
Child Thread Paused for 5
seconds Child thread resumes
Destroying Threads

The **Abort()** method is used for destroying threads. The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         try
         {

            Console.WriteLine("Child thread starts");
```

```
            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++)
            {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }
            Console.WriteLine("Child Thread Completed");

         }
         catch (ThreadAbortException e)
         {
            Console.WriteLine("Thread Abort Exception");
         }
         finally
         {
            Console.WriteLine("Couldn't catch the Thread Exception");
         }

      }

      static void Main(string[] args)
      {
         ThreadStart childref = new
         ThreadStart(CallToChildThread); Console.WriteLine("In
         Main: Creating the Child thread"); Thread childThread =
         new Thread(childref); childThread.Start();
         //stop the main thread for some
         time Thread.Sleep(2000);
         //now abort the child
         Console.WriteLine("In Main: Aborting the Child
         thread"); childThread.Abort();
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result: In Main: Creating the Child thread
Child thread
starts 0 1 2

In Main: Aborting the Child
thread Thread Abort Exception
Couldn't catch the Thread Exception

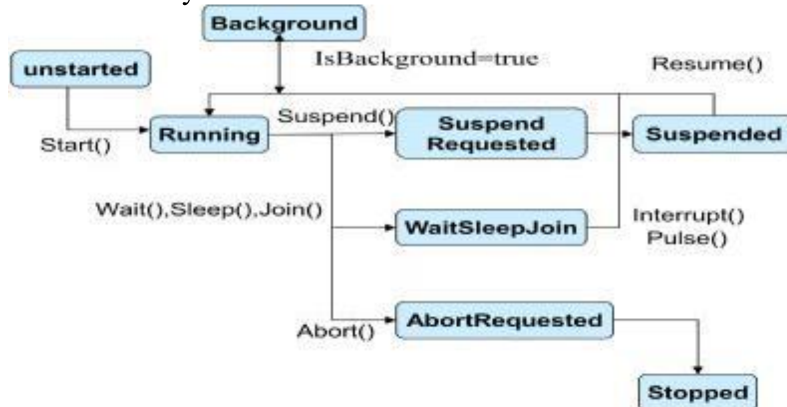
ThreadPriority and ThreadState:

A thread's Priority property determines how much execution time it gets relative to other active threads in the operating system, on the following scale:

This becomes relevant only when multiple threads are simultaneously active.Think carefully before elevating a thread's priority — it can lead to problems such as resource starvation for other threads.Elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority. To perform real-time work, you must also elevate the process priority using the Process class in System.Diagnostics (we didn't tell you how to do this):

```
public enum ThreadPriority
        { Highest,
        AboveNormal,
        Normal,
        BelowNormal,
        Lowest,
}
```

Thread States: You can query a thread's execution status via its **ThreadState** property. This returns a flags enum of type **ThreadState**, which combines three "layers" of data in a bitwise fashion. Most values, however, are redundant, unused, or deprecated. The following diagram shows one "layer"




```
public enum ThreadState {
        Background,
        Unstarted,
        Running,
        WaitSleepJoin,
        SuspendRequested,
        Suspended,
        AbortRequested,
        Stopped
}
```

The following code strips a ThreadState to one of the four most useful values: Unstarted, Running, WaitSleepJoin, and Stopped:

```
public static ThreadState SimpleThreadState (ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
           ThreadState.WaitSleepJoin
           | ThreadState.Stopped);
}
```

The ThreadState property is useful for diagnostic purposes, but unsuitable for synchronization, because a thread's state may change in between testing ThreadState and acting on that information.

Foreground and Background Threads

A *foreground thread* runs indefinitely, whereas a *background thread* stops as soon as the last foreground thread has stopped. You can use the IsBackground property to determine or change the background status of a thread.

Multithreading with Forms and Controls

While multithreading is best suited to running procedures and class methods, you can also use it with forms and controls. If you do so, be aware of the following points:

- Whenever possible, execute the methods of a control only on the thread with which it was created. If you must call a method of a control from another thread, you must use Invoke to call the method.
- Do not use the **SyncLock** (Visual Basic) or **lock** (C#) statement to lock threads that manipulate controls or forms. Because the methods of controls and forms sometimes call back to a calling procedure, you can end up inadvertently creating a deadlock—a situation in which two threads wait for each other to release the lock, causing the application to halt.

Synchronizing threads

Synchronization constructs can be divided into four categories:

1. Simple blocking methods: These wait for another thread to finish or for a period of time to elapse. Sleep, Join, and Task.Wait are simple blocking methods.

2.Locking constructs: These limit the number of threads that can perform some activity or execute a section of code at a time. *Exclusive* locking constructs are most common—these allow just one thread in at a time, and allow competing threads to access common data without interfering with each other. The standard exclusive locking constructs are lock (Monitor.Enter/Monitor.Exit), Mutex, and SpinLock. The nonexclusive locking constructs are Semaphore, SemaphoreSlim, and the reader/writer locks.

3. Signaling constructs: These allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling. There are two commonly used signaling devices: event wait handles and Monitor's Wait/Pulse methods. Framework 4.0 introduces the CountdownEvent and Barrier classes.

4. Nonblocking synchronization constructs: These protect access to a common field by calling upon processor primitives. The CLR and C# provide the following nonblocking constructs: Thread.MemoryBarrier, Thread.VolatileRead, Thread.VolatileWrite, the volatile keyword, and the Interlocked class.

The lock

The **lock** (C#) and **SyncLock** (Visual Basic) statements can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block.

A **lock** or **SyncLock** statement is given an object as an argument, and is followed by a code block that is to be executed by only one thread at a time. For example:

```
public class TestThreading
{
  private System.Object lockThis = new System.Object();

  public void Process()
  {
```

```
        lock (lockThis)
        {
            // Access thread-sensitive resources.
        }
    }

}
```

The argument provided to the **lock** keyword must be an object based on a reference type, and is used to define the scope of the lock.

Monitors

Like the **lock** and **SyncLock** keywords, monitors prevent blocks of code from simultaneous execution by multiple threads. The Enter method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls Exit. This is just like using the**lock** keyword. For example:

```
lock (x)
{
    DoSomething();
}
```

This is equivalent to:

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

Using the **lock** (C#) keyword is generally preferred over using the Monitorclass directly, both because **lock** is more concise, and because **lock** insures that the underlying monitor is released, even if the protected code throws an exception. This is accomplished with the **finally** keyword, which executes its associated code block regardless of whether an exception is thrown.

Synchronization Events and Wait Handles

Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another. This requiressynchronization events, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled. If a thread attempts to wait on an event that is already signaled, then the thread continues to execute without delay.

There are two kinds of synchronization events: AutoResetEvent, and ManualResetEvent. They differ only in that AutoResetEvent changes from signaled to unsignaled automatically any time it activates a thread. Conversely, a ManualResetEvent allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its Reset method is called.

Threads can be made to wait on events by calling one of the wait methods, such as WaitOne, WaitAny, orWaitAll. WaitHandle.WaitOne() causes the thread to wait until a single event becomes signaled,WaitHandle.WaitAny() blocks a thread until one or more indicated events become