**Threading in C#**

C# supports **parallel execution of code** through **multithreading**. A thread is an independent execution path, able to run simultaneously with other threads.

A C# client program (Console, WPF, or Windows Forms) starts in a single thread created **automatically by the CLR** and operating system (the **“main” thread**), and is made multithreaded by creating additional threads.

Namespace: using System.Threading;

Example: static void main()

{

Thread t = new Thread (WriteY);

}

Static void WriteY(){---}

* Once started, a thread’s **IsAlive** property returns **true**, until the point where the thread ends.
* A thread ends when the delegate passed to the **Thread**’s constructor finishes executing. Once ended, a thread cannot restart.
* CLR assigns each thread its own memory stack so that local variables are kept separate.

Example: Class tt = new Class();*//Create a common instance*

* Different thread handling same object will share data work as one.
* Static fields are shared between all threads so execution of one process depends on the shared static value which lacks thread safety.
* C# provides the [lock](http://www.albahari.com/threading/part2.aspx#_Locking) statement for just this purpose:

Example: static readonly object locker = new object();

static void Go(){

lock (locker){---}}

* A thread, while *blocked*, doesn't consume CPU resources.

JOIN AND SLEEP

* You can wait for another thread to end by calling its **Join** method.
* Next thread will start only after execution of this thread.
* **Thread.Sleep** pauses the current thread for a specified period

THREADING:

* Multithreading is managed internally by a **thread scheduler**, a function the CLR typically delegates to OS.
* On a single-processor computer, a thread scheduler performs *time-slicing* — rapidly switching execution between each of the active threads. Like ROUND ROBIN.
* A thread is said to be *preempted* when its execution is interrupted due to an external factor such as time-slicing. In most situations, a thread has no control over when and where it’s pre-empted

Thread and Process:

* Just as processes run in parallel on a computer, threads run in parallel within a single process
* one thread can fetch data in the background, for instance, while another thread can display the data as it arrives.

**ThreadStart** delegate can be used or unused.

Thread t = new Thread (new ThreadStart (Go));

Thread t = new Thread (Go);

**Passing Data to a Thread**

* Another shortcut is to use a lambda expression or anonymous method

new Thread (() =>

{

  Console.WriteLine ("I'm running on another thread!");

  Console.WriteLine ("This is so easy!");

}).Start();

* Another technique is to pass an argument into **Thread**’s **Start** method

t.Start ("Hello from t!");

The output is nondeterministic! Here’s a typical result

for (int i = 0; i < 10; i++)

  new Thread (() => Console.Write (i)).Start();

OUTPUT: 0223557799

Threads execute at the time of START and not at creation.

## Naming Threads

**Thread.CurrentThread.Name** Is a property that allows us to name or get the current thread in execution

## Exception Handling

* You need an exception handler on all thread entry methods in production applications because thread has a separate path from main thread.