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.

The `System.Threading` namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource.

What Are Threads in C#?

In CLR or Windows environment each program you run creates a virtual address space and known as process. Contents of a process is not addressable directly to another process.

Each process has its own thread(s) and this thread has access to all data in that process, In .Net framework, we have managed code and we can say that this thread as managed thread which has access to all data in that process's app domain which is subdivision of process.

So a process in its execution lifetime has this main thread that runs execution starting from the main method and in during this execution it may create one or more thread. This thread can execute code of the same executable or code defined in other dll in the same process

In Windows world, if a process does not have a thread it gets terminate.

When To Use Thread In C#?

You can use threads in the following cases:

Scalability (Be parallel) - If you have long running CPU bound operations, like to compute if 80 digit number is prime or not, you can scale this operation by paralleling this operation to multiple threads

Responsive - You can keep client application responsive by keeping off lengthy operations from main thread (like CPU bound operation) and thus can also leverage the benefit of canceling the task

Leverage asynchronous technique - If you have IO bound operation such reading a web content it may require some time in order of minutes, so you can

leverage another thread to wait for this operation while you perform other task and thus even keep UI responsive

However, C# provides async await syntax for this kind of asynchronous technique.

Also, being asynchronous is not parallel it just keeps the application responsive. Asynchronous means not waiting for an operation to finish, but registering a listener instead.

In general, use parallel threads (using Thread class and Task class) or asynchronous technique (using async await keyword) depending upon whether the problem is CPU bound or IO bound respectively.

Thumb rule is to use threads for CPU bound operation and async for IO bound operation for a client application, and always use async for a server application.

Limitations Of Threads In C#

- Multithreading leads to complex code thus reduce readability. It also increases the difficulty of debugging and testing. However, you can overcome this with good programming practice and commented code.
- Also in single core processor machine, Threads increase execution time (a little) than a sequential program, due to context switching. But still for scalability its good to use threads because when it will run on multi-core processor machine it will scale better.

C# Start New Thread

Threads in C# are modelled by Thread Class. When a process starts (you run a program) you get a single thread (also known as the main thread) to run your application code. To explicitly start another thread (other than your application main thread) you have to create an instance of thread class and call its Start method to run the thread using C#, Let's see an example:

```
using System;
 2  using System.Diagnostics;
 3  using System.Threading;
 4
 5  public class Example
 6  {
 7      public static void Main()
 8      {
 9          //initialize a thread class object
10          //And pass your custom method name to the constructor parameter
11          Thread t = new Thread(SomeMethod);
12
13          //start running your thread
14          t.Start();
15
16          //while thread is running in parallel
17          //you can carry out other operations here
```

```
18
19          Console.WriteLine("Press Enter to terminate!");
20          Console.ReadLine();
21      }
22
23      private static void SomeMethod()
24      {
25          //your code here that you want to run parallel
26          //most of the time it will be a CPU bound operation
27
28          Console.WriteLine("Hello World!");
29      }
30  }
```

When you run this program you may see *Press Enter to terminate!* message first and then *Hello World!* as they both run in parallel, so it is not guaranteed which execute first.

So, We can use Thread's `Join()` method to halt our main thread until reference thread (that is "t" variable in our case) is truly shutdown.

Another method to do this would be by using boolean `IsAlive` property of thread which gives instantaneous snapshot of thread's state whether it is running or not. Like this,

```
while ( t.IsAlive ) { }
```

However, `t.Join()` is the recommended method.

Here is an example:

```
using System;
 2 using System.Diagnostics;
 3 using System.Threading;
 4
 5 public class Example
 6 {
 7     public static void Main()
 8     {
 9         //initialize a thread class object
10         //And pass your custom method name to the constructor parameter
11         Thread t = new Thread(SomeMethod);
12
13         //start running your thread
14         t.Start();
15
16         //while thread is running in parallel
17         //you can carry out other operations here
18
```

```
19          //wait until Thread "t" is done with its execution.
20          t.Join();
21
22          Console.WriteLine("Press Enter to terminate!");
23          Console.ReadLine();
24      }
25
26      private static void SomeMethod()
27      {
28          //your code here that you want to run parallel
29          //most of the time it will be a CPU bound operation
30
31          Console.WriteLine("Hello World!");
32      }
33  }
```

Now,

Thread doesn't start running until you call `thread.Start()` method, So before calling this Start method you can set some properties of a thread like its name and priority. Setting name of the thread will only help you in debugging, by setting name you can easily point out your thread in Visual Studio Thread window, Let's see an example:

```
Thread t = new Thread(SomeMethod);
2
```

```
3 t.Name="My Parallel Thread";

4

5 t.Priority=ThreadPriority.BelowNormal;

6

7 //start running your thread

8 t.Start();
```

Difference Between Foreground And Background Thread In C#

There is also this another thread property IsBackground. If set to true your thread will be a background thread otherwise it will be a foreground thread, by default its false so it will always be a foreground thread, Let's see an example

```
1 Thread t = new Thread(SomeMethod);

2

3 //set thread object as a background thread

4 t.IsBackground = true;

5

6 //start running your thread

7 t.Start();
```

Suppose if a foreground thread is the only thread (your main thread is done with execution and terminated) in your process, so your process is about to exit. However, it won't, your process will wait for foreground thread to complete its execution. Thus, It will prevent application to exit until the foreground thread is done with the execution.

However, background thread will exit as soon as your process exits even though background thread is not completely done with the execution.

C# Start Thread With Parameters

As you saw in example before that we pass method name to thread constructor parameter like this,

```
1 Thread t = new Thread(SomeMethod);
```

We able to do this because this thread constructor takes delegate as parameter. Its supports two type of delegates, Here is the definition of first delegate

```
1 public delegate void ThreadStart()
```

this we already saw in the above example, other is

```
1 public delegate void ParameterizedThreadStart(object obj)
```

If your custom method takes argument you can pass a ParameterizedThreadStart delegate to constructor, Let's see an example:

```
using System;

 2 using System.Diagnostics;

 3 using System.Threading;

 4

 5 public class Example

 6 {

 7     public static void Main()

 8     {
```

```
 9          //initialize a thread class object
10          //And pass your custom method name to the constructor
parameter
11          Thread t = new Thread(Speak);
12
13          //start running your thread
14          //dont forget to pass your parameter for the Speak method
15          //in Thread's Start method below
16          t.Start("Hello World!");
17
18          //wait until Thread "t" is done with its execution.
19          t.Join();
20
21          Console.WriteLine("Press Enter to terminate!");
22          Console.ReadLine();
23      }
24
25      private static void Speak(object s)
26      {
27          //your code here that you want to run parallel
28          //most of the time it will be a CPU bound operation
29
```

```
30          string say = s as string;
31          Console.WriteLine(say);
32
33      }
34  }
```

Did you notice now we need to pass the Speak method argument to Start method.

So far we have used only static method. However, you can also use instance methods as a thread constructor parameter, Let's see an example

```
using System;
 2 using System.Diagnostics;
 3 using System.Threading;
 4
 5 public class Example
 6 {
 7     public static void Main()
 8     {
 9         Person person = new Person();
10
11         //initialize a thread class object
12         //And pass your custom method name to the constructor parameter
13         Thread t = new Thread(person.Speak);
```

```
14
15          //start running your thread
16          //dont forget to pass your parameter for
17          //the Speak method in Thread's Start method below
18          t.Start("Hello World!");
19
20          //wait until Thread "t" is done with its execution.
21          t.Join();
22
23          Console.WriteLine("Press Enter to terminate!");
24          Console.ReadLine();
25      }
26  }
27
28  public class Person
29  {
30      public void Speak(object s)
31      {
32          //your code here that you want to run parallel
33          //most of the time it will be a CPU bound operation
34
```

```
35          string say = s as string;
36          Console.WriteLine(say);
37
38      }
39  }
```

In the above example, we used ParameterizedThreadStart delegate however same applies to ThreadStart delegate, both of them can be used with an instance method.

Thread Life Cycle In C#

So now we know how thread class models a thread. This thread, however, doesn't stay for infinity and has lifespan which is up to the return of the thread delegate method, Let's see an example

```
1  using System;
2  using System.Diagnostics;
3  using System.Threading;
4
5  public class Example
6  {
7      public static void Main()
8      {
9          //initialize a thread class object
10         //And pass your custom method name to the constructor parameter
11
```

```
12          Thread t = new Thread(Speak);
13
14          //start running your thread
15          //dont forget to pass your parameter for the Speak method in
16          //Thread's Start method below
17          t.Start("Hello World!");
18
19          //wait until Thread "t" is done with its execution.
20          t.Join();
21
22          Console.WriteLine("Press Enter to terminate!");
23          Console.ReadLine();
24      }
25
26      private static void Speak(object s)
27      {
28          //your code here that you want to run parallel
29          //most of the time it will be a CPU bound operation
30
31          string say = s as string;
32          Console.WriteLine(say);
```

```
33
34      } // <-- this line is where thread exit and shutdown
35 }
```

Here, line no 33 will be the last execution statement after which thread will be shutdown.

Some other reasons of thread shutdown are as follows:

Synchronous exception

Thread also gets exit if it runs into an unhandled exception. This exception is considered as synchronous exception which occurs in normal sequential program like IndexOutOfRangeExecption.

Asynchronous exception

This exception is an explicit exception raised by calling thread's Abort or Interrupt method in the running thread by some other thread which has reference to the running thread. This exception also exits thread execution. However, this is not a recommended method to shutdown a thread as it leaves the program to some improper state.

C# Stop Thread

Lets start with an example this time,

```
1 using System;
2 using System.Diagnostics;
3 using System.Threading;
4
5 public class Example
6 {
```

```
 7     //set to volatile as its liable to change so we JIT to don't cache
the value
 8     private static volatile bool _cancel = false;
 9
10     public static void Main()
11     {
12         //initialize a thread class object
13         //And pass your custom method name to the constructor parameter
14
15         Thread t = new Thread(Speak);
16
17         //start running your thread
18         //dont forget to pass your parameter for the
19         //Speak method (ParameterizedThreadStart delegate) in Start
method
20         t.Start("Hello World!");
21
22         //wait for 5 secs while Speak method print Hello World! for
multiple times
23         Thread.Sleep(5000);
24
25         //signal thread to terminate
26         _cancel = true;
```

```
27
28
29          //wait until CLR confirms that thread is shutdown
30          t.Join();
31      }
32
33      private static void Speak(object s)
34      {
35
36          while (!_cancel)
37          {
38              string say = s as string;
39              Console.WriteLine(say);
40          }
41
42      }
43  }
```

Here we used a boolean field to signal another thread Speak method to stop running when `_cancel` is set to true.

Did you notice how we need to set the `_cancel` field as volatile. JIT usually cache this kind of fields as it doesn't seem to change within Speak method in the loop. By setting it to volatile we are signaling JIT not to cache this field because it is liable to change.

You can use your own communication mechanism to tell the ThreadStart method to finish, which is recommended method. Alternatively the Thread class has in-built support for instructing the thread to stop. The two principle methods are `Thread.Interrupt()` and `Thread.Abort()`, which is not recommended.

C# THREADPOOL

What is C# Threadpool?

As we learned in previous chapter thread shutdown after its work is done which is a great thing, CLR clears the resource after thread shutdown and thus free up space for smooth program execution without you to write any code for thread management and garbage collection.

However, Creation of thread is something that costs time and resource and thus will be difficult to manage when dealing with a large number of threads. Thread pool is used in this kind of scenario. When you work with thread pool from .NET you queue your work item in thread pool from where it gets processed by an available thread in the thread pool.

But,

After work is done this thread doesn't get shutdown. Instead of shutting down this thread get back to thread pool where it waits for another work item. The creation and deletion of this threads are managed by thread pool depending upon the work item queued in the thread pool. If no work is there in the thread pool it may decide to kill those threads so they no longer consume the resources.

C# Thread Pool Queue

ThreadPool.QueueUserWorkItem is a static method that is used to queue the user work item in the thread pool. Just like you pass a delegate to a thread constructor to create a thread you have to pass a delegate to this method to queue your work. Here is an example,

```
1 using System;
2 using System.Threading;
3
4 class Example1
5 {
6    public static void Main()
7    {
8       // call QueueUserWorkItem to queue your work item
9       ThreadPool.QueueUserWorkItem(Speak);
10
11       Console.WriteLine("Press Enter to terminate!");
12       Console.ReadLine();
13    }
14
15    //your custom method you want to run in another thread
16    public static void Speak(object stateInfo)
17    {
18       // No state object was passed to QueueUserWorkItem, so stateInfo is null.
19       Console.WriteLine("Hello World!");
20    }
21 }
```

as you can see we can directly pass this Speak method name to the QueueUserWorkItem method as it takes WaitCallback delegate as a parameter.

Here is the definition of this delegate,

```
1 public delegate void WaitCallback(object state);
```

See how it share the same signature like our Speak method with void as return type and take object as parameter.

QueueUserWorkItem also has overload for parameterised method like this,

```
1 QueueUserWorkItem(WaitCallback,  Object)
```

Here the first parameter is your method name and the second parameter is the object that you want to pass to your method.

Here is an example,

```
1 using System;
2 using System.Threading;
3
4 class Example1
5 {
6    public static void Main()
7    {
8       // call QueueUserWorkItem to queue your work item
9       ThreadPool.QueueUserWorkItem(Speak, "Hello World!");
10
```

```
11        Console.WriteLine("Press Enter to terminate!");
12        Console.ReadLine();
13    }
14
15    //your custom method you want to run in another thread
16    public static void Speak(object s)
17    {
18        string say = s as string;
19        Console.WriteLine(say);
20    }
21 }
```

Did you notice how we passed our required parameter for Speak method to QueueUserWorkItem as the second parameter.

Limitations To Thread Pool Queue

ThreadPool.QueueUserWorkItem is really easy way to schedule your work into thread pool however it has its limitation, like you cannot tell whether a particular work operation is finished and also it does not return a value.

However, A Task is something that you can use in place of ThreadPool.QueueUserWorkItem. It tells whether an operation is completed and also returns a value after the task is completed.