C# - MULTITHREADING

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.

So far we wrote the programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

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 is not 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 executable, 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 completes execution or is 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();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is MainThread
```

Properties and Methods of the Thread Class

The following table shows some most commonly used **properties** of the **Thread** class:

Property	**Description**
CurrentContext	Gets the current context in which the thread is executing.
CurrentCulture	Gets or sets the culture for the current thread.
CurrentPrinciple	Gets or sets the thread's current principal $forrole - basedsecurity$.
CurrentThread	Gets the currently running thread.
CurrentUICulture	Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
ExecutionContext	Gets an ExecutionContext object that contains information about the various contexts of the current thread.
IsAlive	Gets a value indicating the execution status of the current thread.
IsBackground	Gets or sets a value indicating whether or not a thread is a background thread.
IsThreadPoolThread	Gets a value indicating whether or not a thread belongs to the managed thread pool.
ManagedThreadId	Gets a unique identifier for the current managed thread.
Name	Gets or sets the name of the thread.
Priority	Gets or sets a value indicating the scheduling priority of a thread.
ThreadState	Gets a value containing the states of the current thread.

The following table shows some of the most commonly used **methods** of the **Thread** class:

Sr.No.	**Methods**
1	**public void Abort** Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2	**public static LocalDataStoreSlot AllocateDataSlot** Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	**public static LocalDataStoreSlot AllocateNamedDataSlot**$stringname$ Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	**public static void BeginCriticalRegion**

Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.

5	**public static void BeginThreadAffinity** Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	**public static void EndCriticalRegion** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	**public static void EndThreadAffinity** Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	**public static void FreeNamedDataSlot*stringname*** Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	**public static Object GetData*LocalDataStoreSlotslot*** Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	**public static AppDomain GetDomain** Returns the current domain in which the current thread is running.
11	**public static AppDomain GetDomain** Returns a unique application domain identifier
12	**public static LocalDataStoreSlot GetNamedDataSlot*stringname*** Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
13	**public void Interrupt** Interrupts a thread that is in the WaitSleepJoin thread state.
14	**public void Join** Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms.
15	**public static void MemoryBarrier** Synchronizes memory access as follows: The processor executing the current thread

	cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.
16	**public static void ResetAbort** Cancels an Abort requested for the current thread.
17	**public static void SetData***LocalDataStoreSlotslot, Objectdata* Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead.
18	**public void Start** Starts a thread.
19	**public static void Sleep***intmillisecondsTimeout* Makes the thread pause for a period of time.
20	**public static void SpinWait***intiterations* Causes a thread to wait the number of times defined by the iterations parameter
21	**public static byte VolatileRead***refbyteaddress* **public static double VolatileRead***refdoubleaddress* **public static int VolatileRead***refintaddress* **public static Object VolatileRead***refObjectaddress* Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above.
22	**public static void VolatileWrite***refbyteaddress, bytevalue* **public static void VolatileWrite***refdoubleaddress, doublevalue* **public static void VolatileWrite***refintaddress, intvalue* **public static void VolatileWrite***refObjectaddress, Objectvalue* Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above.
23	**public static bool Yield** Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

Creating Threads

Threads are created by extending the Thread class. 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();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
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();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
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
```