C# Multithreading

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.Threding** namespace.

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. A list of commonly used classes are given below:

* Thread
* Mutex
* Timer
* Monitor
* Semaphore
* ThreadLocal
* ThreadPool
* Volatile etc.

Process and Thread

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

C# Thread Life Cycle

In C#, each thread has a life cycle. The life cycle of a thread is started when instance of *System.Threading.Thread class* is created. When the task execution of the thread is completed, its life cycle is ended.

There are following states in the life cycle of a Thread in C#.

* Unstarted
* Runnable (Ready to run)
* Running
* Not Runnable
* Dead (Terminated)

Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

## Runnable State

When start() method on the thread is called, it is in runnable or ready to run state.

## Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

## Not Runnable State

The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

## Dead State

After completing the task, thread enters into dead or terminated state.

C# Thread class

C# Thread class provides properties and methods to create and control threads. It is found in System.Threading namespace.

C# Thread Properties

A list of important properties of Thread class are given below:

|  |  |
| --- | --- |
| **Property** | **Description** |
| CurrentThread | returns the instance of currently running thread. |
| IsAlive | checks whether the current thread is alive or not. It is used to find the execution status of the thread. |
| IsBackground | is used to get or set value whether current thread is in background or not. |
| ManagedThreadId | is used to get unique id for the current managed thread. |
| Name | is used to get or set the name of the current thread. |
| Priority | is used to get or set the priority of the current thread. |
| ThreadState | is used to return a value representing the thread state. |

C# Thread Methods

A list of important methods of Thread class are given below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| Abort() | is used to terminate the thread. It raises ThreadAbortException. |
| Interrupt() | is used to interrupt a thread which is in *WaitSleepJoin* state. |
| Join() | is used to block all the calling threads until this thread terminates. |
| ResetAbort() | is used to cancel the Abort request for the current thread. |
| Resume() | is used to resume the suspended thread. It is obselete. |
| Sleep(Int32) | is used to suspend the current thread for the specified milliseconds. |
| Start() | changes the current state of the thread to Runnable. |
| Suspend() | suspends the current thread if it is not suspended. It is obselete. |
| Yield() | is used to yield the execution of current thread to another thread. |

C# Main Thread Example

The first thread which is created inside a process is called Main thread. It starts first and ends at last.

Let's see an example of Main thread in C#.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** ThreadExample
4. {
5. **public** **static** **void** Main(**string**[] args)
6. {
7. Thread t = Thread.CurrentThread;
8. t.Name = "MainThread";
9. Console.WriteLine(t.Name);
10. }
11. }

C# Threading Example: static method

We can call static and non-static methods on the execution of the thread. To call the static and non-static methods, you need to pass method name in the constructor of ThreadStart class. For static method, we don't need to create the instance of the class. You can refer it by the name of class.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **static** **void** Thread1()
6. {
7. **for** (**int** i = 0; i < 10; i++)
8. {
9. Console.WriteLine(i);
10. }
11. }
12. }
13. **public** **class** ThreadExample
14. {
15. **public** **static** **void** Main()
16. {
17. Thread t1 = **new** Thread(**new** ThreadStart(MyThread.Thread1));
18. Thread t2 = **new** Thread(**new** ThreadStart(MyThread.Thread1));
19. t1.Start();
20. t2.Start();
21. }
22. }

### C# Threading Example: non-static method

For non-static method, you need to create instance of the class so that you can refer it in the constructor of ThreadStart class.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. **for** (**int** i = 0; i < 10; i++)
8. {
9. Console.WriteLine(i);
10. }
11. }
12. }
13. **public** **class** ThreadExample
14. {
15. **public** **static** **void** Main()
16. {
17. MyThread mt = **new** MyThread();
18. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
19. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
20. t1.Start();
21. t2.Start();
22. }
23. }

### C# Threading Example: performing different tasks on each thread

Let's see an example where we are executing different methods on each thread.

1. **using** System;
2. **using** System.Threading;
4. **public** **class** MyThread
5. {
6. **public** **static** **void** Thread1()
7. {
8. Console.WriteLine("task one");
9. }
10. **public** **static** **void** Thread2()
11. {
12. Console.WriteLine("task two");
13. }
14. }
15. **public** **class** ThreadExample
16. {
17. **public** **static** **void** Main()
18. {
19. Thread t1 = **new** Thread(**new** ThreadStart(MyThread.Thread1));
20. Thread t2 = **new** Thread(**new** ThreadStart(MyThread.Thread2));
21. t1.Start();
22. t2.Start();
23. }
24. }

C# Threading Example: Sleep() method

The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. **for** (**int** i = 0; i < 10; i++)
8. {
9. Console.WriteLine(i);
10. Thread.Sleep(200);
11. }
12. }
13. }
14. **public** **class** ThreadExample
15. {
16. **public** **static** **void** Main()
17. {
18. MyThread mt = **new** MyThread();
19. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
20. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
21. t1.Start();
22. t2.Start();
23. }
24. }

C# Threading Example: Abort() method

The Abort() method is used to terminate the thread. It raises ThreadAbortException if Abort operation is not done.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. **for** (**int** i = 0; i < 10; i++)
8. {
9. Console.WriteLine(i);
10. Thread.Sleep(200);
11. }
12. }
13. }
14. **public** **class** ThreadExample
15. {
16. **public** **static** **void** Main()
17. {
18. Console.WriteLine("Start of Main");
19. MyThread mt = **new** MyThread();
20. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
21. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
23. t1.Start();
24. t2.Start();
25. **try**
26. {
27. t1.Abort();
28. t2.Abort();
29. }
30. **catch** (ThreadAbortException tae)
31. {
32. Console.WriteLine(tae.ToString());
33. }
34. Console.WriteLine("End of Main");
35. }
36. }

C# Threading Example: Join() method

It causes all the calling threads to wait until the current thread (joined thread) is terminated or completes its task.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. **for** (**int** i = 0; i < 5; i++)
8. {
9. Console.WriteLine(i);
10. Thread.Sleep(200);
11. }
12. }
13. }
14. **public** **class** ThreadExample
15. {
16. **public** **static** **void** Main()
17. {
18. MyThread mt = **new** MyThread();
19. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
20. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
21. Thread t3 = **new** Thread(**new** ThreadStart(mt.Thread1));
22. t1.Start();
23. t1.Join();
24. t2.Start();
25. t3.Start();
26. }
27. }

C# Threading Example: Naming Thread

You can change or get the name of the thread by using Name property of Thread class. Let's see an example where we are setting and getting names of the threads.

1. **using** System;
2. **using** System.Threading;
4. **public** **class** MyThread
5. {
6. **public** **void** Thread1()
7. {
8. Thread t = Thread.CurrentThread;
9. Console.WriteLine(t.Name+" is running");
10. }
11. }
12. **public** **class** ThreadExample
13. {
14. **public** **static** **void** Main()
15. {
16. MyThread mt = **new** MyThread();
17. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
18. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
19. Thread t3 = **new** Thread(**new** ThreadStart(mt.Thread1));
20. t1.Name = "Player1";
21. t2.Name = "Player2";
22. t3.Name = "Player3";
23. t1.Start();
24. t2.Start();
25. t3.Start();
26. }
27. }

C# Threading Example: ThreadPriority

Let's see an example where we are changing the priority of the thread. The high priority thread can be executed first. But it is not guaranteed because thread is highly system dependent. It increases the chance of the high priority thread to execute before low priority thread.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. Thread t = Thread.CurrentThread;
8. Console.WriteLine(t.Name+" is running");
9. }
10. }
11. **public** **class** ThreadExample
12. {
13. **public** **static** **void** Main()
14. {
15. MyThread mt = **new** MyThread();
16. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
17. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
18. Thread t3 = **new** Thread(**new** ThreadStart(mt.Thread1));
19. t1.Name = "Player1";
20. t2.Name = "Player2";
21. t3.Name = "Player3";
22. t3.Priority = ThreadPriority.Highest;
23. t2.Priority = ThreadPriority.Normal;
24. t1.Priority = ThreadPriority.Lowest;
26. t1.Start();
27. t2.Start();
28. t3.Start();
29. }
30. }

# C# Thread Synchronization

Synchronization is a technique that allows only one thread to access the resource for the particular time. No other thread can interrupt until the assigned thread finishes its task.

In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system. We deal with it by making threads synchronized.

It is mainly used in case of transactions like deposit, withdraw etc.

Advantage of Thread Synchronization

* Consistency Maintain
* No Thread Interference

C# Lock

We can use C# **lock keyword** to execute program synchronously. It is used to get lock for the current thread, execute the task and then release the lock. It ensures that other thread does not interrupt the execution until the execution finish.

Here, we are creating two examples that executes asynchronously and synchronously.

C# Example: Without Synchronization

In this example, we are not using lock. This example executes asynchronously. In other words, there is context-switching between the threads.

1. **using** System;
2. **using** System.Threading;
3. **class** Printer
4. {
5. **public** **void** PrintTable()
6. {
7. **for** (**int** i = 1; i <= 10; i++)
8. {
9. Thread.Sleep(100);
10. Console.WriteLine(i);
11. }
12. }
13. }
14. **class** Program
15. {
16. **public** **static** **void** Main(**string**[] args)
17. {
18. Printer p = **new** Printer();
19. Thread t1 = **new** Thread(**new** ThreadStart(p.PrintTable));
20. Thread t2 = **new** Thread(**new** ThreadStart(p.PrintTable));
21. t1.Start();
22. t2.Start();
23. }
24. }

### C# Thread Synchronization Example

In this example, we are using lock. This example executes synchronously. In other words, there is no context-switching between the threads. In the output section, we can see that second thread starts working after first threads finishes its tasks.

1. **using** System;
2. **using** System.Threading;
3. **class** Printer
4. {
5. **public** **void** PrintTable()
6. {
7. **lock** (**this**)
8. {
9. **for** (**int** i = 1; i <= 10; i++)
10. {
11. Thread.Sleep(100);
12. Console.WriteLine(i);
13. }
14. }
15. }
16. }
17. **class** Program
18. {
19. **public** **static** **void** Main(**string**[] args)
20. {
21. Printer p = **new** Printer();
22. Thread t1 = **new** Thread(**new** ThreadStart(p.PrintTable));
23. Thread t2 = **new** Thread(**new** ThreadStart(p.PrintTable));
24. t1.Start();
25. t2.Start();
26. }
27. }