

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);
```

```
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
16     //dont forget to pass your parameter for the Speak method in
17
18     //Thread's Start method below
19
20     t.Start("Hello World!");
21
22
23     //wait until Thread "t" is done with its execution.
24
25     t.Join();
26
27
28     Console.WriteLine("Press Enter to terminate!");
29
30     Console.ReadLine();
31
32 }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
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 `IndexOutOfRangeException`.

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.