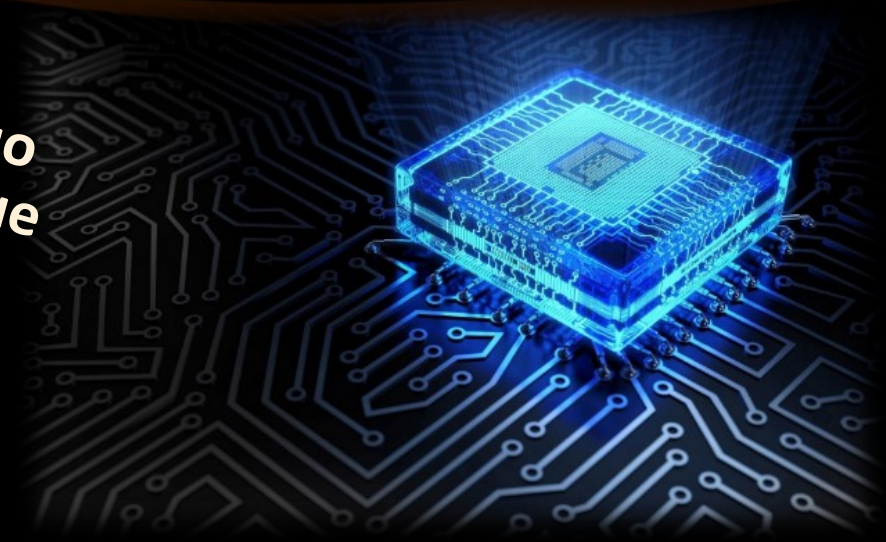
Нишки в C#

Конкурентно програмиране



Инж. Венцеслав Кочанов

Какво е клас на нишка в C#?

Класът Thread в C# отговаря за създаването на персонализирани нишки.

Нишки, ресурси и производителност

Създаването на повече нишки не е свързано със скоростта на обработка или производителността. Всички линии споделят един и същ процесор и ресурси, с които разполага машината. В случай на множество нишки, планировчикът на нишки, с помощта на операционната система, планира нишки и разпределя време за всяка от тях. Но **в машина с един процесор само една нишка може да се изпълнява едновременно. Останалите нишки трябва да изчакат, докато процесорът стане достъпен.** Ако не се управлява правилно, създаването на повече от няколко нишки на машина с един процесор може да създаде пречка за ресурсите. В идеалния случай имаме няколко нишки на процесор. В случай на двуядрени процесори наличието на 4 нишки е идеално. В случай на четириядрен процесор можеме да създадеме до 8 нишки, без да забележим проблеми.

С помощта на класа Thread можем да създаваме както предни, така и фонови нишки. Класът Thread също ни позволява да зададем приоритета на нишката.

Ако отидем на дефиницията на класа Thread, тогава ще видим, че това е запечатан клас и тъй като е запечатан клас, този клас не може да бъде наследен, т.е. по-нататъшно наследяване не е възможно.

Дефиниция:

```
public sealed class Thread :
System.Runtime.ConstrainedExecution.CriticalFinalizerObject
```

Нека да видим какво означава това определение:

- ***public:*** този тип или член може да бъде достъпен от всеки друг код в същото асембли или друго асембли.
- ***sealed:*** Тази ключова дума не позволява други класове да наследяват от класа Thread. С други думи, той гарантира, че класът не може да се използва като базов клас за други класове.
- ***class:*** Показва, че Thread е клас, който е референтен тип в C#.

System.Runtime.ConstrainedExecution.CriticalFinalizerObject: Това е основният клас за обекти, които изискват финализиране, и е предназначен да се използва от средата за изпълнение. В контекста на многонишковостта, критичното финализиране се използва, за да се гарантира, че определени операции се изпълняват по време на финализирането на обект, дори ако времето за изпълнение е в област с ограничено изпълнение.

Класът Thread създава и контролира нишка, задавайки нейния приоритет и получавайки нейния статус.
Когато стартира процес, общата езикова среда за изпълнение /common language runtime - CLR/ автоматично създава единична нишка на преден план за изпълнение на кода на приложението. Заедно с тази основна нишка на преден план, процесът може да създаде една или повече нишки, за да изпълни част от програмния код, свързан с процеса. Тези нишки могат да се изпълняват или на преден план, или на заден план.

По-нататък ще разгледаме и методите по-отделно. А сега да създадем нишка.

Създаване на нишка

В c-sharp за прилагане на нишки в нашето приложение първо трябва да импортираме пространство от имена на нишки, т.е. чрез " System.Threading;". Така че чрез използване на обекти на нишки можем да ги използваме за да създаваме нишки в приложения.

След като импортираме пространството от имена, следващата стъпка е да създадем Thread обекти, както е показано във фрагмента по-долу:

```
using System;
using System.Threading;

class Program{

 public static void Main(){

    //Creates thread objects

    Thread objthread = new Thread();

 }

}
```

В C# CLR автоматично стартира конзолно приложение с една нишка или главна нишка. Ако добавим нова нишка в нашата главна програма, тогава главната програма създава нова нишка (работна нишка) и двете нишки започват своята работа в многонишкова среда едновременно:

```
using System.Threading;

class Thread_Ex {
    static public void Main()
    {
        Console.WriteLine("Before Sleep");

        // Pausing program for 3 seconds
        Thread.Sleep(3000);

        Console.WriteLine("After Sleep");
    }
}
```

Но първо нека видим как работи приложение с два метода без използване на нишки т.е. синхронно:
/Да си припомним какво означава **Синхронен**: Синхронен означава изпълнение на една или повече задачи една след друга. Тук втората работа трябва да изчака, докато първата работа бъде завършена./

```
class Program
   {
      static void Main(string[] args)
      {
         Method1();
         Method2();
      }

      static void Method1()
      {
         for (int i = 0; i <= 10; i++)
         {
            Console.WriteLine("Method One Executed " + i.ToString());

         }
      }
```

```
static void Method2()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("Method Two Executed " + i.ToString());

            }
        }
    }
```

Както виждаме "Method1()" се изпълнява първо, след това последвано от "Method2()", изпълнено по синхронен или последователен начин. Но това, което искаме, е и двата метода да се изпълняват едновременно. Тук механизмът за нишки е полезен. Така че за същото приложение нека добавим механизъм за нишка.

Да разгледаме приложения с един метод и една нишка:

```
using System;
using System.Threading;

static void PrintNumbers()
{
   Console.WriteLine("Starting...");
   for (int i = 1; i < 10; i++)
    {
       Console.WriteLine(i);
    }
}

void Main()
{
  Thread t = new Thread(PrintNumbers);
  t.Start();
  PrintNumbers();
}
```

Да обясним как работи кода по-горе:

Дефинираме метода PrintNumbers, който ще се използва както в основната, така и в новосъздадената нишка. След това в стъпка 5 създадохме нишка, която изпълнява PrintNumbers. Когато конструираме нишка, екземпляр на делегата ThreadStart или ParameterizedThreadStart се предава на конструктора. Компилаторът на C# създава този обект /без ние да виждаме това как се случва/ когато просто напишем името на метода, който искаме да изпълним в различна нишка. След това стартираме нишка и стартираме PrintNumbers по обичайния начин в основната нишка. В резултат на това ще има два диапазона от числа от 1 до 10, пресичащи се на случаен принцип. Това илюстрира, че методът PrintNumbers се изпълнява едновременно в главната нишка и в другата нишка.

Пример 2:

```
using System.Threading;

public class Program
{
    public static void Main()
    {
        Thread thread = new Thread(new ThreadStart(Work));
        thread.Start();  // Starts the thread.

        for (int i = 0; i < 100; i++) Console.Write('Main');
    }

    public static void Work()
    {
        for (int i = 0; i < 100; i++) Console.Write('Other');
    }
}
```

В този кодов фрагмент създаваме нова нишка и я насочваме към метод Work(). Веднага след като инициираме thread.Start(), той се изпълнява заедно с метода Main(), интегриран в основната нишка.

Бележка:

Нишките в C# са свързани с изпълнението на множество процеси или задачи едновременно. Едноядрен процесор не може да изпълнява множество задачи точно по едно и също време. Той изпълнява една задача, превключва към друга задача, изпълнява малко от нея и след това се връща към предишната работа. Този процес е известен като превключване на контекста, а илюзията за паралелизъм, породена от него, се нарича едновременност.

Съвременните процесори са с множество ядра, които са способни да изпълняват задачи паралелно – постигайки истинска многозадачност.

Пример 3
Използване на основна нишка и метода Sleep():

```
using System.Text;
using System.Threading;

namespace Threading
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread objworkerthread = new Thread(WorkerThread);
            objworkerthread.Start();

            for (int k = 0; k <= 10; k++)
            {
                Console.WriteLine("main thread");
                Thread.Sleep(4000); //Sleep for 4 seconds
            }

        }
```

```
static void WorkerThread()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("worker thread");
                Thread.Sleep(4000); //Sleep for 4 seconds

            }
        }


    }
}
```

Да се върнем на предишното приложение с двата синхронно изпълняващи се метода. По-долу създаваме клас нишки и обекти нишки:

```
using System;
using System.Threading;

class Program{

 public static void Main(){
//Creates thread objects

  Thread thread1 = new Thread(Method1);
  Thread thread2 = new Thread(Method2);

 }

}
```

Извикваме на обекти на нишка:

```
using System;
using System.Threading;

class Program{
static void Method1()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("Method One Executed " + i.ToString());

                Thread.Sleep(4000); //Sleep for 4 seconds
            }
        }

        static void Method2()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("Method Two Executed " + i.ToString());
                Thread.Sleep(4000); //Sleep for 4 seconds

            }
         }
```

```
public static void Main(){

  //Creates thread objects
  Thread thread1 = new Thread(Method1);
  Thread thread2 = new Thread(Method2);

  thread1.Start();
  thread2.Start();
}
```

В нашия горен пример в конзолно приложение създадохме два метода (съответно Method1 и Method2). Вътре в двете тела на метода създадохме цикъл от 10 пъти и след всеки изчакахме 4 секунди. Имайте предвид, че за чакане сме използвали функция от клас "Thread", т.е. "Sleep()". Сега вътре в тялото на основната функция създадохме обектите на нишката от класа "Thread" и използвайки тези обекти, като извикваме функцията "Start()" и започнахме изпълнението на двете функции едновременно.

Приложение „Truck“ – Товарене на камион с методи:

```
using System;
class Program
{
    // This is the entry point of a C# program
    static void Main(string[] args)
    {
        Console.WriteLine("Main thread starts here.");
        HeavyLifting();
        Working();
        Console.WriteLine("Main thread ends here.");
    }
```

```
public static void HeavyLifting()
    {
        Console.WriteLine("I'm lifting a truck!!");
        Thread.Sleep(1000);
        Console.WriteLine("Stop! You need 3 seconds of rest.");
        Thread.Sleep(1000);
        Console.WriteLine("1....");
        Thread.Sleep(1000);
        Console.WriteLine("2....");
        Thread.Sleep(1000);
        Console.WriteLine("3....");
        Console.WriteLine("I'm ready!");
    }
    public static void Working()
    {
        Console.WriteLine("We work  here!");
        for (int i = 0; i < 20; i++)
            Console.Write($"{i} ");
        Console.WriteLine();
        Console.WriteLine("I'm done.");
    }
}
```

В **Листинг Truck** HeavyLifting е първият метод, който отнема около 4 секунди. След това се извиква методът Working. В този случай методът Working трябва да изчака, докато се изпълни методът HeavyLifting. Но ако трябва и двете операции да се вършат едновременно?
Тогава е го решението:

Заменяме този код,
HeavyLifting();

със следния код:

```
// Създаваме нишка и извикваме фонов метод
Thread backgroundThread = new Thread(new ThreadStart(HeavyLifting));
// Стартираме нишката
backgroundThread.Start();
```

Ето го и новият код:

```
static void Main(string[] args)
{
    // Стартира Main нишката
    Console.WriteLine("Main thread starts here.");


    Thread backgroundThread = new Thread(new ThreadStart(HeavyLifting));

    backgroundThread.Start();


    Working();
    // Изпълнениет свършва тук
    Console.WriteLine("Main thread ends here.");
    Console.ReadKey();
}
```

Сега когато стартираме програмата няма да видиме забавяне в изпълнението на метода Working.

Методът Thread.Start() стартира нова нишка. Тази нова нишка се нарича работна нишка или вторична нишка. В този код създадохме нов обект на нишка, използвайки класа Thread, който приема **делегат ThreadStart като параметър** с метода, изпълняван във фонов режим.
ThreadStart е делегат в C#, който представлява метод, който не приема никакви аргументи и не връща никаква стойност. Обикновено се използва при създаване на нова нишка в C#, чрез предаване на метода, който искате да изпълните на новата нишка, като параметър на ThreadStart. Без значение колко време отнема методът на работната нишка, кодът на основната нишка ще бъде изпълнен един до друг.

Като примера по-долу:

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating the ThreadStart Delegate instance by passing the
            //method name as a parameter to its constructor
            ThreadStart obj = new ThreadStart(DisplayNumbers);

            //Passing the ThreadStart Delegate instance as a parameter to its constructor
            Thread t1 = new Thread(obj);

            t1.Start();
            Console.Read();
        }

        static void DisplayNumbers()
        {
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine("Method1 :" + i);
            }
        }
    }
}
```

В примера по-горе първо създаваме екземпляр на делегат ThreadStart и на конструктора на делегат ThreadStart предаваме функцията DisplayNumbers като параметър. След това създаваме екземпляр на класа Thread и на конструктора на класа Thread предаваме екземпляра на делегата ThreadStart като параметър, който сочи към функцията DisplayNumbers. И накрая, когато извикаме метода Start на екземпляра Thread, който ще извика екземпляра на делегата ThreadStart и екземплярът на делегата ще извика метода, към който сочи, т.е. функцията DisplayNumbers ще започне своето изпълнение.

Пример: да създадем приложение за ресторант като го реализираме с и без нишки:

Първо без нишки:

```
public class Restaurant {
  public void Cooking () {
    for (int i = 0; i < 100; i++)
      Console.WriteLine("Cooking food");
  }

  public void Orders() {
    for (int i = 0; i < 100; i++)
      Console.WriteLine("Accepts requests for the kitchen");
  }
}

static void Main(string[] args) {
    Restaurant rest = new Restaurant();
    rest.Cooking;
    rest.Orders;
    Console.WriteLine("End of lunch!");
}
```

Сега с нишки:

```
public class Restaurant {
  public void Cooking () {
    for (int i = 0; i < 100; i++)
      Console.WriteLine("Cooking food");
  }

  public void Orders() {
    for (int i = 0; i < 100; i++)
      Console.WriteLine("Accepts requests for the kitchen");
  }
}

static void Main(string[] args) {
    Restaurant rest = new Restaurant();
    Thread t1 = new Thread( rest.Cooking);
    Thread t2 = new Thread(  rest.Orders);
// Стартираме нишките
    t1.Start();
    t2.Start();

    t1.Join();
    t2.Join();
    Console.WriteLine("End of lunch!");
}
```

В този пример функцията Main() стартира две нишки – t1 за „Готвене“ и t2 за „Заявки“. Това е подобно на Готвач, който се справя с множество задачи в ресторанта. Когато и двете нишки завършат изпълнението (сигнализирано от t1.Join() и t2.Join()), обядът свършва.

Ще разгледаме методите Thread.Join(), Thread.Sleep() и Thread.Abort()

Метод Thread.Join() с пример:
Присъединяването изчаква нишката да приключи. Методът на присъединяване, когато е прикрепен към която и да е нишка, кара тази нишка да завърши изпълнението си първа или да приключи първа и спира други процеси. С прости думи можем да изчакаме друга нишка да приключи, като извикаме нейния метод Join. Можем да включим TimeSpan или милисекунди с метода Join.

```
class Program
    {
        static void Main(string[] args)
        {
            Thread objThread = new Thread(ProcessJoin);
            objThread.Start();
            objThread.Join();
            Console.WriteLine("work completed..!");
        }
        static void ProcessJoin()
        {
            for (int i = 0; i <= 10; i++)
            {
                Console.WriteLine("work is in progress..!");
            }
        }

    }
```

Кодът по-горе е прост пример за метод Join. В тази програма създадохме нова нишка за метода "ProcessJoin" и изчакахме "ProcessJoin" да завърши работата си чрез добавяне на метода Join. Примерът по-горе първо отпечатва "работата е в ход..!" 10 пъти и изход, след което отпечатва "работата е завършена..!" и изход от главния метод.

Метод Thread.Sleep() с пример

Методът на заспиване се използва за спиране на текущата нишка или поставяне на пауза в текущата нишка за определено време. Времето може да бъде зададено в милисекунди или TimeSpan и в режим на заспиване нишката не консумира и CPU ресурси, което индиректно спестява памет за други процеси.

```
using System.Threading;
using System.Diagnostics;

    class Program
    {

        static void Main(string[] args)
        {
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();

            Thread objThread = new Thread(ProcessJoin);
            objThread.Start();
            objThread.Join();

            stopWatch.Stop();
            TimeSpan ts = stopWatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}",ts.Hours, ts.Minutes, ts.Seconds);
            Console.WriteLine("TotalTime " + elapsedTime);
          }
```

```
        Console.WriteLine("work completed..!");


    }

    static void ProcessJoin()
    {
        for (int i = 0; i <= 5; i++)
        {
            Console.WriteLine("work is in progress..!");
            Thread.Sleep(4000); //Sleep for 4 seconds

        }

    }
```

В горния пример за код "ProcessJoin" прави цикъл 5 пъти и за всеки цикъл той заспива за 4 секунди. Изчислихме общото време на сън с помощта на пространството на имената на System.Diagnostics „хронометър“.

Thread.Abort() Метод с пример

Методът Thread.Abort помага за прекратяване или завършване на нишка. Методът Abort предизвиква ThreadAbortException в нишката, за да извърши процес на прекратяване, и хвърля ThreadAbortException в нишката, за да я прекрати. Това изключение може да бъде уловено в кода на приложението.

```
Thread objThread = new Thread(ProcessJoin);
        objThread.Start();
        objThread.Join();

        objThread.Abort();
```

Друг пример:

```
using System;
using System.Threading;
static void PrintNumbersWithDelay()
{
Console.WriteLine("Starting...");
for (int i = 1; i < 10; i++)
{
Thread.Sleep(TimeSpan.FromSeconds(2));
Console.WriteLine(i);
}
}
Console.WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
Thread.Sleep(TimeSpan.FromSeconds(6));
t.Abort();
Console.WriteLine("A thread has been aborted");
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

Горният пример работи по следният начин:

Когато основната програма и отделна нишка за отпечатване на числа се изпълняват, изчакваме 6 секунди и след това извикваме метод t.Abort на нишка. Това инжектира метод ThreadAbortException в нишка, причинявайки нейното прекъсване. Много е опасно, като цяло поради това изключение може да се случи във всеки момент и може напълно да унищожи приложението. Освен това не винаги е възможно да се прекрати нишка по този начин. Текущата нишка може да откаже да прекъсне, като обработи това изключение и извика метода Thread.ResetAbort. Така че не е Препоръчително е да използваме метода Abort, за да затворите нишка. Има различни методи, които са предпочитани, като предоставяне на метод CancellationToken за отмяна на изпълнение на нишка.

Получаване на текущата нишка в C#

Thread.CurrentThread връща текущата нишка, която изпълнява текущия код. Следният кодов фрагмент отпечатва свойствата на текущата нишка, като нейния идентификатор, приоритет, име и култура,

```
Thread currentThread = Thread.CurrentThread;
Thread currentThread = Thread.CurrentThread;
Console.WriteLine("Thread Id {0}: ", currentThread.ManagedThreadId);
Console.WriteLine("Is thread background: {0}", currentThread.IsBackground);
Console.WriteLine("Priority: {0}", currentThread.Priority);
Console.WriteLine("Culture: {0}", currentThread.CurrentCulture.Name);
Console.WriteLine("UI Culture: {0}", currentThread.CurrentUICulture.Name);
Console.WriteLine();
```

Нишката наследява културата на потребителския интерфейс от текущата система.

Приоритети на нишки в C#

В C# всяка нишка има приоритет, който определя колко често нишката получава достъп до процесора. Като цяло нишка с нисък приоритет ще получи по-малко процесорно време от нишка с висок приоритет. Важното тук е колко процесорно време ще получи една нишка, това не зависи само от нейния приоритет, но зависи и от вида на операцията, която изпълнява.

Например, ако нишка с висок приоритет чака някои споделени I/O ресурси да изпълнят задачата си, тогава тя ще бъде блокирана и изтеглена от процесора. И в същото време нишка с по-нисък приоритет може да получи времето на процесора и да завърши изпълнението си, ако не изисква такива споделени I/O ресурси. В сценарии като този нишка с висок приоритет може да получи по-малко процесорно време от нишка с нисък приоритет за определен период от време.

Друг фактор, който определя колко процесорно време е разпределено за една нишка, е как планирането на задачите се изпълнява от операционната система.

Приоритет на нишка

Когато създадохме екземпляр на класа Thread, обектът на нишката получава настройка за приоритет по подразбиране. Можем да получим или зададем приоритета на нишка, като използваме следното свойство Priority на класа Thread.

ThreadPriority Priority {get; set;}: Това свойство се използва за получаване или задаване на стойност, указваща приоритета на планиране на нишка. Връща една от стойностите на ThreadPriority. Стойността по подразбиране е ThreadPriority.Normal.

ThreadPriority предоставя следните 5 свойства:

Най-нисък = 0: Нишката може да бъде планирана след нишки с всеки друг приоритет. Това означава, че нишки с най-нисък приоритет могат да бъдат планирани след нишки с всеки друг по-висок приоритет.

BelowNormal = 1: Нишката може да бъде планирана след нишките с нормален приоритет и преди тези с най-нисък приоритет. Това означава, че нишки с приоритет BelowNormal могат да бъдат планирани след нишки с нормален приоритет и преди нишки с най-нисък приоритет.

Нормално = 2: Нишката може да бъде планирана след нишки с приоритет AboveNormal и преди тези с приоритет BelowNormal. Нишките имат нормален приоритет по подразбиране. Това означава, че нишки с нормален приоритет могат да бъдат планирани след нишки с приоритет AboveNormal и преди нишки с приоритет BelowNormal и най-нисък.

AboveNormal = 3: Нишката може да бъде планирана след нишките с най-висок приоритет и преди тези с нормален приоритет. Това означава, че нишки с приоритет AboveNormal могат да бъдат планирани след нишката с най-висок приоритет и преди нишки с нормален, под нормален и най-нисък приоритет.

Най-висок = 4: Нишката може да бъде планирана преди нишки с друг приоритет. Това означава, че нишки с най-висок приоритет могат да бъдат планирани преди нишки с друг приоритет.

По подразбиране, когато създаваме нишка, тя получава приоритет по подразбиране 2, т.е. ThreadPriority.Normal

Пример, за да разберем как да зададем и получим приоритетите на нишка като използваме свойството Priority на класа Thread:

```
using System;
using System.Threading;
namespace ThreadStateDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread thread1 = new Thread(SomeMethod)
            {
                Name = "Thread 1"
            };
```

```
//Setting the thread Priority as Normal
        thread1.Priority = ThreadPriority.Normal;

        Thread thread2 = new Thread(SomeMethod)
        {
            Name = "Thread 2"
        };
        //Setting the thread Priority as Lowest
        thread2.Priority = ThreadPriority.Lowest;
 Thread thread3 = new Thread(SomeMethod)
        {
            Name = "Thread 3"
        };
        //Setting the thread Priority as Highest
        thread3.Priority = ThreadPriority.Highest;

        //Getting the thread Prioroty
        Console.WriteLine($"Thread 1 Priority: {thread1.Priority}");
        Console.WriteLine($"Thread 2 Priority: {thread2.Priority}");
        Console.WriteLine($"Thread 3 Priority: {thread3.Priority}");

        thread1.Start();
        thread2.Start();
        thread3.Start();
```

```
            Console.ReadKey();
        }
 public static void SomeMethod()
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine($"Thread Name: {Thread.CurrentThread.Name} Printing {i}");
            }
        }
    }
}
```

Забележка: Резултатът е непредвидим, тъй като нишките са силно зависими от системата. Планировчикът на нишки на операционната система планира нишки без гаранция, но се опитва да ги вземе предвид. За дълго изпълняващи се задачи се възползвайте от настройката на приоритет.
По принцип това не се изисква в обичайните случаи. В някои случаи обаче може да искате да повишите приоритета на някои нишки. Един такъв пример може да бъде, когато искате определени задачи да бъдат изпълнени първи пред други.

Това което е важно:

- Програмистът може изрично да присвои приоритет на нишка.
- Стойността по подразбиране е ThreadPriority.Normal
- Операционната система не дава приоритет на нишките.
- Той ще хвърли ThreadStateException, ако нишката е достигнала крайно състояние, като например Aborted.
- Той ще хвърли ArgumentException, ако стойността, посочена за операция за набор,не е валидна стойност на ThreadPriority.
- Не е гарантирано, че нишката с висок приоритет ще се изпълни първа, а нишката с нисък приоритет ще се изпълни след това. Поради превключване на контекст нишката с най-висок приоритет може да се изпълни след нишката с най-нисък приоритет.

Име на нишка

Всяка нишка има свойство Name. Можеме да зададеме име на нишка само веднъж. Статичното свойство Thread.CurrentThread ни дава текущо изпълняваната нишка. В следния пример задаваме името на основната нишка:

```
class ThreadNaming
{
  static void Main()
  {
    Thread.CurrentThread.Name = "main";
    Thread worker = new Thread (Go);
    worker.Name = "worker";
    worker.Start();
    Go();
  }

  static void Go()
  {
    Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
  }
}
```

Типове нишки в C#

В C# има два типа нишки.
1. Нишка на преден план
2. Фонова нишка

Освен основната нишка на приложението, всички нишки, създадени чрез извикване на конструктор на клас Thread, са нишки на преден план. Фоновите нишки се създават и използват от ThreadPool, който е набор от работни нишки, поддържани от средата за изпълнение.

Нишка на преден план

Предните нишки са онези нишки, които продължават да работят, за да завършат работата си, дори ако основната нишка се откаже. С прости думи работната нишка ще продължи да работи (за да завърши работата), дори ако основната нишка е приключила сесията. Работната нишка може да бъде жива без основна нишка.

Пример за нишка на предн план:

```
class Program
   {
       static void Main(string[] args)
       {
           Thread objThread = new Thread(WorkerThread);
           objThread.Start();
           Console.WriteLine("Main Thread Quits..!");
       }
```

```
static void WorkerThread()
        {
            for (int i = 0; i <= 4; i++)
            {
                Console.WriteLine("Worker Thread is in progress..!");
                Thread.Sleep(2000); //Sleep for 2 seconds
            }

            Console.WriteLine("Worker Thread Quits..!");
        }
    }
```

В горния пример за нишка на преден план, "workerthread" все още е в процес, дори ако основната нишка е приключила сесията.

Фонова нишка

Фоновите нишки са онези нишки, които се затварят, ако основният метод на приложение се затвори. Тук продължителността на живота на работната нишка зависи от основната нишка. Работната нишка се затваря, ако нишката на основното приложение се затвори. Фоновите нишки са идентични с нишките на преден план с едно изключение: Нишката на фона не поддържа процес, работещ, ако всички нишки на преден план са прекратени. След като всички нишки на преден план бъдат спрени, времето за изпълнение спира и всички нишки на фона и се изключват. За да използваме фоновата нишка в приложение, трябва да зададем свойство, наречено "IsBackground", на true.

Пример за фонова нишка:

```
class Program
  {

    static void Main(string[] args)
    {
        Thread objThread = new Thread(WorkerThread);
        objThread.Start();
        objThread.IsBackground = true;

        Console.WriteLine("Main Thread Quits..!");
    }

    static void WorkerThread()
    {
        for (int i = 0; i <= 4; i++)
        {
            Console.WriteLine("Worker Thread is in progress..!");
            Thread.Sleep(2000); //Sleep for 2 seconds
        }

        Console.WriteLine("Worker Thread Quits..!");
    }
}
```

Въпроси?