الخيوط أو المسارات (Threads) وتعدد المسارات (Multithreading) :

نقصد بتعدد المسارات إمكانية تنفيذ أكثر من شفرة في نفس الوقت, يطلق على كل شفرة من هذه الشفرات خيط أو مسار (**Thread**). تصور أن لديك عملتين منفصلتين(لا تعتمد أي منهما على الأخرى) مثلا : إدخال اسم موظف وتحديد أعلى الموظفين أجرا, من المفيد جدا أن يستغل الوقت المستنفذ في انتظار إدخال اسم موظف, في تنفيذ العملية الأخرى الخاصة بتحديد أعلى الموظفين أجرا. إن تنفيذ عمليتين معا يعطي شعورا لدى المستخدم بالتوازي (وان كان التوازي ظاهريا في حالة وجود معالج وحيد) ويوفر الكثير من وقت المعالج.

متى نلجأ إلى استخدام الخيوط ؟

عندما تتعامل الشفرات المراد تنفيذها مع موارد مختلفة مثلا: تقوم الأولى بحساب الأعداد الأولية من1 إلى 100 بينما تقوم الثانية بالتفاعل مع المستخدم (انتظار دخل أو تقديم خرج) أو مثلا تقوم الأولى برسم مثلث بينما تقوم الثانية بإرسال بريد الكتروني و الخ. من غير المفيد أبدا استخدام الخيوط أو المسارات إذا كانت الشفرتان تؤديان عملا حسابيا كأن تقوم الأولى بحساب مربعات عشرة أعداد وتقوم الثانية بحساب جذور15 عددا مثلا, والسبب هو أن تنفيذ هاتين الشفرتين يحتاج إلى المعالج لإتمام عملهما (نفس المورد) ومن ثم فإن إتمام عمليهما لن يعطي أي أرباح, سواء كان قمت به على التوالي (تنفيذ الأولى ثم الثانية) أو قمت به على التوازي(باستخدام الخيوط).

من المفيد أن نعرف أن هناك صنفا (**Class**) مخصصا للتعامل الخيوط في بيئة الNet. يطلق عليه **Thread** يحتوي على العديد من المناهج والخاصائص كما سيتضح لاحقا.

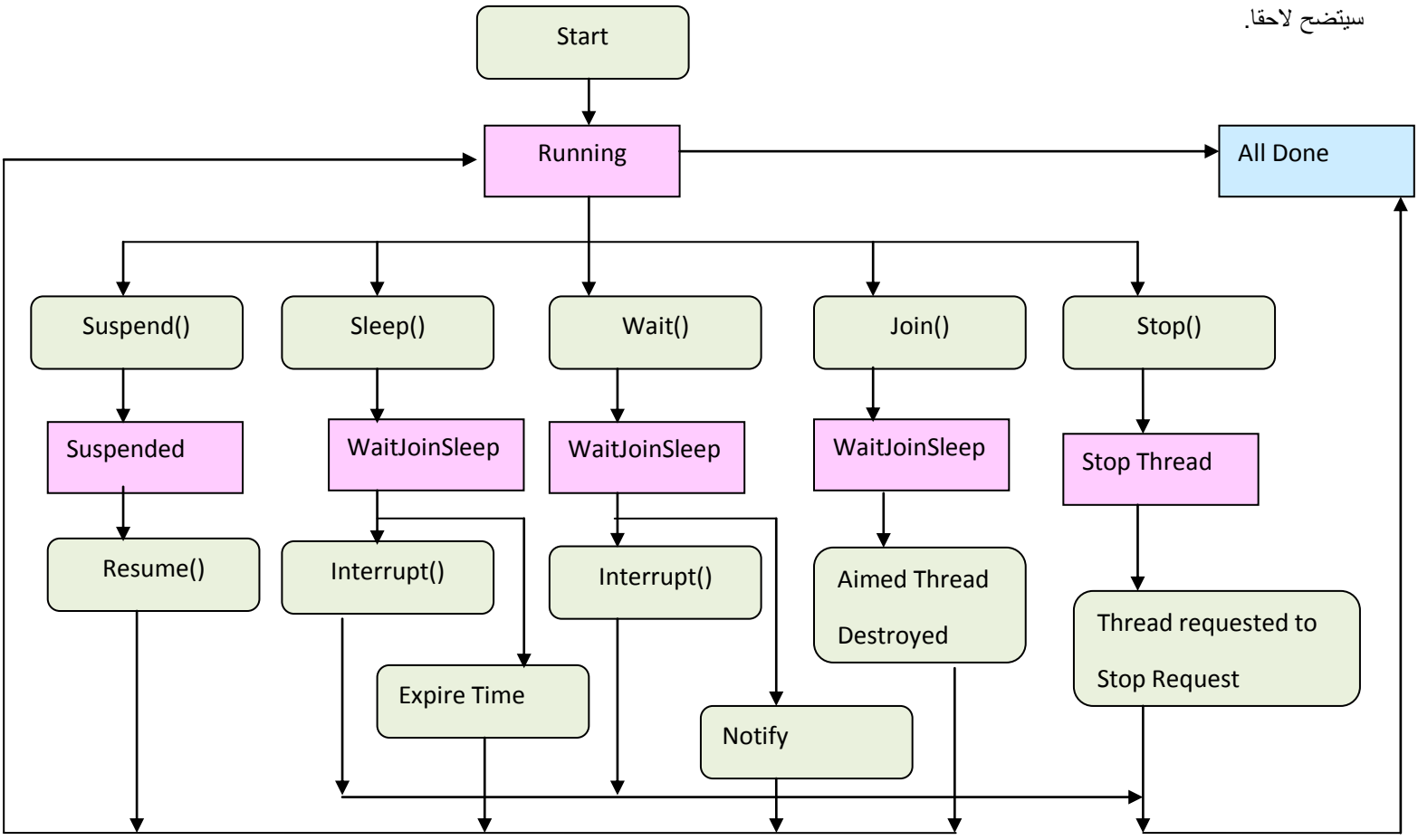

Figure1 Thread Life Cycle

هذه قائمة بأهم المناهج الموجودة في الصنف Thread

المنهج	الوصف
Start	يبدأ عمل الخيط ويستخدم لبداية عمل الخيط
Sleep	يوقف عمل الخيط الزمن محدد بالبارامتر الذي يرسل إليه
Join	يجعل الخيط الذي استدعينا من اجله المنهج يعمل في حين يتوقف الخيط الذي ذكرت الشفرة حتى ينتهي الأول من شفرته.
Interrupt	يجعل الخيط يخرج من حالته إلى حالة العمل إلا إذا كان الخيط في حالة انتظار
Suspend	يجعل الخيط يخرج من حالة الحركة إلى حالة الجمود (يتوقف تنفيذ أوامر الخيط حتى إشعار آخر)
Resume	يخرج الخيط من حالة الجمود إلى حالة الحركة
abort	يوقف عمل الخيط الذي يستدعي هذا المنهج

يوجد كائن الصنف **Thread** أو الخيط في عدة حالات، تخزن حالة الخيط في الحقل **ThreadState** وتحدث كل منها بسبب منهج معين:

1. يكون الخيط قبل استدعاء المنهج **Start** في حالة **Unstarted**
2. بعد استدعاء المنهج **Start** يصبح في حالة **Running**
3. إذا استدعينا احد المناهج (**Wait**,**Sleep**,**Join**) يصبح خيط التنفيذ في **WaitJoinSleep** أي يصبح في حالة انتظار حدث ما
4. إذا استدعينا المنهج **Interrupt** إما يتوقف الخيط إذا كان في حالة **WaitJoinSleep** أو يصبح في حالة **Running** فيما عدا ذلك غير ذلك .
5. إذا استدعينا المنهج **Suspend** وكان الخيط في حالة **Runnig** فإن حالة الخيط ستتحول حالة **SuspendRequested** ثم سيتحول إلى حالة **Suspended**.
6. إذا استدعينا المنهج **Resume** وكان الخيط في حالة **Suspended** فإن حالة الخيط ستتحول إلى **Suspended**
7. إذا استدعينا المنهج **Abort** فإن المنهج سيحول حالة الخيط إلى **AbortRequested** ثم إلى **Stopped**.

المنهج **Wait** يستدعى من قبل كائنات التزامن (**Synchronization Strategies**) مثل المسيرات والمنظمات والمراقبات (**mutex**, **semaphore**,**Monitors**) وتجعل الخيط في حالة **WaitJoinSleep.**

لا حقا سنرى بامثلة كيف سنستفيد من هذه المناهج.

تنفيذ الخيوط في لغة ال#C:

1 - من النافذة **File** اختر **New** ثم **Project** ومن النافذة **New Project** اختر العنصر **Console Application** (**Figure** 1)

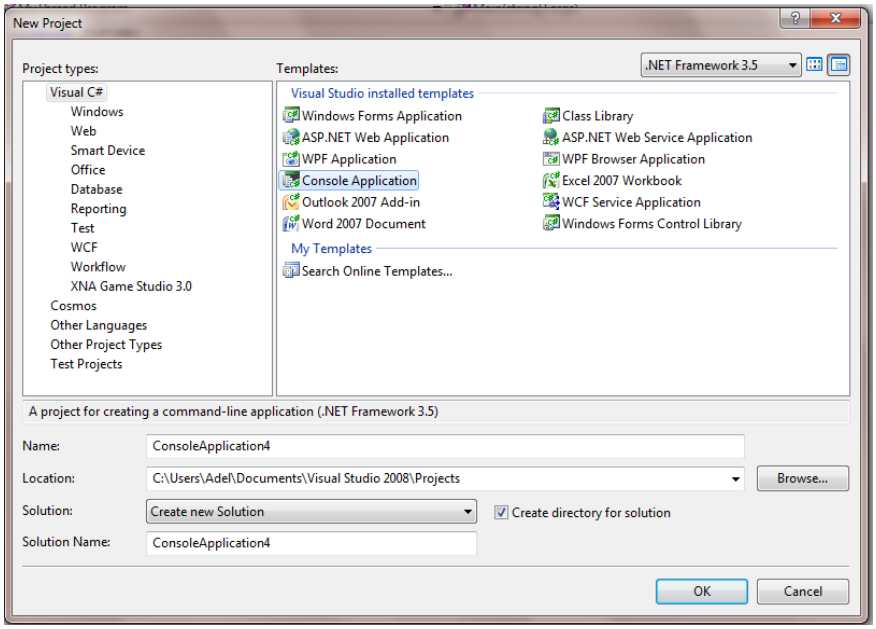

Figure2

في الحقل **Name** أختر اسم للمشروع و

وليكن **MyThread** ستجد أن هناك نافذة تكونت كماهو مبين في **Figure** 2.

2 - أضف السطر `using System.Threading;`

كما هو موضح اسفله:

لاحظ التعديلات على الجزء الخاص بالمكتبات:

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

لا نحتاج حاليا لأكثر من هذين السطرين لذلك، يمكنك مسح الباقي إذا شئت.

هذه شفرة ابسط برنامج :

```
using System;
using System.Threading;


namespace MyThread
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

عدل الشفرة في البرنامج السابق كي تصبح كما يلي :

```
namespace MyThread
{
    class Program
    {
        public static void SimpleMethod()
        {
            int i = 5;
            int x = 10;
```

```
            int result = i * x;
            for(i=1;i<=20;i++)
                Console.WriteLine("New Thread " + i.ToString());
        }

        static void Main(string[] args)
        {
            ThreadStart ts = new ThreadStart(SimpleMethod);
            Thread t = new Thread(ts);
            t.Start();
            for (int i = 1; i <= 20; i++)
              Console.WriteLine("Main Thread "+i.ToString());
            Console.ReadLine();
        }
    }
}
```

في السطر الأول من المنهج **Main** نرى الشفرة :

```
ThreadStart ts = new ThreadStart(SimpleMethod);
```

في هذه الشفرة نرى الكائن **ts** من الصنف **ThreadStart**, يمثل هذا الكائن نقطة البداية لأي خيط (**Thraed**)سيرتبط به، بكلمات أخرى أننا هنا نعد الشفرة الموجودة في المنهج **SimpleMethod** لكي تصبح الشفرة التي ينفذها الخيط لذي سيرتبط بالكائن **ts**

والآن سنقوم بتعريف خيط وسنقوم بربطه بنقطة بداية

```
Thread t = new Thread(ts);
```

لكي يقوم الخيط ببدء عمله يجب نلجأ إلى المنهج Start كما هو مبين في الشفرة تذكر هذه النقطة دائما.

```
t.Start();
```

شغل البرنامج وراقب النتائج, قم بتشغيل البرنامج أكثر من مرة, ستلاحظ تغير النتائج كل مرة.

ملحوظة : إذا لم يكن هناك أي إعلان عن خيط (**Thread**) في البرنامج فهذا يعني أن البرنامج يحتوي على خيط وحيد (وان كان الأمر خفيا) مربوط بالشفرة الواقعة داخل المنهج **Main** يطلق عليه الخيط الرئيس (**Main Thread**)وهذا يفسر تنقل النتائج بين شفرة المنهج **Main** و شفرة المنهج **SimpleMethod** المرتبطة بالخيط **t** . والخلاصة انه لا يوجد برنامج لا يحتوي على أي خيط .في المثال التالي سنرى شفرة تحتوي على أكثر من خيط (إضافة إلى الخيط الرئيس المرتبط بالمنهج Main) وسنرى كيف يقوم البرنامج باختيار احد الخيوط ليتوقف فجأة ويقوم باختيار خيط آخر للعمل وهكذا .

```
using System;
using System.Threading;

class Program
    {
        //===============================================
        public static void FirstMethod()
        {
            int i ;
            for(i=1;i<=20;i++)
                Console.WriteLine("First Thread " + i.ToString());
        }
        //===============================================
```

```
    public static void SecondMethod()
      {
          int i;
          for (i = 1; i <= 20; i++)
              Console.WriteLine("Second Thread " + i.ToString());
      }
      //=================================================
      static void Main(string[] args)
      {
          ThreadStart FirstStart = new ThreadStart(FirstMethod);
          Thread FirstThread = new Thread(FirstStart);
          //*******************************
          ThreadStart SecondStart = new ThreadStart(SecondMethod);
          Thread SecondThread = new Thread(SecondStart);
          //*******************************
          FirstThread.Start();
          SecondThread.Start();
          //*******************************
          for (int i = 1; i <= 20; i++)
            Console.WriteLine("Main Thread "+i.ToString());
          Console.ReadLine();
      }
  }
```

قمنا بإنشاء نقطتي بداية **FirstStart** و **SecondStart** و ربطناهما بالمنهجين **FirstMethod** و **SecondMethod**، بعد ذلك قمنا بربط نقطتي البداية بالخيطين **FirstThread** و **SecondThread** على الترتيب، ثم جعلنا الخيطين يباشران عملهما باستدعاء المنهج **Start** كما هو موضح اعلاه. قم بتشغيل البرنامج عدة مرات ولاحظ اختلاف النتائج كل مرة (لماذا؟) لان المجدول (**scheduler**) هو من يختار الخيط الذي يكون نشيطا بعد مدة معينة (المجدول يقرر المدة) و يكون التأخير واضحا إذا كان وقت إطلاق الخيوط متقاربا كما هو في حالتنا هذه.

وضع الخيط في حالة Sleep

لم يعد سرا أننا نجعل الخيط يبدأ عمله بعد إنشاءه بواسطة الأمر **Start** ولكن إذا أردنا أن نجعل الخيط يتوقف عن عمله لفترة وجيزة فإننا نستخدم المنهج **Sleep** الذي يجعل الخيط يتوقف لفترة وجيزة تقدر بالمللي ثانية يحددها البارامتر الذي يتلقاه المنهج انظر المثال التالي:

```
using System;
using System.Threading;
public class ThreadSleep
{
    public static Thread worker1;
    public static Thread worker2;

    public static void Main()
    {
        Console.WriteLine("\nEntering the void Main!");
        worker1 = new Thread(new ThreadStart(Counter1));
        worker2 = new Thread(new ThreadStart(Counter2));
        worker1.Start();
        worker2.Start();
        Console.WriteLine("\nExiting the void Main!");
    }
```

```
    public static void Counter1()
     {
         Console.WriteLine("\nEntering Counter1");
         for (int i = 1; i < 50; i++)
         {
             Console.Write(i + " ");
             if (i == 10)
                 Thread.Sleep(1000);
         }
         Console.WriteLine("\nExiting Counter1");
     }
     public static void Counter2()
     {
         Console.WriteLine("\nEntering Counter2");
         for (int i = 51; i < 100; i++)
         {
             Console.Write(i + " ");
             if (i == 70)
                 Thread.Sleep(5000);
         }
         Console.WriteLine("\nExiting Counter2");
     }
}
```

قم بتشغيل البرنامج السابق ولاحظ النتائج. لابد أن لاحظت تداخل القيم المطبوعة والناتج عن التنقل بين الخيوط المختلفة فتارة القيم الناتجة عن تنفيذ الخيط الأول وتارة القيم الناتجة عن تنفيذ الخيط الرئيس و تارة عن تنفيذ الخيط الثاني وقد تتبادل الخيوط الأدوار. لاحظ انك ستشعر ببعض التأخير هذه المرة هذا بسبب وجود السطر `Thread.Sleep(5000)` لاحظ أيضا استدعاء المنهج **Sleep** بواسطة اسم الصنف **Thread** مما يدل على التوصيف **static** لهذا المنهج . إن وجود هذا السطر في أي شفرة يدل على إيقاف تنفيذ تلك الشفرة مدة من الزمن تساوي القيمة تلقاها المنهج**Sleep** مقدرة بالمللي ثانية ففي مثالنا هذا سيتم تأخير الخيط الأول مقدار ثانية واحدة أو 1000 مللي ثانية سيتوقف تنفيذ الخيط الثاني مقدار 5000 مللي ثانية .

سؤال بما أننا نزعم هناك خيط رئيس موجود دائما ومرتبط بشكل تلقائي بالمنهج **Main** هل سيقوم المنهج **Sleep** بجعل الخيط الرئيس يدخل في حالة Sleep؟

نفذ الشفرة التالية لترى الجواب على السؤال المطروح.

```
using System;
using System.Threading;
public class ThreadSleep
{
    public static void Main()
    {
        for (int i = 1; i < 50; i++)
        {
            Console.Write(i.ToString());
            if (i % 10 == 0)
                Thread.Sleep(2000);
         }
    }
}
```

لاحظ التأخير الذي طرأ على سلوك البرنامج مع أننا لم ننشئ أي خيط، مما يدل علي وجود خيط رئيس مرتبط بالمنهج **Main** ومنشأ بشكل إفتراضي من قبل البرنامج . الجدير الذكر أن الخيط سيدخل إلى حالة **WaitSleepJoin** بعد استدعاء المنهج **Sleep** والطريقة الوحيدة لإخراج الخيط من هذه الحالة (دون انتظار نهاية المدة المحددة بالبارامتر المرسل إلى المنهجSleep) هي باستدعاء المنهج **Interrupt**

```
using System;
using System.Threading;
public class Interrupt
{
    public static Thread sleeper;
    public static Thread worker;
    //*************************************************************
    public static void Main()
    {
        Console.WriteLine("Entering the void Main!");
        sleeper = new Thread(new ThreadStart(SleepingThread));
        worker = new Thread(new ThreadStart(AwakeTheThread));
        sleeper.Start();
        worker.Start();
        Console.WriteLine("Exiting the void Main!");
    }
    //*************************************************************
    public static void SleepingThread()
    {
        for (int i = 1; i < 50; i++)
        {
            try
            {
            Console.Write(i + " ");
            if (i == 10 || i == 20 || i == 30)
            {
                Console.WriteLine("Going to sleep at: " + i);
                Thread.Sleep(20);
            }
            }

           catch (ThreadInterruptedException Ex)
            {
                Console.WriteLine("Bad Way To Interrupt");
                Console.WriteLine("Sleeper Will Die ");
            }
        }
    }
    //*************************************************************
    public static void AwakeTheThread()
    {
        for (int i = 51; i < 100; i++)
        {
            Console.Write(i + " ");
            if (sleeper.ThreadState ==
               System.Threading.ThreadState.WaitSleepJoin)
            {
                Console.WriteLine("Interrupting the sleeping thread");
                sleeper.Interrupt();
            }
        }
    }
    //*************************************************************
}
```

عند استدعاء المنهج **Interrupt من اجل خيط واقع في حالة WaitSleepJoin** يحدث خطأ ويتوقف الخيط عن العمل نهائيا بعدها؛السبب ان المنهج Sleep يجعل الخيط خارج عملية الجدولة (**The thread will not be scheduled**) وبالتالي لا يمكن عمليا مقاطعة خيط هو واقع خارج العمل اصلا , لتجاوز هذا الخطأ لدينا الكتلة **catch** . على كل حال إذا لم يكن هناك ما يدعوك إلى استدعاء المنهجInterrupt فلا تفعل ذلك.

لاحظ في الشفرة السابقة أن الإجراء **AwakeTheThread** يقوم بفحص حالة الخيط قبل أن يخرجه من حالة **WaitSleepJoin**.لا بد من فحص حالة الخيط قبل استدعاء المناهج المذكورة سابقا. لكل منهج حالة معينة يمكن استدعاءه عندها ، خذ هذا الامر نصب عينيك دائما والا ستحصل على رسائل الخطأ المزعجة.

وضع الخيط في حالة جمود وإخراجه منه

لإدخال الخيوط إلى حالة جمود (**Suspended**) نحتاج إلى المنهج **Suspend** ونخرجه من هذه الحالة بواسطة المنهج **Resume** الذي يعيده إلى الحالة **Running**

```
class Program
{
 //***********************************************
    private static Thread First;
    private static Thread Second;
    //***********************************************
    static void Main(string[] args)
    {
        First = new Thread(new ThreadStart(FirstFun));
        Second = new Thread(new ThreadStart(SecondFun));
        Second.Start();
        First.Start();
    }
    //***********************************************
    public static void FirstFun()
    {
        int i=0;
        for (i = 0; i < 100; i++)
            Console.WriteLine("First is working "+i.ToString());

    }
    //***********************************************
    public static void SecondFun()
    {
        int i = 0;
        for (i = 0; i < 100; i++)
        {
            Console.WriteLine("Second is working " + i.ToString());
            if (i == 10 && First.ThreadState == ThreadState.Running)
                First.Suspend();
            if (i==80&&First.ThreadState == ThreadState.Suspended)
                First.Resume();
        }

    }
    //***********************************************
    private static Thread First;
    private static Thread Second;
    //***********************************************

}
```

تبدو شفرة المنهج **Main** مألوفة لدينا، حيث قمنا بتعريف كائنين من الصنف **Thread** من أجل الخيوط ، وربطنا كل واحد منهما بدالة (لم نعد نعرف متغيرات لنقط بداية بل أرسلناها مباشرة إلى باني الصنف **Thread**). يقوم إجراء الخيط الأول بطباعة سطر في حين يقوم إجراء الخيط الثاني بطباعة سطر ثم عند قيمة معينة يجعل الخيط الأول في حالة جمود وبعد ان يصل **i** إلى القيمة **80** يقوم الخيط الثاني بتحرير الأول . بواسطة المنهج **Resume**.

المنهج Join

يقوم المنهج يايقاف عمل الخيط الذي استدعي داخل المنهج شفرته، ويجعل الخيط الذي استدعي من اجله (وليس داخل شفرته) يعمل. فقط بعد انتهاء الخيط الذي استدعي من اجله من شفرته. يقوم الخيط الذي استدعي المنهج داخل شفرته يعمل. كيف نوضح ما قصدناه تأمل الشفرة التالية :

```
public class JoiningThread
        {
            //*******************************************************
            public static Thread SecondThread;
            public static Thread FirstThread;
            //*******************************************************
            static void First()
            {

                for (int i = 1; i <= 250; i++)
                    Console.WriteLine("First  " + i + " ");
            }

            static void Second()
            {
                FirstThread.Join();
                for (int i = 1; i <= 250; i++)
                    Console.WriteLine("Second " + i + " ");
            }
            public static void Main()
            {
                FirstThread = new Thread(new ThreadStart(First));
                SecondThread = new Thread(new ThreadStart(Second));

                FirstThread.Start();
                SecondThread.Start();
            }

        }
```

سترى ان المنهج **Join** تم استدعاؤه من داخل الشفرة التي تخص الخيط **SecondThread** (داخل الدالة **Second**)؛ اذا سيتوقف الخيط **SecondThread** عن العمل ,حتى يتمكن **FirstThread** (الخيط الذي استدعي من أجله المنهج) من إنهاء شفرته. يكون هذا المنهج مفيد جدا إذا كانت نتائج عمل خيط ما، ضرورية لعمل خيط آخر. كأننا نريد أن نقول باستدعائنا للمنهج **Join** أن الخيط الحالي يعتمد في عمله على الخيط صاحب المنهج **Join** لذلك أوقف الخيط الحالي حتى يكمل صاحب المنهج Join عمله، ثم أبدأ العمل.

إرسال بارامترات إلى الدالة المرتبطة بالخيط:

لكي ترسل بارامترات إلى الدالة المرتبطة نستخدم الصنف **ParameterizedThreadStart** بدلا من ال **ThreadStart** **أثناء ربط الخيط الدالة انظر إلى الشفرة التالية:**

```
static void Main(string[] args)
        {
            int a=10;
            Thread MyThread = new Thread(new ParameterizedThreadStart(ThreadFun));
            MyThread.Start(a);
        }

        static void ThreadFun(object myParam)
        {
            int a = (int)myParam;
           Console.Write(a.ToString()+" is Sended Value ");
        }
```

لاحظ الأمور التالية:

1. كيفية إرسال البارامترات: يتم إرسال البارامترات عند استدعاء المنهج **Start**.
2. نوعية البارامترات : لاحظ أن البارامتر الذي تستقبله الدالة ThreadFun هو من النوع object وهذا من حسن الحظ (تشتق جميع الأنواع في لغة C# من الصنف object) لأنه يمكننا من إرسال أي نوع البارامترات . انظر في الشفرة التالية كيف سنرسل بارامتر من النوع string :

```
static void Main(string[] args)
        {
            string a = "C# is The Best Languae";
            Thread MyThread = new Thread(new ParameterizedThreadStart(ThreadFun));
            MyThread.Start(a);
        }
        static void ThreadFun(object myParam)
        {
          string str = (string)myParam;
          Console.WriteLine(str);
        }
```

3. عدد البارامترات : إذا كنت تريد إرسال أكثر من بارامتر يمكنك استخدام مصفوفة أو structure أو حتى class انظر المثال التالي لتعرف كيف يمكن إرسال بارامترين من النوع int

```
static void Main(string[] args)
        {
            int i = 4, j = 12;
            Thread MyThread = new Thread(new ParameterizedThreadStart(ThreadFun));
            MyThread.Start(new int[] { i, j });
        }
        static void ThreadFun(object myParam)
        {
            int[] a = (int[])myParam;
            //*************************************
            for (int i = 0; i < a.Length; i++)
                Console.Write(a[i].ToString()+" ");
            Console.WriteLine();
        }
    }
```

اذا اختلف نوعية البارمترات يمكن استخدام مصفوفة من النوعobject والمثال التالي يوضح كيف نعمل ذلك:

```
static void Main(string[] args)
        {
            int[] a ={1,2,3};
            object[] b = { 10, "name" ,a};
            Thread MyThread = new Thread(new ParameterizedThreadStart(ThreadFun));
            MyThread.Start(b);
        }
        static void ThreadFun(object myParam)
        {
            object[] a = (object[])myParam;
            int Param1 = (int)a[0];
            string Param2 = (string)a[1];
            int[] array = (int[])a[2];
            //*************************************
            Console.WriteLine("First Parameter is " + Param1.ToString());
            Console.WriteLine("Second Parameter is " + Param2);
            Console.Write("Third Parameter is ");
            for (int i = 0; i < array.Length; i++)
                Console.Write(array[i].ToString()+" ");
            Console.WriteLine();
        }
```

لاحظ ان الاستقبال كان على هيئة مصفوفة من النوعobject وبعد ذلك حولنا كل عنصر من عناصر المصفوفة إلى النوع المناسب حسب الارسال.

المنهج **Reaborted**

يقوم المنهج **Reaborted** بإعادة إحياء خيط تم إيقافه بالمنهج Abort

```
class Program
    {
        static Thread FirstThread;
        static Thread SecondThread;
        static void Main(string[] args)
        {
            FirstThread = new Thread(new ThreadStart(FirstFun));
            SecondThread = new Thread(new ThreadStart(SecondFun));
            //*************************************
            SecondThread.Start();
            FirstThread.Start();
        }
        static void FirstFun()
        {
            int i = 0;
            bool flag = true;

            do{
                try
               {
                for (i = 1; i < 2000; i++)
                {
                   Console.WriteLine("First is working "+i.ToString());
                }
```

```
                Console.WriteLine("First Ended ++++++++++++++++++++++++++++++ \a\a\a");
                flag = false;
                }

  catch (Exception ex)
                {
                    Console.WriteLine("Bad way to Abort ****************************");
                    //Thread.ResetAbort();
                    Console.WriteLine("I hope I can Live Pack ************************");
                }
        }while(flag);

        }
        /************************************************/
        static void SecondFun()
        {
            int i = 1;
            for (i = 1; i < 2000; i++)
            {
                Console.WriteLine("Second is working " + i.ToString());
                if (i == 700)
                {
                    FirstThread.Abort();
                    Console.WriteLine("First Aborted \a");
                    Thread.Sleep(100);
                }
            }
        }
    }
```

في الشفرة السابقة انظر إلى الجزء الخاص بالدالة **Main** لايوجد فيها شيء جديد , مجرد إعلان عن خيطين وربطهما بدالتين ومن ثم بدء عمل الخيطين.الآن انظر إلى الدالة **SecondFun** وهي الدالة المرتبطة بال**SecondThread** ، سترى ان الدالة تقوم بكتابة سطر معين حتى تصبح قيمة المتغير i مساوية ل 700 حينها تقوم الدالة بقطع عمل الخيط **FirstThread** عن طريق إستدعاء المنهج **Abort** من اجل الخيط **FirstThread** **ومن ثم اصدار صفير بسبب طباعة الرمز '\a'** .الان انظر إلى الشفرة الخاصة بالدالة **FirstFun** وهي الدالة المرتبطة بالخيط **FirstThread** ستجد أنها تنفذ الأمور التالية:

1. تدخل إلى الكتلة **try** وبعدها إلى الحلقة **for** التي تطبع سطر معينا عند كل دورة
2. إذا انتهت الحلقة بشكل طبيعي يتم طباعة العبارة **First Ended** ويصدر البرنامج صفير 3 مرات بسبب طباعة الرمز \a 3 مرات وتجعل المتغير **flag** يأخذ القيمة **flase** يضمن هذا الإجراء عدم العودة إلى بداية الحلقة**do** التي تبدأ عملها في بداية البرنامج
3. إذا لم تنتهي الحلقة بشكل سليم معناها التوقف كان بسبب المنهج **Abort** الذي استدعيناه في شفرة الخيط الثاني، مما يؤدي إلى تنفيذ الجزء الموجود داخل الكتلة **catch** في شفرة الخيط الأول.

لاحظ ان السطر **Thread.ResetAbort();** محذوف من البرنامج (على شكل تعليق). إذا كان هذا الجزء محذوفا فهذا يعني ان الخيط سينهي عمله مكتفيا بالشفرة داخل **catch**. إذا أزلنا علامة التعليق سيصبح هذا السطر نشيطا لذا سيقوم الخيط بتنفيذ الشفرة التي تقع داخل الكتلة **catch** و **أي شفرة تقع بعدها**، وفي حالتنا هذه هناك أمر واحد يقع بعد الكتلة **catch** وهو شرط اختبار قيمة المتغير **flag** الموجود في آخر الحلقة **do ,** فإذا تحقق الشرط عاد من البداية و إلا فإن البرنامج سينهي عمله. جرب البرنامج عدة مرات بعد تفعيل السطر و ستلاحظ أن البرنامج اصدر صفيرا 4 مرات. لترى الفرق قم بتعديل شفرة الخيط الاول بهذا الشكل:

```
……..
 catch (Exception ex)
                {
                    Console.WriteLine("Bad way to Abort ****************************");
                    Thread.ResetAbort();
                    Console.WriteLine("I hope I can Live Pack ************************");
                }
   if(!flag)

  Console.WriteLine("11111111111111111111111111111111111111111111111111");
  ……..……..
```

تدل النقط على أن بقية الشفرة قبل وبعد هذا الجزء من شفرة الدالة المرتبطة بالخيط الأول لن تتغير(الحلقة ستظلdo على حالها) . قم بتشغيل البرنامج مرة أخرى وافحص النتائج.

قم بإلغاء السطر **`Thread.ResetAbort();`** ثم شغل البرنامج. أعده للعمل وراقب النتائج.نلاحظ ماياتي:

1. عندما نسمع صفيرا وحيد فإن السطر 11111111111111 لن يظهر وهذا دليل على أن البرنامج اكتفى بالشفرة داخل الكتلة**catch** وانتهى عمل الخيط (رغم أن **flag=true**)
2. عندما نسمع صفيرا اربع مرات فالشفرة انتهت بشكل سليم وستظهر السلسلة 111111 معناهاتم تنفيذ ماداخل الكتلة **catch** وما بعدها وستلاحظ أن الشفرة واصلت عمل الحلقة **do** وانتهى الأمر بشكل طبيعي وادى هذا إلى سماعنا صفيرا أكثر من مرة.

أهم الحقول الموجودة في الصنف Thread

المنهج	الصنف
public static CurrentThread	كما هو اضح أن هذه الخاصية تعيد كائنا يشير إلى الخيط الحالي (يمكن التعامل معها باسم الصنف لماذا ؟) وهو الخيط الذي ينفذ شفرة معينة في اللحظة الراهنة.
public bool IsAlive	تعطي هذه الخاصية القيمة true إذا كان الخيط يعمل، و false إذا أنهى العمل أو أوقفه خيط آخر (بواسطة المنهج Abort)
public ThreadState ThreadState	تعطينا هذه الخاصية حالة الخيط ، هل الخيط يعمل الآن , هل هو متوقف الخ. تعي لنا أحد قيم التعداد ThreadState
public string Name	تحدد هذه الخاصية اسم الخيط، توجد إمكانية منح اسم لكائنات الخيط
public ThreadPriority Priority	تحدد هذه الخاصية الأولوية التي يعطيها المجدول للخيوط
public bool IsBackground	تحدد إذا كان الخيط من خيوط الخلفية أم هو غير ذلك.

يوضح المثال التالي عمل الخاصية **Name** كما يوضح عمل الخاصية **CurrentThread**:

```
static void Main()
    {
        Thread First = new Thread(new ThreadStart(ThreadFun));
        First.Name = "I'm first";
        Thread Second = new Thread(new ThreadStart(ThreadFun));
        Second.Name = "I'm Second";
        //*********************************
        First.Start();
        Second.Start();
    }
```

```
public static void ThreadFun()
{
    string ThreadName = Thread.CurrentThread.Name;
    Console.WriteLine(ThreadName);
}
```

لاحظ أن الخيطين مرتبطان بدالة واحدة هي **ThreadFun** ومع ذلك فإن الخاصية **CurrentThread** تشير إلى الخيط الحالي لحظة اللجوء إليها . **الخيط الحالي هو الخيط الذي ينفذ الشفرة في اللحظة الراهنة**. مثلا في لحظة معينة كان الخيط **First** هو الخيط الحالي لذلك عندما لجأنا إلى الخاصية **Name** من الخيط الحالي حصلنا على السلسلة المخزنة في الخاصية **Name** من كائن الخيط **First**.

الخاصية IsBackground : أريد أن أوضح انه لا فرق بين خيوط المقدمة والخلفية إلا في شيء واحد، وهو أن البرنامج ينهي عمله إذا أنهت جميع خيوط المقدمة أعمالها ، ولا يهم إذا أنهت خيوط الخلفية عملها ام لا. لذلك يمكن وضع الأعمال التي لا يرى المستخدم عملها في خيوط خلفية ثم استدعاء المنهج **Join** داخل خيط أمامي إذا كان عمل الخيط الأمامي يعتمد على نتائج عمل الخيط الخلفي. يذكر أن القيمة الافتراضية لهذه الخاصية هو **false** يوضح المثال القادم مفهوم الخيط الخلفي والأمامي وكيفية استخدام الخاصة **Name**:

```
static void Main()
{

    Thread Foreground = new Thread(new ParameterizedThreadStart(ThreadFun));
    Thread BackGround = new Thread(new ParameterizedThreadStart(ThreadFun));
    Foreground.Name = "Foreground";
    BackGround.Name = "Background ";
   // BackGround.IsBackground = true;
    Foreground.Start(20);
    BackGround.Start(100);
}


public static void ThreadFun(object Param)
{
    int max = (int)Param;
    string ThreadName = Thread.CurrentThread.Name;
    for (int i = 0; i < max; i++)
    {
        Console.WriteLine("{0} count: {1}", ThreadName, i.ToString());
        Thread.Sleep(250);
    }
    Console.WriteLine("{0} finished counting.", ThreadName);
}
```

لاحظ داخل **Main** أن أننا أنشأنا خيطين وربطناهما بدالة **ThreadFun** و أسندنا إلى الخاصية **Name** اسم معينا، نستطيع أن نصل إلى القيمة المسندة في أي لحظة كما سنفعل لاحقا. تذكر السطر غير المفعل **BackGround.IsBackground = true;** سنعود إليه بعد قليل. لاحظ أننا أرسلنا قيما مختلفا إلى دوال الخيوط، ولاحظ أننا تعمدنا إرسال قيمة كبيرة إلى الخيط **Background**. قم بتشغيل البرنامج ستلاحظ أن الخيطين يتناوبان العمل ثم ينهي الخيط **Foreground** عمله لأنه يعالج قيمة أصغر بينما . والآن قم بتفعيل السطر **BackGround.IsBackground = true;** ثم شغل البرنامج ، ستجد أن الخيطين يتناوبان العمل حتى يصل الخيط إلى نهاية عمله، ستلاحظ أن البرنامج أنهى عمله فجأة مع أن الخيط **BackGround** لم ينه عمله . لاحظ أيضا أن **Name** مفيدة في تحديد هوية الخيط الفعال، خاصة إذا كانت الخيوط مرتبطة بنفس الدالة كما هو الحال في مثالنا هذا.

الخاصية IsAlive تساعدنا في معرفة حالة الخيط واتخاذ قراراتنا تبعا لهذا النتيجة انظر إلى الشفرة التالية

```
class Test
{

    static void Main()
    {
        First = new Thread(new ThreadStart(FirstFun));
        First.Name = "I'm first";
        Second = new Thread(new ThreadStart(SecondFun));
        Second.Name = "I'm Second";
        //*********************************
        //First.Start();
        Second.Start();
    }
    //*********************************
    static Thread First;
    static Thread Second;
    //*********************************
    public static void FirstFun()
    {
        int i=0;
        string ThreadName="";
        for (i = 0; i < 100; i++)
        {
            ThreadName = Thread.CurrentThread.Name;
            Console.WriteLine(ThreadName);
        }
    }
    public static void SecondFun()
    {
        do
        {
            if (!First.IsAlive)
            {
                Console.WriteLine("First Will Start Now, How Great..!");
                First.Start();
            }
            else
            {
                while (First.IsAlive)
                    Console.WriteLine("Second's Working While First is working");
                return;
            }
        } while (true);
    }
```

في الدالة **Main** لا يوجد جديد، صرحنا عن الخيطين سلفا ونقوم بربطهما بدوال داخل الدالة**Main** . الدالة **FirstFun** أيضا لا تحتوي على جديد، شفرة تقوم بكتابة اسم الخيط الحالي 100 مرة. أما الدالة **SecondFun** فتقوم بفحص حالة الخيط **First** فإذا كان الخيط نشيطا (**IsAlive=true**) فإنها تدخل إلى الكتلة else فيما عدا ذلك (**IsAlive=false**) فإنها تقوم الشفرة بتنشيط الخيط بواسطة المنهج **Start** .

الخاصية Priority : نعني بالأولوية أحقية الخيط بالتنفيذ قبل الخيوط الأخرى يوجد لدى كل خيط خاصية تحدد قيمة الأولوية لهذا الخيط أثناء التنفيذ، تخزن هذه القيمة داخل الخاصية Priority. إن عملية الجدولة تعتمد بشكل أساسي على أولوية الخيط . في كثير من نظم التشغيل تنفذ الخيوط ذات قيم الأولوية عليا أولا بينما يعطي نصيب متساويا من الوقت للخيوط ذات الأولوية المتساوية و في حالة ظهور خيوط ذات أولوية أعلى فأنها تجدول وتقصى الخيوط ألأخرى عن العمل ويعتمد هذا الأمر على خوارزمية الجدولة المتبعة من قبل نظام التشغيل. يوضح المثال التالي كيفية استخدام الخاصية**Priority**

```
static void Main()
    {
        First = new Thread(new ThreadStart(ThreadFun));
        First.Name = "I'm first";
        Second = new Thread(new ThreadStart(ThreadFun));
        Second.Name = "I'm Second";
        //*****************************************
        First.Priority = ThreadPriority.Highest;
        Second.Priority = ThreadPriority.Lowest;
        //*****************************************
        First.Start();
        Second.Start();
    }
    //*********************************
    static Thread First;
    static Thread Second;
    //*********************************
    public static void ThreadFun()
    {
        int i=0;
        string ThreadName="";
        for (i = 0; i < 100; i++)
        {
            ThreadName = Thread.CurrentThread.Name;
            Console.WriteLine(ThreadName+"   "+i.ToString());
        }
    }
```

قم بتغيير قيمة الخاصية **Priority** من داخل الدالة Main من أجل كل خيط ثم شغل البرنامج أكثر من مرة وشاهد الفرق .

هناك مواضيع أخرى لم أتطرق اليها مثل Thread Pooling و Threading Traps ولكني ساتركها للقاريء