

## TPL (Task Parallel Library)

TPL paralel programlamayı kolaylaştırmak için Framework 4.0 ile eklenmiş bir kütüphanedir. Paralel programlama bir işin thread'lere ayrılıp farklı CPU ya da çekirdeklere dağıtılarak mümkünse aynı anda yaptırılmasına denilmektedir. Paralel programlama faaliyetinin söz konusu olabilmesi için aynı makinada bir'den fazla CPU ya da çekirdeğin bulunuyor olması gerekir. İşlerin bölünerek ağ içerisindeki farklı bilgisayarlara aynı anda yaptırılmasına paralel programlama denilmemektedir. Buna dağıtık programlama (distributed programming) denilmektedir.

Şüphesiz paralel programlama hiçbir kütüphane kullanılmadan da manuel olarak gerçekleştirilebilir. Bunun için programcı işini thread'lere ayırmalı ve bu thread'leri CPU ya da çekirdeklere atmalıdır (processore affinity). .NET'te de böyle manuel paralel programlama yapılabilir.

Bir paralel programlama kütüphanesi kullanmanın manuel işleme göre ne avantajı olabilir? Kütüphane birtakım işlemleri daha kolay ve daha az çabayla yapmamızı sağlayabilir. Kütüphane ölçeklenebilir (scalable) bir ortam bize sunabilir. (Yani sistemde kaç CPU ya da çekirdek varsa ona uygun çalışabilir.)

Paralel programlama özellikle son yıllarda kendinden çok söz edilen bir konu haline gelmiştir. Başka dillerde çeşitli kütüphaneler zaten uzun süredir kullanılıyordu. Bunların en ünlüleri MPI (Intel'in)ve Open MP (açık kaynak kod)'dir

Paralel programlamayı sentaks olarak destekleyen diller de vardır. Bunlara paralel programlama dilleri de denilmektedir. Fakat genel olarak paralel programlama sentaksla değil kütüphanelerle ve framework'lerle desteklenmektedir.

.NET'te paralel programlamaya ilişkin sınıflar ve türler System.Threading.Tasks isim alanı içerisinde. Bu sınıflar mscorlib.dll'ye yerleştirildiğinden ayrıca bir dll'e referans etmeye gerek yoktur.

### Task Sınıfına Giriş

TPL'nin en önemli sınıflarından biri Task sınıfıdır. Aslında Task sınıfı iki tanedir. Bunlardan biri generic olmayan diğeri generic olandır. Task sınıfı aslında özetle bizden bir delege ister ve o delege metodunu ayrı bir thread'te çalıştırır. Tabi bunu genel olarak TPL kavramı içerisinde ekstra birtakım özellikler sunarak yapar.

Task sınıfının generic olmayan versiyonu geri dönüş değeri void olan metotları kullanır, generic versiyonu geri dönüş değeri herhangi bir türdn olan metotları kullanır. Task sınıfları başlangıç metotlarında bizden ayrı thread'te çalıştırılacak metodu alır. Genel olarak metotların void parametreye ve object parametreye sahip olan versiyonları vardır. Parametrelerde Action delegeesi parametresi olmayan Action<object> ise parametresi object olan metotları tutan delegeyi belirtir. Task sınıfının iki önemli başlangıç metodu şöyledir:

```
public Task(Action action)
public Task(Action<Object> action, Object state)
```

Bir Task nesnesi ile alınan metot sınıfın Start metoduyla çalıştırılabilir. Örneğin:

```
Task task = new Task(new Action(Sample.Foo));
task.Start();
```

Örneğin:

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

```

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task1 = new Task(new Action(Sample.Foo));
            Task task2 = new Task(Sample.Foo);
            Task task3 = new Task(delegate { Console.WriteLine("This is a test"); });
            Task task4 = new Task(() => { Console.WriteLine("This is a test"); });

            Task task5 = new Task(new Action<object>(Sample.Bar), "This is a test");
            Task task6 = new Task(delegate(object o) { Console.WriteLine(o); }, "This is a test");
            Task task7 = new Task(o => { Console.WriteLine(o); }, "This is a test");

            task1.Start();
            task2.Start();
            task3.Start();
            task4.Start();
            task5.Start();
            task6.Start();
            task7.Start();

            Console.ReadLine();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("This is a test");
        }

        public static void Bar(object o)
        {
            Console.WriteLine(o);
        }
    }
}

```

Task nesnesi Task sınıfının static Run metotlarıyla da yaratılabilmektedir. Bu metotlar kendi içerisinde Task nesnesini yaratıp Start ederler ve bize Task nesne referansını verirler. Tipik bir Run metodunun parametrik yapısı şöyledir:

```
public static Task Run(Action action)
```

Bu metot aşağıdaki gibi yazılmıştır:

```

public static Task Run(Action action)
{
    Task task = new Task(action);
    task.Start();

    return task;
}

```

Örneğin:

```

using System;
using System.Threading.Tasks;

```

```

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task = Task.Run(new Action(Sample.Foo));

            Console.ReadLine();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("This is a test");
        }
    }
}

```

Task sınıfının paralel programlamayı kolaylaştıran pek çok elemanı vardır. Bunlar izleyen bölümlerde ele alınacaktır.

## Parallel Sınıfı

Parallel sınıfının iki önemli metodu vardır: For ve ForEach. Bu iki metod da overload edilmiştir. Bu metodlar normal for ve foreach döngülerini taklit ederler. For metodu kabaca şöyle çalışmaktadır: Biz bu metoda bir başlangıç indeksi bir de bitim indeksi ayrıca da bir delege referansı veririz. For metodu bu aralıktaki her sayaç için delege metodunu çağırır. Yani delege metodu adeta for döngüsünün döngü deymi (yani gövdesi) gibi işlem görür. Fakat For metodu bu işlemi parallik sağlayarak yapar. Yani kabaca sistemdeki CPU ya da çekirdek sayısı kadar arka planda thread oluşturur ve döngünün bu kadar thread'le paralel çalışmasını sağlar. Örneğin sistemimizde 4 çekirdekbulunuyor olsun. Biz de For metoduyla 1000000 kere dönmek isteyelim. Bu metod kendi içerisinde thread havuzu yoluyla 4 thread oluşturur. Her thread'i 250000 kez döndürür. Sonuç olarak 1000000'luk döngü 4 ayrı thread tarafından paralel olarak 250000 kez döndürülür. For metodunun tipik overload'u şöyledir:

```

public static ParallelLoopResult For(int fromInclusive, int toExclusive, Action<int> body)

```

Metodun birinci parametresi indeksin başlangıç değerini, ikinci parametresi bitim değerini belirtir. Üçüncü parametre her yinelemeye çalıştırılacak metodu belirtir. Metodun geri dönüş değeri ParallelLoopResult isimli bir yapı türündendir.

Örneğin:

```

using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Stopwatch sw = new Stopwatch();

            sw.Start();
            Parallel.For(0, 10000000, new Action<int>(Sample.Foo));
            sw.Stop();
        }
    }
}

```

```

        Console.WriteLine(sw.ElapsedMilliseconds);

        sw.Reset();

        sw.Start();
        for (int i = 0; i < 10000000; ++i)
            Sample.Foo(i);

        sw.Stop();
        Console.WriteLine(sw.ElapsedMilliseconds);
    }
}

class Sample
{
    public static void Foo(int i)
    {
        Random r = new Random();

        r.Next(100);
    }
}

```

Şüphesiz paralel for ve foreach işlemlerinde anonim metotlar ya da lambda ifadeleri çok daha etkindir.Çünkü bu döngülerin gövdelerinde üst bloktaki değişkenlere erişmek isteyebiliriz. Yukarıdaki kodun lamdalı versiyonu şöyledir:

```

using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Stopwatch sw = new Stopwatch();

            sw.Start();
            Parallel.For(0, 10000000, index =>
            {
                Random r = new Random();
                r.Next();
            });
            sw.Stop();
            Console.WriteLine(sw.ElapsedMilliseconds);

            sw.Reset();

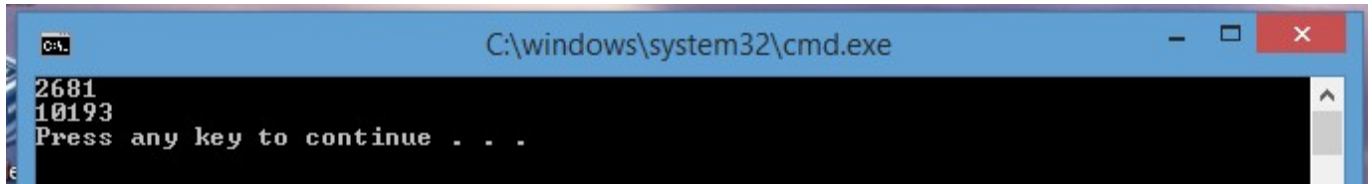
            sw.Start();
            for (int i = 0; i < 10000000; ++i) {
                Random r = new Random();
                r.Next();
            }

            sw.Stop();
            Console.WriteLine(sw.ElapsedMilliseconds);
        }
    }
}

```

Bu kodun çalıştırıldığı makinada 4'ü gerçek 4'ü yapay (hyper threading) olmak üzere 8 çekirdekli bir

işlemci vardır. Aşağıdaki gibi bir çıktı elde edilmektedir:



```
C:\windows\system32\cmd.exe
2681
10193
Press any key to continue . . .
```

Görüldüğü gibi 4 kat civarı bir hızlanma oluşmuştur.

For metodunda dikkat edilmesi gereken birkaç nokta vardır: Birincisi bu metotlar indisler sırasıyla gitmek zorunda değildir. Bu durum şöyle test edilebilir:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Parallel.For(0, 100, (int i) =>
            {
                Console.Write("{0} ", i);
            });
            Console.WriteLine();
        }
    }
}
```

Dikkat edilmesi gereken ikinci nokta paralel for ve foreach döngülerinin gereksiz kullanımında hız kazancı yerine hız kaybının oluşabilmesidir. Örneğin biz 1000'e kadar dönen bir döngüyü paralel for ile oluşturmak istersek bundan kazanç sağlayamayabiliriz. Tam tersine bu işten zararlı çıkabiliriz. Çünkü paralel for kendi içerisinde thread'leri yaratırken bir zaman kaybı oluşmaktadır. Bizim bu tür işlemlerden hız kazancı sağlayabilmemiz için buradaki kayıpların döngünün yaptığı şeylere göre önemsiz olması gerekir. (Fakat döngünün miktarı tek belirleyici değildir. Önemli olan her yinelemedeki işlem süresidir. Yani örneğin 1000'lik bir döngüde her yinelemede çok uzun işlemler yapılıyorsa paralelleştirmeye bundan kazanç sağlanabilir). Fakat ne olursa olsun programcının oluşacak kazancı test etmesi tavsiye edilir. Tabi programcı deneyim kazandıkça bu test işlemlerini yapmayabilir. Bu tür testlerde ismine “profiler” denilen araçlar kullanılabilir. “Open source” ya da “Ücretli” profiler'lar olmakla birlikte zaten Visual Studio IDE'sinde (paralı versiyonlarında) profiler IDE'ye entegre edilmiştir.

Parallel sınıfının ForEach metotları foreach deyimini taklit eder. Foreach metoduna biz bir dizi ya da collection, bir de delege veririz. Bu metot dizi ya da collection'ın her elemanı için delege metodunu çağırır. Tabi delege metoduna dizi ya da collection'ın elemanını parametre olarak geçirir. ForEach metodunun en yalın overload biçimi şöyledir:

```
public static ParallelLoopResult ForEach<TSource>(
    IEnumerable<TSource> source,
    Action<TSource> body
)
```

Metodun birinci parametresi dizi ya da collection referansını alır, ikinci parametresi çağrılacak delegeyi alır. Metot döngünün nasıl sonlandığına ilişkin ParallelLoopResult değerine geri dönmektedir. Örneğin:

```
using System;
```

```

using System.Threading.Tasks;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            int[] a = new int[10] { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

            Parallel.ForEach(a, new Action<int>(Sample.Body));
        }
    }

    class Sample
    {
        public static void Body(int a)
        {
            Console.WriteLine(a);
        }
    }
}

```

Aynı programın lamdalı versiyonu şöyle oluşturulabilir:

```

using System;
using System.Threading.Tasks;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            int[] a = new int[10] { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

            Parallel.ForEach(a, val =>
            {
                Console.WriteLine(val);
            });
        }
    }
}

```

### **Parallel For ve ForEach metotlarında Break kullanımı**

Nasıl normal for ve foreach deyimlerinde break kullanabiliyorsak bunların paralel versiyonlarında da break kullanabiliriz. Ancak Break metodunu kullanabilmek için For ve ForEach metotlarının ParallelLoopState içeren iki generic parametrelili Action versiyonlarından faydalanmak gerekir:

```

public static ParallelLoopResult For(
    int fromInclusive,
    int toExclusive,
    Action<int, ParallelLoopState> body
)

```

Burada kullanılacak delege metodunun birinci parametresi int türden, ikinci parametresi ParallelLoopState türünden olmalıdır. Break metodu bu ParallelLoopState sınıfının metodudur. Örneğin:

```

using System;
using System.Threading.Tasks;

```

```

namespace _070_ParallelProgramming
{
    class Program
    {
        public static void Main()
        {
            Parallel.For(0, 100, Sample.Foo);
        }
    }

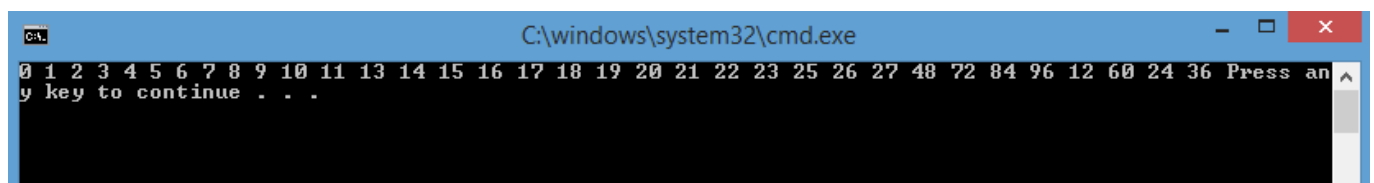
    class Sample
    {
        public static void Foo(int index, ParallelLoopState pls)
        {
            Console.WriteLine("{0} ", index);
            if (index == 27)
                pls.Break();
        }
    }
}

```

Break işlemi uygulandığında artık o işlemden sonra sayacı break yapılan index'ten büyük olan thread'ler sonlandırılır. Fakat break yapılan indekse kadar olan bütün index'ler işleme sokulmuş olur. Örneğimizde 27<sup>inci</sup> index'te Break yapılmıştır. Atrık index'i 27'den büyük olan thread'ler sonlandırılma sürecine sokulur. (Tabi bu da bir süreçtir. Yani bu süreçte hala bazı thread'ler biraz daha işlemine devam edebilir.) Fakat index'i 27'den küçük olanların çalışması sağlanır. Böylece Break yapılan indeks'e kadar tüm indekslerin çalışmış olması garanti edilir.

Ayrıca ParallelLoopState sınıfının bir de Stop metodu vardır. Stop metodu ile Break metodu arasındaki fark şöyledir: Biz Break metodunu n'inci indekste uygulamışsak n'inci indekse kadar olan bütün indeks'ler için delege metodunun çağırılması garanti edilir. Ancak biz n'inci indeks'te Stop uygulamışsak bütün thread'ler sonlandırma sürecine sokulur. Yani artık n'inci indeksten öncekilerin hepsinin çalışmış olması garanti edilmemektedir.

**Anahtar Notlar:** Örneğin biz Break metoduyla 27'indekste Break uygulamış olalım. Bundan sonra da birkaç tane 27'den büyük indeksle işlem yapılabilir. Çünkü 27'den büyük diğer thread'lerin sonlandırılması aniden yapılamamaktadır. Bu sonlandırma sürecinde birkaç tane 27'den büyük indeks için işlem yapılmış olabilir. Örneğin yukarıdaki programın ekran çıktısı şöyledir:



Ayrıca ParallelLoopState sınıfının bir de LowestBreakIteration isimli bir property elemanı vardır. Bu property bize Break oluştuğundaki en küçük indeksi verir.

Benzer biçimde ForEach metodunda da Break ve Stop kullanılabilir. Örneğin:

```

using System;
using System.Diagnostics;
using System.Threading.Tasks;

```

```

namespace _070_ParallelProgramming
{
    class Program
    {
        public static void Main()
        {
            int[] a = new int[1000000];
            Random rand = new Random();

            a[700000] = 123456789;

            Sample s = new Sample(a);

            Stopwatch sw = new Stopwatch();

            sw.Start();
            Parallel.For(0, a.Length, s.Find);
            sw.Stop();
            Console.WriteLine("Paralel versiyon: {0}", sw.ElapsedMilliseconds);

            sw.Reset();
            sw.Start();
            for (int i = 0; i < a.Length; ++i)
            {
                // Zaman alıcı birşeyler
                Random r = new Random();
                r.Next();
                if (a[i] == 123456789)
                {
                    Console.WriteLine("Değer Bulundu: {0}", i);
                    break;
                }
            }
            sw.Stop();
            Console.WriteLine("Normal versiyon: {0}", sw.ElapsedMilliseconds);
        }
    }

    class Sample
    {
        private int[] m_a;

        public Sample(int[] a)
        {
            m_a = a;
        }

        public void Find(int index, ParallelLoopState pls)
        {
            // Zaman alıcı birşeyler
            Random r = new Random();
            r.Next();

            // Asıl arama işlemi
            if (m_a[index] == 123456789)
            {
                Console.WriteLine("Değer Bulundu: {0}", index);
                pls.Stop();
            }
        }
    }
}

```



## Paralel For ve Foreach Metotlarının Geri Dönüş Değerleri

For ve ForEach metotları ParallelLoopResult isimli bir yapı türüne geri döner. Bu yapının IsCompleted property'si döngünün normal olarak işini bitirip bitirmediği bilgisini bize verir. Örneğin biz döngüden Break ya da Stop ile çıkmışsak IsCompleted false değerini verir. Yapının LowestBreakIteration elemanı döngülerden Break ile çıkılması durumunda bize Break sırasındaki index numarasını verir. Örneğin:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            var result = Parallel.For(0, 1000000, (index, loopState) =>
            {
                if (index == 500000)
                    loopState.Break();
            });

            if (result.IsCompleted)
                Console.WriteLine("Normal sonlandı");
            else
                Console.WriteLine("Break ya da Stop ile sonlandı. LowestBreakIteration = {0}",
result.LowestBreakIteration);
        }
    }
}
```

## Paralel For ve Foreach Metotlarında Exception Oluşursa Ne Olur?

Paralel For ve Foreach metotlarının gövdesinde exception oluşursa ve bu exception orada ele alınmamışsa For ve Foreach için oluşturulmuş olan tüm thread'ler sonlandırılır ve exception For ve Foreach çağrılarının yapıldığı yere yönlendirilir. Yani biz bu exceptionları orada ele alabiliriz. Ancak dışarıdan aldığımız Exception gerçek fırlatılan Exception değildir. Bu durumda dışarıdan AggregateException isimli bir exception yakalanır. Örneğin:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            try
            {
                var result = Parallel.For(0, 1000000, new Action<int>(Sample.Foo));
            }
            catch (AggregateException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }

    class Sample
```

```

{
    public static void Foo(int index)
    {
        if (index == 500000)
            throw new Exception();
    }
}
}

```

## Paralel Metot Çağrıları

Bazen bir grup metodun aynı anda paralel çalışması istenir. Bu işlemi Parallel sınıfının Invoke metotları yapmaktadır. İki Invoke metodu vardır:

```

public static void Invoke(params Action[] actions);
public static void Invoke(ParallelOptions parallelOptions, params Action[] actions)

```

Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Parallel.Invoke(Sample.Foo, Sample.Bar, Sample.Tar, Sample.Car);
            Console.WriteLine("Ok");
        }
    }

    class Sample
    {
        public static void Foo()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Foo");
                Thread.Sleep(200);
            }
        }

        public static void Bar()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Bar");
                Thread.Sleep(200);
            }
        }

        public static void Tar()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Tar");
                Thread.Sleep(200);
            }
        }
    }
}

```

```

public static void Car()
{
    for (int i = 0; i < 10; ++i)
    {
        Console.WriteLine("Car");
        Thread.Sleep(200);
    }
}
}
}

```

Aynı örneğin lamdalı versiyonu:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Parallel.Invoke(
                () =>
                {
                    for (int i = 0; i < 10; ++i)
                    {
                        Console.WriteLine("Foo");
                        Thread.Sleep(200);
                    }
                },
                () =>
                {
                    for (int i = 0; i < 10; ++i)
                    {
                        Console.WriteLine("Bar");
                        Thread.Sleep(200);
                    }
                },
                () =>
                {
                    for (int i = 0; i < 10; ++i)
                    {
                        Console.WriteLine("Tar");
                        Thread.Sleep(200);
                    }
                },
                () =>
                {
                    for (int i = 0; i < 10; ++i)
                    {
                        Console.WriteLine("Car");
                        Thread.Sleep(200);
                    }
                }
            );

            Console.WriteLine("Ok");
        }
    }
}

```

Invoke metotları tüm paralel çağrılar bitene kadar onun çağrıldığı thread'i bloke eder. Yani tüm çağrılar

sonlandığında akış Invoke'tan sonraki deyime geçecektir.

Paralel Invoke işlemi ne zaman kullanılmalıdır? Bir kere genel olarak paralel programlama bir hız artışı sağlamak için başvurulması gereken bir tekniktir. Yani programımızın hız konusunda hiçbir sorunu yoksa, programımız istenileni makul süre içerisinde yapıyorsa bizim paralel programlamayla zaten bir ilgimiz olmaz. İşte hız kazancının sağlanması istendiği bir durumda bazı işler birbirlerine bağlı değilse onlar paralel bir biçimde yürütülebilir. Bu da ciddi bir hız kazancı sağlayabilir (tabi yanlış tercih bizim bu konuda zarar etmemeye yol açabilir.) Örneğin Foo, Bar, Tar ve Car metotları birbirinden bağımsız işlem yapıyor olsun. Biz de bunların sonuçları üzerinde işlem yapan Mar metodunu çağırmak isteyelim. Bu işlemin seri halde yapımı şöyle olur:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Stopwatch sw = new Stopwatch();

            sw.Start();
            Sample.Foo();
            Sample.Bar();
            Sample.Tar();
            Sample.Car();
            Sample.Mar();
            sw.Stop();

            Console.WriteLine(sw.ElapsedMilliseconds);
        }
    }

    class Sample
    {
        public static void Foo()
        {
            for (int i = 0; i < 2000000000; ++i )
                ;
        }

        public static void Bar()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }

        public static void Tar()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }

        public static void Car()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }
    }
}
```

```

    public static void Mar()
    {
        for (int i = 0; i < 2000000000; ++i)
            ;
    }
}

```

Kursumuzdaki makinada bu işlem 20648 milisaniye sürmüştür. Şimdi aynı işlemi paralel bir biçimde aşağıdaki gibi yapabiliriz:

```

using System;
using System.Threading.Tasks;
using System.Threading;
using System.Diagnostics;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Stopwatch sw = new Stopwatch();

            sw.Start();
            Parallel.Invoke(Sample.Foo, Sample.Bar, Sample.Tar, Sample.Car);
            Sample.Mar();
            sw.Stop();

            Console.WriteLine(sw.ElapsedMilliseconds);
        }
    }

    class Sample
    {
        public static void Foo()
        {
            for (int i = 0; i < 2000000000; ++i )
                ;
        }

        public static void Bar()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }

        public static void Tar()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }

        public static void Car()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }

        public static void Mar()
        {
            for (int i = 0; i < 2000000000; ++i)
                ;
        }
    }
}

```

```
}  
}
```

Bu işlem ise kursumuzdaki makinada 8637 milisaniye sürmüştür.

Invoke metodunun ParallelOptions parametrelili versiyonu çizelgeleme ayarı yapmakta ve işlemi iptal etmekte kullanılmaktadır. ParallelOptions sınıfı ileride Task sınıfının detaylarında gösterilecektir.

## Action Delegeleri

Görüldüğü gibi paralel işlemlerde Action isimli bir delegeden faydalanılmaktadır. Kütüphanedeki Action delegeleri şöyle organize edilmiştir.

- Generic olmayan Action delegesi geri dönüş değeri void parametresi olmayan metotları tutar.
- 1 generic parametreden başlayarak 10 generic parametreye kadar toplam 10 farklı generic Action delegesi de vardır. Bunlar geri dönüş değeri void olan parametreleri farklı türlerden olabilen fakat generic parametresi kadar olan metotları tutabilir. Yani örneğin 3 generic parametresli Action delegesi geri dönüş değeri void olan herhangi türden 3 parametreye sahip bir metodu tutabilir. Örneğin:

```
class Program  
{  
    public static void Main()  
    {  
        Action<int, long> a = new Action<int, long>(Foo);  
        Action<int, long> b = (int x, long y) =>  
        {  
            //...  
        };  
        Action<int, long> c = (x, y) =>  
        {  
            //...  
        };  
    }  
  
    public static void Foo(int a, long b)  
    {  
        //...  
    }  
}
```

## Task Sınıfının Ayrıntıları

İki Task sınıfı vardır: Generic olan ve generic olmayan. Generic olmayan geri dönüş değeri void olan metotları çalıştırmak için, generic olan geri dönüş değeri herhangi bir tür olan metotları çalıştırmak için kullanılır. Örneğin:

```
Task t1;           // geri dönüş değeri void olan metotları çalıştırmak için  
Task<int> t2;      // geri dönüş değeri int olan metotları çalıştırmak için
```

Task sınıfının çalıştıracağı metotların parametreleri başlangıç metoduna bağlıdır. Generic olmayan Task sınıfının Action parametrelili başlangıç metotları parametresi olmayan, Action<object> parametrelili başlangıç metotları parametresi object olan metotları çalıştırır. Yani özetle generic olmayan Task sınıfı bizden geri dönüş değeri void parametresi olmayan ya da object olan metotları alabilir. Örneğin:

```
using System;  
using System.Threading.Tasks;  
  
namespace _070_ParallelPrograming  
{
```

```

class Program
{
    public static void Main()
    {
        Task task = new Task(Foo);

        task.Start();

        Console.ReadLine();
    }

    public static void Foo()
    {
        //...
    }
}

```

object parametrelili metodu kullanırken başlangıç metodu iki parametre alır. Birincisi Action<object> türünden delege, ikincisi de o metodu geçirilecek argümandır. Örneğin:

```

using System;
using System.Threading.Tasks;

namespace _070_ParallelProgramming
{
    class Program
    {
        public static void Main()
        {
            Task task = new Task(Foo, "this is a test");

            task.Start();
            Console.ReadLine();
        }

        public static void Foo(object o)
        {
            Console.WriteLine(o.ToString());
        }
    }
}

```

Generic parametrelili Task sınıfı çalıştıracağı metodun geri dönüş değerini generic parametresi olarak alır. Generic Task sınıfı bizden metodu Func isimli bir delege türüyle ister. Func delegesinin Action delegesinden basit bir farkı vardır: Action delegesinde metodun geri dönüş değeri void biçimdedir. Halbuki Func delegesinde metodun geri dönüş değeri istenildiği gibi generic parametreyle ayarlanabilir. Örneğin:

```

Action a = new Action(Foo);           // Foo'nun geri dönüş değeri void parametresi olmayan metot
Action<int> b = new Action(Bar);       // Bar'ın geri dönüş değeri void parametresi int
Func<int> c = new Func(Tar);           // Tar geri dönüş değeri int, parametresiz olmayan metot

```

Func isimli delegenin generic versiyonlarında son generic parametre her zaman geri dönüş değerinin türüne ilişkin, diğer generic parametreler metot parametrelerinin türlerine ilişkindir. Örneğin:

```

// Foo geri dönüş değeri int parametresi olmayan metot
Func<int> a = new Func<int>(Foo);

```

```

// Bar geri dönüş değeri long parametresi int olan metot
Func<int, long> b = new Func<int, long>(Bar);

```

```
// Tar geri dönüş değeri double , parametresi int ve long olan metot
Func<int, long, double> c = new Func<int, long, double>(Tar);
```

Bu biçimde yine 10 parametreue kadar generic Func vardır.

Peki Action ve Func isimli iki delegenin bulunması zorunlu muydu? Yani yalnızca Func ile tüm ihtiyaçlar karşılanamaz mıydı? Yanıt: Malesef C# buna izin vermemektedir. Çünkü aşağıdaki gibi bir açım C#'ta yapılamamaktadır:

```
Func<int, void> f;           // error!
```

Örneğin C++'ta yukarıdaki bir açım geçerlidir. Ancak C#'ta geçerli değildir. Bu nedenle geri dönüş değeri void olan metotları için ayrı bir delege oluşturulmuştur.

İşte Task<T> sınıfının başlangıç metotlarının parametreleri geri dönüş değeri T türünden olan metotları almaktadır. Örneğin:

```
public Task(Func<TResult> function)
```

Örneğin aşağıdaki başlangıç metodu bizden geri dönüş değeri TResult, parametresi object olan bir metot ister:

```
public Task(Func<Object, TResult> function, Object state)
```

Örneğin:

```
Task<int> task = new Task<int>(Foo, "this is a test");
```

Burada Foo metodu geri dönüş değeri int, parametresi object olan bir metottur. "this is a test" yazısı da bu Foo metodu thread'te çalıştırıldığında ona geçirilecek argümanı belirtir. Örneğin:

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

namespace _070_ParallelProgramming
{
    class Program
    {
        public static void Main()
        {
            Task<int> task = new Task<int>(Foo, "this is a test");

            task.Start();
            Console.ReadLine();
        }

        public static int Foo(object o)
        {
            Console.WriteLine(o.ToString());

            return 100;
        }
    }
}
```

### Soru Cevap

Soru: Geri dönüş değeri int, parametresi olmayan bir metodu tutabilecek Task nesnesi yaratınız



Yanıt:

```
Task<int> task = new Task<int>(Foo);
```

Soru: Geri dönüş değeri int olan parametresi object olan Bar metodu vardır. Bunu tutacak Task nesnesi yaratınız?

Yanıt:

```
Task<int> task = new Task<int>(Bar);
```

Soru: Geri dönüş değeri int, parametresi long ve double olan bir Tr metodu vardır. Buna tutacak Task nesnesi yaratınız:

Yanıt: Böyle bir Task nesnesi oluşturulamaz. Çünkü Task sınıfının böyle bir başlangıç metodu yoktur. Task sınıfı yalnızca object parametresli metotları ya da parametresi olmayan metotları tutabilir.

Task sınıfının Result isimli property elemanı çalıştırılan metodun geri dönüş değerini bize verir. Örneğin:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<int> task = new Task<int>(Foo, "this is a test");

            task.Start();
            Console.WriteLine(task.Result);
            Console.WriteLine("Ok");
        }

        public static int Foo(object o)
        {
            Thread.Sleep(3000);
            Console.WriteLine(o.ToString());

            return 100;
        }
    }
}
```

Result property'si Task bitene kadar blokeye yol açar. Tabi generic olmayan Task sınıfında böyle bir property yoktur.

Task sınıfının .NET'e dahil edilmesiyle mevcut bazı IO işlemi yapan sınıflara çeşitli eklemeler de yapılmıştır. Bu konu izleyen bölümlerde ele alınacaktır.

### **Çalışmakta Olan Task'ların Bitmesinin Beklenmesi**

Bazen biz bazı Task'ları paralel çalıştırıp onların bitmesini bekleyebiliriz. Çünkü yapacağımız işlemler için onların işlerini bitirmiş olması gerekir. İşte bunun için Task sınıflarının Wait metotları kullanılır. Örneğin static olmayan Wait metodu sanki Thread sınıfının Join metodu gibidir. O Task bitene kadar bekleme yapar. Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task = new Task(() =>
            {
                for (int i = 0; i < 10; ++i)
                {
                    Console.Write("{0} ", i);
                    Thread.Sleep(300);

                }
                Console.WriteLine();
            });

            task.Start();
            task.Wait();
            Console.WriteLine("Ok");
        }
    }
}

```

Wait metodunun zaman aşımli versiyonu da vardır.

Task sınıfının static WaitAll metotları tüm task'lar bitene kadar bekleme yapar. Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task1 = new Task(() =>
            {
                for (int i = 0; i < 10; ++i)
                {
                    Console.Write(".");
                    Thread.Sleep(300);

                }
                Console.WriteLine();
            });

            Task task2 = new Task(() =>
            {
                for (int i = 0; i < 10; ++i)
                {
                    Console.Write(",");
                    Thread.Sleep(300);

                }
                Console.WriteLine();
            });

            Task.WaitAll(task1, task2);
        }
    }
}

```

```

});

task1.Start();
task2.Start();

Task.WaitAll(new Task[] { task1, task2 });

Console.WriteLine("Ok");
}
}
}

```

Task sınıfının WaitAny isimli static metotları herhangi ilk task bitene kadar bekleme yapar. Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task1 = new Task(() =>
            {
                for (int i = 0; i < 10; ++i)
                {
                    Console.Write(".");
                    Thread.Sleep(300);
                }
                Console.WriteLine();
            });

            Task task2 = new Task(() =>
            {
                for (int i = 0; i < 5; ++i)
                {
                    Console.Write(",");
                    Thread.Sleep(300);
                }
                Console.WriteLine();
            });

            task1.Start();
            task2.Start();

            Task.WaitAny(new Task[] { task1, task2 });

            Console.WriteLine("Ok");
        }
    }
}

```

## Task'ların Koordine Edilmesi

Bir Task bittiği zaman diğer bir Task'ın çalıştırılması sağlanabilir. Böylece task'lar bir plan çerçevesinde çizelgelenmiş olurlar. Örneğin biz Foo, Bar bittiğinde Tar'ın çalışmasını Tar bittiğinde Car ve Mar'ın çalışmasını isteyebiliriz. Bunu baştan kurgulayıp otomatize edebiliriz.

Bir Task bittiğinde diğer bir Task'ın çalışması sağlamak için Task sınıflarının static olmayan ContinueWith metotları kullanılır. Örneğin:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task task1 = new Task(Sample.Foo);
            Task task2 = task1.ContinueWith(Sample.Bar);

            task1.Start();
            task2.Wait();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Foo");
                Thread.Sleep(200);
            }
        }

        public static void Bar(Task task)
        {
            Console.WriteLine("Bar");
        }
    }
}
```

ContinueWith metotları ile önceki Task bitince yeni bir Task başlatılır. Bu yeni Task'ın metoduna önceki Task'ın referansı geçirilir. Böylece programcı isterse önceki Task'ın geri dönüş değerini vs. alabilir.

Task sınıflarının ContinueWith metotlarının bizden istediği delegelerin parametreleri kendi Task sınıfları türündendir. Örneğin Foo metodunun parametresi string geri dönüş değeri void olsun.

```
Task task1 = new Task(Foo);
Task task2 = task1.ContinueWith(Bar);
```

Burada Bar metodunun parametresi Task, geri dönüş değeri void türünden olmalıdır. Aynı işlem açıkça (yani otomatik deduction yapılmadan) şöyle de yazılabilirdi:

```
Task task1 = new Task(new Action<string>(Foo));
Task task2 = task1.ContinueWith(new Action<Task>(Bar));
```

Örneğin burada Foo metodunun parametresi string ve geri dönüş değeri string türünden olsun:

```
Task<string> task1 = new Task<string>(Foo);
Task task2 = task1.ContinueWith(Bar);
```

Burada Bar metodunun parametresi Task<string>, geri dönüş değeri void olmalıdır. Aynı işlem açıkça (yani otomatik deduction yapılmadan) şöyle de yazılabilirdi:

```
Task<string> task1 = new Task<string>(new Func<string, string>(Foo));
Task task2 = task1.ContinueWith(new Action<Task<string>>(Bar));
```

Peki devam edecek olan Task'ın geri dönüş değeri int olsun:

```
Task<string> task1 = new Task<string>(Foo);
Task<int> task2 = task1.ContinueWith<int>(Bar);
```

Burada Bar metodunun parametresi Task<string> geri dönüş değeri int olmalıdır. Aynı işlem açıkça (yani otomatik deduction yapılmadan) şöyle de yazılabilirdi:

```
Task<string> task1 = new Task<string>(new Func<string, string>(Foo));
Task<int> task2 = task1.ContinueWith<int>(new Func<Task<string>,int>(Bar));
```

Bu son durum için aşağıdaki örnek verilebilir:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string> task1 = new Task<string>(new Func<string>(Sample.Foo));
            Task<int> task2 = task1.ContinueWith<int>(new Func<Task<string>, int>(Sample.Bar));

            task1.Start();
            task2.Wait();
            Console.WriteLine(task1.Result);
            Console.WriteLine(task2.Result);
        }
    }

    class Sample
    {
        public static string Foo()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Foo");
                Thread.Sleep(200);
            }

            return "Foo ends";
        }

        public static int Bar(Task<string> task)
        {
            Console.WriteLine("Bar");

            return 100;
        }
    }
}
```

Otomatik deduction ile aynı işlem şöyle yapılabilirdi:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string> task1 = new Task<string>(Sample.Foo);
            Task<int> task2 = task1.ContinueWith<int>(Sample.Bar);

            task1.Start();
            task2.Wait();
            Console.WriteLine(task1.Result);
            Console.WriteLine(task2.Result);
        }
    }

    class Sample
    {
        public static string Foo()
        {
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("Foo");
                Thread.Sleep(200);
            }

            return "Foo ends";
        }

        public static int Bar(Task<string> task)
        {
            Console.WriteLine("Bar");

            return 100;
        }
    }
}
```

Aynı işlem lamdalı olarak şöyle yapılabilirdi:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string> task1 = new Task<string>(() => {
                for (int i = 0; i < 10; ++i)
                {
                    Console.WriteLine("Foo");
                    Thread.Sleep(200);
                }
            });
        }
    }
}
```

```

        return "Task1 ends";
    });
    Task<int> task2 = task1.ContinueWith<int>((taskStr) => {
        Console.WriteLine("Bar");

        return 100;
    });

    task1.Start();
    task2.Wait();
    Console.WriteLine(task1.Result);
    Console.WriteLine(task2.Result);
}
}
}

```

Bir task bitince birden fazla task paralel çalıştırılabilir. Bunun için yapılacak tek şey birden fazla ContinueWith kullanmaktır. Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string> task1 = new Task<string>(() => {
                for (int i = 0; i < 10; ++i)
                {
                    Console.WriteLine("Foo");
                    Thread.Sleep(200);
                }

                return "Task1 ends";
            });
            Task<string> task2 = task1.ContinueWith<string>((taskStr) => {
                for (int i = 0; i < 3; ++i)
                {
                    Console.WriteLine("task2");
                    Thread.Sleep(500);
                }

                return "Task2 ends";
            });
            Task<string> task3 = task1.ContinueWith<string>((taskStr) =>
            {
                for (int i = 0; i < 3; ++i)
                {
                    Console.WriteLine("task3");
                    Thread.Sleep(500);
                }

                return "Task3 ends";
            });

            task1.Start();

            Console.WriteLine(task1.Result);
            Console.WriteLine(task2.Result);

```

```

    }
  }
}

```

Benzer bir örnek:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string>[] tasks = new Task<string>[10];

            Task<string> task1 = new Task<string>(() => {
                for (int i = 0; i < 10; ++i)
                {
                    Console.WriteLine("Foo");
                    Thread.Sleep(200);
                }

                return "Task1 ends";
            });

            for (int k = 0; k < 10; ++k)
                tasks[k] = task1.ContinueWith<string>((taskStr) => {
                    for (int i = 0; i < 3; ++i)
                    {
                        Console.WriteLine("task");
                        Thread.Sleep(500);
                    }

                    return "ends";
                });

            task1.Start();
            Task.WaitAll(tasks);
        }
    }
}

```

Yukarıdaki işlemin tersi de yapılabilir. Yani bir'den fazla task bittikten sonra bir task çalıştırılabilir. Bunun için TaskFactory sınıfının ContinueWhenAll metotları kullanılır. TaskFactory nesnesi Task sınıfının Factory static property'si ile elde edilir (singleton kalıbı). Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Task<string>[] tasks = new Task<string>[10];

            for (int k = 0; k < 10; ++k)
            {

```



```

        tasks[k] = new Task<string>(() =>
        {
            for (int i = 0; i < 3; ++i)
            {
                Console.WriteLine("task");
                Thread.Sleep(500);
            }

            return "ends";
        });
        tasks[k].Start();
    }
    Task finalTask = Task.Factory.ContinueWhenAll(tasks, (allTasks) =>
    {
        Console.WriteLine("finally task");
    });

    Task.WaitAll(finalTask);
}
}
}

```

Ayrıca TaskFactory (bu nesne Task.Factory ifadesiyle elde edilmelidir) sınıfının ContinueWhenAny isimli metotları da vardır. Bu metotlar da bu task'lardan herhangi biri bitince çalıştırılır.

### Asnkron BeginXXX, EndXXX Yöntemlerine Alternatif Task Yöntemi

Bloke olmaktan kaçmak için kullanılabilecek iki önemli yöntem vardı. Birincisi manuel thread yaratıp blokeye ayol açabilecek işlemi o thread'te yapmak, ikincisi BeginXXX, EndXXX kalıbını kullanmak. BeginXXX, EndXXX yönteminde işlem BeginXXX ile başlatılır. Fakat BeginXXX blokeye yol açmaz. İlgili olay gerçekleştiğinde bizim istediğimiz bir metot çağrılır. Orada da biz EndXXX yaparak işlemi bitiririz. İşte bu yöneme alternatif olarak Task yöntemi kullanılabilir. Şöyle ki: Blokeye yol açabilecek işlem bir Task nesnesi yaratılarak task metoduna yaptırılır. Sonra bu metot bittiğinde sonuçlar ContinueWith ile devam ettirilen task'ta işlenir. İstenirse de duruma göre yeniden aynı döngüye girilir. Örneğin:

```

Task task = new Task(() => {
    <blokeye yol açan işlem>
});
task.ContinueWith((taskPrev) => {
    <işlem bitti gereğini yap duruma göre onu yeniden devam ettir>
});

task.Start();

```

TPL'de biten bir Task yeniden başlatılamaz. Bunun için Task nesnesinin yeniden yaratılması gerekir.

Yeniden başlatma örneği şöyle verilebilir:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {

```

```

for (; ; )
{
    Task task1 = new Task(() =>
    {
        Console.WriteLine("Task1");
    });

    Task task2 = task1.ContinueWith((prevTask) =>
    {
        Console.WriteLine("Task2");
    });

    task1.Start();
    task2.Wait();
}

Console.ReadLine();
}
}
}

```

## Async Metotlar ve await Operatörü

Asenkron paralel programlamayı desteklemek için Framework 4.5'te (yani resmi son framework'te) async metot ve await operatörü eklenmiştir. Bir metodu async metot yapmak için metot bildiriminde async belirleyicisi kullanılır. Örneğin:

```

public static async void FooAsync()
{
    //...
}

```

async belirleyicisi erişim belirleyici anahtar sözcüklerle ve static anahtar sözcüğüyle aynı sentaktik gruptadır. Bu nedenle yer değiştirmeli olarak kullanılabilir. Microsoft async metotların isimlerinin XXXAsync biçiminde verilmesini tavsiye etmektedir.

async belirleyicisi kendi başına bir duruma yol açmaz. Ancak bize metot içeirisinde await operatörünün kullanılmasına olanak sağlar.

await operatörü yalnızca async metotlarda kullanılabilir. Ayrıca async bir metotta await kullanılmadıysa uyarı oluşur.

await operatörünün genel biçimi şöyledir:

**await <ifade>;**

await operatörünün yanındaki ifade tipik olarak Task ya da Task<T> türünden olmalıdır. Aslında Microsoft "C# Language Specification 5.0"da await operatörünün yanındaki ifadenin awaitable bir ifade olması gerektiğini belirtmektedir. Burada Microsoft bir ifadenin awaitable olması için onun bazı elemanlara sahip bir sınıf türünden olması gerektiğini belirtmiştir. Zaten Task ve Task<T> türleri bu elemanlara sahiptir. Dolayısıyla Tsk ve Task<T> türleri kütüphane içerisinde bulunan yegane awaitable türlerdir. Tabi programcı isterse kendi awaitable türlerini oluşturabilir. Fakat bunun görülen bir amacı yoktur.

Bir async metot çağrıldığında akış normal senkron metot çağırması gibi ilerler. Ta ki await operatörünü görene kadar. Yani await operatörü hiç kullanılmazsa (tabi bu normal bir durum değildir) bu işlemin normal bir metot çağrısından farkı kalmaz. Çatallanma yani paralelleşme ya da asenkronlaşma await

işlemleri gerçekleştirir. Programın akışı await anahtar sözcüğünü gördüğünde artık async metodu çağıran akış metottan çıkar. await operatörü ise operand olarak aldığı Task'ın bitmesini bekler. Peki await işlemi ile Task'ı kim beklemektedir. İşte bu durum dokümantasyonlarda açıkça belirtilmemiş olsa da en muhtemel gerçekleştirim framework tarafından bir Task (yani thread) yaratılıp beklemeyi onun yapmasıdır. Bu durumda tipik bir async metod çağrısında şu olayla gelişir:

- 1) async metodu çağıran thread await işlemine kadar senkron bir biçimde ilerler. Bu sırada bu akış bir Task yaratmış olabilir.
- 2) await işlemiyle birlikte framework bir Task (yani thread yaratıp) await operatörünün operandı belirtilen Task'ın bitmesini bekler. Fakat await işlemini gören asıl akış (yani async metodu çağıran akış) await gördüğünde geri döner.
- 3) await ile bitmesi beklenen Task bittiğinde akış await operatörünün aşağısından devam eder. Yani await operatörünün aşağısındaki kodu await operatörünün operandı belirtilen Task'ı bekleyen (framework tarafından yaratılmış olan Task) bekler.

Görüldüğü gibi senaryoda 3 thread akışı vardır: Birincisi async metodu çağıran akıştır. İkincisi async metod içerisinde yaratılan Task akışıdır. Üçüncüsü de framework tarafından o Task'ı beklemek için yaratılmış olan akıştır.

Tabi derleyici ve framework yukarıdaki işlemi daha optimize yapabilir. Örneğin async metodun await aşağısındaki kodu async metodun sonuna taşıyıp belki de hiç Task'ı bekleyen thread yaratmayabilir. Bu durum tamamen sistemin gerçekleştirimine bağlıdır. Fakat tipik gerçekleştirip yukarıdaki 3 maddede olduğu gibidir.

Örneğin:

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

namespace _070_ParallelProgramming
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("Main begins");
            FooAsync();
            Console.WriteLine("Main ends...");
            Console.ReadLine();
        }

        public static async void FooAsync()
        {
            Console.WriteLine("FooAsync begins");
            Task task = new Task(new Action(Sample.Foo));
            task.Start();

            await task;

            Console.WriteLine("FooAsync ends");
        }
    }

    class Sample
    {
        public static void Foo()
```

```

    {
        for (int i = 0; i < 5; ++i)
        {
            Console.WriteLine(i);
            Thread.Sleep(1000);
        }
    }
}

```

await operatörü de bir değer üretir. await operatörü ürün olarak bize beklenen Task'ın Result property değerini (yani task olarak çalıştırılan delege metodunun geri dönüş değerini) verir. Tabi eğer await'in operandı Task türündense çalıştırılan delege metodu void olduğu için await'ten bir değer elde edilmeyecektir. Fakat await beklenen Task<T> türündense await bize T türünden bir değer verecektir. Örneğin:

```

using System;
using System.Threading.Tasks;
using System.Threading;

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("Main begins");
            FooAsync();
            Console.WriteLine("Main ends");
            Console.ReadLine();
        }

        public static async void FooAsync()
        {
            Console.WriteLine("FooAsync begins");
            Task<string> task = new Task<string>(new Func<string>(Sample.Foo));
            task.Start();

            string str = await task;

            Console.WriteLine(str);
            Console.WriteLine("FooAsync ends");
        }
    }

    class Sample
    {
        public static string Foo()
        {
            for (int i = 0; i < 5; ++i)
            {
                Console.WriteLine(i);
                Thread.Sleep(1000);
            }

            return "this is is a test";
        }
    }
}

```

async bir metodun geri dönüş değeri herhangi bir türden olamaz. Ya void, ya Task ya da Task<T> türünden olmak zorundadır. Örneğin:

```
public static async int Foo()          // error!
{
    //...
}
```

Fakat örneğin:

```
public static async Task<int> Foo()     // geçerli
{
    //...
}
```

Eğer async metodun geri dönüş değeri void ise ya da Task ise bu metot içerisinde bir değerle return edilemez. Ancak async metodun geri dönüş değeri Task<T> ise bu metottan T ile geri dönmek gerekir. Örneğin:

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

namespace _070_ParallelPrograming
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("Main begins");
            Task<string> task = FooAsync();
            Console.WriteLine("Main ends");
            Console.WriteLine(task.Result);
            Console.ReadLine();
        }

        public static async Task<string> FooAsync()
        {
            Console.WriteLine("FooAsync begins");
            Task task = new Task(new Action(Sample.Foo));
            task.Start();

            await task;
            Console.WriteLine("FooAsync ends");

            return "async method's return value";
        }
    }

    class Sample
    {
        public static void Foo()
        {
            for (int i = 0; i < 5; ++i)
            {
                Console.WriteLine(i);
                Thread.Sleep(1000);
            }
        }
    }
}
```

Peki async metodun bize geri döndürdüğü Task ya da Task<T> hangi task'tır? async metot bize await

işlemi sırasında yaratılan ve await'in operandı ile belirtilen task'ı bekleyen task'ı verir. async metodun await'ten sonrasını bu bekleyen task çalıştırdığı için async metodun await'ten sonraki kısmı adeta ayrı bir metot gibi düşünülebilir. İO halde async metottaki return aslında await ile beklenen task'ı bekleyen task'ın geri dönüş değeri gibidir. Özetle eğer async metot örneğin Task<string> ile geri dönüyorsa async metot içerisinde biz string ile dönmeliyiz. Örneğin:

```
public static async Task<string> Foo()
{
    //...
    await <x task'ı>;
    //...
    return "this is a test";
}
```

Burada biz Foo metodunu çağırdığımızda bir Task<string> elde ederiz. Bu nesne x task'ını bekleyen task metodunun geri dönüş değeridir. Zaten await'in aşağısı da adeta x task'ını bekleyen task metodu gibi düşünülmelidir.

### async Metot Kullanımına Örnekler

async metotlar GUI uygulamalarında cross thread problemi oluşturmazlar. Bu nedenle async metotları uzun süren GUI eventlerind kullanabiliriz. Örneğin:

```
private async void m_butonOk_Click(object sender, EventArgs e)
{
    await Task.Run(() =>
    {
        for (int i = 0; i < 10; ++i) {
            m_labelCounter.Text = i.ToString();
            Thread.Sleep(1000);
        }
    });
}
```

Burada düğmeye tıklandığında async metot çağrılmaktadır. Orada da bir task yaratılıp işlem ona devredilmiştir.

Framework 4.5 ile birlikte pek çok IO sınıfına async metotlar eklenmiştir. Örneğin StreamReader sınıfına birkaç XXXAsync metodu eklenmiştir. Bunlardan biri ReadLineAsync metodudur:

```
public override Task<string> ReadLineAsync()
```

Bu metot dosya göstericisinin gösterdiği yerden bir satırlık bilgiyi okur. Bize Task<string> verir. Biz de Result property'si ile okunan satırı alabiliriz. Tabi bu metot okuma işlemini içeride bir thread açarak ona yaptırmaktadır. Biz ReadLineAsync metodunu çağırdığımızda bloke oluşmaz. Satırın okunması bu metot içerisinde açılan thread yoluyla yapılmaktadır. Ancak tabi biz Result ile okunan satırı elde etmek istediğimizde eğer işlem bitmemişse bloke bekleyebiliriz.

Benzer biçimde Stream sınıfına da ReadAsync ve WriteAsync metotları da eklenmiştir:

```
public override Task<int> ReadAsync(
    char[] buffer,
    int index,
    int count
)
```

```
public Task WriteAsync(  
    byte[] buffer,  
    int offset,  
    int count  
)
```

Bu modelde ilgili işlemin bittiğini bloke olmadan nasıl anlarız? Daha önceden de belirtildiği gibi bu modelde tipik olarak `ContinueWith` metotları bu amaçla kullanılmaktadır. Yani örneğin:

```
stream.Read(.....).ContinueWith(() => {  
    //...  
});
```

Örneğin:

```
private async void m_butonOk_Click(object sender, EventArgs e)  
{  
    await Task.Run(() =>  
    {  
        for (int i = 0; i < 10; ++i) {  
            m_labelCounter.Text = i.ToString();  
            Thread.Sleep(1000);  
        }  
    }).ContinueWith((task) => {  
        MessageBox.Show("Operation completed");  
    });  
}
```

Burada düğmeye tıklandığında arka planda bir `Task` yaratılıp uzun süren bir işlem yaptırılmıştır. İşlem bittiğinde de `ContinueWith` ile belirtilen `Task` çalıştırıldığı için ekrana `MessageBox` çıkacaktır.

Sonuç olarak `Task` modeli `BeginXXX`, `EndXXX` modelinin bir alternatifi olarak kullanılabilir. Anımsanacağı gibi `BeginXXX` ile işlem başlatılıyordu. İşlem bitince belirlenen delege metodu çağrılıyordu. İşte bu metot bu modelde `ContinueWith`'e karşılık gelmektedir.