عنوان: **اصول برنامه نویسی موازی درNET. نسخه 4 بخش اول - 1**
نویسنده: اشکان 56
تاریخ: ۱۳۹۱/۰۳/۳۱ ۱۲:۷
آدرس: www.dotnettips.info
برچسب‌ها: C#, Task Programming Language, Threading, Parallel Extensions

بدون هیچ مطلب اضافی به سراغ اولین مثال می‌رویم. قطعه کد زیر را در نظر بگیرید :

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

namespace Listing_01 {

class Listing_01 {

static void Main(string[] args) {
  Task.Factory.StartNew(() => {
         Console.WriteLine("Hello World");
  });

  // wait for input before exiting
  Console.WriteLine("Main method complete. Press enter to finish.");
  Console.ReadLine();
 }
}
```

در کد بالا کلاس Task نقش اصلی را بازی می‌کند.این کلاس قلب کتابخانه برنامه نویسی Task یا Task Programming Library می‌باشد.

در این بخش با موارد زیر در مورد Taskها آشنا می‌شویم:

- ایجاد و به کار انداختن انواع مختلف Task ها.
- کنسل کردن Task ها.
- منتظر شدن برای پایان یک Task.
- دریافت خروجی یا نتیجه از یک Task پایان یافته.
- مدیریت خطا در طول انجام یک Task

خب بهتر است به شرح کد بالا بپردازیم:

رای استفاده از کلاس Task باید فضای نام System.Threading.Tasks را بصورت ریر مورد استفاده قرار دهیم.

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

این فضای نام نقش بسیار مهمی در برنامه نویسی Taskها دارد . فضای نام بعدی معروف است : System.Threading . اگر با برنامه نویسی تریدها بروش مرسوم وکلاسیک آشنایی دارید قطعاً با این فضای نام آشنایی دارید. اگر بخواهیم با چندین Task بطور همزمان کار کنیم به این فضای نام نیاز مبرم داریم. پس :

```
using System.Threading;
```

خب رسیدیم به بخش مهم برنامه :

```
Task.Factory.StartNew(() => {
   Console.WriteLine("Hello World");
});
```

متد استاتیک Task.Factory.StartNew یک Task جدید را ایجاد و شروع می‌کند که متن Hello Word را در خروجی کنسول نمایش می‌دهد. این روش ساده‌ترین راه برای ایجاد و شروع یک Task است.

در بخش‌های بعدی چگونگی ایجاد Taskهای پیچیده‌تر را بررسی خواهیم کرد . خروجی برنامه بالا بصورت زیر خواهد بود:

```
Main method complete. Press enter to finish.

Hello World
```

روشهای مختلف ایجاد یک Task ساده :

- ایجاد کلاس Task با استفاده از یک متد دارای نام که در داخل یک کلاس Action صدا زده می‌شود. مثال :

```
Task task1 = new Task(new Action(printMessage));
```

استفاده از یک delegate ناشناس (بدون نام). مثال :

```
Task task2 = new Task(delegate {
   printMessage();
});
```

- استفاده از یک عبارت لامبدا و یک متد دارای نام . مثال :

```
Task task3 = new Task(() => printMessage());
```

- استفاده از یک عبارت لامبدا و یک متد ناشناس (بدون نام). مثال :

```
Task task4 = new Task(() => {
   printMessage();
});
```

قطعه کد زیر مثال خوبی برای چهار روشی که در بالا شرح دادیم می‌باشد:

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

namespace Listing_02 {

class Listing_02 {

static void Main(string[] args) {

   // use an Action delegate and a named method
   Task task1 = new Task(new Action(printMessage));

   // use a anonymous delegate
   Task task2 = new Task(delegate {
   printMessage();
});

  // use a lambda expression and a named method
  Task task3 = new Task(() => printMessage());

  // use a lambda expression and an anonymous method
  Task task4 = new Task(() => {
    printMessage();
  });
  task1.Start();
  task2.Start();
  task3.Start();
  task4.Start();

  // wait for input before exiting
```

```
  Console.WriteLine("Main method complete. Press enter to finish.");
  Console.ReadLine();
 }

 static void printMessage() {
  Console.WriteLine("Hello World");
  }
 }
}
```

خروجی برنامه بالا بصورت زیر است :

```
Main method complete. Press enter to finish.

Hello World

Hello World

Hello World

Hello World
```

نکته 1 : از مند استاتیک Task.Factory.StartNew برای ایجاد Task هایی که رمان اجرای کوتاه دارند استفاده می‌شود.

نکته 2 : اگر یک Taskدر حال اجرا باشد نمی‌توان آنرا دوباره استارت نمود باید برای یک نمونه جدید از آن Task ایجاد نمود و آنرا استارت کرد.

نظرات خوانندگان

نویسنده: رحمت رضایی
تاریخ: ۱۳۹۱/۰۳/۳۱ ۱۷:۵۱

از مند استاتیک Task.Factory.StartNew برای ایجاد Task هایی که زمان اجرای طولانی هم دارند استفاده می‌شود :

```
Task.Factory.StartNew(() =>
            {
                Thread.Sleep(1000);
            }, TaskCreationOptions.LongRunning);
```

نویسنده: ali
تاریخ: ۱۳۹۱/۰۳/۳۱ ۱۸:۳۷

سلام
مرسی از آموزشتون

این روش چه برتری نسبت به شیوه کلاسیک موازی کاری داره ؟
آیا همه امکانات شیوه کلاسیک رو پوشش می‌ده ؟

بی صبرانه منتظر ادامه آموزش هستم.
پیروز باشید.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۴/۰۱ ۱:۵۹

اگر منظور شما از روش‌های کلاسیک استفاده از Threadهاست باید بدانید که آن روش‌ها برای CPUهای تک هسته ای در نظر گرفته شده بودند. همانطور که می‌دانید در CPUهای تک هسته ای ، CPU تنها قادر به اجرای یک وظیفه در یک واحد زمان می‌باشد. در این CPUها برای اینکه بتوان چندین وظیفه را همراه با هم انجام داد CPU بین کارهای در حال انجام در بازه‌های زمانی مختلف سوییچ میکند و برای ما اینطور به نظر می‌آید که CPU در حال انجام چند وظیفه در یک زمان است.
اما در CPUها چند هسته ای امروزی هر هسته قادر به اجرای یک وظیفه به صورت مجزا می‌باشد و این CPUها برای انجام کارهای همزمان عملکرد بسیار بسیار بهتری نسبت به CPUهای تک هسته ای دارند.
با توجه به این موضوع برای اینکه بتوان از قابلیتهای چند هسته ای CPUهای امروزی استفاده کرد باید برنامه نویسی موازی (Parallel Programming) انجام داد و روش‌های کلاسیک مناسب این کار نمی‌باشند.

نویسنده: محمد
تاریخ: ۱۳۹۱/۰۴/۰۱ ۹:۴۸

ممنون

نویسنده: saleh
تاریخ: ۱۳۹۱/۰۴/۱۷ ۰:۱۲

شما در کد خودتون taskها را قبل از دستور چاپ متن main method ... نوشته و استارت داده بودید ولی در خروجی برعکس این موضوع اتفاق افتاده! میشه درموردش توضیح بدید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۴/۱۷ ۰:۳۰

توضیحات بیشتر در اینجا

عنوان: **اصول برنامه نویسی موازی درNET. نسخه 4 بخش اول - 2**
نویسنده: اشکان 56
تاریخ: ۱۳۹۱/۰۴/۰۱ ۱:۱۹
آدرس: www.dotnettips.info
برچسب‌ها: C#, Task Programming Language, Threading, Parallel Extensions

تنظیم وضعیت برای یک Task

در مثال ذکر شده در قسمت قبل هر چهار Task یک عبارت را در خروجی نمایش دادند حال می‌خواهیم هر Task پیغام متفاوتی را نمایش دهد.برای این کار از کلاس زیر استفاده می‌کنیم :

```
System.Action<object>
```

تنظیم وضعیت برای یک Task این امکان را فراهم می‌کند که بر روی اطلاعات مختلفی یک پروسه مشابه را انجام داد.

مثال :

```
namespace Listing_03 {
class Listing_03 {
  static void Main(string[] args) {
   // use an Action delegate and a named method
   Task task1 = new Task(new Action<object>(printMessage),"First task");

   // use an anonymous delegate
   Task task2 = new Task(delegate (object obj) {
     printMessage(obj);
   }, "Second Task");

   // use a lambda expression and a named method
   // note that parameters to a lambda don't need
   // to be quoted if there is only one parameter
   Task task3 = new Task((obj) => printMessage(obj), "Third task");

   // use a lambda expression and an anonymous method
   Task task4 = new Task((obj) => {
    printMessage(obj);
   }, "Fourth task");

  task1.Start();
  task2.Start();
  task3.Start();
  task4.Start();

  // wait for input before exiting
  Console.WriteLine("Main method complete. Press enter to finish.");
  Console.ReadLine();
}

 static void printMessage(object message) {
   Console.WriteLine("Message: {0}", message);
  }
 }
}
```

کد بالا را بروش دیگری هم می‌توان نوشت :

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

namespace Listing_04 {
class Listing_04 {
  static void Main(string[] args) {
   string[] messages = { "First task", "Second task",
    "Third task", "Fourth task" };

   foreach (string msg in messages) {
     Task myTask = new Task(obj => printMessage((string)obj), msg);
```

```
     myTask.Start();
  }

  // wait for input before exiting
  Console.WriteLine("Main method complete. Press enter to finish.");
  Console.ReadLine();
}

  static void printMessage(string message) {
   Console.WriteLine("Message: {0}", message);
  }
 }
}
```

نکته مهم در کد بالا تبدیل اطلاعات وضعیت Task به رشته کاراکتری است که در عبارت لامبدا مورد استفاده قرار می‌گیرد. System.Action فقط با داده نوع object کار می‌کند.

خروجی برنامه بالا بصورت زیر است :

```
Main method complete. Press enter to finish.

Message: Second task

Message: Fourth task

Message: First task

Message: Third task
```

البته این خروجی برای شما ممکن است متفاوت باشد چون در سیستم شما ممکن است Taskها با ترتیب متفاوتی اجرا شوند.با کمک Task Scheduler برا حتی می‌توان ترتیب اجرای Taskها را کنترل نمود

نظرات خوانندگان

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۴/۰۱ ۳:۳۲

در برنامه بالا ابتدا Taskها را Start کرده و سپس کد زیر اجرا می‌شود:

```
Console.WriteLine("Main method complete. Press enter to finish.");
```

سوال من اینه که چرا عبارت Main Method Complete.Press Enter to finish اول از همه در خروجی نمایش داده می‌شود؟!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۴/۰۱ ۹:۴۴

نوشتن متد Start به این معنا نیست که همین الان باید Start صورت گیرد. بعد Start دوم و بعد مورد سوم و الی آخر. پردازش موازی به همین معنا است و قرار است این موارد به موازات هم اجرا شوند و نه ترتیبی و پشت سر هم .
در یک برنامه کنسول، متد Main یعنی کدهایی که در ترد اصلی برنامه اجرا می‌شوند. زمان اجرای تمام taskهای تعریف شده، با زمان اجرای ترد اصلی برنامه بسیار بسیار نزدیک است اما ممکن است یک تاخیر چند میلی ثانیه‌ای اینجا وجود داشته باشد و آن هم وهله سازی و در صف قرار دادن taskها و اجرای آن‌ها است.
Task در دات نت 4 از thread pool مخصوص CLR استفاده می‌کند که همان thread pool ایی است که توسط متد ThreadPool.QueueUserWorkItem موجود در نگارش‌های قبلی دات نت، مورد استفاده قرار می‌گیرد؛ با این تفاوت که جهت کارکرد با Tasks بهینه سازی شده است (جهت استفاده بهتر از CPUهای چند هسته‌ای).
همچنین باید توجه داشت که استفاده از یک استخر تردها به معنای درصف قرار دادن کارها نیز هست. بنابراین یک زمان بسیار کوتاه جهت در صف قرار دادن کارها و سپس ایجاد تردهای جدید برای اجرای آن‌ها در اینجا باید درنظر گرفت.

یک منبع بسیار عالی برای مباحث پردازش موازی به همراه توضیحات لازم:
http://www.albahari.com/threading/part5.aspx#_Task_Parallelism

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۴/۰۱ ۱۶:۵۳

مرسی
خیلی مفید بود
اینطور که من فهمیدم CLR همه Taskهای Start شده را جمع آوری کرده و جهت اجرا درون یک صف قرار می‌دهد.
اما شما گفتید که قرار نیست کارها به ترتیب و پشت سر هم اجرا شوند! حال سوال اینجاست که هدف از درون صف قرار دادن Taskها چیست؟! مگر به صورت موازی اجرا نمیشوند؟!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۴/۰۱ ۱۷:۲۱

برای اینکه CPUها از لحاظ پردازش موازی دارای توانمندی‌های نامحدودی نیستند و لازم است مکانیزم صف وجود داشته باشد و همچنین برنامه شما تنها برنامه‌ای نیست که حق استفاده از توان پردازشی مهیا را دارد.

عنوان: **دسترسی به Collectionها در یک ترد دیگر در WPF**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۲/۰۷ ۱:۴۵
آدرس: www.dotnettips.info
گروه‌ها: WPF, Threading

اگر در WPF سعی کنیم آیتمی را به مجموعه اعضای یک Collection مانند یک List یا ObservableCollection از طریق تردی دیگر اضافه کنیم، با خطای ذیل متوقف خواهیم شد:

```
This type of CollectionView does not support changes to its SourceCollection
from a thread different from the Dispatcher thread
```

راه حلی که برای آن تا دات نت 4 در اکثر سایت‌ها توصیه می‌شد به نحو ذیل است:

Adding to an ObservableCollection from a background thread

مشکل!

اگر همین برنامه را که برای دات نت 4 کامپایل شده‌است، بر روی سیستمی که دات نت 4.5 بر روی آن نصب است اجرا کنیم، برنامه با خطای ذیل متوقف می‌شود:

```
System.InvalidOperationException: This exception was thrown because the generator for control
'System.Windows.Controls.ListView Items.Count:62' with name '(unnamed)' has received sequence of
CollectionChanged events that do not agree with the current state of the Items collection.
The following differences were detected:  Accumulated count 61 is different from actual count 62.
```

مشکل از کجاست؟

در دات نت 4 و نیم، دیگر نیازی به استفاده از کلاس MTObservableCollection یاد شده نیست و به صورت توکار امکان کار با Collectionها از طریق تردی دیگر میسر است. فقط برای فعال سازی آن باید نوشت:

```
private static object _lock = new object();
//...
BindingOperations.EnableCollectionSynchronization(persons, _lock);
```

پس از اینکه برای نمونه، مجموعه‌ی فرضی persons وهله سازی شد، تنها کافی است متد جدید EnableCollectionSynchronization بر روی آن فراخوانی شود.

برای برنامه‌ی دات نت 4 ایی که قرار است در سیستم‌های مختلف اجرا شود چطور؟

در اینجا باید از Reflection کمک گرفت. اگر متد EnableCollectionSynchronization بر روی کلاس BindingOperations یافت شد، یعنی برنامه‌ی دات نت 4، در محیط جدید در حال اجرا است:

```
public static void EnableCollectionSynchronization(IEnumerable collection, object lockObject)
{
      MethodInfo method = typeof(BindingOperations).GetMethod("EnableCollectionSynchronization",
            new Type[] { typeof(IEnumerable), typeof(object) });
      if (method != null)
      {
         method.Invoke(null, new object[] { collection, lockObject });
      }
}
```

در این حالت فقط کافی است این متد جدید یافت شده را بر روی Collection مدنظر فراخوانی کنیم.
همچنین اگر بخواهیم کلاس MTObservableCollection معرفی شده را جهت سازگاری با دات نت 4 و نیم به روز کنیم، به کلاس ذیل خواهیم رسید. این کلاس با دات نت 4 و 4.5 سازگار است و جهت کار با ObservableCollectionها از طریق تردهای مختلف

تهیه شده‌است:

```
using System;
using System.Collections;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Linq;
using System.Windows.Data;
using System.Windows.Threading;

namespace WpfAsyncCollection
{
    public class AsyncObservableCollection<T> : ObservableCollection<T>
    {
        public override event NotifyCollectionChangedEventHandler CollectionChanged;
        private static object _syncLock = new object();

        public AsyncObservableCollection()
        {
            enableCollectionSynchronization(this, _syncLock);
        }

        protected override void OnCollectionChanged(NotifyCollectionChangedEventArgs e)
        {
            using (BlockReentrancy())
            {
                var eh = CollectionChanged;
                if (eh == null) return;

                var dispatcher = (from NotifyCollectionChangedEventHandler nh in eh.GetInvocationList()
                                  let dpo = nh.Target as DispatcherObject
                                  where dpo != null
                                  select dpo.Dispatcher).FirstOrDefault();

                if (dispatcher != null && dispatcher.CheckAccess() == false)
                {
                    dispatcher.Invoke(DispatcherPriority.DataBind, (Action)(() =>
OnCollectionChanged(e)));
                }
                else
                {
                    foreach (NotifyCollectionChangedEventHandler nh in eh.GetInvocationList())
                        nh.Invoke(this, e);
                }
            }
        }

        private static void enableCollectionSynchronization(IEnumerable collection, object lockObject)
        {
            var method = typeof(BindingOperations).GetMethod("EnableCollectionSynchronization",
                                    new Type[] { typeof(IEnumerable), typeof(object) });
            if (method != null)
            {
                // It's .NET 4.5
                method.Invoke(null, new object[] { collection, lockObject });
            }
        }
    }
}
```

در این کلاس، در سازنده‌ی آن متد عمومی enableCollectionSynchronization فراخوانی می‌شود. اگر برنامه در محیط دات نت 4 فراخوانی شود، تاثیری نخواهد داشت چون method در حال بررسی نال است. در غیراینصورت، برنامه در حالت سازگار با دات نت 4.5 اجرا خواهد شد.

عنوان: **اثبات قانون مشاهده‌گر در برنامه نویسی**
نویسنده: میثم نوایی
تاریخ: ۱۳۹۳/۰۷/۲۷ ۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Threading, Random

امروز حین کدنویسی به یک مشکل نادر برخورد کردم. کلاسی پایه داشتم (مثلا Person) که یک سری کلاس دیگر از آن ارث بری میکردند (مثلا کلاس‌های Student و Teacher).در اینجا در کلاس پایه بصورت اتوماتیک یک ویژگی(Property) را روی کلاس‌های مشتق شده مقدار دهی میکردم؛ مثلا به این شکل:

```
public class Person
    {
        public Person()
        {
            personId= this.GetType().Name + (new Random()).Next(1, int.MaxValue);
        }
     }
```

سپس در یک متد مجموعه‌ای از Studentها و teacherها را ایجاد کرده و به لیستی از Personها اضافه میکنم:

```
var student1=new Student(){Name="Iraj",Age=21};
var student1=new Student(){Name="Nima",Age=20};
var student1=new Student(){Name="Sara",Age=25};
var student1=new Student(){Name="Mina",Age=22};
var student1=new Student(){Name="Narges",Age=26};
var teacher1=new Student(){Name="Navaei",Age=45};
var teacher2=new Student(){Name="Imani",Age=50};
```

اما در نهایت اتفاقی که رخ میداد این بود که PersonId همه Studentها یکسان می‌شد ولی قضیه به همین جا ختم نشد؛ وقتی خط به خط برنامه را Debug و مقادیر را Watch می‌کردم، مشاهده می‌کردم که PersonId به درستی ایجاد می‌شود.
در فیزیک نوین اصلی هست به نام عدم قطعیت هایزنبرگ که به زبان ساده میتوان گفت نحوه رخداد یک اتفاق، با توجه به وجود یا عدم وجود یک مشاهده‌گر خارجی نتیجه‌ی متفاوتی خواهد داشت.
کم کم داشتم به وجود قانون مشاهده‌گر در برنامه نویسی هم ایمان پیدا میکردم که این کد فقط در صورتیکه آنرا مرحله به مرحله بررسی کنم جواب خواهد داد!
جالب اینکه زمانیکه personId را نیز ایجاد میکردم، یک دستور برای دیدن خروجی نوشتم مثل این

```
public class Person
    {
        public Person()
        {
            personId= this.GetType().Name + (new Random()).Next(1, int.MaxValue);
            Debug.Print(personId)
        }
     }
```

در این حالت نیز دستورات درست عمل میکردند و personId متفاوتی ایجاد می‌شد! قبل از خواندن ادامه مطلب شما هم کمی فکر کنید که مشکل کجاست؟
این مشکل ربطی به قانون مشاهده‌گر و یا دیگر قوانین فیزیکی نداشت. بلکه بدلیل سرعت بالای ایجاد وهله ها(instance) از کلاسی‌های مطروحه (مثلا در زمانی کمتر از یک میلی ثانیه) زمانی در بازه یک کلاک CPU رخ می‌داد.

هر نوع ایجاد کندی (همچون نمایش مقادیر در خروجی) باعث می‌شود کلاک پردازنده نیز تغییر کند و عدد اتفاقی تولید شده فرق کند.
همچنین برای حل این مشکل میتوان از کلاس تولید کننده اعداد اتفاقی، شبیه زیر استفاده کرد:

```
using System;
using System.Threading;

public static class RandomProvider
```

```
{
    private static int seed = Environment.TickCount;

    private static ThreadLocal<Random> randomWrapper = new ThreadLocal<Random>(() =>
        new Random(Interlocked.Increment(ref seed))
    );

    public static Random GetThreadRandom()
    {
        return randomWrapper.Value;
    }
}
```

نظرات خوانندگان

نویسنده: رحمت اله رضایی
تاریخ: ۱۳۹۳/۰۷/۲۷ ۱۴:۴

به جای کلاس Random از جایگزین بهتر آن RNGCryptoServiceProvider استفاده کنید و دوباره برنامه رو تست کنید.

نویسنده: سعید
تاریخ: ۱۳۹۳/۰۸/۰۶ ۱۲:۴۸

شما اگر برای تولید عدد تصادفی از یک آبجکت کلاس Random استفاده می‌کردید به چنین مشکلی بر نمی‌خوردید.

نویسنده: میثم نوایی
تاریخ: ۱۳۹۳/۰۸/۰۶ ۱۳:۲

اگه مطلب را کامل مطالعه بفرمایید و شبیه سازی کنید به مشکل مطروحه برخواهید خورد.

نویسنده: سعید
تاریخ: ۱۳۹۳/۰۸/۰۶ ۱۴:۱۵

من نیاز مسئله شما را به صورت زیر نوشتم و برای هر شی Person یک مقدار متفاوت دارم.

```
class Program
    {
        static void Main(string[] args)
        {
            var lst = new List<Person>();
            Random rnd = new Random();
            for (int i = 0; i < 50; i++)
            {
                lst.Add(new Person(rnd));
            }


            foreach (var item in lst)
            {
                Console.WriteLine(item.PersonId);
            }

            Console.ReadKey();
        }
    }

    public class Person
    {
        public Person(Random rnd)
        {
            PersonId = this.GetType().Name + rnd.Next(1, int.MaxValue);
        }

        public string PersonId { get; set; }
    }
```

البته من فکر میکنم اگر شما نیاز به یک ای دی منحصر به فرد دارید راه بهتر استفاده از GUID باشد.

نویسنده: میثم نوایی
تاریخ: ۱۳۹۳/۰۸/۰۶ ۱۴:۴۱

ممنون بابت نظراتتان.
عرض کردم مطلب را کامل بخوانید.این مشکل در سیستم هایی با پردازش بالا و در زمان‌های هزارم ثانیه رخ میدهد.به این معنی که کلاس Random مقادیر متفاوتی ایجاد نمیکند.و عملا کارایی ندارد.

عنوان: **پشتیبانی توکار از ایجاد کلاس‌های Singleton از دات نت 4 به بعد**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۰۲ ۱۲:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Threading

روش‌های زیادی برای ایجاد یک وهله‌ی Singleton وجود دارند. وهله‌ای که در طول عمر یک برنامه، تنها یکبار ایجاد شده و حفظ می‌شود. برای مثال شاید متداول‌ترین حالت آن که در بسیاری از کدها دیده می‌شود، تعریف یک متغیر استاتیک در کلاس، غیرعمومی تعریف کردن سازنده‌ی کلاس و وهله سازی این فیلد استاتیک در صورت نال بودن آن است:

```
public class WrongSingleton
    {
        static WrongSingleton _instance;

        WrongSingleton()
        {
        }

        public static WrongSingleton Instance
        {
            get { return _instance ?? (_instance = new WrongSingleton()); }
        }
    }
```

هرچند این روش کار می‌کند اما thread-safe نیست. به این معنا که ممکن است دو ترد در آن واحد به بررسی قسمت ?? _instance بپردازند و چون هنوز نال است، دوبار وهله سازی کلاس، با فراخوانی new WrongSingleton صورت خواهد گرفت و هر ترد در آن لحظه به وهله‌ی متفاوتی دسترسی خواهد داشت.
راه حل‌های زیادی برای رفع این مشکل با اعمال مباحث قفل گذاری تا نکات ریز مربوط به کامپایلر وجود دارند که لیست آن‌ها را در اینجا می‌توانید مطالعه کنید.

از دات نت 4 به بعد با معرفی الگوی Lazy ، امکان پیاده سازی lazy thread safe singletons به صورت توکار در دسترس می‌باشد. نمونه‌ای از آن در کدهای IoC Container معروفی به نام StructureMap بکار رفته‌است :

```
public class Container
    {
        // ...
    }

    public static class ObjectFactory
    {
        private static readonly Lazy<Container> _containerBuilder =
            new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

        public static Container Container
        {
            get { return _containerBuilder.Value; }
        }

        private static Container defaultContainer()
        {
            return new Container();
        }
    }
```

در اینجا کلاس ObjectFactory یک وهله از کلاس Container را در اختیار مصرف کننده قرار می‌دهد؛ با این شرایط:
- چون این وهله توسط کلاس Lazy محصور شده‌است، صرفا در اولین بار دسترسی به آن، نمونه سازی خواهد شد. این مورد سبب کاهش مصرف حافظه‌ی برنامه و همچنین بالا رفتن سرعت برپایی اولیه‌ی آن می‌شود. بسیاری از اشیایی که در یک برنامه تعریف می‌شوند، شاید الزاما جهت ارائه راه حلی برای مساله‌ای خاص، مورد استفاده قرار نگیرند. تعریف آن‌ها به صورت Lazy، سربار نمونه سازی الزامی آن‌ها را حذف خواهد کرد و آن‌را به اولین بار استفاده از شیء مورد نظر، به تعویق خواهد انداخت.
- با استفاده از LazyThreadSafetyMode.ExecutionAndPublication، چندین ترد می‌توانند به خاصیت Container دسترسی پیدا کنند، اما تنها یکی از آن‌ها موفق به وهله سازی این کلاس خواهد شد. البته حالت ExecutionAndPublication، حالت پیش فرض

است و الزاما نیازی به ذکر آن نیست.

اینبار بازنویسی کلاس ابتدای بحث با توجه به نکات ذکر شده به صورت زیر خواهد بود:

```
public sealed class LazySingleton
    {
        private static readonly Lazy<LazySingleton> _instance =
            new Lazy<LazySingleton>(() => new LazySingleton(),
LazyThreadSafetyMode.ExecutionAndPublication);

        private LazySingleton()
        {
        }

        public static LazySingleton Instance
        {
            get { return _instance.Value; }
        }
    }
```

- در آن سازنده‌ی کلاس، خصوصی تعریف شده‌است.
- تنها وهله‌ی در دسترس کلاس، به صورت استاتیک و نمونه سازی کلاس، توسط کلاس Lazy با پارامتر LazyThreadSafetyMode.ExecutionAndPublication انجام می‌شود.
- علت استفاده از lambda در سازنده‌ی کلاس Lazy، امکان دسترسی به اعضای private کلاس، از طریق یک خاصیت static است.