



Parallel and Asynchronous programming

.NET

**DE HOGESCHOOL
MET HET NETWERK**

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook



Contents

- Plain Threads
- Parallel vs Asynchronous
- TPL
- async / await



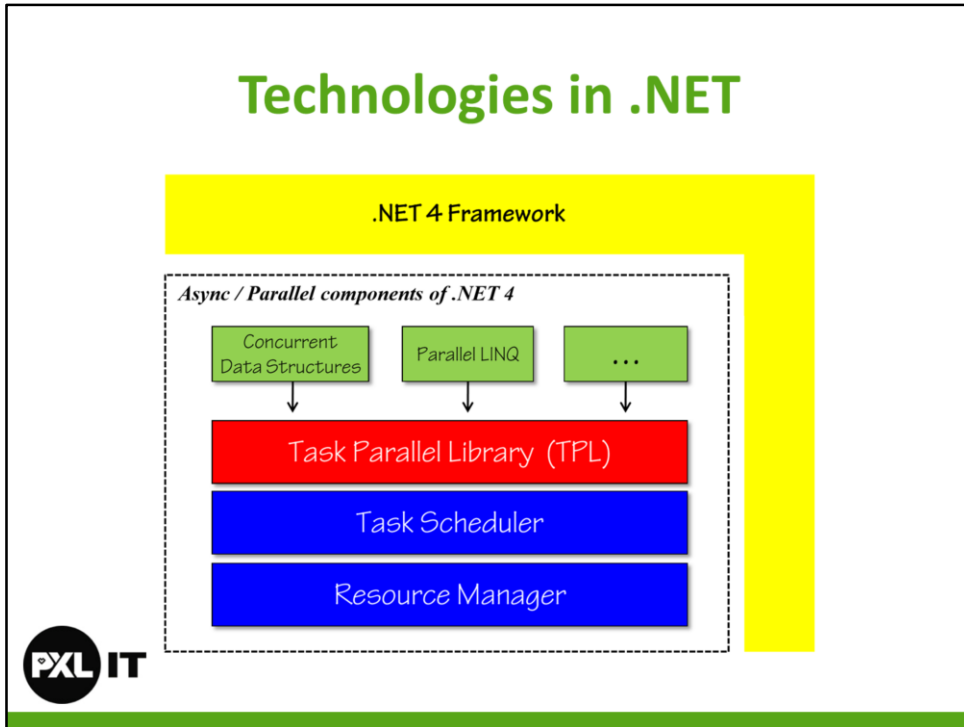
This module uses multithreading techniques to realize parallel and asynchronous tasks. First we introduce plain threads and explain why they are not enough to write complex programs. Then we introduce the concepts of “Parallel” and “Asynchronous” programming. The .NET framework introduced the Task Parallel Library (TPL) in .NET 4.0 to realize both parallel and asynchronous programming. However using the latest C# keywords `async` and `await`, the asynchronous part becomes much more readable. However this hides much of its complexity.

Working with tasks

- Creating
- Waiting
- Harvesting Results



Technologies in .NET



TPL provides the framework you use as a programmer ($\text{Task}<\text{T}>$).

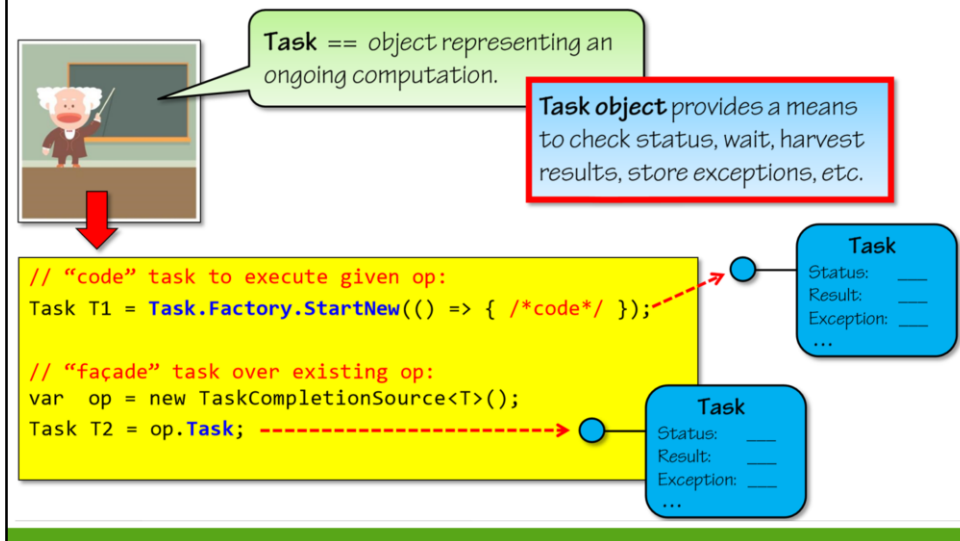
Task Scheduler maps tasks to threads.

Resource manager is responsible for managing the pool of worker threads.

Concurrent Data Structures: Queue, Bag, Dictionary

Parallel LINQ: concurrent execution of queries

Review – what's a task?



Task objects hold properties for the State of a task:

- Status
- Result
- Exception

These act as placeholder and are filled in by the executing task when available or upon completion.

There are 2 types of tasks:

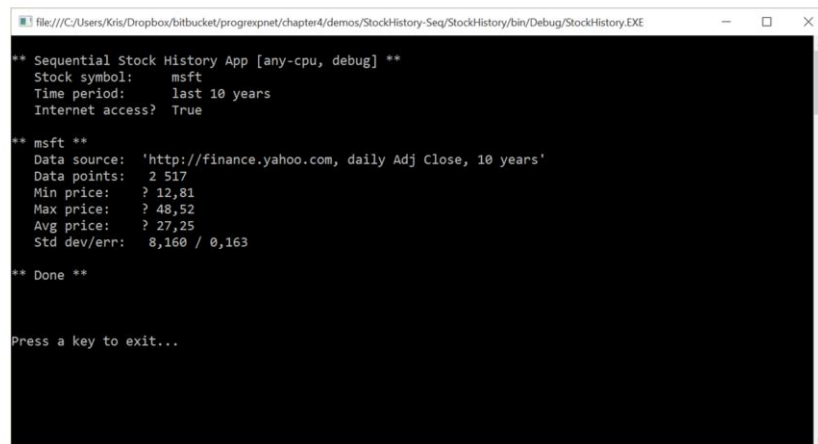
Code tasks → for parallel operations

- Created with Task.Factory.StartNew()
- Scheduled on a separate thread
- Has some code to be executed, typically with a lambda

Façade tasks → for asynchronous operations

- No code during task creation, but:
- The computation is specified elsewhere and already running (e.g. network request)
- You create a TaskCompletionSource object and no Task object itself
- They are a façade over an existing (asynchronous) operation

Demo: Stock History



```
file:///C:/Users/Kris/Dropbox/bitbucket/progrexpnet/chapter4/demos/StockHistory-Seq/StockHistory/bin/Debug/StockHistory.EXE

** Sequential Stock History App [any-cpu, debug] **
Stock symbol:    msft
Time period:     last 10 years
Internet access? True

** msft **
Data source: 'http://finance.yahoo.com, daily Adj Close, 10 years'
Data points:    2 517
Min price:      ? 12,81
Max price:      ? 48,52
Avg price:      ? 27,25
Std dev/err:    8,160 / 0,163

** Done **

Press a key to exit...
```



Demo: StockHistory-Seq

Note: Pluralsight code must be adapted

- American number notation
- MSN stock API does not work anymore

Demo: Stock History – step 1

- Parallelize front-end

```
decimal min = 0, max = 0, avg = 0;
Task t_min = Task.Run(() =>
{
    min = data.Prices.Min();
});

Task t_max = Task.Run(() =>
{
    max = data.Prices.Max();
});

Task t_avg = Task.Run(() =>
{
    avg = data.Prices.Average();
});
```

```
Console.WriteLine();
Console.WriteLine("*** {0} **", symbol);
Console.WriteLine("  Data source: '{0}'", data.DataSource);
Console.WriteLine("  Data points: {0:#,##0}", N);

t_min.Wait();
Console.WriteLine("  Min price: {0:C}", min);

t_max.Wait();
Console.WriteLine("  Max price: {0:C}", max);

t_avg.Wait();
Console.WriteLine("  Avg price: {0:C}", avg);
Console.WriteLine("  Std dev/err: {0:0.000} / {1:0.000}", s,
```

Demo: StockHistory-Step1

Create a Task around every calculation.

Wait for them to finish before outputting.

Note: Task.Factory.StartNew may be replaced (more efficiently) by Task.Run (introduced in .NET 4.5)

There are subtle differences, references:

<http://jeremybytes.blogspot.be/2015/02/taskrun-vs-taskfactorystartnew.html>

<http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229468.aspx>

Demo: Stock History – step 2

- Read code from **GetDataFromInternet**
- This illustrates a **TaskCompletionSource**
 - Asynchronous downloading of the internet
 - Attach a Task to this (older) code
 - Wait for it to complete



Demo: StockHistory-Step2

The code in `GetDataFromInternet` is written before .NET 4 (with TPL) came out: `IAAsyncResult` and `WaitHandle` are other techniques to create async requests.

How to

- Wait for a task to finish
- Wait for one or more tasks to finish
- Return a value from a task
- Compose tasks



The TPL provides several methods to synchronize tasks. You can:

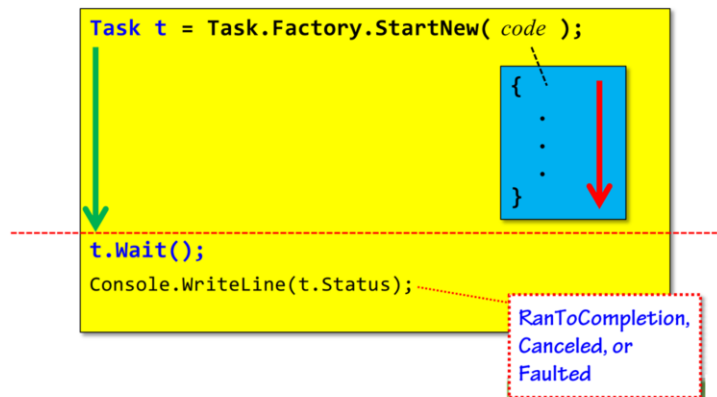
- Wait for a task to finish
- Wait for one or more tasks to finish
- Retrieve a result from a task
- Compose tasks

We review these methods in the slides. For the demo's, you can watch the second module of this Pluralsight course:

<http://www.pluralsight.com/courses/intro-async-parallel-dotnet4>

Wait

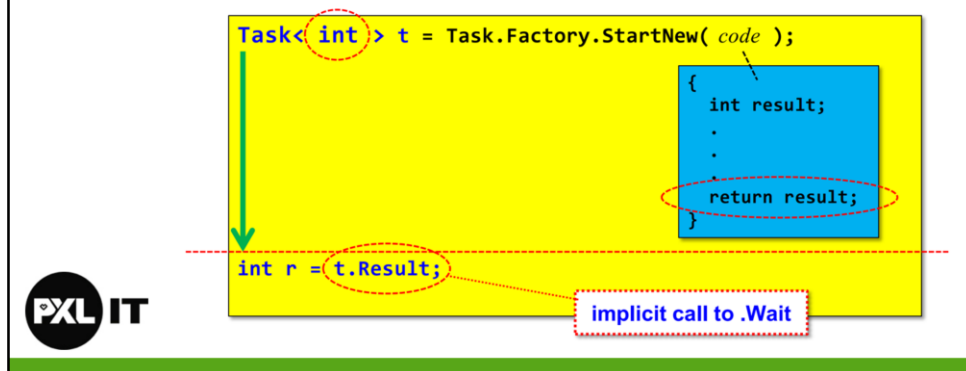
- Need to wait for a task to finish?
- Call **.Wait** on task object



The **Wait()**-method blocks the calling thread until the task finishes.

Harvesting results

- Is task computing a result?
- Specify type when creating a task
- Harvest value via **.Result**



Result calls **Wait** implicitly, so you wait until there is a result from the task (or an exception).

Waiting on multiple tasks

- What's the best way to wait for a **set** of tasks to finish?
 - Calling `.Wait` on each task implies an ordering that might not hold
- What's the best way to wait for the **first** task to finish?
 - E.g. parallel search of multiple sites, first one wins

```
Task t1 = Task.Factory.StartNew( code );  
Task t2 = Task.Factory.StartNew( code );  
Task t3 = Task.Factory.StartNew( code );
```

```
Task[] tasks = { t1, t2, t3 };  
.  
.
```

```
Task.WaitAll( tasks ); // wait for ALL to finish:
```

```
int index = Task.WaitAny( tasks ); // wait for FIRST to finish:  
Task first = tasks[index];
```

Task composition

- The completion of a task triggers the start of another

```
Task T1 = Task.Factory.StartNew( () =>
{
    :
    :
}
);

Task T2 = T1.ContinueWith( (antecedent) =>
{
    :
    :
}
);
```

parameter denotes task that just completed — i.e. T1 in this case

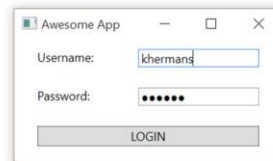
Implicit .Wait — T2 does not start until T1 completes

Why? Allows .NET to optimize scheduling...

You *could* potentially wait on a previous task to start a next one, but from a scheduling point of view, this is not optimal. Instead use the built-in method **ContinueWith** on the first task to start the next.

Going towards async / await

- TPL
 - Fine for parallel tasks
 - Reasoning on higher level than threads
 - Messy code for asynchronous code
- Demo: MyLogin-Sync



Demo: MyLogin-Sync

This demo uses simply `Thread.Sleep()` to simulate a potentially long login process, with a call to a backend etc.

In the next steps we rework this sample to use an asynchronous task to make the UI responsive. However, the code will grow complex very quickly.

In .NET 4.5 this resulted in the addition of the **async / await** keywords.

This demo comes from module 3 of the pluralsight course:

<http://www.pluralsight.com/training/player?course=asynchronous-programming-dotnet-getting-started&author=filip-ekberg&name=asynchronous-programming-dotnet-getting-started-m3&clip=1&mode=live>

Demo: MyLogin-TPL-Step1

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    LoginButton.IsEnabled = false;
    var task = Task.Run(() =>
    {
        Thread.Sleep(5000);
    });

    task.ContinueWith((t) =>
    {
        Dispatcher.Invoke(() =>
        {
            LoginButton.IsEnabled = true;
        });
    });
}
```



Demo: MyLogin-TPL-Step1

Create a Task, use the more up-to-date version **Task.Run** instead of **Task.Factory.CreateNew**

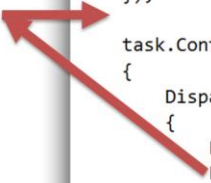
Enable button again when this task is complete. You use **ContinueWith**, but be aware to run this code on the UI thread, so therefore **Dispatcher.Invoke()**. This second task is called a **continuation task**.

Note: other solution:

```
task.ContinueWith((t) =>
{
    LoginButton.IsEnabled = true;
}, TaskScheduler.FromCurrentSynchronizationContext());
```


Demo: MyLogin-TPL-Step2

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    LoginButton.IsEnabled = false;
    var task = Task<string>.Run(() =>
    {
        Thread.Sleep(5000);
        return "Login succeeded!";
    });
    task.ContinueWith((t) =>
    {
        Dispatcher.Invoke(() =>
        {
            LoginButton.IsEnabled = true;
            LoginButton.Content = t.Result;
        });
    });
}
```



Demo: MyLogin-TPL-Step2

Let the “Sleep”-task (login routine) return a value: “Login succeeded” and put it onto the button.

Question: move `LoginButton.Content = t.Result` to the arrow. What happens? Can you explain?

Demo: MyLogin-TPL-Step3

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    LoginButton.IsEnabled = false;
    var task = Task<string>.Run(() =>
    {
        Thread.Sleep(5000);
        throw new UnauthorizedAccessException();
        return "Login succeeded!";
    });

    task.ContinueWith((t) =>
    {
        if (t.IsFaulted)
        {
            Dispatcher.Invoke(() =>
            {
                LoginButton.IsEnabled = true;
                LoginButton.Content = "Login failed";
            });
        }
        else
        {
            Dispatcher.Invoke(() =>
            {
                LoginButton.IsEnabled = true;
                LoginButton.Content = t.Result;
            });
        }
    });
}
```



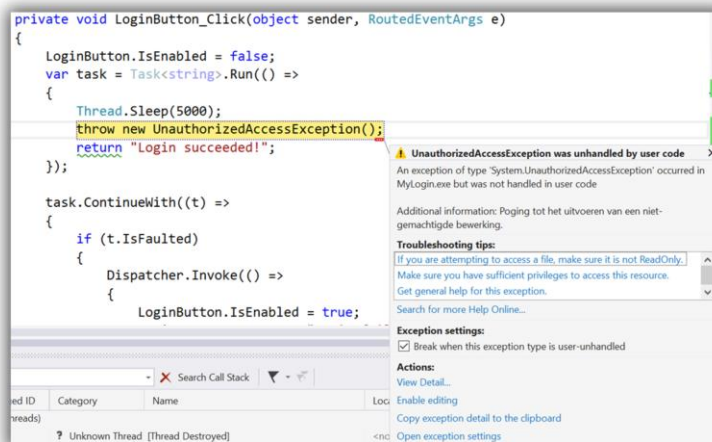
Demo: MyLogin-TPL-Step3

What happens if the login routine throws an exception? Simulate this using an `UnauthorizedException`.

A task effectively swallows this exception, so you as a programmer are responsible for checking the outcome of a task. E.g. **IsFaulted**

Note: this little update in requirements results in more messy code...

Demo: MyLogin-TPL-Step3



Demo: MyLogin-TPL-Step3

Note: VS2015 breaks on the exception, uncheck the box "break when this exception type is user-handled"

More info: <https://msdn.microsoft.com/en-us/library/x85tt0dd.aspx>

Demo: MyLogin-TPL-Step4

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    LoginButton.IsEnabled = false;
    var task = Task<string>.Run(() =>
    {
        Thread.Sleep(5000);
        return "Login succeeded!";
    });

    task.ConfigureAwait(true)
        .GetAwaiter()
        .OnCompleted(() =>
        {
            LoginButton.Content = task.Result;
            LoginButton.IsEnabled = true;
        });
}
```



Demo: MyLogin-TPL-Step4

How to avoid the ContinueWith (continuation) task?

Task.ConfigureAwait(true)

[https://msdn.microsoft.com/en-us/library/hh194876\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/hh194876(v=vs.110).aspx)

in plain english: we will try to execute the continuation task (later, hence an awaiter) on the original context, thus the UI thread.

GetAwaiter() → get the waiting object that waits on the task

[https://msdn.microsoft.com/en-us/library/hh139083\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/hh139083(v=vs.110).aspx)

OnCompleted → code to execute after the waiting is finished

[https://msdn.microsoft.com/en-us/library/hh138394\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/hh138394(v=vs.110).aspx)

Remark: this is solely for demonstration purposes and for understanding the **await** keyword.

Demo: MyLogin-TPL-Step5

```
private async void LoginButton_Click(object sender, RoutedEventArgs e)
{
    await Task.Run(() => Thread.Sleep(5000));
    LoginButton.Content = "Login succeeded.";
}
```

Runs as a **continuation** of the awaited Task above it



Demo: MyLogin-TPL-Step5

Mark the LoginButton_Click as async, now inside the body you can write asynchronous code, but not automatically! E.g. Thread.Sleep(5000) still blocks the UI thread.

The **async** keyword makes it possible to create **continuations** in a readable manner.

Now wrap the Thread.Sleep in a Task.Run() → VS suggest to put in an await because the task will run in a separate thread. Now if you run the program, the UI is responsive (drag the window).

IMPORTANT:

Everything below "await Task.Run" is executed inside a continuation of that task, inside the thread of the calling context => in this case the UI thread.

Demo: MyLogin-TPL-Step6

```
var t = Task<string>.Run(() =>
{
    Thread.Sleep(5000);
    return "Login success";
});
await t;
LoginButton.Content = t.Result;
```

```
var result = await Task.Run(() =>
{
    Thread.Sleep(5000);
    return "Login success";
});
LoginButton.Content = result;
```



Demo: MyLogin-TPL-Step6

Modify so the task returns a string result that is placed on the button.
The two code blocks on the slide do the same thing.

Demo: MyLogin-TPL-Step7/8

```
private async void LoginAsync()
{
    //throw new UnauthorizedAccessException();
    var result = await Task.Run(() =>
    {
        //throw new UnauthorizedAccessException();
        Thread.Sleep(5000);
        return "Login succes";
    });
    LoginButton.Content = result;
}
```

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        LoginAsync();
    }
    catch (Exception)
    {
        // does not get hit!!
    }
}
```



Demo: MyLogin-TPL-Step7 and MyLogin-TPL-Step8

First we refactor the “login” routine to a separate method → private async void LoginAsync

Notice the naming convention: if you have a method **Foo** that uses async, then you call it **FooAsync**

Now if you throw an exception inside this method (to simulate a login failure), the method does not get hit! How come?

Demo: MyLogin-TPL-Step9

```
private async Task LoginAsync()
{
    throw new UnauthorizedAccessException();
    var result = await Task.Run(() =>
    {
        //throw new UnauthorizedAccessException();
        Thread.Sleep(5000);
        return "Login succes";
    });
    LoginButton.Content = result;
}

private async void LoginButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        await LoginAsync();
    }
    catch (Exception)
    {
        LoginButton.Content = "Login Failed!";
    }
}
```



Demo: MyLogin-TPL-Step9

The LoginAsync method returns a Task which can be awaited → now the Task status gets revealed, including possible exceptions.

So: ALWAYS await your tasks!!

Demo: MyLogin-TPL-Step10

```
private async Task<string> LoginAsync()
{
    try
    {
        var result = await Task.Run(() =>
        {
            Thread.Sleep(5000);
            return "Login success";
        });
        return result;
    }
    catch (Exception)
    {
        return "Login failed!";
    }
}
```

```
try
{
    LoginButton.IsEnabled = false;
    BusyIndicator.Visibility = Visibility.Visible;

    string result = await LoginAsync();

    LoginButton.Content = result;
    LoginButton.IsEnabled = true;
    BusyIndicator.Visibility = Visibility.Hidden;
}
catch (Exception)
{
    LoginButton.Content = "Login Failed!";
}
```



Demo: MyLogin-TPL-Step10

This demo demonstrates a far more responsive gui.

By using async / await → code becomes much cleaner!

Demo: MyLogin-TPL-Step11

```
private async Task<string> LoginAsync()
{
    try
    {
        var result = await Task.Run(() =>
        {
            Thread.Sleep(5000);
            return "Login success";
        });
        // UI
        await Task.Delay(2000); // log login to DB
        // UI
        await Task.Delay(1000); // Fetch purchases
        // UI
        return result;
    }
    catch (Exception)
    {
        return "Login failed!";
    }
}
```



Demo: MyLogin-TPL-Step11

Now introduce some extra steps (`Task.Delay`) that needs to be done in a login procedure. The “//UI” denotes the continuation task that executes inside the UI thread.

However, these continuations make the code execute sequentially. This is solved in the next and last demo.

Demo: MyLogin-TPL-Step12

```
try
{
    var loginTask = Task.Run(() =>
    {
        Thread.Sleep(5000);
        return "Login success";
    });

    var logTask = Task.Delay(2000); // log login to DB
    var fetchTask = Task.Delay(1000); // Fetch purchases

    await Task.WhenAll(loginTask, logTask, fetchTask);

    return loginTask.Result;
}
catch (Exception)
{
    return "Login failed!";
}
```



Demo: MyLogin-TPL-Step12

By doing a `Task.WhenAll` you wait until all tasks finish (in parallel). The `loginTask.Result` does not block, because the task has already finished on that point.