Language execution basics

.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

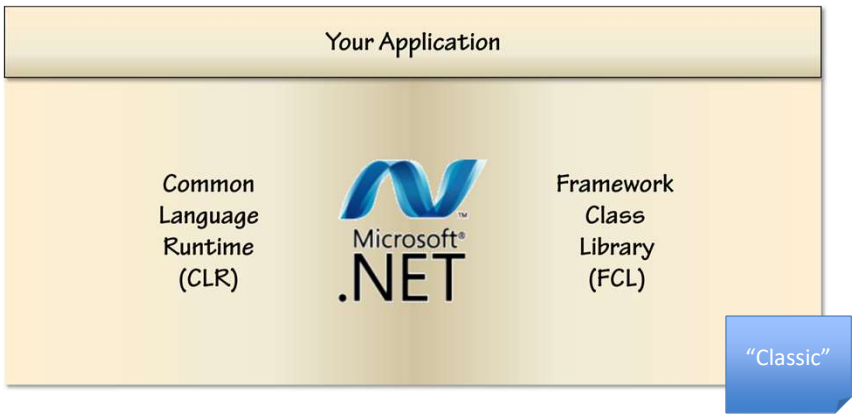
- What is .NET exactly?
  - The Common Language Runtime (CLR)
  - The Framework Class Libraries (FCL)
  - Intermediate Language (IL)
- C# in depth (some examples)
  - C# is strongly typed
  - struct vs class (stack vs heap)
  - Span
- Benchmarking / profiling

PXL IT

This module focuses on the "Virtual Machine" that runs .NET and we start to add some skipped language constructs in C# in order to write more advanced programs. Finally we focus on how to measure performance.

What is .NET?

- A Software framework



Common Language Runtime => Virtual Machine

Framework Class Library => Supporting classes upon which you can build your programs

Types of programs:

- Standalone (desktop)
- Apps
- Web
- Windows Services
- …

Correct version of .NET needs to be installed on the target machine. The shipping version often needs to be updated.

Note: this is the "Classical" framework. We will discuss "the future of .NET" further on.

CLR virtualizes execution and manages it, so you as a developer doesn't have to worry about things like memory management.

.NET languages currently supported: C#, F#, VB, C++ and Python

OS

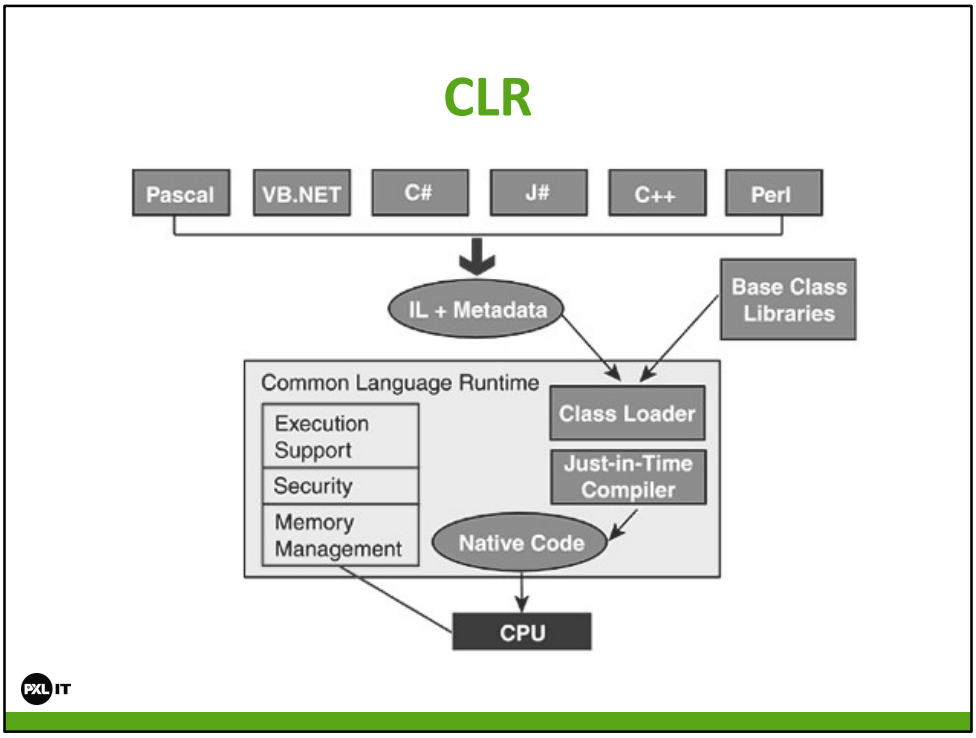
- Windows 95 → Windows 10
- Windows Phone
- Mono: Linux and Mac
- Xamarin: iOS and Android

FCL

- Framework Class Library
  - A library of functionality to build applications
- BCL (Base Class Library)
  - Subset that works everywhere
  - Will be replaced by CoreFx
- Networking, UI, Web, etc

Interesting to follow:

http://blogs.msdn.com/b/dotnet/
https://bcl.codeplex.com/ → https://github.com/dotnet/corefx



The CLR defines a common programming model and a standard type system for cross-platform, multi-language development.

All .NET-aware compilers generate Intermediate Language (IL) instructions and metadata.

The runtime's Just-in-Time (JIT) compiler converts the IL to a machine-specific (native) code when an application actually runs.

Because the CLR is responsible for managing this IL, the code is known as managed code.

Intermediate Language (IL)

- Inspecting IL
  - Virtual Machine Language of the CLR
  - Target language of C#, Visual Basic, etc
  - JIT compiled into native code
- ILDASM
  - Ships with VS
  - IL roundtripping with ILASM (textual language for IL)
- Other tools: .NET Reflector and ILSpy

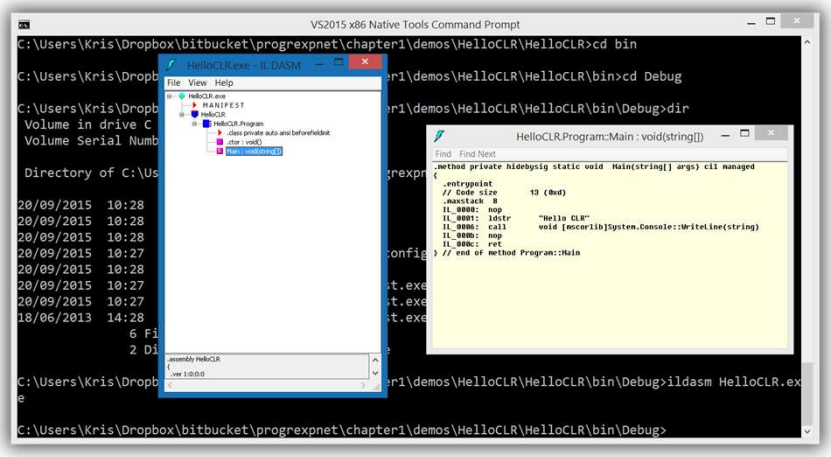
It's interesting to have a basic understanding of the mechanics of IL. There is a basic tool called ILDASM that ships with VS and generates a textual representation of IL. You can actually modify this code and generate IL back (= roundtripping) with ILASM.

Other tools to explore: .NET reflector and ILSpy

https://www.red-gate.com/products/dotnet-development/reflector/ (not free)

http://ilspy.net/ (open source)

Demo ildasm



More info:

https://docs.microsoft.com/en-us/dotnet/framework/tools/ildasm-exe-il-disassembler

https://docs.microsoft.com/en-us/dotnet/framework/tools/developer-command-prompt-for-vs

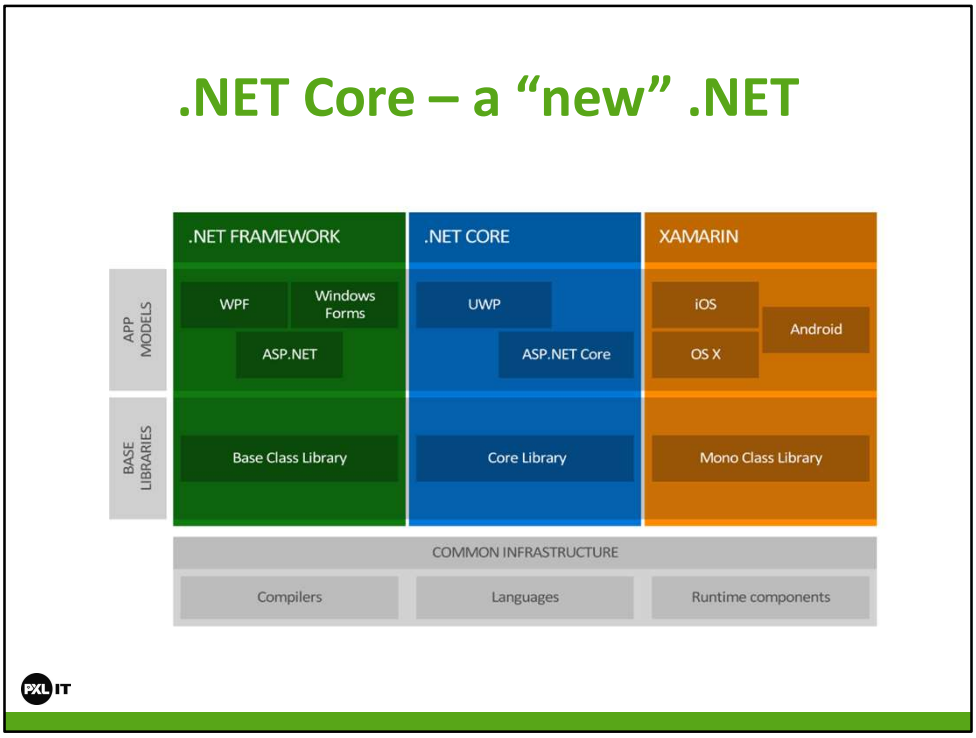
Search for the VS2017 x86 Native Tools Command Prompt, it should be installed together with Visual Studio. This command prompt has several expert tools in its PATH variable, like ildasm.exe

Once opened, go to the Debug folder of your project and execute the command:

ildasm.exe HelloCLR.exe

Ildasm.exe HelloCLR.dll (for .NET Core apps)

Exercise: tryout IlSpy with a Console program that calls a method.



Traditionally, there was the “.NET Framework”:

- You build desktop apps (WPF or Winforms) and web apps (ASP.NET)
- On Windows machines targeting windows machines
- Using the BCL

Then there was Mono, an open source implementation of .NET running on Linux and Mac. From this foundation, Xamarin was build. The “Mono Class Library” is a port of the BCL to the mono runtime. Not everything exists there (like WPF and Winforms).

Now recently “.NET Core” is released, yet again a new beginning:

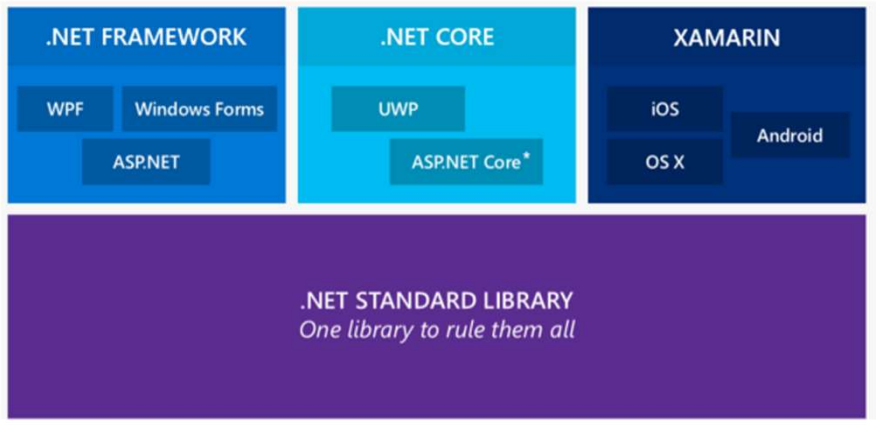
- A .NET runtime running on all major platforms
- Another framework library implementation
- A set of tools (compilers and application host)

Reference (old post):

http://www.telerik.com/blogs/the-new-net-core-1-is-here

.NET Core - .NET Standard

At Build 2016, Scott Hunter presented the following slide:



One library to rule them all, where you don't have to worry about platform incompatibilities.

Reference

https://docs.microsoft.com/en-us/dotnet/standard/net-standard

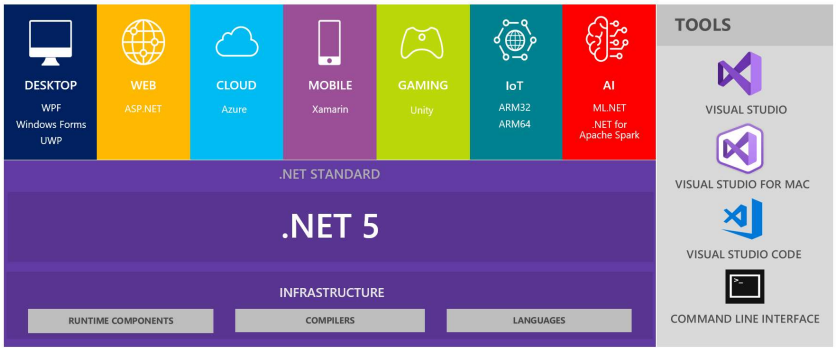
In the above reference, the table shows which runtime platform implements which version of .NET Standard.

Key point to take away: if you use a library in your project, choose a .NET Standard library, not a Portable Class Library. If you are a library author, create .NET Standard libraries as your library will be supported on more platforms.

Demo: .NET Core on linux or Mac

Introducing .NET 5

.NET – A unified platform



https://devblogs.microsoft.com/dotnet/introducing-net-5/

https://devblogs.microsoft.com/dotnet/introducing-net-5/

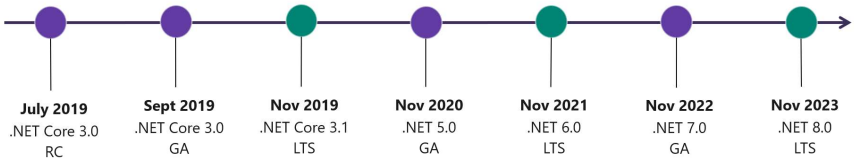
In November 2020, there will be only 1 .NET going forward. You will be able to use it to target Windows, Linux, macOS, iOS, Android, tvOS, watchOS and WebAssembly and more.

.NET 5 = .NET Core vNext

- Produce a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences.
- Expand the capabilities of .NET by taking the best of .NET Core, .NET Framework, Xamarin and Mono.
- Build that product out of a single code-base that developers (Microsoft and the community) can work on and expand together and that improves all scenarios.

.NET Schedule

.NET Schedule



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

PXL IT

C# is strongly typed

- Every variable should be declared with a type
- Assigning one type to another requires conversion
  - string to int
- Primitive types: int, long, etc.
- Creating a class means defining a new type

Demo: typeconversion

Questions:

- Where are types converted between each other?
- Run this from the console, where is the executable located?
- Create a .NET Core app, how do you run this from the command line?

The var keyword

- Only for local vars
- The type of the variable will (and must) be determined from the initialisation

```
var firstname = Console.ReadLine();

var firstname = "Kris";
```

➔ C# remains strongly typed, the type of the variable is clear at compile-time!

The keyword is only usable for local variables, so not for member variables etc.

So illegal statements are:

var firstname;

var firstname = null;

…

Why is this introduced? → Readability for long classnames or classnames with generics, eg:

ObservableCollection<Employee> list = new ObservableCollection<Employee>();

becomes:

var list = new ObservableCollection<Employee>();

Watch out for readability issues:

```
var a = 12; // int
var b = 123456778990; // long
```

Reference types

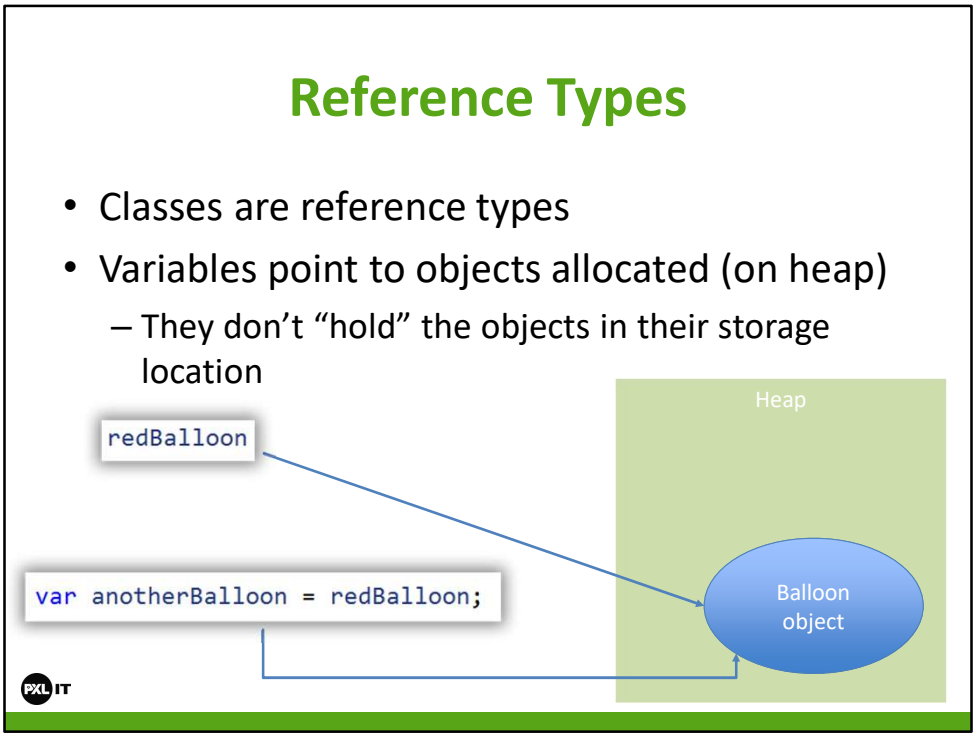
- Denoted by “class”

```
public class Balloon
{
    public double Size { get; set; }
    public string ColorHex { get; set; }
}
```

Reference types

```
[Test]
public void TwoBalloonReferencesSameObject()
{
    var redBalloon = new Balloon()
    {
        ColorHex = "#FF0000",
        Size = 15.0
    };

    var anotherBalloon = redBalloon;

    Assert.AreSame(anotherBalloon, redBalloon);
}
```

Demo: RefsVsValue.BalloonTests



https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types

There are two reference variables (pointers) to the SAME Balloon object.

If you write a class, you create a new reference type.

Note that objects are always allocated on a special storage location called the heap.

Value Types

- Variables hold the value
  - No pointers, no references
- Many built-in primitives are value types
  - Int (System.Int32), DateTime, double, float
- You can make your own with the `struct` keyword
- Value types are fast to instantiate

Reference: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types

Every type is either a reference type, or a value type

A variable of a Value type, holds the actual value.

e.g. int y = 32 ➔ y holds the actual value 32 and not a reference to it.

If you assign a variable to another, you COPY the value

e.g: int x = y → you COPY the value of 32 and assign it to x

Because you don't hold a reference but the actual value, these types are fast to instantiate.

Value Types

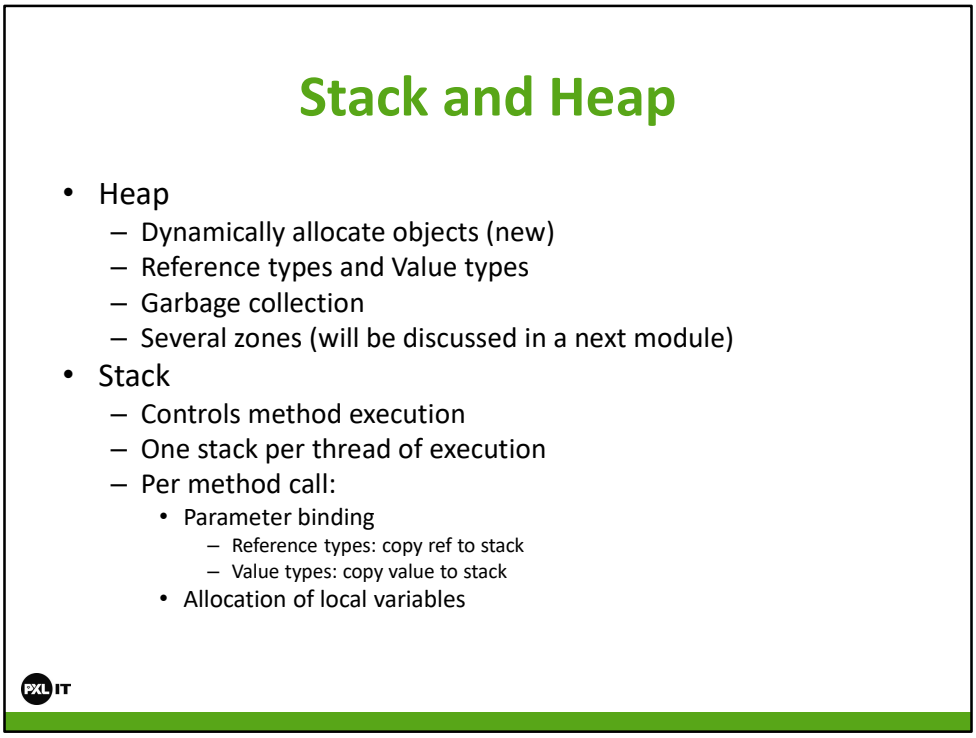
```
[Test]
public void TwoIntsAreNotTheSame()
{
    int x = 32;
    int y = x;
    Assert.AreNotSame(x, y);
}
```

```
[Test]
public void IntIsImmutable()
{
    int x = 32;
    int y = x;

    Assert.AreEqual(x, y);

    x = 33;

    Assert.AreNotEqual(x, y);
}
```

PXL IT

x is assigned to y, but the objects are not the same: explain!

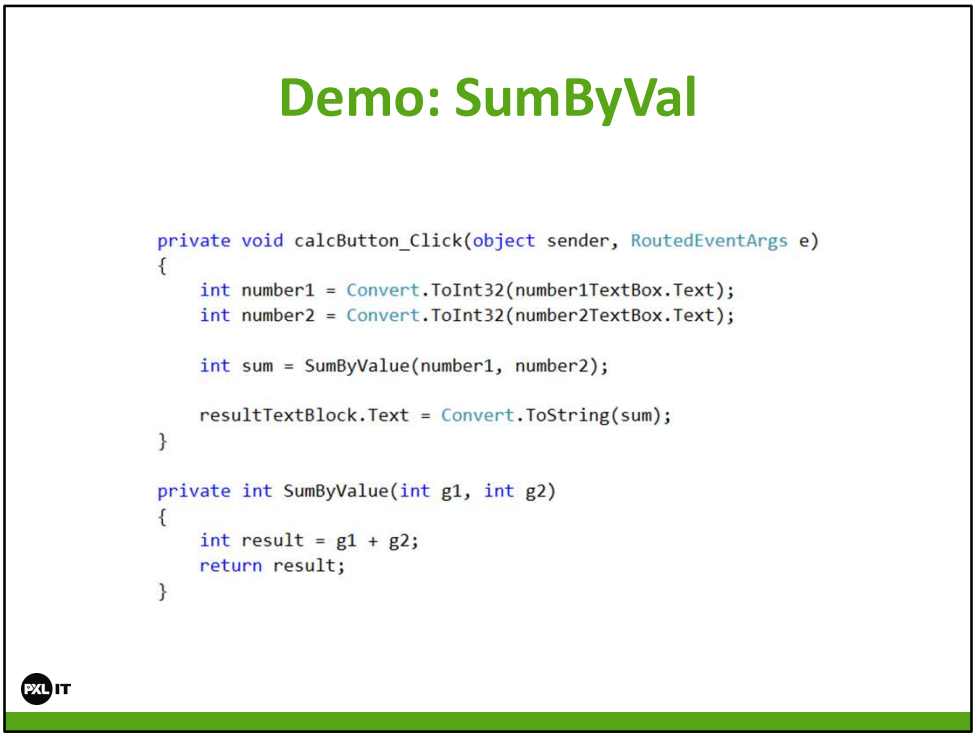
If you change x, y is not changed accordingly, because they are different objects!



When you call a method, a new block of memory is allocated on the Stack. A Stack is a LIFO structure, so the last allocated block is released first (e.g. a stack of cards).

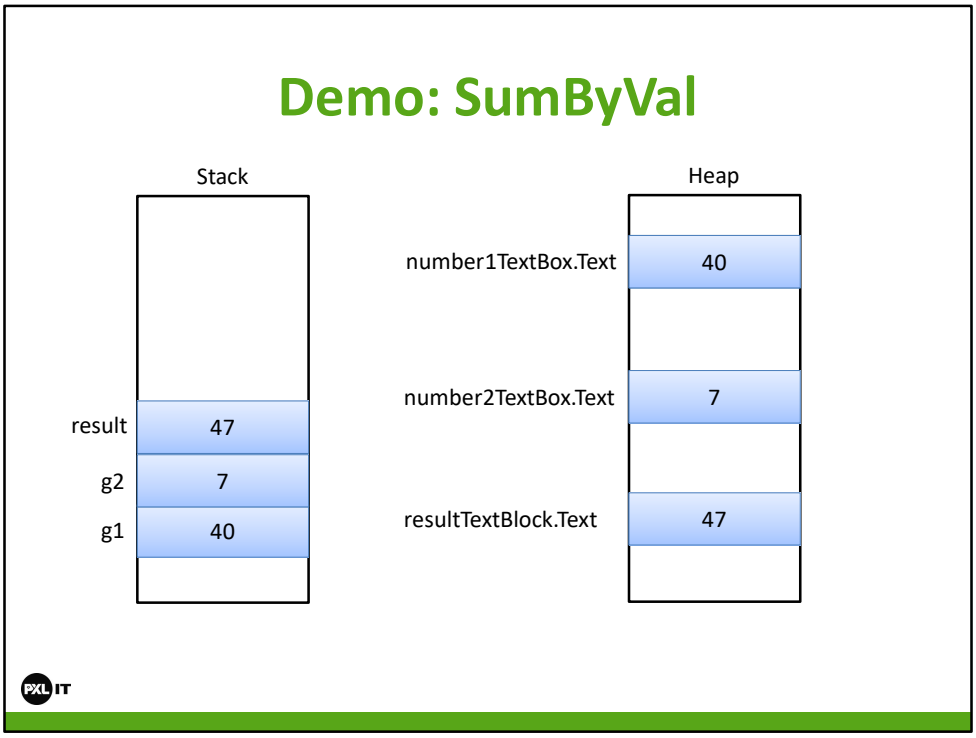
Per block of memory, you will find:

- Actual values of the parameters (parameter binding). Depending on the way this occurs (by value or by reference) you will find different values:
    - Reference types : copy a reference to the object. The actual object resides on the heap
    - Value types: copy the whole object onto the stack. This is a thus a copy.
    - When you use the ref/out keyword, you always copy a reference (pointer) to the object.
- Local variables are also allocated on the stack. However, depending on the type (Value or Reference) you allocate the object itself or a reference to the heap.



Demo: SumByVal

➔ Parameter values are copied on to the Stack



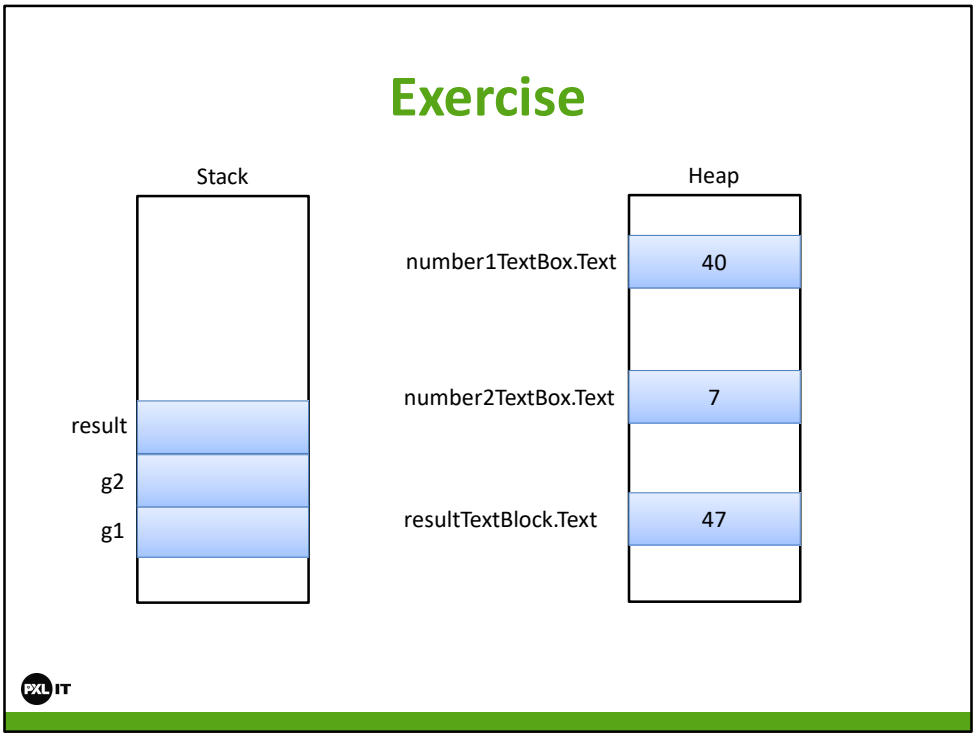
number1TextBox and number2TextBox live on the heap, because they are reference types (class TextBox).

When you execute the method, you copy the int values to the Stack

Demo: SumByRef

```
private void calcButton_Click(object sender, RoutedEventArgs e)
{
    int number1 = Convert.ToInt32(number1TextBox.Text);
    int number2 = Convert.ToInt32(number2TextBox.Text);

    int sum = SumByRef(ref number1, ref number2);

    resultTextBlock.Text = Convert.ToString(sum);
}

private int SumByRef(ref int g1, ref int g2)
{
    int result = g1 + g2;
    return result;
}
```

Now you don’t copy the values onto the stack, but you pass a reference to the original locations on the heap.



number1TextBox and number2TextBox live on the heap, because they are reference types (class TextBox).

When you execute the method, you copy the int values to the Stack

Struct

- Struct definitions create value types
  - Should represent a single value
  - Should be small
  - Are passed by value

```
public struct DateTime
{
    // ...
}
```

When do you create a class and when a struct?

→ Normally always a class, unless you have performance considerations

→ Are passed by value, so copied onto the stack

Example: DateTime, int → String is NO struct!!

More info: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/structs

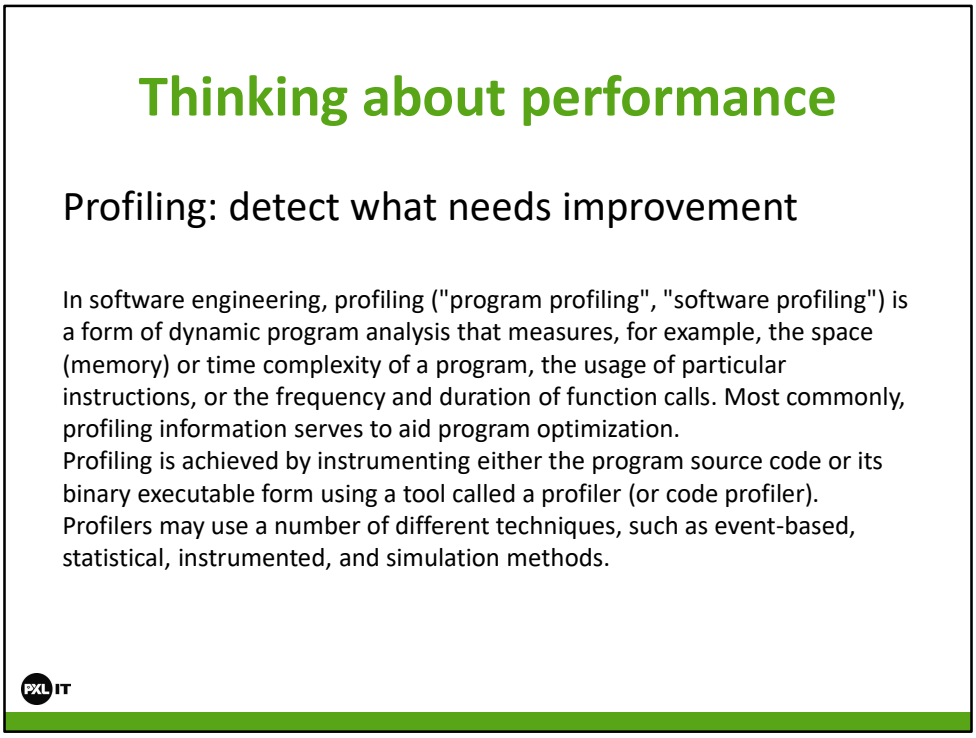
Enum

- An enum creates a value type
  - A set of named constants
  - Underlying data type is int by default

```
public enum PayrollType
{
    Contractor = 1,
    Salaried,
    Executive,
    Hourly
}
```

```
if(employee.Role == PayrollType.Hourly)
{
    // ...
}
```

You could change the data type, but it should be “enumerable” → int, long, etc (no string)



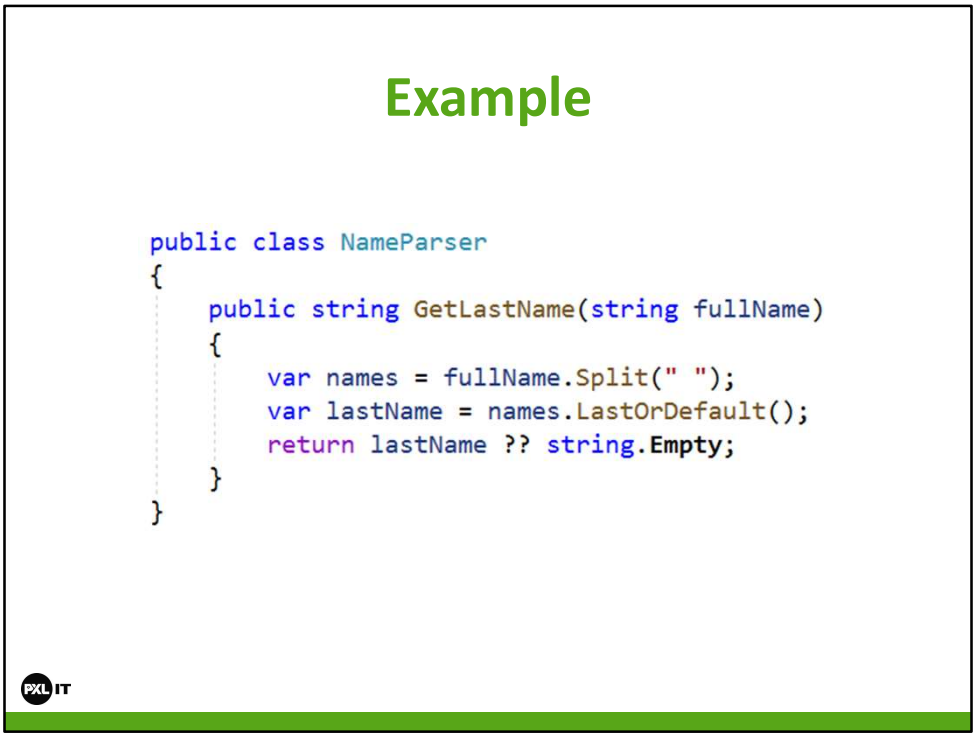
https://en.wikipedia.org/wiki/Profiling_(computer_programming)

You use profilers to detect bottlenecks in your code. Where could you improve? Typically, you profile on two main area's:

1. **Memory allocations**: where in your code occurs the most allocations? Do you free (release) your objects to prevent memory leaks?
2. **CPU**: where in your code is the most time spent by the CPU? These methods are prime candidate for optimisation.

Once you now a method or class that could be improved, you can benchmark it. This means that you measure its performance (execution time, memory consumption) to set a base line. Then you switch the implementation and compare with your original implementation.

Note: benchmarks are always (by definition) run outside the context of the real system in action, so be aware that sometimes results could be not as satisfying as you would like. E.g. : some camera manufacturers optimize only only to get a high DxOMark ranking, but that does not mean your camera will perform good in real life… The same argument can be made for optimizing software.



This is a simple implementation to guess the lastname from a full name. Question: how does it perform? We will figure this out by writing a benchmark.

Attempt 1: Stopwatch

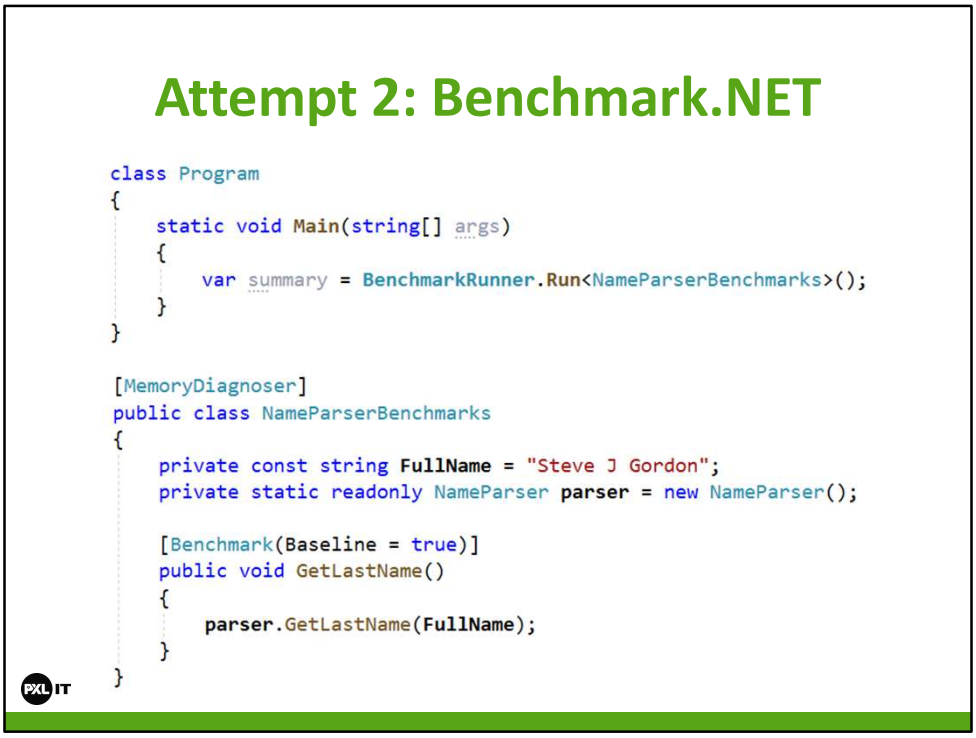
```
class Program
{
    private const string FullName = "Steve J Gordon";
    private static readonly NameParser parser = new NameParser();

    static void Main(string[] args)
    {
        Console.WriteLine("Naive benchmark");

        var stopwatch = new Stopwatch();
        stopwatch.Start();
        parser.GetLastName(FullName);
        stopwatch.Stop();
        Console.WriteLine($"Elapsed time: {stopwatch.ElapsedMilliseconds} ms");
    }
}
```

You should run this from the command line and outside of Visual Studio. Also, use the Release version of your application!

dotnet <<name of dll>>

Problems with this approach

- Not statistically correct
  - You should repeat a lot of times and take average
- Only cold start is measured
- Only elapsed time, no memory consumption
- No output data available (CSV, Json, html, etc)



Demo: Benchmarking

Reference: https://www.stevejgordon.co.uk/introduction-to-benchmarking-csharp-code-with-benchmark-dot-net

Create a new console project and add BenchmarkDotNet NuGet package. Add a reference to the solution under test (this is the same approach as with unit tests).

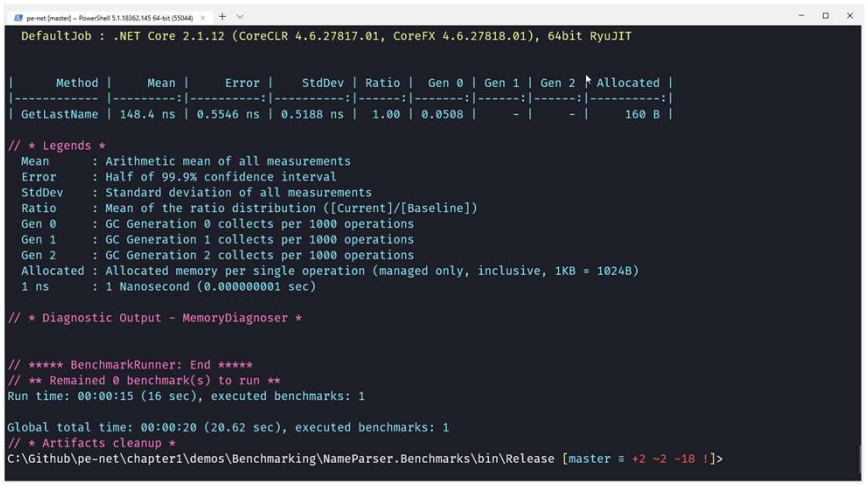
[MemoryDiagnoser]

Collect memory usage

[Benchmark]

Annotates a special kind of “Test”, ie. a benchmark. Baseline = true sets the baseline test to compare with other benchmark methods.

Attempt 2: results



The runner does a warm up fase, then collects data and prints out a nice report.

Note: details in nanoseconds. Difficult to achieve with the Stopwatch class.

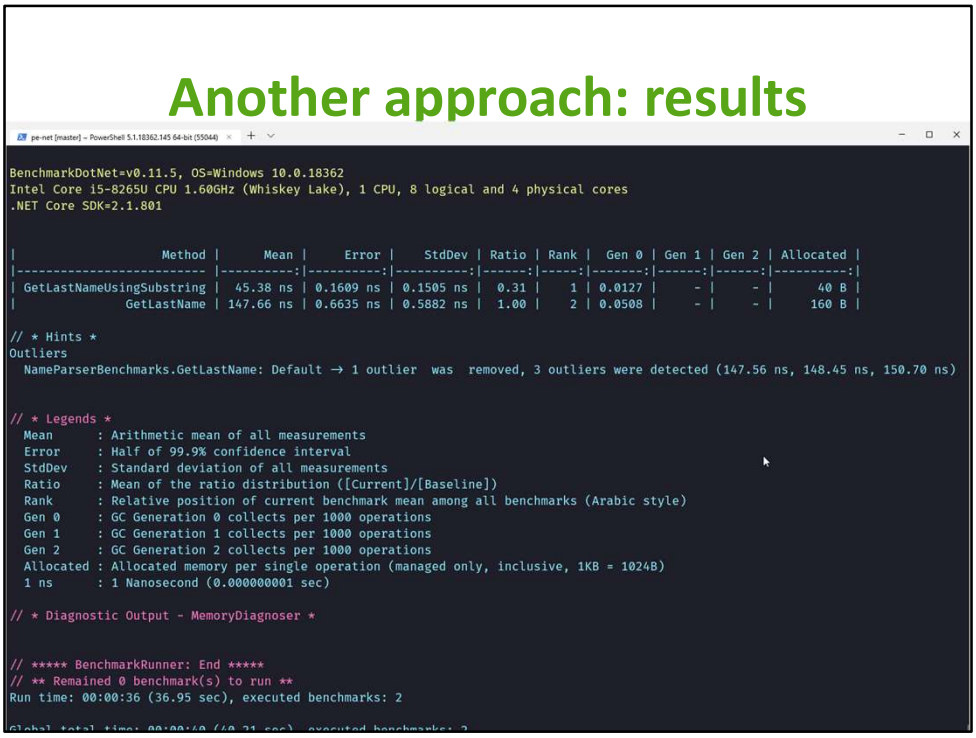
Another approach

```
[RankColumn]
[Orderer(SummaryOrderPolicy.FastestToSlowest)]
[MemoryDiagnoser]
public class NameParserBenchmarks


public string GetLastNameUsingSubstring(string fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(" ", StringComparison.Ordinal);

    return lastSpaceIndex == -1
        ? string.Empty
        : fullName.Substring(lastSpaceIndex + 1);
}
```

Is this approach faster or slower than the other method?

[RankColumn] and [Ordered…] provide for a sorted output…

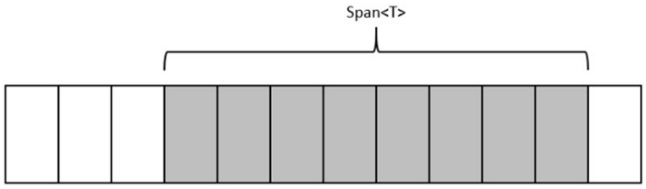
Another approach: results



```
pe-net [master] - PowerShell 5.1.18362.145 64-bit (55044)

BenchmarkDotNet=v0.11.5, OS=Windows 10.0.18362
Intel Core i5-8265U CPU 1.60GHz (Whiskey Lake), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=2.1.801


|                    Method |      Mean |     Error |    StdDev | Ratio | Rank |  Gen 0 | Gen 1 | Gen 2 | Allocated |
|------------------------- |----------:|----------:|----------:|------:|-----:|-------:|------:|------:|----------:|
| GetLastNameUsingSubstring |  45.38 ns | 0.1609 ns | 0.1505 ns |  0.31 |    1 | 0.0127 |     - |     - |      40 B |
|              GetLastName | 147.66 ns | 0.6635 ns | 0.5882 ns |  1.00 |    2 | 0.0508 |     - |     - |     160 B |

// * Hints *
Outliers
  NameParserBenchmarks.GetLastName: Default → 1 outlier  was  removed, 3 outliers were detected (147.56 ns, 148.45 ns, 150.70 ns)


// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Gen 1     : GC Generation 1 collects per 1000 operations
  Gen 2     : GC Generation 2 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:36 (36.95 sec), executed benchmarks: 2

Global total time: 00:00:40 (40.21 sec), executed benchmarks: 2
```

Span<T>

- Provides type-safe access to a continuous area of memory. This memory can be on the Heap or Stack
- ReadOnlySpan<T> for readonly access
  - Used for immutable types like String
- The Span<T> object itself cannot be used inside a class as a member field

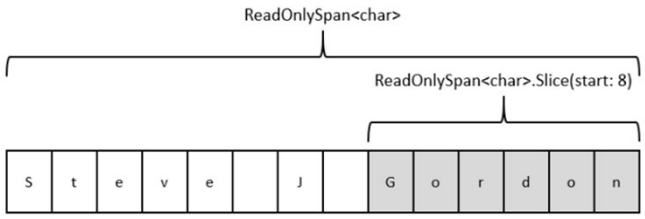


https://www.stevejgordon.co.uk/an-introduction-to-optimising-code-using-span-t

https://docs.microsoft.com/en-us/dotnet/api/system.span-1?view=netstandard-2.1

Version 3

```
public ReadOnlySpan<char> GetLastNameWithSpan(ReadOnlySpan<char> fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(' ');

    return lastSpaceIndex == -1
        ? ReadOnlySpan<char>.Empty
        : fullName.Slice(lastSpaceIndex + 1);
}
```



Because you use Span, you don't allocate new strings. The Slice methods also accesses the same memory

Final results

```
pe-net [master] – PowerShell 5.1.18362.145 64-bit (55044)
  [Host]     : .NET Core 2.1.12 (CoreCLR 4.6.27817.01, CoreFX 4.6.27818.01), 64bit RyuJIT
  DefaultJob : .NET Core 2.1.12 (CoreCLR 4.6.27817.01, CoreFX 4.6.27818.01), 64bit RyuJIT


|                    Method |      Mean |     Error |    StdDev | Ratio | Rank |  Gen 0 | Gen 1 | Gen 2 | Allocated |
|------------------------- |----------:|----------:|----------:|------:|-----:|-------:|------:|------:|----------:|
|       GetLastNameWithSpan |  17.27 ns | 0.0810 ns | 0.0757 ns |  0.12 |    1 |      - |     - |     - |         - |
| GetLastNameUsingSubstring |  45.41 ns | 0.1385 ns | 0.1296 ns |  0.30 |    2 | 0.0127 |     - |     - |      40 B |
|              GetLastName | 149.63 ns | 0.4468 ns | 0.3960 ns |  1.00 |    3 | 0.0508 |     - |     - |     160 B |

// * Hints *
Outliers
  NameParserBenchmarks.GetLastName: Default → 1 outlier  was  removed (154.89 ns)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Gen 1     : GC Generation 1 collects per 1000 operations
  Gen 2     : GC Generation 2 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:55 (55.41 sec), executed benchmarks: 3

Global total time: 00:00:58 (58.71 sec), executed benchmarks: 3
// * Artifacts cleanup *
```

Span allocates NO memory!

Some interesting tools and refs

- ILDASM / ILASM
- ILSpy
- JetBrains: dotTrace
- BenchmarkDotNet

Events and delegates

.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

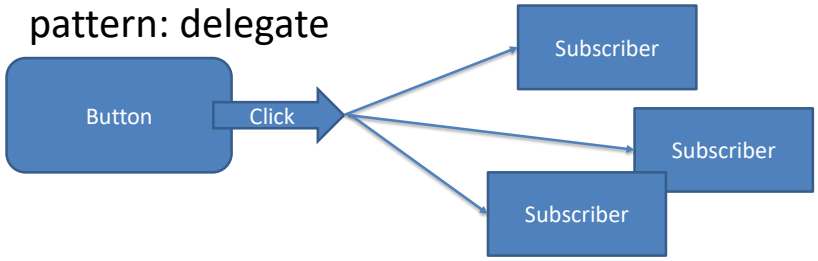
- Introduce “delegate” as a mechanism to implement PubSub (Observer) patterns
- Relate with “event” keyword

This module focuses on the events and delegates and the mechanism behind it. You are already familiar with events (e.g. the Button click event). Behind the scenes delegates come into play to create what is called a PubSub pattern.

Knowing the internals about delegates and events allows you to write more complex software in a clearly manner.

Events

- Allows a class to send notifications to other classes or objects
  - Publisher raises the event
  - One or more subscribers process the event
- C# added a special construct to support this pattern: delegate



This pattern is known as:

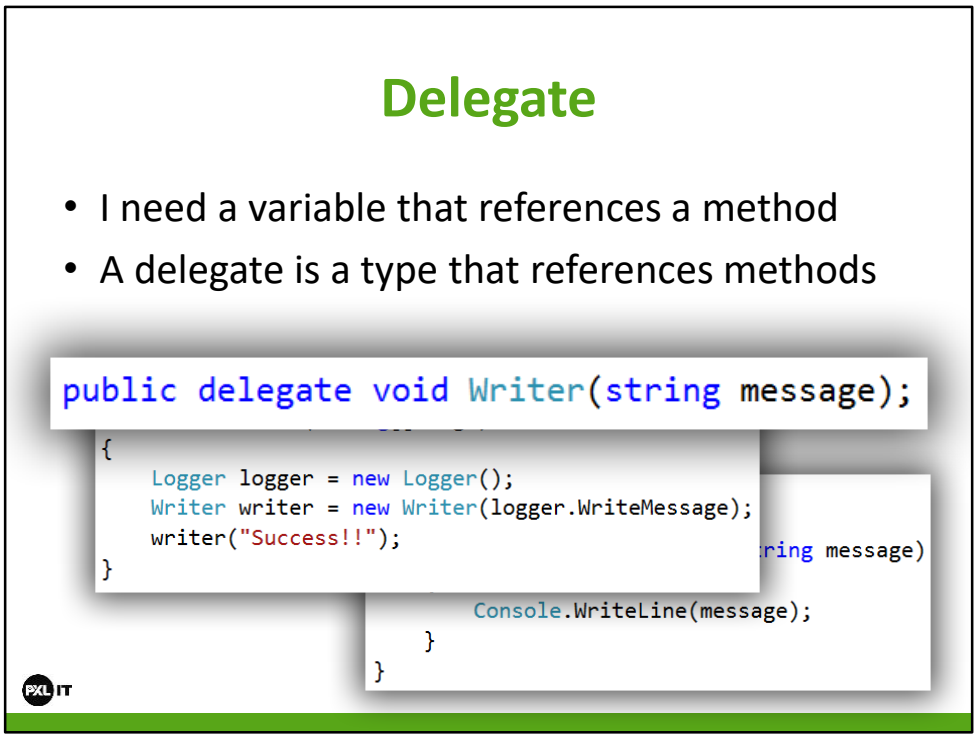
- PubSub
- Observer

This patterns is so important → C# adds specialized language constructs and execution mechanisms.

The pattern promotes separation of concerns and isolation:

- The Button (Publisher) does not need to know who receives its events
- There can me multiple listeners (Subscribers) who don't know each other and can do different things

Events are implemented in C# by a special construct called a **delegate**.



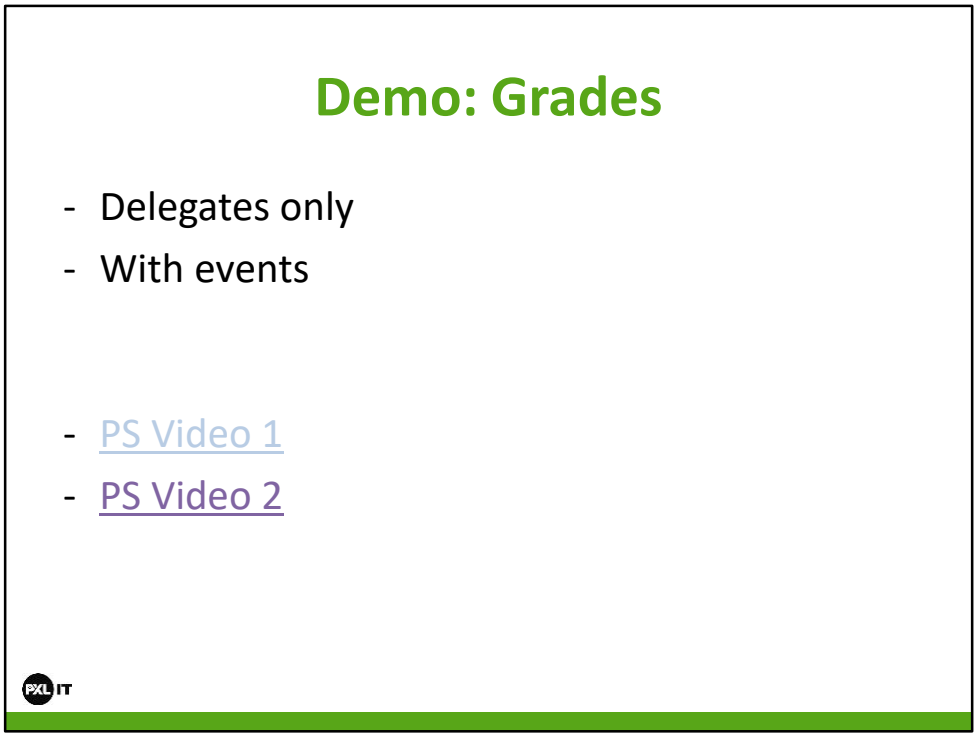
Demo: LoggerWriter

The variable does not hold a value in the traditional way, but holds a reference to a method. To do this in a typesafe way, you declare a delegate, which is a class, to define what kind of methods the variable can hold (parameters, return values).

Then you instantiate the delegate by invoking the constructor (new) and passing along a reference to a method.

Invoking the delegate [returnval] = delegatevar([params])

This executes the method.



Demo: Grades

PS Video 1:

http://www.pluralsight.com/training/player?author=scott-allen&name=csharp-fundamentals-csharp5-m4&mode=live&clip=4&course=csharp-fundamentals-csharp5

PS Video 2:

http://www.pluralsight.com/training/player?author=scott-allen&name=csharp-fundamentals-csharp5-m4&mode=live&clip=5&course=csharp-fundamentals-csharp5

Only delegates

You could implement it using only delegates, but the =-operator could overwrite existing methods.

Therefore you need events → only += and -= and certain conventions

Notice the Name property had NO IDEA which method(s) will be executed upon change!

The two video’s give a good overview of the demo’s.

Other Viewpoint

Event Raiser: girl provides notification, a message that goes out to one or more subscribers

EventArgs: extra info that comes along the event to the subscribers

Delegate: the channel through which the events pass

Event Handler: the subscriber(s)

What is an event?

- Events are notifications
- Play a central role in the .NET framework
- Provide a way to trigger notifications from end users or from objects

What is a Delegate?

- A delegate class is a specialized class often called a “Function Pointer”
- The glue between an event and an event handler
- Based on a MulticastDelegate base class
- A pipeline

What is an Event Handler

- Event handler is responsible for receiving and processing data from a delegate
- Normally receives two parameters
  - Sender
  - EventArgs
- EventArgs responsible for encapsulating event data

Creating Delegates

- Custom delegates are defined using the **delegate** keyword

```
public delegate void WorkPerformedHandler(int hours, WorkType workType);
```

Look at the demo: WorkerDemo

This delegate defines the method signature for the handlers.

Delegate and Handler Method Parameters

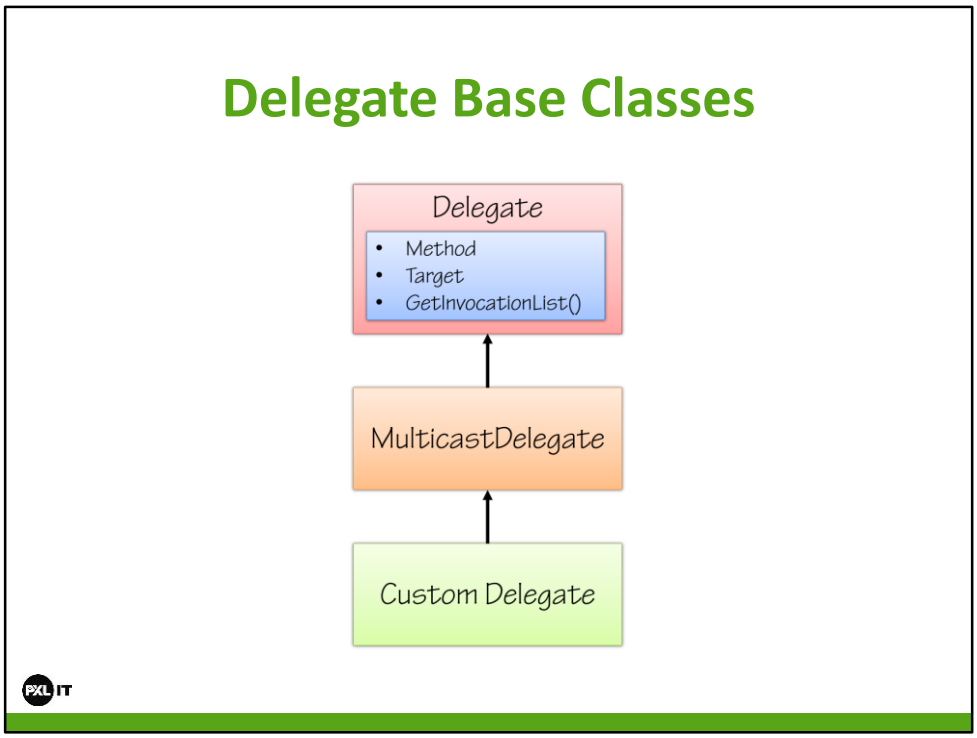
- The delegate signature must be mimicked by a handler method:

```
public delegate void WorkPerformedHandler(int hours, WorkType workType);
```

```
public void Manager_WorkPerformed(int workHours,
                                  WorkType wType)
{
    //...
}
```

Manager_WorkPerformed → parameter int and WorkType, returns void

Adheres to WorkPerformedHandler delegate



Delegate: abstract base class

https://docs.microsoft.com/en-us/dotnet/api/system.delegate

Method: Gets the method represented by the delegate.

Target: Gets the class instance on which the current delegate invokes the instance method.

GetInvocationList: usefull for multicast

MultiCastDelegate: Represents a multicast delegate; that is, a delegate that can have more than one element in its invocation list.

https://docs.microsoft.com/en-us/dotnet/api/system.multicastdelegate

Custom Delegate: user defined

NOTE: you don't explicitly use the inheritance mechanism, but you must use de delegate keyword.

Multicast Delegate

- Can reference one or more delegate functions
- Tracks delegate references using an invocation list
- Delegates in the list are invoked sequentially

Creating a Delegate Instance

```
public delegate void WorkPerformedHandler(int hours,
                                          WorkType workType);
```

```
var del1 = new WorkPerformedHandler(WorkPerformed1);
```

```
static void WorkPerformed1(int hours, WorkType workType)
{
    Console.WriteLine("Workperformed1 called.");
}
```

del1 is the “delegate instance”

It is really an object instantiation of the class WorkPerformedHandler

Invoking a Delegate instance

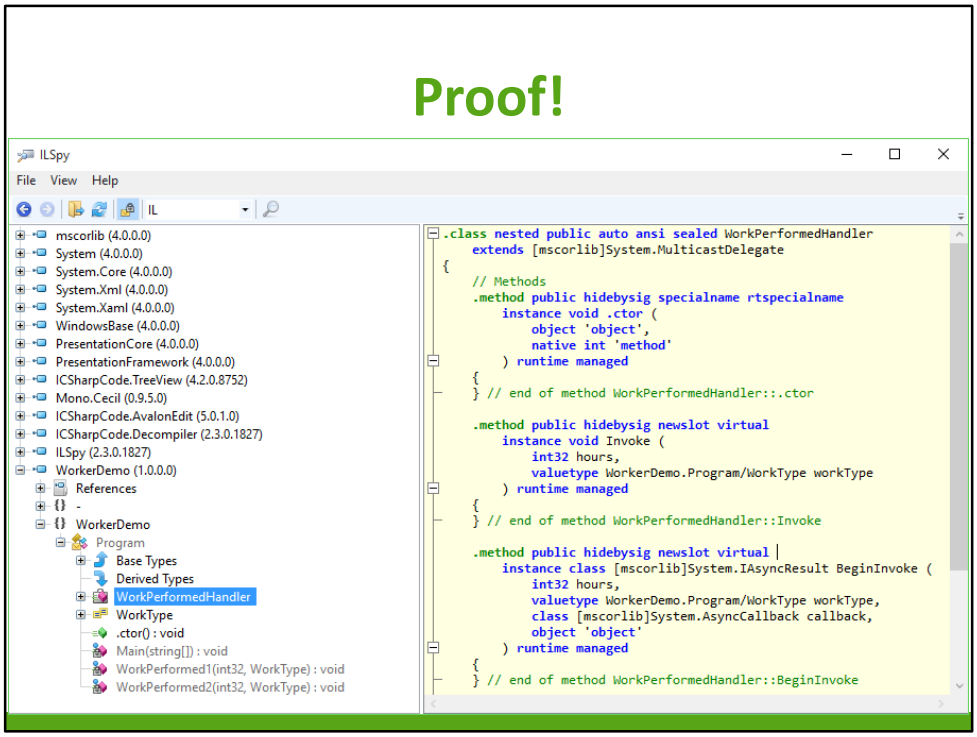
```
del1(5, WorkType.Golf);
```

Adding to the Invocation List

```
var del1 = new WorkPerformedHandler(WorkPerformed1);
var del2 = new WorkPerformedHandler(WorkPerformed2);

del1 += del2;
```

What is the output of the call del1(5, WorkType.GoToMeetings) ?

Proof!



By opening up ILSpy, you can “proof” that a delegate is a class that inherits from MulticastDelegate

Return a value from a delegate

- a delegate returns a type
- Which value is returned from the Invocation List?
- → the last one

Proof this in your labs

Define an event

- Events can be defined by the **event** keyword

```
public delegate void WorkPerformedHandler(int hours,
                                           WorkType workType);

public event WorkPerformedHandler WorkPerformed;
```

Raising Events

- Events are raised by calling the event like a method:

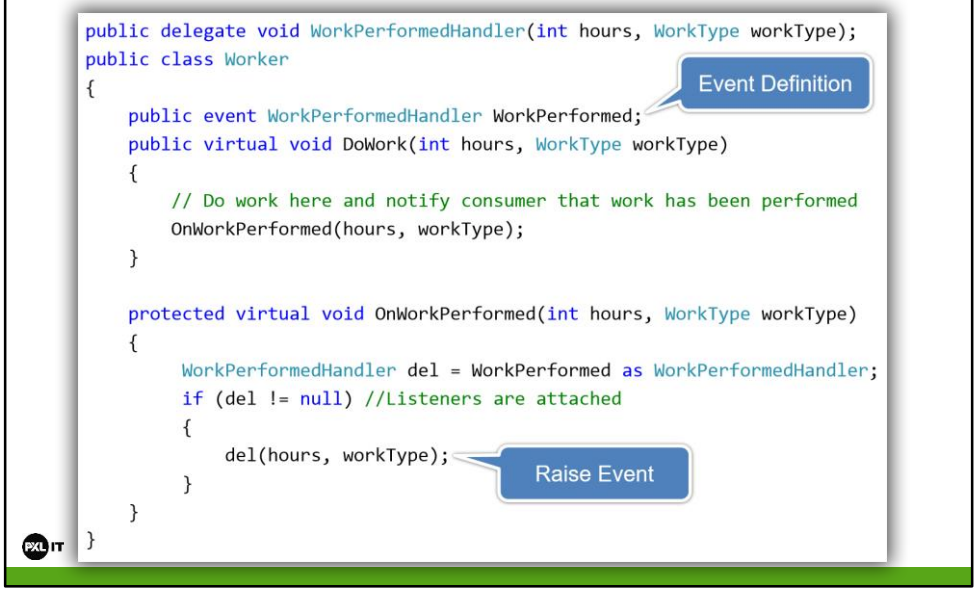
```
if (WorkPerformed != null)
{
    WorkPerformed(8, WorkType.GenerateReports);
}
```

Raising Events

- Or by accessing the event's delegate and invoking it directly:

```
WorkPerformedHandler del = WorkPerformed as WorkPerformedHandler;
if (del != null)
{
    del(8, WorkType.GenerateReports);
}
```

Exposing And Rasing Events



Best Practices / Conventions

You could raise the event inside DoWork directly: WorkPerformed(hours, workType)

Instead:

OnEventName → **On**WorkPerformed

Define this as a protected method, so you could override it.

Demo

- Project: WorkerDemoEvent

Two event types:

- WorkPerformedHandler → based on custom delegate
- EventHandler → built in .NET delegate, the same you use with e.g. a button click.

Lambdas

.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

- Delegates / Events
  - Generic version
  - Delegate inference
  - anonymous methods
- Lambdas
  - What?
  - Action<T>
  - Func<T,TResult>
- Real World Examples

This module focuses on implementing delegates and events in a CONCISE (= clear and short) manner. This is accomplished by lambdas, which are introduced by means of anonymous methods.

Last, we conclude with some real world examples where events and delegates are used. Before we go into lambdas, we introduce generic versions of common event types (EventHandler<T>).

Events – Delegates Recap

- A delegate is a type which defines a function pointer
- A delegate has an invocation list
- An event is a special kind of delegate
  - Protected access to the invocation list
  - Convention: object sender and EventArgs args
- Publisher / Subscriber design pattern

We use the example from the Pluralsight Course:

http://www.pluralsight.com/courses/csharp-events-delegates

Example (Recap)

```
public delegate void WorkPerformedHandler(object sender, WorkPerformedEventArgs args);

public class Worker
{
    public event WorkPerformedHandler WorkPerformed;
    public event EventHandler WorkCompleted;

    public void DoWork(int hours, WorkType workType)
    {
        for (int i = 0; i < hours; i++)
        {
            System.Threading.Thread.Sleep(1000);
            OnWorkPerformed(i + 1, workType);
        }
        OnWorkCompleted();
    }
```

This slides uses the code from: **WorkerWithInference**

WorkPerformedHandler: delegate definition using conventions used by the .NET framework, e.g. sender and args

Two events:

- WorkPerformed
- WorkCompleted → has no information to carry from sender to receiver, so uses default EventArgs class

DoWork:

- Simulates a delay via Sleep → do not use in production code!

Example (Recap)

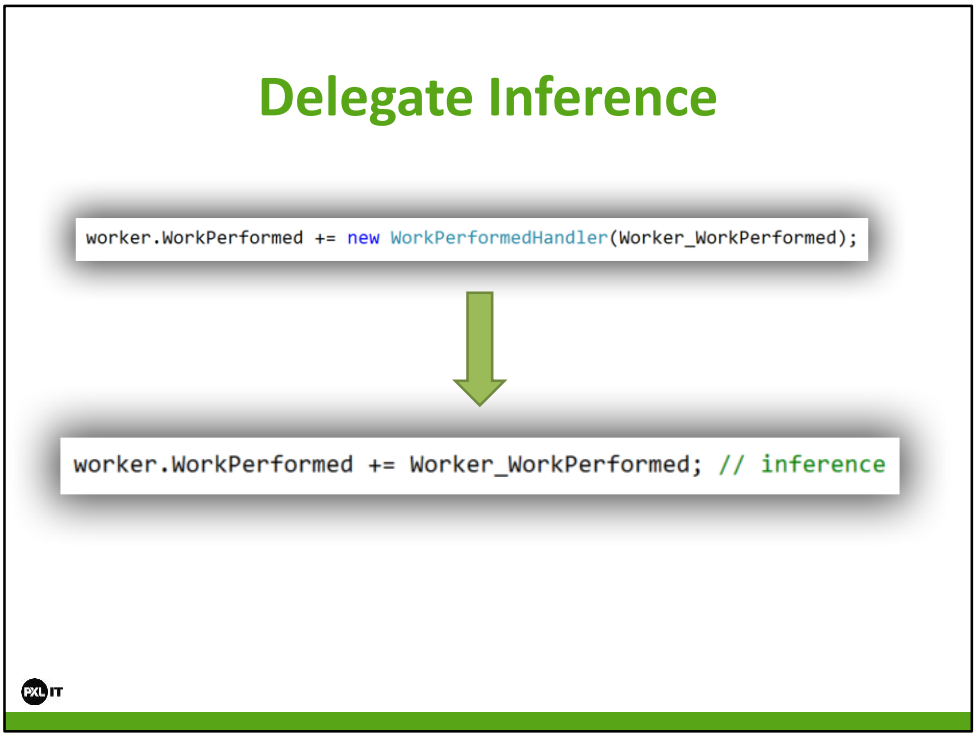
```
protected virtual void OnWorkPerformed(int hours, WorkType workType)
{
    if (WorkPerformed != null)
    {
        WorkPerformed(this, new WorkPerformedEventArgs(hours, workType));
    }
}

protected virtual void OnWorkCompleted()
{
    if (WorkCompleted != null)
    {
        WorkCompleted(this, EventArgs.Empty);
    }
}
```

OnWorkPerformed / OnWorkCompleted

- Invoke event handler by raising the event and passing the arguments
- EventArgs.Empty : no data
  https://docs.microsoft.com/en-us/dotnet/api/system.eventargs.empty?view=netframework-4.7.2
- null check, because delegates could be null
  This could be rewritten using the ? (Null-conditional operator)
  e.g.:

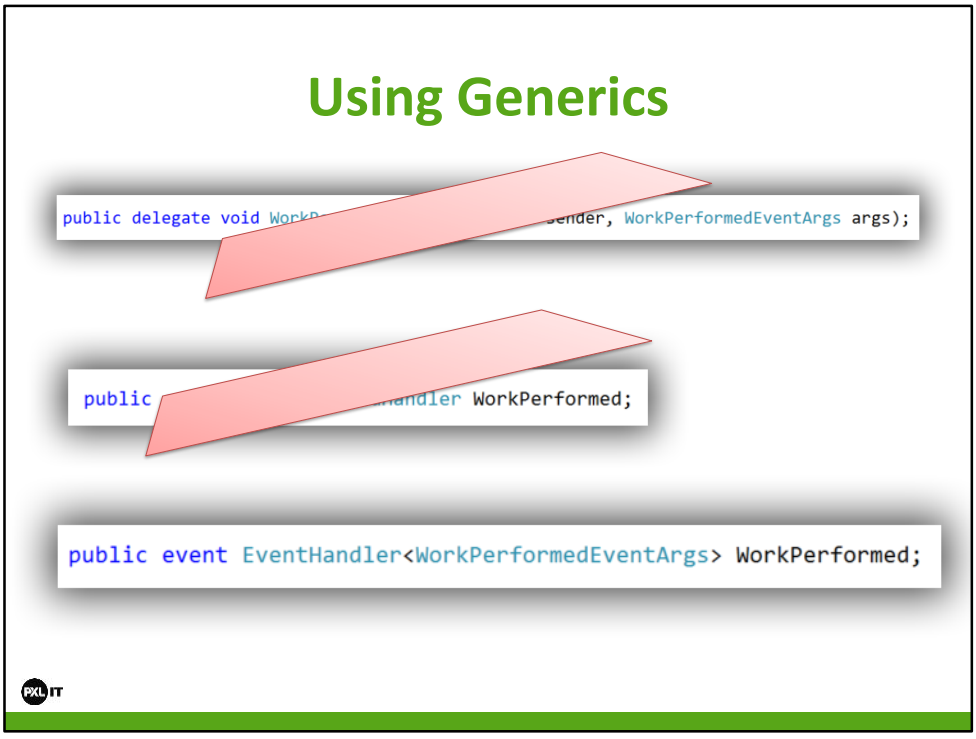
WorkCompleted?.Invoke(this, EventArgs.Empty)

- protected virtual : these methods could be overridden, this is not an important element for the remainder of the discussion.



Demo: WorkerWithInference

Instead of writing the full delegate constructor, the compiler can **derive** the correct signature of the handler and therefore you only have to specify the method to be added to the invocation list.



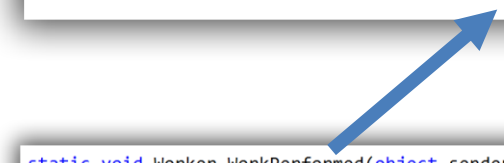
If you think about it, using events always involves writing a custom delegate, but the only thing that differs is de derived EventArgs class (e.g. WorkPerformedEventArgs).

Therefore you could use the generic class **EventHandler<T>** where you specify the derived EventArgs class between the < and >

https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler-1

Attach Directly

```
var worker = new Worker();
worker.WorkPerformed += Worker_WorkPerformed;
```



```
static void Worker_WorkPerformed(object sender, WorkPerformedEventArgs args)
{
    Console.WriteLine("Work performed: " + args.Hours);
}
```

If the code for the event handler is used only once (as if often the case), it makes sense to attach the code immediately without creating a (named) method first.

You can do this using an anonymous methods. This is not used so often anymore.

Anonymous methods

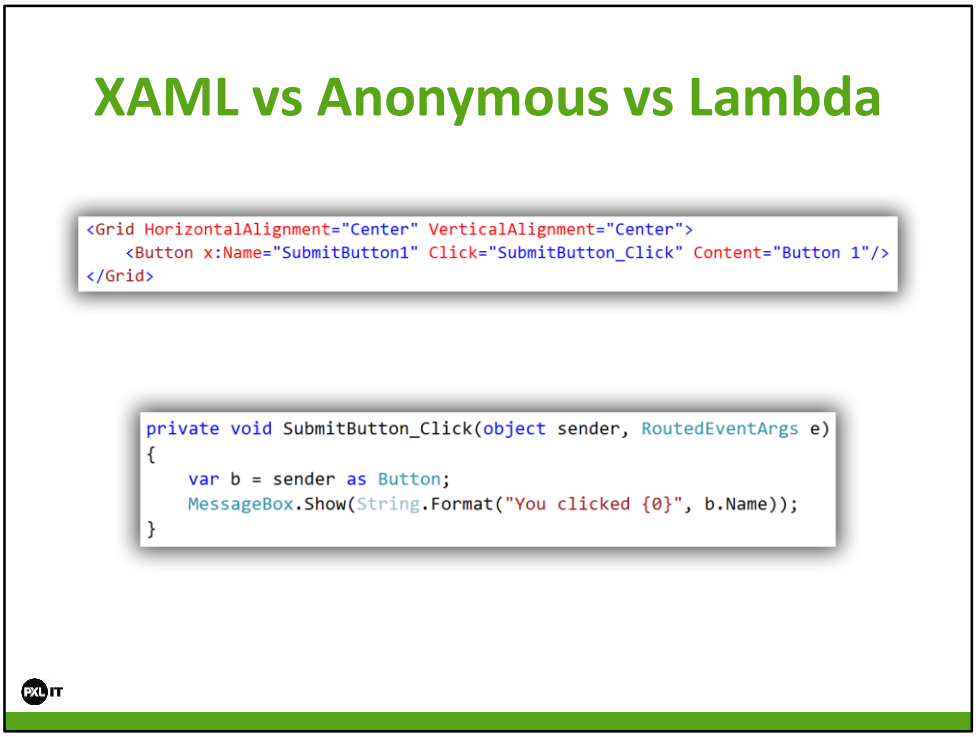
```
var worker = new Worker();
worker.WorkPerformed += delegate(object sender, WorkPerformedEventArgs e)
{
    Console.WriteLine("Work performed: " + e.Hours);
};
```

Anonymous methods allow event handler code to be hooked directly to an event. They are defined using the **delegate** keyword.

Advantage: for short methods it makes the code more readable.

Disadvantage: the method implementation is not reusable.

Demo: WorkerAnonymous



Demo: SubmitButtonClick

XAML

The Click attributes corresponds with a Click Event which defines a delegate from type RoutedEventHandler:

https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.primitives.buttonbase.click

https://docs.microsoft.com/en-us/previous-versions/windows/embedded/cc545009(v=msdn.10)

The XAML syntax adds your method to the invocation list of this event.

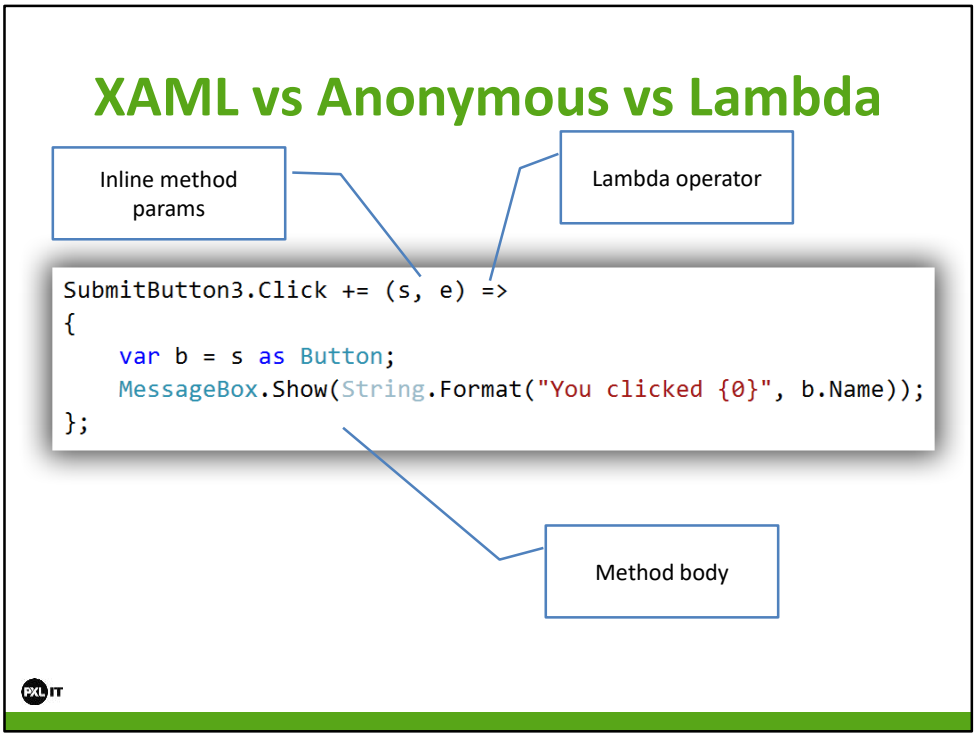
XAML vs Anonymous vs Lambda

```
SubmitButton2.Click += delegate (object sender, RoutedEventArgs e)
{
    var b = sender as Button;
    MessageBox.Show(String.Format("You clicked {0}", b.Name));
};
```

Demo: SubmitButtonClick

C# Anonymous methods

You could also write the event handler using anonymous methods. This is not used so often anymore since the introduction of lambdas in 2007 (!)



Demo: SubmitButtonClick

Or you use a lambda, this gives the most elegant (concise) code.

A lambda has 3 parts:

(s, e) are the **inline method parameters**. There is **no** type declaration, because the compiler derives these based on the event (and underlying delegate) type.

=> is the **lambda operator**. It seperates the method parameters from the body.

Method body is the code block that will be executed eventually and can make use of the parameters that are passed in. Note: { } and ;

Assigning a Lambda to a Delegate

```
delegate int AddDelegate(int a, int b);

static void Main(string[] args)
{
    AddDelegate ad = (a, b) => a + b;
    int result = ad(1, 1); // result = 2
}
```

Demo: LambdaDelegate

If you have a delegate, you can add a method to it by means of a lambda expression.

- Compiler detects the parameter types
- Names a, b don't have to be the same, e.g: AddDelegate ad = (foo1, foo2) => foo1 + foo2;
- No return keyword necessary, because there is no { } → just one statement → this is supposed to return a value

Handling empty parameters

```
delegate bool LogDelegate();

static void Main(string[] args)
{
    LogDelegate ld = () =>
    {
        UpdateDatabase();
        WriteToEventLog();
        return true;
    };

    bool status = ld();
}
```

Demo: LambdaDelegate

Demo: Lambdas with events

```
var worker = new Worker();
worker.WorkPerformed += (s, e) =>
{
    Console.WriteLine("Work performed: " + e.Hours);
};


worker.WorkCompleted += (s, e) =>
{
    Console.WriteLine("Worker is done");
};
```

Demo: WorkerLambdas

This is almost exact the same code as using an anonymous method, but thanks to lambdas, parameter declaration is much shorter:

(s, e) => *instead of* delegate(object sender, WorkPerformedEventArgs e)

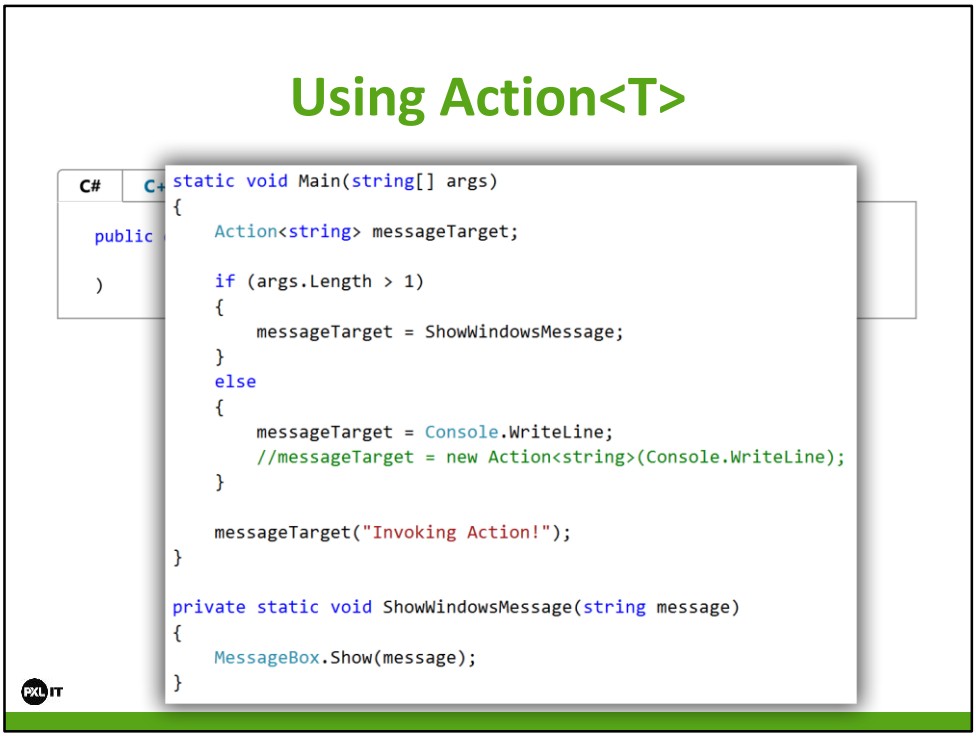
Delegates in .NET

- The .NET framework provides several different delegates that provide flexible options:
  - **Action<T>** - Accepts a single parameter and returns nog value
  - **Func<T,TResult>** - Accepts a single parameter and return a value of type Tresult
  - Several variations for these delegates exist, so you don't have to explicitly declare a new delegate type!

Built-in delegates that saves you code.

Generics because types can differ.

https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=netframework-4.8
https://docs.microsoft.com/en-us/dotnet/api/system.func-2?view=netframework-4.8

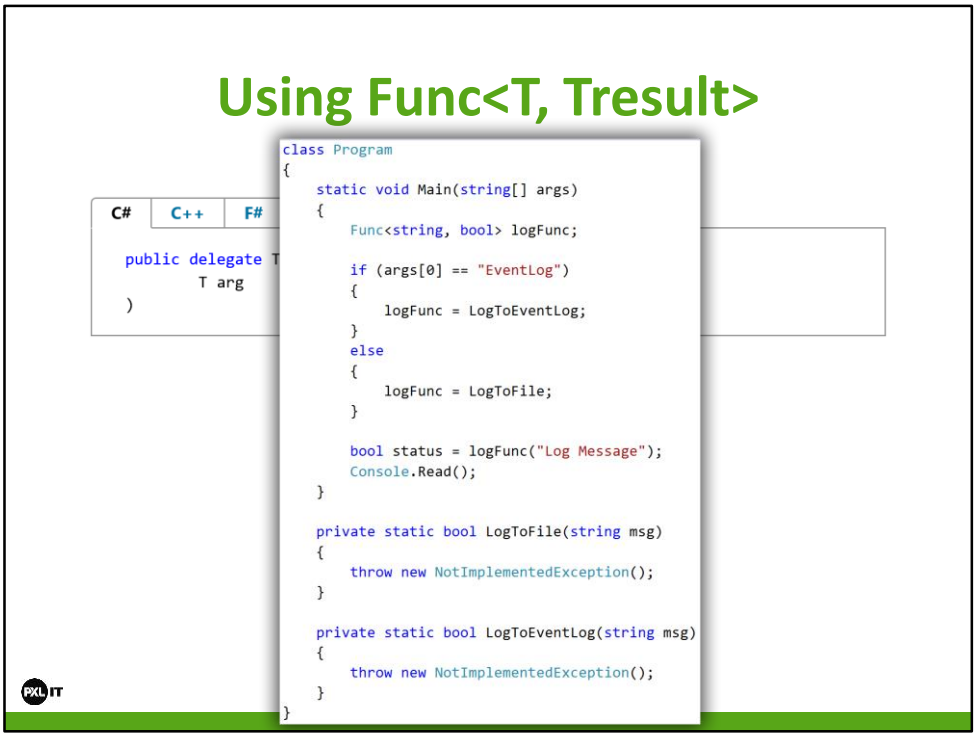


Demo: ActionMessages

https://docs.microsoft.com/en-us/dotnet/api/system.action-1

Action<T> encapsulates a method that has a single parameter and does not return a value.

You pass in a method like a “self-written” delegate. By using generics, you can pass in different types of parameters.



Demo: FuncMessages

https://docs.microsoft.com/en-us/dotnet/api/system.func-2

Func<T, TResult> encapsulates a method that has one parameter and returns a value of the type specified by the *TResult* parameter.

You pass in a function like a “self-written” delegate. By using generics, you can pass in different types of parameters en result types.

Demo: WorkerActionFunc

```
public delegate int BizRulesDelegate(int x, int y);

class Program
{
    static void Main(string[] args)
    {
        // using custom delegates
        var data = new ProcessData();
        BizRulesDelegate addDel = (x, y) => x + y;
        BizRulesDelegate multiplyDel = (x, y) => x * y;
        data.Process(2, 3, multiplyDel);

        // using Actions
        Action<int, int> myAction = (x, y) => Console.WriteLine(x + y);
        Action<int, int > myMultiplyAction = (x, y) => Console.WriteLine(x * y);
        data.ProcessAction(2, 3, myAction);
        data.ProcessAction(2, 3, myMultiplyAction);

        // using Func
        Func<int, int, int> funcAddDel = (x, y) => x + y;
        Func<int, int, int> funcMultiplyDel = (x, y) => x * y;
        data.ProcessFunc(2, 3, funcAddDel);
        data.ProcessFunc(2, 3, funcMultiplyDel);
```

PXL IT

Demo: WorkerActionFunc

Step through the code and make sure you understand every line of it.

Lambdas and LINQ

- LINQ uses extension methods on object collections
- These extension methods take delegates as arguments to filter data
- These delegates are expressed most elegantly as lambdas
- Demo: LambdasLINQ

Demo: LambdasLINQ

```
var custs = new List<Customer>
{
    new Customer { City = "Phoenix", FirstName = "Jo
    new Customer { City = "Phoenix", FirstName = "Ja
    new Customer { City = "Seattle", FirstName = "Su
    new Customer { City = "New York City", FirstName
};

var phxCusts = custs
    .Where(c => c.City == "Phoenix" && c.ID < 500)
    .OrderBy(c => c.FirstName);

foreach (var cust in phxCusts)
{
    Console.WriteLine(cust.FirstName);
}
```

Definition of the **Where** extension method:
https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where

Definition of the **OrderBy** Extension method:
https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderby

Arguments of these extension methods are passed in as lambda expressions.

Extension methods

- You want to add some members to a type
- You don’t need to add extra data to the type
- You can’t change the type itself
  - You don’t have the code
  - You don’t want to break existing code
- They are in fact a special kind of static methods

More info:https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C#, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

Extension Methods

```
// Define an extension method in a non-nested static class.
public static class Extensions
{
    public static Grades minPassing = Grades.D;
    public static bool Passing(this Grades grade)
    {
        return grade >= minPassing;
    }
}
```

The class itself is static, and the extension method also.

By using the **this** keyword, you “extend” the method towards the object instance, making it appear like a normal method the struct Grade.

Extension Methods

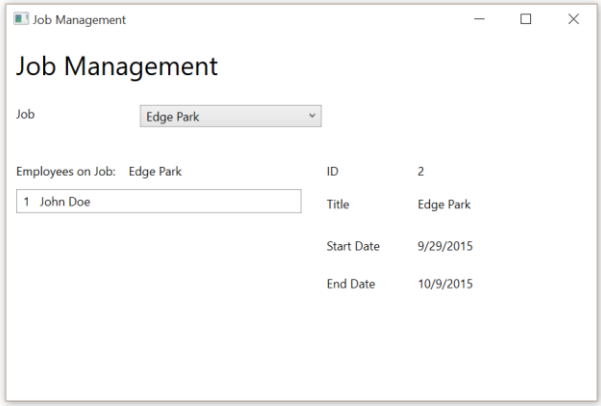
```
public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
class Program
{
    static void Main(string[] args)
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

        Extensions.minPassing = Grades.C;
        Console.WriteLine("\r\nRaising the bar!\r\n");
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
    }
}
```

This example can be found in the MSDN docs:

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/how-to-create-a-new-method-for-an-enumeration

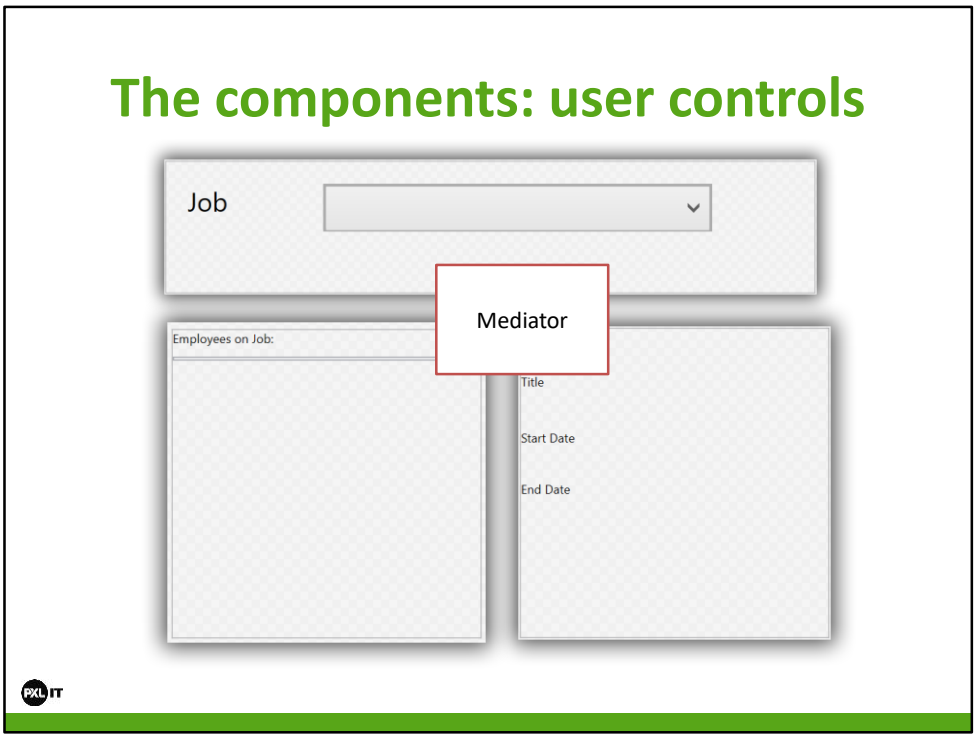
Study the demos in Pluralsight yourself.

Communicating between components



The problem:

Different areas of the screen about the same data should work together. If you select a different job, the list of Employees should update (left) AND the job details (RIGHT).



A User Control is simply a collection of (WPF) elements grouped together, so it can be reused in diffferent places.

We want to design this in a loosely coupled way: the “Employees on Job” control and “Job Details” control may not know about the Job combobox. How could this be done?

One possible way to realize this is using the **mediator** design pattern.

Mediator: http://www.dofactory.com/net/mediator-design-pattern

They all know about the mediator object, but not about each other

Demo: CommunicatingBetweenControls

The Mediator creates events of type JobChanged, which the User Controls subscribe to. The combobox actually changes the Job, so is considered the sender of the event.

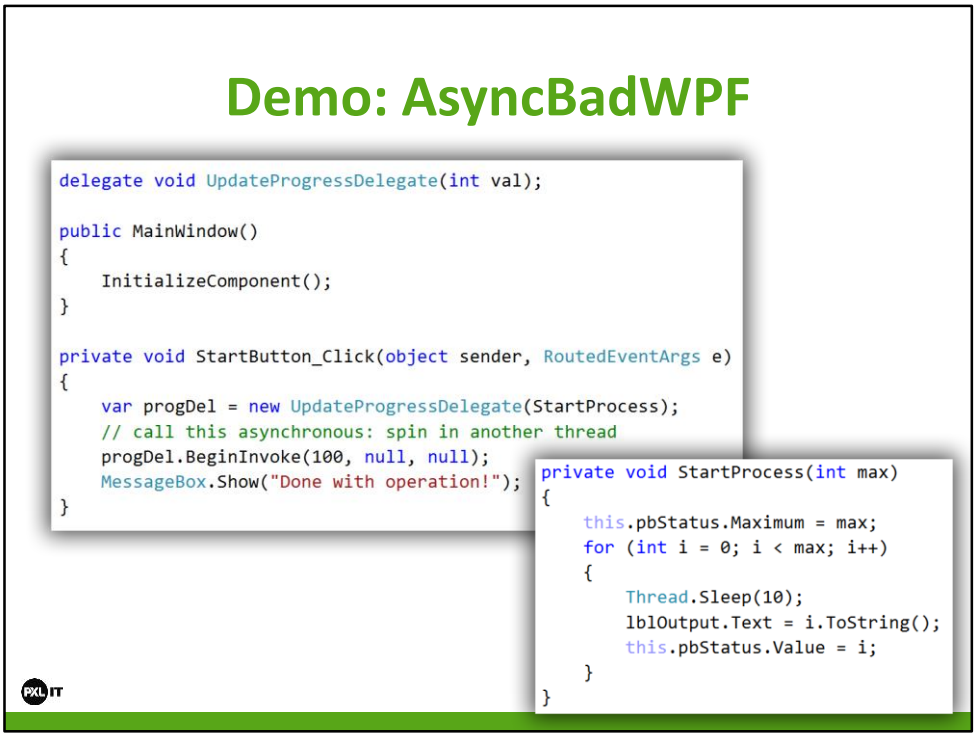
NOTE: This is not the only way to solve this problem. Another popular pattern is the Model-View-ViewModel pattern.

Async delegates: bad

- You call a delegate using “BeginInvoke”
- This starts a new thread
- In this thread you NEVER may update gui components directly

More info on **BeginInvoke**:

https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/calling-synchronous-methods-asynchronously



Demo: AsyncBadWPF

You call BeginInvoke on the delegate → starts another thread → execute StartProcess

StartProcess → change Text property of lblOutput and Value property of pbStatus → NOT ALLOWED!

Exeption: System.InvalidOperationException

Demo: AsyncGoodWPF

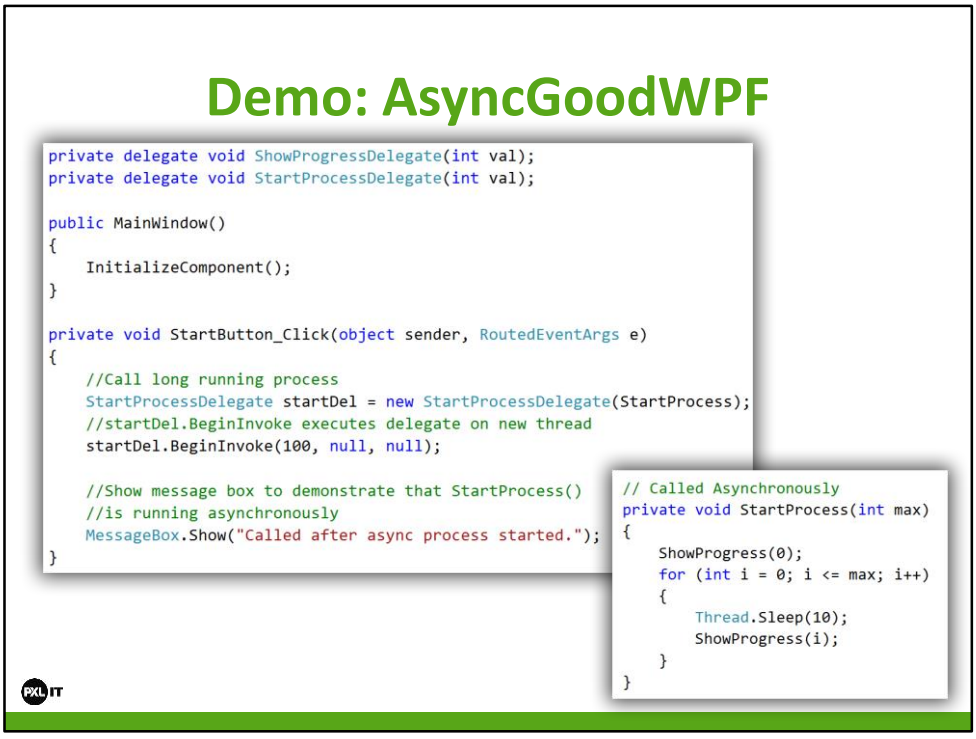
- Dispatcher: provices services for managing a queue on a Thread => associated with GUI
- Dispatcher.CheckAccess: is the currently running Thread the GUI thread?
- Dispatcher.BeginInvoke: invoke the delegate on the Thread associated with the Dispatcher => the GUI thread
- Solution: introduce an extra delegate

CheckAccess:

https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcher.checkaccess

BeginInvoke:

https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcher.begininvoke



Demo: AsyncGoodWPF

Demo: AsyncGoodWPF

```
private void ShowProgress(int i)
{
    //On helper thread so invoke on UI thread to avoid
    //updating UI controls from alternate thread
    if (!Dispatcher.CheckAccess())
    {
        ShowProgressDelegate del = new ShowProgressDelegate(ShowProgress);
        //this.BeginInvoke executes delegate on thread used by form (UI thread)
        Dispatcher.BeginInvoke(del, new object[] { i });
    }
    else
    { //On UI thread so we are safe to update
        this.lblOutput.Text = i.ToString();
        this.pbStatus.Value = i;
    }
}
```

Set a BreakPoint on the first line of this method.

Show Threads (Debug > Windows > Thread)

Watch how this code gets invoked from different threads.

BackgroundWorker

- Executes an operation on a separate thread
- Delegates:
  - **Do_Work** : execute the time consuming operation on a seperate thread. Be careful not to update the GUI
  - **ProgressChanged** : runs on the gui thread, so you can report progress
  - **RunWorkerCompleted**: runs on the gui thread, so you can report completion

More info:https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.backgroundworker

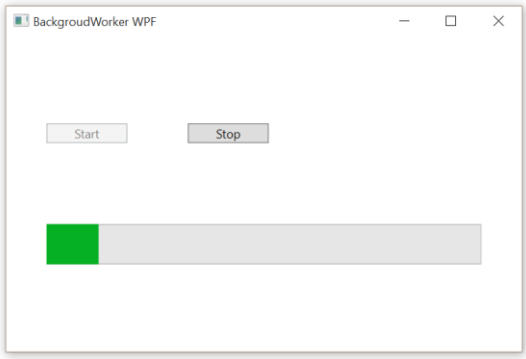
To execute a time-consuming operation in the background, create a BackgroundWorker and listen for events that report the progress of your operation and signal when your operation is finished.

To set up for a background operation, add an event handler for the DoWork event. Call your time-consuming operation in this event handler. To start the operation, call RunWorkerAsync. To receive notifications of progress updates, handle the ProgressChanged event. To receive a notification when the operation is completed, handle the RunWorkerCompleted event.

Note:

You must be careful not to manipulate any user-interface objects in your DoWork event handler. Instead, communicate to the user interface through the ProgressChanged and RunWorkerCompleted events.

BackgroundWorkerDemoWPF



Study the code of the demo yourself.

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. Using the latest C# keywords **async** and **await**, the asynchronous part becomes much more readable. However this hides much of its complexity.

Plain Threads

- Class Thread accepts a delegate
- Methods for interacting with threads

```
public class Alpha
{
    // This method that will be called when the thread is started
    public void Beta()
    {
        while (true)
        {
            Console.WriteLine("Alpha.Beta is running in its own thread.");
            Thread.Sleep(500);
        }
    }
};
```

Demo: OnlyThreads

Reference: https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx

Class Alpha defines a method that will be run by a separate thread.

Thread.Sleep simulates a long computation. Namespace: System.Threading

Working with Threads

```
Alpha oAlpha = new Alpha();

// Create the thread object, passing in the Alpha.Beta method
// via a ThreadStart delegate. This does not start the thread.
Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

// Start the thread
oThread.Start();

// Spin for a while waiting for the started thread to become
// alive:
while (!oThread.IsAlive) ;

// Put the Main thread to sleep for 3 seconds to allow oThread
// to do some work:
Thread.Sleep(3000);
```

Demo: OnlyThreads

ThreadStart: delegate to create a Thread

- Start: start the thread
- IsAlive: checks thread
- Thread.Sleep: put current thread to sleep

Working with threads

```
// Request that oThread be stopped
oThread.Abort();

// Wait until oThread finishes. Join also has overloads
// that take a millisecond interval or a TimeSpan object.
oThread.Join();

Console.WriteLine();
Console.WriteLine("Alpha.Beta has finished");

try
{
    Console.WriteLine("Try to restart the Alpha.Beta thread");
    oThread.Start();
}
catch (ThreadStateException)
{
    Console.Write("ThreadStateException trying to restart Alpha.Beta. ");
    Console.WriteLine("Expected since aborted threads cannot be restarted.");
}
```

Demo: OnlyThreads

- Abort: aborts thread, you can't restart an aborted thread
- Join: wait for thread to finish

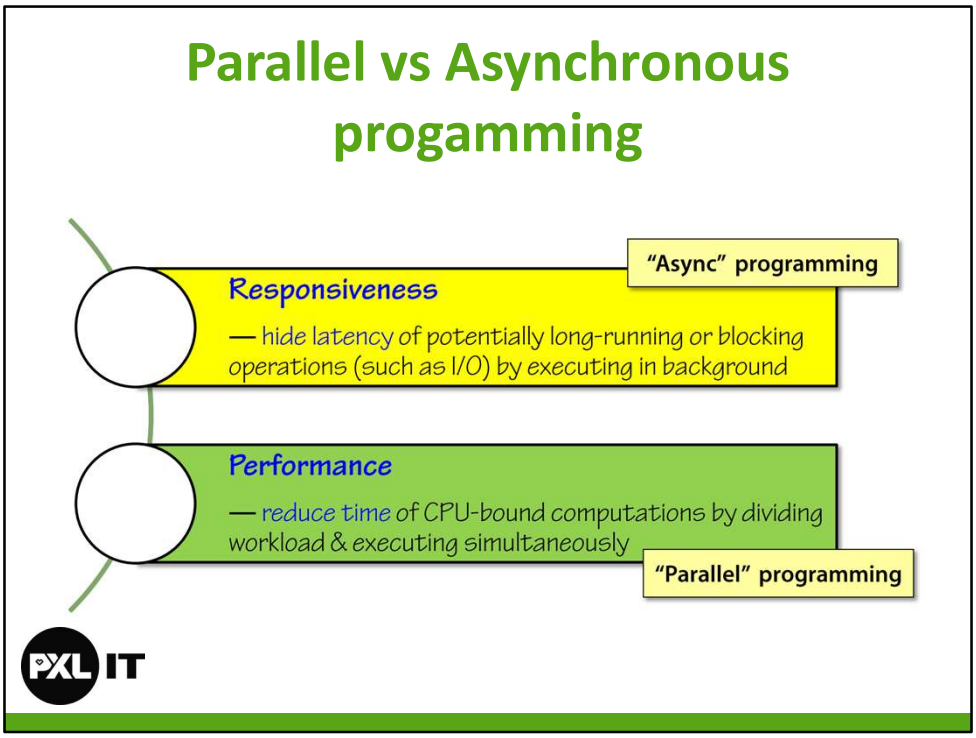
Threads are not enough

- A threads creates a separate virtual CPU inside your process (program)
- A thread is a resource, use it sparingly
- Multithreading is difficult:
  - Synchronization between threads
    - Deadlock
    - Race conditions
  - Thread pooling (resource sharing)
- Frameworks are created on top of threads

The concept of a Thread class is not enough to create complex programs. .NET has several frameworks to help manage the complexity and readability of your programs.

Deadlock: threads wait indefinitely for each other

Race condition: when sharing a variable, threads could possibly introduce errors



Responsiveness: you can still do other things in the UI while some other long or blocking operation is executed

Performance: to several things at once to get things done faster

Example Asynchronous

- Show a loading indicator
- Start an asynchronous operation
  - Read large CSV file from disk
  - Count the rows (one by one)
  - Signal that the asynchronous operation has completed
- Render the result on screen
- Remove the loading indicator

Suppose you need to load a long file from disk. How would you present this to the user?

Remark: while the program is loading, the user could still do other things, e.g. move or resize the window.

→ This example is NOT parallel programming! See the next slide.

Example Async and Parallel

- Show a loading indicator
- Start an asynchronous operation
  - Read large CSV file from disk
  - Process rows in parallel
  - Signal that the asynchronous operation has completed
- Render the result on screen
- Remove the loading indicator

PXL IT

Parallel programming is used to take a big problem and chop it up into smaller chunks that can be executed simultaneously.

Anology: boiling eggs

Parallel

- You can boil 10 eggs one by one for 10 minutes → 100 minutes or;
- You boil them all at once. This speeds up the process (all ready in 10 minutes)

Async

- If you set an egg timer, you could do something else
- The boiling process becomes asynchronous
- You will be notified when eggs are ready

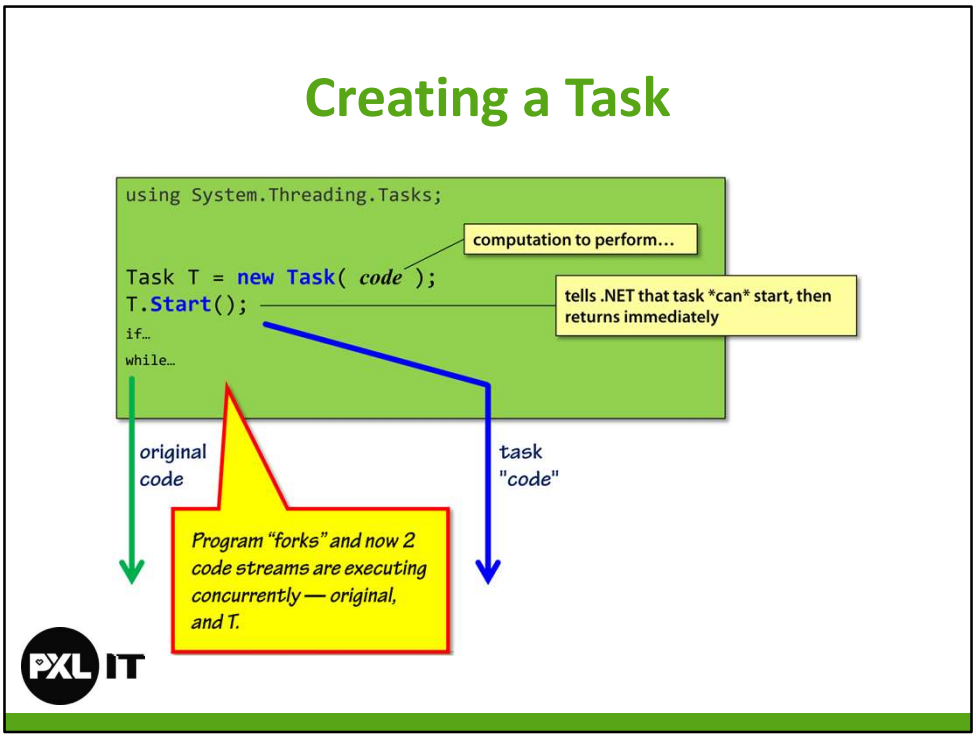
Task

- Definition
  - A unit of work
  - An object denoting an ongoing operation or computation

The Task Programming Library was introduced to allow to create parallel and asynchronous programs. It is build on top of plain threads and frees the developer of the complexities of working with threads alone.

Central concept: Task

https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task



The *code* parameter → delegate

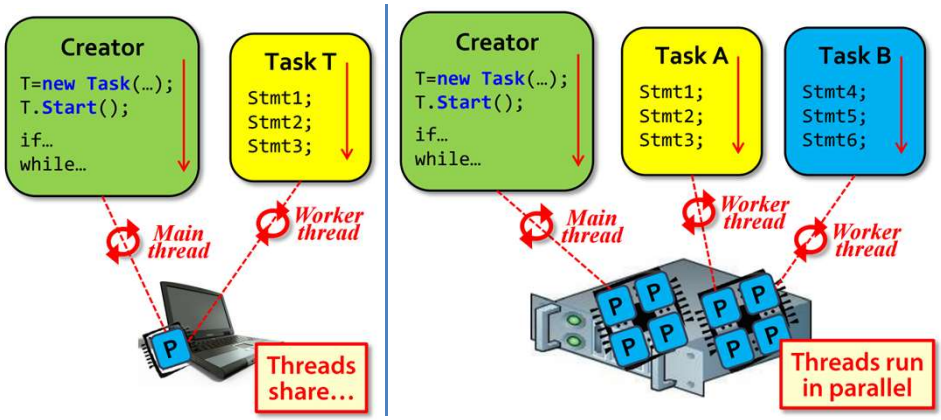
Start

- It *can* start
- Returns immediately
- Put into a queue for execution (so no direct Thread!)

The code forks into two execution paths

Execution model – quick look

- Code-based tasks are executed by a thread on some processor
- A thread is dedicated to task until task completes



Left: single core

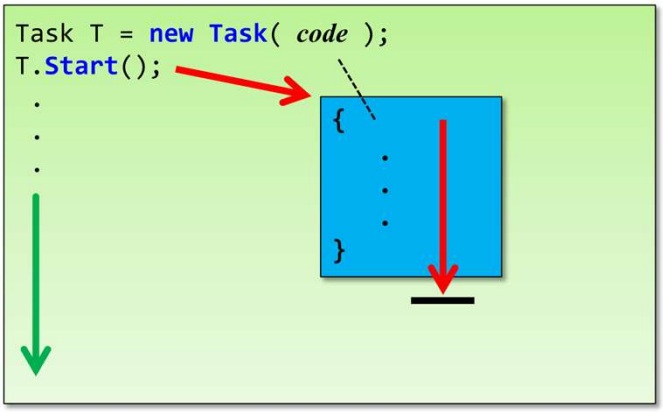
- Multiple threads are shared by the single core
- Context switching between threads does not seem useful
- But: the UI thread gets executed → responsiveness increases!

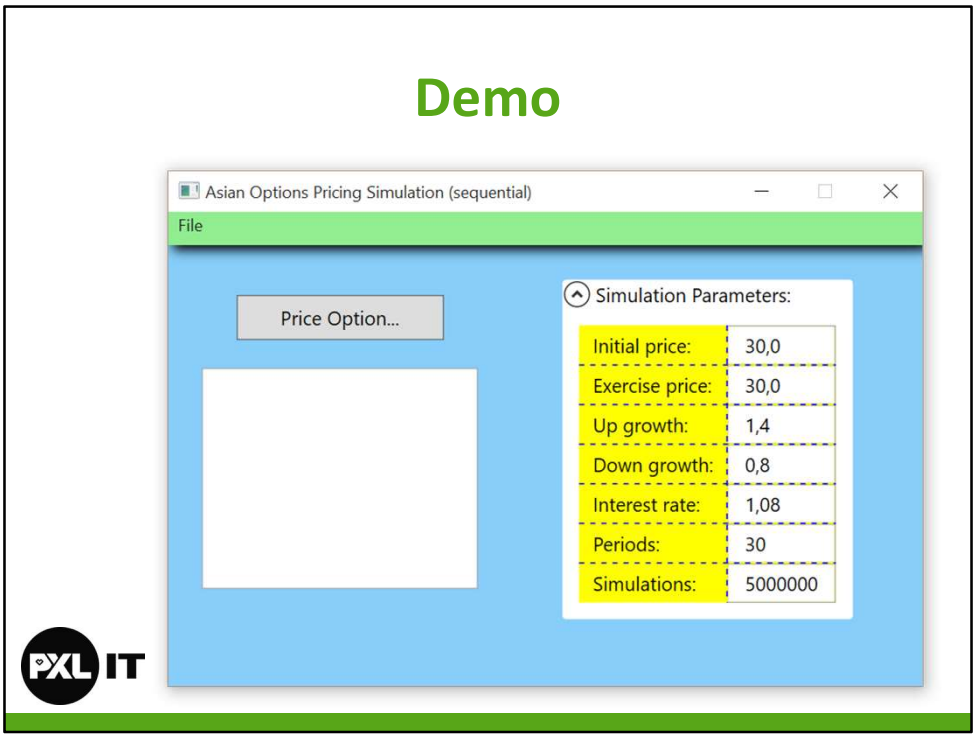
Right: multiple processors

- Tasks can really run in parallel

Task completion

- When does a task complete?
  - When the code block is exited – normally
  - Throwing an exception





Demo: AsyncOptions-Seq (and subsequent versions)

This demo is about asynchronous programming for **responsiveness**.

The application performs a large number of Monte Carlo simulations to model stock prices, in particular Asian options. In this demo, the app is parallelized at the coarse-grain level, i.e. each click of the "Simulation" button launches a task so the simulation can be done in parallel if a compute core is available. For example, on a quad-core machine, you can have as many as 4 simulations running in parallel.

The first goal is to make the app more responsive, so the UI does not freeze while a simulation is in progress.

Run the demo and notices that the program freezes while computation is going on.

There *is* code for a spinner, but this has no effect, since the computation is in the UI thread.

Note: the spinner itself is in a assembly called "Fenestra" and comes from this article:

http://blogs.msdn.com/b/pigscanfly/archive/2010/01/01/bizzyspinner-a-wpf-spinning-busy-sate-indicator-with-source.aspx

Production quality spinners can be found here:

http://mahapps.com/controls/progress-ring.html

https://github.com/100GPing100/LoadingIndicators.WPF

Demo Summary

- Run the simulation as a separate Task
- Update UI in context of UI thread

```
Task T = new Task( () =>
  {
    // Asian options simulation code:
  }
);
T.Start();
```

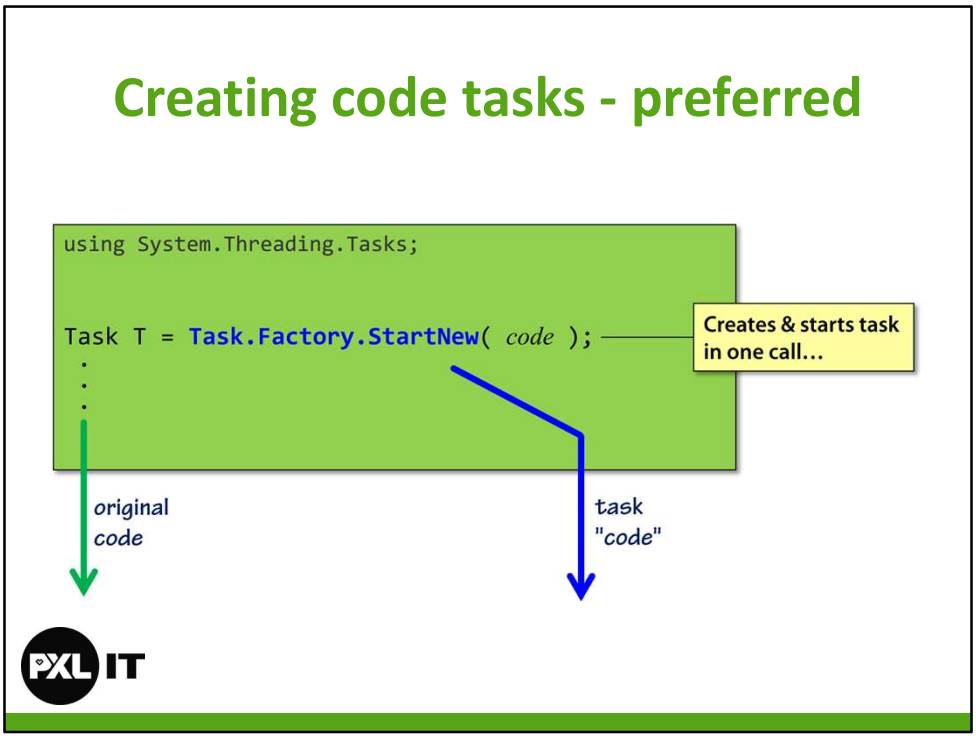
```
Task T2 = T.ContinueWith( (antecedent) =>
  {
    // code to update UI once simulation task is finished:
  },
  TaskScheduler.FromCurrentSynchronizationContext()
);
```

Demo: AsianOptions-Step3

T.ContinueWith

Wacht tot T gedaan is en ga verder met volgende taak => parameter antecedent

TaskScheduler.FromCurrentSynchronizationContext => UI thread

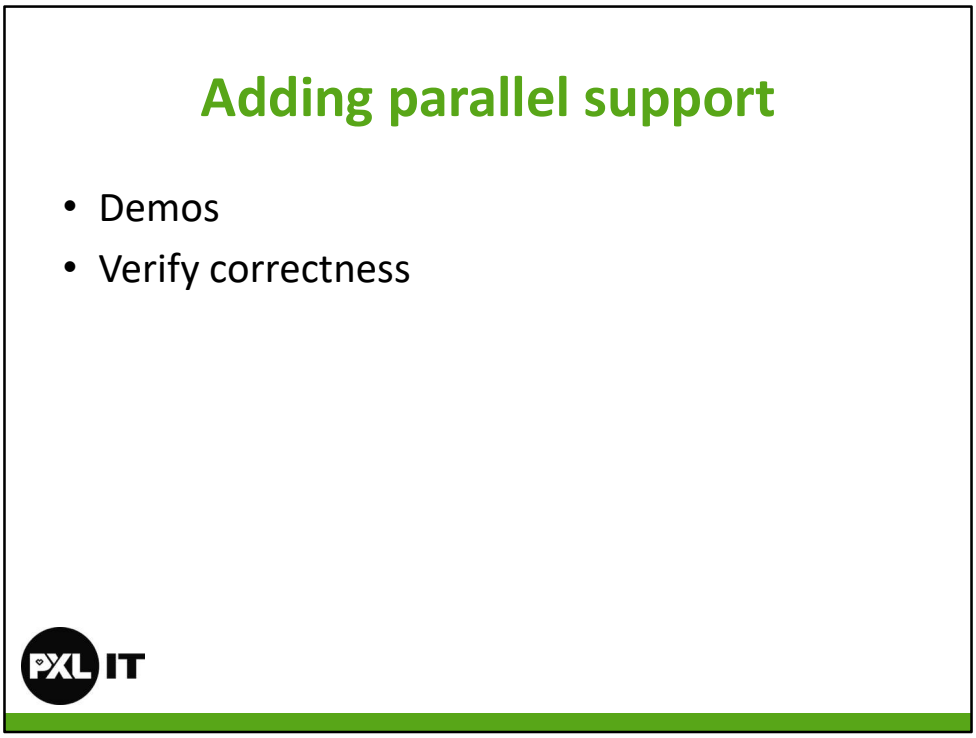


Task.Factory.StartNew

Create and Start in 1 call => more efficient

Nowadays, it is more common to do:

Task.Run



Demo: AsianOptions-Step4 en AsianOptions-Step5

Because variable m_counter is incremented en decremented in the same UI thread, there is no race condition.

Language support

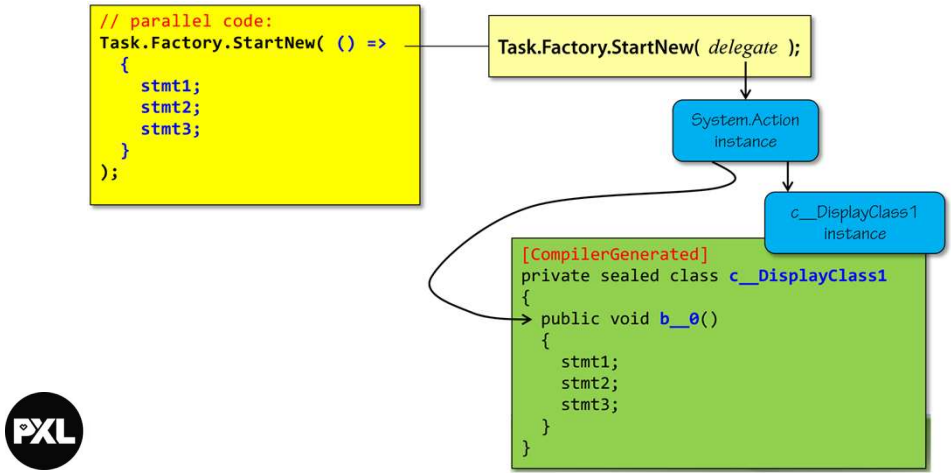
- Task Parallel Library (TPL) takes support of .NET language features:
  - Lambda expressions
  - Closures

Lambda expressions → already known

In orde to explain Closures, we need to know how lambda expressions are compiled to IL

Behind the scenes

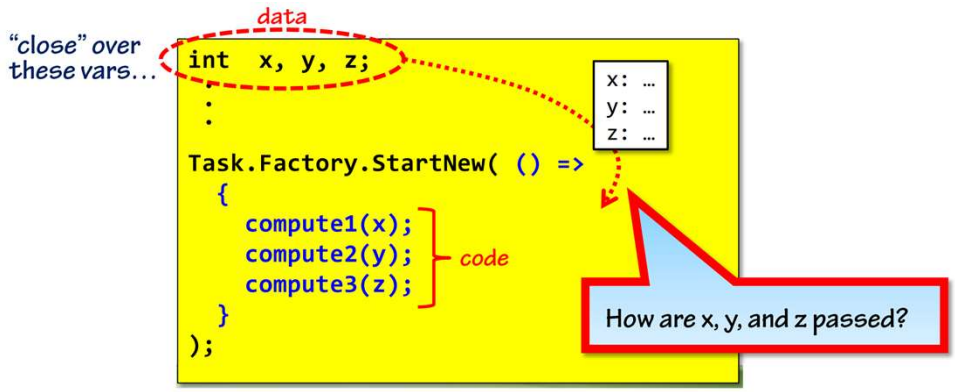
- Lambda expression == custom class + delegate



For each delegate, there is an Action instance which has a custom delegate that points to a method in a (generated) class.

Closures

- **Closure** == code + supporting data environment
- Compiler computes closure in response to lambda expression



How are x, y, z passed into the method body of the closure?

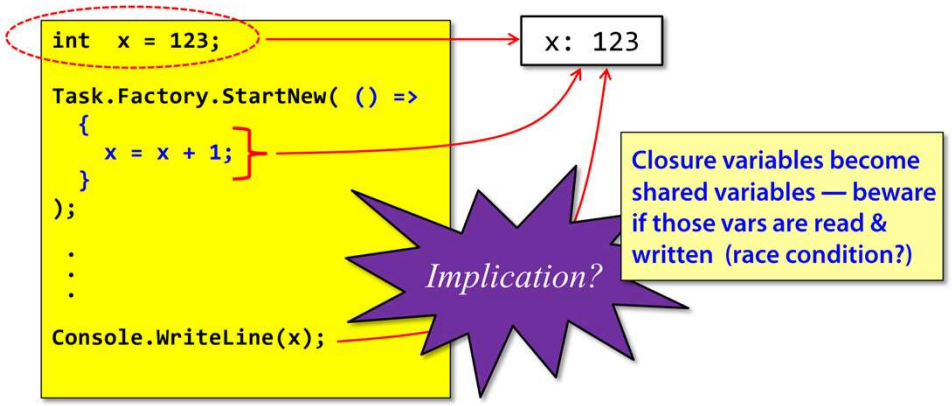
Compiler generates the necessary code to pass these in **by reference**, so x, y, z inside the lambda point to the same variables as x, y and z (this can be verified with ILSpy).

Note: even if the types are value types (int, struct, etc) !!

This is very handy, because you don't have to maintain parameters for methods. You simply know that the surrounding context from outside the lambda is available inside.

Closure variables

- Passed **by reference**



x is passed by reference into the lambda

This is a task → executes on a different thread

So what is the output? That depends on who is first → race condition.

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

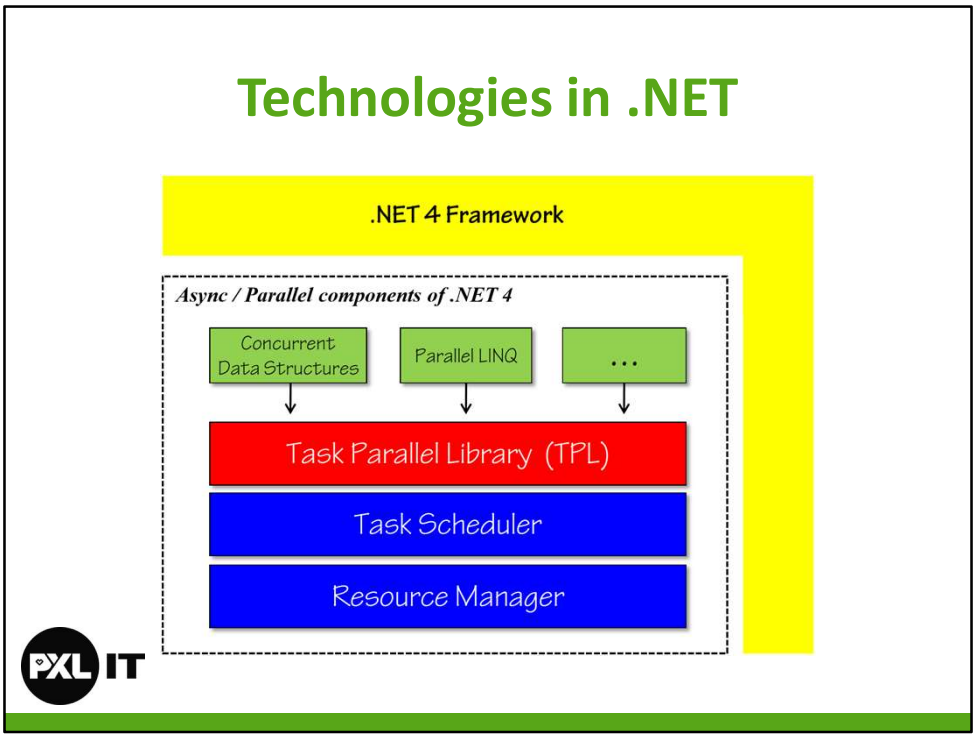
Contents

- 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. 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



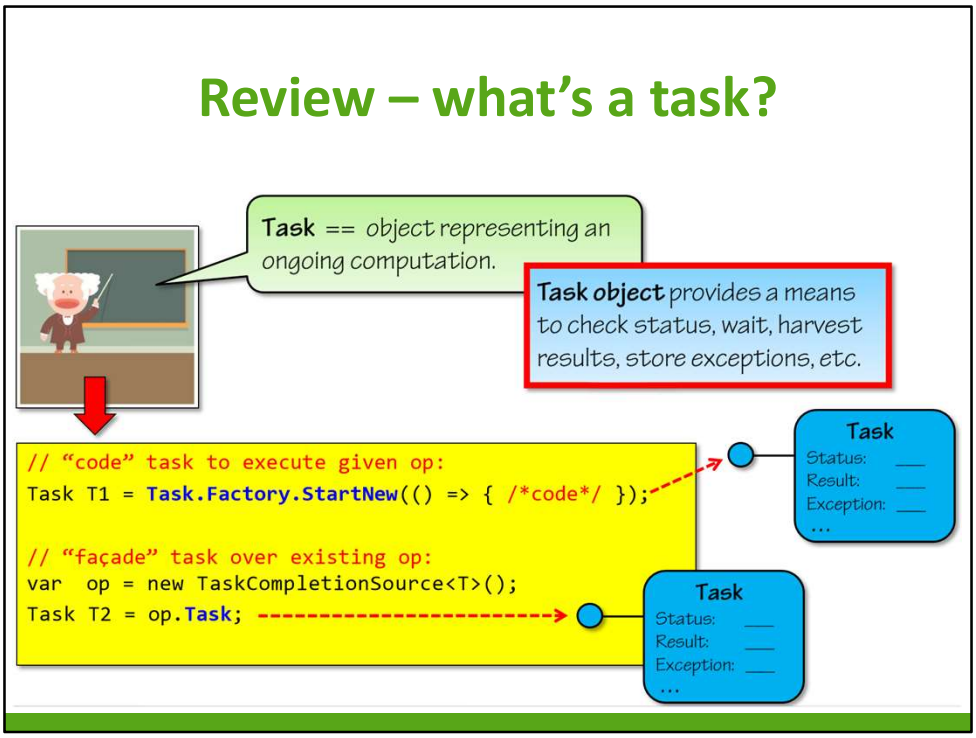
TPL provides the framework you use as a programmer (Task<T>).

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



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.

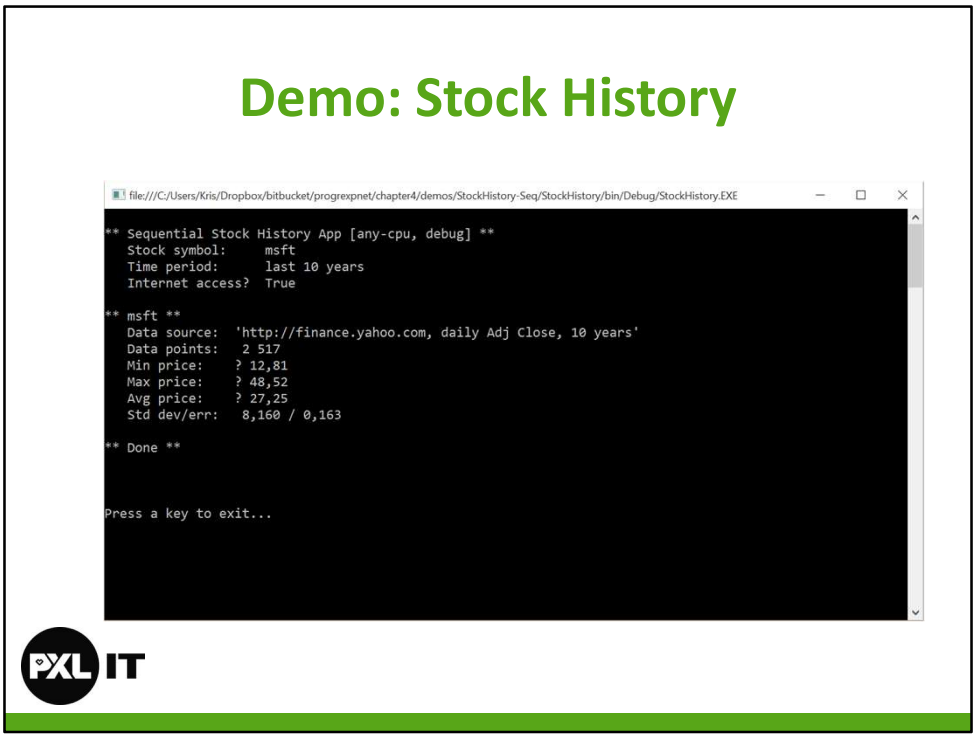
There are 2 types of tasks:

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. netwerk request)
- You create a TaskCompletionSource object and no Task object itself
- They are a façade over an existing (asynchronous) operation



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}",
```

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

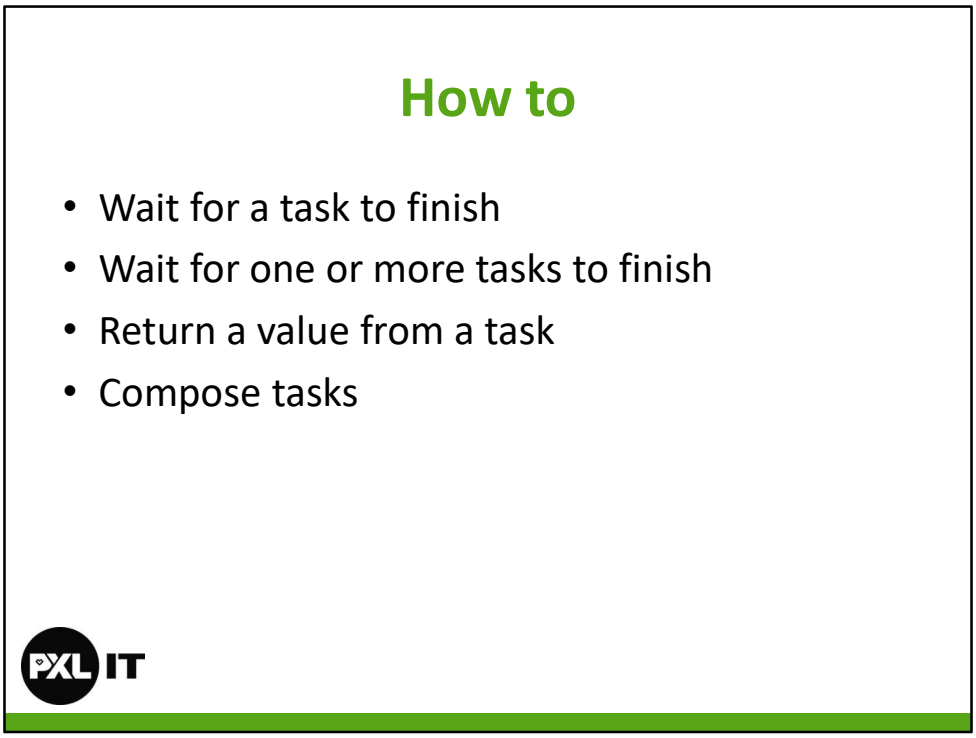
http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229468.aspx

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: IAsyncResult and WaitHandle are other techniques to create async requests.



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

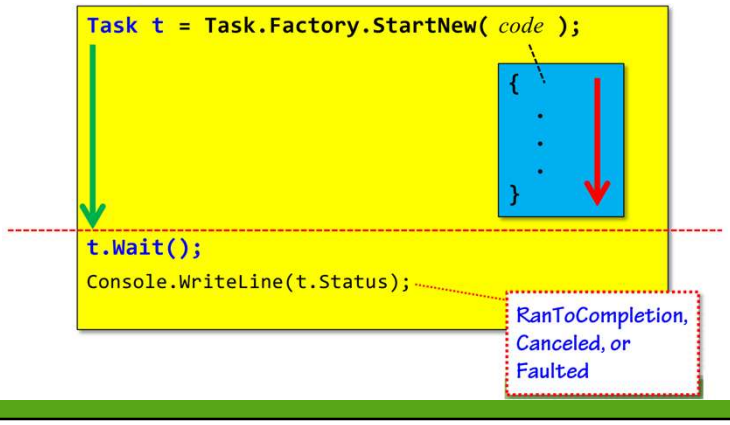
- Wait for a task to finish
- Wait for one or more tasks to finish
- Retrieve a result from a task
- Compose tasks

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

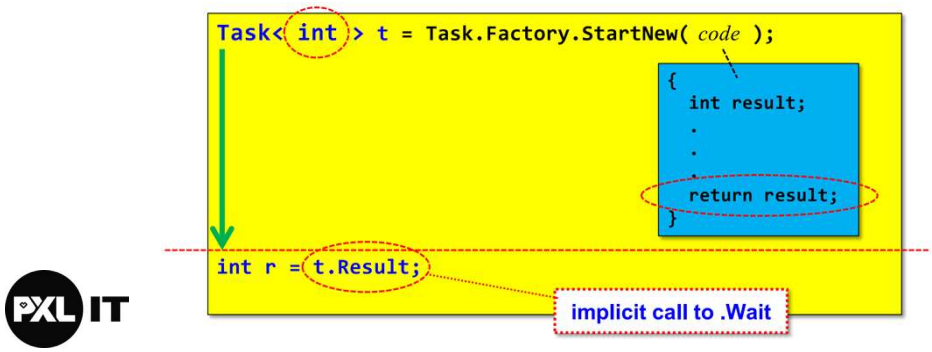
- Need to wait for a task to finish?
- Call **.Wait** on task object



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).

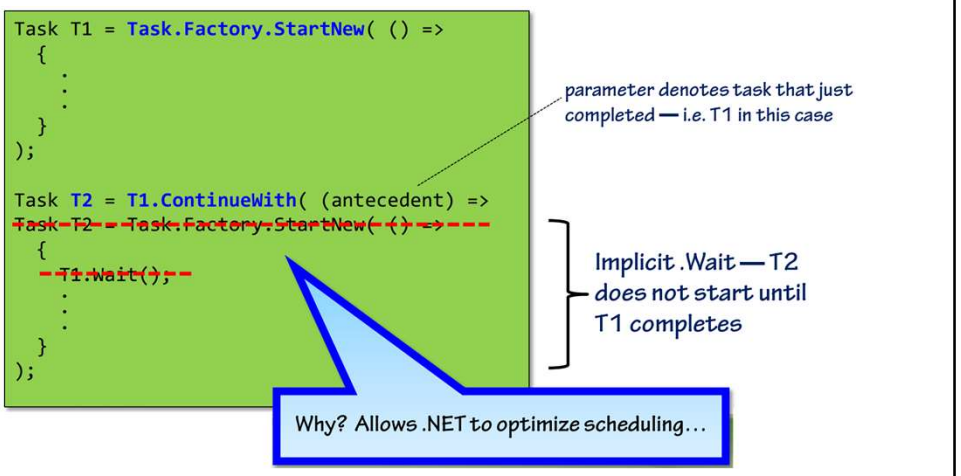
Waiting on multiple tasks

- 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

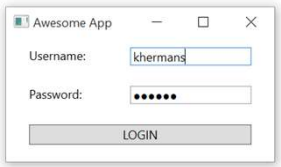
- The completion of a task triggers the start of another



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



PXL IT

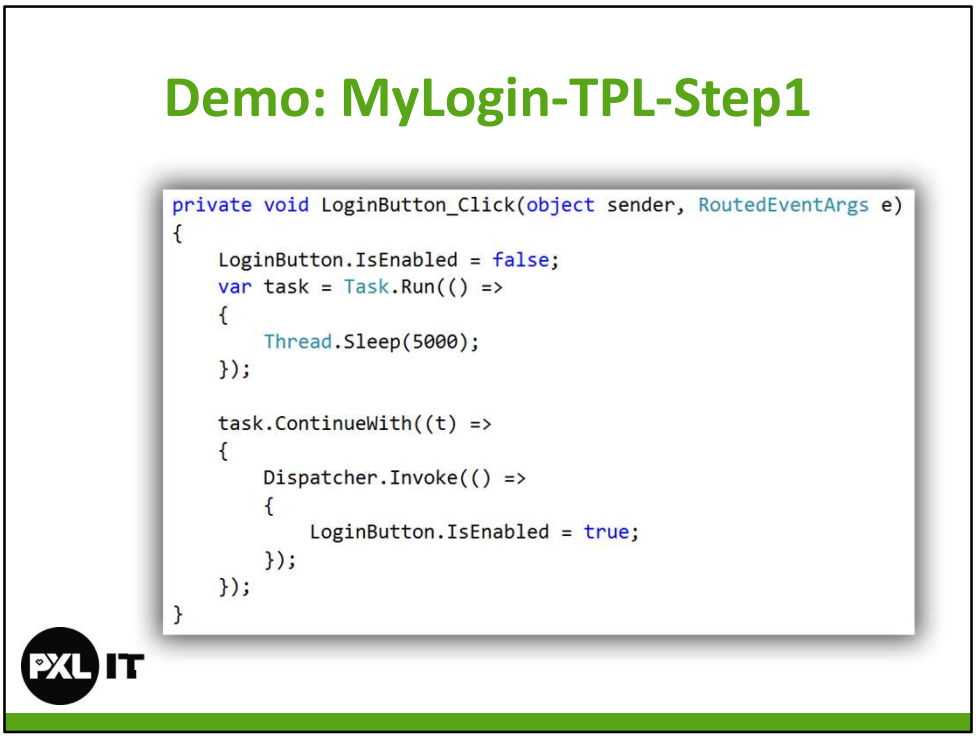
Demo: MyLogin-Sync

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 taks 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



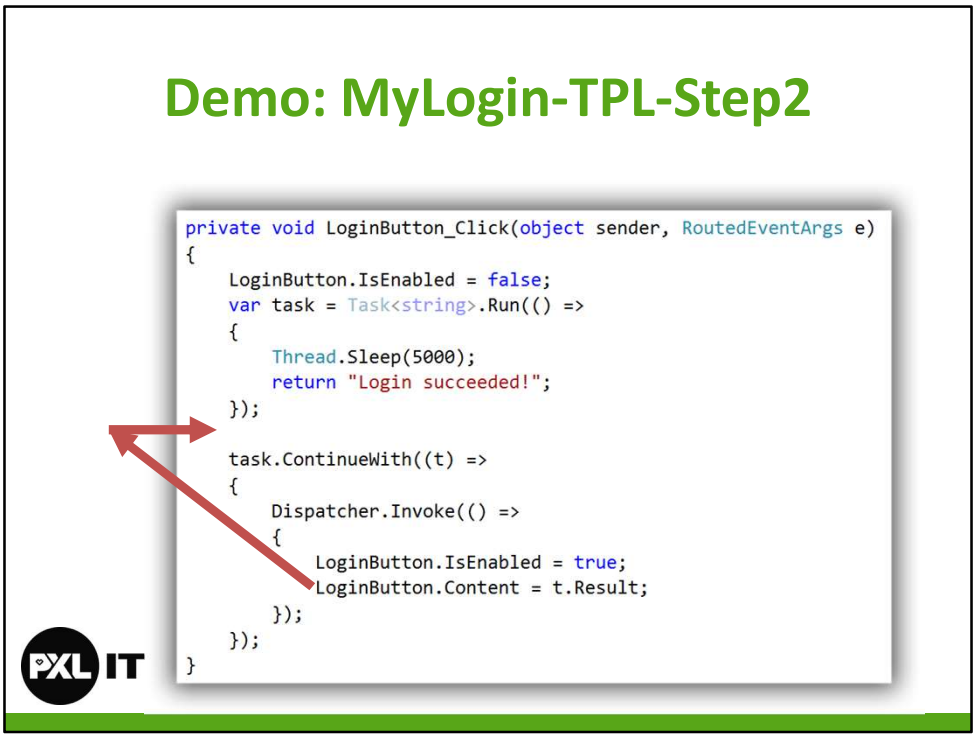
Demo: MyLogin-TPL-Step1

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:

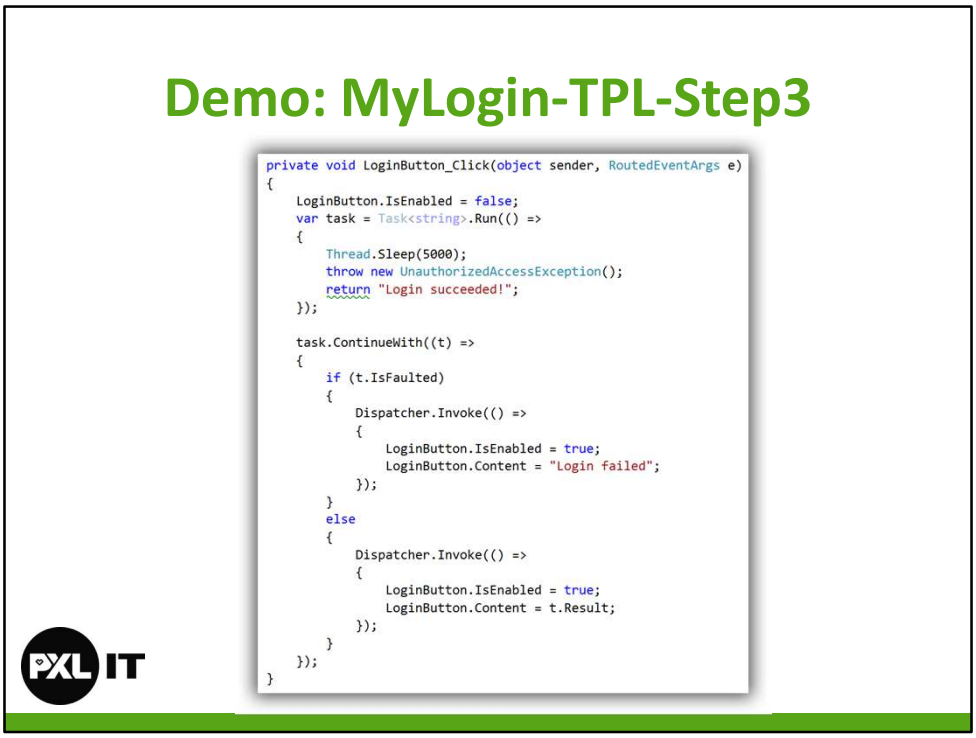
```
task.ContinueWith((t) =>
{
   LoginButton.IsEnabled = true;
}, TaskScheduler.FromCurrentSynchronizationContext());
```



Demo: MyLogin-TPL-Step2

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?



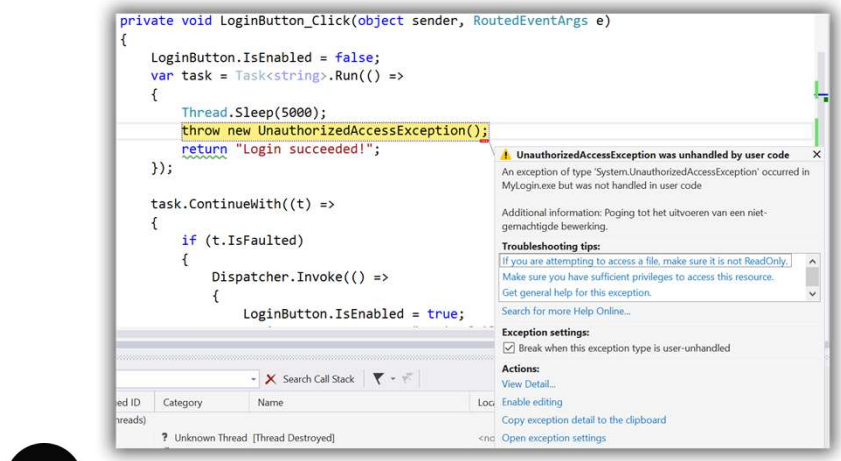
Demo: MyLogin-TPL-Step3

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…

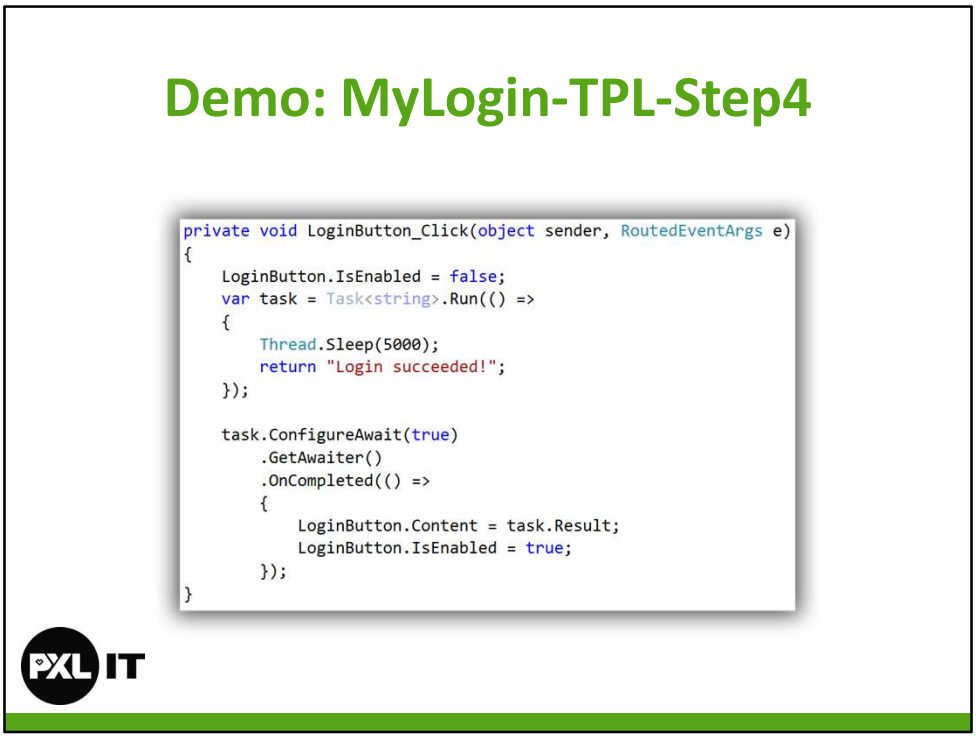
Demo: MyLogin-TPL-Step3



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

How to avoid the ContinueWith (continuation) task?

Task.ConfigureAwait(true)

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.

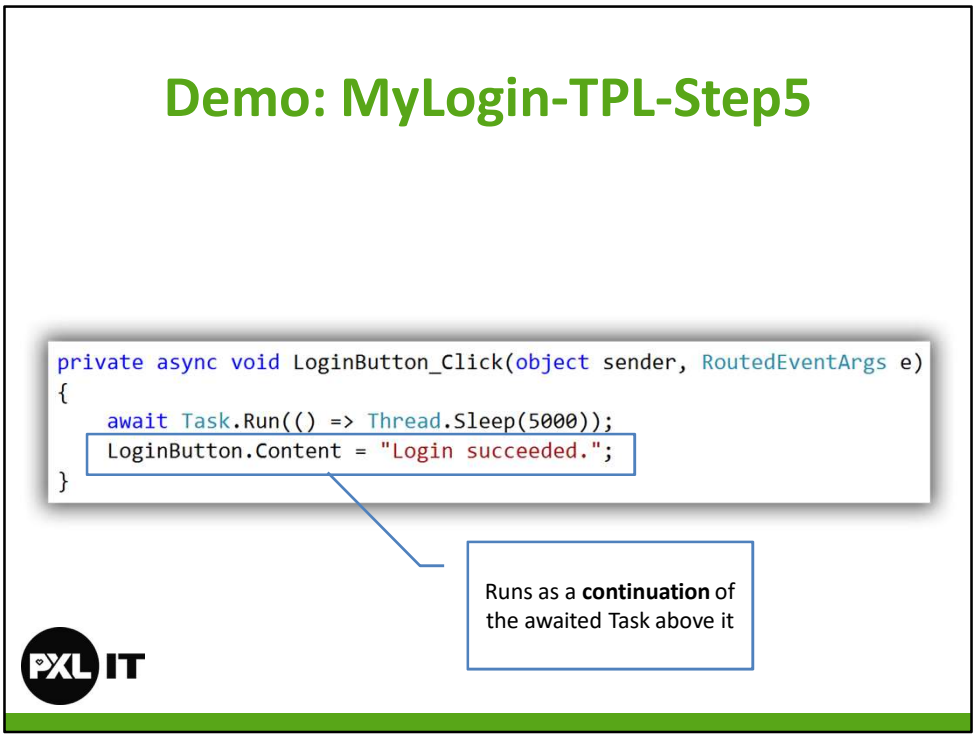
GetAwaiter() → get the waiting object that waits on the task

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

OnCompleted → code to execute after the waiting is finished

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

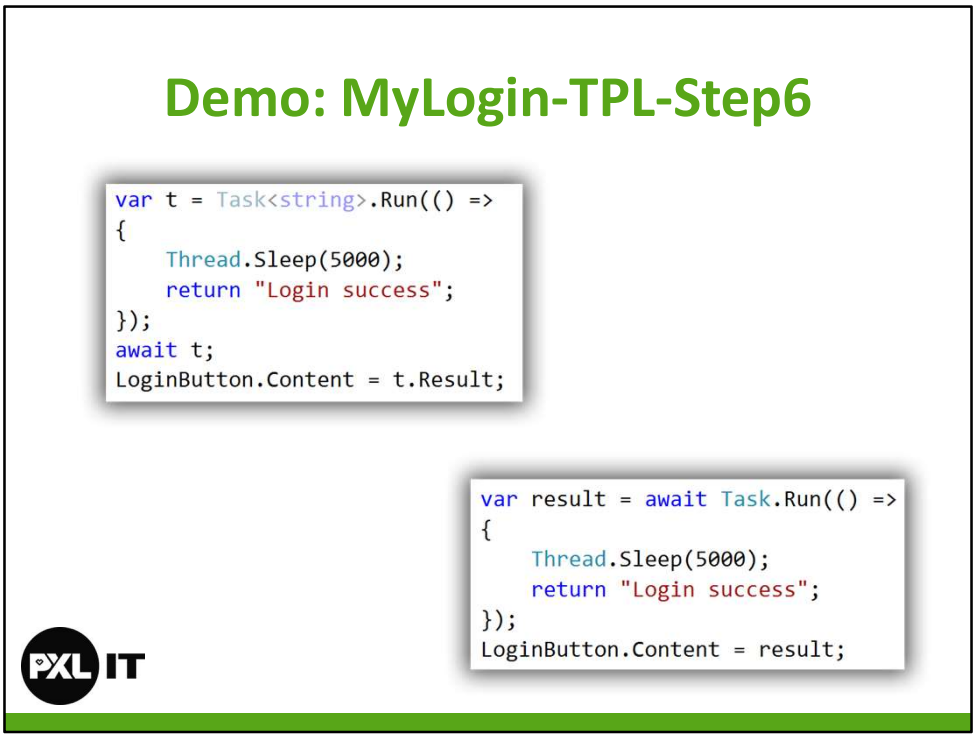
Mark the LoginButton_Click as async, now inside the body you can write asynchronous code, but not automatically! E.g. Thread.Sleep(5000) still blocks the UI thread.

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:

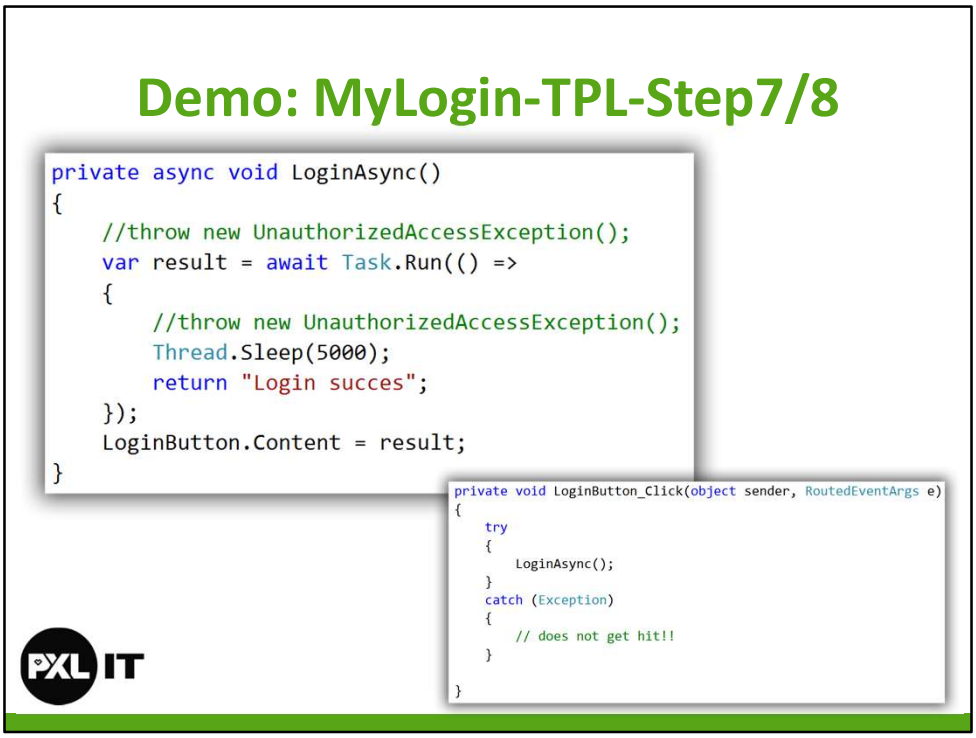
Everything below “await Task.Run” is executed inside a continuation of that task, inside the thread of the calling context => in this case the UI thread.



Demo: MyLogin-TPL-Step6

Modify so the task returns a string result that is placed on the button.

The two code blocks on the slide to the same thing.

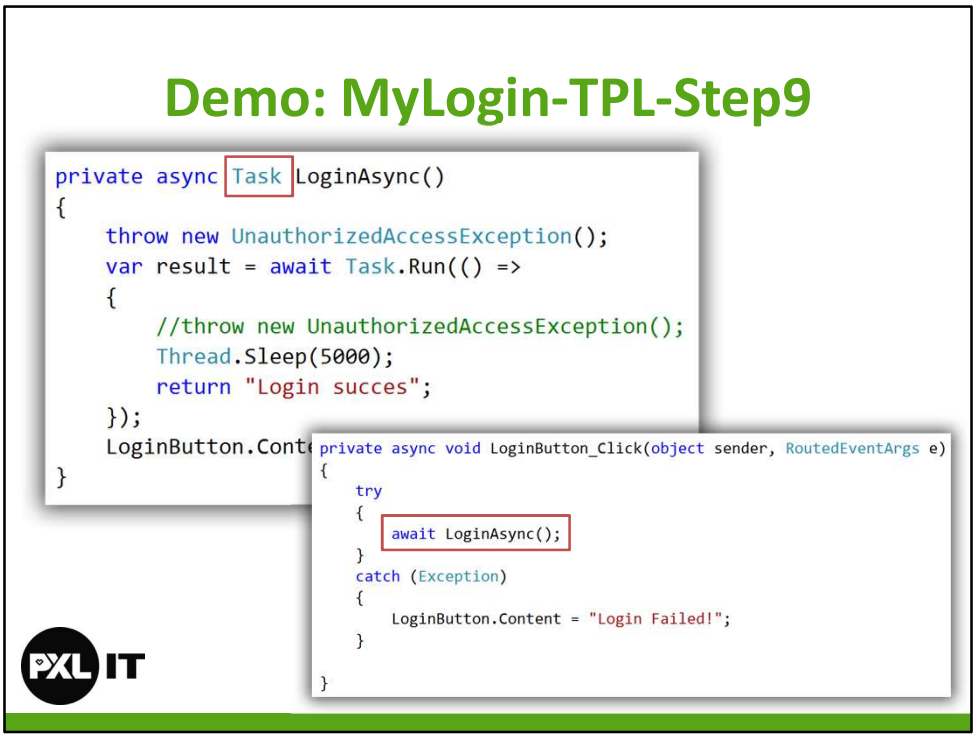


Demo: MyLogin-TPL-Step7 and MyLogin-TPL-Step8

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**

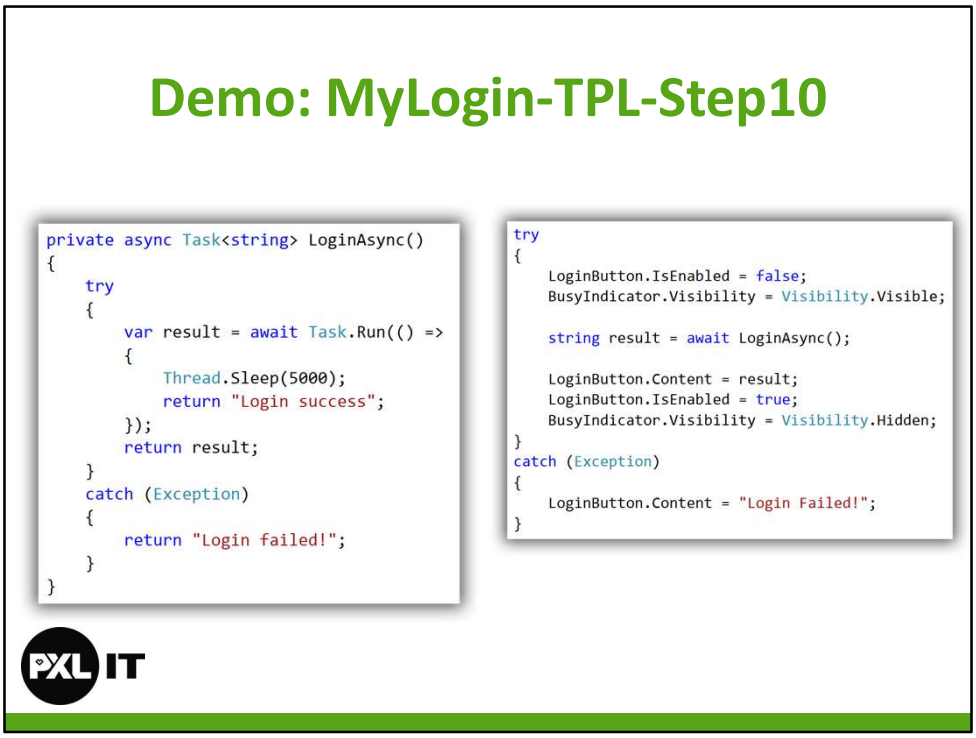
Now if you throw an exception inside this method (to simulate a login failure), the method does not get hit! How come?



Demo: MyLogin-TPL-Step9

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

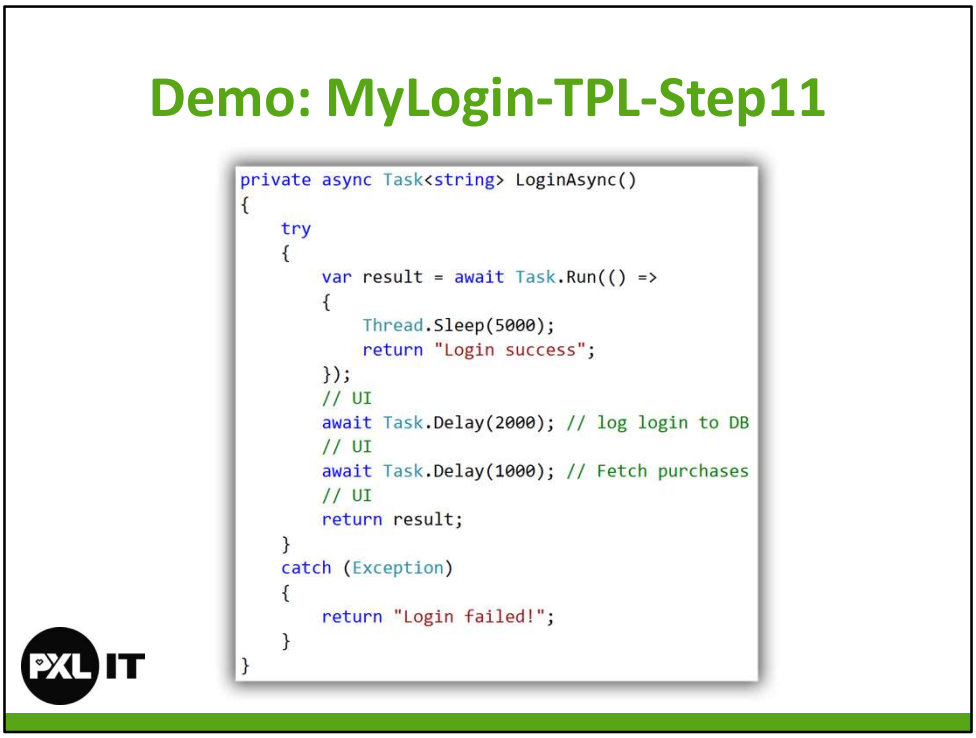
So: ALWAYS await your tasks!!



Demo: MyLogin-TPL-Step10

This demo demonstrates a far more responsive gui.

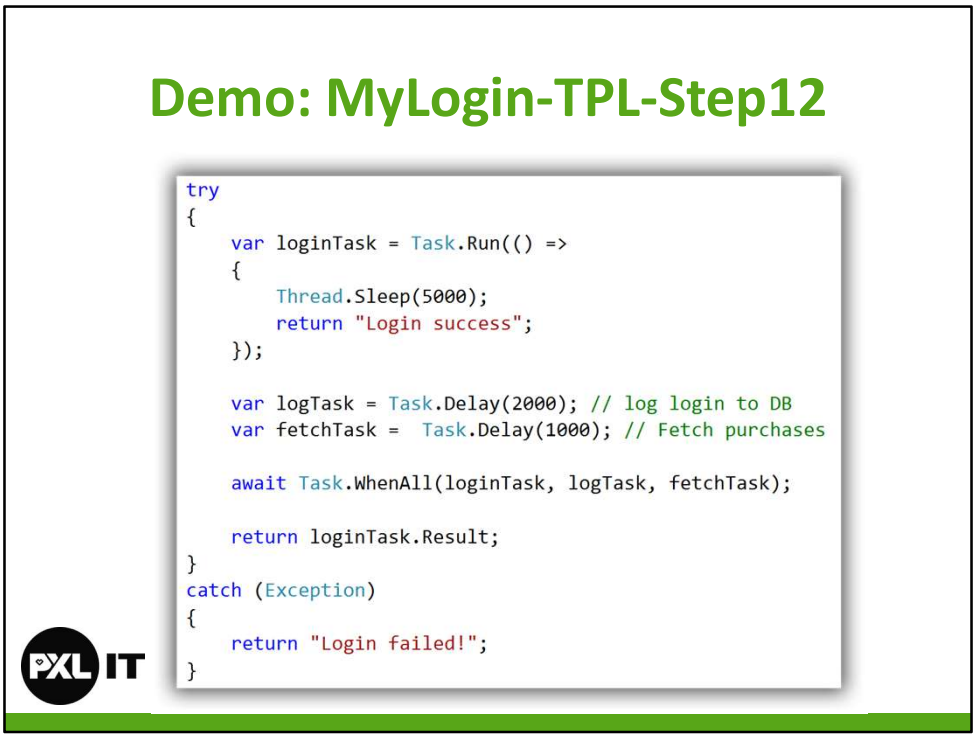
By using async / await → code becomes much cleaner!



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

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.

Garbage Collection

.NET

DE HOGESCHOOL
MET HET NETWERK

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

This lecture is based on the following Pluralsight Course:

http://www.pluralsight.com/courses/making-dotnet-applications-even-faster

Contents

- Basis GC concepts
- GC Flavors
- Generations
- Finalization

Garbage Collection

- GC means: you don't have to manually free memory
- GC isn't free and has performance trade-offs
  - Questionable on real-time systems, mobile devices, etc.
- The CLR garbage collector (GC) is an almost-concurrent, parallel, compacting, mark-and-sweep, generational, tracing GC
- Source code (open source) available

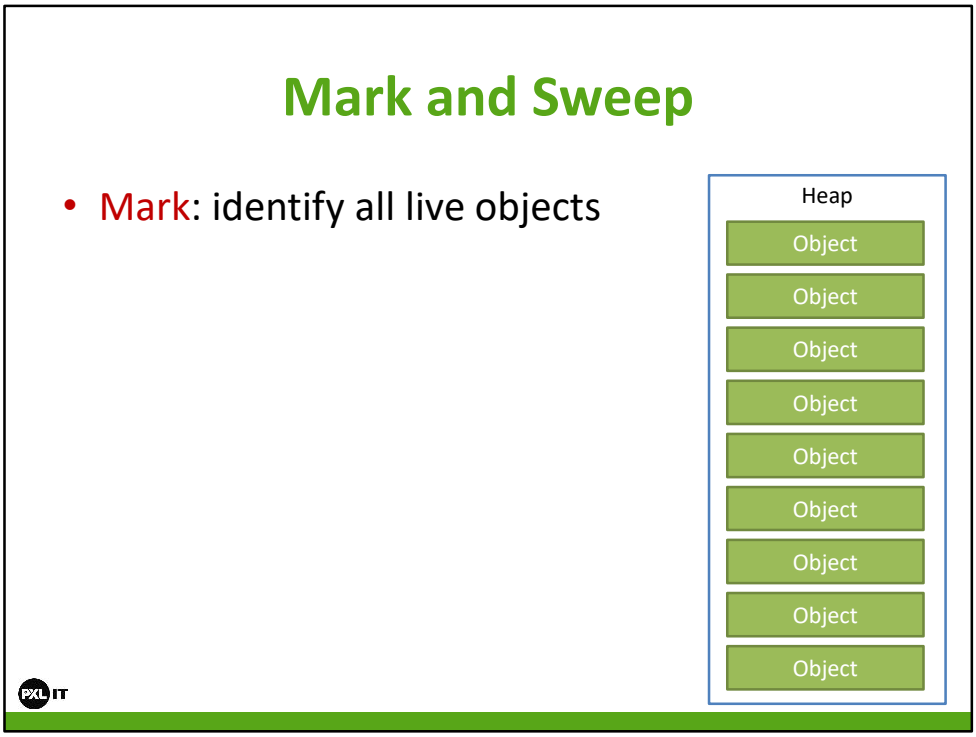
The GC is a run-time component that runs together with your program. It solves an **extremely difficult problem** of identifiying unused objects (or resources in general) and freeing up memory for new allocations.

For the curious, the source code is online (and documented):

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/garbage-collection.md

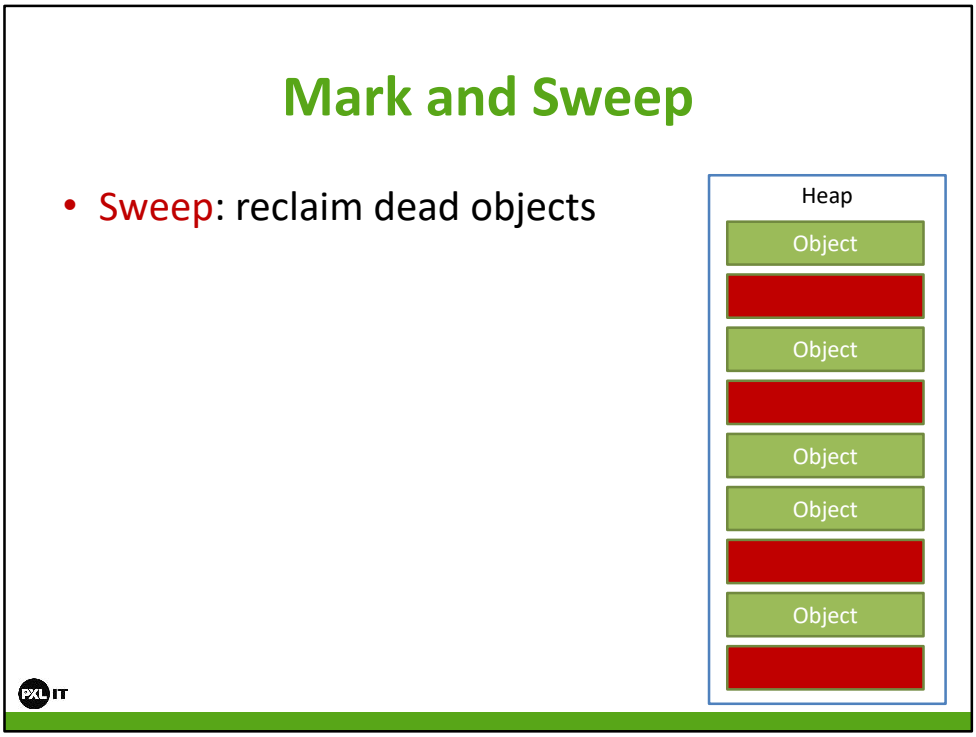
https://github.com/dotnet/coreclr/blob/master/src/gc/gc.cpp

We will explain the different properties of the GC CLR in the following slides.



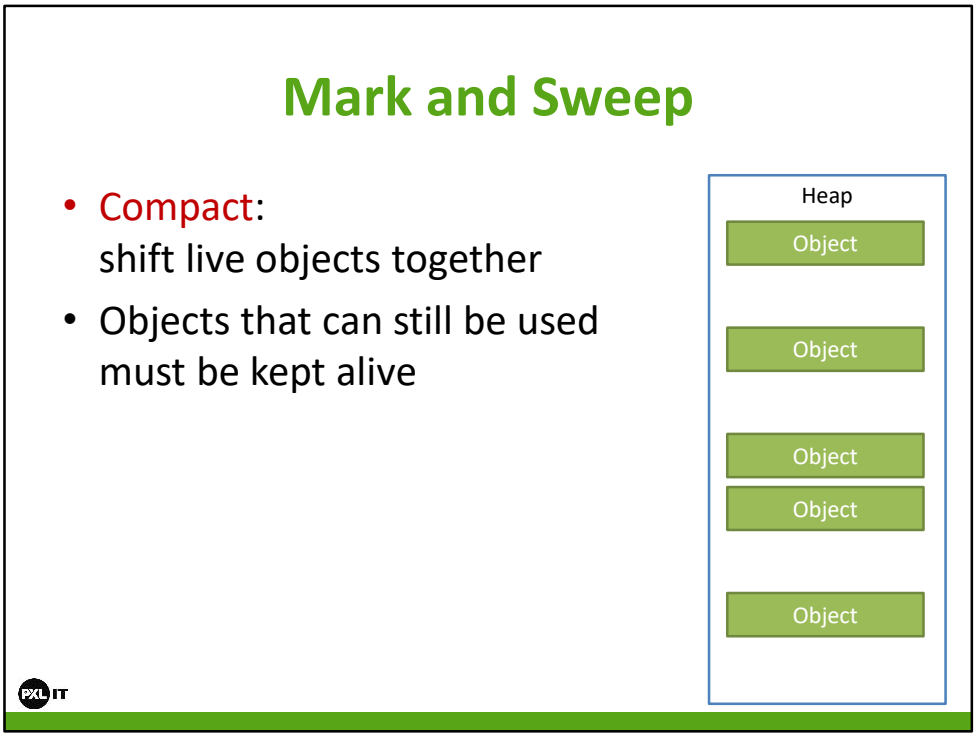
Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

In the **mark phase**, the garbage collector identifies all live objects in the heap, and this is done recursively. The GC starts from a group of objects that are certainly live and follows references between the objects until all the referenced objects are explored. The GC keeps track of objects it has already visited to avoid getting into an infinite loop if there is a reference cycle between objects.



Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

In the **sweep phase**, the garbage collector reclaims all unused memory that's occupied by dead objects.



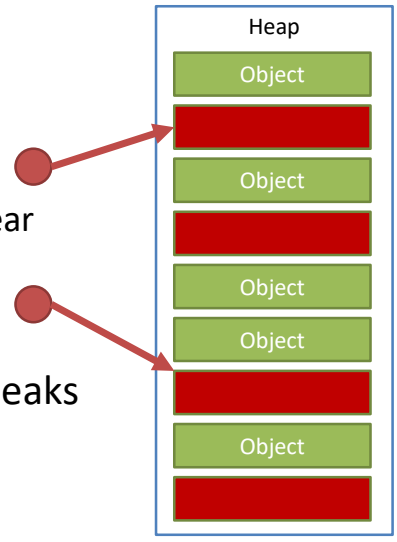
Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

The sweep phase often ends with a **compaction** phase. Compaction means the GC shifts live objects together so that they are consecutive in memory. The free space is then also consecutive, and this can make allocations cheaper.

But how does the garbage collection determine which objects are okay to die and which objects have to remain in memory. It all boils down to whether the program is still using these objects. Any object that the program still has a chance of using in the future has to remain, and it cannot be reclaimed by the GC. Any object that's completely unreachable can be reclaimed, and its memory can be used for future allocations. The problem, then, is that identifying those dead objects is hard, because they are unreachable, and the mark phase really is all about finding the live objects. The rest of the heap are considered dead objects by definition.

Roots

- Starting points for the gc
- Static variables
- Local variables
  - More tricky than they appear
- Other types:
  finalization queue etc.
- Roots can cause memory leaks



To find live objects, the garbage collector has to start somewhere. Some objects have to be considered alive even if they're not referenced by other objects. References to these root objects are called roots, and there're multiple kinds of roots in a .NET program.

Demo: Mod01_GCRoots

Conclusion: use GC.KeepAlive(obj) to manually extend the lifetime of an object.

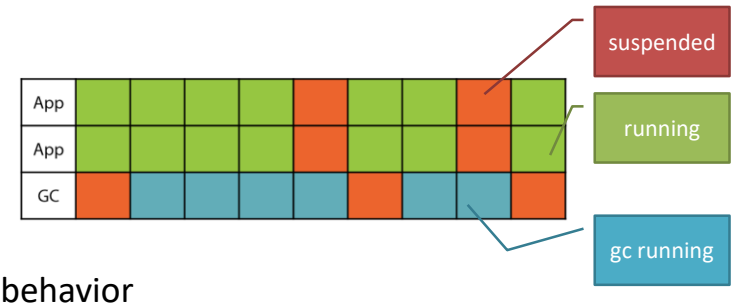
https://docs.microsoft.com/en-us/dotnet/api/system.gc.keepalive

Workstation GC

- There are multiple GC flavors
- Workstation GC is “kind of” suitable for client apps
- It is the default for almost all .NET applications
- GC runs on a single thread
- Workstation GC doesn’t use all CPU cores

The biggest problem with workstation GC today is that it does not take advantage of modern hardware: it runs on a single thread and doesn’t use all CPU cores.

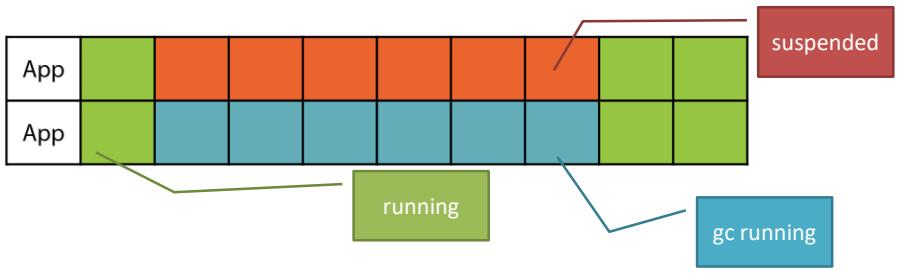
Concurrent Workstation GC



- Default behavior
- Special GC thread
- GC is scheduled and runs concurrently with application threads
- Only short suspensions of other threads

Let's look at concurrent workstation GC first, because, again, that's the default. In concurrent workstation GC, the CLR creates a special GC thread as soon as your application starts running, and this thread monitors the GC heap and performs garbage collections occasionally. It doesn't wait until all memory's exhausted, which would be wasteful. It tries to schedule garbage collections in a way that takes lots of memory and reclaims it without badly affecting application performance. The key here is that the GC thread runs concurrently with the application threads most of the time. Most parts of the garbage collection process can be done in the background while the other threads are running. Occasionally, the GC thread does have to suspend all the applications threads, but these suspensions are usually very short. And, as a result, if you have lots of memory allocations in your app, the GC thread will work hard in the background and only occasionally stop your other threads.

Non-concurrent Workstation GC



- One of the app threads does gc
- All threads are suspended during gc
- No real reason anymore to use it

In non-concurrent workstation GC, there is no special GC thread. One of the application threads is picked out to do the GC when it performs an allocation that would trigger a garbage collection. And that application thread, of course, doesn't do anything else during GC. But neither do the other threads. Non-concurrent GC actually has to suspend all the application threads for the duration of the GC. And that probably sounds like a bad idea, because in most cases, it is a bad idea. If the GC takes two seconds, your whole app stops responding for two seconds.

Server GC

- One GC thread per logical processor, working concurrently
- Until CLR 4.5, server GC was non-concurrent
- In CLR 4.5, server GC becomes concurrent
  - Now a reasonable default for many high-memory apps

With server GC, the main difference is that there are multiple GC threads. The CLR creates a GC thread for each logical processor in fact. So, for example, if you're on a quad core i7 processor with hyper-threading enabled, Windows thinks you have eight logical processors, and so your .NET app will have eight GC threads. When a GC is needed, these threads are just all awakened at once and start churning away. Garbage collection can be parallelized quite well, so you'd see considerable speedups in most cases if you switched to server GC. To further improve GC performance, server GC also has a separate heap area for each logical processor. So when a thread running on processor 0 allocates memory, it doesn't contend with another thread that happens to be running on processor #1 and also allocating at the same time. And this also improves cache locality for those threads because objects allocated closely in time will also be close together in memory. Now the only practical reason to avoid server GC used to be the fact that it didn't have a concurrent flavor. Until CLR 4.5, server GC was non-concurrent, which means all the application threads were blocked while doing garbage collection, and that could introduce unacceptable delays, especially in client applications again. But even so, it still sometimes made sense to switch to server GC in a client application just because it speeds up the overall collection process so much. If server GC turns a one-second collection into a 200-msec collection speeding it up by a factor of five, it might be worth the pain of suspending all app threads during the collection. In CLR 4.5, however, there now is a concurrent server GC flavor, and that's the best of both worlds. You have multiple threads doing GC, and your application threads can keep

working at the same time. So, if you have an application that spends a lot of time in the garbage collector, I think it's a pretty good idea that you should check server GC out.

Switching GC Flavors

- Configure preferred flavor in app.config
  - Ignored if invalid (e.g. concurrent GC on CLR 2.0)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <gcServer enabled="true|false" />
    <gcConcurrent enabled="true|false" />
  </runtime>
</configuration>
```

- You can't switch flavors at runtime
  - You can query flavor using GCSettings class

GCSettings

https://docs.microsoft.com/nl-nl/dotnet/api/system.runtime.gcsettings?view=netframework-4.7.1

For .NET Core projects, you edit the .csproj file to include the following properties:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
  <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>
</PropertyGroup>
```

https://docs.microsoft.com/en-us/dotnet/core/tools/project-json-to-csproj

Generational garbage collection

- A full GC is expensive and inefficient
- Divide the heap into regions and perform small collections often
  - Modern server apps can't live with frequent full GCs
  - Frequently-touched regions should have many dead objects

- New objects die fast, old objects stay alive
  - Typical behavior for many applications, although exceptions exist

PXL IT

One Big Heap is not efficient: try to avoid full gc.

Divide into regions => many dead objects (gc first) and mostly alive objects (gc not usefull)

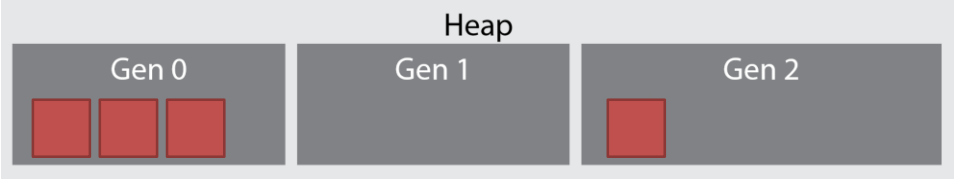
Observation: **new** objects die fast, **old** objects stay alive.

The younger the object, the more likely it will be gc soon.

For example, in a web server, there are many objects created when the server is processing a request. They will all die when the request is finished. On the other hand, old objects that were created during server initialization, like singletons and managers, they're probably going to stay alive until the end of the process. In the .NET garbage collector, there are three areas for objects of various ages.

.NET Generations

- Three heap regions (generations)



- Gen 0 and gen 1 are typically quite small
  - A high allocation rate leads to many fast gen 0 collections
- Survivors from gen 0 are promoted to gen 1 and so on
- Make sure your temporary objects die young and avoid frequent promotions to gen 2

Generation 0 is where objects are born. Generation one is the nursery and generation two is for tenured old objects that are likely to survive for a long time. The garbage collector will try to perform lots and lots of generation 0 collections and touch generation two only infrequently. Because most objects in generation 0 are young, they die fast, so the generation 0 collections are going to be very efficient and will reclaim lots of unused space. The sizes of the three generations depend on numerous runtime parameters. Generations are larger on 64-bit systems than on 32-bit systems. Generation sizes depend on the CPU cache size and even the amount of physical memory. You can also control the generation size by the application hosting the CLR, such as the ASP.NET host. But a typical generation 0 size on a 32-bit system is going to be a few MB, and that's something that fits in a CPU cache and means a garbage collection can be extremely fast. And most generation 0 garbage collections on modern hardware should complete in less than a msec. So you could have hundreds of them per second. Because generation 0 is so small, it fills up very fast. An application that allocates small objects at a rate of 500 MB/sec is going to cause a generation 0 collection every few msec. But, again, these collections are very fast. So we can live with that expense. When a garbage collection is complete in generation 0, the surviving objects are promoted to generation one, and that's the nursery. These objects are still considered young. The GC still assumes it's useful to run frequent collections in generation one. Most of the objects should have got here by accident, accidentally surviving a generation 0 collection. Finally, objects that survive a generation one collection are promoted to

generation two, and that's when the fun ends. Generation two is the resting place of tenured objects that should not die soon. The GC really tries its best to never touch objects in generation two. Once in a while, there might be a gen two collection, and these can take very long because generation two does not have a size limit. So, ideally, there should be very few generation two collections in your application. In terms of best practices, it would make sense to say that you should strive to work within the limits set by the generational model. You should try to make sure that your temporary objects die young in generation 0 or generation one in the worst case, and you should be very careful to avoid frequent promotions of temporary objects into generation two.

The Large Object Heap

- Large objects are stored in a separate heap region (LOH)
  - Large means larger than 85,000 bytes or array of >1000 doubles
- The GC doesn't compact the LOH
  - This may cause fragmentation
- The LOH is considered part of generation 2
  - Temporary large objects are a common GC performance problem

Large objects go into a separate heap region, which is logically a part of generation two. This region is called the **large object heap**, or **LOH**, as opposed to the rest, which is the small object heap, or **SOH**. Currently, the CLR considers objects that are larger than 85,000 bytes to be large. There is also a special exception for arrays of doubles. An array of more than 1,000 doubles is also considered a large object, even though it can be much smaller than 85,000 bytes. This heuristic dates to .NET 1.0, and it hasn't changed since. What used to be large in 2001 isn't that large today, but the threshold remains as it is.

There is an important difference between the GC behavior in the LOH and the SOH. In the LOH, the GC does not compact the heap. The idea is that compaction requires copying objects around, and copying large objects is pretty expensive. The downside is that there can be fragmentation in the large object heap as a result. How does the large object heap actually affect us in real-world programs? The main thing to understand is that *large objects are automatically considered old because they are placed in generation two*. So if you end up creating lots of temporary large objects, you're going to be putting pressure on the garbage collector.

Explicit LOH Compaction

- LOH fragmentation leads to a waste of memory
  - .NET 4.5.1 introduces LOH compaction

```
GCSettings.LargeObjectHeapCompactionMode =
                GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

- You can test for LOH fragmentation using the !dumpheap -stat SOS command

Try to avoid as much as possible.

Foreground and background GC

- In concurrent GC, application threads continue to run during full GC
- What happens if an application thread allocates during GC?
  - In CLR 2.0, the application thread waits for full GC to complete

  - In CLR 4.0, the application thread launches a foreground GC

- In server concurrent GC, there are special foreground GC threads
- Background/foreground GC is only available as part of concurrent GC

CLR 4.0 introduced another optimization that has to do with garbage collection and generations.

The problem it's trying to solve is the following: suppose you have a concurrent garbage collection working in the background and collecting generation two. Because garbage collection is concurrent, application threads are still allowed to run most of the time, and they can even allocate additional memory. But what happens if an application thread hits the generation 0 cap (= full)? There is already a garbage collection in progress, and it's cleaning up generation two. So, if you have a thread that hits the generation 0 limit, it has to suspend and wait for the full GC to complete, and that can take a while. And that makes concurrent garbage collection look much less attractive.

So CLR 4.0 introduces **background garbage collection for workstation GC** as an extension for concurrent garbage collection. It's in a separate GC flavor. It's just how concurrent GC works after CLR 4.0. When you have background GC, the background GC thread performs generation two collections and doesn't suspend application threads. But if you have an application thread that hits the generation 0 allocation cap, it actually launches a foreground collection on the application thread. So, the foreground GC waits for an acknowledgement from the background GC thread. The background GC thread says, "Hey, I just suspended myself," which usually happens quite quickly, and then the

foreground GC cleans up generation 0, possibly also generation one, and then it resumes the background GC thread. And this solves the problem of allowing threads to exceed the generation 0 limit, even many times during a generation two collection that's still happening in the background.

In CLR 4.5, there is **background GC for the server** as well. There are some slight differences. The major one being that on the server, there is a background GC thread per logical processor and a foreground GC thread per logical processor.

Resource Cleanup

- The GC only takes care of memory, not all reclaimable resources
  - Sockets, file handles, database transactions, etc.
  - When a database transaction dies, it has to abort the transaction and close the network connection
- C++ has destructors: deterministic cleanup
- The .NET GC doesn’t release objects deterministically

In an ideal world, the garbage collector could take care of all the resources our application needs. In the real world, the GC can only take care of memory. It's very, very good about reclaiming memory, but it's not so good at reclaiming other kinds of resources. Anything else that needs to be closed or disposed or cleaned up explicitly that isn't memory. So, we're talking about file handles, network sockets, database connections, database transactions. These are all examples of resources that have to be cleaned up because they refer to something that isn't just memory. For example, when a database transaction dies, it has to tell the database to either commit or abort the transaction. And it also possibly has to close an open network connection to the database server. Both of these cleanup operations must be explicitly performed. They're not a part of normal memory management, which is what the GC is good at.

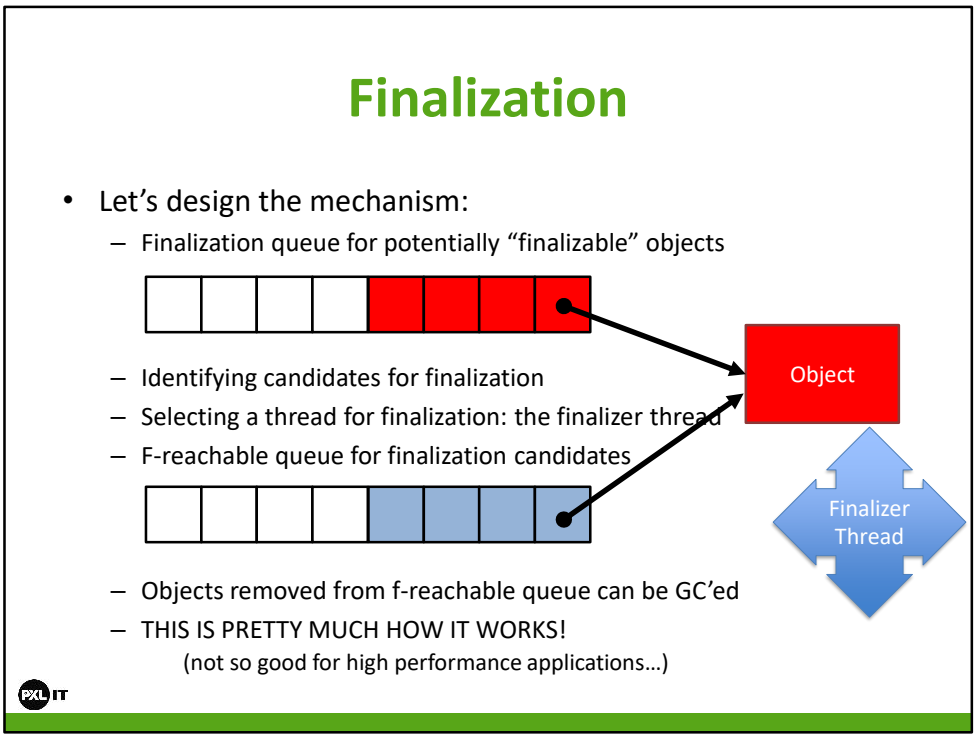
Now, in languages without garbage collection, such as C++, you often have deterministic resource cleanup in the form of destructors. In C#, objects do not die at the deterministic point in time. The garbage collector reclaims unused memory at some point in the future after the object has become unreachable. And this actually makes it hard to plan for the releasing of non-memory resources.

Finalization

- The CLR runs a finalizer after the object becomes unreachable

```
class Resource
{
    ~Resource()
    { /* cleanup */ }
}
```

The .NET GC has the ability to run cleanup code automatically at some point in the future after an object becomes unreachable. The programmer has to write a special method called the **finalizer**, which the garbage collector invokes. This method is the class's opportunity to release any non-memory resources, such as network sockets and database transactions.



But how exactly does the GC run the finalizer if the object is already unreachable? There is no way to reach unreachable objects, you know, by definition, not to mention the call methods on unreachable objects. So, let's try to design this ourselves and see that what we get is pretty close to how CLR finalization actually works.

First, we need a data structure for keeping track of all the objects that *potentially* require finalization. Let's call this the **finalization queue** and put an object in it when it's created. Of course, we should only put an object in the queue if it has a finalizer. The finalization queue should be a GC root, so we don't accidentally reclaim memory for an object that hasn't been finalized yet.

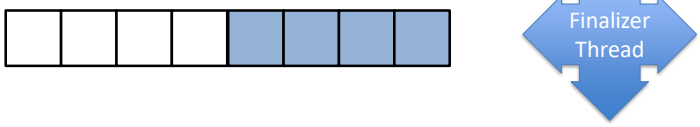
Next, we need a way to know that an object is *eligible for finalization*. And this is easy. We start marking the heap from all the roots except the finalization queue. And then, we'll look at the finalization queue. Any objects that weren't marked yet are now eligible for finalization because they're only referenced by the finalization queue.

Now suppose we have an object that has to be finalized. Just to remind you, we're in the middle of a garbage collection right now, so we can't really afford to stop in the middle of a GC and start calling user defined methods, which can take, you know, an unbounded

amount of time. So, here's an idea. Let's postpone finalization and schedule it on a different thread. Which thread exactly? It can't be an application thread. It can't be the GC thread. **So, let's create a new thread, the finalizer thread**. Now to put work on the finalizer thread, we need another queue of objects that are **ready for finalization**. We'll probably call this the finalizable queue, but the CLR calls it the **F-reachable queue**. Objects on this queue are live only because they are waiting for finalization. Finally, the finalizer thread picks objects from the F-reachable queue, runs their finalizers, and objects with finalizers that have run can be removed from the F-reachable queue and can be collected by the garbage collector. So, the next GC cycle is going to pick them up.

Performance problems with Finalization

- Finalization extends object lifetime
- The f-reachable queue might fill up faster than the finalizer thread can drain it



  - Can be addressed deterministic finalization (Dispose)
- It's possible for a finalizer to run while an instance method hasn't returned yet

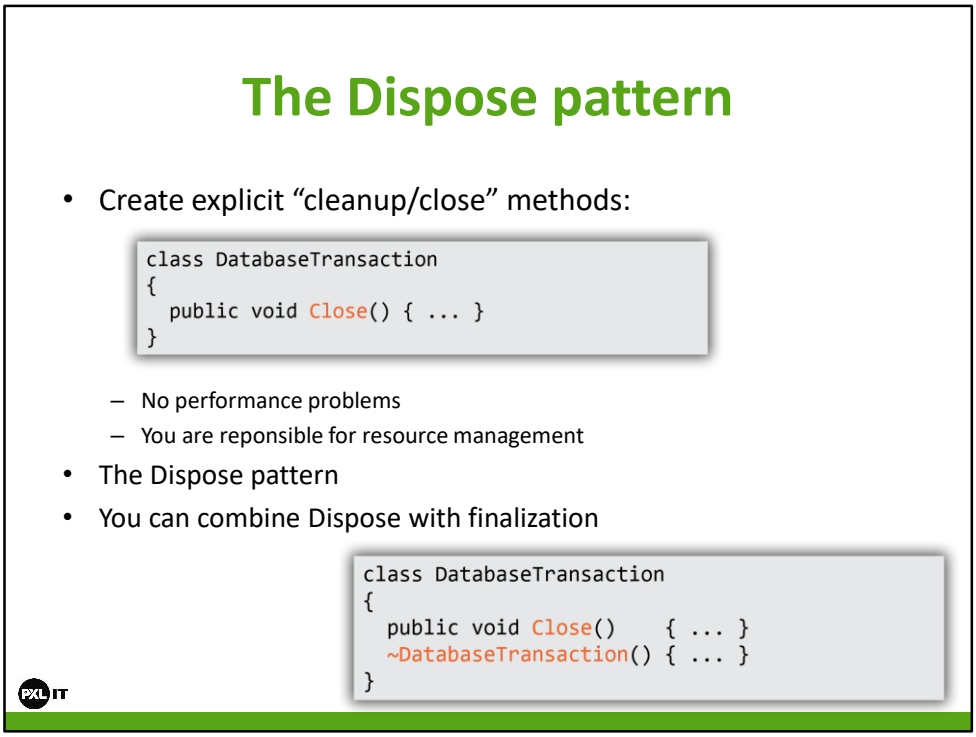
The *first problem* with finalization is that **it extends object lifetime**. An object will survive at least one garbage collection if it has a finalizer. Objects that used to die in generation 0 might die in generation one. Objects that used to slip into generation one will now slip into generation two. Now take many of these objects, and you've got yourself a performance problem, because there will be many generation two garbage collections.

The *second problem* with finalization is that it is **subject to an unbounded queue problem**. The F-reachable queue can fill at a faster rate than the finalizer thread can drain it. For example, suppose it takes 20 msec to finalize a database transaction object. Now if the application creates a hundred of these objects per second, there is an immediate memory leak. One hundred of these objects can be queued to the F-reachable queue every second, but the finalizer thread can only clean up 50 of these objects every second, because each object takes 20 msec to clean. This kind of memory leak can be pretty hard to discover during development. It might only manifest under heavy load or when finalizers run slower in the production environment than in development. It's a very unfortunate situation to be in, and we'll talk about ways to address this problem by moving finalization to the application thread and making it deterministic. And that's the dispose pattern in a nutshell.

The *third problem* I'd like to show you is rather subtle. The underlying problem is that finalization happens on a separate thread, which can introduce **nontrivial race conditions**. The crux of the matter is that it's possible for a finalizer of an object to run while the method of that very object is still executing (see demo).

Demo: race condition finalization

- Watch the demo on Pluralsight
  - Module 1, 6:46 – 13:26
- Key points:
  - Local variables can get cleanup before the end of their scope
  - Finalizer gets run while the object is still executing methods
  - Stay away from finalization and use deterministic cleanup



Just create a method that does the cleanup and call it when necessary. The biggest disadvantage? You have to call it!

This pattern is known as the **Dispose** pattern. In the Dispose pattern, you call the cleanup method Dispose, and you make the class implement the **IDisposable** interface. It's merely a convention, though. No one calls the Dispose method automatically for you. It's entirely your responsibility.

MSDN: https://docs.microsoft.com/nl-nl/dotnet/api/system.idisposable?view=netframework-4.7.1

Sometimes, it might also make sense to combine the Dispose pattern with a finalizer. The finalizer serves as a fallback in case the user forgot to call Dispose. It seems like the best of both worlds again. The finalizer could log a wording message or even fire an assertion to make sure next time, the developer isn't going to forget to call Dispose. For all the nitty-gritty details, consult the MSDN documentation on the Dispose pattern.

Resurrection and Object Pooling

- Bring an object back to life from the finalizer
- Can be used to implement an object pool
  - A cache of objects, like DB connections, that are expensive to initialize

```
class DatabaseConnection
{
  ~DatabaseConnection()
  {
    ConnectionPool.ReturnToPool(this);
    GC.ReRegisterForFinalize(this);
  }
}
```

A pretty cool and twisted thing that you can do from inside a finalizer is to bring the object back to life. After all, nothing stops you from creating a new root reference to the current object. The GC won't be able to reclaim it in that case. This technique is called **resurrection**, and it's often used to create object pools. The idea of an object pool is that instead of recreating objects every time you need them, your objects sit in a cache, and you can take them from the cache when you need them and return them back to the cache, or to the pool, when you're done with them. This can save the initialization costs for some expensive objects.

Database frameworks usually use this approach for pulling database connections instead of creating a new connection for every operation. Now what does it have to do with finalization? One way to return the object to the pool is by using a finalizer that will do this automatically. The finalizer can put the object back in the pool even if the user forgot to. But if you do this, consider what happens the next time the objects become unreferenced. There is no reference to the object from the finalization queue, so it's going to die right away. And to prevent this, we have the **GC.ReRegisterForFinalize** method, one of the most obscure in the .NET Framework that puts the object back in the finalization queue. You should call ReRegisterForFinalize when you return the object back to the pool, and then the next pool user is not going to lose the object forever.