Language execution basics

.NET

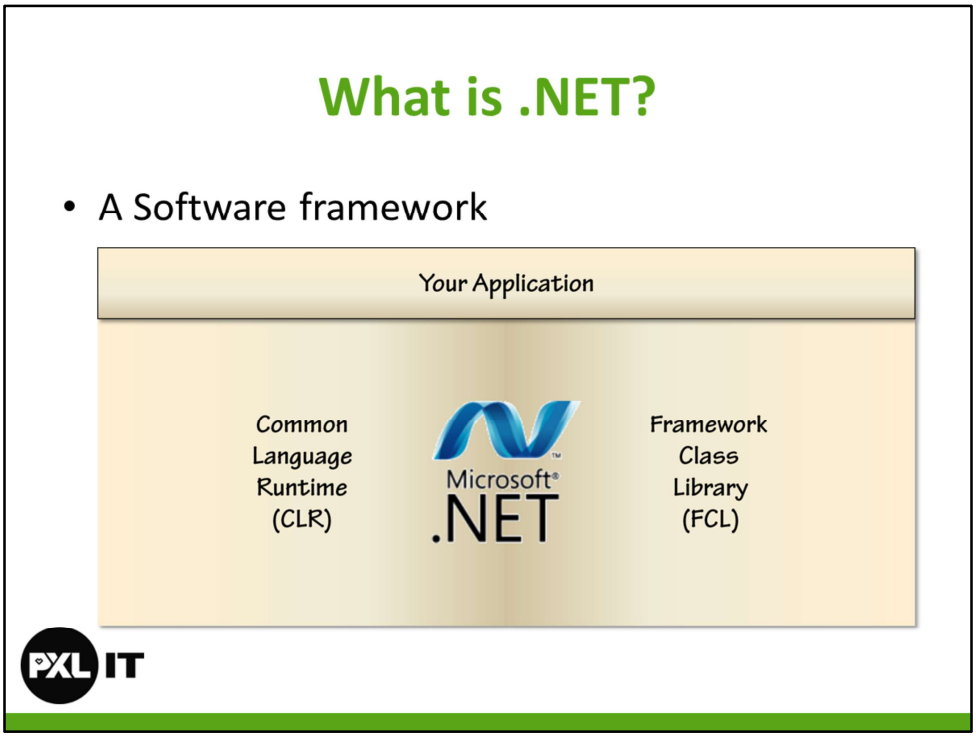
DE HOGESCHOOL
MET HET NETWERK

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)
  - var vs dynamic
  - struct vs class
  - enum

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.



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.

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

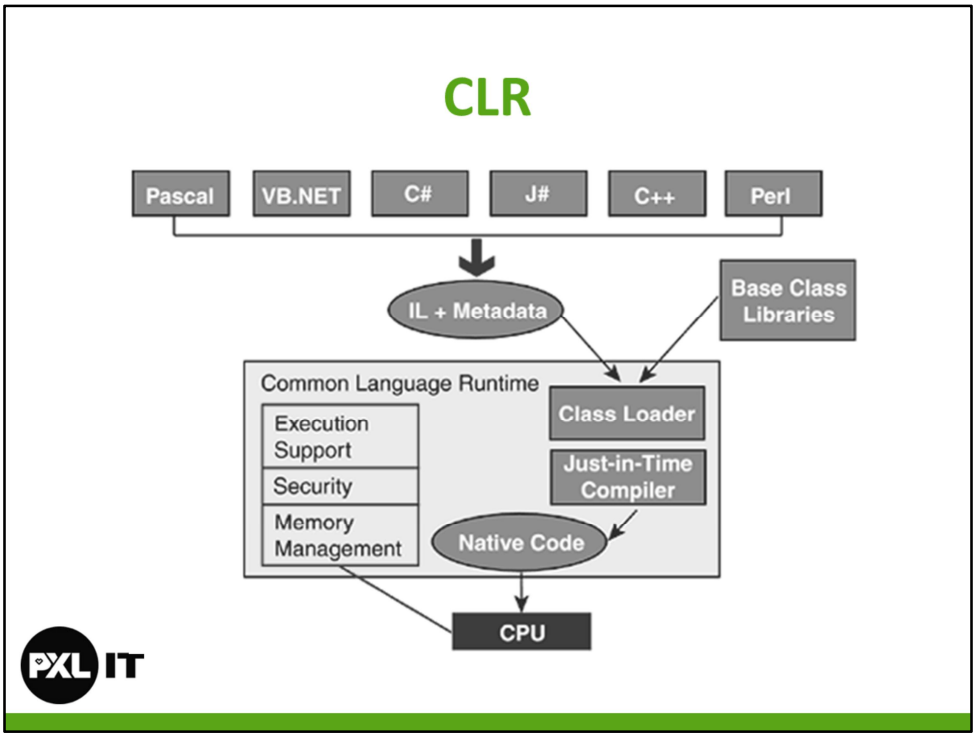
- Windows 95 → Windows 10
- Windows Phone
- Mono: Linux and Mac
- Xamarin: iOS and Android

FCL

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

Interesting to follow:

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

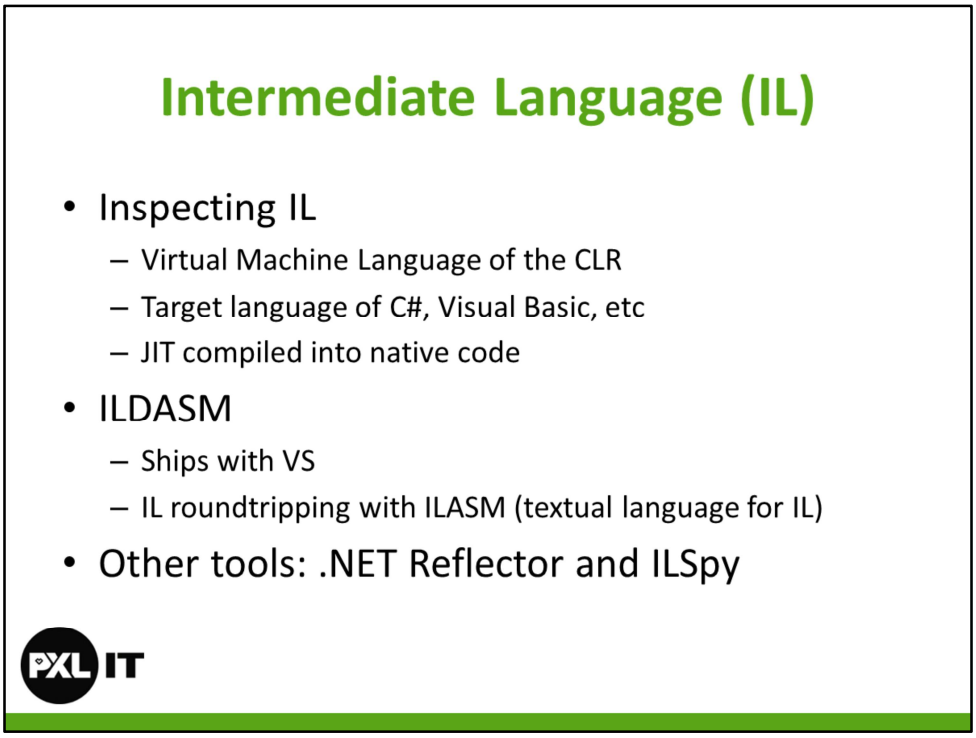


The CLR defines a common programming model and a standard type system for cross-platform, multi-language development."

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

The runtime's Just-in-Time (JIT) compiler convert 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.

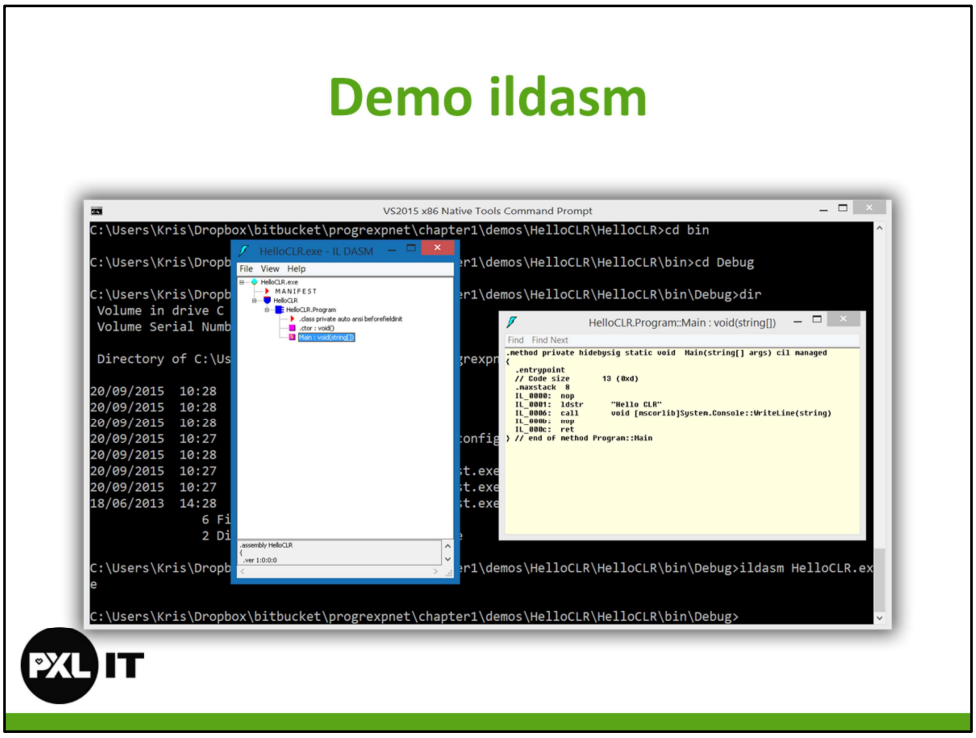


It's interesting to have a basic understanding of the mechanics of IL. There is a basic tool called ILDASM that ships with VS and generates a textual representation of IL. You can actually modify this code and generate IL back (= roundtripping) with ILASM.

Other tools to explore: .NET reflector and ILSpy

https://www.red-gate.com/products/dotnet-development/reflector/ (niet gratis)

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

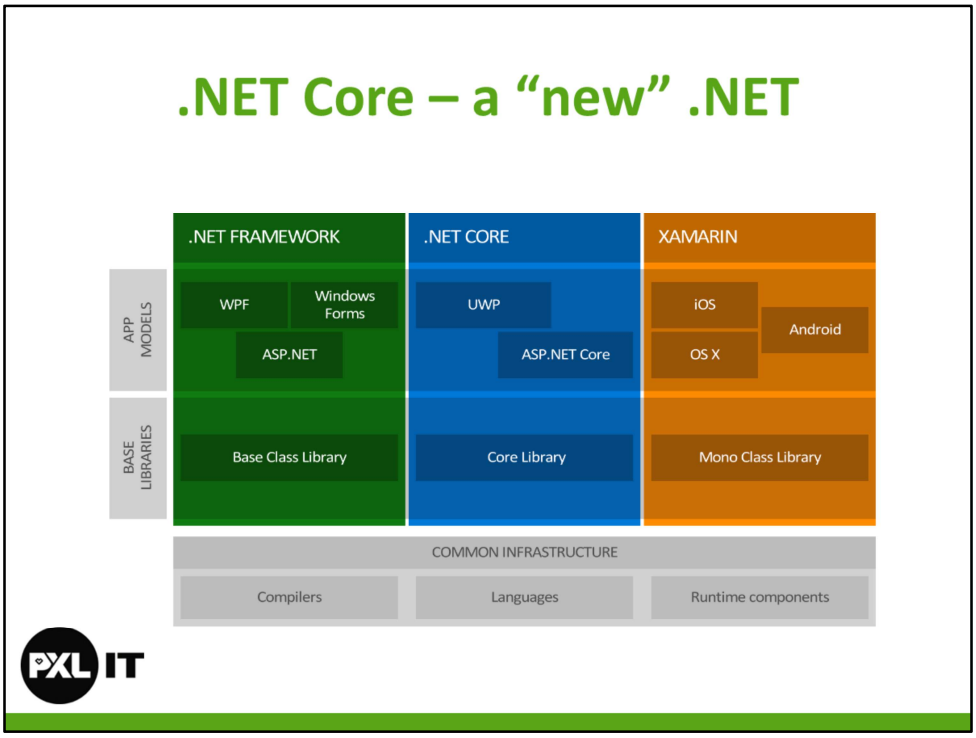


Search for the VS2015 x86 Native Tools Command Prompt, it should be installed together with Visual Studio. This command prompt has several expert tools in its PATH variable, like ildasm.exe

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

ildasm.exe HelloCLR.exe

Exercise: tryout IlSpy and CLR Profiler 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).

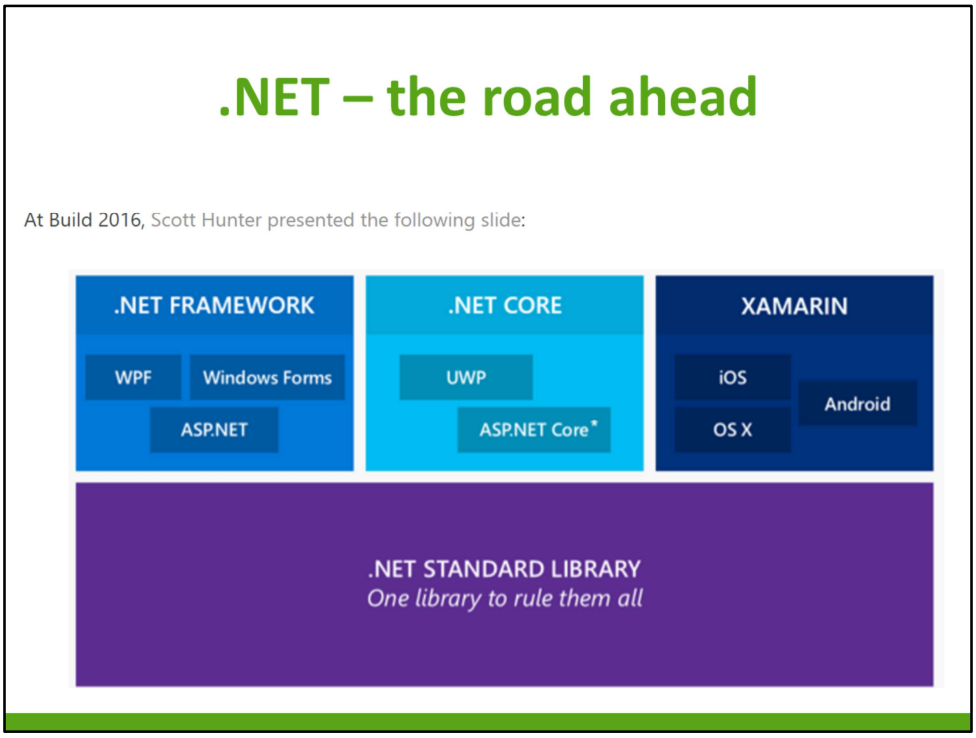
Now recently “.NET Core” is released, yet again a new beginning:

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

References

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

https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0/



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

Reference

https://blogs.msdn.microsoft.com/dotnet/2016/05/27/making-it-easier-to-port-to-net-core/

Demo: .NET Core on linux

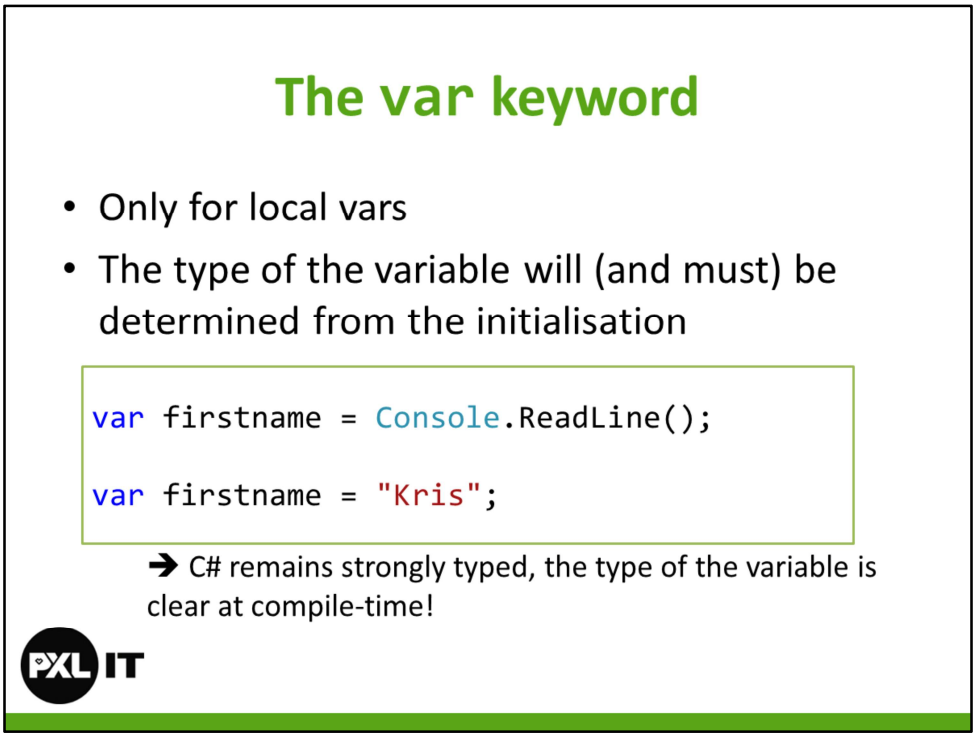
C# is strongly typed

- 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?



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>();

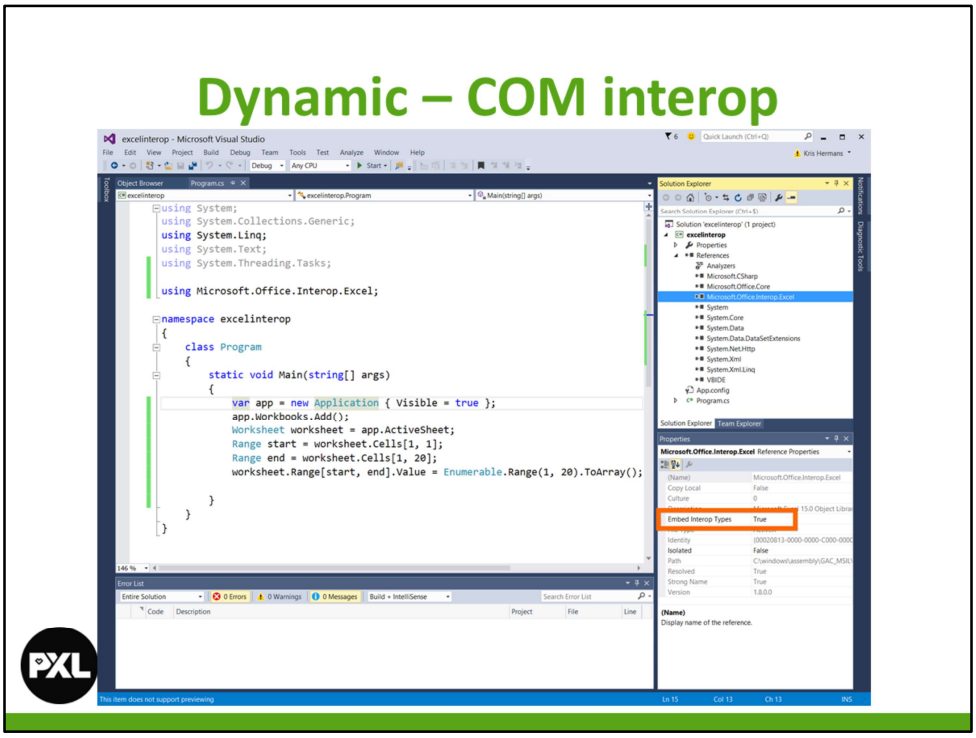
Watch out for readability issues:

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

The dynamic keyword

- Often confused with `var`
- A dynamic variable can have *any* type and can change during runtime
- You loose compile-time checking
- Performance suffering
- Try to avoid it

Demo: dynamics



One useful application of dynamic is COM interop.

Demo: excelinterop (try to set “Embed Interop Types” to false → what happens?)

You have to add the assembly called Microsoft.Office.Interop.Excel and mark property “Embed Interop Types” to true.

This way, everything in the API that would be declared “object” becomes dynamic.

Advantage: a lot less casting

Disadvantage: you loose intellisense (compile-time checking)

Initialize objects

```
Balloon[] CreateBalloonArray()
{
    var balloons = new Balloon[3];

    balloons[0] = new Balloon();
    balloons[0].X = 10;
    balloons[0].Y = 10;
    balloons[0].Diameter = 40;
    balloons[0].Color = "Red";
    balloons[1] = new Balloon();
    balloons[1].X = 14;
    balloons[1].Y = 50;
    balloons[1].Diameter = 400;
    balloons[1].Color = "Green";
    balloons[2] = new Balloon();
    balloons[2].X = 30;
    balloons[2].Y = 80;
    balloons[2].Diameter = 4000;
    balloons[2].Color = "Blue";

    return balloons;
}
```

Classic, “brute force”

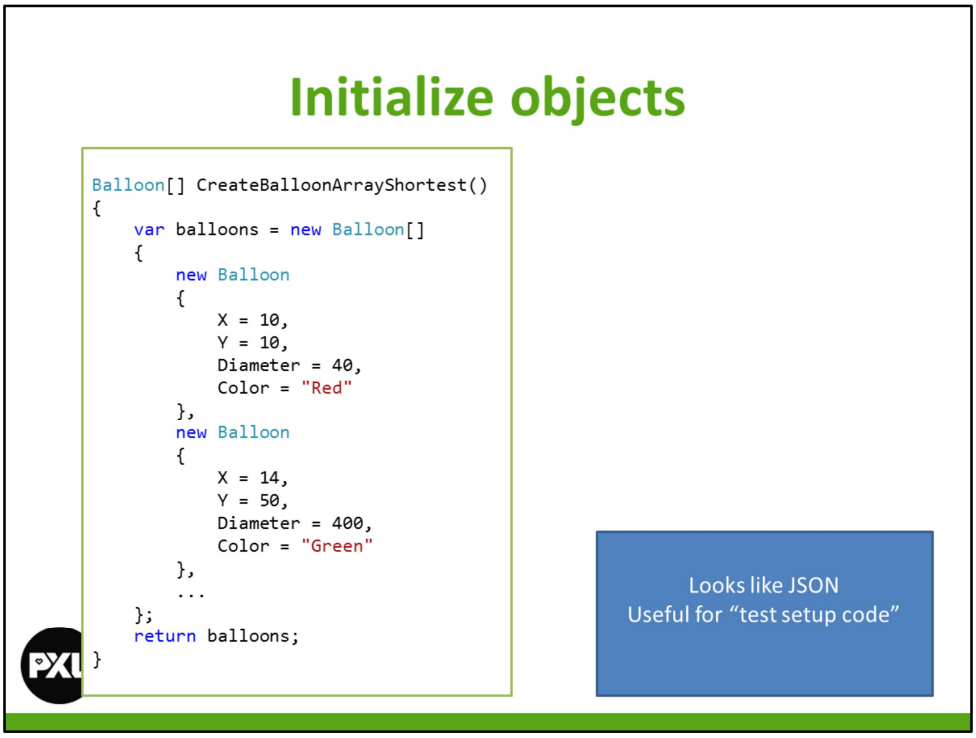
Demo: objectinitialize

Initialize objects

```
Balloon[] CreateBalloonArrayShorter()
{
    var balloons = new Balloon[3];
    balloons[0] = new Balloon() { X = 10, Y = 10, Diameter = 40, Color = "Red" };
    balloons[1] = new Balloon { X = 14, Y = 50, Diameter = 400, Color = "Green" };
    balloons[2] = new Balloon
    {
        X = 30,
        Y = 80,
        Diameter = 4000,
        Color = "Blue"
    };

    return balloons;
}
```

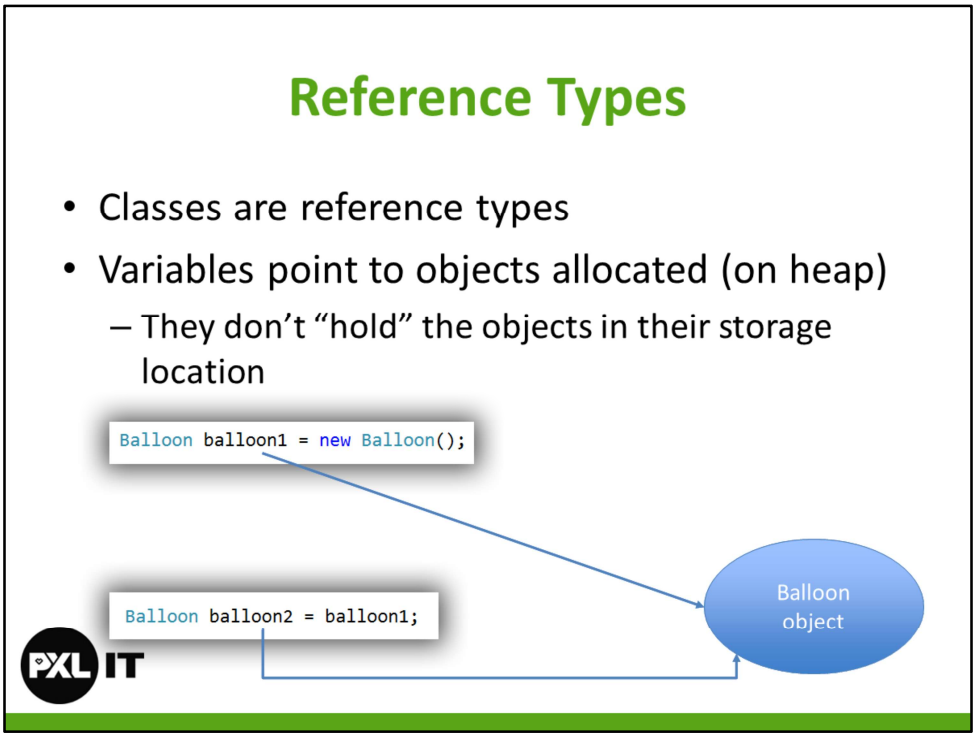
Using *Object Initializer* syntax

Demo: objectinitialize



Demo: objectinitialize

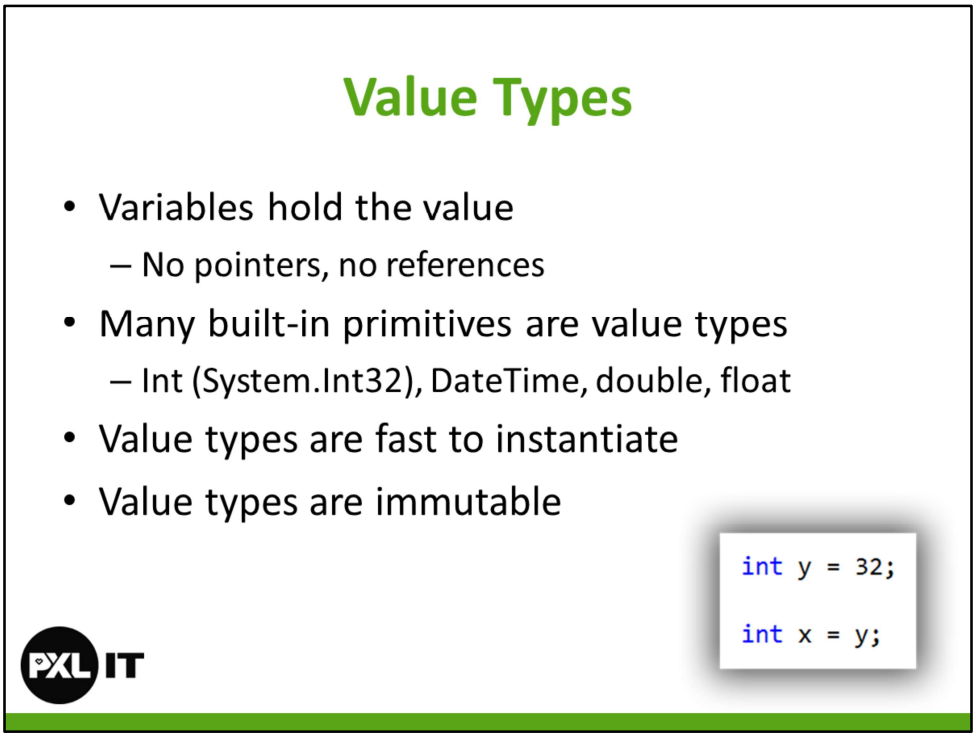
More info: https://msdn.microsoft.com/en-us/library/bb397680.aspx



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.



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.

You cannot change these values, eg: you can't change the value 32, you can assign a NEW value 33 to y.

Stack and Heap

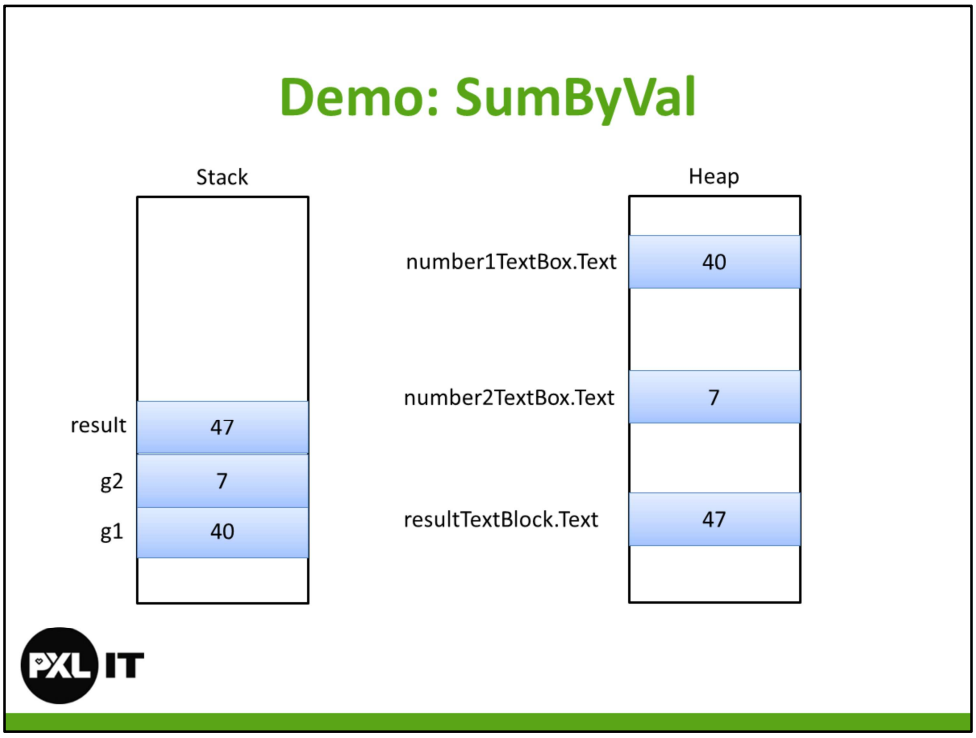
- Heap
  - Dynamically allocate objects
  - Reference types and Value types
  - Garbage collection
  - Several zones (further module)
- Stack
  - Controls method execution
  - Parameter binding
    - Reference types: copy ref to stack
    - Value types: copy value to stack

Demo: SumByVal

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

    int sum = SumByValue(number1, number2);

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

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

Demo: SumByVal

➔ Parameter values are copied on to the Stack



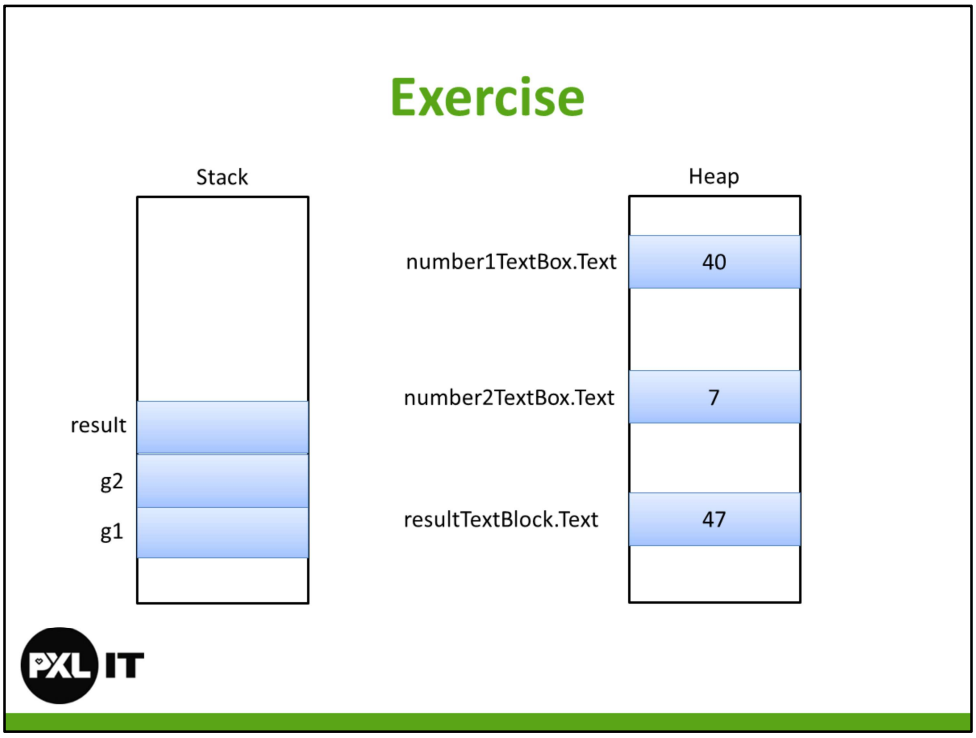
number1TextBox and number2TextBox live on the heap, because they are reference types (class TextBox).

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

Demo: SumByRef

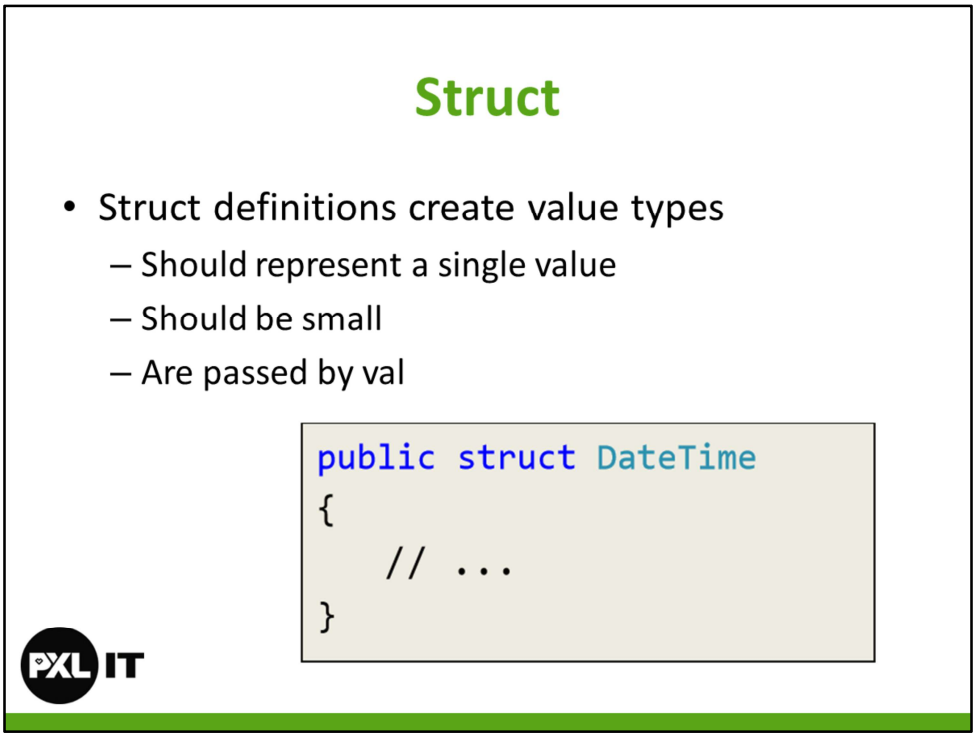
```
private void calcButton_Click(object sender, RoutedEventArgs e)
{
    int number1 = Convert.ToInt32(number1TextBox.Text);
    int number2 = Convert.ToInt32(number2TextBox.Text);

    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



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://msdn.microsoft.com/en-us/library/saxz13w4.aspx

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)

Some interesting tools and refs

- ILDASM / ILASM
- ILSpy
- PerfMon, PerfView
- www.writinghighperf.net

http://www.microsoft.com/en-us/download/details.aspx?id=16273