

C# and .NET Framework

Architecture

Language

Object-techniques

Framework basics

Getting started

- Open Visual Studio
- Clone Repository
 - Location: <https://github.com/tutorit/csh230607.git>
 - Path: c:\course\samples
- The cloned directory holds Materials-folder with these slides as pdf
 - C:\course\samples\Materials
- Keep this Visual Studio running and occasionally pull the changes made by the instructor to see the samples
 - Do not change code, it will result in merge conflicts
- As you start with exercises, open a new instance of Visual Studio

Course contents

Architecture overview

- .Net-platform
- Versions
- Features

C#-Language

- Type system
- Language syntax
- Exceptions
- Basics of OOP

OOP-Reminder

- Why OOP
- OOP Techniques
- SOLID-principles

OOP-Techniques

- Interfaces
- Abstract classes
- Generics

AOP

- Reflection
- Attributes

Functional programming

- Delegates
- Lambda expressions
- Events

Framework specific techniques

- Collections
- LINQ
- I/O-classes
- Serialization
- Multithreaded programming

Other topics to consider

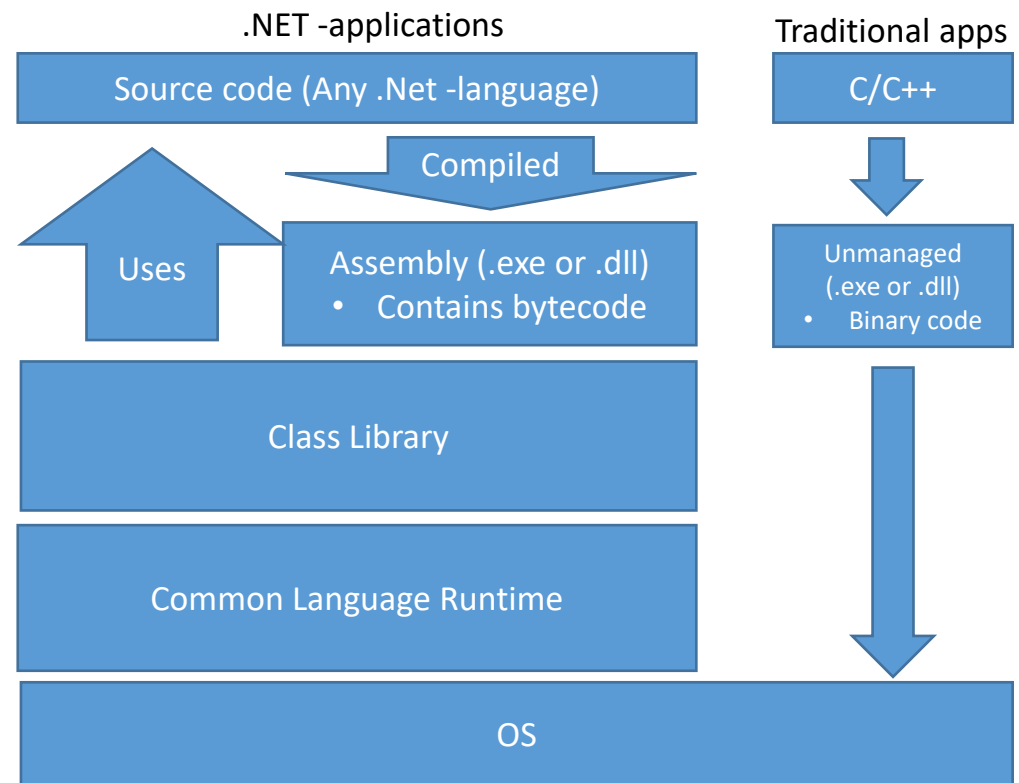
- UI-techniques
- ASP.NET web-apps and – services
- WCF
- Windows Service, event log

.NET Framework

An overview of the platform

.NET Framework

- Abstraction layer on top of operating system
- Common language runtime
 - Type system
 - Memory management
 - JIT-compilation
 - Security
 - Etc
- Huge class-library
 - I/O, Collections, Threads, XML, Databases, Web Apps, Web Services etc



Framework versions

- Version 1.0 (2002), 1.1 (2003), 2.0 (2005)
 - Visual Studio 2005
- Version 3.0 (2007), 3.5 (2007/2008)
 - Visual Studio 2008
 - Included in Windows 7 and Windows Server 2008 R2
- Versio 4 (2010-2017)
 - Visual Studio 2010-2017
- Version 4.8. (2019)
 - Visual Studio 2019
- Version 5.0 (2020)
 - Visual Studio 2019 (r 16.8)
- Version 6.0 (2021)
 - Visual Studio 2022

.Net Core
1.0 (2016)
...
3.1 (2019)

Some features of 3.x

- WPF (Windows Presentation Foundation)
- WCF (Windows Communication Foundation)
- WF (Workflow Foundation)
- LinQ (Language integrated query)

Some features of 4.x

- TPL (Task Parallel Library)
- Dynamic languages

Some features of 5.x

- Core-branding removed

.NET Core -what is (was) it

- Some technologies evolved so that they are no longer backward compatible with previous versions
 - They are not even more but at least to some extent just a subset
 - Target was to have a true cross-platform environment
 - Thus a “restart” in versioning
- So basically we are had
 - .NET Core, new cross-platform version of .NET Framework
 - ASP.NET Core
 - ASP.NET Core includes ASP.NET MVC and Web API, so no longer separate versioning
 - Entity Framework Core
- .NET 5.0 Brought all these back to one
 - But to add confusion for ASP.NET the term “ASP.NET Core Web App” is still used

What is UWP

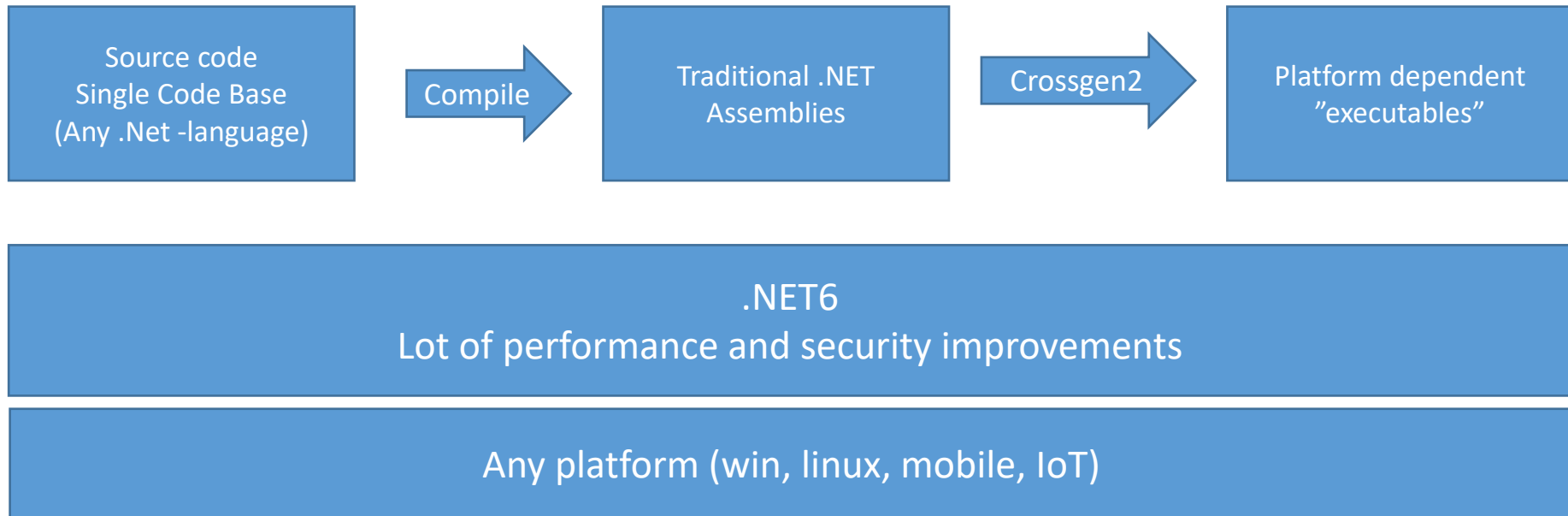
- Universal Windows Platform
 - Next stage of evolution for WinRT (Windows Runtime) introduced for Windows 8.1
 - Targets Windows 10.x on various devices
- Single uniform set of APIs
 - Device independent WinRT API
 - Builds on `Net.Core.UniversalWindowsPlatform`
 - Device dependent Win32 and .NET APIs
 - Possible native compilation
- Single application package than can run on multiple devices
 - Unified tools
 - Unified codebase
 - Unified distribution

So, .NET 6

- Unifies in platform independent manner (Win, Linux, IoT)
 - SDK
 - Base library
 - Runtime
- What's “hot” (according to Microsoft)
 - System.text.Json – APIs, no need for Newtonsoft.Json
 - Support for HTTP/3
- Crossgen => Crossgen2
 - Tool that improves startup time by AOT compilation
 - Possibility to create platform dependent single file publish packages

.NET6 Continued

Platform independency finally completed with single toolset



Back to basics, Hello world with C#

- Every file begins with using-statements
 - Which namespaces from the Framework do we need
 - At least the System-namespace that declares all the basic types
- You may (and should) organize your own code to namespaces
 - Say three to ten in actual projects (UI, BL, DA is a good starting point, perhaps also Utils, Entities etc)
- You always need a class that holds the Main-method
 - Main may or may not return int-value
 - Main may or may not take parameters

This slide applies up to
.NET 5

Hello world

```
using System;

namespace MyFirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world");
        }
    }
}
```

Static is a must for all functions
until we start to talk about objects.

Hello world with C#, .NET 6

- We have implicit using statements
- We have global using statements
- We have top level statements

Hello world

```
Console.WriteLine("Hello, World!");
```

So the code is somewhat simpler?

Exercise, Hello world (Visual Studio 2019)

- Create the “Hello world” -application
 - If you start Visual Studio for the first time it asks for your preferred environment
 - Select C# and you’ll have C# project types available the easiest
 - File -> New -> Project
 - Visual C# Templates -> Windows-category -> Console Application
 - Solution “Course”, Project “Syntax”
 - Extend the main to display “Hello world” when run
 - Use Ctrl-F5 to run, or Menu -> Debug -> Start without debugging
- Study (briefly) the files created
 - App.config and AssemblyInfo.cs
- Right-click the project name -> Open folder in File Explorer
 - Study the directory structure of the project
- Also visit Menu -> Project -> [project name] properties

Exercise, Hello world (Visual Studio 2022)

- Create the “Hello world” -application
 - File -> New -> Project
 - Console Application (Make sure the language is C#)
 - Solution “Course”, Project “Syntax”
 - Study the file, follow the link to <https://aka.ms/new-console-template>
 - Use Ctrl-F5 to run, or Menu -> Debug -> Start without debugging
- Right-click the project name -> Open folder in File Explorer
 - Study the directory structure of the project
 - Study (briefly) Syntax.csproj
- Also visit Menu -> Project -> [project name] properties

What is an assembly?

- Short answer: .net exe or dll
 - Contains executable code in "portable executable" (PE) format
 - Not a binary executable as previously
 - May also contain "static resources"
 - Bitmaps etc
 - And also holds meta-data about dependencies and internals of the assembly
- Private assembly
 - Often the executable and the dll:s required by it are placed into same directory. Those dll:s are considered to be "private" to the application that uses them
- Public assembly
 - Administrator can place dll:s into \Windows\Assembly –directory (Global Assembly Cache, GAC)
 - Assembly must be digitally signed
 - Assembly is versioned, several versions of same Assembly may exist side by side, user decides which one to use
 - Assembly is globally available to all applications in the system
- Satellite assembly
 - Specialized (dll) that only serves static resources to the application
 - Often used for extensive localization, one assembly for each language supported

Demo

- Developer command prompt
- Ildasm, ilasm


C#-Language

Straightforward to start with...
...but some peculiar features

We are on the version 7-10 of the language
Depending on Visual Studio Release

Variables

- Variables may be declared anywhere within a function
 - Or as class members (covered later)
- Type name is followed by self-given variable name and possibly (recommended) initialization
- Two kinds of types
 - Value types
 - Object types
- Also value types (basic types) inherit object
 - Methods common to all objects are available to all variables
 - Value types have some type specific functionality implemented (Parse)



We'll talk about these later
on in more detail

Basic types

C# Type	.NET Framework Type	Note
<code>bool</code>	<code>System.Boolean</code>	true or false
<code>byte</code>	<code>System.Byte</code>	8 bits
<code>sbyte</code>	<code>System.SByte</code>	Signed byte
<code>char</code>	<code>System.Char</code>	16bit unicode
<code>decimal</code>	<code>System.Decimal</code>	128bit, smaller range but better precision
<code>double</code>	<code>System.Double</code>	Double precision floating point
<code>float</code>	<code>System.Single</code>	Single precision floating point
<code>int</code>	<code>System.Int32</code>	32bit integer
<code>UInt</code>	<code>System.UInt32</code>	Unsigned 32bit integer
<code>long</code>	<code>System.Int64</code>	64bit integer
<code>ulong</code>	<code>System.UInt64</code>	Unsigned 64bit integer
<code>object</code>	<code>System.Object</code>	All classes inherit Object-class
<code>short</code>	<code>System.Int16</code>	16bit integer
<code>ushort</code>	<code>System.UInt16</code>	Unsigned 16bit integer
<code>string</code>	<code>System.String</code>	Class holding a character string

Types are actually defined in System-namespace

- Each language has "aliases" defined for types
- Recommendation is to use language-specific types
- Basic types apart from string are declared as structs, making them value types

In addition to types named here all class-, interface and delegate names are also type names

Some common operators

See also: [Operators at msdn](#)

Operator	Meaning
a=b	Assignment
a==b, a!=b	Comparison equals, not equals
a<b, >, <=, >=	Less than, greater than, less than or equal, greater than or equal
a!=b	Not, a and b must be boolean, when b is true a becomes false...
a=b+1	Typical arithmetic operation (+,-,*,/), % is modulus: a=b%2 String concatenation is also done with +operator
a+=b	Increment a by b, also -=, *=, /=, %=
a++	Increment a by one, also a--
a && b	Logical and, a and b must be boolean
a b	Logical or, a and b must be boolean

Few samples with strings and dates

- String-class is rather extensive
 - Study [msdn about strings](#)
- Dates are represented as DateTime-structs
 - Study [msdn about DateTime](#)

```
string tx = "Today is: ";
DateTime dt = DateTime.Now; // Current timestamp
tx += dt.ToLongDateString(); // Concatenate date to string
Console.WriteLine(tx);
int yearIndex = tx.IndexOf(dt.Year.ToString());
tx = tx.Substring(0, yearIndex);
Console.WriteLine(tx); // Date without year-part
dt = DateTime.Parse("4.5.2016");
Console.WriteLine(dt); // May fourth, 2016, depending on locale. . .

string name = "John";
int age = 12;
string descr1 = string.Format("{0} is {1} years old", name, age);
string descr2 = $"{name} is {age} years old";
```

String
interpolation

Variable declaration

- typename variable_name [=assigned value];
- Always assign value even though it is optional
- For numeric constants int and double types are default
 - For other types a conversion or constant type must be specified

Variable declarations

```
int i1 = 32;      // Declare variable and assign value
Int32 i2 = 32;    // int is the same as System.Int32
i1 = 0x20;        // Hex-values may be used
long l1 = 32L;    // L denotes a long constant
bool b = true;    // or false
string s = "Hello world"; // Double quotes
char c = 'H';     // Single quotes
double d = 3.14;
float f = 3.14f;  // f denotes a float constant
```

Type casts and conversions

- All types provide ToString-method returning value as string
- Most types provide Parse(string) -method that converts the string into type from which it was called
- On other cases type-conversion may be attempted
 - Or use as-operator with object types

```
int i = int.Parse("987");  
float f = float.Parse("123,456"); // Local decimal separator, comma in Finland  
string sv = f.ToString();  
i = (int)f; // Decimals are lost  
byte bt = (byte)i; // Precision is lost  
object o = (object)i; // So called boxing  
i = (int)o; // So called unboxing  
  
object o = new Person();  
Person p1 = (Person)o;  
Person p2 = o as Person;  
Console.WriteLine("Is person?", o is Person);
```

Throws exception if
fails

Returns null if fails

Conditionals

- If (condition) { . . . } [else { . . . }]
- Conditional assignment (condition) ? valueIfTrue : valueIfFalse
- Switch - case also exists
 - Compare one variable to several alternatives
 - The variable doesn't need to be numeric as in some other programming languages

Using conditionals

```
int i1 = 32, i2=(new Random()).Next(100);
string s = "";
if (i1 > i2)
{
    s = "It is bigger";
}
else
{
    s = "It is smaller";
}
// Can also be written
s=(i1>i2) ? "It is bigger" : "It is smaller";
```


Loops

- for (initialization;condition;endIteration) {. . .}
- while (condition) {. . .}
 - do {. . .} while(condition)
- foreach(type item in collection) {. . .}
 - Iterates an array or collection

Two almost similar loops

```
for(int counter = 0; counter < 10; counter++)
{
    Console.WriteLine("For iteration " + counter);
}
int i = 0;
while (i < 10)
{
    i++;
    if (i == (new Random()).Next(10)) continue;
    Console.WriteLine("While iteration " + i);
    if (i == (new Random()).Next(10)) break;
}
```

Continue "ignores" rest of the statements on this iteration

Break leaves the loop

Exercise

- Create a number guessing game
- In the Main generate a “secret” random number 1-100
 - `int secret=new Random().Next(100)+1;`
- And initialize a variable for users guess
 - `int guess=0;`
- Stay in a loop until the user guesses the correct number
 - `while(guess!=secret)`
 - `string guess=Console.ReadLine(); // To read input`
 - Convert the guess to int
 - Show hint whether the guess was too big or too small
- Extra
 - After leaving the loop show how many guesses it took
 - Change the loop into `while(true)`, you’ll need to break out of the loop
 - Try using `continue` if the guess is not valid (zero or less, bigger than 100)
 - Can you figure out how `string.Format` –works
 - Also use string interpolation when presenting data

Const and enum

- Const-keyword is used to define a constant value
- Enum is set of named integer constants
 - By default int, other integer sizes may be used too
 - Enum defines a type name, type cast to int can be done

```
enum Days { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
enum Days2:byte { Mon = 2, Tue, Wed = 6, Thu, Fri = 12, Sat = 1, Sun = 0 };

const int WeekLength = 7;
const Days WeekStart = Days.Sun;

static void Enums()
{
    Days d = Days.Wed;
    int i = (int)d;
    string s = d.ToString();
    Console.WriteLine("d=" + d + ", i=" + i + ", s="+s); // d=Wed, i=2, s=Wed
}
```

Var and Dynamic

- Var keyword lets the compiler decide the type for the variable
 - Still strongly typed, type cannot be changed later on
- Dynamic-keyword declares dynamic typing
 - The type may change later on

```
var s = "World";  
Console.WriteLine(s.GetType());  
// s = 3;
```

```
dynamic a = "Hello";  
Console.WriteLine(a.GetType());  
a = 4;  
Console.WriteLine(a.GetType());
```

"s" becomes a string and the type will remain. Assignment of int-value is not accepted by the compiler

"a" uses dynamic typing. The actual type is checked at runtime and the type may change on each assignment

Arrays

- Arrays are collections of items
 - having a preset size
- Length-property can be used to query the number of items
- Individual item is referred to with [] -operator
 - Zero-based index

With C# version eight you also get ranges `ar[3..6]` and indexes from end `^5`

Array samples

```
int[] ia = { 1, 2, 3 };
string[] sa = new string[] { "Hello", "world" };
DateTime[] da = new DateTime[4];

for(int i = 0; i < ia.Length; i++)
{
    Console.WriteLine(ia[i]);
}
foreach(string s in sa)
{
    Console.WriteLine(s);
}
```

Basics of collections, working with List

- Arrays are static in size, you decide the size of the array upon initialization
- Collection-classes are dynamic
 - You may freely add and delete items
 - With list you still have indexed access
 - And you may iterate the list with foreach
 - Or query the size from Count-property

Collections of different types will be covered in more detail later on

This list holds strings

Initialize with these items

```
List<string> weekdays = new List<string> { "Monday", "Tuesday", "Wednesday" };
weekdays.Add("Thursday");
weekdays.Add("Friday");
weekdays.Add("Saturday");
weekdays.Insert(0, "Sunday");
Console.WriteLine(weekdays[1]); // Monday
foreach (string s in weekdays)
    Console.WriteLine(s);
Console.WriteLine("Count " + weekdays.Count); // 7
```

Add couple new items

Insert item into index 0

Demo/Exercise

- Experiment with arrays and Lists
- Create a string holding comma-separated weekdays
 - "Mon,Tue,Wed,Thu,Fri,Sat,Sun"
- Use `string.split` to create an array of weekdays
- Move data from array into a list

Methods (functions)

And parameters

Methods

- We don't write all code to the Main-method
 - Instead, we break the application into small pieces of execution (methods) that are called
- Methods may take parameters and return a value
- Method declaration takes form

[static] [visibility] return_type MethodName(parameter_list) {. . .}

Couple of examples

```
static int CalcSum(int a, int b)
{
    return a + b;
}

static int PromptForInteger(string prompt)
{
    Console.Write(prompt + ": ");
    string s = Console.ReadLine();
    return int.Parse(s);
}

// From Main etc:
int sum = CalcSum(3, 5);
int val = PromptForInteger("Give a number");
```

Just take the static-keyword for granted for now...

Word about parameter passing

- Basically everything is always passed by value, method works with a copy of that value, caller doesn't see the changes
 - When working with objects the object-reference is passed by value
 - When working with structs the entire struct is passed by value
- By using the ref-keyword we change that
 - Object reference is passed in as a reference => we are able to change the value of the reference => to which object we are referencing
 - If a reference to a struct is passed then we basically handle the contents of the struct passed in (as we normally do with objects)

Ref and out

- When ref-parameter is passed the method may change the parameter value
- When out-parameter is passed the method must change the parameter value
- Also the caller must specify the parameter type

Ref and out parameters

```
static void Init(out int a, out int b)
{
    a = 1;
    b = 2;
}

static int Calc(int a, int b, ref int prod)
{
    prod = a * b;
    return a + b;
}

static void Main(string[] args)
{
    int a,b,p,s = 0;
    Init(out a, out b);
    s = Calc(a, b, ref p);

    int.TryParse("123", out int value);
    Console.WriteLine("value="+value);
}
```

Parameters a and b **will be** given a value

Parameters a and b **may be** given new values

Default values and named parameters

- Parameters may have default values
 - From right to left
- When calling named parameters may be used
 - Especially useful with default values, you may skip some of the first parameters

```
static void ShowPrice(double price=0, string currency="EUR")
{
    Console.WriteLine(price + currency);
}

static void Main()
{
    ShowPrice(200, "USD"); //200USD
    ShowPrice(120); // 120EUR
    ShowPrice(); // 0EUR
    ShowPrice(currency: "USD"); // 0USD
    ShowPrice(currency: "USD", price: 400); // 400USD
}
```

Exercise

Extend the number guessing game

- Create method that initializes the “secret”-value
 - First create a InitRandom-method that returns the generated random value
 - Also experiment with an other version of InitRandom that takes either ref or out parameter instead of returning a value. Which one is more convenient for this purpose?
- Create a PromptForInt -method that displays the string passed as parameter and returns Console.ReadLine converted into int
 - Experiment with default value for parameter: if parameter is not passed just display “Enter an integer”
- Declare a const int MaxValue at class-level
 - Use MaxValue when generating the the random value
 - Use MaxValue for prompt-string: “Enter int between 1-MaxValue”

Some extra features about methods

- Variable length parameter list may be declared
 - Params-keyword
- Inner (local) functions are supported

```
static int SomeRecursion(string s,params char[] values)
{
    int OneIteration(string current,int value=0)
    {
        if (current.Length == 0) return value;
        if (values.Contains(current[0])) value++;
        return OneIteration(current.Substring(1), value);
    }

    return OneIteration(s);
}

int val = SomeRecursion("Hello world", 'l', 'e', 'd');
Console.WriteLine("What do I tell " + val);
```

Pointers, unsafe-blocks

- In C# you can use pointers in similar manner to C++
 - Pointers can however only be used in codeblocks marked unsafe
 - Feature must be enabled in with compiler options
- The only reason to use is if you need to use dll implemented with C/C++

```
static unsafe void GotThePointer(int *ip) // ip is a pointer to int
{
    (*ip) = 32; // Place the value to the specified address
}

static void Main()
{
    unsafe
    {
        int i = 0;
        GotThePointer(&i); // Pass in the address of i
        Console.WriteLine(i); // 32
    }
}
```

Classes and objects

Encapsulation

Inheritance and polymorphism

Basics of classes

Class declares an object type holding data members and methods

```
class Vector
```

```
{  
    public int i, j;
```

Data-members, fields

```
    public Vector(int i,int j)  
    {  
        this.i = i;  
        this.j = j;  
    }
```

Constructor is used to
initialize object

```
    public void Add(Vector v)  
    {  
        this.i += v.i;  
        this.j += v.j;  
    }
```

Add-method

```
}
```

```
static void TestVector(Vector v)  
{  
    v.Add(new Vector(3, 4));  
}
```

V-parameter is a reference to a vector-object. When working with the object we modify the original object through the reference.

```
static void Main(string[] args)  
{
```

The values held are: i=1, j=2

```
    Vector vect = new Vector(1, 2);  
    TestVector(vect);  
    Console.WriteLine(vect.i + "," + vect.j);  
}
```

Object reference is passed

What are the values of i and j?

...and structs

Struct declares a value type holding data members and methods

```
struct Vector
```

```
{  
    public int i, j;
```

Data-members, fields

```
    public Vector(int i, int j)
```

Constructor

```
{  
        this.i = i;  
        this.j = j;  
    }
```

```
    public void Add(Vector v)
```

Add-method

```
{  
        this.i += v.i;  
        this.j += v.j;  
    }
```

```
}
```

```
static void TestVector(Vector v)
```

```
{  
    v.Add(new Vector(3, 4));  
}
```

```
static void Main(string[] args)
```

```
{  
    Vector vect = new Vector(1, 2);  
    TestVector(vect);  
    Console.WriteLine(vect.i + ", " + vect.j);  
}
```

All parameters are passed "by value", now the struct itself. Basically we work with a copy of the original vector.

The values held are: i=1, j=2

Entire struct is passed

What are the values of i and j?

...and records

Record declares a reference type holding data members and methods

The diagram illustrates the C# record syntax for a `Vector` type. It shows two ways to declare a record: a full class-like declaration and a simplified function-based declaration. Annotations point to specific parts of the code:

- Data-members, often properties:** Points to the `public int i`, `public int j`, and `public double Dist` declarations in the first record.
- Constructor:** Points to the `public Vector(int i, int j)` constructor in the first record.
- Methods:** Points to the `public Vector Add(Vector a)` method in the first record.
- Function parameters will become properties:** Points to the `int i, int j` parameters in the simplified declaration `record Vector(int i, int j)`.
- Everything else within function body:** Points to the `public double Dist` and `public Vector Add` declarations within the simplified record's body.

```
record Vector
{
    public int i { get; set; }
    public int j { get; set; }
    public double Dist => Math.Sqrt(i * i + j * j);

    public Vector(int i, int j)
    {
        this.i = i;
        this.j = j;
    }

    public Vector Add(Vector a)
    {
        return new Vector(this.i + a.i, this.j + a.j);
    }
}

// But we can also use simplified declaration
record Vector(int i, int j)
{
    public double Dist => Math.Sqrt(i * i + j * j);
    public Vector Add(Vector a)
    {
        return new Vector(this.i + a.i, this.j + a.j);
    }
}
```

- Unlike classes, records have content based equality
- Record can only inherit another record

Null

- Variable that refers to an object may be set to null
 - "I don't refer to any object"
- Null reference cannot be used in any way
 - If you try to refer to a member of an object with null reference the application crashes
- If there is a possibility the reference may be null always test it!
 - On some occasions the questionmark-operator can be used

```
static int TakeRight(string s, int len, out string ret)
{
    ret = "";
    if (s == null) return 0;
    ret = s.Substring(len);
    return ret.Length;
}
```

Traditional, test for null with if-statement

```
static int? TakeRightV2(string s, int len, out string ret)
{
    ret = s?.Substring(len);
    return ret?.Length;
}
```

?-operator returns null if left hand operand is null and doesn't evaluate expression further

```
string s = null;
var len=s?.Length;
string s2 = s ?? "" ;
```

s2=s==null ? "" : s;

```
len=TakeRight(s, 4, out string ret);
Console.WriteLine($"{ret},{len}");
```

```
len = TakeRightV2(s, 4, out ret);
Console.WriteLine($"{ret},{len}");
```

What is the type for len?

Nullables

- Value types don't allow null value
 - `int i=null; // not allowed`
 - `DateTime dt=null; // not allowed`
- Sometimes it would be convenient to indicate "I don't have a real value to place here"
 - Especially with method parameters
 - Can be used for return values and variable declarations also

```
static int? Square(int? s)
{
    if (!s.HasValue) return null;
    return s.Value * s.Value;
}

static void Main()
{
    int? sq = Square(5);
    Console.WriteLine(sq.Value);
    sq = Square(null);
    Console.WriteLine(sq.Value);
}
```

Also:
`Nullable.GetValueOrDefault()`

With C# 7 and 8 you get
some new ways of using ?-
operator: ?? And ??=

Nested types

- Types may be declared within classes and structs
 - Consider the class (struct) to be a namespace
- Inner types may access private members of containing types
- Inner types are private by default, can only be used from containing type
 - But can be made public

```
new Game().Throw();  
new Game.Dice(new Game()).Roll();
```

```
class Game  
{  
    private Dice[] dice;  
    private Random random = new Random();  
  
    public Game()  
    {  
        dice = new Dice[]{ new Dice(this), new Dice(this) };  
    }  
  
    public void Throw()  
    {  
        foreach (Dice d in dice) d.Roll();  
    }  
  
    public void GotResult(int i)  
    {  
        Console.WriteLine("Got result " + i);  
    }  
  
    public class Dice // Could also be Die but....  
    {  
        Game parent;  
        public Dice(Game parent)  
        {  
            this.parent = parent;  
        }  
  
        public void Roll()  
        {  
            parent.GotResult(parent.random.Next(6) + 1);  
        }  
    }  
}
```

Basics of encapsulation, properties

- The object should always be responsible of its own state
 - Validate the changes to the data members
- In most cases this leads to solution where data is "hidden" with private-keyword
 - Can only be accessed through the public methods provided by the class
- In many cases use of properties simplifies this procedure a lot
 - Property has a backing field (traditional data-member) declared as private
 - Each property has accessors: get and set "methods" where implementation may include validation logic
 - On occasions you might want to implement just the "get"-part making a readonly property
 - Read-only property may also be calculated

```
private int _j;  
public int j  
{  
    get  
    {  
        return _j;  
    }  
    set  
    {  
        if (value < 100) _j = value;  
    }  
}  
  
public int i { get; private set; }
```

Backing field

Property

Getter often just returns the backing field

Consider setter to be a method that takes "value" as parameter and places it to the backing field, after validation

Getter is public, data can be queried from the object. Setter is private. Value can only be set from other methods of this class

Property with autoimplemented accessors

Working with encapsulating vector

- Only public members may be accessed through the object reference
 - Private fields cannot be accessed directly
 - Properties with private setters cannot be accessed

```
Vector v = new Vector(1, 2);  
// v.i = 10; // cannot be done setter for i is private  
v.j = 300; // Can be done, but j remains at value 2  
v.j = 10; // OK, value of j is changed
```


Exercise

- Add a simple Person class having properties name, birthday, and email
 - Name (string) cannot be empty or null
 - Email (string) can be empty but not null
 - Birthday (DateOnly) may be null ("not known")
 - Not in future
 - Implement suitable constructor(s)

Exercise

Implement

- Vector-struct (int i, int j)
 - Create showAndIncrement(Vector v) to program.cs
 - Passing null?
 - Incrementing?
- Customer-record (string Name, double Purchases)
 - Test the comparison
 - Create a new instance using with-statement
 - Create makePurchase(Customer cust, double amount)

What is going on here?

```
Vector v = new() { i = 2, j = 3 };  
int i, j;  
(i, j) = v;  
(int x, int y) = v;  
  
(x, y) = (5, 2);
```

- Vector has implemented Deconstruct-method

```
public void Deconstruct(out int i, out int j)  
{  
    i = this.i;  
    j = this.j;  
}
```

Basics of inheritance

- Extend the base class
 - Add some data members
 - Add new methods
- Alter the base class (polymorphism)
 - Provide alternate implementation

Base-class has an implementation to a method but allows sub-classes to make their own implementations

In 3d-world we also require k-component for the vector

We cannot access i and j, even though they are our members. Call base class constructor to initialize those

Let the Vector-implementation add i and j. If the parameter is Vector3d-object we add the k

```
class Vector
{
    public int i { get; private set; }
    public int j { get; set; }

    public Vector(int i, int j)
    {
        this.i = i;
        this.j = j;
    }

    virtual public void Add(Vector v)
    {
        this.i += v.i;
        this.j += v.j;
    }
}
```

Inherit everything from the base class

```
class Vector3D : Vector
{
    public int k { get; private set; }

    public Vector3D(int i, int j, int k)
    {
        : base(i, j)
        this.k = k;
    }

    public override void Add(Vector v)
    {
        base.Add(v);
        if (v is Vector3D)
            k += (v as Vector3D).k;
    }
}
```

Parameter may be Vector object or Vector3d-object

Object references in class hierarchy

- We can use an object reference typed to the base class when referring to the object of an inheriting class
 - When method requires an object reference as parameter, object of sub-class can always be passed
- Object-class is the final base class for all classes, we can refer to anything with a variable typed to "object"
- Actual object type is decided upon instantiation
 - But the reference type determines which members we can use
 - If we know what we are doing we can typecast the reference up and down in the class hierarchy

```
static void TestVector(Vector v)
{
    v.Add(new Vector(3, 4));
}

static void Main(string[] args)
{
    Vector3D v1 = new Vector3D(1, 2, 3);
    Vector v2 = new Vector3D(1, 2, 3);
    object v3 = new Vector3D(1, 2, 3);
    // Vector3D v4 = new Vector(1, 2); // Not allowed

    Vector3D v5 = v3 as Vector3D;

    TestVector(v1);
}
```

Vector3D can also be passed in

If it is Vector3D, Vector3D.Add is actually called

Object-class

- Object-class is the final base class for all other classes
- Three methods that may be implemented to own class with override-keyword
 - ToString - return string-representation of the object
 - Equals - compare contents of the object to an other object
 - GetHashCode - return int-value identifying object in reasonable precision, should always be implemented if Equals is implemented
 - If Hashcode is different for two objects the contents cannot be the same
 - If HashCodes are equal it is possible that object contents are equal
- And a method for querying type information about the object
 - GetType

Demo/Exercise

- Implement ToString to the Person-class
- Inherit Customer from Person, add double Purchases – property
 - Constructor that may take purchases as parameter
 - ToString-method
- In the Program.cs
 - Create TestIt –method that takes an object as parameter and just Console.WriteLine it
 - Pass Person and Customer objects to TestIt

Exercise

- Construct a List that holds Person-objects and Lists of
 - Person-objects and Lists of
 - Person-objects and Lists of
 - Person-objects and Lists of
- So it is a tree-like construct....
- Implement ShowList-method that just prints the hierarchy

Static members

- Without the static-modifier the members belong to an object of the class
 - You instantiate the class to get object, then the fields get their values
 - In the property accessors and methods you refer to the fields of a specific object
 - You refer to the member through an object reference
- A member declared with static modifier is not object specific
 - All objects see the same value for a static field
 - If static field is public it is globally available
 - Static methods may not access object specific members (this-keyword is not available)
 - Even static properties are available
 - (public) Static members are accessed with class name, not object reference

???

using static System.Math

```
class Car
{
    private static string[] makes= {"Volvo", "Saab"};

    static public string GetMake(int index)
    {
        return makes[index];
    }
}

string m = Car.GetMake(1);
```

Exercise

- Add int Id-property to the Person-class
 - Auto implemented accessors, private setter
- Figure out a mechanism to automatically generate a unique Id for each Person object

Extension methods

- You can add new methods to existing classes
- Very convenient if you end up repeating same few steps with objects of Framework-classes
- Requires static class
 - methods with parameter using this-keyword in peculiar location
- Surprisingly extension methods can be added also to sealed classes

Extension methods

```
static public class MyExtensions
{
    static public string Left(this string s,int n)
    {
        return s.Substring(0, n);
    }

    static public void WriteXML(this object o,TextWriter writer)
    {
        XmlSerializer xmlser = new XmlSerializer(o.GetType());
        xmlser.Serialize(writer, o);
    }
}
```

Anonymous classes

- Occasionally you need a very temporary -helper object just holding a few members
 - No logic associated with the class
 - Would make no sense to declare a class-type
 - Often used just locally within one method
 - But can be passed as parameter to a function taking any object as parameter

Constructing an anonymous class

```
var altPerson = new { n = "Tom", a = 6 };
Type tp = altPerson.GetType();
Console.WriteLine(tp.Name);           /
Console.WriteLine(tp.IsPublic);      // false/ <>f__AnonymousType0`2
Console.WriteLine(altPerson);        // { n = Tom, a = 6 }
```

Tuples

Tuple is a very dynamically created collection of few items of (different) types

System.ValueTuple NuGet-package is required with older Visual Studio installations

```
static (int Sum, int Product) calculate(int a, int b)
{
    return (a + b, a * b);
}

static void Tuples()
{
    var john = ("John", 18);
    Console.WriteLine(john.Item1 + ", " + john.Item2);

    var tom = (name: "Tom", age: 21);
    Console.WriteLine(tom.name + ", " + tom.age);

    (string name, int age, string email) tim = ("Tim", 19, "tim@tutors.net");
    Console.WriteLine(tim.name + ", " + tim.age + ", " + tim.email);

    var result = calculate(2, 4);
    Console.WriteLine("Sum: " + result.Sum + ", Product: " + result.Product);

    (int sum, int prod) = calculate(3, 5);
    Console.WriteLine("Sum: " + sum + ", Product: " + prod);
}
```

Operator overloading

```
class Cup {  
    public int Amount { get; set; }  
  
    static public Cup operator ++(Cup c) {  
        c.Amount++;  
        return c;  
    }  
  
    static public Cup operator +(Cup a, Cup b) {  
        return new Cup { Amount = a.Amount + b.Amount };  
    }  
  
    static public Cup operator +(Cup a, int b) {  
        return new Cup { Amount = a.Amount + b };  
    }  
  
    static public bool operator true(Cup a) {  
        return a.Amount != 0;  
    }  
  
    static public bool operator false(Cup a) {  
        return a.Amount == 0;  
    }  
  
    static public bool operator !(Cup a) {  
        return a.Amount == 0;  
    }  
}
```

Nice and rather often used feature in C++

- Perhaps not so in C#

Still a feature that is available if needed

- Implemented as static methods to class
- Carefully think what should be returned
- Typically the operands shouldn't be changed

Experiment

- Experiment with features demonstrated on earlier slides
- For example create a list that holds tuples of person-objects and strings (social security number)
 - Loop through the list
- If you implemented Vector-class earlier, you might want to implement + operator there in

Exception handling

- Exceptions are a common way object oriented languages notify about different errors
 - Technical: Division by zero, I/O-errors, security issues, memory-related errors, conversion errors etc
 - Domain: Errors specific to your own application
- If error is found an Exception is “thrown”
 - Exception is a class defined in the framework
 - With huge amount of more specific sub-classes
 - Documentation tells what types of exceptions a method may throw
 - You may specify domain-specific exceptions by inheriting your own class from Exception
- If the exception is not immediately “caught” it bubbles in the call-stack towards the application entry point (Main)
 - If not caught at all the default-handler ends the application

Sample, exception handling

```
string[] args=GetArgs(); // Returns string-array of some items
```

```
try {  
    int fig1 = int.Parse(args[0]);  
    int fig2 = int.Parse(args[2]);  
    switch (args[1])  
    {  
        case "+":  
            Console.WriteLine("{0} + {1} = {2}", fig1, fig2, fig1 + fig2);  
            break;  
        case "*":  
            Console.WriteLine("{0} * {1} = {2}", fig1, fig2, fig1 * fig2);  
            break;  
        default:  
            throw new Exception("Invalid operator");  
    }  
}
```

Try to do something that may cause exceptions

Indicate error

```
catch (FormatException fex) {  
    Console.WriteLine("Numbers needed");  
}  
catch (IndexOutOfRangeException rex) {  
    Console.WriteLine("Three items needed");  
}  
catch (Exception ex) {  
    Console.WriteLine(ex.Message);  
}  
finally {  
    Console.WriteLine("All done calculating");  
}
```

Item cannot be converted into int

Not enough items in the array.

"Invalid operator"

Finally-block is executed always

Sample, own exception

- Basically just give more meaningful names for errors
- Also pass extra information about error as properties
- If you throw the exception as result of catching an other (technical) exception you might want to store the InnerException

```
class CalcException : Exception
{
    public string fig1 { get; private set; }
    public string fig2 { get; private set; }

    public CalcException(string f1, string f2, Exception inner=null)
        : base("Error in calculation", inner)
    {
        fig1 = f1;
        fig2 = f2;
    }
}
```

Exercise

- Implement a Calculate-function that takes a string as parameter
 - Expect string to be in format "1 + 2"
- Use split to create an arrays of substrings
- Perform the calculation based on operands and operator
- Try passing invalid strings
 - "1 +"
 - "a + 3"
- Use exception handling to cover from errors
- Implement own exception that you throw if one of the operands is bigger than 10
 - Or unknown operand is given

OOP-Reminder

Why objects

- Why
 - Improved productivity through re-use
 - Improved maintainability
 - Safely extend and modify implementation
 - More precise responsibilities between pieces of source code
- How
 - Encapsulation
 - Inheritance
 - Polymorphism

How to design objects

- Most importantly who is responsible of what
 - Where to place each function
 - Hide complexities, ease of use
- What data is manipulated by objects
 - Who is responsible for the data validity
 - Encapsulation
- How to modify and extend behaviour
 - "Do not touch"
 - Inheritance and polymorphism

Solid

- **Single Responsibility**
 - Class should only have single responsibility, only one reason to change the implementation
 - Don't process data and display data in the same class
- **Open Closed**
 - Entities should be closed for modifications but open for extensions
 1. Class can/should only be changed to correct errors
 2. Interfaces should describe behaviour, implementation can be changed
- **Liskow Substitution**
 - Types should be replaceable with their subtypes
 - Objects of subclass can always be used when an object of base class is needed
- **Interface Segregation**
 - Many client specific interfaces instead of one big interface
 - Design interfaces from client's (or service consumer's or object user's) point of view
 - What services are needed by the client
 - Use cases
- **Dependency Inversion**
 - Depend on abstractions instead of concretions (Interfaces instead of Class-types)
 - High-level modules should not depend on low-level modules. Both should depend on abstractions.
 - Abstractions should not depend on details. Details should depend on abstractions.

Design patterns

- Design pattern is a formalized and generalized but not obvious solution to a specific, common problem
- Design patterns are not good or bad, they are more or less usefull
- Good reading to more fully understand OO-design
- Some most common design patterns of modern architecture
 - Singleton, Immutable
 - Factory, Factory Method
 - Mediator, Adapter
 - Value Object, Data Access Object
 - Facade, Command
 - Decorator

Patterns and practises

- Microsoft's recommendations on various topics
 - Fundamentals, cloud, desktop, phone, services, web
 - <https://msdn.microsoft.com/en-us/library/ff921345.aspx>
- For the overview the catalog is a good starting point
 - <https://msdn.microsoft.com/en-us/library/ee658089.aspx>
- Lists some familiar design patterns
 - Service Locator
 - Dependency Injection and Inversion of Control
 - Three-tiered application architecture
 - Data Transfer Object (Value Object)
 - Observer
 - Filter
- And also some technical points
 - Data replication
 - ETL-process
 - Clustering
 - etc

Classes

Roughly two types of classes

- For objects that hold data
 - Class or struct
 - Not much difference in declaration
 - Structs are passed by value, objects of classes by reference
 - Most often you want to use classes, more performant
 - Mostly contain publicly available data
 - Still data-validity should be responsibility of the object
 - Setters should contain logic
- For objects that provide services to manipulate data
 - Need for interface segregation?
 - Need for generics, can same logic serve different types

Don't mix responsibilities

Don't implement both aspects to single class

Class declaration

```
[modifiers] class ClassName [: InheritanceList] { ...members... }
```

- Unless otherwise declared the class may only be used in the declaring namespace
 - public - class can be used outside of declaring namespace
 - Internal - class is available within the current assembly (project: exe or dll)
- Modifiers may also include
 - static - class only holds static members
 - sealed - class cannot be inherited
 - abstract - class cannot be used unless inherited
 - partial - class definition is continued in a separate file

```
class Car {}  
  
// The above declares type Car  
// Which can be used as follows:  
Car c = new Car();
```

Class members - Fields

- Fields (member variables) hold the state for objects of the class
 - Variables declared at class “scope”
 - Available only for objects of the class
- May have modifiers
 - private - default and should be used in most cases, member can only be accessed from the methods of the class
 - protected - as private but also methods of inheriting classes may access the field
 - public - should be avoided to enforce encapsulation, field is available through object reference
 - internal - may be followed by protected or public but still only available within the assembly
 - readonly - The value may only be set upon initialization or by constructor

```
class Car
{
    int speed=0;
    private int direction=0;
}
```

Class members - Properties

- Properties are the preferred way of encapsulating the data held by the class
 - Property typically has a “backing field” but may also be calculated
 - May have implementation for accessors: get and set
 - Both may have different access modifiers
- Property is used as it was a field
 - Possible implementation is hidden from the user

Properties sample

```
class Calculation
{
    public int Operand1 { get; set; } = 0;
    private int operand2 = 0;
    public int Operand2
    {
        get
        {
            return operand2;
        }
        set
        {
            if (value >= 0) operand2 = value;
        }
    }
    public int Sum
    {
        get
        {
            return Operand1 + Operand2;
        }
    }
}
```

Auto-implemented property, backing field is hidden. Often the setter should be private

Field used as backing field for property Operand2

Typical full implementation of a property

Calculated readonly property

Class members - Constructors

- Constructor is a special method in the class
 - No return value not even void, name is the same as the name of the class
- Constructor is always called when object is instantiated
 - Constructor should initialize the data for the object
- There may be several versions of the constructor identified by parameter list
 - Constructor parameters may also have default values
 - The version that takes no parameters is called “default constructor”, it is available if no other constructors are defined
- Most often constructors have public-modifier
 - On special cases protected and private can also be used

Constructor samples

```
class Car
{
    int speed;
    private int direction;

    public Car()
    {
        speed = 0;
        direction = 0;
    }

    public Car(int speed, int direction)
    {
        this.speed = speed;
        this.direction = direction;
    }
}
```

Default constructor
initialized data to reasonable
defaults

This refers to the object from whom
the method is called

Parameters are used to
initialize data.
this-keyword refers to
the object whose method
is being executed.

```
// Now car may be instantiated:
Car c1 = new Car();
Car c2 = new Car(80, 30);
```


Class members - Methods

- Class may also contain functionality called from a object
 - The same access modifiers as for fields
 - Basically everything learned about methods apply
 - Default parameters, named parameters, nullables, return values etc
- All the member variables are available in the implementation
 - Again this-keyword may be used to emphasize a reference to an other member of the same object

```
public int Accelerate(int ds)
{
    this.speed += ds;
    return speed;
}
```

This-keyword may be used but is not required unless there is conflict between parameter and member name.

```
public void Print()
{
    Console.WriteLine("Direction " + direction + ", speed " + speed);
}
```

Object creation

In C# our classes don't need extensive amounts of constructors

- Default values, named parameters, object initializers
- Build a constructors
 - Taking the absolutely required parameters
 - Possibly for some very common cases
 - For value objects you most often need to implement the default constructor

Instantiation

```
class Person {  
    public string name { get; set; }  
    public int age { get; set; }  
    public Person(string name="", int age=0) { . . . }  
}
```

```
Person p = new Person();  
p = new Person(age: 3, name: "John");  
p = new Person("Mike") { age = 6 };  
p = new Person { name = "Tim", age = 5 };
```

Exercise

1. Create a console application ("Reporting"-project)
2. We need access to Person-class from earlier project
 - Follow your instructors lead to add Reference to "Basics"-project
3. Add a Reporter-class so that the following works:
 - BeginReport takes Report title as parameter
 - EndReport takes some kind of report-footer as parameter
 - Provide some PrintData-methods with different parameter types (string, DateTime, int), all take string itemTitle as first parameter

```
rep.BeginReport("Person info");  
rep.PrintData("Name", p.Name);  
...  
rep.EndReport("");
```

```
Person info  
Name=John  
Birthday=1.2.1981  
Email=john@test.net
```

Inheritance

And polymorphism

Inheritance

- Inheritance is perhaps the most important OO-technique promoting reuse and productivity
- Existing class is used as a base class
 - When using inheritance everything implemented to the base class is available (though not necessarily accessible)
 - New features may be added
 - Base class is extended, behavior becomes more specialized
 - Existing features may be modified without touching the existing implementation
 - Overriding, polymorphism
- Class only has one base class, the object-class unless otherwise described
 - But the class may also inherit (implement) interfaces
- Liskow Substitution: When referring to an object a variable that is typed to the base class can always be used

Polymorphism

- Polymorphism means that the same functionality produces different effects depending on call context
 - Different implementations of same method for different objects
- The design of polymorphism starts from the base class
 - Mark methods that might become polymorphic with virtual-keyword
- The inheriting class may then mark its implementation with
 - new -keyword, the reference type determines which method is called
 - override-keyword, the objects implementation of the method is used regardless how the object is referenced

Base class sample

- If you want the objects of inheriting classes be able to access members mark them as protected
- If you know that sub-classes may give a new implementation to a method mark it with virtual-keyword

```
class Car
{
    public string Make { get; protected set; }

    public Car(string make)
    {
        this.Make = make;
    }

    virtual public int NumPsgs()
    {
        return 5;
    }
}
```

Virtual : The inheriting class may give a new implementation to the method

Sub-class sample

- All the members of base class exist
 - But are not accessible if declared private in the base class
- Before the constructor body is executed the base class constructor is called
 - The default constructor unless otherwise specified with base-keyword
- Override some virtual members of the base class
 - Or mark with new-keyword if polymorphism is not required

```
class Bus : Car
{
    public int Passangers { get; private set; }
    public Bus(string make, int psgs) : base(make)
    {
        Passangers = psgs;
    }

    public override int NumPsgs()
    {
        return Passangers;
    }
}
```

Override: Here is my implementation for the function implemented in the base class

Exercise

1. Discussion: how to continue the implementation of reporting solution to be able to more easily print Person info?
 - Reporter.report(Person) vs Person.Report()
 - PersonReporter vs PersonReport
2. Let's implement PersonReport
 - Inherits Reporter
 - Ok, so it is PersonReporter, but later on we will see why PersonReport is better name
 - Constructor takes Person object as parameter
 - Single DoReport then prints the report

Interfaces

- Interface just names a set of methods that need to be implemented to a class
 - Methods are always public
- Framework declares numerous interfaces for different purposes
 - You implement the interface and the framework knows that certain methods are available from your objects
- Class may have only one base class but it may inherit several interfaces
 - Comma separated inheritance list
 - All methods declared in the interface must be implemented
- Since C# version 8 also a “default implementation” may appear in the interface
 - However, the implementation is not inherited into the implementing class
- Interface may also contain properties, again the implementation is not inherited into the class

```
interface IMotorized
{
    void Start();
    void Run();
    void Stop();
}
```

Demo

- Declaring and using interfaces

Exercise

- Occasionally we want to create PersonReport to a file
- We need a FileReporter
 - Filename as constructor parameter
 - Instead of screen produces the report to a disk file
 - You can start by making a copy of original Reporter and then
 - BeginReport opens the file
 - PrintData uses it
 - EndReport closes it
- Rename the original Reporter to ScreenReporter
- Create IReporter interface that declares the methods of the original Reporter
- Both FileReporter and ScreenReporter should implement the IReporter
- Now we notice that it was actually a bad idea to inherit PersonReport from Reporter
 - Use containment instead
 - Pass the correct Reporter-object to the PersonReport as constructor parameter

Write to file, remember to add: using System.IO

```
StreamWriter sw = new StreamWriter(@"c:\file.txt");  
sw.WriteLine("Hello world!");  
sw.Flush();  
sw.Close();
```

Abstract class

When implementing the base class...

- We know that a certain method "belongs" to the class
- But at base class we don't know how to implement it

We can...

- Mark the method and the class itself abstract

```
abstract class Cat
{
    abstract public string ThisIsWhatINeed();
    public void ShowNeed()
    {
        Console.WriteLine(ThisIsWhatINeed());
    }
}

class Garfield : Cat {
    public override string ThisIsWhatINeed()
    {
        return "Lasagne";
    }
}
```

Abstract class cannot be instantiated. A sub-class actually implementing the abstract method can. Therefore we can safely call the abstract method

When ShowNeed is called from Garfield-object, Lasagne is displayed....

Exercise

- ScreenReporter and FileReporter contain exact copies of some of the methods
- Create ReporterBase-class and move common functionality there
 - What do you notice?
- ReporterBase needs to be abstract....

Delegates and lambdas

Delegates

- Delegates are references to a method with a specific signature
- Delegate-name is a type-name
 - "Variables of this type refer to a method having this return type and taking these parameters"
 - Type can of course be used for a variable, parameter or return value
- Delegates are used for
 - Callbacks
 - Partial algorithms
 - Events

Delegate sample

```
public delegate int MyDelegate(string a);  
.  
.  
.  
  
static public int MyStrLen(string a) // Matches the delegate declaration  
{  
    return a == null ? -1 : a.Length;  
}  
  
static public int MyCalc(string a) // Also matches the delegate declaration  
{  
    string[] arr = a.Split();  
    return int.Parse(arr[0]) + int.Parse(arr[1]);  
}  
  
static public void PrintIt(string s, MyDelegate d) // Delegate as parameter  
{  
    Console.WriteLine(s + ":" + d(s)); // Call the method passed as parameter  
}  
  
static void Main()  
{  
    PrintIt("3 5", MyStrLen); // 3 5:3  
    PrintIt("3 5", MyCalc); // 3 5:8  
}
```

Anonymous methods

- Sometimes it doesn't make sense to have a separate implementation for method matching the delegate type
 - Short implementation
 - Only used in one location
- Delegate-keyword may be used to declare an anonymous function

Sample of anonymous function

```
PrintIt("Hello", delegate (string a) {  
    return a.Length % 2;  
});
```

Lambda expressions

- Lambda expressions just give a short-hand notation for implementing anonymous methods
 - With these parameters do this
- Can be used anywhere a delegate is needed
- Sample from previous slide can be written as

```
PrintIt("Hello", a => a.Length % 2);
```

More samples

```
// Delegate takes no parameters and returns a string  
( ) => "Hello"  
  
// Lambda expression may have an execution block  
// Here the delegate requires two parameters and returns bool  
(a, b) => {  
    if (a == 3) return true;  
    return b < 5;  
}
```

Predefined delegates

- The framework defines quite a few delegates for different purposes
- We call the framework functionality and provide a method that should be called at some point of execution
 - Indicate completion
 - Provide algorithm for sorting
 - Etc
- Sorting a list, we can provide the method that compares to items
 - Comparison-delegate
- Test if items match a condition, we provide the method that tests the condition and returns true/false
 - Predicate-delegate
- Something needs to be done with an object
 - Action-delegate, Func-delegate
- ETC

Exercise

- Implement a method `PrintPrice(double net, double vat, Calculator c)`
 - First parameter is the net price
 - Second parameter somehow indicates the vat, but it may be in euros or a percentage value
 - Last parameter is a delegate that knows how to calculate the total price from the first two parameters

For the reporting solution

- Declare a delegate `string Formatter(string itemtitle, string itemdata)`
- Figure out a mechanism that allows you to produce the reports in
 - Original text format: `Name=John`
 - (partial) XML format: `<Name>John</Name>`
 - (partial) JSON format: `"Name": "John"`

Asynchronous delegate

- Any method may be executed asynchronously
- Use a delegate to refer to the method and call BeginInvoke from that delegate

```
static double Add(double a, double b)
{
    Thread.Sleep(1000); // Time consuming, difficult operation
    return a + b;
}

delegate double CalcDelegate(double, double);

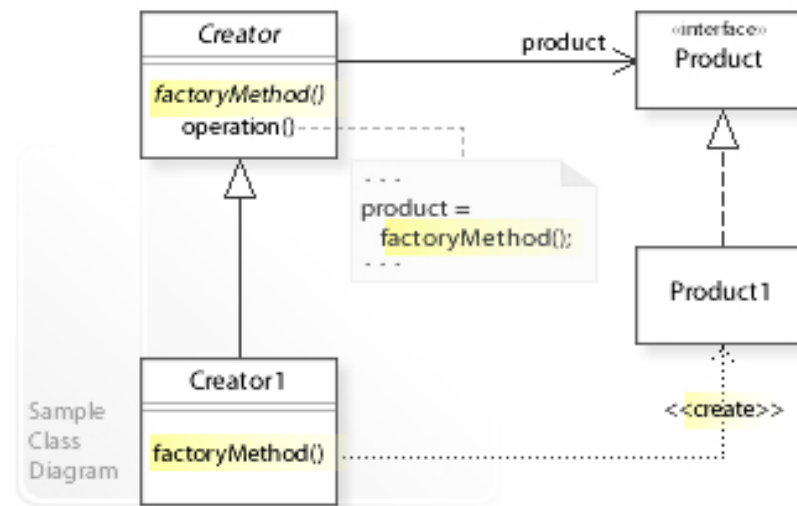
static public void Main(string[] args)
{
    CalcDelegate calc = Add;
    IAsyncResult res = calc.BeginInvoke(5,6,null, null);

    Thread.Sleep(2000); // Do some other stuff
    if (res.IsCompleted)
    {
        double d = calc.EndInvoke(res);
        Console.WriteLine("Result: " + d);
    }
}
```

Parameter may be a callback that is executed when the method completes

Factory method

- One of the original GOF-design patterns
 - Class diagram may be hard to comprehend
- Basic idea is that the instantiation of specific object type requires some logic, possibly several helper objects
 - Hide complexities of the instantiation into single function
- Very often we instantiate an object from a class hierarchy
- The base-class in the hierarchy provides "Create" –method(s)
- Create method from its parameters decides which class to instantiate



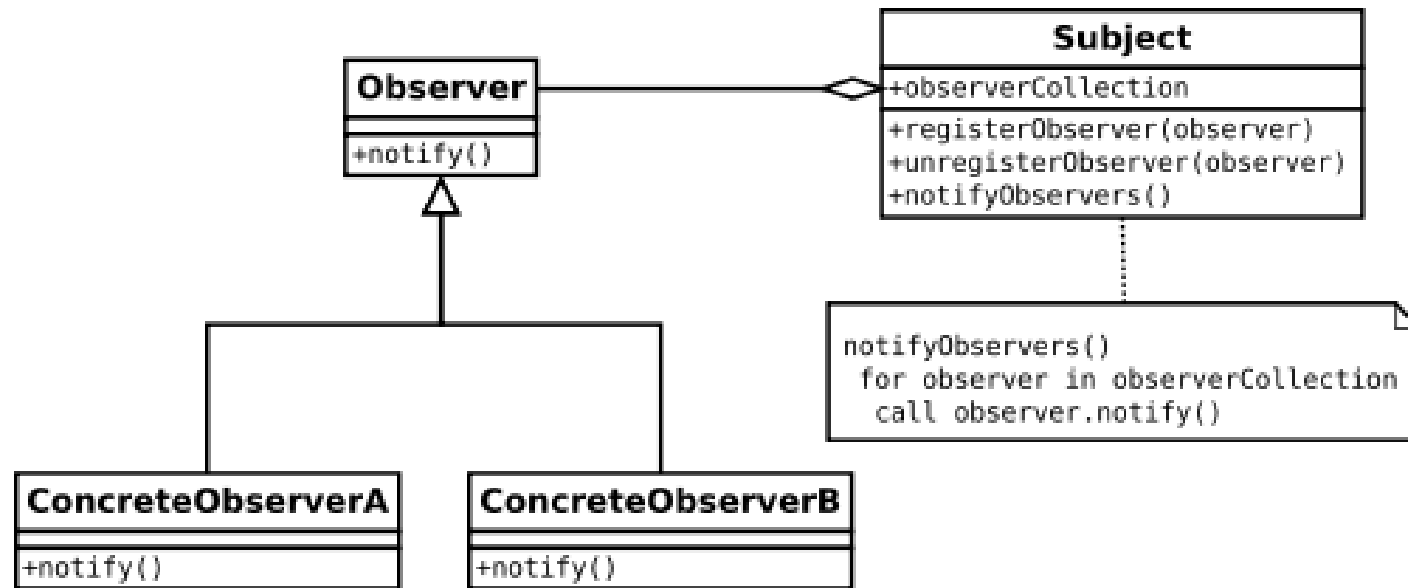
Source: Wikipedia

Exercise

- Add a simple Company-class to your solution (Name and Phone properties)
 - Also Create CompanyReport as we did PersonReport earlier
- Create a abstract Report-class to your solution
 - Abstract DoReport-method
 - PersonReport and CompanyReport should inherit Report
- Add some static Create-methods to the Report-class
 - All take object as parameter (either Person or Company)
 - And instantiate either PersonReport or CompanyReport based on
if (obj is Person)
 - Extra parameters may be used to specify filename for FileReporter and Formatter to be used
 - So you hide the complexity of instantiation into a factory method

Demo/Excercise

- How to implement observer?



Source of image: Wikipedia, the free Encyclopedia

Events

- Events are used to signal that something happened
- Widely used on gui-apps to signal “User did this”
- Events are based of delegates, kind of combining a delegate and collection that holds listeners to that event
- In the the example Car-class holds two events

```
public event CarChangedDelegate SpeedChangedEvent;  
public event EventHandler<CarEventArgs> DirectionChangedEvent;
```

Event-keyword is
used to define
event

Any delegate-
type may be
used

But standard
EventHandler-
delegate is
recommended

The EventHandler-delegate
takes two parameters

- Object that is sending the event
- Object that inherits EventArgs

```
CarEventArgs a = new CarEventArgs { NewSpeed = speed, NewDirection = Direction };  
if (DirectionChangedEvent != null) DirectionChangedEvent(this, a);
```

Exercise

One more feature to the Reporting-solution:

- Declare ReportEventArgs:EventArgs holding properties
 - Report title
 - Reported object
 - Step (0=begin, 1=end)
- ReportBase should hold
 - event EventHandler<ReportEventArgs>
 - And signal the event when the report begins and ends
- Implement a method with which you may process those events and register to listen for them
 - eventObject += handlerMethod

Generic constructs

With attributes

Generics

- Generics allow you to use abstract type names in implementation
 - Algorithm remains the same regardless of the type actually used
- No need to implement specialized versions
- Basically the same as if you would use object instead of the abstract type name
 - But when using the generics you gain type safety
 - The actual type is selected upon usage
- Generic methods
- Generic classes

Demo

- Implementing generics
 - Generic method
 - Generic class

```
static void swap<T>(ref T a, ref T b) where T : struct
{
    T c = a;
    a = b;
    b = c;
}

class Pair<T>
{
    public T First { get; set; }
    public T Second { get; set; }

    public Pair(T a, T b)
    {
        First = a;
        Second = b;
    }

    public override string ToString()
    {
        return "(" + First + "," + Second + ")";
    }
}
```

Exercise

- There could be a generic class in the Report-class hierarchy between the Report-class and actual Report-classes
- What items could it hold?
- Can you make PersonReport (and CompanyReport) appear like follows
 - Creation of new report-types should be as easy as possible

```
class PersonReport : GenReport<Person>
{
    override public void PrintData(Reporter reporter, Person data)
    {
        reporter.PrintData("ID", data.Id);
        reporter.PrintData("Name", data.Name);
        reporter.PrintData("Birthday", data.Birthday);
    }
}
```

Reflection

- Runtime type information
- Type-object operates as starting point
 - `Type t=typeof(Person);`
 - `Type t=person.GetType();`
- After that you may drill down to methods, fields, constructors, base classes, implemented interfaces, attributes etc
 - Classes describing extra information are declared in `System.Reflection`
- You may even
 - Instantiate objects based on type information
 - Invoke methods
 - Set properties

Demo

- Instantiate Person (Activator.CreateInstance)
- set properties
- call method

Attributes

- Attributes describe the use of an item
 - Class, method, property
 - Attached to the entity with square brackets [SomeAttribute]
- Attribute is a class
 - Inheriting System.Attribute
 - Having constructor that can be used to set properties for the attribute
 - Should have a name ending with word Attribute (recommendation)

Declaring and using attribute

```
class TesterAttribute : Attribute
{
    public string Email { get; private set; }
    public TesterAttribute(string email)
    {
        this.Email = email;
    }
}

[TesterAttribute("tim.tester@programmers.ltd")]
class MyActualClass { . . . }
```

Accessing attributes at runtime

- Reflection is needed
- But very generic, type independent behavior can be implemented
 - Extremely loose coupling
 - AOP, Attributes describe behavior

Accessing attributes from any object

```
Type t = obj.GetType();

// We are getting collection, possibly attribute is used on base classes also
var taa = t.GetCustomAttributes<TesterAttribute>() ;
Console.WriteLine(taa.First.Email);

PropertyInfo pi = t.GetProperty("name");
TesterAttribute ta = pi.GetCustomAttribute<TesterAttribute>() ;
if (ta!=null)
{
    // Send email to the tester
}
```

Exercise

- Create a ReportingAttribute-class having a property Title
- Create a very generic DoReport-function that can report any object
 - Go through the properties of an object in a loop
 - If the property doesn't have ReportingAttribute ignore it
 - Display the title of the attribute and value of the property

Caller-info

- Caller-info attributes may help your debugging and tracing hugely
 - Using System.Runtime.CompilerServices
- You may add parameters to your methods that tell who called the method

```
static void CallerInfo(string msg, [CallerFilePath] string fp = "",  
                                [CallerMemberName] string mn = "",  
                                [CallerLineNumber] int ln=0)  
{  
    Console.WriteLine(msg + "Called from " + fp + ", " + mn+", line "+ln);  
}
```

Collections and Linq

Collection classes

System.Collections.Generic contains the classes

- List<>
 - LinkedList<>
 - Queue<>
 - Stack<>
- HashSet<>
 - SortedSet<>
- Dictionary<>
 - SortedDictionary<>

And interfaces for abstraction

- IEnumerable<>
 - IEnumerator<>
- ICollection<>
- IList<>
- IDictionary<>
- ISet<>

All collections implement ICollection and IEnumerable

Collection initializers

- With collection initializers you may create a new collection without “adding” the initial items

Collection initializers

```
List<Person> p1 = new List<Person>
{
    new Person("Tom",3),
    new Person("John",6)
};
Dictionary<string, Person> pd = new Dictionary<string, Person>
{
    ["123"] = new Person("Tom", 3),
    ["456"] = new Person("John", 6)
};
```


Demo

- Basics of set
- Basics of dictionary

IEnumerable

- Objects that implement IEnumerable can supply Enumerator that is used when looping through the object (foreach)
- Generic and non-generic version of the interface exist
- You can also implement a method that "builds" an IEnumerable object with yield return.

Expose object as a map

```
public class Person : IEnumerable<KeyValuePair<string,object>>
{
    public string name { get; set; }
    public int age { get; set; }

    public IEnumerator<KeyValuePair<string,object>> GetEnumerator()
    {
        yield return new KeyValuePair<string, object>("name", name);
        yield return new KeyValuePair<string, object>("age", age);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Ridicilous, generic version if IEnuberable extends non-generic
        yield return new KeyValuePair<string, object>("name", name);
        yield return new KeyValuePair<string, object>("age", age);
    }

    public object this[string key]
    {
        get
        {
            return key == "name" ? (object)name : (object)age;
        }
        set
        {
            if (key == "name") name = value.ToString();
            else age = int.Parse(value);
        }
    }
}
```

Exercise

- Create PersonList-class
 - Inheritance might be more convenient but let's use containment so that we are able to experiment with few things
 - List<Person> persons as field
 - Add-method
 - Print-method for easy testing: prints something about each person
 - Why not add ToString to Person
 - Test
- Add following features to the PersonList
 - Create PrintReverse-method to the PersonList-class (list remains in original order)
 - Add SortByName and SortByBirthday methods (sorts the list)
 - Implement IEnumerable so that PersonList may be looped with foreach
 - Implement Reverse-method so that PersonList may be looped with foreach in reverse order, original sorting of the list must not change
 - Implement indexer through which you may query the with the given name
 - Person p=pl["Tom"]
 - Implement Find(string nameBegin) that returns a collection of persons whose name begins with nameBegin
 - Implement EnumerateByName that return IEnumerable that allows looping in name-order
 - Implement EnumerateByAge(string n) that returns IEnumerable that allows looping in age-order, but only persons whose name contains the parameter are included

LINQ - Language Integrated Query

- LINQ can be used to make SQL-like queries to collections of different type
 - Databases, ADO.NET sources and XML can also be queried
- Simple as basics but the queries may become very complex
 - Expression form and functional form can be used
- Quite a few of the features of SQL-language are supported
 - Sorting Data
 - Set Operations
 - Filtering Data
 - Quantifier Operations
 - Projection Operations
 - Partitioning Data
 - Join Operations
 - Grouping Data
 - Generation Operations
 - Equality Operations
 - Element Operations
 - Converting Data Types
 - Concatenation Operations
 - Aggregation Operations

Basic query

```
List<Person> p1 = . . .  
var r1 = from Person p in p1 where p.age > 3 select p;  
// OR  
var r2 = p1.Where(p => p.age > 3).Select(p => p);
```

- The expression form is often preferred
- from can be read as foreach
 - Loop through the collection “p1”, in each iteration call the item found “p”
- From-part is followed by other “operators”
 - where- gives the condition, which items should be processed
- And select describes how the items included are processed before being “added” to the returned collection
- The return value from the query is always a collection

Sorting, Selecting

```
var r1 = from Person p in pl where p.age > 3 orderby p.age select p;  
var r2 = pl.Where(p => p.age > 3).OrderBy(p => p.name).Select(p => p);
```

- Orderby specifies the sort-order of returned collection
- Select defines what is done to the selected items before they are placed into the returned collection

```
// Here we get a collection of strings
```

```
var r1 = from Person p in pl where p.age > 3 select p.name;  
var r2 = pl.Where(p => p.age > 3).Select(p => p.name);
```

```
// Here we get collection of instances of anonymous class
```

```
// Each object having members a and n
```

```
var r1 = from Person p in pl where p.age > 3 select new {a=p.age,n=p.name };  
var r2 = pl.Where(p => p.age > 3).Select(p => new {a=p.age,n=p.name });
```

Grouping

- We will be creating a collection of collections....
- Top level will be the collection of groups
 - Each item will be collection of data-items belonging to that group

Sample with person-list

```
var r1 = from Person p in pl where p.age>10
        group p by p.age/10 into agegroup select agegroup;
var r2 = pl.Where(p => p.age > 10)
        .GroupBy(p => p.age/10,p=>p).Select(g => g);

foreach(var x in r1) // We actually loop through the groups
{
    Console.WriteLine(x.Key);    // Will be age/10
    foreach(var pi in x)         // All the persons belonging to that group
    {
        Console.WriteLine(pi.name);
    }
}
```


Combining expression and functional forms

- Functional form of LINQ provides us with extension methods that can be applied to collections
 - Many methods take delegates as parameter that operate on single item
- Expression form just provides us with an alternate notation for writing the queries
 - May be more readable at least on simple cases
 - But not all features are available in consistent notation

Calculating average

```
var r3 = from Person p in p1 where p.age > 20 select p;  
double da=r3.Average(p => p.age);  
  
// The above can also be written  
double avg2 = (from Person p in p1 where p.age > 20 select p.age).Average();
```

Demo, Exercise

Experiment with

- LINQ
- Anonymous types
- Implicit typing

PersonList

- Implement "EnumerateByName" and "EnumerateByAge" in both functional and expression forms

I/O and Serialization

Working with files

Binary, XML and JSON

System.io

- Namespace contains quite a few classes to work with
 - See [documentation from msdn](#)
- Especially stydy
 - Directory and File
 - StreamReader and StreamWriter
 - StringReader and StringWriter
 - FileStream

Operating with text files

- Some simple samples

```
// Write to a text file
StreamWriter sw = new StreamWriter(@"c:\file.txt");
sw.WriteLine("Line to write with LF");
sw.Write("Line to write without LF");
sw.Flush(); // Make sure data goes to the device
sw.Close();

// Read a text file
StreamReader sr = new StreamReader(@"c:\file.txt");
string s = sr.ReadToEnd();
Console.WriteLine(s);
sr.Close();

// Alternate read
string s = File.ReadAllText(@"c:\file.txt");
Console.WriteLine(s);
```

IDisposable

- Stream-classes (and other resource-using-classes) implement IDisposable
 - And therefore Dispose-method
 - Allowing to take control when the resource should be freed with using statement (no Close required)

```
using (StreamWriter sw = new StreamWriter(@"c:\file.txt"))  
{  
    sw.WriteLine("Line to write with LF");  
    sw.Write("Line to write without LF");  
}
```

IDisposable, continued

- If your own class keeps the resource open for duration of several method calls consider implementing IDisposable to your own class also
 - Dispose closes the resource
 - Destructor calls Dispose

```
class SomeResourceUsingClass : IDisposable { . . .  
  
~SomeResourceUsingClass()  
{  
    Dispose();  
}  
  
public void Dispose()  
{  
    if (myResource == null) return;  
    myResource.CloseSomeHow();  
    myResource = null;  
    GC.SuppressFinalize(this);  
}  
  
. . . }
```

Exercise, Text files

- Implement a new class "FileHandler" to operate with text files
- Static WriteFile that takes file name as parameter
 - Use StreamWriter to write to the file
 - Loop asking user for a string and write it to the file
 - End the loop when user enters an empty string
- Constructor that takes file name as parameter
 - Opens a StreamReader
- ReadString that returns a string read from the file and indicates with ref parameter if the end of file has been reached
- Implement IDisposable and destructor to make sure the StreamReader is closed at some point

Binary serialization

- Objects may be stored to and read from a stream in a binary form using BinaryFormatter –object
 - formatter.Serialize(stream, objectRef) –saves object
 - objectRef=formatter.Deserialize(stream) – reads object
 - [Sample in msdn](#)
- Serialized object must have [Serializable] attribute

```
[Serializable]
public class Person { . . .

public void SaveBin(string fn)
{
    using (FileStream fs = new FileStream(fn, FileMode.Create, FileAccess.Write))
    {
        IFormatter frm = new BinaryFormatter();
        frm.Serialize(fs, this);
        fs.Flush();
    }
}

. . . }
```

XML Serialization

- Objects data may also be serialized in XML-format
- Serialized class must be public
- It must have a public default constructor
- Only public members are serialized
- XmlAttribute-, XmlElement and XmlTransient –attributes are used to control XML-format in target class

```
public void SaveXML(string fn)
{
    using(StreamWriter sw=new StreamWriter(fn))
    {
        XmlSerializer ser = new XmlSerializer(this.GetType());
        ser.Serialize(sw, this);
        sw.Flush();
    }
}
```

JSON Serialization

- Public class, public default constructor
- DataContract-attribute should precede serialized class
- DataMember-attribute annotates serialized members
- System.Runtime.Serialization-assembly must be referenced

```
public void SaveJSON(string fn)
{
    using(FileStream fs = new FileStream(fn, FileMode.Create, FileAccess.Write))
    {
        var ser = new DataContractJsonSerializer(this.GetType());
        ser.WriteObject(fs, this);
        fs.Flush();
    }
}
```

Exercise, Serialization

- To your Person-class implement
 - SaveBin that takes filename as parameter and serializes Person object to the file
 - Static ReadBin that takes filename as parameter, deserializes person from the file and return the person
- In similar manner implement XML Serialization
 - SaveXML
 - Static ReadXML
- In similar manner implement JSON Serialization
 - SaveJSON
 - Static ReadJSON

Newtonsoft.Json

NuGet-package
Newtonsoft.Json

- Framework features for Json are somewhat clumsy
- Newtonsoft.Json has become favoured and also recommended replacement
- Extremely easy to get started with

```
class Person
{
    public string Name { get; set; } = "John";
    public int Age { get; set; } = 19;
}

string json = JsonConvert.SerializeObject(new Person());
Console.WriteLine(json);

Person copy = JsonConvert.DeserializeObject<Person>(json);
Console.WriteLine(copy.Name + ", " + copy.Age);
```

JsonSerializer (.NET6)

- Originally there was DataContractJsonSerializer
 - Failed attempt
- Everybody used Newtonsoft JSON (NuGet)
- Then came System.Text.Json
 - Of which finally can be said that it provides powerful and versatile json api
 - Lot of effort has been put into this

Try serializing class, record and tuple

```
// Just basic use
Company c = new();
string json = JsonSerializer.Serialize(c);
Console.WriteLine(json);
Company? cx = JsonSerializer.Deserialize(json,
                                         typeof(Company)) as Company;
```

Localization

Translations

Formattings

Resource-files (.resx)

- Resource-files are xml-files containing "resource definitions"
 - Strings and also bitmaps and icons
- You may specify Resource-files for specific locales
 - Translations.resx (the default)
 - Translations.fi.resx (Finnish translations)
 - Translations.en.resx (English translations)
- Resource files are compiled into actual classes
 - And you may refer to the properties of the class directly getting the result depending on your UICulture
 - UI-culture may be set for the CurrentThread

We have:
Translations.resx
Translations.fi.resx
Translations.en.resx

```
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fi");  
Console.WriteLine(Translations.Greeting);
```

All resx-files contain key "Greeting" translated to different languages

Formattings

- We also need to set Thread.CurrentThread.CurrentCulture
- Or define the Culture to the ToString-methods

```
Thread.CurrentThread.CurrentUICulture= CultureInfo.GetCultureInfo("en");
double d = 123345567.653456;
Console.WriteLine("Number "+d.ToString(CultureInfo.GetCultureInfo("en")));
DateTime dt = DateTime.Now;
Console.WriteLine(dt.ToString("d", CultureInfo.GetCultureInfo("en")));

Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("en");
Console.WriteLine("Number " + d);
Console.WriteLine(dt.ToString("d"));
Console.WriteLine($"Number {d}, date {dt}");
```

Experiment

- As demonstrated on previous slides

Multithreaded programming

Basics concepts

Synchronization

Threads and ThreadPool

Tasks, TPL

Async methods

Multithreaded programming

- With threads an application (process) executes several operations simultaneously
 - Several methods in simultaneous execution
 - Threads require some resources (stack)
- Each application has the main thread
 - Execution point that is started from Main
 - Additional threads may be started
- Framework on top of operating system schedules executable threads to give impression of simultaneous execution
 - In any given system there are most likely hundreds of executable threads
- Threads most often "hide" waiting a resource
 - Wait response from server
 - Wait for file operation to complete
 - etc

Asynchronous programming

- Typical programming is synchronous
 - We make calls to methods and wait for the return value
- In asynchronous programming we start the operation (in background thread) without waiting for return value
 - The return value may often be queried later on either through polling for it or with a callback-mechanism
- Many of the multithreaded solutions may be implemented more simply and trustworthy with asynchronous models
 - A short-running algorithm whose return value we require but don't expect to be immediately available

Parallel

- In parallel programming one operation is broken into pieces which are given to separate processors (cores) to execute
 - Overall execution time is reduced since several cores are used to accomplish the final result
- The above given is perhaps the most precise description of parallel programming model in .Net, though in theory parallel execution may also be distributed between devices

Thread

- Thread-class has been available since the beginning of .NET
- A delegate that will start as a thread is given as a constructor parameter
 - The delegate (thead function) is either parameterless or takes a single object as parameter
- Calling Start from the Thread-object starts the given delegate as a thread

```
static void Main()
{
    new Thread(Job).Start();    // Starts Job as thread
    Job();                     // Traditional call
}

static void Job()
{
    for (int i = 0; i < 5; i++)
        Console.Write("Counter "+i);
}
```

Properties

- Static `CurrentThread`: reference to the Thread currently executing
- `IsBackground`: When application ends all background threads are killed automatically (uncontrolled).
 - Foreground threads are automatically waited for
- `Name`: Thread may have a name as an identifier
 - You can also use `GetHashCode()`
- `ThreadState`: only for debugging, never for synchronization
- `IsAlive`: Is the thread function still executing or returned
- `Culture`, `UICulture`: For example in ASP.NET web-applications use `Culture` for localisation.

Priority

- Only change priority after thorough consideration
 - There is a risk that a thread with high priority uses up all processor-time
 - Or thread doesn't get any processing time at all
- Showing the effects of priority changes may be demanding
 - It is infact quite difficult to implement a demonstrative thread that remains executable all the time
- Basic rule
 - You may give a high priority to a thread that most of the time waits for input from some source and when it gets it, it must process the information quickly before going into the next wait

ThreadState (msdn)

Member name	Description
<i>Aborted</i>	<i>The thread state includes AbortRequested and the thread is now dead, but its state has not yet changed to Stopped.</i>
<i>AbortRequested</i>	<i>The Thread.Abort method has been invoked on the thread, but the thread has not yet received the pending System.Threading.ThreadAbortException that will attempt to terminate it.</i>
2. Background	The thread is being executed as a background thread, as opposed to a foreground thread. This state is controlled by setting the Thread.IsBackground property.
2. Running	The thread has been started, it is not blocked, and there is no pending ThreadAbortException .
4. Stopped	The thread has stopped.
<i>StopRequested</i>	The thread is being requested to stop. This is for internal use only.
<i>Suspended</i>	<i>The thread has been suspended.</i>
<i>SuspendRequested</i>	<i>The thread is being requested to suspend.</i>
1. Unstarted	The Thread.Start method has not been invoked on the thread.
3. WaitSleepJoin	The thread is blocked. This could be the result of calling Thread.Sleep or Thread.Join , of requesting a lock — for example, by calling Monitor.Enter or Monitor.Wait — or of waiting on a thread synchronization object such as ManualResetEvent .

Methods

- Static Sleep: Do nothing for given number of milliseconds, other threads are rotated
- Static Yield: Give up the given time-slice, the scheduler may rotate between other threads
- Join: Current thread waits for the thread from which Join is called to end
- Abort: End the thread

Exercise, Basic threads

- Implement a console application that starts a thread from the Main

```
static public void ThreadFunc()  
{  
    for (int i = 0; i < 100; i++)  
    {  
        Console.WriteLine("Value " + i);  
    }  
}
```

- Test
- Change the IsBackground=true, how does this affect the execution
- How can you wait for the thread at the end of the main
- Give the thread a name and display it in the ThreadFunc
- Also display the GetHashCode of the thread
- Start the thread a few times, give each thread a unique name

Thread safety

- Several threads may access common variable or resource
- Thread safety means that all threads should always see valid state for the variable (or resource)
 - Don't read half written information
- The state shouldn't change during a logical operation entity
 - The same as in database transactions
- Variables local to the thread function are always safe
- The parameter given to the thread function may be problematic
- Use of common objects must be designed
- Use of static members must be designed

Collections

- `System.Collections.Concurrent` –namespace declares thread safe collections

Type	Description
<code>BlockingCollection<T></code>	Provides blocking and bounding capabilities for thread-safe collections that implement <code>IProducerConsumerCollection<T></code> . Producer threads block if no slots are available or if the collection is full. Consumer threads block if the collection is empty. This type also supports non-blocking access by consumers and producers. <code>BlockingCollection<T></code> can be used as a base class or backing store to provide blocking and bounding for any collection class that supports <code>IEnumerable<T></code> .
<code>ConcurrentBag<T></code>	A thread-safe bag implementation that provides scalable add and get operations.
<code>ConcurrentDictionary<TKey, TValue></code>	A concurrent and scalable dictionary type.
<code>ConcurrentQueue<T></code>	A concurrent and scalable FIFO queue.
<code>ConcurrentStack<T></code>	A concurrent and scalable LIFO stack.

Volatile

- In multiprocessor environments a value for a variable may be located in the processor cache
 - A thread doesn't see the value changed by a thread executing under different processor
- Volatile –keyword may be used in variable declaration to “force” the changes immediately available for all threads
 - Basic types and references
- If the volatile –keyword is not used for variable declaration you may use VolatileRead and VolatileWrite from Thread-class to ensure use of the latest value for the variable

Interlocked

- Even the simplest operations are not thread safe:

```
a=a+b;
```

- Other threads may change the values for operands in middle of calculation or perhaps just query the value for a-variable that already should have been increased
- Interlocked class provides atomic methods for simple operations

```
Interlocked.Add(ref a,b);
```

- Increment
- Decrement
- Exchange
- CompareExchange

Synchronization

- Through synchronization you affect the scheduler
 - Forbid simultaneous execution of code blocks
 - Wait for something to happen
 - Allow certain number of simultaneous threads
- Carefull design and implementation of synchronization is imperative in multithreaded applications
- The design of synchronization logic may be quite challenging
 - Protect a resource from simultaneous use
 - The protection must be bullet-proof or extremely random problems occur
 - At the same time dead-locks must be avoided

Lock

- Most simple and common method of synchronization
 - Similar to critical section on some environments
- Any object common to several threads may operate as a lock
 - One thread owns the lock, other threads wait for it to be released
 - All threads need to "see" the lock-object, be careful not to use the same lock for different purposes

```
class LockIt
{
    decimal value;
    private Object myLock = new Object();

    public void Decrease(decimal amount)
    {
        lock (myLock)
        {
            // One thread at a time may own myLock
            if (amount < value) value -= amount;
        }
    }
}
```

ReaderWriterLock

- ReaderWriterLock-object provides separate Reader- and Writer- locks
- Any number of ReaderLocks may be given
- When WriterLock is requested all other locks must be released

```
public class Coordinates
{
    private int x, y;
    ReaderWriterLock rwLock = new ReaderWriterLock();

    public void GetCoordinates(out int x, out int y)
    {
        rwLock.AcquireReaderLock(Timeout.Infinite);
        x = this.x;
        y = this.y;
        rwLock.ReleaseLock();
    }

    public void SetCoordinates(int x, int y)
    {
        rwLock.AcquireWriterLock(Timeout.Infinite);
        this.x = x;
        this.y = y;
        rwLock.ReleaseLock();
    }
}
```

Excercise, Thread safety

- Create somewhat strange Point-class
 - X and Y should always hold same values
- In Program-class declare a static member that references a Point-object
- Start few threads that all
 - Set the value for the Point-object to a unique, thread-specific value
 - Sleep 30ms
 - Display the point object and the value that was previosly set. All values should be equal

```
class Point
{
    int x, y;

    public void Set(int v)
    {
        x = v;
        Thread.Sleep(20);
        y = v;
    }

    public override string ToString()
    {
        return $"({x},{y})";
    }
}
```

Point is in valid state if both x and y hold the same value

Synchronization objects

- **Mutex**
 - Mutex is very similar to lock, only one thread may own the mutex
 - Request the lock with `WaitOne`, release with `ReleaseMutex`
 - In most cases lock-object can be used in synchronization within one application
 - Mutex may be named => Interprocess synchronization
 - One process creates the mutex and gives the name
 - Other processes may open the same mutex
- **AutoResetEvent, ManualResetEvent**
 - Event signals that something has happened
 - Wait with `WaitOne`
 - Set and Reset change the state of the Event
 - The state of `AutoResetEvent` is automatically changed to non-signaled when a thread is released from its wait

ThreadPool

- ThreadPool makes use of threads more efficient
 - Memory allocations for threads are done beforehand
 - Application uses the ready-made Thread-objects
- Threads given by ThreadPool are background-threads
- Thread-function still takes one parameter

```
ThreadPool.QueueUserWorkItem(ThreadFunc);
```

- ThreadPool can also be used to start a thread when an event becomes signaled

```
ThreadPool.RegisterWaitForSingleObject(...);
```

Exercise, ThreadPool

- Start the thread from the first exercise from ThreadPool
- Study the information you can query from the ThreadPool
- Use ManualResetEvent to start a thread
 - Create the event object
 - Call RegisterWaitForSingleObject
 - Call Console.ReadLine()
 - Set the event
 - Could you use similar mechanism in your socket application

Task Parallel Library (TPL)

- Simplifies implementation of parallel and multithreaded applications
- More performant applications
- ThreadPool is used in background (yes, we are still implementing threads)
 - More precise control of threading
- The lightweight synchronization objects and thread safe collections came with TPL
- Lazy classes for lazy initialization
- Dataflow-blocks for inter-thread communication

Task-class

- Similar but more advanced as compared to Thread-class
- A method is started in a background thread

```
static void TaskMethod()
{
    Console.WriteLine("Thread " + Thread.CurrentThread.GetHashCode());
    Console.WriteLine("Bkr " + Thread.CurrentThread.IsBackground);
}

static public void Main(string[] args)
{
    Console.WriteLine("Main " + Thread.CurrentThread.GetHashCode());
    // Options to start
    Task t = new Task(TaskMethod);
    t.Start();
    Task.Run(() => TaskMethod());
    Parallel.Invoke(TaskMethod);
    Console.ReadLine();
}
```

Parameters and return values

- A task may have a return value
 - Querying the Result-property blocks until the return value is available
- Lambdas may be used for passing parameter

```
static double TaskMethod(double a, double b)
{
    Thread.Sleep(1000);
    return a + b;
}

static public void Main(string[] args)
{
    Task<double> t = Task.Run(() => TaskMethod(5, 4));
    Console.WriteLine("In a second " + t.Result);
}
```

TaskFactory

- TaskFactory-class provides flexibility and versatile operations for manipulating tasks
- Most recommended when tasks are widely used
- Can be instantiated but more often a factory is used

```
static public void Main(string[] args)
{
    // Previous task-method without parameters
    Task.Factory.StartNew(TaskMethod);
    // Method with parameters and return value
    Task<double> t = Task.Factory.StartNew<double>(() => TaskMethod2(6, 7));
    Console.WriteLine("Result " + t.Result);
}
```

Exercise, Executing tasks

- Start both versions of PI-calculation as tasks
 - Directly with Task
 - With TaskFactory
- Exception handling
 - Cause an exception in calculation (with return value)
 - What exception do you get and where?

Async-methods

- Yet another variation to asynchronous execution
- When implementing a method mark it as async
 - Return value must be void or Task
- Await "spawns" the call to another thread

```
static double Sum(double a, double b)
{
    Thread.Sleep(1000);
    return a + b;
}

async static Task<double> SumAsync(double a, double b)
{
    Task<double> td = Task.Run(() => Sum(a, b));
    await td; // Execution is transferred to caller
    return td.Result; // We are actually returning double
}
```

Calling

- Await waits for a method but also continues the execution of previous non-async method

```
async static void TestIt()
{
    double r = await SumAsync(5, 2);
    Console.WriteLine("Result " + r);
}

static public void Main(string[] args)
{
    TestIt();
    Console.WriteLine("Main is waiting");
    Console.ReadLine();
}
```

Exercise, Async methods

- Implement PI-calculation (with return value) with Async

Parallel looping

- `Parallel.ForEach` loops through each item in a collection and processes each item with given delegate (item as parameter)
- Operations are executed in separate thread, the overall execution time may be shorter
 - Possibly core for each item
 - For small collections the benefit is questionable

```
string[] ar = { "First", "Second" };  
Parallel.ForEach<string>(ar, s => {  
    Console.WriteLine(s + ": " + Thread.CurrentThread.GetHashCode());  
});
```


Exercise, Parallel loopin

- Implement a simple Person-class
 - Public properties Name (string) and Age (int)
 - ToString-method

```
return Name+":" Age+":"+Thread.CurrentThread.GetHashCode();
```

- Create a List<Person>, add few persons to it
- Use Parallel.ForEach to display each person
- How would you calculate the average age with Parallel.ForEach

Parallel LinQ

- Linq may also be run multithreaded
- Call AsParallel from the collection you are querying

```
string[] ar = { "First", "Second" };  
var ret = from string s in ar.AsParallel() where s.Length > 3 select s;  
foreach (string s in ret)  
{  
    Console.WriteLine(s);  
}
```

Exercise, Parallel Linq

- Implement a query selecting Persons whose age is greater than a given
- Change the age property so that when queried it `Console.WriteLine` the `CurrentThread.GetHashCode()`
- Change the LINQ query to parallel

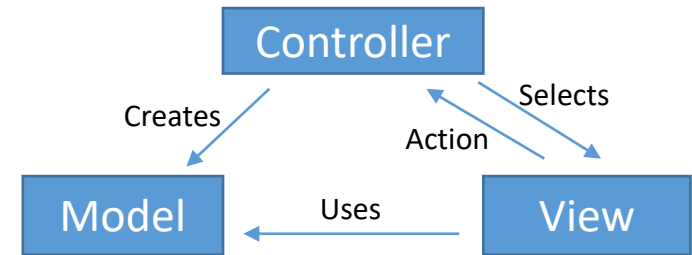
WPF and MVVM

This also works as an introduction to UWA and
Xamarin....

UI-patterns

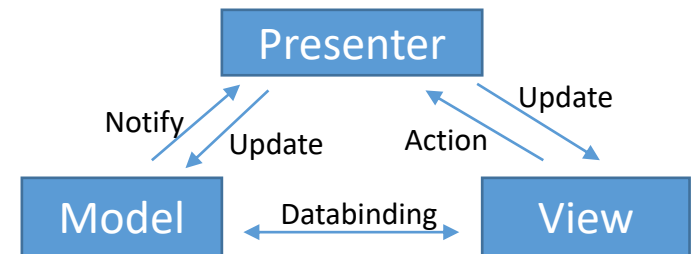
• Model-View-Controller

- Decouples the user interface from the data model
- Most suitable for Web development



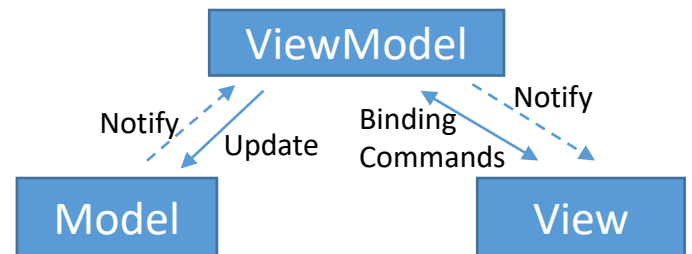
• Model-View-Presenter

- Evolves the MVC pattern for event-driven applications
- Most suitable for forms-over-data development
- Introduces databinding

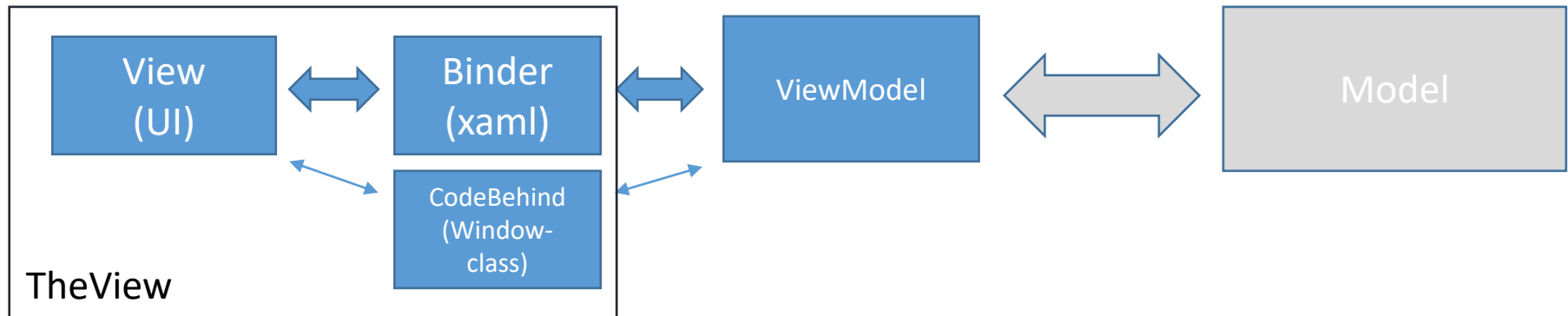


• Model-View-ViewModel

- Evolves from the MVP pattern
- Most suitable for WPF applications
- More loosely coupled

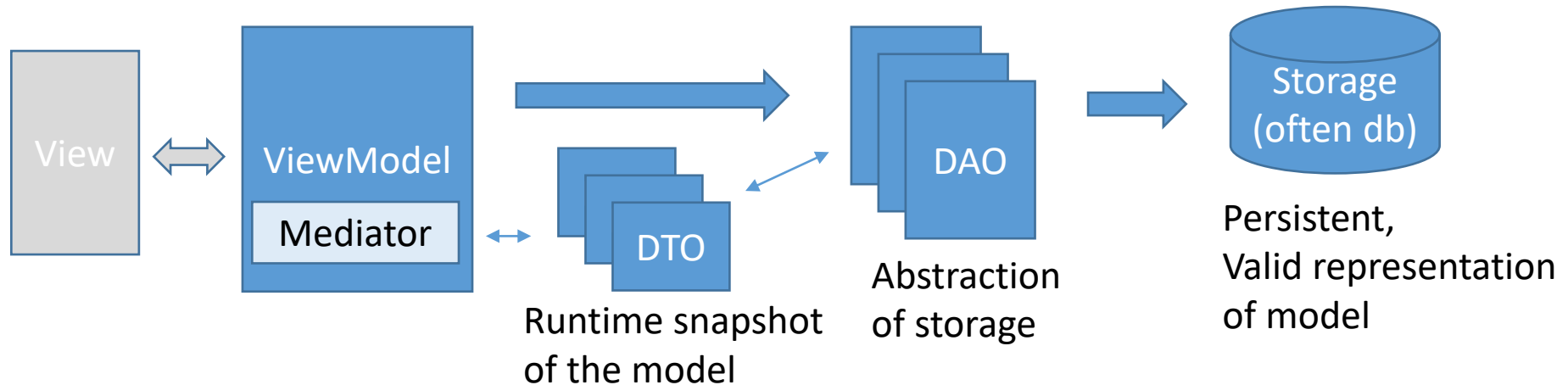


MVVM - View, Binder and ViewModel



- MVVM is a pattern created by Microsoft especially for WPF-applications
 - Now-a-days used on other platforms also
- Like any pattern can actually be implemented in many ways
- The key idea is the abstraction of data- and command-binding to xaml
 - Basis for WPF-programming
 - Deep understanding of xaml and binding mechanisms provided by it are needed
- ViewModel provides the data to the view in format it is easily used in the ui: Mediator, Adapter
 - Objects should/could implement `INotifyPropertyChanged`
 - Collections should/could implement `INotifyCollectionChanged` (or inherit `ObservableCollection`)
- CodeBehind may implement UI-logic associated with the view
 - ViewModel also may contain logic, but it should be view -independent

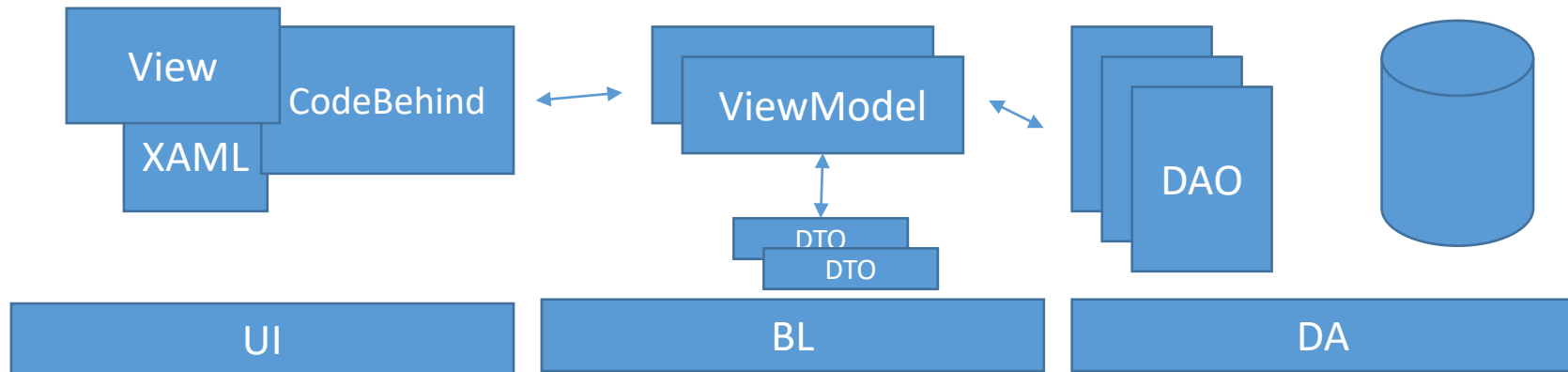
MVVM - ViewModel and Model



THERE IS NOT JUST ONE STRAIGHT-FORWARD, ALWAYS TO USE SOLUTION

- At runtime the model is represented by collection of objects holding the data application is to maintain
- Some abstraction is needed on how the model is generated
 - Factories, Repositories
 - If data is stored into a database or available through web services DAO-pattern is very convenient
- The ViewModel shouldn't create the model-objects but ask them from a "third-party"
 - And most likely we end up with having Data Transfer Objects (Value Objects) holding the data
- The ViewModel requests data from DAO-objects getting multiple DTOs
 - ViewModel then constructs (or operates as) mediator providing services that combine the manipulation of otherwise unrelated DTOs

Three-tiered design



- No, it is not just for distributed architectures
 - Originally it was developed for desktop applications to separate concerns
- All application
 - Present data : UI, the view and the logic associated with it
 - Manipulate data: BL, manipulation is based on rules
 - Store data: DA, Often storage is DB but it may be a disk file or accessed through web services

So patterns, patterns, patterns...

- Patterns offer solutions for specific problems
 - Abstracting technology specific implementation
 - Abstracting complicated logic
 - Providing flexibility to changes
- But
 - If misused they just complicate the solution
 - Keep simple cases simple
 - If you can rewrite something in an hour don't spend four hours trying to force a pattern into it (Golden Hammer antipattern)
- If your architecture requires the use of patterns
 - Then use them consistently

Role of XAML

- The XAML describes the view
 - How should it look like
- The XAML also describes the databinding
 - What data is presented, how it is presented
- The XAML also describes event handling
 - OnClick
- The XAML also describes commanding
 - Cut, Copy, Paste and own commands
- The XAML can also describe some behaviors for the view
 - Animations are a good example

Describing the UI

- The XAML is used to describe the UI
- We need the root-element
 - And quite a few XML-namespaces declared
- And we describe the UI for the root element
- Quite a few of the controls available inherit ContentControl
 - Very complicated content can be designed for those

MainWindow.xaml

```
<Window x:Class="WpfTests.MainWindow" . . .  
    Title="MainWindow" Height="350" Width="525">  
    <Grid>  
        <Calendar HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" />  
        <Button x:Name="button" Margin="25,183,0,0" Width="135">  
            <StackPanel Orientation="Horizontal">  
                <Ellipse Width="20" Height="20" Fill="Red" />  
                <Label Content="Press me" />  
            </StackPanel>  
        </Button>  
    </Grid>  
</Window>
```

Events

- Event handling is quite straight-forward
 - Event is described in XAML, the handler is implemented to Window-class
- Just remember that events traverse the control-hierarchy, routed events
 - Preview[EVENT] -handlers kind of peek if the event is coming, handled from bottom-most control upwards
 - [EVENT]-handler actually processes the event, handled from top-most control downwards

MainWindow.xaml

```
<Button x:Name="button" . . . Click="button_Click" MouseMove="ButtonMouseMove" >
    <StackPanel Orientation="Horizontal">
        <Ellipse Width="20" Height="20" Fill="Red" />
        <Label x:Name="buttonLabel" Content="Press me" MouseMove="LabelMouseMove"
    />
    </StackPanel>
</Button>
<Label x:Name="labelInfo" Content="Label" . . . />
<Label x:Name="buttonInfo" Content="Label" . . . />
```

```
private void LabelMouseMove(object sender, MouseEventArgs e)
{
    labelInfo.Content = "Move " + e.GetPosition(buttonLabel).X;
    e.Handled = true; // Now ButtonMouseMove will not be called
}
```

Databinding

- Databinding is one of the strongest features of WPF
 - Sadly sometimes ignored and misused
- We have
 - Binding target => Attribute of an element is described with binding
 - Binding source => Property of an object providing the data
 - Type conversion, in many cases automatic
 - Validation, can be declared and implemented
 - Two-way, One-way, One-way to the source, One-time

MainWindow.xaml (Different binding modes between UI-elements)

```
<TextBox x:Name="textBox1" Text="{Binding ElementName=slider1,Path=Value}" . . ./>
<Slider x:Name="slider1" Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox2"
        Text="{Binding ElementName=slider2,Path=Value,Mode=OneWay}" . . ./>
<Slider x:Name="slider2" Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox3"
        Text="{Binding ElementName=slider3,Path=Value, Mode=OneWayToSource}" . . ./>
<Slider x:Name="slider3" Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox4"
        Text="{Binding ElementName=slider4,Path=Value,Mode=OneTime}" . . ./>
<Slider x:Name="slider4" Minimum="0" Maximum="20" Value="10" . . ./>
```

Setting DataContext to custom object

- Often you should create a new object-type to contain data for your ui (the ViewModel)
 - Instead of adding huge number of properties to you Window-class
- Then you can set the DataContext-property to point to a new object of your class
 - On window or any other container
- Of course the object can also be instantiated and the DataContext be set in the constructor

```
class MyData
{
    public string data { get; set; }

    public MyData()
    {
        data = "Hello";
    }
}
```

```
<!-- local namespace -->
<Window.DataContext>
    <local:MyData />
</Window.DataContext>
```

OR

```
public MainWindow()
{
    DataContext=new MyData();
    InitializeComponent();
}
```

DataContext and ViewModel

- DataContext is most often set for a container
 - You might want to consider DataContext to be the ViewModel for that container
 - On some cases also to control that is not a container
- Different controls may have different DataContexts
- It is still possible to refer to properties of basically any object with data binding

Different bindings

```
<!-- Current containers DataContext -->
<TextBox Text="{Binding data}" . . ./>

<!-- Any resource available -->
<TextBox Text="{Binding Source={StaticResource otherData},Path=data}" . . ./>

<!-- Windows properties, two notations basically synonyms -->
<TextBox
    Text="{Binding RelativeSource={RelativeSource
        AncestorType={x:Type Window}},Path=data}" . . ./>
<TextBox Text="{Binding RelativeSource={RelativeSource
    Mode=FindAncestor, AncestorType=Window},Path=data}" . . ./>
```

Notifications

- To reflect changes of data to UI the model objects need to provide a PropertyChanged-event

MyData.cs

```
class MyData : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _data = "";
    public string data
    {
        get { return _data; }
        set
        {
            _data = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs("data"));
        }
    }
}
```


What else?

Quite a lot...

- Commanding
- Collections
- Content controls
- Templates
- Resources
- Styles
- Custom controls
- Triggers
- Animations
- Multimedia
- Etc

And basically all these work in similar manner in UWA-applications and even Xamarin-applications