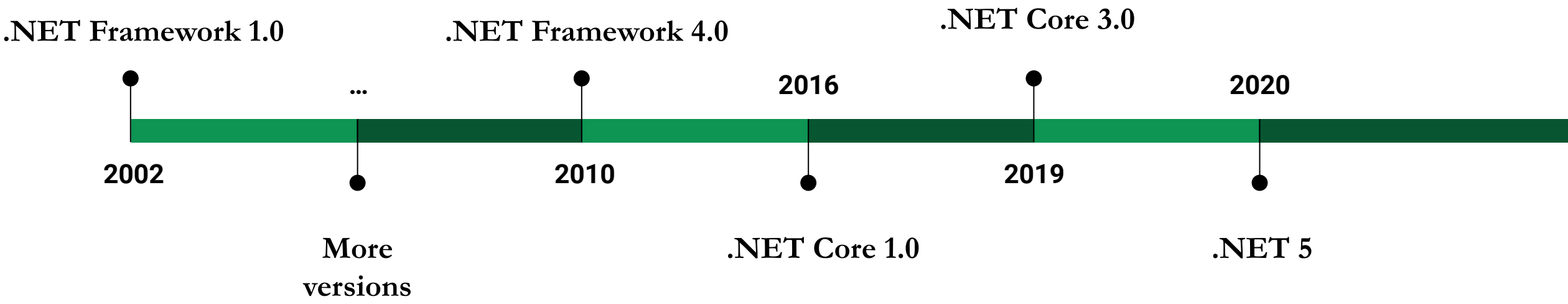
SOFTWARE ENGINEERING I

2nd lecture

Today

- .NET history
- Operators in C#
- Loops
- Type system
- Some of the main OOP pillars (generics, methods, encapsulation)

.NET history (1)



.NET History (2)

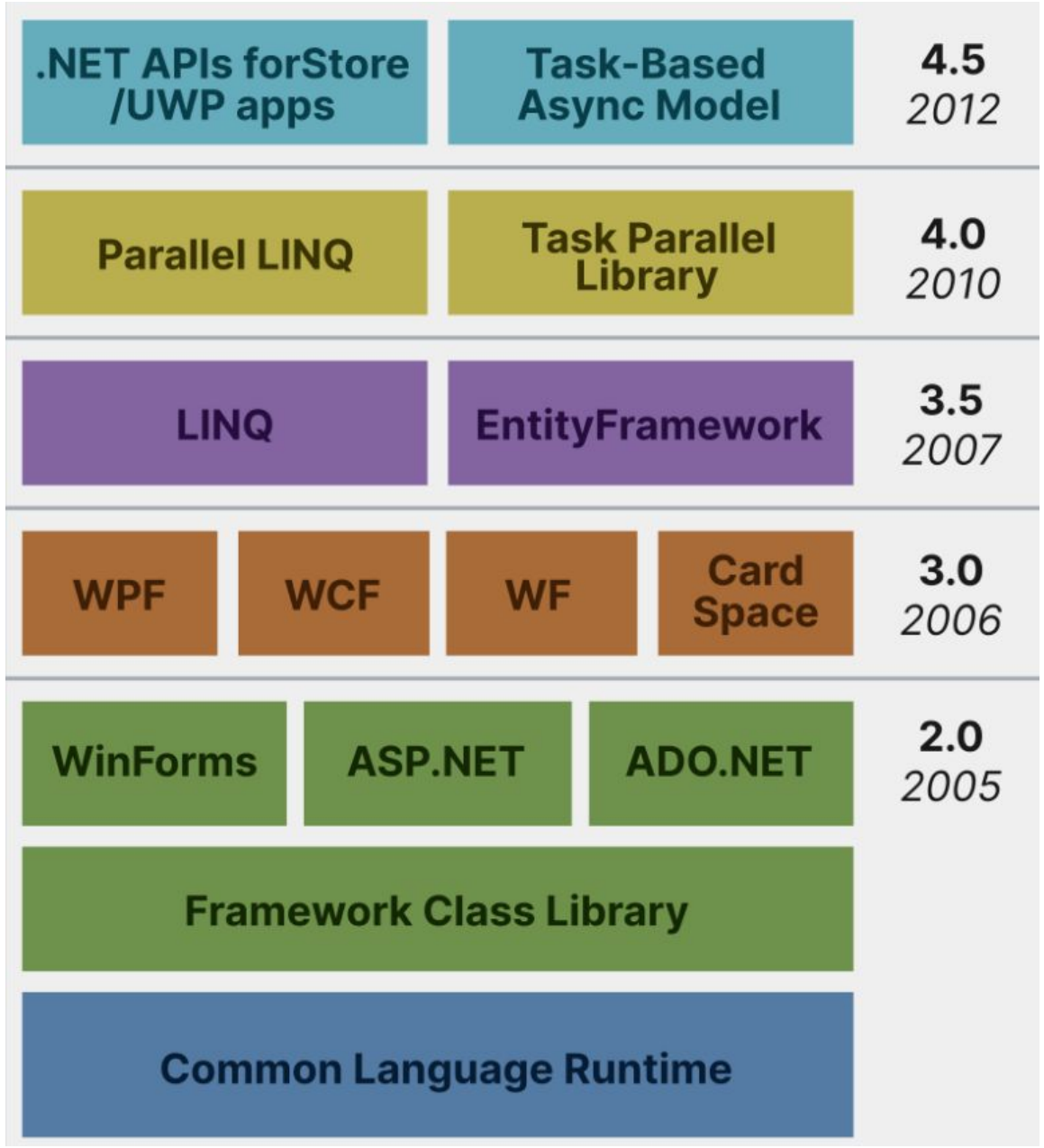


Image source: https://en.wikipedia.org/wiki/.NET_Framework

.NET history (3)

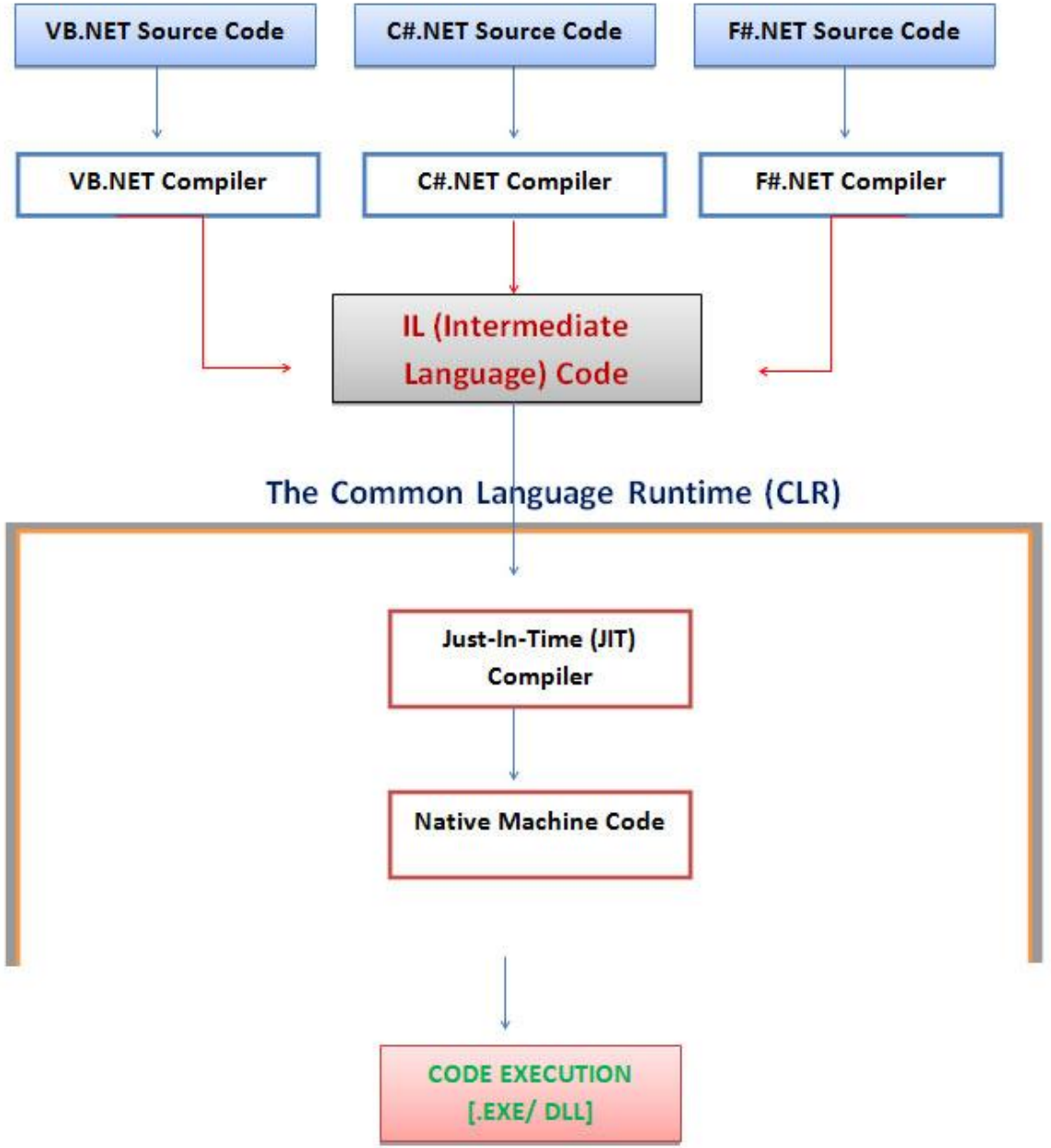
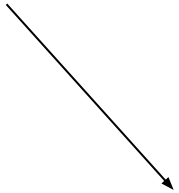


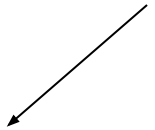
Image source:
http://resources.infosecinstitute.com/net-framework-clr-common-language-runtime/

Code compilation

```
Module Hello
    Public Class TestClass
        Public Shared Sub Main()
            Console.WriteLine("Hello World")
        End Sub
    End Class
End Module
```

```
namespace Hello
{
    public class TestClass
    {
        public static void Main()
        {
            Console.WriteLine("Hello World");
        }
    }
}
```





```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method Test::Main
```

Disassembling

If you are interested you can use tools like:

- https://github.com/icsharpcode/ILSpy

to decompile and check what IL code looks like.

.NET (Core)

- Cross-platform .NET development platform
- Runs on windows/macOS/linux
- Open source https://github.com/dotnet/core
- Languages that can be used with: C#, F# and VB
- .NET 8 scheduled for November 2023

.NET – A unified platform

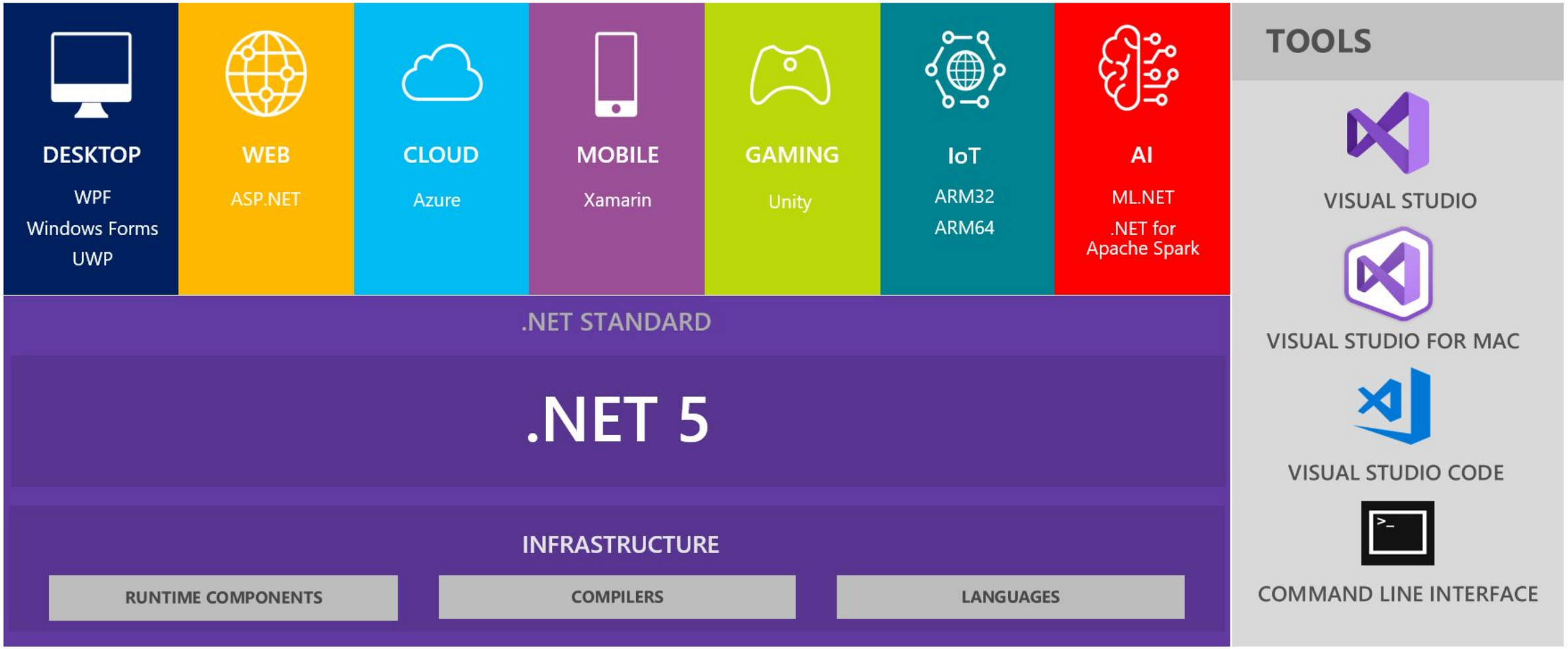


Image source: https://devblogs.microsoft.com/dotnet/introducing-net-5/

Let’s quickly go through the very basics

Arithmetic operators

- A = 10; B = 20

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Comparison operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical operators

- *Short-circuit evaluation*: look in **and.cs**
  A = True, B = False

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Short circuits

```
class LogicalAnd
{
    static bool Method1()
    {
        Console.WriteLine("Method1 called.");
        return false;
    }

    static bool Method2()
    {
        Console.WriteLine("Method2 called.");
        return true;
    }

    static void Main()
    {
        Console.WriteLine("Regular AND:");
        if (Method1() & Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned false.");

        Console.WriteLine("\nShort-circuit AND:");
        if (Method1() && Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned false.");
    }
}
```

```
Regular AND:
Method1 called.
Method2 called.
At least one of the methods returned false.

Short-circuit AND:
Method1 called.
At least one of the methods returned false.
```

Binary operators (1)

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

- A = 0011 1100
- B = 0000 1101
- ----------
- A&B = 0000 1100
- A|B = 0011 1101
- A^B = 0011 0001
- ~A = 1100 0011

Binary operators (2)

- A = 60 (0011 1100), B = 13 (0000 1101)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12. which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Assignment operators (1)

Left operand <**operator**> **Right** operand

Operator	**Description**	**Example**
=	Simple assignment operator. Assigns evaluated value from the right side to the left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. Adds right operand to left operand and assigns the result to left operand	C += A is equivalent to C = C + A
-=	Subtract and assignment operator. Subtracts the right operand from left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply and assignment operator. It multiplies left and right operands and assigns the results to left operand	C *= A is equivalent to C = C * A

Assignment operators (2)

Operator	Description	Example
/=	Divide and assignment operator. Divides left side side operand by right side operator and assigns the result to left side operand	C /= A is equivalent to C = C / A
%=	Module and assignment operator. It takes modulus of left operand of right operand and assigns the result to left operand	C %= A is equivalent to C = C % A
??=	Null coalescing with assignment. Assigns the value to the left operand if it is null. Also returns the resulting value of expression.	A = null B = A ?== 5 Results in A = 5 and B = 5

Assignment operators (3)

Operator	Description	Example
<<=	Bitwise left shift and assignment operator	A = 1; A <<= 1; Result is A = 2;
>>=	Bitwise right shift and assignment operator	A = 2; A >>= 1; Result is A = 1;
&=	Bitwise AND and assignment operator	A = 3; A &= 7; Result is A = 3;
^=	Bitwise XOR and assignment operator	A = 3; A ^= 7; Result is A = 4;
|=	Bitwise OR and assignment operator	A = 3; A |= 7; Result is A = 7;

Syntactic sugar and salt

Sugar

Syntax that allows you to do something easier.

```
var a = 5;
```

Salt

Syntax that forces you to do something to prevent accidents.

```
switch (variable) {
    case “a”:
        break;
}
```

Syntactic saccharin

- Gratuitous syntax that does not making anything any sweeter.

var

- int i = 10;
  - Explicit definition
- var i = 10;
  - Implicit definition
- Automatically inferred from right side.
- Implicit type can be:
  - Any primitive type: int, bool, char etc.
  - Any class
  - Anonymous type

Ternary operator

- condition ? statement if true : statement if false

```
static double mySin(double x)
{
    return x != 0.0 ? Math.Sin(x) : 1.0;
}

static void Main()
{
    Console.WriteLine(mySin(30 * Math.PI / 180.0));
    Console.WriteLine(mySin(0.0));
}
```

Operator overload

Operator	Description
+, -, !, ~, ++, --, true, false	Can be overloaded (unary)
+, -, *, /, %, &, |, ^, <<, >>	Can be overloaded (binary)
==, !=, <, >, <=, >=	Can be overloaded
&&, ||	Cannot do it simple way
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Cannot do it simple way

See **operators.ipnyb**

Null-coalescing operator

Answer = Answer1 ?? Answer2 ?? Answer3 ?? Answer4

```
result = tempResult ?? makeResult();

result = tempResult != null ? tempResult : makeResult();

if (tempResult != null)
{ result = tempResult; }
else
{ result = makeResult(); }
```

switch

- *int* and *string* types (before c# 8)
- *Break* is a must:

```
switch (number)
{
    case 0:
    case 1:
    case 2:
        Console.WriteLine("Buvo 0-1-2");
        break;
    case 3:
    case 4:
        Console.WriteLine("Buvo 3-4");
        break;
    default:
        Console.WriteLine("Other");
        break;
}
```

```
static void Main(string[] args)
{
    switch (args[0])
    {
        case "copy":
            //...
            break;

        case "move":
            //...
            goto case "delete";

        case "del":
        case "remove":
        case "delete":
            //...
            break;

        default:
            //...
            break;
    }
}
```

switch expressions (c# 8)

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand)),
    };
```

Switch expressions (c# 8 and 10)

- Property pattern – switch to filter on specific property;
- Tuple pattern – switch to filter on more than one property;
- Positional pattern – using deconstruct
- More to read - https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#switch-expressions
- See **switch.ipnyb**

Cycles: for and foreach

```
// for statement syntax
for(initializer; condition; iterator)
{
   statement(s);
}
```

```
// infinite for loop in C#
for(;;)
{
   statement;
}
```

```
// foreach syntax
foreach(type in collection)
{
   statement;
}
```

```
int[] arrGrades = new int[] { 78, 89, 90, 76, 98, 65 };
int total = 0;
int gradeCount = 0;
double average = 0.0;

foreach (int grade in arrGrades)
{
    total = total + grade;
    gradeCount++; //
}
average = total / gradeCount;
Console.WriteLine(average);
```

While and do-while

```
// while statement syntax
while(condition)
{
    statement;
}
```

```
int someValue = 0;
while (someValue < 10)
{
    Console.WriteLine(someValue);
    someValue++;
}
```

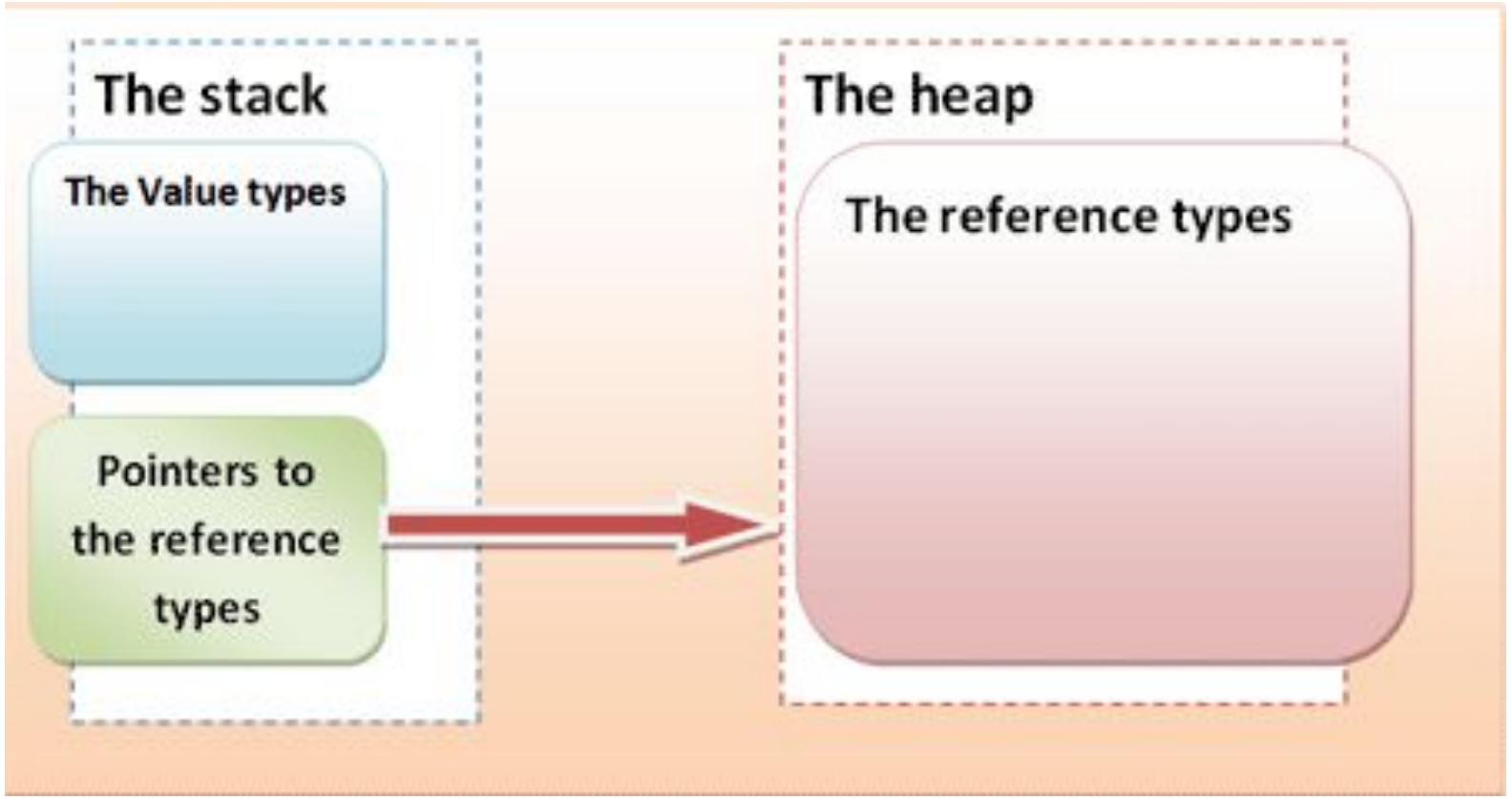
```
// do-while loop syntax
do
{
    statement;
} while (condition);
```

```
int someValue = 0;
do
{
    Console.WriteLine(someValue);
    someValue++;
} while (someValue < 10);
```


QUESTION:
DO YOU HAVE ANY QUESTIONS?
imgflip.com

Types

- *Value types*
- *Reference types*
- *Pointer types*



Types

- Ref vs value

Value types

- Enumerations
- Structs (all below „under the hood“ is struct):
  - Numeric types
    - Integral types
    - Floating-point types
    - decimal
  - bool
  - User defined structs

Value types

TYPE	VALUES	SIZE	.NET TYPE
`bool`	true, false	1 byte	`System.Boolean`
`Byte`	0–255	1 byte	`System.Byte`
`char`	0000–FFFF Unicode	16-bit	`System.Char`
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	28–29 significant digits	`System.Decimal`
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15–16 digits	`System.Double`
`enum`	User-defined set of name constants		
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 digits	`System.Single`
`int`	–2,147,483,648 to 2,147,483,647	Signed 32-bit	`System.Int32`
`long`	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit	`System.Int64`
`sbyte`	–128 to 127	Signed 8-bit	`System.SByte`

Value types

TYPE	VALUES	SIZE	.NET TYPE
`short`	–32,768 to 32,767	Signed 16-bit	`System.Int16`
`struct`	Includes the numeric types listed in this table as well as bool and user-defined structs		
`uint`	0 to 4,294,967,295	Unsigned 32-bit	`System.UInt32`
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit	`System.UInt64`
`ushort`	0 to 65,535	Unsigned 16-bit	`System.Uint16`

- GetType:

```
System.Int32
System.Int64
```

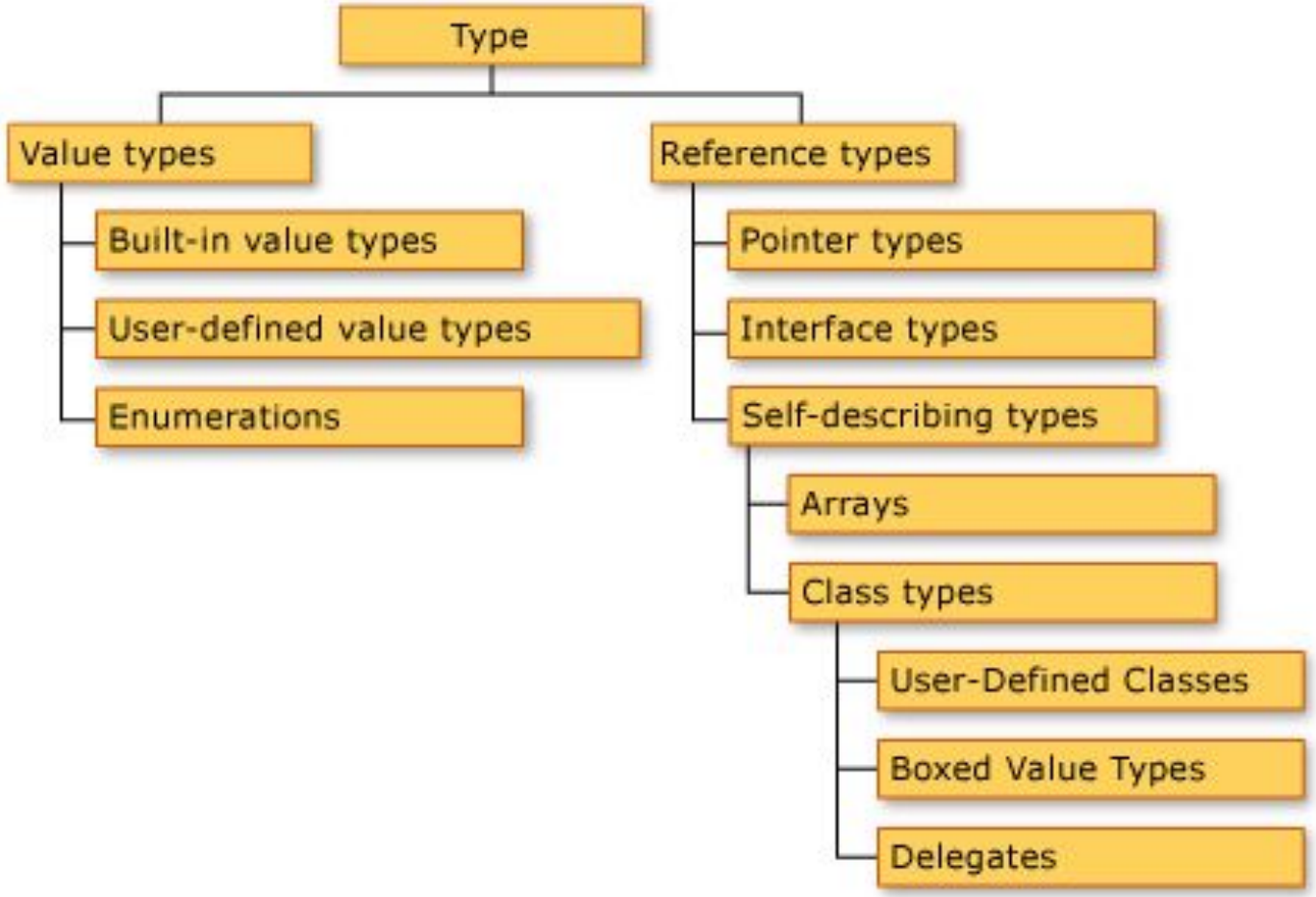
```
int n1 = 1;
long n2 = 1;
Console.WriteLine(n1.GetType().ToString());
Console.WriteLine(n2.GetType().ToString());
```

Formating numeric types

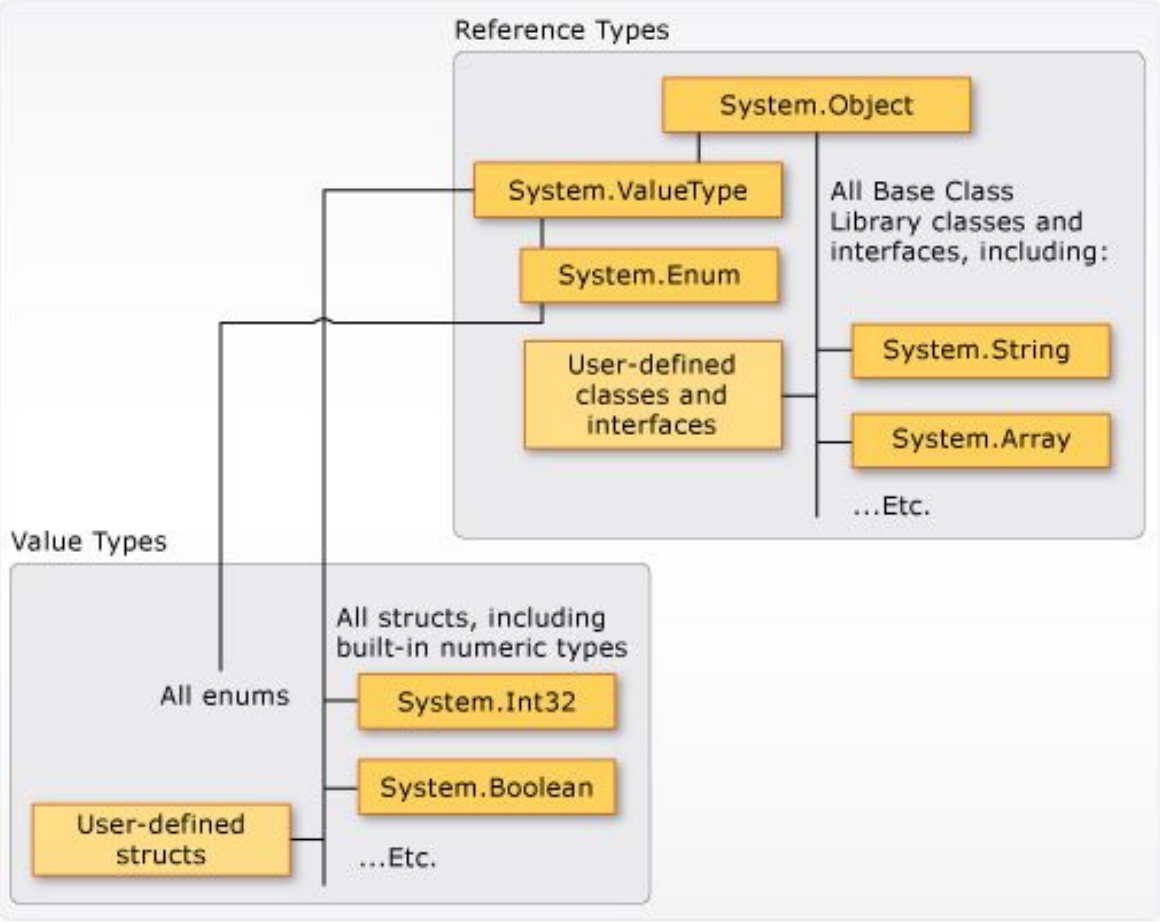
Symbol	Definition	Example	Result
C or c	***C**urrency*	Console.Write("{0:C}", 2.5); Console.Write("{0:C}", -2.5);	$2.50 ($2.50)
D or d	***D**ecimal*	Console.Write("{0:D5}", 25);	00025
E or e	*Scientific*	Console.Write("{0:E}", 250000);	2.500000E+005
F or f	***F**ixed-point*	Console.Write("{0:F2}", 25); Console.Write("{0:F0}", 25);	25.00 25
G or g	*General*	Console.Write("{0:G}", 2.5);	2.5
N or n	***N**umber*	Console.Write("{0:N}", 2500000);	2,500,000.00
X or x	*He**x**adecimal*	Console.Write("{0:X}", 250); Console.Write("{0:X}", 0xffff);	FA FFFF

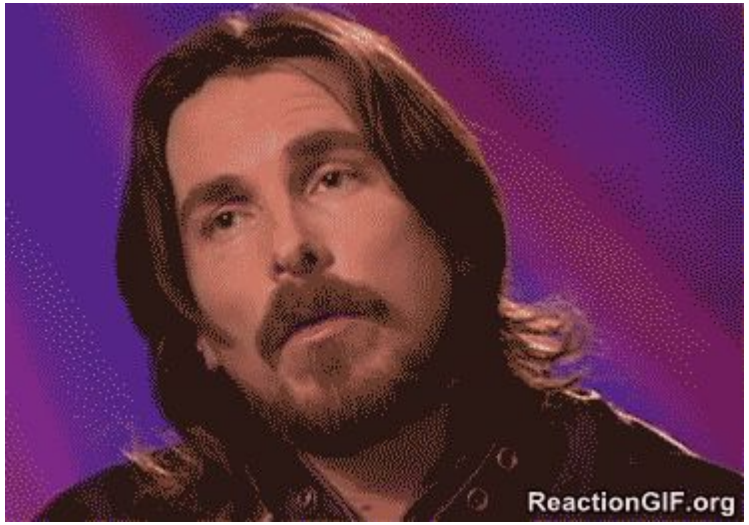

VILNIAUS UNIVERSITETAS
·1579·
UNIVERSITAS VILNENSIS

Type system


Type
Value types
Built-in value types
User-defined value types
Enumerations
Reference types
Pointer types
Interface types
Self-describing types
Arrays
Class types
User-Defined Classes
Boxed Value Types
Delegates

Type system


Reference Types
System.Object
System.ValueType
System.Enum
User-defined classes and interfaces
All Base Class Library classes and interfaces, including:
System.String
System.Array
...Etc.
Value Types
All structs, including built-in numeric types
All enums
System.Int32
System.Boolean
User-defined structs
...Etc.


ReactionGIF.org

Enum

- Value type, an enumeration, a distinct type that consists of a set of named constants called the enumerator list
- Has:
  - Name,
  - Hidden integer type
    - e.g. Byte, Int32 ar UInt64,
  - A set of fields
- Cannot:
  - Have methods
  - Implement interfaces
  - Define properties or events
  - Be *generic*

Enum

```
enum Months
{
    Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec
};

class Program
{
    static void Main(string[] args)
    {
        string name = Enum.GetName(typeof(Months), 8);
        Console.WriteLine("The 8th month in the enum is " + name);
        Console.WriteLine("The underlying values of the Months enum:");

        //by changing "int" to "var" below, enumeration of ints would be changed to name values
        foreach (int values in Enum.GetValues(typeof(Months)))
        {
            Console.Write(values + " ");
        }
    }
}
```

```
The 8th month in the enum is Aug
The underlying values of the Months enum:
1 2 3 4 5 6 7 8 9 10 11 12
```

Enum

- Hidden type – int
- Starts with 0:

```
// enum called Months, using default initializer
enum Months { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec };
// enum call Months, using an overidden initializer
enum Months { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec };
```

```
enum AirSpeeds
{
    Vx = 55,
    Vy = 65,
    Vs0 = 50,
    Vs1 = 40,
    Vne = 120
}
```

Enum with [Flags

```
[Flags]
public enum CarOptions
{
    // The flag for SunRoof is 0001.
    SunRoof = 0x01,
    // The flag for Spoiler is 0010.
    Spoiler = 0x02,
    // The flag for FogLights is 0100.
    FogLights = 0x04,
    // The flag for TintedWindows is 1000.
    TintedWindows = 0x08,
}

class Program
{
    static void Main()
    {
        // The bitwise OR of 0001 and 0100 is 0101.
        CarOptions options = CarOptions.SunRoof | CarOptions.FogLights;

        // Because the Flags attribute is specified, Console.WriteLine displays
        // the name of each enum element that corresponds to a flag that has
        // the value 1 in variable options.
        Console.WriteLine(options);

        Console.WriteLine((int)options);
    }
}
```

```
SunRoof, FogLights
5
```

See **enums.ipnyb**

Struct

- Value type
- Defines:
  - Data
  - Operations on data
- Possible to do:
  - Create an *instance*
  - Pass them as parameters
  - Save as local types
  - Have it as a field inside both – reference and value types
- Struct **can** implement an interface.
- Implicitly *sealed*.
- Cannot be *abstract*.

Struct

Has:

- Fields
- Methods (incl. Constructor)
- Properties
- Operators
- Events
- Delegates.


```
public struct Coordinates
{
    public int x, y;

    public Coordinates(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}

static void Main()
{
    // Initialization possible with empty (default) constructor:
    Coordinates c1 = new Coordinates();

    //as well as with provided constructor
    Coordinates c2 = new Coordinates(1, 1);

    //as well as no initialization at all:
    Coordinates c3;

    // Initialize:
    c3.x = 2;
    c3.y = 2;

    Console.Write("CoOrds 1: ");  //0;0
    Console.WriteLine("x = {0}, y = {1}", c1.x, c1.y);

    Console.Write("CoOrds 2: ");
    Console.WriteLine("x = {0}, y = {1}", c2.x, c2.y);

    Console.Write("CoOrds 3: ");
    Console.WriteLine("x = {0}, y = {1}", c3.x, c3.y);
}
```

```
CoOrds 1: x = 0, y = 0
CoOrds 2: x = 1, y = 1
CoOrds 3: x = 2, y = 2
```


```
class DemoClass
{ public string value;}

struct DemoStruct
{ public string value;}

class Program
{
    //To return the changed value of s ref or out might be used:
    //out: http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx
    //ref: http://msdn.microsoft.com/en-us/library/14akc2c7.aspx
    public static void ChangeStruct(DemoStruct s)
    {
        s.value = "Suddenly, le wild NEW value appers";
    }

    public static void ChangeClass(DemoClass c)
    {
        c.value = "Suddenly, le wild NEW value appers";
    }

    public static void Main()
    {
        DemoStruct s = new DemoStruct();
        DemoClass c = new DemoClass();

        s.value = "Normal value";
        c.value = "Normal value";

        //this one will not change value of s, beacuse ChangeStruct get's it as a value,
        //and it is not returned.
        ChangeStruct(s);
        ChangeClass(c);

        Console.WriteLine("s.value = {0}", s.value);
        Console.WriteLine("c.value = {0}", c.value);
    }
}
```

```
s.value = Normal value
c.value = Suddenly, le wild NEW value appers
```

See **ref_vs_value.ipnyb**

struct differences from *class*

- Value vs reference
- Struct cannot be null
- Class is to be used for larger objects.
- Classes can be inherited, structs - cannot.
- Struct does not have destructor
- Struct cannot be abstract
- In struct you can *override* only these:
  - Equals()
  - GetHashCode()
  - GetType()
  - ToString()

struct differences from *class*

- Events in classes are automatically *locked* and thread-safe, while in structs - not *thread-safe*.
- Struct cannot have ***volatile*** (it indicates that the variable might be accessed by several threads simultaneously and thus compiler should not do certain optimizations to it).
- *Sizeof* does not work with classes
- Field initialization is allowed from c# 10 if you have an empty parameterless constructor

struct differences from *class*

- Struct cannot have not empty constructor without params (**actually can from c# 10**)
- Static constructor works only with classes.
- *Equals* is behaving differently. See **struct_equality.ipnyb**

Next lecture we continue from here

record

- Syntactic sugar for working with types responsible for carrying values.
- Equality works by comparing the *values*.
- Can be mutate using *with* keyword.
- See **records.ipnyb**

Literature

- Basics:
  - MCSD Certification toolkit: second chapter, a half of the third chapter
  - MSDN: Operator Overloading Tutorial
- Extra: nullable types (highly recommended!)
  - http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx
  - http://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx
- Extra-extra:
  - Decimal floating point in .NET. *C# in Depth*. (on web).

OOP basics

- **Encapsulation** – programming methodology,
  - Forbids access to specifics of the class.
  - Allows modify class properties and fields only through exposed methods.
- **Inheritance**:
  - Possibility to reuse, extend or modify class implementation.
- **Polymorphism**:
  - At run time objects of derived class may be treated as objects of a base class in methods, parameters or collections. Also, derived classes may implement different methods behaviour than base class, if base class methods are marked as *virtual*

Classes

- Class:
  - Description, specifying some sort of object data structure and behavior.
- Single responsibility – classes are created using this principle!
- Class can contain:
  - constructor, constants, fields, methods, properties, delegates, classes and more.

SRP (Single responsibility principle)

„A class should only have one reason to change“

~

„A class should only have one responsibility“

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may break the class' ability to fulfil other responsibilities.

Can be applied to:
- Methods
- Modules
- Etc.

Bad example – what are reasons to change?

```
Console.WriteLine("Please enter your name:");
var name = Console.ReadLine();
if (string.IsNullOrEmpty(name))
{
    Console.WriteLine("Please provide a valid name!");
    return;
}

Console.WriteLine("Please enter your surname:");
var surname = Console.ReadLine();
if (string.IsNullOrEmpty(surname))
{
    Console.WriteLine("Please provide a valid username!");
    return;
}

var username = name.Substring( startIndex: 0, length: 2)
               + surname.Substring( startIndex: 0, length: 2);
Console.WriteLine($"Hi {name} {surname}! Your username is {username}");
```

Possible reasons to change

```
Console.WriteLine("Please enter your name:");
var name = Console.ReadLine();
if (string.IsNullOrEmpty(name))
{
    Console.WriteLine("Please provide a valid name!");
    return;
}

Console.WriteLine("Please enter your surname:");
var surname = Console.ReadLine();
if (string.IsNullOrEmpty(surname))
{
    Console.WriteLine("Please provide a valid username!");
    return;
}

var username = name.Substring( startIndex: 0, length: 2)
             + surname.Substring( startIndex: 0, length: 2);
Console.WriteLine($"Hi {name} {surname}! Your username is {username}");
```

What if I decide to read from DB?

What if I decide to change validation logic?

What if I change generation logic?

Better example

```
Console.WriteLine("Please enter your name:");
var name = Console.ReadLine();
if (PersonDataValidator.IsNameValid(name))
{
    Console.WriteLine("Please provide a valid name!");
    return;
}

Console.WriteLine("Please enter your surname:");
var surname = Console.ReadLine();
if (PersonDataValidator.IsSurnameValid(surname))
{
    Console.WriteLine("Please provide a valid username!");
    return;
}

var username = UsernameGenerator.GenerateUsername(name, surname);
Console.WriteLine($"Hi {name} {surname}! Your username is {username}");
```

Bad example – many responsibilities

```
class Customer
{
    public void Add()
    {
        try
        {
            // Database code goes here
        }
        catch (Exception ex)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }
}
```

Better example

```
class FileLogger
{
    public void Log(string error)
    {
        System.IO.File.WriteAllText(@"c:\Error.txt", error);
    }
}
class Customer
{
        private FileLogger _logger = new FileLogger();
        public virtual void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                _logger.Log(ex.ToString());
            }
        }
}
```

Possible inheritance

Inheritance	Example
None	`class ClassA { }`
Unitary	`class DerivedClass: BaseClass { }`
None, implementing interfaces	`class ImplClass: IFace1, IFace2 { }`
Unitary and implementing interfaces	`class ImplDerivedClass: BaseClass, IFace1 { }`

Access modifiers

Accessibility	Description
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly (same .dll).
protected internal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

Default accessibility (1)

- Top level types (not nested) can only be **internal** or **public**. Default – **internal**.

Members of	Default accessibility	Other allowed accessibility levels
interface	**public**	**internal**
struct	**private**	**public** **internal** **private**

Default accessibility (2)

Members of	Default accessibility	Other allowed accessibility levels
enum	**public**	**internal**
class	**private**	**public** **protected** **internal** **private** **protected internal** **private protected**

Static vs *instance*

- Static class is in a way the same as non static, difference is that there is no possibility to create static class object. (*no* ***new***).
- Static classes should be used, when you don't need to save state.
- Static class can only have static methods, non static class can have both.
- Memory is divided to three parts when its loaded: Stack, Heap, and Static (in .NET it is known as ***High Frequency Heap***).

Constructor

- Constructor is a method that is being called when class is being initialized.

```
class ClassName
{

    public ClassName()
    {
        //optional initializing statements;
    }
}
```

Methods (1)

- Syntax:

```
[accessibilityModifier] returnType  name (parameters)
{
    statements;
}
```

- For value types as a parameters the copy of a value is passed, for reference types as a parameters – reference is passed. Look **ref_vs_value.ipnyb**

Methods (2)

- Method overloading:
  - Different signatures (return type is not counted in signature)
- Abstract method:
  - Defined method signature, but there is no implementation.
  - Only possible in abstract class.
  - Inherit class must implement abstract method (using override). Look **AbstractMethod.cs**
- Virtual method: allows (optional) to override it and must have implementation.

```
public double calcArea(double radius)
{
    double area = Math.PI * (radius * radius);
    return area;
}

public double calcArea(double length, double width)
{
    double area = length * width;
    return area;
}
```

Methods (3)

- Extensions:
  - Possibility to extend standard class.
  - Syntax:

public static type MethodName(this typeToExtend str)

  - Look **ExtensionMethods.cs**
  - Cannot override standard extension methods:
  - Works in same namespace or by importing namespace with „**using**“

Named parameters

- Allows to pass parameters in different order than method signature.
- Brings more clarity.
- Named parameters can be passed after standard ones, but not before.

```
public static double rectArea(double length, double width)
{
    return length * width;
}

static void Main(string[] args)
{
    double area11 = rectArea(length: 1.0, width: 1.0);
    double area12 = rectArea(width: 1.5, length: 1.5);
    double area22 = rectArea(2.5, width: 2.0);
    double area23 = rectArea(length: 3.0, 3.5);
}
```

Optional parameters

- Syntax:
  - Providing default (or constant) value to a parameter
  - „new ValType()“, if parameter is reference type (class)
  - default(ValType), if parameter is value type (e.g enum or struct).
- Allowed to be used in:
  - methods, constructors, indexed properties and delegates

```
public void displayName(string first, string initial = "", string last = "")
{
    Console.WriteLine(first + " " + initial + " " + last);
}
```

Optional parameters

- Must be placed in the end of parameters list.
- If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters.

```
public void displayName(string first, string initial = "", string last = "")
{
    Console.WriteLine(first + " " + initial + " " + last);
}
```

Encapsulation

OOP encapsulation term means that some entity members, behavior and fields can be wrapped in a class.

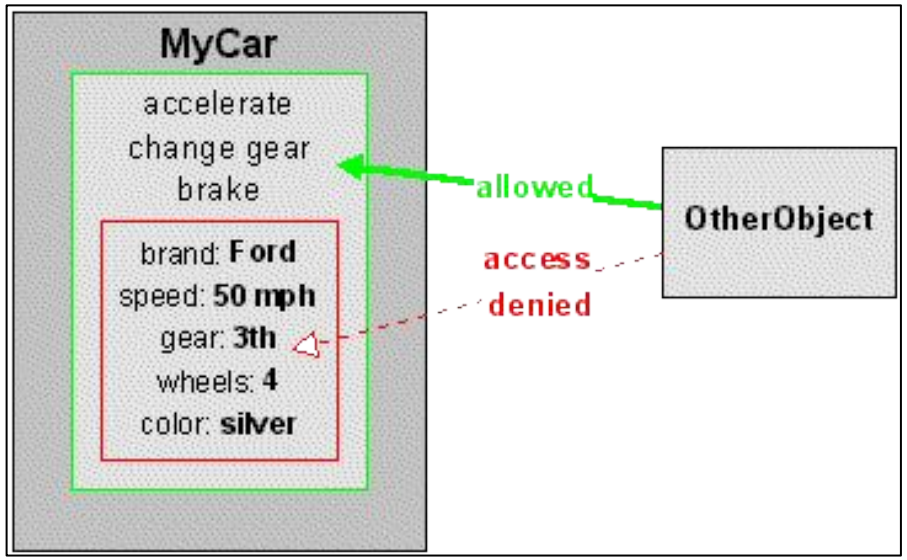
Meaning, that internal (private) fields are hidden from the user, and you can only modify those using exposed (public) methods.

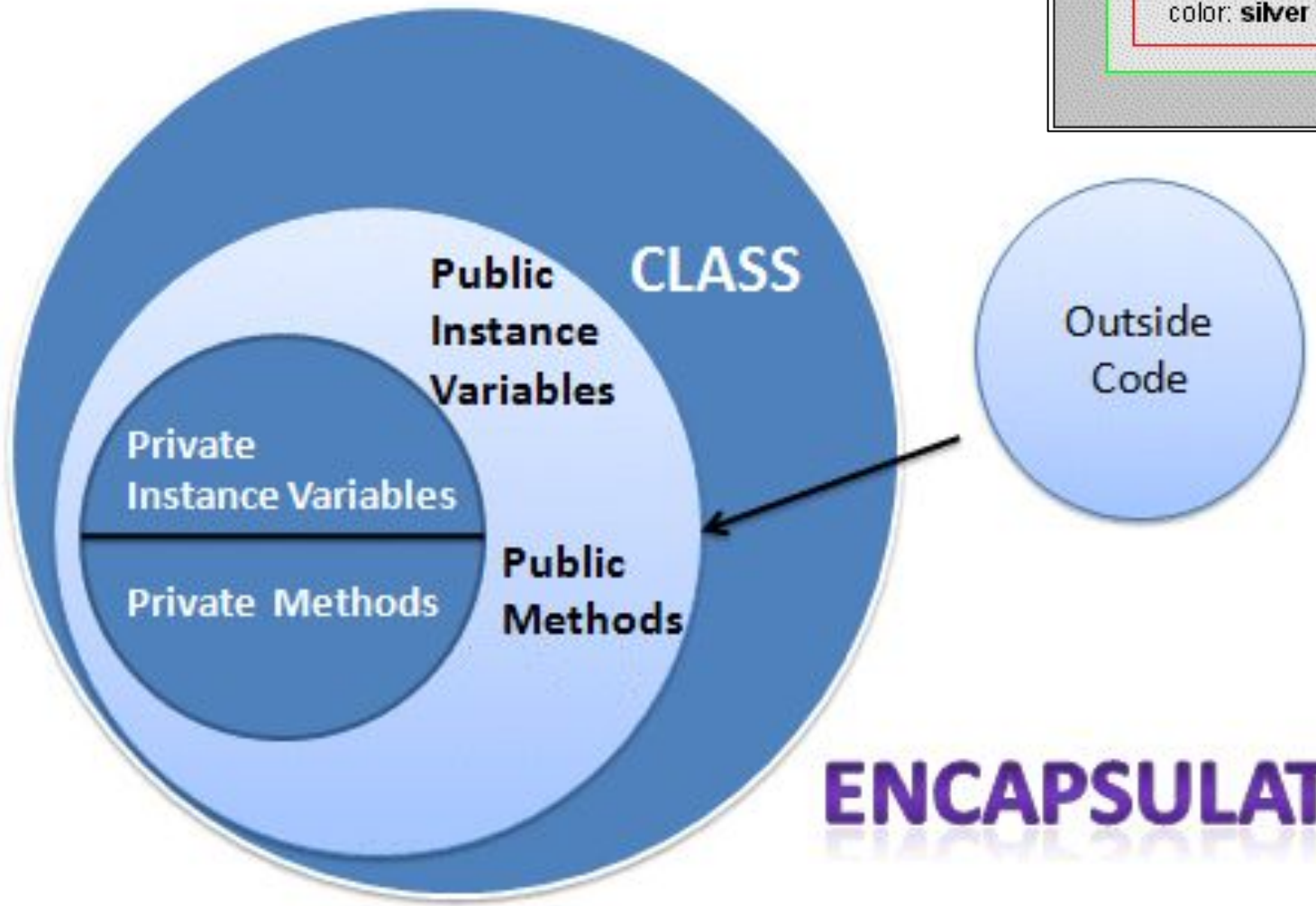
Encapsulation allows to be loosely coupled from the actual implementation, and that allows us:

To change from one type of object to another;

Refactor/change class without changing usages.

Encapsulation





Properties

```
private string firstName;
public string FirstName
{
    get { return firstName; }
    set { firstName = value; }
}
```

- Property – method to access private field.
- Can be: *public*, *private*, *protected*, *internal* or *protected internal*.
- Can be *static* – enables to access it without creating instance of a class.

Auto-properties

- If you don't need additional logic inside.
- Compiler creates hidden private backing field:

- Properties can have restricted access:

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

- Look **Properties.cs**

```
public string Name { get; private set; }
public string Address { get; private set; }
```

Generics

```
public class GenericQueue<T>
{
    public void Enqueue(T obj)
    {
        ; //some wild stuff
    }
    public T DeQueue()
    {
        T result = default(T);
        return result;
    }
}
```

- ***Generics***: enables specifying type of class or method only when creating it.
- Advantages: code reusability, types safe, efficiency (no need for unboxing).
- C# syntax **<T>** (letter(s) can be different).
- Look **Generics1.cs**

Generic method

- Accepts specified T type parameters:

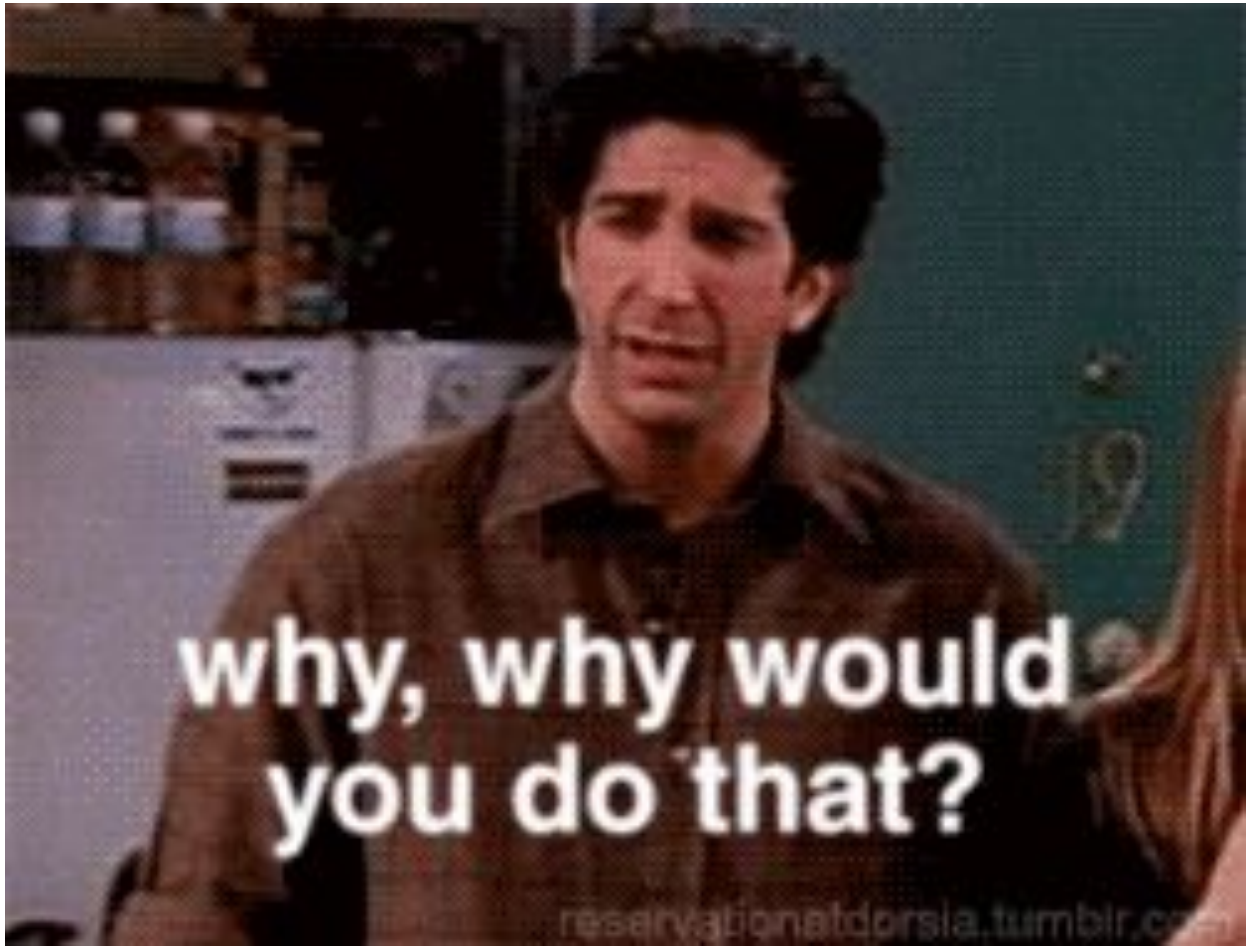
```
public void Swap<T>(ref T valueOne, ref T valueTwo)
{
    T temp = valueOne;
    valueOne = valueTwo;
    valueTwo = temp;
}
```

- It is possible to have more types

```
public void Fix<T1, T2>(ref T1 valueOne, ref T2 valueTwo)
{
    //something here
}
```

- **ref** specifies that reference is passed
- Look **Generics2.cs**

Generics



Generics

- Ensure types safety:
  - **List<string>** must be filled with only string values.
- Compiler would display an error if you would try to add integer to a List<string>. If you would use ArrayList – it would not, because it accepts object.

Generics

```
// without generics
var arrayList = new ArrayList();
arrayList.Add("1");
// no compiler error
arrayList.Add(1);

// with generics
var listOfStrings = new List<string>();
listOfStrings.Add("1");
// generates compiler error
listOfStrings.Add(1);
```

struct System.Int32

Represents a 32-bit signed integer.To browse the .

Argument 1: cannot convert from 'int' to 'string'

Generics

- Faster than using *object* because it prevents boxing/unboxing to happen or casting to required type/value from object.
- Promotes code reusability:
  - More: MSDN - When to Use Generic Collections
  - Look **Generics3.cs**

Next time

- Continuation on OOP pillars:
  - Inheritance
  - Polymorphism
- Conversions
- Standard .NET interfaces


QUESTION:
DO YOU HAVE ANY QUESTIONS?
imgflip.com