Pentalog

Module 01 – Week 03

Clean Code & Data Types

DATA ANALYSIS
UX/UI
FRONT-END
TECHNOLOGY
NEAR/OFFSHORE
AGILITY
BACK-END
SPRINT
KANBAN
CAMPAIGNS
GROWTH HACKING
SCRUM
BACKLOG
DEVOPS
DESIGN
CONTINUOUS INTEGRATION
MOBILE
QA
AUTOMATION
RESPONSIVE
UNIT TESTING

Today’s Agenda

Clean Code and best practices for writing code in C#

Data Types:

- Integral, Char, Floating, Boolean
- Structs
- Reference types vs. Value types
- Enums
- Nullable types
- Immutable types (string, DateTime, TimeSpan)

Our Code – Git Repo

https://github.com/nadiacomanici/PentastagiuDotNet2019Brasov

Myth vs Reality

What people think programming means

Programmers
spend their time writing code

✗

What programming actually means

Programmers:
- **Read code (50%)**
- Test code (25%)
- Write code (20%)
- Delete code (5%)

✓

Clean Code

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand"

(Martin Fowler)

Clean Code

The way a software developer writes code is his business card.

- Bibliography:
  - Clean Code - A Handbook of Agile Software Craftsmanship (Robert C. Martin)
  - Code Complete: A Practical Handbook of Software Construction (Steve McConnell)

Why it’s important to write clean code?

- Less time to accommodate to the code
- Code is easier to read
- Code is easier to change and fix
- Less possible bugs from misunderstanding code
- Easier to understand the code by others
- Easier to understand it yourself after some time passes

Coding Standards (1)

- A coding standard is a set of recommendations about how to write code and what are the naming and organizing conventions of code.

Why?

- Better code quality
- Faster development and integration of new members
- Same writing style for the entire team – proof of professionalism
- Easier to add new team members
- Easier to work on someone else's piece of code

Coding Standards (2)

- Each programming language has its own coding standard, usually recommended by the authors of the language
- Each team should adopt a standard that best fits their needs and is appropriate to the programming language they use

- More:
  - Microsoft recommendations: https://msdn.microsoft.com/en-us/library/ff926074.aspx
  - Tools: FxCop, Sonar etc.

Why don’t we write clean code?

BEGINNER,
NOT ENOUGH EXPERIENCE,

HURRY,
DEADLINES,
PRESSURE

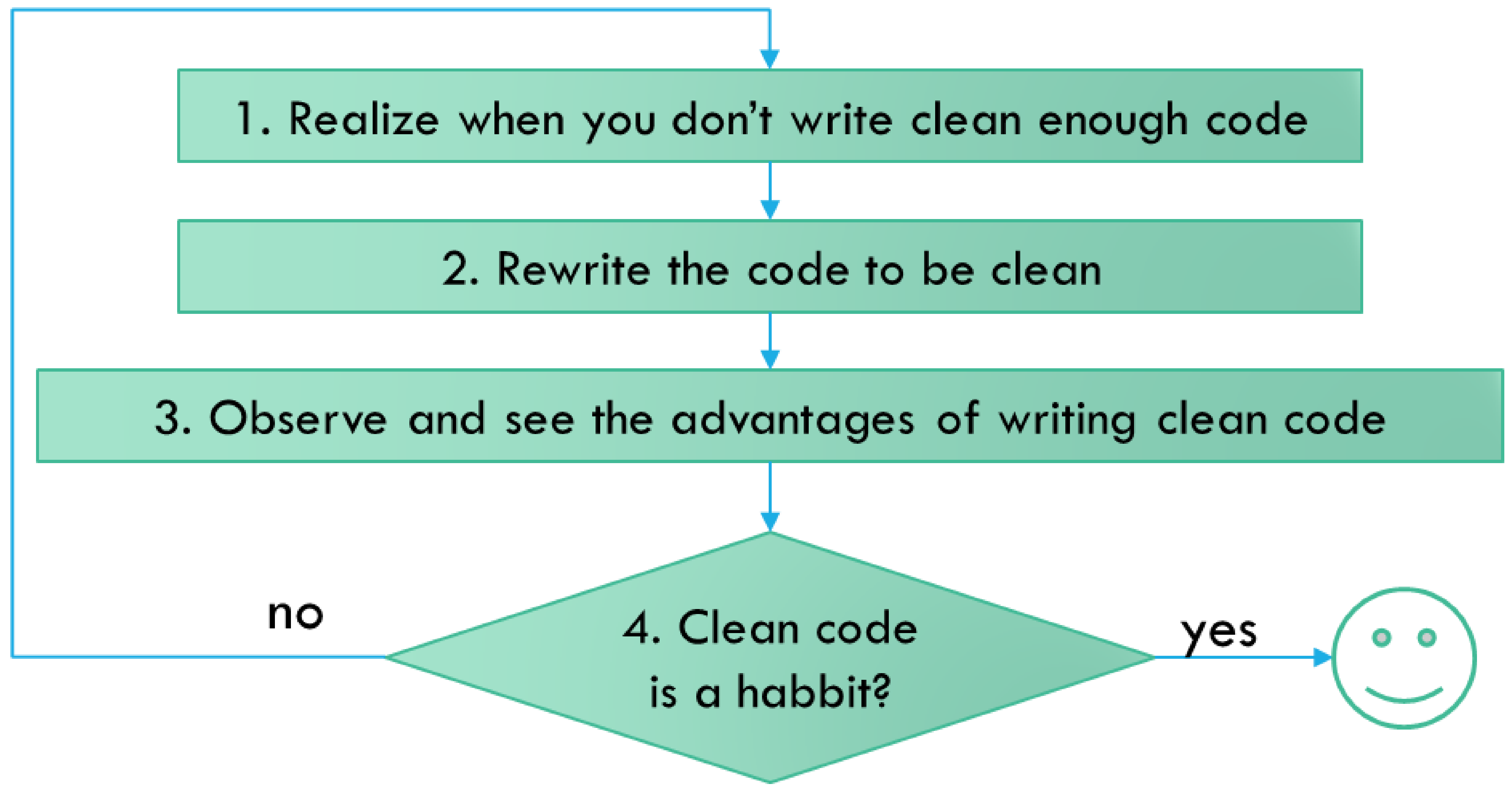
TEST CODE,
FREQUENT CHANGES
NO LONG-TERM VISION

LAZYNESS,
INDIFERENCE,
UNPROFESSIONALISM

LATER = NEVER

How do we learn to write clean code?

- It’s not enough to know how clean code looks like: It’s like tasting food – we can say if it’s good or not, but that doesn’t mean we know how to prepare it to be good.


1. Realize when you don’t write clean enough code
2. Rewrite the code to be clean
3. Observe and see the advantages of writing clean code
4. Clean code is a habbit?
no
yes

Code Improvement Techniques

- ✓ Read code and compare clean code with dirty code
- ✓ Verbalize code
- ✓ Code Review
- ✓ Pair Programming
- ✓ Feedback
- ✓ Try to unit test it

Comments: //

The optimization operator

Comments – when to use and when not to use

- Comments should optimize reading and understanding code, but should not compensate for bad code.

<u>When to use comments:</u>

- They have to be useful and written only when they are needed because the code itself is not enough to understand
- Up to date and reflecting what the code does
- Brief and explicative
- Clear, not misleading

<u>When NOT to use comments:</u>

- Comments should not compensate for bad code. You should just rewrite that code
- Redundant – not DRY
- Zombie code
- Too much text that can be easily understood from code

Where to Place Comments?

- Always before the code block
- Indented at the same level as the code
- Space after // or /*

Good

```
// list must contain exactly 3 elements
bool isValid = (cases.size() == 3);
```

Evil

```
bool isValid = (cases.size() == 3);//list must contain 3 elements
```

Comments – Encapsulation

- Easier to read and understand
- Each class should be responsable for it’s own internal logic
- Encapsulation - Don’t expose more then needed

Evil

```
// checks if the employee is eligible to get extra vacation days
if ((employee.Flags == WorkSchedule.FullTime) && employee.YearsOfExperience > 10)
```

Good

```
if (employee.IsEligibleForExtraVacationDays)
```

Comments – Useless

- Funny or apology, but useless
  - Don’t answer „what did the author want to say?”

Evil

```
// if you made it this far without having a nervous breakdown
// congratulations! Go get yourself a beer!

// Magic. Do not touch

// When I wrote this, only God and I understood what I was doing.
// Now, only God knows

// Bug #1234

// Sorry, this crashes and I don’t know why

// Here starts the hack – See John
```

Comments – Redundant

- Comments that state the same as the line of code itself
- Methods that have good names, don’t need description that is actually the same as the method name (DRY)

Evil

```
// returns true
return true;

/// <summary>
/// Counts the females
/// </summary>
/// <returns></returns>
public int CountFemales()
```

Comments – Misleading

- Compensates bad code
- Does not reflect reality

Evil

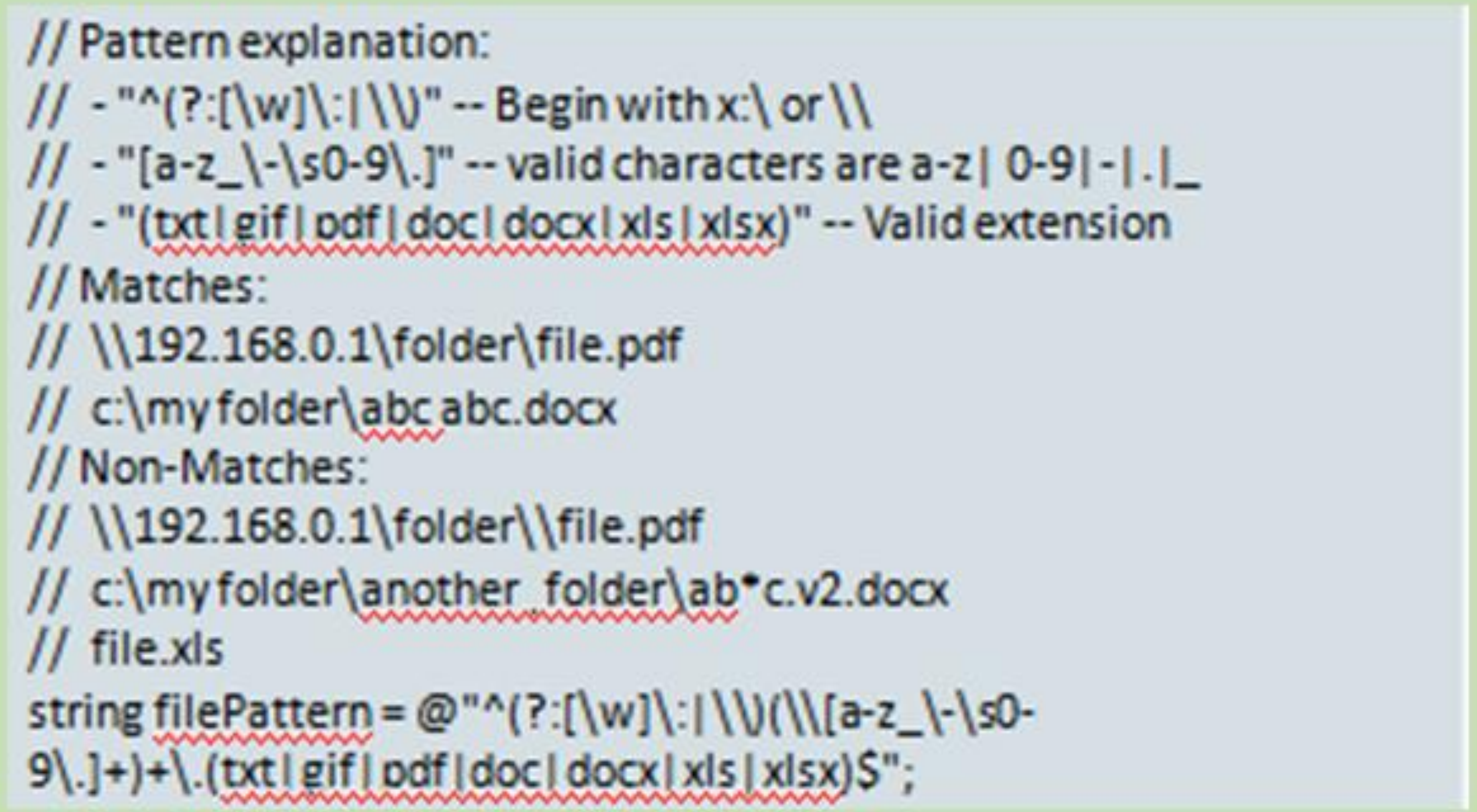
```
public int GetRandomNumber()
{
   // chosen by rolling die
   return 4;
}
```

Comments – Zombie Code

- Unused code should not be commented – but removed
- You have the code under source control
- Makes reading more difficult and distracts attention from the active code
  - Clutters reading code with “noise code”
  - Search results might return zombie code as well
- Creates ambiguity – was the code commented by mistake?

Comments – Useful

Good



```
// Pattern explanation:
//  - "^(?:[\w]\:|\\)" -- Begin with x:\ or \\
//  - "[a-z_\-\s0-9\.]" -- valid characters are a-z| 0-9|-|.|_
//  - "(txt|gif|pdf|doc|docx|xls|xlsx)" -- Valid extension
// Matches:
//  \\192.168.0.1\folder\file.pdf
//  c:\my folder\abc abc.docx
// Non-Matches:
//  \\192.168.0.1\folder\\file.pdf
//  c:\my folder\another_folder\ab*c.v2.docx
//  file.xls
string filePattern = @"^(?:[\w]\:|\\)(\\[a-z_\-\s0-
9\.]+)+\.(txt|gif|pdf|doc|docx|xls|xlsx)$";
```

Braces {}

Keep your keyboard close. And your code closer.

Braces - Blocks of Code

- The blocks of code in C# are marked with curly braces { and }
- The braces should be each on a separate line, without any other command/comment
- The braces should be specified even if there is only 1 instruction in that code block

Evil

```
public Student GetStudentById(int id) {
    return students.SingleOrDefault(s => s.Id == id);
}
```

```
if (x > 0)
    Console.WriteLine("Positive");
else Console.WriteLine("Negative or zero");
```

Good

```
public Student GetStudentById(int id)
{
    return students.SingleOrDefault(s => s.Id == id);
}
```

```
if (x > 0)
{
    Console.WriteLine("Positive");
}
else
{
    Console.WriteLine("Negative or zero");
}
```

Braces - Conditions

- Specify the condition between round paranthesis ()

Evil

```
if x > 0
{
    Console.WriteLine("Positive");
}
```

Good

```
if (x > 0)
{
    Console.WriteLine("Positive");
}
```

Naming Conventions

Would you name your pet: dog1 or myDog?

Camel Case

- C# uses Camel Case for naming variables, methods, classes etc. and Microsoft recommends the same coding standards for developers that use C#
- Camel Case means that every different word in a name should start with the first letter capitalized. This makes reading easier for the brain, because the words are easier to identify and separate inside the long sequence of letters

Casing type	Upper Camel Case	Lower Camel Case
Used for naming	- Classes, Interfaces, Structs - Methods - Properties - Public Fields	- Private/protected fields - Local variables

Examples	`public void GetStudentById(int id)`
	`public class Student`
	`public string FirstName {get; private set;}`
		`private List<Student> students;`
		`string nextIndex;`

Good Names (1)

- The names you give to classes, variables, methods are essential for understanding code. Even if it takes some time to „baptize” properly a variable, this will eventually save you time in the future because it will be easier to understand what the variable is/does, just by taking a look at it.

- Let’s suppose you get a pet. What name would you give him?
  - Dog1
  - myDog
  - d

- Each name should represent what the variable/method/class does.

Good Names (2)

- Give good names instead of using comments to compensate for the bad code
- If the name of the variable would be too long or there is a catch, you should write a comment to explain more.

Evil

```
// elapsed time in days
int d;
```

Good

```
int daysSinceModification;
int fileAgeInDays;
```

Meaningful Names - Demo

Evil

```
public double Compute(List<int> a)
{
    double x = 0;
    int y = 0;
    foreach (int nr in a)
    {
        x += nr;
        y++;
    }
    if (y == 0)
    {
        return 0;
    }
    else
    {
        return x / y;
    }
}
```

Good

```
public double ComputeAverage(List<int> numbers)
{
    double sum = 0;
    int howManyNumbers = 0;

    foreach (int number in numbers)
    {
        sum += number;
        howManyNumbers++;
    }

    if (howManyNumbers == 0)
    {
        return 0;
    }
    else
    {
        return sum / howManyNumbers;
    }
}
```

Bad Names

- If the objects represent different things, then their names should reflect the difference
  - Don’t add prefixes like: the, my, a
  - Don’t add numeric suffixes
- Avoid too general names or prefixes

Evil

```
string theString;
string myString;
string aString;

string string1;
string string2;
string string3;
```

Evil

Utility
Common
...Manager
...Processor
...Info
...Data

Bad Names – Redundancies

- Avoid redundant prefixes/suffixes

Evil

```
string stringName;
decimal moneyAmount;
```

Good

```
string name;
decimal money;
```

Good Names – Prefixes & Suffixes

- Avoid prefixes/suffixes that don't help you make clear differences

Evil

```
class StudentInfo
class StudentData
```

Good

```
class StudentMedicalData
class StudentSchoolData
```

Evil

```
List<Account> theList
List<Account> aList
```

Good

```
List<Account> usedAccounts
List<Account> newAccounts
```

Good Names - Symmetry

- Use symmetrical naming for opposing states or prefixes

Evil	Good
`on/disable`	`on/off` `enable/disable`
`quick/slow`	`fast/slow`
`lock/open`	`lock/unlock` `close/open`
`slow/max`	`min/max`

Naming Advice

- Avoid very similar names: itemInList/itemsInList
- Don’t use lower „L” and „O” for naming variables
- Be consistent when giving names (use GetList all the times, not GetList sometimes and other times RetrieveList)
- Don’t be a cheapskate with names – give variable names longer than a letter, because each variable has a significance
- You can use „i” and „j” for iterating through a list, but it the iterator spreads over a few tens of lines, maybe it would be better to rename the „i” and „j” so that you don’t have to scroll back and see what each does.

Verbalizing Code

- Avoid names that cannot be pronounced.
- Avoid abbreviations
- It will be very difficult to explain to someone else what it does

Evil

```
class DtaRcrd102
{
   private DateTime genymdhms;
   private DateTime modymdhms;
   private String pszqint = "102";
}
```

Good

```
class Customer
{
  private DateTime generationTimestamp;
  private DateTime modificationTimestamp;
  private String recordId = "102";
}
```

Good Names in C#

- Avoid names that have underscore separators between the words
- Exception: the event handlers, that are generated automatically have controlName_eventName

Evil

```
int next_Index;
void get_student_by_id(int id)
```

Good

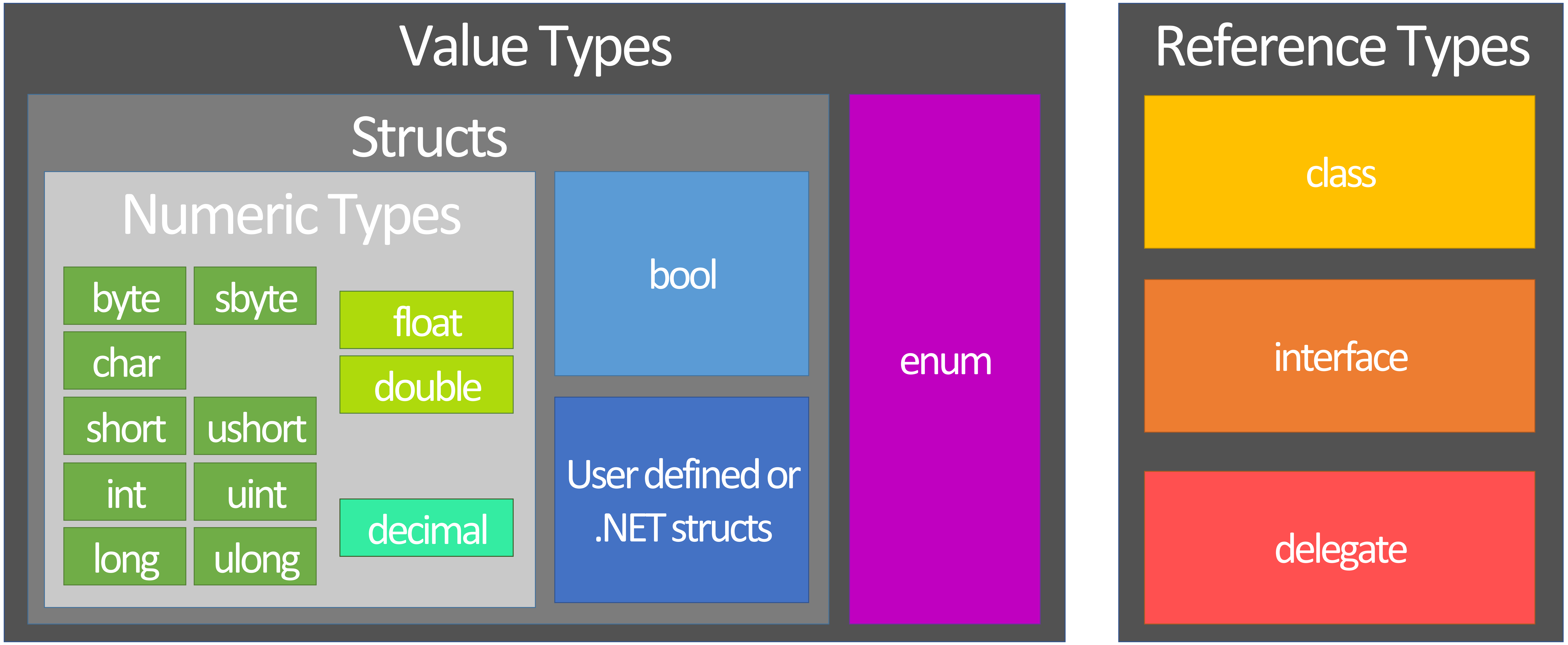
```
int nextIndex;
void GetStudentById(int id)

void btnSave_Click(object sender, RoutedEventArgs e)
```

Data Types

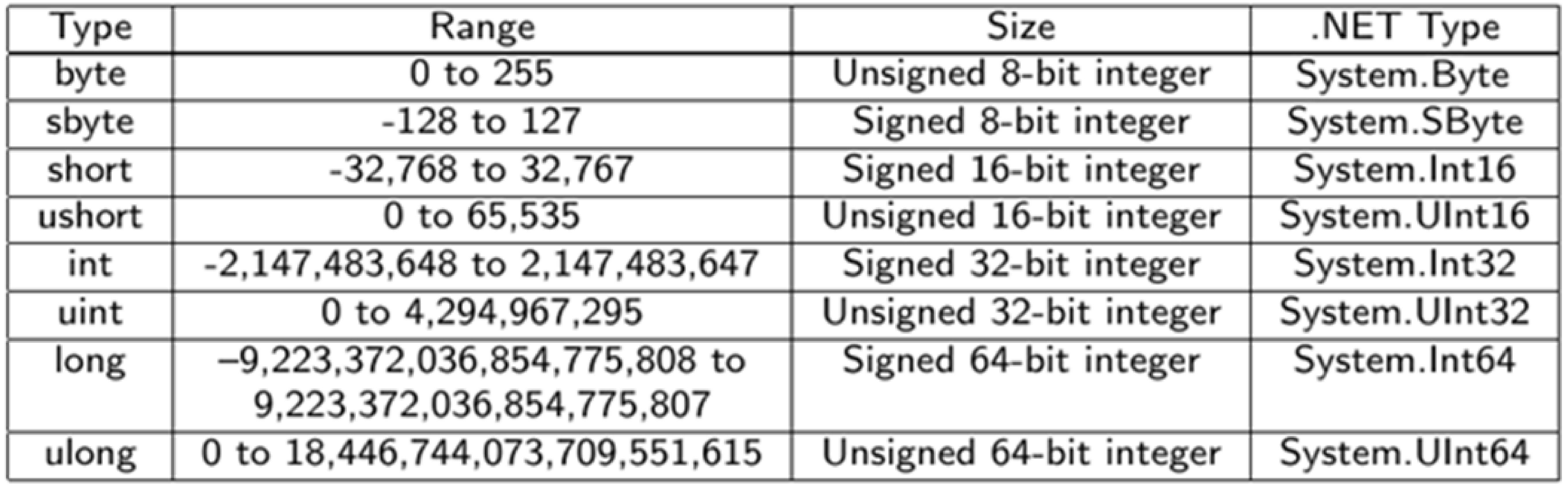
Don’t compare apples with pears. Just eat them, they are healthy ;)

Types of Variables in C#



Integral Variables

- Demo: TypesOfVariablesSample



Type	Range	Size	.NET Type
byte	0 to 255	Unsigned 8-bit integer	System.Byte
sbyte	-128 to 127	Signed 8-bit integer	System.SByte
short	-32,768 to 32,767	Signed 16-bit integer	System.Int16
ushort	0 to 65,535	Unsigned 16-bit integer	System.UInt16
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
uint	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64

```
byte byteNumber = 255;
sbyte sByteNumber = 127;
short shortNumber = 32767;
ushort uShortNumber = 65535;
int intNumber = 123;
uint uIntNumber = 4294967290;
long longNumber = 4294967296;
ulong uLongNumber = 9223372036854775808;
```

Char Variables

- Demo: TypesOfVariablesSample

Type	Range	Size	.NET Type
char	U+0000 to U+FFFF	Unicode 16-bit character	System.Char

```
// Character literal
char character = 'A';

// Hexadecimal
char charAsHexazecimal = '\x0041';

// Cast from integral type
char charFromIntCast = (char)65;

// Unicode
char charInUnicode = '\u0041';

Console.WriteLine(character + charAsHexazecimal + charFromIntCast + charInUnicode);
Console.WriteLine(character + " " + charAsHexazecimal + " "
                    + charFromIntCast + " " + charInUnicode);

int intValueOfCharacter = (int)character;
```

Floating Point Variables

- Demo: TypesOfVariablesSample

Type	Range	Size	Precision	.NET Type
float	-3.4×10^{38} to $+3.4 \times 10^{38}$	32 bits	7 digits	System.Single
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	64 bits	15-16 digits	System.Double
decimal	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28})/(100^{0to28})$	128 bits	28-29 digits	System.Decimal

```
// correct
float floatValueWithSuffix = 3.5F;
// incorrect
float floatValue = 3.5;

// correct
double doubleValue = 3;
double doubleNumber = 3D;

// correct
decimal decimalValueWithSuffix = 300.5m;
// incorrect
decimal decimalValue = 300.5;
```

Boolean Variables

- Demo: TypesOfVariablesSample

```
// correct
bool canVote = true;
bool hasGraduated = false;
```

```
// incorrect
bool isAdult = 1;
bool isMale = 0;
```

```
// correct
if (canVote)
{
    Console.WriteLine("I can vote");
}

if (!hasGraduated)
{
    Console.WriteLine("Still in school");
}

if (hasGraduated == false)
{
    Console.WriteLine("Still in school");
}
```

Struct

- Struct (structure) is a value type that is used to group (encapsulate) a small number of related variables
- It is similar to a class, but the class is a reference type and a struct is a value type

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}

private static void StructSample()
{
    Point3D pt = new Point3D();
    Console.WriteLine($"Point: x={pt.X} y={pt.Y} z={pt.Z}");
}
```

- It is similar to a class, but the class is a reference type and a struct is a value type
- Structs can implement interfaces, but cannot derive from other structs or classes
- The members of a struct are by default private

Value Types vs Reference Types

- More about parameters: ref and out parameters

Value types	Reference types
When assigning X to variable and X is of type value – the content of X is **copied** from the place where X is in memory where the variable is in memory	When assigning Y to variable and Y is of type reference – the content of Y is only **referenced**, not copied. The variable will point to the same place in memory where Y is
Sending parameters – makes a copy	Sending parameters – sends a reference (pointer)
Default value: type default value	Default value: null
Cannot be null	Can be null
struct	class

Value Types vs Reference Types - Assigning

- Demo: ValueVsReferenceSample

```
// x are y are value types
int x = 3;
int y = x;
x = x + 1;
Console.WriteLine($"x={x} and y={y}");
```

```
// Point is a struct, so it is a value type
Point p1 = new Point(10, 20);
Point p2 = p1;
p1.X = 30;
p1.Y = 40;
Console.WriteLine($"FirstPoint: x={p1.X} and y={p1.Y}");
Console.WriteLine($"SecondPoint: x={p2.X} and y={p2.Y}");
```

```
// Book is a class, so it's a reference type
Book book1 = new Book();
book1.Name = "A tale of two cities";
Book book2 = book1;
book1.Name = "Great expectations";
Console.WriteLine($"book1.Name = {book1.Name}");
Console.WriteLine($"book2.Name = {book2.Name}");
```

Value Types vs Reference Types - Parameters

- Demo: ValueVsReferenceSample

```
static void IncrementNumber(int number)
{
    number++;
}

private static void ValueParametersSample()
{
    int x = 32;
    Console.WriteLine("Before: x = {0}", x);
    IncrementNumber(x);
    Console.WriteLine("After: x = {0}", x);
}
```

```
static void SetBookName(Book book)
{
    book.Name = "Alice in wonderland";
}

private static void ReferenceParametersSample()
{
    Book book = new Book();
    Console.WriteLine($"Before: {book.Name}");

    SetBookName(book);
    Console.WriteLine($"After: {book.Name}");
}
```

Enum (1)

- Enum (enumeration) is a value type that contains a finite set of named constants.
- Enum names use UpperCamelCase
- Demo: EnumSample

```
public enum PlayerStates
{
    NotInitialized,
    MediaLoaded,
    Playing,
    Paused,
    Stopped
}
```

Enum (2)

- Each element (named constant) in the enum has an integer value associated with it.
- By default, the values start from 0 and increment by 1 in the definition order
- You can define different values from the default ones if you wish to, but each value should be unique inside the enum
- Demo: EnumSample

```
public enum PlayerStates
{
    NotInitialized, // implicit = 0
    MediaLoaded, // implicit = 1
    Playing, // implicit = 2
    Paused, // implicit = 3
    Stopped // implicit = 4
}
```

```
public enum CustomPlayerStates
{
    NotInitialized = 1,
    MediaLoaded = 2,
    Playing = 4,
    Paused = 8,
    Stopped = 16
}
```

Enum (3)

- Each value of the enum can be used like this: EnumName.ConstantInEnum
- Demo: EnumSample

```
public enum PlayerStates
{
    NotInitialized = 1,
    MediaLoaded = 2,
    Playing = 4,
    Paused = 8,
    Stopped = 16
}
```

```
private static void EnumSample()
{
    // initialization
    PlayerStates currentState = PlayerStates.NotInitialized;
    int numericValue = (int)currentState;

    // check current value
    if (currentState == PlayerStates.MediaLoaded)
    {
        // start player
    }
}
```

Nullable Types (1)

- All variables of value type cannot have null values
- But sometimes we will need a value type variable to be able to be null (not initialized for example) – in this case we can use nullable types
- Demo: NullableSample

```
// value type, not nullable, can store an integer value
int index = 0;

// incorrect: compile error because int is a value type
int anotherIndex = null;
```

Nullable Types (2)

- Use the “?” operator after the value type when declaring the variable

```
// correct, because int? is nullable. It can store either null or a value
int? nullableIndex = null;
nullableIndex = 2;

if (nullableIndex.HasValue)
{
    Console.WriteLine("NullableIndex=: " + nullableIndex.Value);
}
else
{
    Console.WriteLine("NullableIndex is null");
}

// 'Nullable<int>' is the same as 'int?'
Nullable<int> anotherNullableIndex = null;
```

Immutable Types

- Immutable variables are variables that cannot be changed after they were created
- Examples:
  - String
  - DateTime

Strings are Immutable

- Concatenating strings is not a very efficient operation, so it’s recommended to use StringBuilder instead, for multiple concatenations of strings

```
string s1 = "Hello";
string s2 = " ,";
string s3 = "World";

// compile error because strings are immutable
s3[0] = 'w';

// it doesn't change the value of the initial string, but creates
// a new area in memory where to store the new concatenated strings
s1 += s2;

// same as above (third area in memory)
s1 += s3;
```

Strings

- Demo: StringSamples

- Strings have powerful build-in functionalities:
  - string.Empty()
  - string.IsNullOrEmpty(value)
  - string.Format(format, value1, value2...)
  - Compare(string1, string2)
  - Replace(string1, string2)
  - IndexOf(substring), LastIndexOf(substring)
  - Concat(string1, string2)
  - StarsWith(substring), EndsWith(substring)
  - ToUpper(), ToLower()
  - Contains(substring), Substring(index, length)
  - Trim(), TrimStart(), TrimEnd()

- More: https://www.dotnetperls.com/string

DateTime

- Demo: DateTimeSamples
- Used to represent a moment in time (year, month, day, hour, minute, second, millisecond)
  - DateTime.Now
  - Year, Month, Day,
  - Hour, Minute, Second, Millisecond
  - Add, AddTicks, AddYears, AddDays, AddMonths, AddDays, AddMinutes, AddSeconds, AddMilliseconds, Subtract
  - ToShortDateString(), ToLongDateString(), ToShortTimeString(), ToLongTimeString()
- More: https://www.dotnetperls.com/datetime
- If you subtract 2 DateTimes, you will get a TimeSpan (interval)

Homework

- Create a solution with 2 projects, one for each homework:
- Read from the console 3 integers representing the year, month and day of a person and a letter (M/F) representing the gender of the person
  - Convert the 3 integers and create a DateTime for the birthdate
  - Compute the age of the person, based on the birthdate
  - Create an enum for the genders (Female, Male)
  - Using the input for gender, set the value of a nullable variable to one of the enum values or to null (if the user wrote an invalid character, other than M or F)
  - If there's a valid Gender, then display a message if the person is retired or at what age he/she will retire (Female at 63, Male at 65)
- Choose 7 methods from the string build-in functionalities and create some examples with them

Visual Studio Shortcuts

REFERENCE

- $F5 \rightarrow$ Run with Debug
- $Ctrl + F5 \rightarrow$ Run without Debug
- $Ctrl + B \rightarrow$ Build
- $F9 \rightarrow$ Adaugare/stergere de breakpoint
- $F10 \rightarrow$ In debugging, trecerea la următoarea linie de cod
- $F11 \rightarrow$ In debugging, intrarea in metoda de la linia de cod
- $Ctrl + Space \rightarrow$ Meniu intellisense
- $Alt + Shift + F10 \rightarrow$ Meniu "possible suggestions"
- $Ctrl + K + C \rightarrow$ Comment
- $Ctrl + K + U \rightarrow$ Uncomment
- $Ctrl + K + D \rightarrow$ Format Code
- $Ctrl + F \rightarrow$ Find in current file
- $Ctrl + Shift + F \rightarrow$ Find in files