.NET Framework and C# language

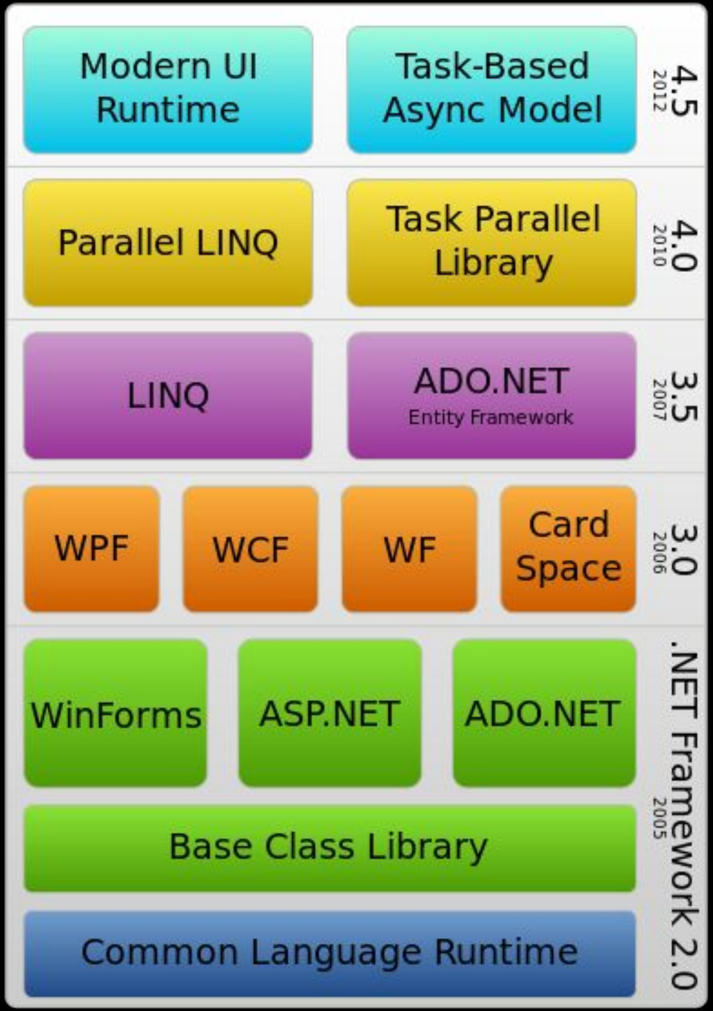
By Ira Zavushchak

softserve

Agenda

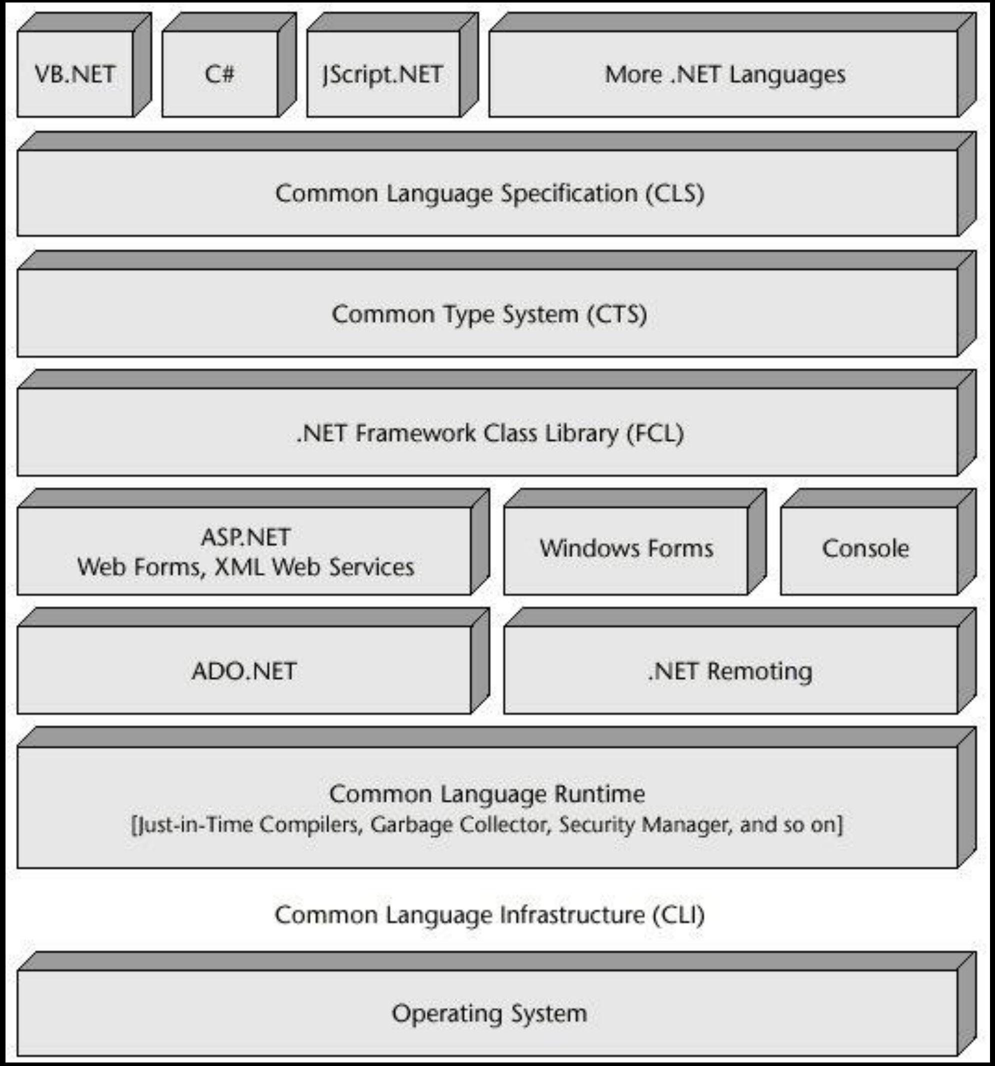
- .Net Framework
- Common Language Runtime
- C# - new .Net language
- Visual Studio. Demo
- C# First program. Demo
- Reading-Writing in Console

.NET Framework



- **.Net Framework** is software technology developed by Microsoft to create applications for Windows and Web applications.
- **.Net Framework** includes *Framework Class Library (FCL)* and provides language interoperability across several programming languages.
- Programs written for **.NET Framework** execute in a software environment - *Common Language Runtime (CLR),* an application virtual machine.

.NET Framework Architecture



VB.NET

C#

JScript.NET

More .NET Languages

Common Language Specification (CLS)

Common Type System (CTS)

.NET Framework Class Library (FCL)

ASP.NET
Web Forms, XML Web Services

Windows Forms

Console

ADO.NET

.NET Remoting

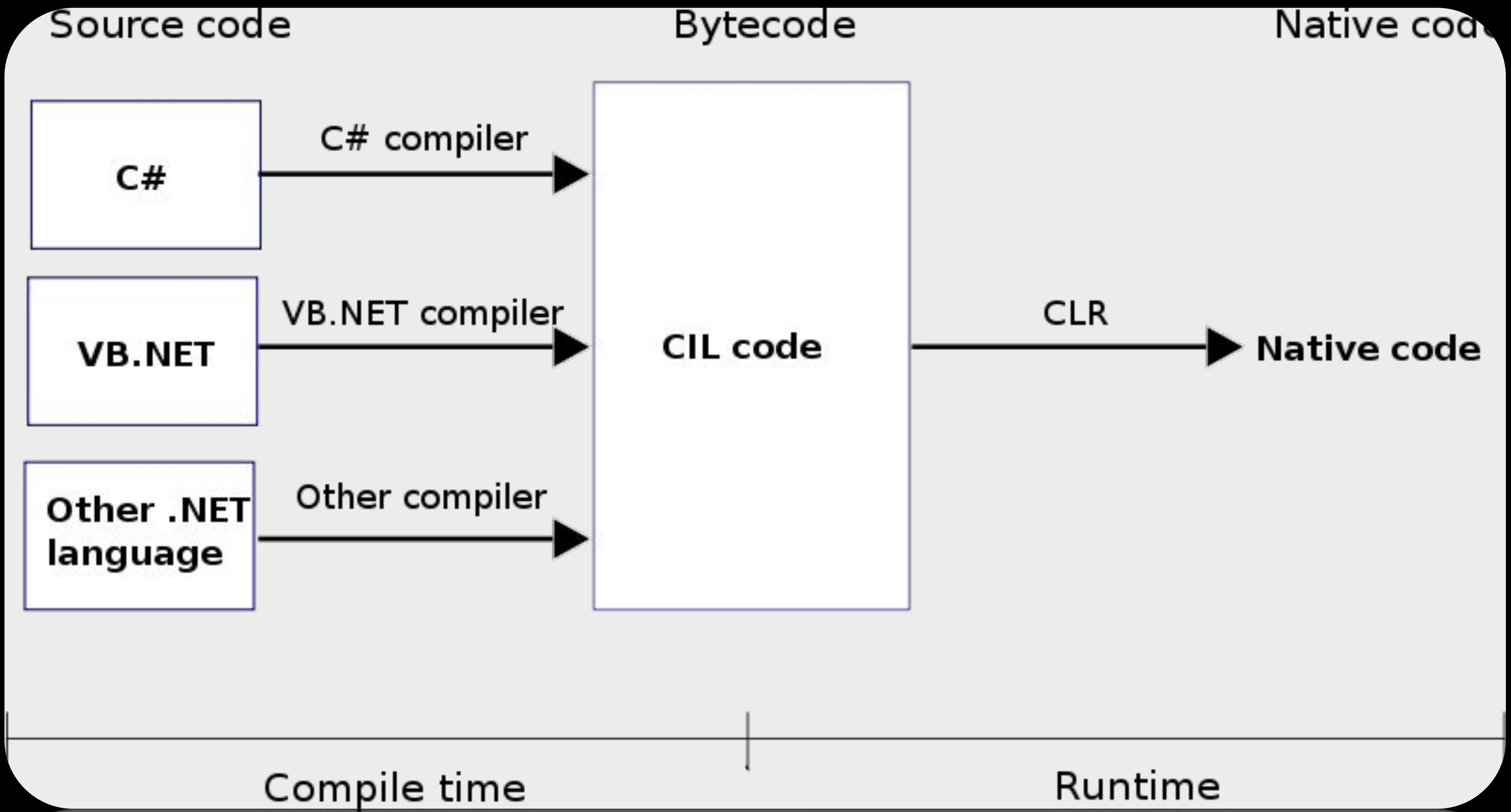
Common Language Runtime
[Just-in-Time Compilers, Garbage Collector, Security Manager, and so on]

Common Language Infrastructure (CLI)

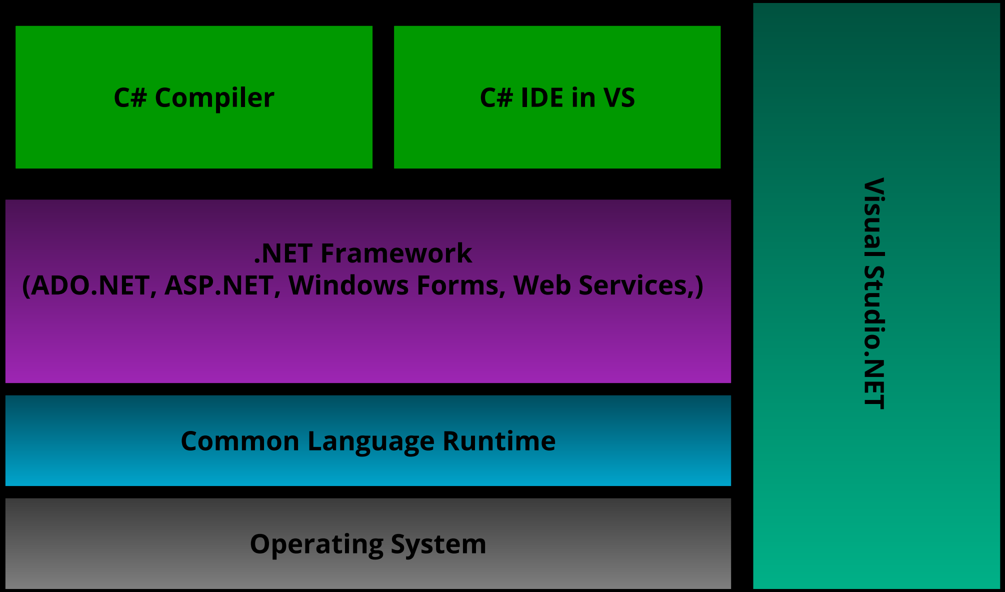
Operating System

- *Common Language Specification: (CLS)* are guidelines, that language should follow for communicating with other .NET languages in a seamless manner. (does mapping)
- *Common Type System (CTS):* is a base class library that contains all type information like *Int32, Int64, String , Boolean* etc.
- *Common Language Runtime (CLR):* is the execution engine for .NET applications and serves as the interface between .NET applications and the operating system.

CLR - Common Language Runtime



C# and Visual Studio .Net



C# Compiler

C# IDE in VS

.NET Framework
(ADO.NET, ASP.NET, Windows Forms, Web Services,)

Common Language Runtime

Operating System

Visual Studio.NET

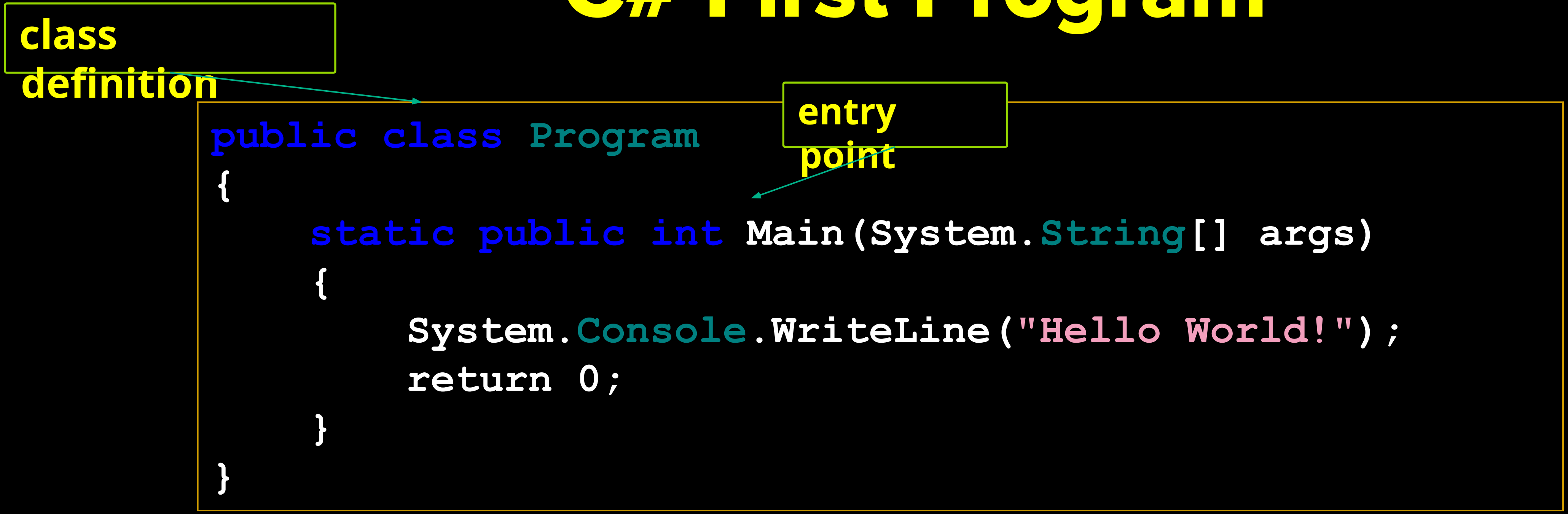
Visual Studio
Professional 2012
Microsoft Visual Studio 2005 Professional Edition
Visual Studio
Professional 2013
Microsoft Visual Studio 2008 Professional Edition
Microsoft Visual Studio 2010 Professional

Integrated development environment (IDE) is a collection of development tools exposed through a common user interface

C# Language

- C# is a new language designed by Microsoft to work with the .NET framework
- C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++.
- C# provides support for software engineering principles:
  - ✓ strong type checking,
  - ✓ array bounds checking,
  - ✓ detection of attempts to use uninitialized variables,
  - ✓ automatic garbage collection.

C# First Program



class definition

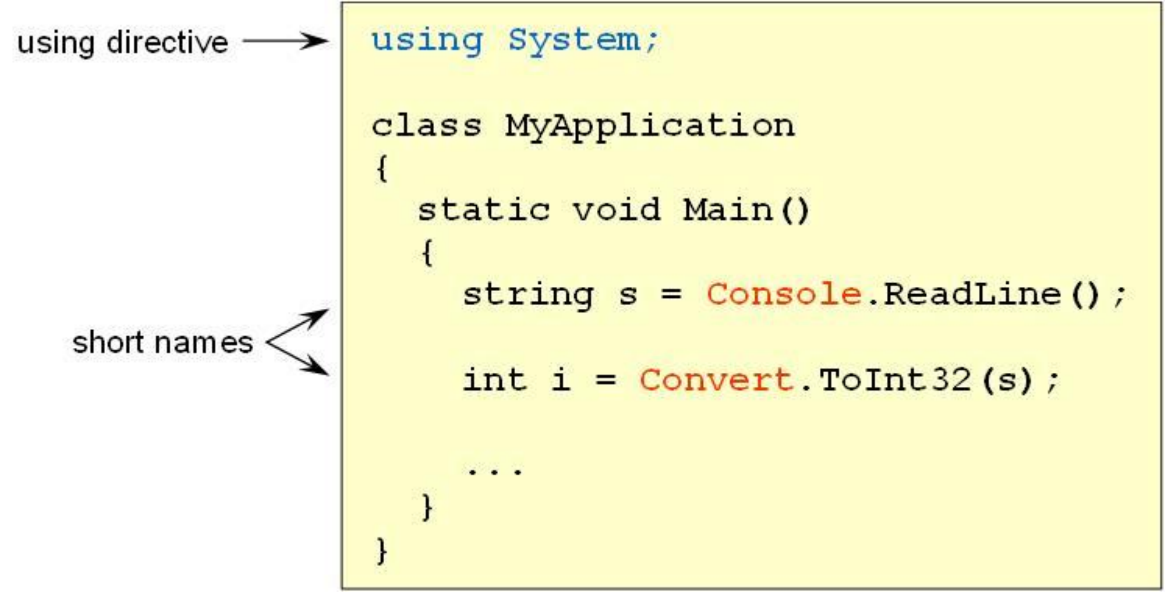
entry point

```
public class Program
{
    static public int Main(System.String[] args)
    {
        System.Console.WriteLine("Hello World!");
        return 0;
    }
}
```

- ✔ `class` is used to define new types.
- ✔ C# code should be put in some class.
- ✔ Method Main() is an entry point of program

Namespaces and using directive

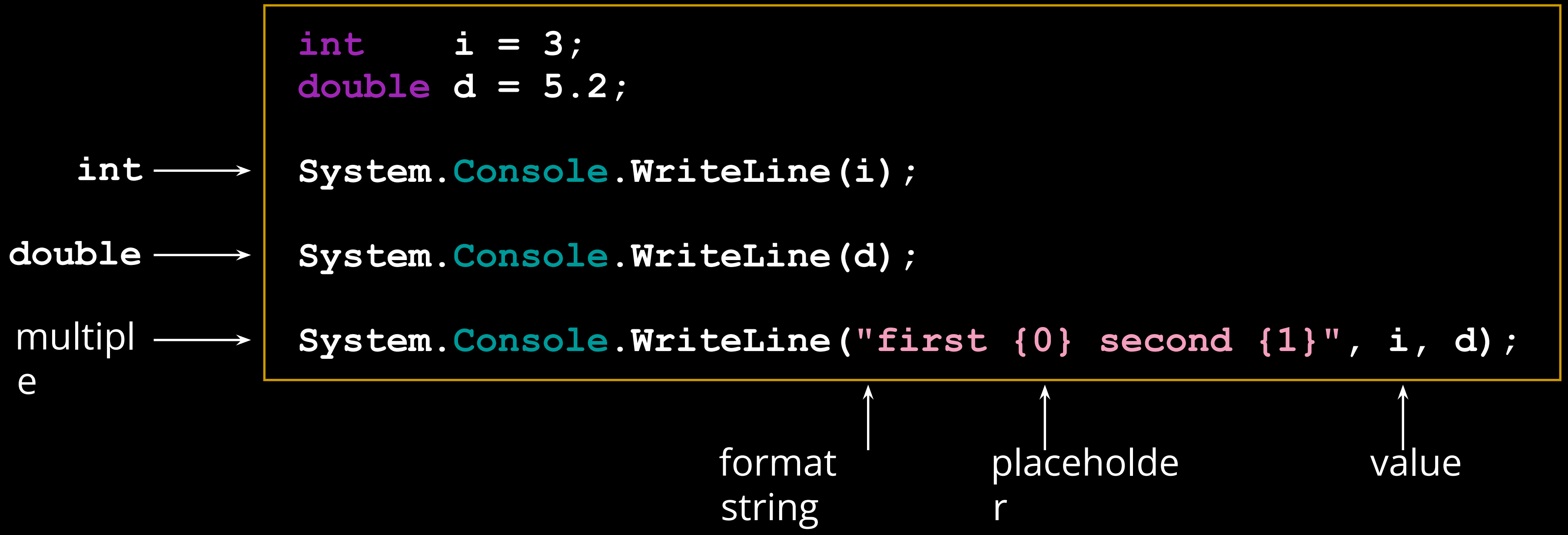
- .NET Framework classes use *namespaces* to organize its many classes.
- Declaring *own namespaces* can help control the scope of class and method names in larger programming projects.
- *Section of **using** directives* lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time that a method that is contained within is used.



```
using System;

class MyApplication
{
  static void Main()
  {
    string s = Console.ReadLine();

    int i = Convert.ToInt32(s);

    ...
  }
}
```

Writing into Console

- **Console .Write()** and **Console .WriteLine()** put line of text (string) into the stream for writing on Console.
- For non-string values **ToString()** method is invoked



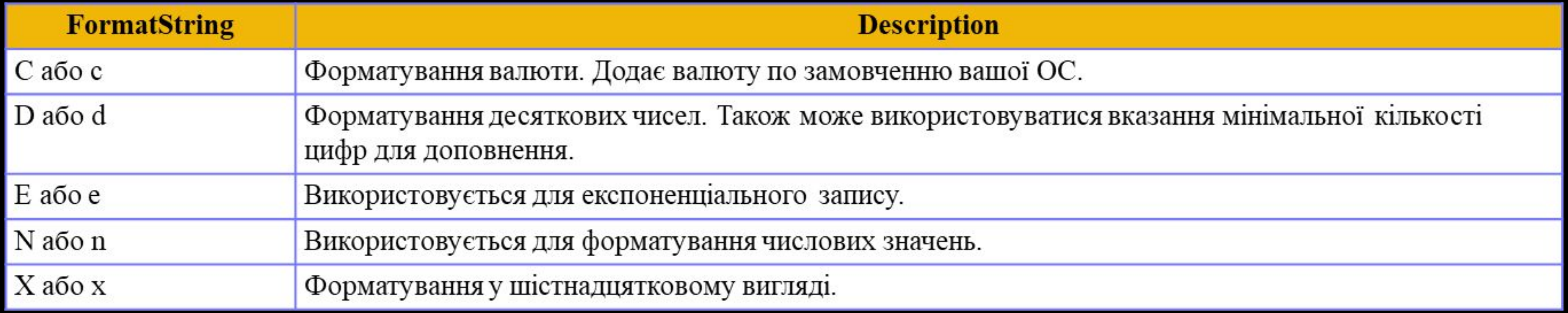
```
int    i = 3;
double d = 5.2;

System.Console.WriteLine(i);

System.Console.WriteLine(d);

System.Console.WriteLine("first {0} second {1}", i, d);
```

Format output

The format item:

{ index [:formatString] }

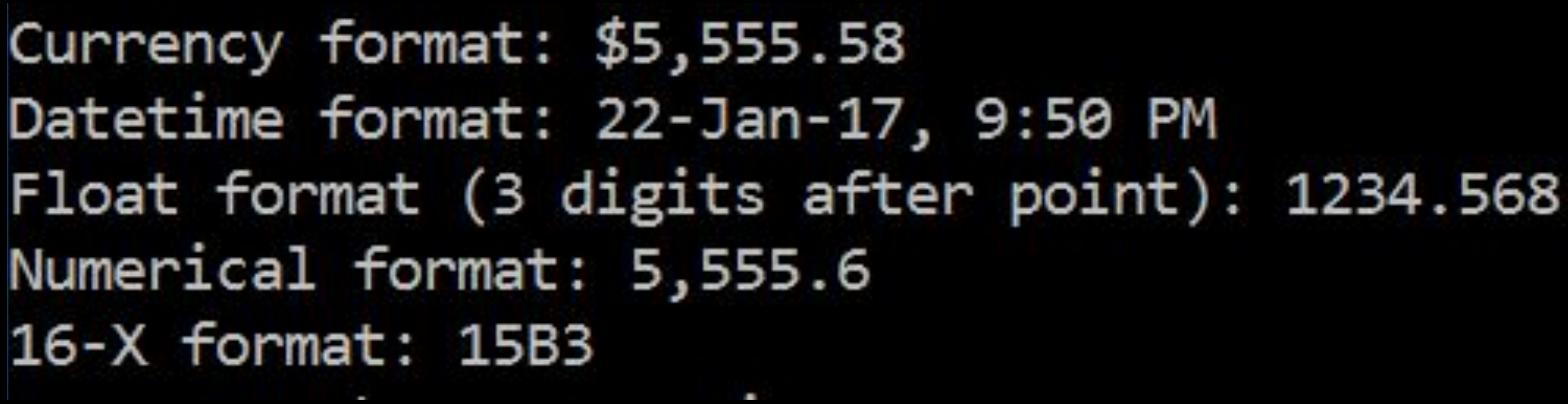
- ✔**Index**: The zero-based index of the argument whose string representation is to be included at this position in the string.
- ✔**formatString**: A string that specifies the format of the corresponding argument's result string.



FormatString	Description
C або c	Форматування валюти. Додає валюту по замовченню вашої ОС.
D або d	Форматування десяткових чисел. Також може використовуватися вказання мінімальної кількості цифр для доповнення.
E або e	Використовується для експоненціального запису.
N або n	Використовується для форматування числових значень.
X або x	Форматування у шістнадцятковому вигляді.

Format output

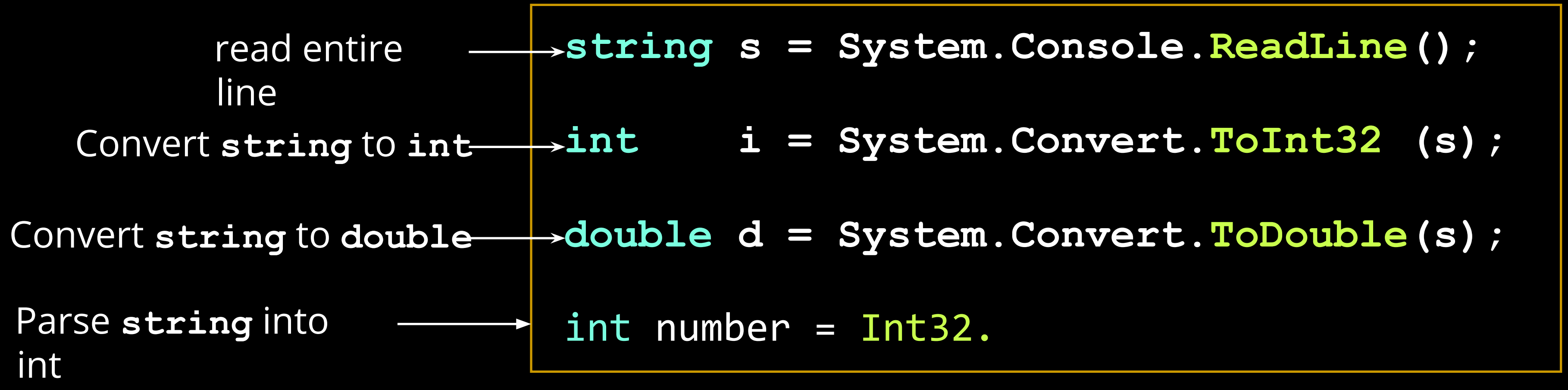
```
Console.WriteLine("Currency format: {0:C}", 5555.5812);
Console.WriteLine("Datetime format: {0:d}, {0:t}",  DateTime.Now);
Console.WriteLine("Float format (3 digits after point): {0:F3}", 1234.56789);
Console.WriteLine("Numerical format: {0:N1}", 5555.5812);
Console.WriteLine("16-X format: {0:X}", 5555);
```



```
Currency format: $5,555.58
Datetime format: 22-Jan-17, 9:50 PM
Float format (3 digits after point): 1234.568
Numerical format: 5,555.6
16-X format: 15B3
```

Reading from Console

- `Console.ReadLine()` - reads line from console and return it as string type
- Use methods from `System.Convert` class for converting string variable to other types
- Or use **Parse**() methods from different system types



read entire line

Convert **string** to **int**

Convert **string** to **double**

Parse **string** into int

```
string s = System.Console.ReadLine();
int    i = System.Convert.ToInt32 (s);
double d = System.Convert.ToDouble(s);
int number = Int32.
```

Reading from Console

- Use **TryParse()** to avoid format exceptions

```
static bool TryParse(string s, out Int32 result);
```

```
string s = Console.ReadLine();
int number ;
bool rez = Int32 TryParse(s, out number);
Console WriteLine("{0}-{1}", rez, number);
```

Program Structure and Code Conventions

C# Coding Standards
and Best Programming
Practices

softserve

Introduction

- The goal of this lecture is to provide a **standard coding technique** for C#. Net projects hold by the members of MS Solutions team.

- The techniques defined here are not proposed to form an inflexible set of coding standards. They are rather meant to serve as a guide for developing a coding standard for a specific software project.

Agenda

- General rules
- File Organization
- Namespaces.Classes. Interfaces.
- Methods. Properties. Fields. Local Variables
- Events and Delegates
- Enum Naming Guidelines
- Comments
- Exception Handling
- Format. Case study

General rules

1.1. General rules

- “A name should tell **‘what’** rather then **‘how’.**
- Long enough to be **meaningful** - short enough to avoid verbosity.
- Must be **comprehensible** by reader .
- **Avoid redundant** class names while naming properties and methods

  `List.ListItem` should be named `List.Item`

- Fully usable from **both case-sensitive and case-insensitive** languages. Don’t use names that differ only by case.
- **Avoid** using class names that **duplicate .NET Framework namespaces**: `System`, `Collections`, `Forms`, `UI`, etc.

General rules

1.2. Capitalization Styles:

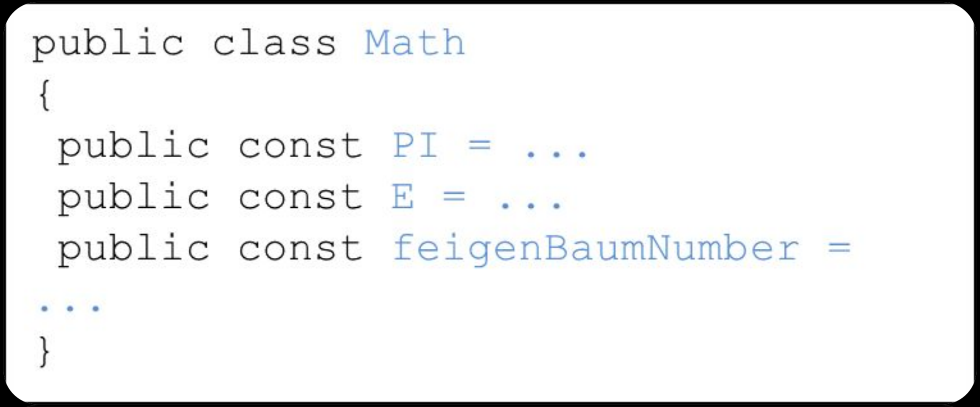
Pascal Casing - capitalize the first character of each word

`TestCounter, Item, GroupName`

Camel Casing - capitalize the first character of each word except the first one.

`testCounter, name, firstName`

Upper case - only use all upper case for identifier-abbreviation of 1 or 2 characters. Identifiers of more then 3 characters should use Pascal Casing instead.

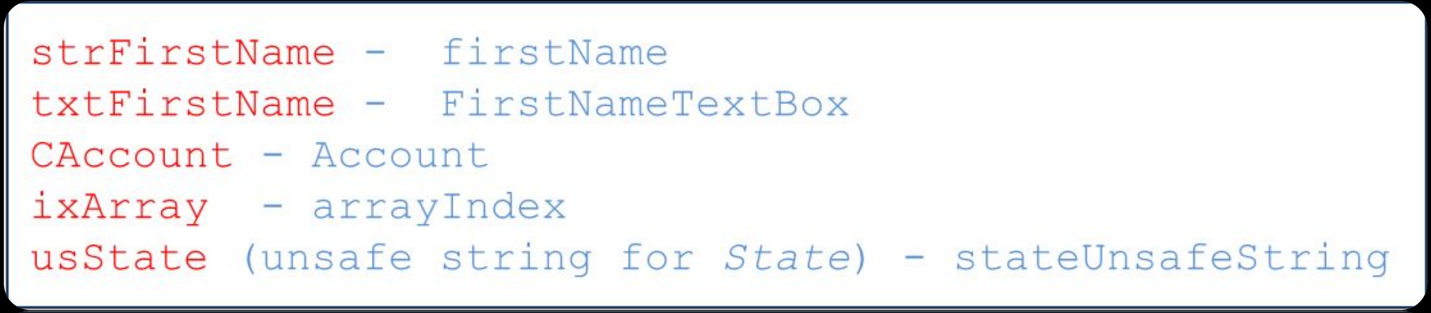


```
public class Math
{
 public const PI = ...
 public const E = ...
 public const feigenBaumNumber =
...
}
```

General rules

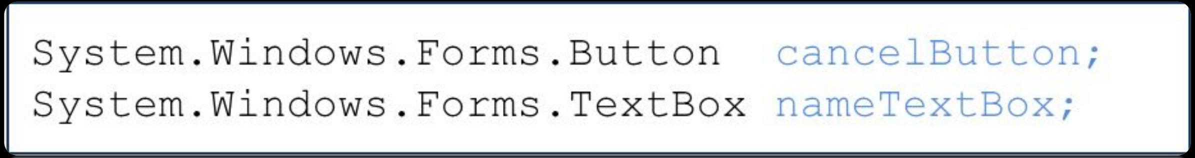
1.3. Hungarian notation

Is a defined set of pre and postfixes to names to reflect the type of the variable. Using Hungarian notation is not allowed.



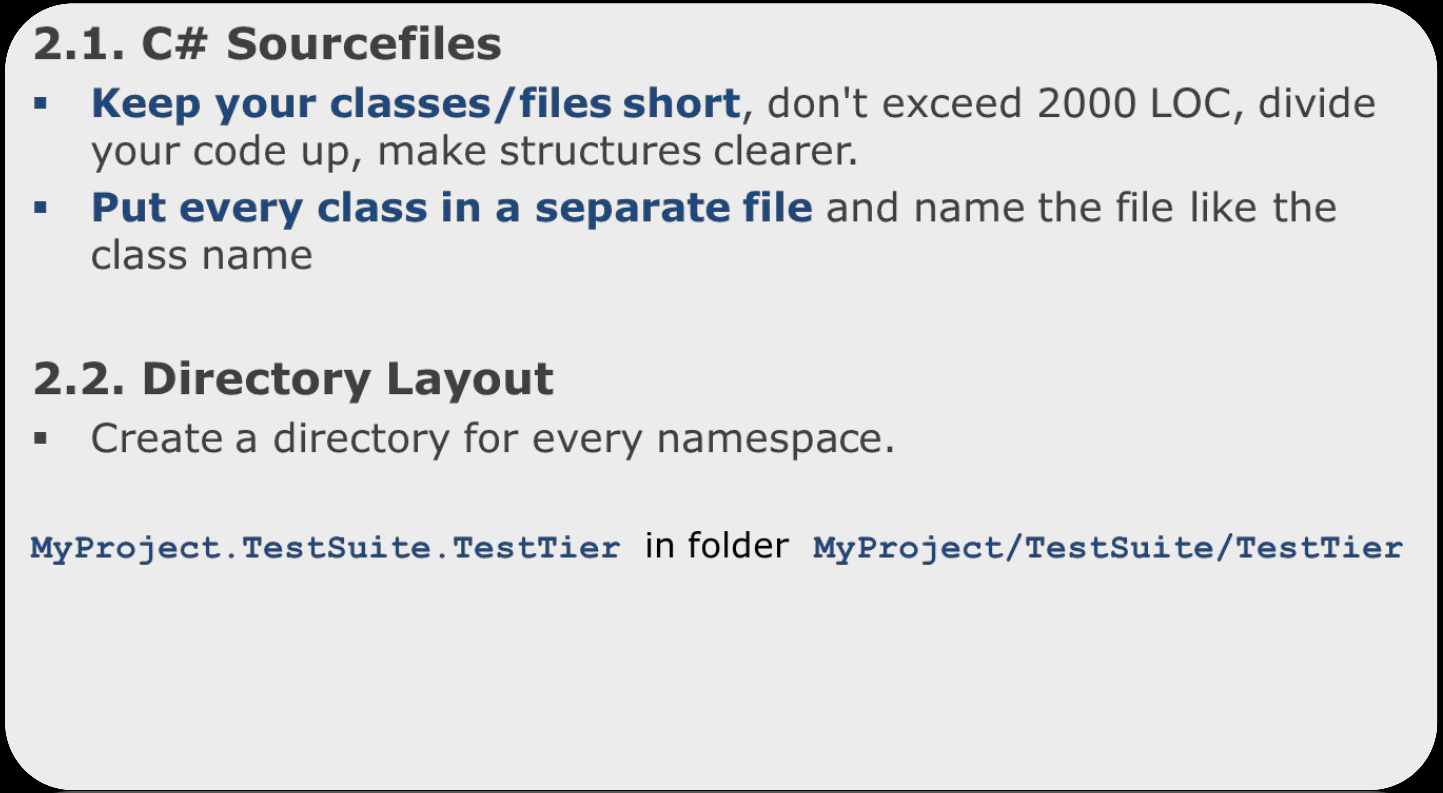
```
strFirstName -  firstName
txtFirstName -  FirstNameTextBox
CAccount - Account
ixArray  - arrayIndex
usState (unsafe string for State) - stateUnsafeString
```

An exception to this rule is GUI code:



```
System.Windows.Forms.Button  cancelButton;
System.Windows.Forms.TextBox nameTextBox;
```

File Organization



2.1. C# Sourcefiles

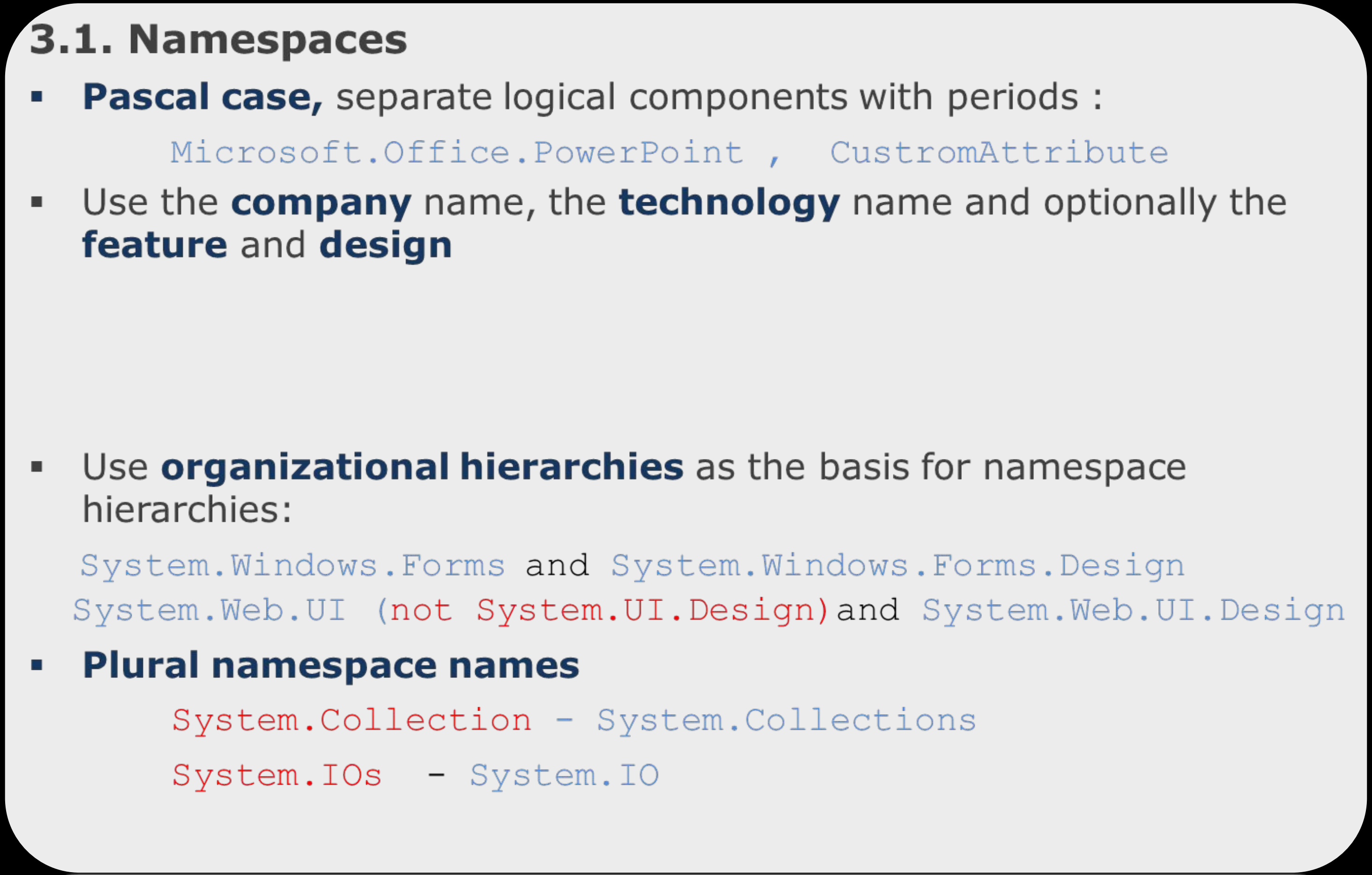
- **Keep your classes/files short**, don't exceed 2000 LOC, divide your code up, make structures clearer.
- **Put every class in a separate file** and name the file like the class name

2.2. Directory Layout

- Create a directory for every namespace.

`MyProject.TestSuite.TestTier` in folder `MyProject/TestSuite/TestTier`

Namespaces



3.1. Namespaces

- **Pascal case,** separate logical components with periods :

  `Microsoft.Office.PowerPoint ,  CustromAttribute`

- Use the **company** name, the **technology** name and optionally the **feature** and **design**

- Use **organizational hierarchies** as the basis for namespace hierarchies:

  `System.Windows.Forms` and `System.Windows.Forms.Design`
  `System.Web.UI (not System.UI.Design)` and `System.Web.UI.Design`

- **Plural namespace names**

  `System.Collection – System.Collections`

  `System.IOs  – System.IO`

Classes names

3.2. Class

- Class names must be **nouns** or noun phrases.
- Use **Pascal Casing**
- Do **not** use the same name for a **namespace** and a class
- Do **not** use any class **prefix**

```
CFileStream _fileStream - FileStream
```

Interfaces names

3.3. Interfaces

- **Nouns,** concatenated nouns or **adjectives** that describe behavior:

```
IComponent,
ICustomAttributeProvider,
IPersistable
```

- Use **I** as **prefix** for the name
- Use **Pascal Casing**

Methods names

3.4. Methods

- Name methods with **verbs** or **verb phrases**
- Use **Pascal Casing** for **public** and **protected** methods
- Use **Camel Casing** for **private** methods:

```
public void CalculateTotal();
  private int getAttribute()
```

- Don’t use names with subjective interpretation:

```
OpenThis()
```

- Method bodies - not more than **25 - 50** lines of code.
  Use private functions to break down the business logic into sub-modules.

Methods. Best practices

Make the **method name obvious**

Good:

```
public void SavePhoneNumber ( string phoneNumber )
    {
            // Save the phone number.
    }
```

Not good:

```
// This method will save the phone number.
void SaveData ( string phoneNumber )
    {
            // Save the phone number.
    }
```

Methods. Best practices

- A method **should do only** "**one job**".

Good:

```
// Save the address.
        public void SaveAddress (
   string address )
    {
          ...
    }
// Send an email to the
   supervisor to inform that the
   address is
// updated.
     public void SendEmail ( string
   email )
    {
         ...
    }
```

Not good:

```
 // Save address and send an
email to the supervisor
 // to inform that the address is
updated.

   SaveAddress ( address, email );
   void SaveAddress ( string
address, string email )
   {
 // Job 1. Save the address.
 // Job 2. Send an email to
inform the supervisor
   }
```

Fields names

3.5. Fields

- Name fields with nouns, noun phrases or abbreviations for nouns
- Use **Camel Casing**
- Do **not** use **public fields**.

```
private int jobId;
```

- Boolean fields (properties, variables, parameters) – have to start with prefix “**is**”, “**has**” or “**does**” :

```
boolean doesFileExist – fileExists
boolean isOpen - open
```

Properties names

3.6. Properties

- Name properties using **nouns** or **noun phrases**
- Use **Pascal Casing**
- Name properties with the same name as appropriated field

```
private int jobId;
public int JobId {get;set;}
```

- Write readonly property – for forbidding changes in the property's data by user.
- Do **not** use **write-only** properties.

Local variables

3.7. Local variables and parameters

- Use **Camel casing**
- Even for short-lived local variables **use a meaningful name**.
- Exceptions: `i, j, k, l, m, n - for loops variables;`

```
x, y, z - for coordinates;
r, g, b - for colors;
e - for event argument.
```

- Avoid **magic numbers**: named constants in conditions instead of numbers (exceptions: 0, 1, –1):

`for(i=0; i<NUM_DAYS_IN_WEEK; i++)` instead of `for(i=0; i<7; i++);`

Local variables

- **Avoid** using hard coded **strings** for messages that are displayed to user. **Use** a **named constant**, a **database record** or **resource file item** instead.
- Use **formatted strings** instead building strings for custom messages :

```
MES_DELETE = "File {0} deleted.";
...
res = String.Format(MES_DELETE, drawFile.Name);
```

Enum

3.9. Enum

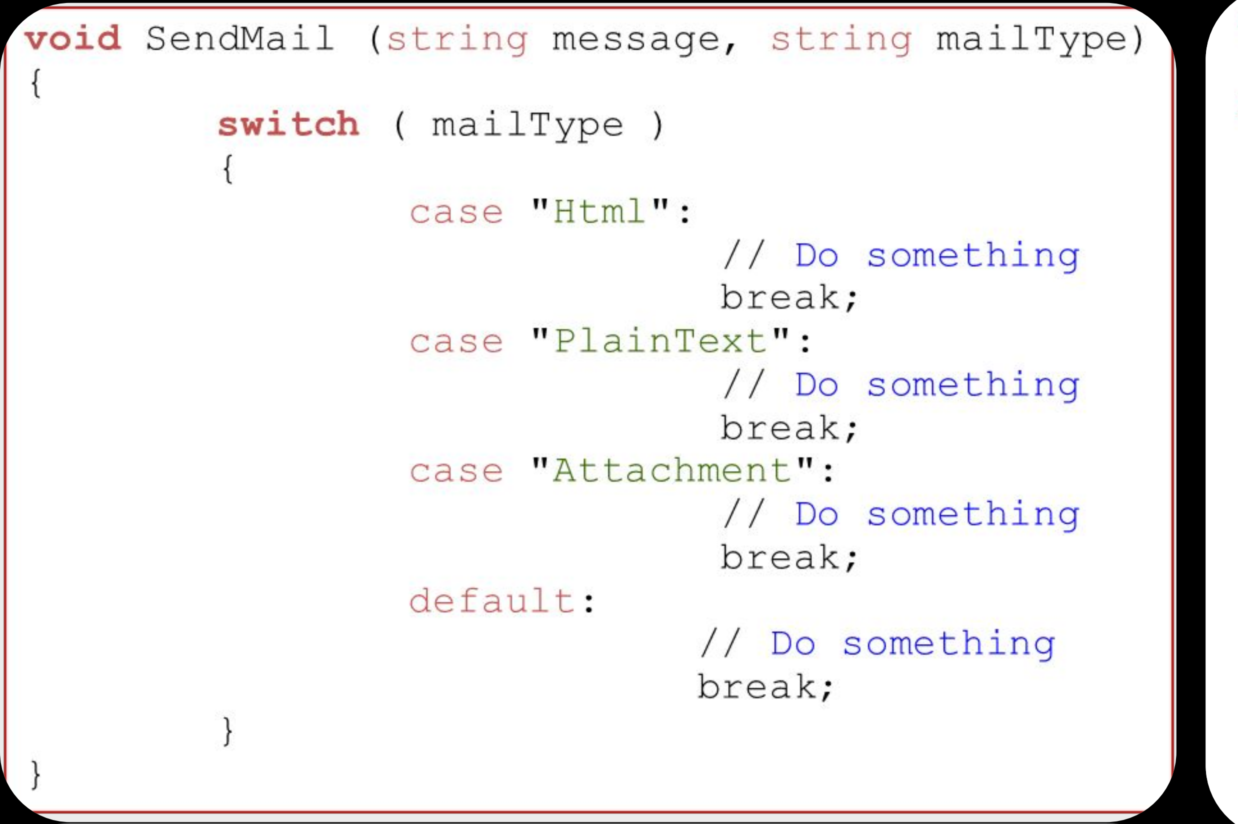
- Use **Pascal Casing** for enum value names and enum type names
- **Don’t prefix (or suffix)** enum type or enum values
- Use singular names for enums
- Use plural name for bit fields.

```
public enum StatusMode
   {
     Planned  = 1,
     Active   = 2,
     InActive = 4,
     All      = 7
   };
```

Enum

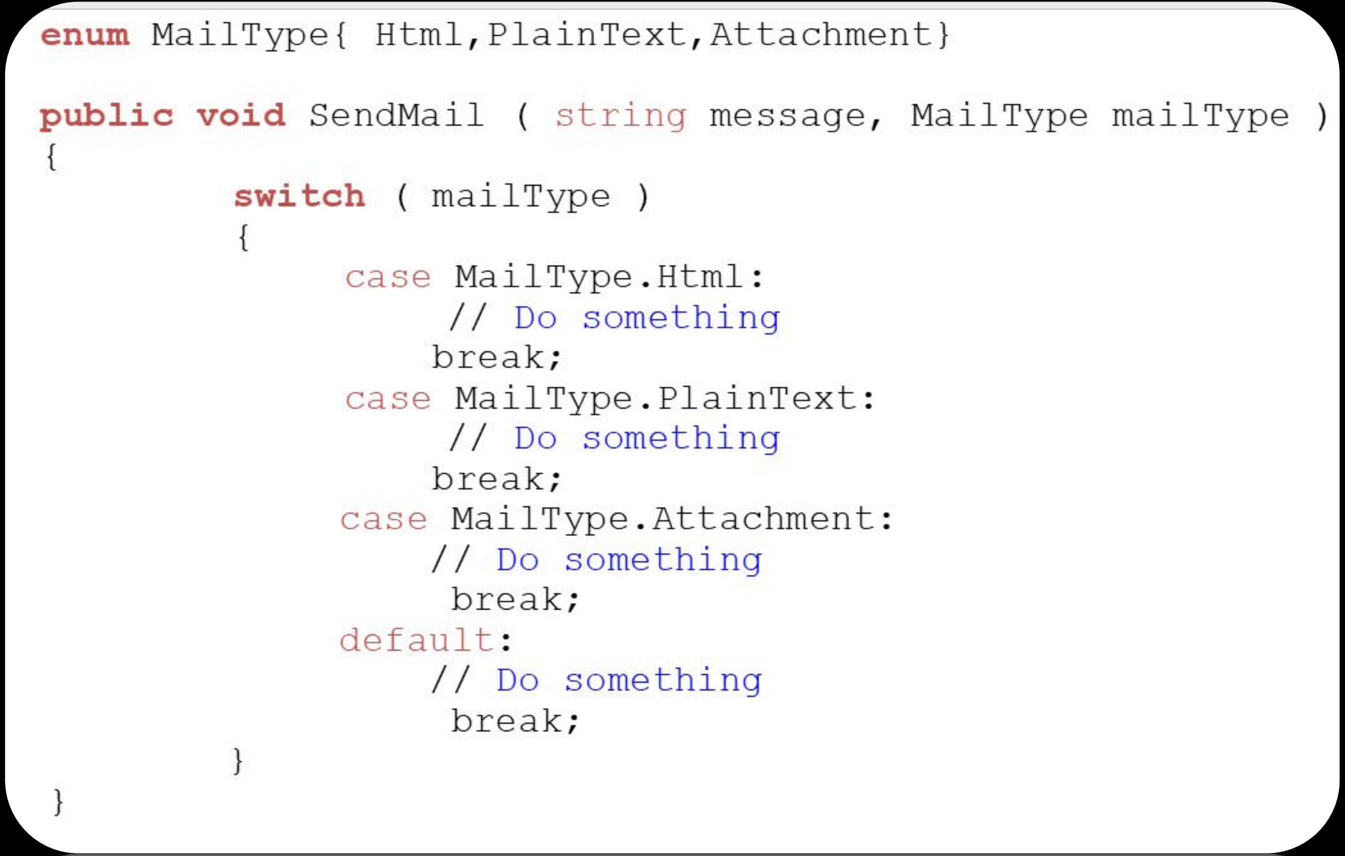
Use enum instead using numbers or strings to indicate discrete values.

Not good



```
void SendMail (string message, string mailType)
{
        switch ( mailType )
        {
                case "Html":
                                // Do something
                                break;
                case "PlainText":
                                // Do something
                                break;
                case "Attachment":
                                // Do something
                                break;
                default:
                               // Do something
                               break;
        }
}
```

Good:



```
enum MailType{ Html,PlainText,Attachment}

public void SendMail ( string message, MailType mailType )
{
        switch ( mailType )
        {
             case MailType.Html:
                 // Do something
                break;
             case MailType.PlainText:
                 // Do something
                break;
             case MailType.Attachment:
                // Do something
                 break;
             default:
                // Do something
                 break;
        }
 }
```

Comments

4.1. Single Line Comments

- Use **complete sentences** when writing comments.
- Comments should be quite informative and understandable by other people

```
int level; // indentation level
int size; // size of table
```

- Always keeps the commenting **up to date** (**actual**).
- **Avoid** adding comments at the **end of a line** of code (except local variable declarations)
- Use comments on **important loops** and **logic branches**.
- Comment all private field declarations (//).
- **Block comments** should usually be **avoided**.

```
/* Line 1
 * Line 2
 * Line 3
 */
```

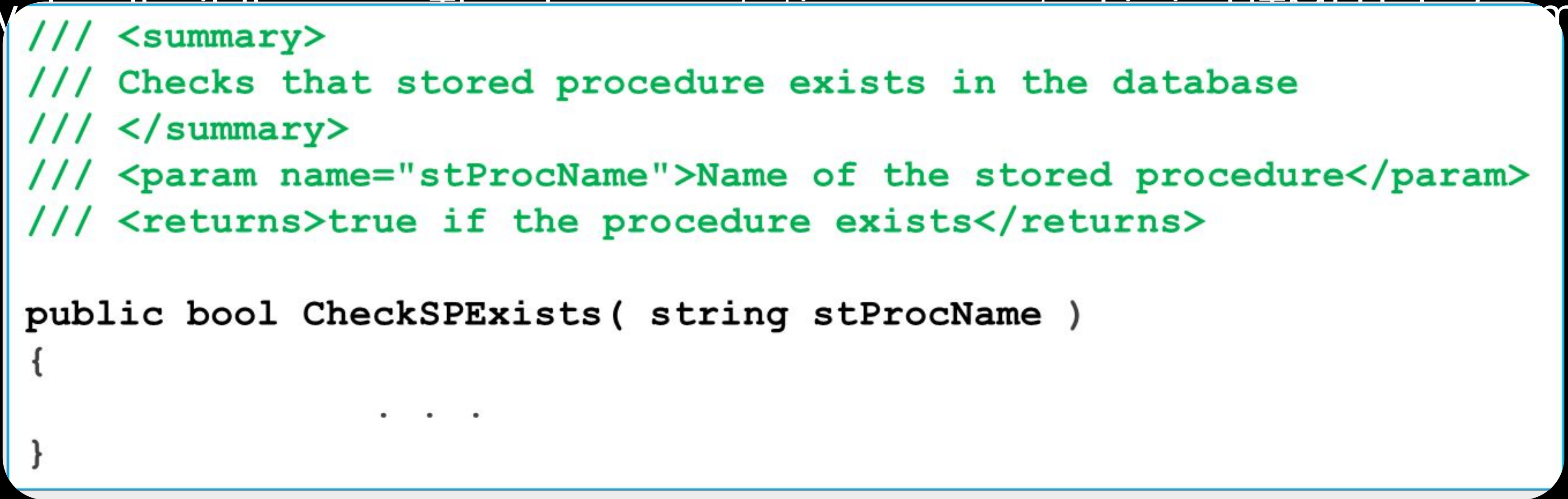
Comments

4.2. XML Documentation

In the .net framework is a **documentation generation system based on XML comments**.

At the beginning of every **construction part of code** (class, method, property, function or protected field declaration, etc.) use “<summary>” XML commenting tag (type “///” for automatically generation)

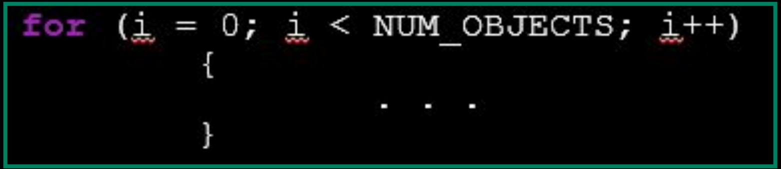
- Provide descriptions of **parameters** and **return value** of methods and functions in the corresponding tags. Documentation can be generated using the 'documentation' item in the #dev mat



```
/// <summary>
/// Checks that stored procedure exists in the database
/// </summary>
/// <param name="stProcName">Name of the stored procedure</param>
/// <returns>true if the procedure exists</returns>

public bool CheckSPExists( string stProcName )
{
                . . .
}
```

Format

- Establish and use a **standard size** for an indent through the project.
- Default indent - tab size (**4 space** characters).
- Line of code - **less than 80 characters**
- Align **open and close braces** vertically :



```
for (i = 0; i < NUM_OBJECTS; i++)
      {
                    . . .
      }
```

```
for (..) {
     . . .
}
```

- **Indent code** along lines of logical construction:

```
if (reportId != BaseTable.INVALID_PK)
{
       try
      {
          recReport = RepManager.GetRecordByPK(reportId);
      }
      catch (Exception ex)
      {
          HandleException(ex);
      }
    }
  else
  {
      recReport = new RecReports();
  }
```

softserve

Format

- Break **long statement** it to several lines and use **double** indenting in next lines.

```
if (Member.Address.Room != null && Member.Address.Room != "" &&
        (Member.Address.Sect > 0 || Member.Address.BuildNo > 0))
      Member.Address.Normalize();
```

Format

- **Break long statement** with logical code structure.

Wrong formatting:

```
if (Address.Room != null && Address.Room != "" && (Address.Sect
        > 0 || ((Address.BuildNo != null && Address.BuildNo !=
        "")?Address.BuildNo:DEFAULT_BUILDING_NO) > 0) &&
        Address.IsNotPrepared)
         Member.Address.Normalize();
```

- **Correct**

```
if (Address.Room != null && Address.Room != "" &&
    (Address.Sect > 0 ||
    ((Address.BuildNo != null && Address.BuildNo != "")?
    Address.BuildNo:DEFAULT_BUILDING_NO) > 0) &&
    Address.IsNotPrepared)
    Member.Address.Normalize();
```

Format

- **Good**

```
if ( ... )
{
    // Do something
    . . .
}
```

- **Not good**

```
if ( ... ) {
    // Do something
    . . .
}
```

Use a single space before and after each operator and brackets.

Good:

```
if

        for   int
```

Not good:

```
if

        for int
```

Task 1

Create Console Application project in VS.

In method Main() write code for solving next tasks:

1. Define integer variables **a** and **b**.Read values a and b from Console and calculate: a+b, a-b, a*b, a/b. Output obtained results.
2. Output question “How are you?“. Define string variable **answer**. Read the value **answer** and output: “You are (answer)".
3. Read 3 variables of char type. Write message: “You enter (first char), (second char), (3 char)”
4. Enter 2 integer numbers. Check if they are both positive – use bool expretion

Homework 1

1. Practical task:

Create Console Application project in VS. In method Main() write code for solving next tasks:

a.define integer variable a. Read the value of a from console and calculate area and perimetr of square with length a. Output obtained results.

b.define string variable name and integer value age. Output question "What is your name?";Read the value name and output next question: "How old are you,(name)?". Read age and write whole information

c.Read double number **r** and calculate the length (l=2*pi*r), area (S=pi*r*r) and volume (4/3*pi*r*r*r) of a circle of given r

2. Learn next C# topics:

a)reference and value types

b) intrinsic Data Types

c) C# operators: if, switch, loop statements