# Advanced Programming 2
# Recitation 1 – Introduction to C#

Roi Yehoshua
2017

# Goals

- Desktop applications (C#, WPF)
- Web development (Java servers)
- Mobile applications (Android)
- Working with databases and ORMs
- Advanced techniques and tools
- Our project includes all the important elements used in the industry
- Also – self learning, and self googling…



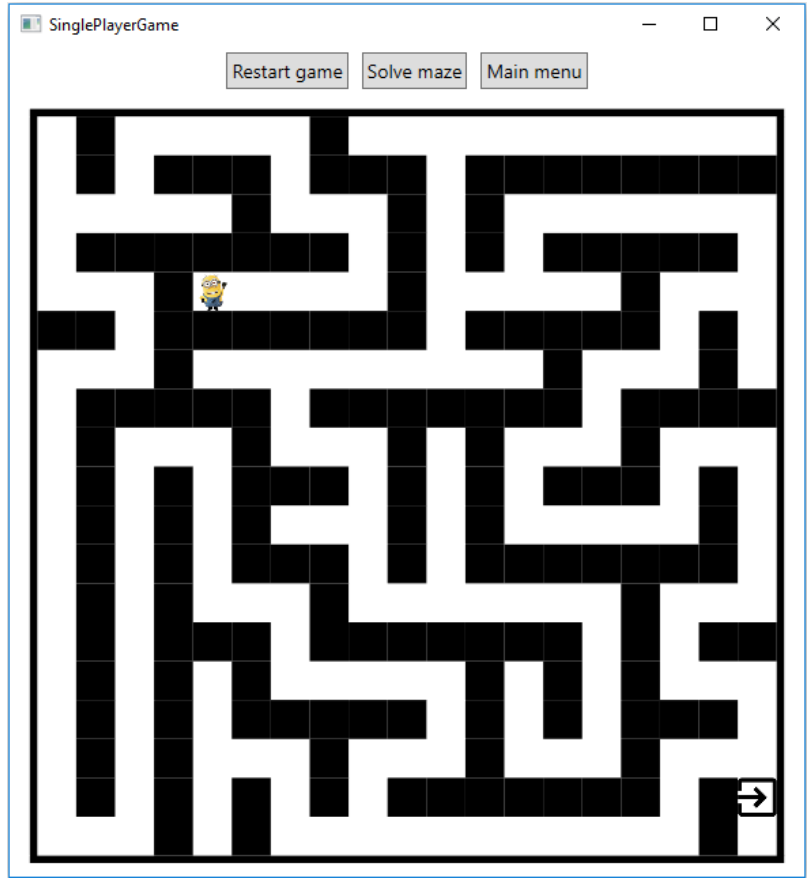


# Administration

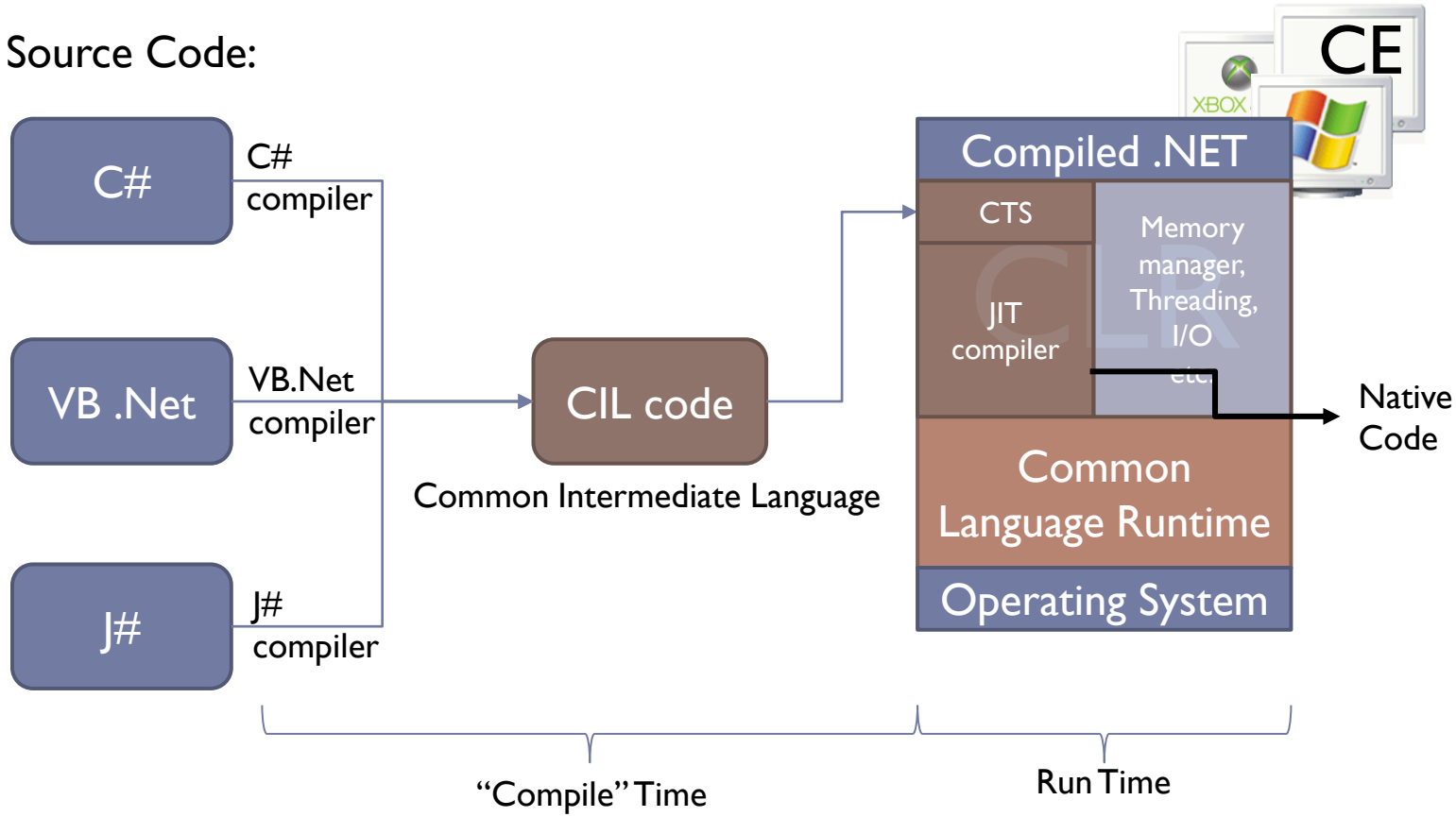
- **My mail:** roiyeho@gmail.com
  - Please do not send me emails about “how to implement”, “my code doesn’t work”, “what is the problem?”, etc.
- **Office hours:** Sunday 14:00
  - Building 202, Room 106 (robotics lab)
- **Course website:** Moodle
  - Visit the site regularly and make sure you’re subscribed to receive notifications
  - Submission of assignments will also be done via Moodle
  - Each submission is until 22:00 on the submission day.
  - **Do not plagiarize!**
- **Forum** will be held in Piazza
  - Search for course 89211 Advanced Programming II

# Project Milestones

- Part I – Client/Server in C# (with CLI)
- Part II – GUI with WPF, MVVM
- Part III – Web application in Java, Web API
- Part IV – Android application

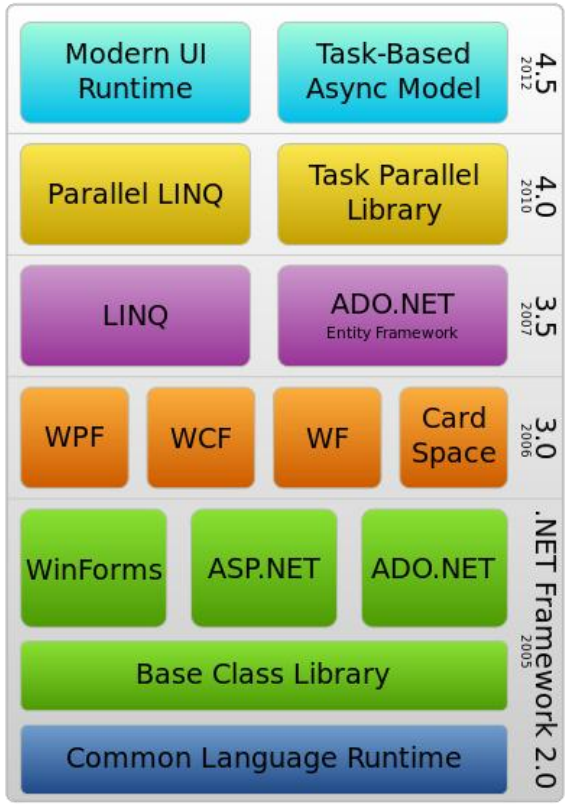


# The .NET Architecture



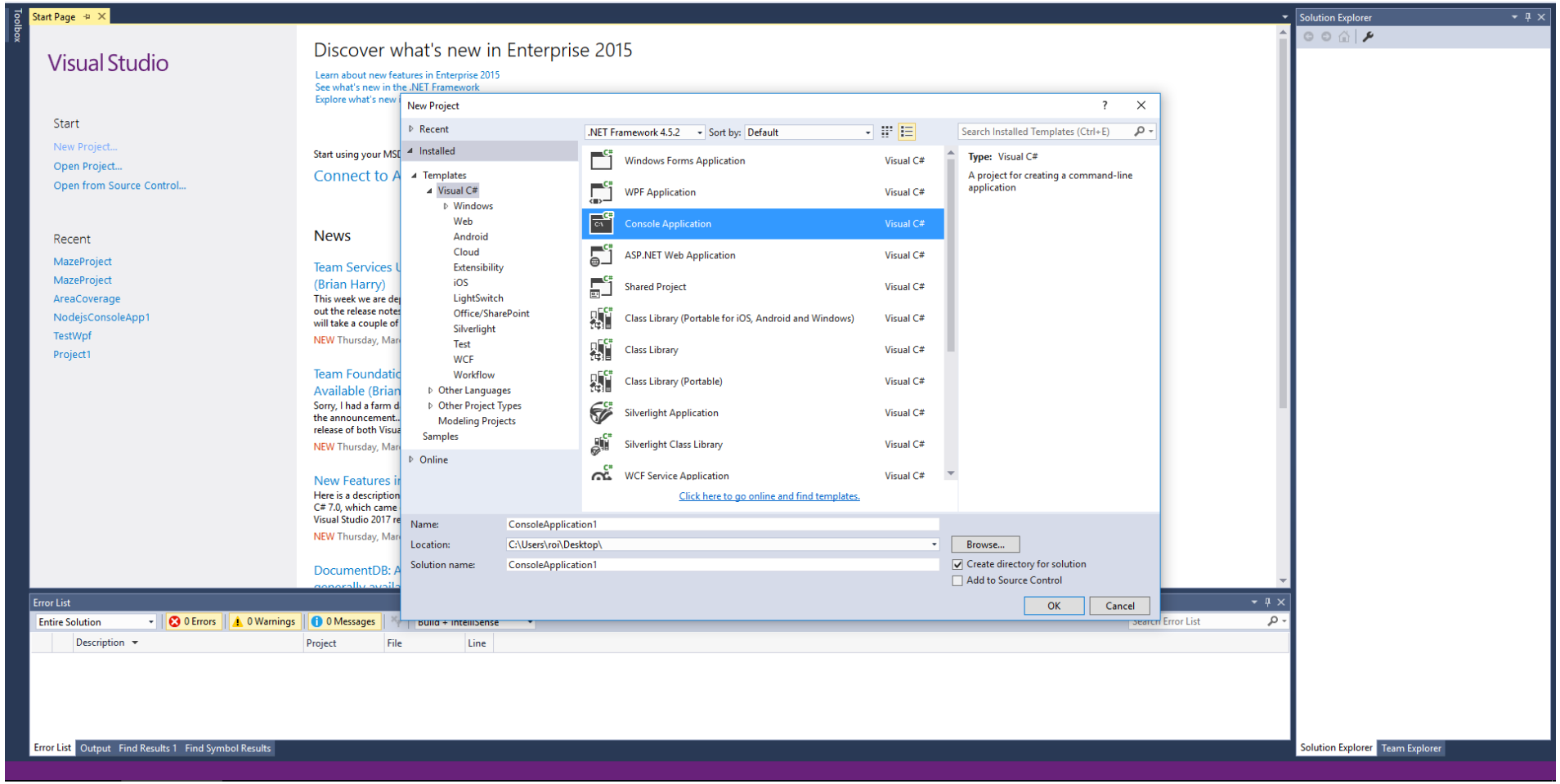
# The .Net Framework Class Library

- Console Applications
- Windows Forms Applications
- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- ADO.NET – for work with data bases
- ASP.NET – for web development



The .NET Framework Stack

# Visual Studio – New Project Wizard



# Hello World

```
using System;

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

Introduces namespaces into the file
(e.g., the class Console is part of System)

Divide the code to logical namespaces

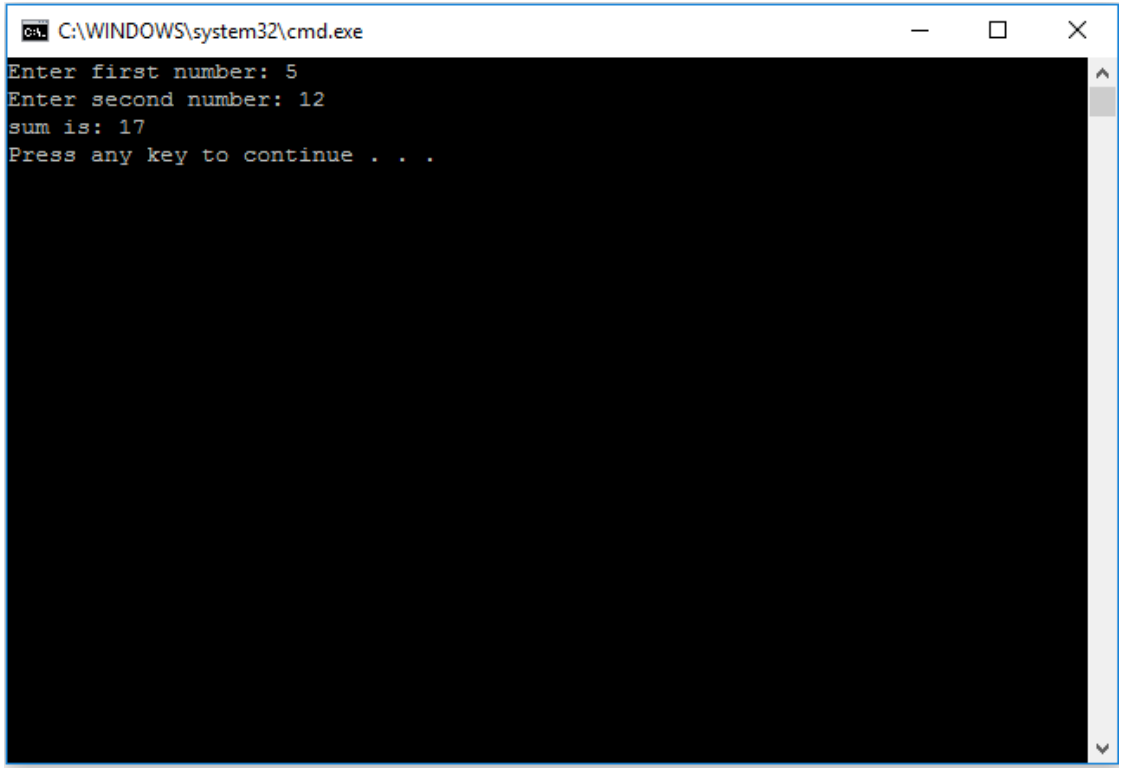
Example class and main function

# Getting Input From User

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            int num1, num2;
            Console.Write("Enter first number: ");
            num1 = int.Parse(Console.ReadLine());
            Console.Write("Enter second number: ");
            num2 = int.Parse(Console.ReadLine());

            Console.WriteLine($"sum is: {num1 + num2}");
        }
    }
}
```

New string formatting expression introduced in C# 6.0

# Getting Input From User



# Parameter Passing in C#

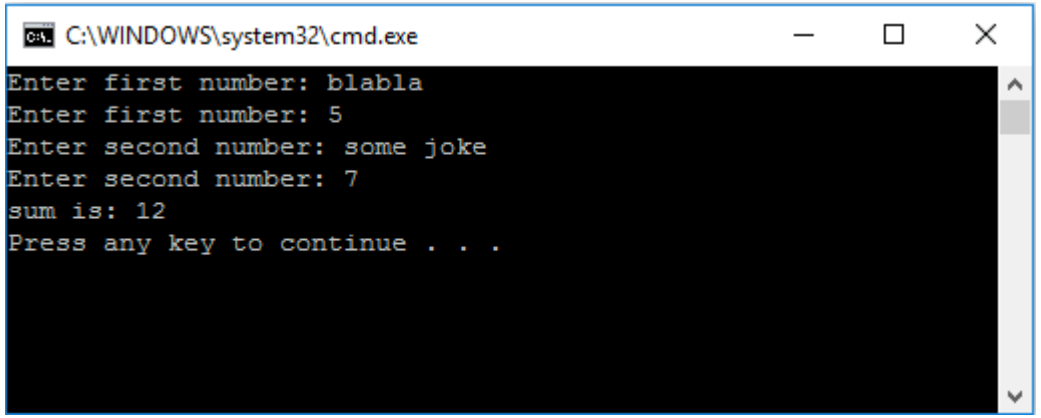
- In C# we can also pass parameters we want to change or even initialize their values
- We can use the **ref** keyword again, but this special case gets the **out** keyword
- What is the difference?
  - The variable does not have to be initialized
  - If it is initialized then its value is ignored
  - The variable must be initialized inside the met

```
static void initialize(out int x)
{ // we must initialize x somewhere in this method
     x = 0;
}
static void increment(ref int x)
{ // not a good idea to use "out". why?
     x++;
}
static void Main(string[] args)
{
     int x;
     initialize(out x);
     Console.WriteLine(x);    // 0
     increment(ref x);
     Console.WriteLine(x);    // 1
     Console.ReadKey();
}
```

# Example – TryParse()

- TryParse() converts a string into another type (e.g. int)
- Returns whether the operation was successful

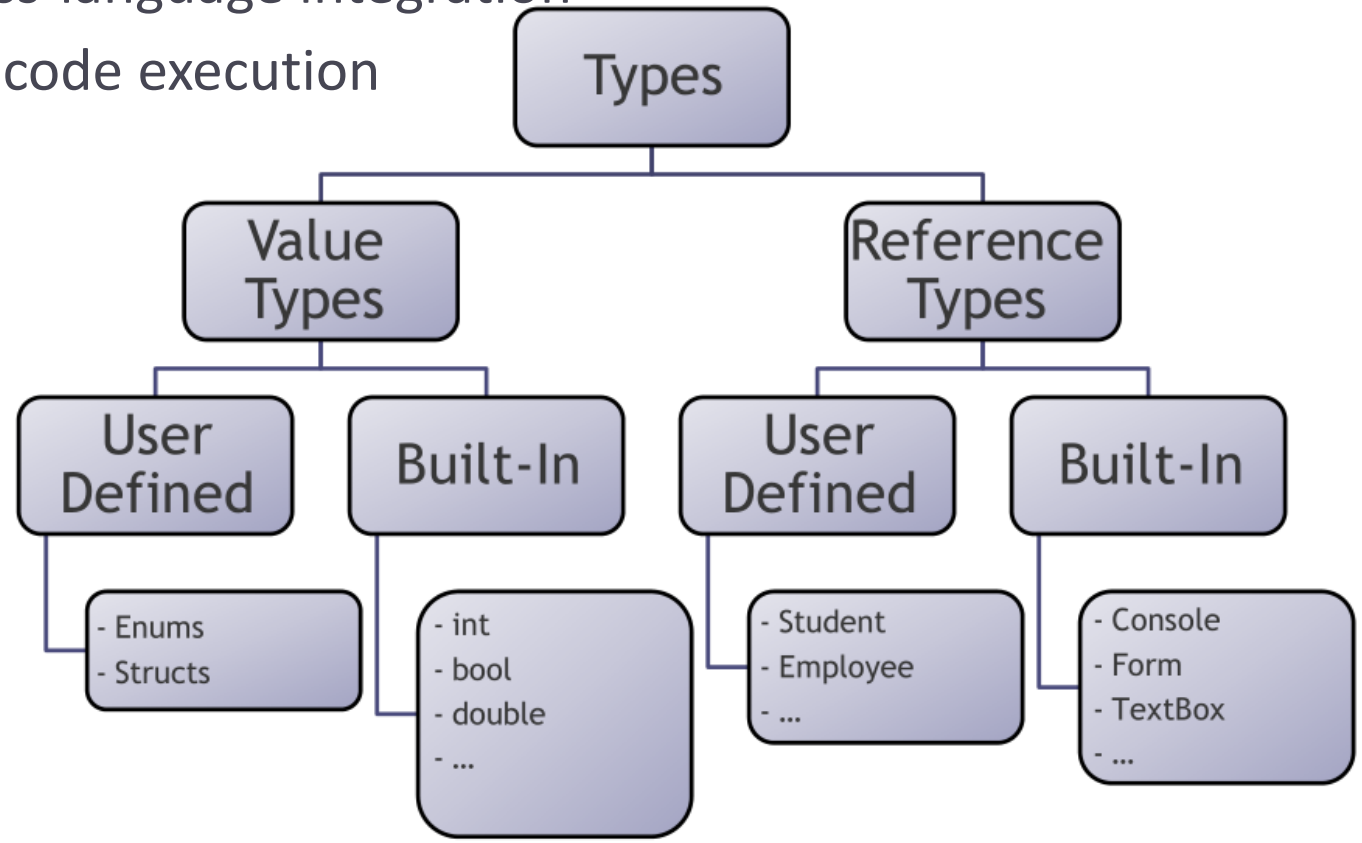
```
static void Main(string[] args)
{
    int num1, num2;

    do
    {
        Console.Write("Enter first number: ");
    } while (!int.TryParse(Console.ReadLine(), out num1));

    do
    {
        Console.Write("Enter second number: ");
    } while (!int.TryParse(Console.ReadLine(), out num2));

    Console.WriteLine($"sum is: {num1 + num2}");
}
```



# Data Types

- CTS – Common Type System
  - a framework that helps enable cross-language integration
  - type safety, and high performance code execution

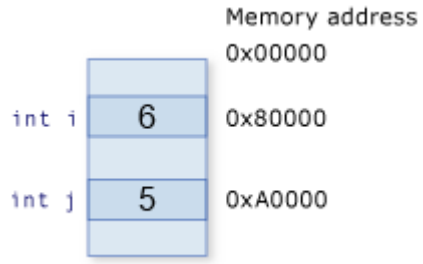
- Structures
- Enumerations
- Classes
- Interfaces
- Delegates



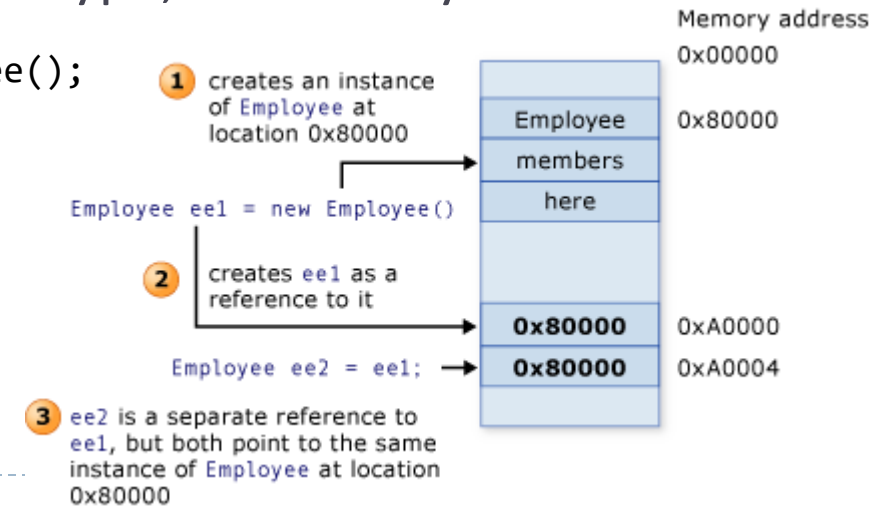
# Value Types vs. Reference Types

- Value types
  - Are the built-in primitive data types, such as char, int, as well as user-defined types declared with **struct**
  - Value types can have methods (in contrast to Java), but cannot use inhertiance
- Reference types
  - Classes and other complex data types that are constructed from the primitive types
  - Variables of such types do not contain an instance of the type, but merely a reference to an instance

```
int i = 5;
int j = i;
i = 6;
Console.WriteLine(i);    // 6
Console.WriteLine(j);    // 5
```




```
Employee ee1 = new Employee();
Employee ee2 = ee1;
```



# Built-in Data Types

| Short Name | .NET Class | Type | Width | Range (bits) |
|---|---|---|---|---|
| **byte** | Byte | Unsigned integer | 8 | 0 to 255 |
| **sbyte** | SByte | Signed integer | 8 | -128 to 127 |
| **int** | Int32 | Signed integer | 32 | -2,147,483,648 to 2,147,483,647 |
| **uint** | UInt32 | Unsigned integer | 32 | 0 to 4294967295 |
| **short** | Int16 | Signed integer | 16 | -32,768 to 32,767 |
| **ushort** | UInt16 | Unsigned integer | 16 | 0 to 65535 |
| **long** | Int64 | Signed integer | 64 | -9223372036854775808 to 9223372036854775807 |
| **ulong** | UInt64 | Unsigned integer | 64 | 0 to 18446744073709551615 |
| **float** | Single | Single-precision floating point type | 32 | -3.402823e38 to 3.402823e38 |
| **double** | Double | Double-precision floating point type | 64 | -1.79769313486232e308 to 1.79769313486232e308 |
| **char** | Char | A single Unicode character | 16 | Unicode symbols used in text |
| **bool** | Boolean | Logical Boolean type | 8 | True or false |
| **object** | Object | Base type of all other types | | |
| **string** | String | A sequence of characters | | |
| **decimal** | Decimal | Precise fractional or integral type that can represent decimal numbers with 29 significant digits | 128 | ±1.0 × 10e−28 to ±7.9 × 10e28 |

- Java has primitive types and wrapper classes e.g.
  - int – Integer
  - double – Double
- Primitive types in C# are Objects!
  - int is an alias for System.Int32
  - double is an alias for System.Double

```
static void Main()
{
    int i = 10;
    object o = i;

System.Console.WriteLine(o.ToString());
}
```

# C# Struct vs. Class

- Classes are reference types and structs are value types
  - Instances of a class are allocated on the heap, while instances of a struct are created on the stack
  - This yields performance gains since you will not be dealing with references to an instance of a struct as you would with classes
- There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class.
  - Structs, however, inherit from the base class object.
  - A struct can implement interfaces, and it does that exactly as classes do

```
class Point
{
    int x;
    int y;

    public void Print()
    {
        Console.WriteLine("X = {0}, Y = {1}", x, y);
    }
}
```

```
struct Point
{
    int x;
    int y;

    public void Print()
    {
        Console.WriteLine("X = {0}, Y = {1}", x, y);
    }
}
```

# Properties

# Properties

- It is a good idea for **data members** to be **private**
- Public setters & getters can provide **managed access** to these private data members
- Yet, it would be nicer to access data members rather than activate a method…

- x and y are private data members
- X and Y are public ***properties***
  - They have a setter and a getter
  - They manipulate x and y

- Outside the class, X and Y are used as a public data members

```
class Point
{
    private int x;
    private int y;

    public int X
    {
        get { return x; }
        set { x = value; }
    }

    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    public void Print()
    {
        Console.WriteLine("X = {0}, Y = {1}", x, y);
    }
}
```

```
Point p = new Point();
p.X = 10;
p.Y = 15;
p.Print();
```

# Properties

- You can add validity checks in the setter part of the property

```
public int X
{
    get { return x; }
    set
    {
        if (value < 0)
            return;
        x = value;
    }
}

public int Y
{
    get { return y; }
    set
    {
        if (value < 0)
            return;
        y = value;
    }
}
```

# Automatic Properties

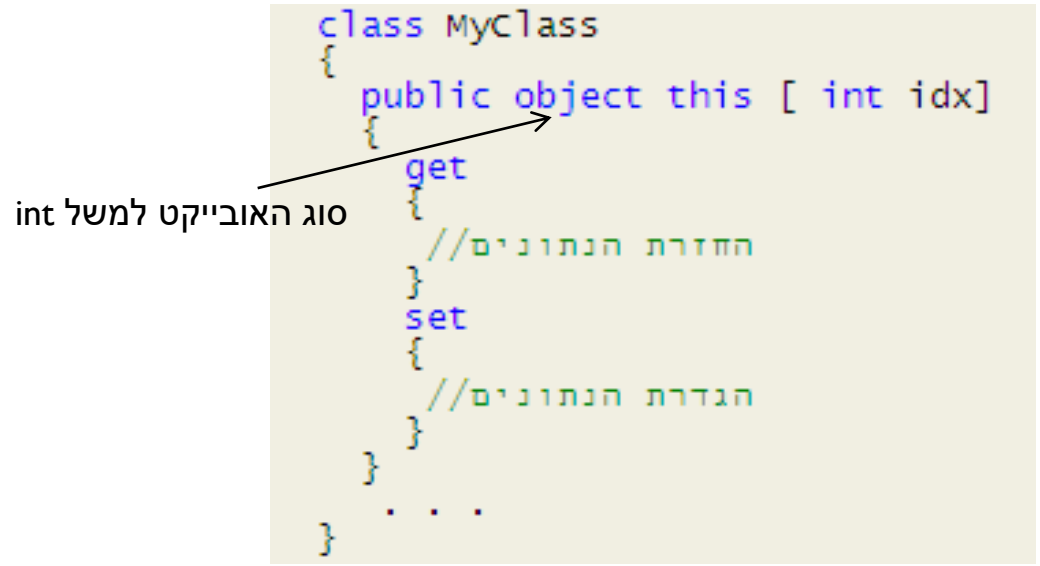
- In C# 3.0 and later, auto-implemented properties make property declaration more concise when no additional logic is required in the property accessors
- When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public void Print()
    {
        Console.WriteLine("X = {0}, Y = {1}", X, Y);
    }
}
```

# Indexers

- An **indexer** allows an object to be indexed such as an array
- When you define an indexer for a class, this class behaves similar to a virtual array
- You can then access the instance of this class using the array access operator ([ ])

- Indexers are defined similarly to properties
  - However, indexers are not defined with names but with the **this** keyword



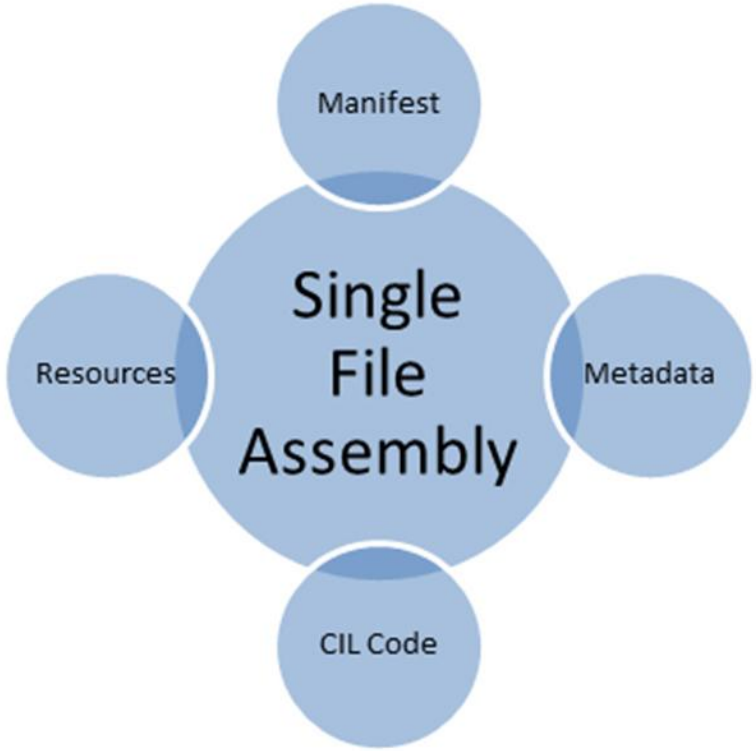

דוגמא לשימוש:

```
MyClass cls = new MyClass();
cls[0] = someObject;
Console.WriteLine('{0}', cls[0]);
```

# Assemblies and Namespaces

# Assemblies

- **Assemblies** are the building blocks of .NET Framework applications
- An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality
- Assemblies have the extension **.exe** or **.dll**
- **GAC (Global Assembly Cache)**
  - A folder in Windows directory that stores the .NET assemblies that are designated to be shared by all applications executed on a system



# Namespaces

- Namespaces are logical groupings of classes into different scopes
- A fully-qualified name of a class contains the name of its namespace
- The using directive allows the use of types in a namespace so that you don't have to qualify the use of a type in that namespace

```
namespace SampleNamespace
{
    class SampleClass { }

    interface SampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

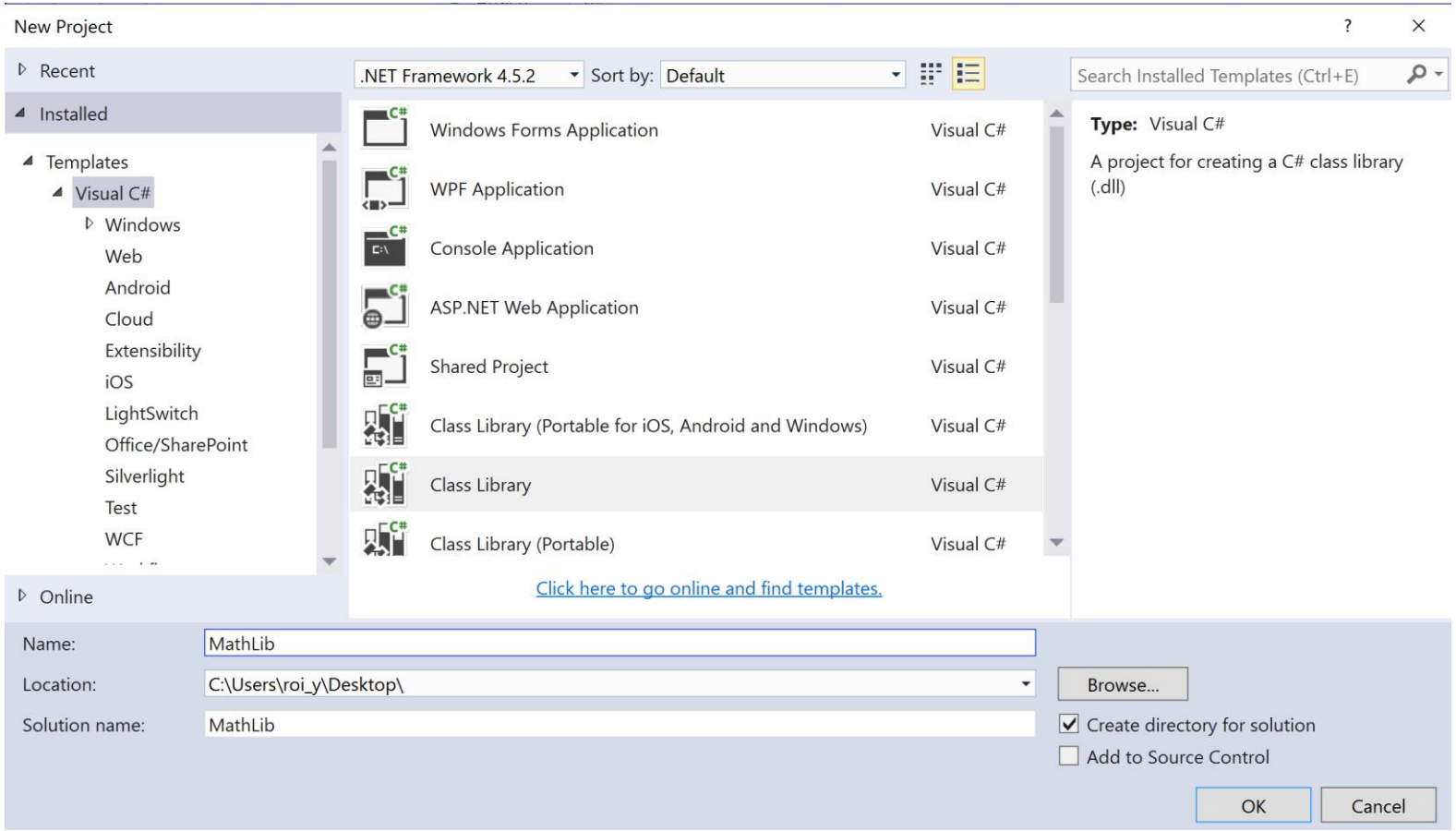
```
using SampleNamespace;

static void Main(string[] args)
{
    SampleClass obj = new SampleClass();
}
```

# Class Library (dll)

- A class library is a set of classes, interfaces, and value types that can be shared among various projects
- Advantages of class libraries are:
  - Code re-usability
  - Better code management
  - Modularization
- When you add a new project of type "Class Library", its output will be a file of type dll
- A dynamic linking library (DLL) is linked to your program at run time

# Creating a New Class Library

- We will create a library called MathLib for mathematical functions



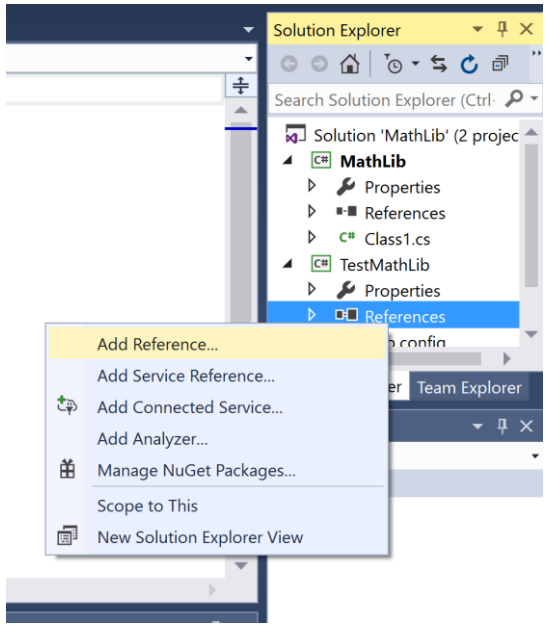
# Creating a New Class Library

- Make sure that the classes you want to export from the class library are defined as **public** (default is internal)

```
namespace MathLib
{
    public class MathFunctions
    {
        public static int Plus(int x, int y)
        {
            return x + y;
        }

        public static int Minus(int x, int y)
        {
            return x - y;
        }
    }
}
```

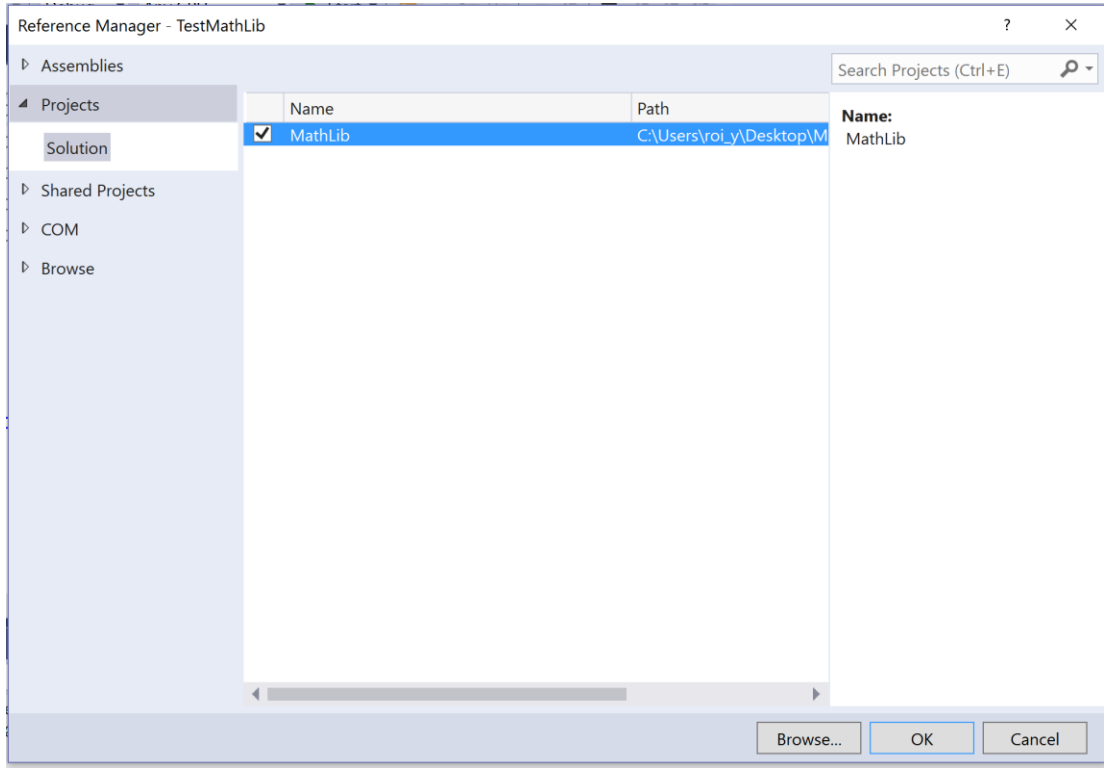
# Referencing a dll

- In order to use a dll in another project, you have to add a reference to it
  - This means the dll will be copied to the bin/debug folder of your project (unless it's in the GAC)
- Add a reference to MathLib.dll by right-clicking the References item under the project name



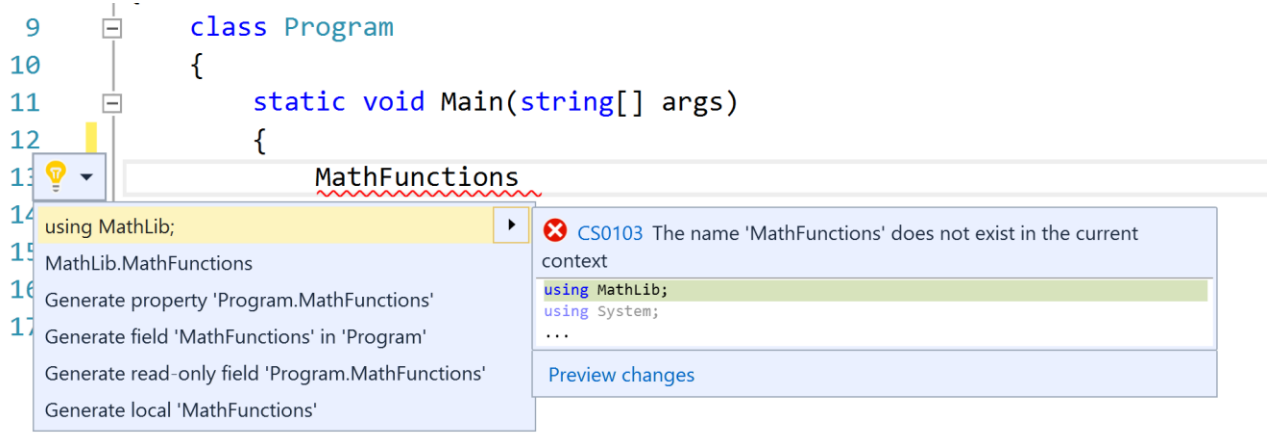
# Referencing a dll

- Choose MathLib from the Projects tab



# Using a dll

- Typically, the classes that you use from the class library belong to a different namespace
- Thus, you will have to add a using statement with the namespace of the class library
  - You can use Ctrl+. to add the missing using statement



# Using a dll

```
using MathLib;
using System;

namespace TestMathLib
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = MathFunctions.Plus(1, 5);
            Console.WriteLine(result);
        }
    }
}
```

# Running the App

- Change the StartUp project of the solution to be the console application by right-clicking the project in the Solution Explorer and choosing “Set as Startup Project”