I. C# / .NET Framework Basics

Contents

1. Introduction

1.1. Prerequisites

- Visual Studio 2013 / 2015 (can be downloaded from http://microsoft.ase.ro or http://www.dreamspark.com/)

1.2. API Reference

- https://msdn.microsoft.com/en-us/library/
- https://msdn.microsoft.com/en-us/library/w0x726c2%28v=vs.110%29.aspx

2. First C# Program

2.1. The Main() method

Every C# application must contain a single Main method specifying where program execution is to begin.

Observations:

- in C#, Main is capitalized, while Java uses lowercase main;
- note the **static** modifier which has a similar behavior with C++.

Activity:

- create a new Microsoft .NET Project in Visual Studio;
- add the following code.

```
using System; //referenced namespace

namespace NameSpaceProgram
{
      internal class Program
      {
            private static void Main(string[] args)
            {
                  //HelloWorld application
                  Console.WriteLine("Hello World!");
                  Console.ReadLine();
            } //end main
      }//end class
} //end namespace
```

Activity:

- download dotPeak from https://www.jetbrains.com/decompiler/;
- decompile the HelloWorld application.

2.2. Reading and Writing using System.Console

Documentation: https://msdn.microsoft.com/en-us/library/system.console(v=vs.110).aspx

Frequently used methods:

- Write / WriteLine;
- Read / Readline.

2.3. Formatting Console Output

Activity:

```
Console.WriteLine("Argument: {0}", 10);
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
Console.WriteLine("c format: {0:c}", 99999);
```

2.4. Specifying an Application Error Code

- the Main() method can only return **int** or **void**;

Activity:

- replace the code from the previous activity with the following one.

```
static int Main(string[] args)
{
      Console.WriteLine("Hello World!");
      Console.ReadLine();
      // Return an arbitrary error code.
      return -1;
}
```

```
@echo off

NameofYourApp

@if "%ERRORLEVEL%" == "0" goto success
:fail
echo This application has failed!
echo return value = %ERRORLEVEL%
goto end
:success
echo This application has succeeded!
echo return value = %ERRORLEVEL%
goto end
:end
echo All Done.
```

2.5. Processing Command-Line Arguments

- the Main() method has an optional string array argument to represent command-line parameters.

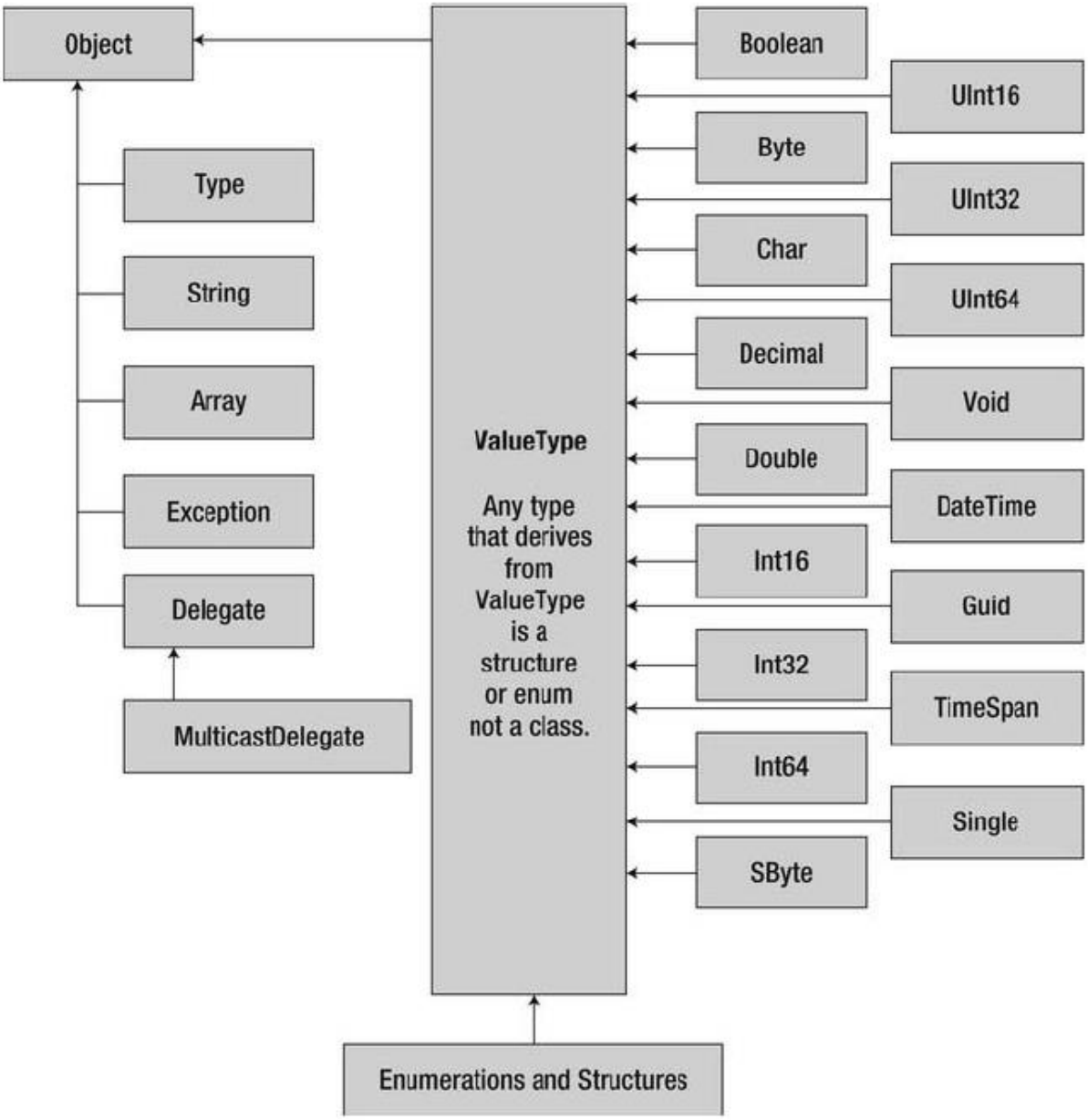
Activity:

- replace the code from the previous activity with the following one.

```
public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            Console.WriteLine("Argument: {0}", arg);
        }
    }
```

```
NameofYourApp.exe /arg1 -arg2
```

3. Data Types



Some of these types have shorthand defined in C#, as shown in the following table.

The Intrinsic Data Types of C#

C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit number
ulong	No	System.UInt64	0 to 18,446,744,073,709,551, 615	Unsigned 64-bit number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character
float	Yes	System.Single	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	32-bit floating-point number
double	Yes	System.Double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	64-bit floating-point number
decimal	Yes	System.Decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	96-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
Object	Yes	System.Object	Can store any data type in an object variable	The base class of all types in the .NET universe

3.1. Value types

- structure, enum, primitive types (derived from System.ValueType)
- allocated: on the stack;
- lifetime: can be created and destroyed very quickly, as its lifetime is determined by the defining scope;

3.2. Reference types

- class, delegate, interface
- allocated: in the heap;
- lifetime: has a lifetime that is determined by a large number of factors

Table 1 Comparison between value types and reference types

	Value type	Reference Type
Where are objects allocated?	Allocated on the stack.	Allocated on the managed heap.

How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Implicitly extends System.ValueType.	Can derive from any other type (except System.ValueType), as long as that type is not “sealed”.
Can this type function as a base to other types?	No. Value types are always sealed and cannot be inherited from.	Yes. If the type is not sealed, it may function as a base to other types.
Default parameter passing behavior	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	For value types, the object is copied-by-value. For reference types, the reference is copied-by-value.
Own constructor for this type	Yes, but the default constructor is reserved (i.e., the custom constructors must all have arguments).	Yes
When do variables of this type die?	When they fall out of the defining scope.	When the object is garbage collected.

Activity:

- Add the following method to the program in the previous activity;
- Call it from the Main method.

```
private static void SystemDataTypes()
{
      Console.Write("First Name: ");

      //declare the variable
      //string firstName;
      //store the input from the keyboard
      //firstName = Console.ReadLine();

      //written more concisely
      string firstName = Console.ReadLine();

      Console.WriteLine("");

      DateTime currentTime = DateTime.Now;

      //{0} and {1} are replaced with the arguments
      //Console.WriteLine(string.Format("Hello {0}! Today is {1}.",
firstName, DateTime.Now));
      //written more concisely
      Console.WriteLine("Hello {0}! Today is {1}.", firstName,
currentTime);

      Console.ReadLine();
}
```

3.3. Implicitly Typed Local Variables

Local variables can be given an inferred "type" of **var** instead of an explicit type. The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement.

Documentation: https://msdn.microsoft.com/en-us/library/bb384061.aspx

Activity:

- Use **var** for variable declarations instead of **string** or **DateTime** in the previous activity.

4. Working with Strings

4.1. Immutable

- Strings are **immutable:** after the initial value is assigned to a string object, the character data cannot be changed. A brand new string is created each time we modify the initial string.

Activity:

```
internal class Program
      {
            private static void Main(string[] args)
            {
                  //Ex1
                  string s1 = "abc";
                  string s2 = s1;
                  s1=s1.Replace("abc", "ba");
                  Console.WriteLine(s1);
                  Console.WriteLine(s2);

                  //Ex2
                  String s3 = "abc";
                  String s4 = s3;
                  s3 += "d";
                  Console.WriteLine(s3);
                  Console.WriteLine(s4);

            /*Strings are immutable--the contents of a string object
cannot be changed after the object is created, although the syntax makes
it appear as if you can do this. */
            }
}
```

4.2. operator==

- The **operator==** is overloaded.

Activity:

```
using System;
using System.Text;

namespace ExempluString
{
      internal class Program
      {
```

```
            private static void Main(string[] args)
            {
                  //2. Operator==
                  string a = "hello";

                  string b = "h";
                  // Append to contents of 'b'
                  b += "ello";

                  Console.WriteLine(a == b); //will return true because
operator== is overloaded
                  Console.WriteLine((object)a == (object)b);      //will
return false because the objects are different

            }
      }
}
```

4.3. StringBuilder

For routines that perform extensive string manipulation (such as apps that modify a string numerous times in a loop), modifying a string repeatedly can exact a significant performance penalty. The alternative is to use StringBuilder, which is a mutable string class. Mutability means that once an instance of the class has been created, it can be modified by appending, removing, replacing, or inserting characters.

Usage example: email containing phone logs or calendar entries, text based reports

Documentation: https://msdn.microsoft.com/en-us/library/system.text.stringbuilder%28v=vs.110%29.aspx

Activity

- Compare the time required to perform the same operation using System.String and System.Text.StringBuilder

```
private static void StringBuilderPerformance()
{
      Console.WriteLine("###StringBuilderPerformance");

      const int noOfRepetitions = 50000;

      var regularString = string.Empty;

      // For a more precise measurement, use a performance counter
instead of a Stopwatch
      var watch = Stopwatch.StartNew();
      for (var i = 0; i < noOfRepetitions; i++)
      {
            regularString += "a";
      }
      watch.Stop();
      var elapsedMs = watch.ElapsedMilliseconds;

      Console.WriteLine("Using System.String: {0}ms", elapsedMs);

      var stringBuilder = new StringBuilder();
```

```
        watch = Stopwatch.StartNew();
        for (var i = 0; i < noOfRepetitions; i++)
        {
                stringBuilder.Append("a");
        }
        regularString = stringBuilder.ToString();
        watch.Stop();
        elapsedMs = watch.ElapsedMilliseconds;

        Console.WriteLine("Using System.Text.StringBuilder: {0}ms",
elapsedMs);

        Console.ReadLine();
}
```

5. Arrays

An array represents a fixed number of variables (called elements) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access. All arrays inherit from the **System.Array** class, providing common services for all arrays.

Some of the most important properties and methods:

- ***Length***: the number of elements in an array;
- ***GetLength(dim)***: gets a 32-bit integer that represents the number of elements in the specified dimension of the Array;
- ***Rank***: gets the rank (number of dimensions) of the Array. For example, a one-dimensional array returns 1, a two-dimensional array returns 2, and so on.;
- ***Clone()*:** Creates a shallow copy of the Array (note that System.Array implements ICloneable);
- *and many others…*

Questions

- Which is the base class for System.Array?

5.1. Default Element Initialization

Creating an array always preinitializes the elements with default values:

- **null** for reference types;
- **0** for numeric types;
- Etc.

Activity

```
private void Array()
{
        //1. Declaration and assignment
        //declaration
        int[] intArray;
```

```
      // allocation
      intArray = new int[3]; //all values will be 0

      // declaration and assignment
      //var doubleArray = new double[]{ 34.23, 23.2 };
      var doubleArray = new[] { 34.23, 10.2, 23.2 }; //data type (double)
is inferred

      //2. Accessing elements
      var arrayElement = doubleArray[0];
      doubleArray[1] = 5.55;

      // for
      for (var i = 0; i < intArray.Length; i++)
            Console.WriteLine(intArray[i]);

      // foreach
      foreach (var value in doubleArray)
            Console.WriteLine(value);

      //3. Other methods
      Array.Sort(doubleArray); //Note: double implements
IComparable<double>

      // for
      for (var i = 0; i < doubleArray.Length; i++)
            Console.WriteLine("doubleArray[{0}]={1}", i, doubleArray[i]);

}
```

Questions

- What kind of method is **Sort()** // what modifiers do you expect it to have

5.2. Multidimensional Arrays

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension (ex: [dim_1, dim_2, …, dim_n]).

Activity

```
private void RectangularArray()
{
      // declaration and allocation
      var cub = new int[5, 2, 7];

      // declaration and assignment
      var matrix = new[,]{
                  { 4, 23,  5,  2 },
                  { 1,  6, 13, 29 }
            };

      // for
      for (var i = 0; i < matrix.GetLength(0); i++)
      {
            for (var j = 0; j < matrix.GetLength(1); j++)
                  Console.Write(" {0}", matrix[i, j]);
            Console.WriteLine();
```

```
        }
}
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension.

```
private void JaggedArray()
{
        int[][] jaggedArray =
        {
                new int[] {0, 1, 2},
                new int[] {3, 4},
                new int[] {6, 7, 8, 9}
        };
}
```