The C# Language

Windows Programming Course

Agenda

1. Overview of the language
2. Variables
3. Types
4. Statements
5. Declarations

1.

Overview of the language

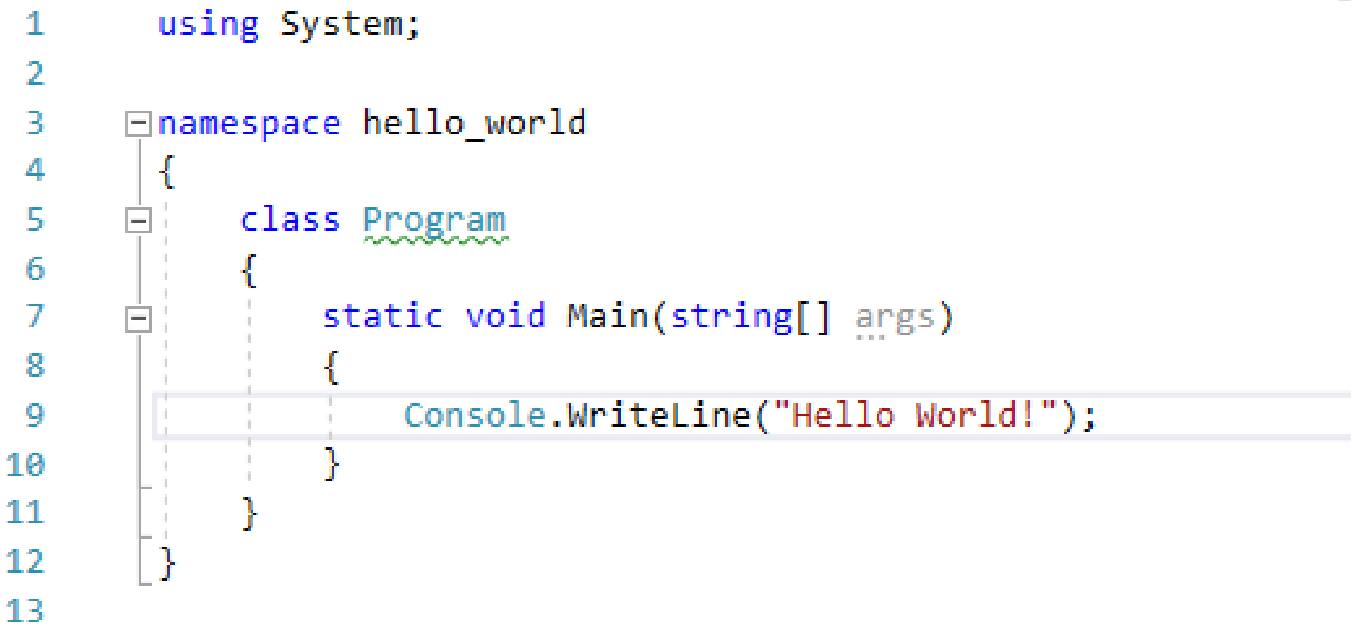
The C# language

- C# is a modern, object-oriented, and type-safe programming language.
- Designed with ideas from C++, Java and Pascal.
- Latest Version:
  - 8.0 is supported on .NET Core 3.x and .NET Standard 2.1
  - 9.0 is supported on .NET 5

Characteristics of C#

- Very similar in syntax to C, C++, and Java.
- Syntax is highly expressive.
- Key features: nullable value type, enumerations, delegates, lambda expressions, and LINQ.

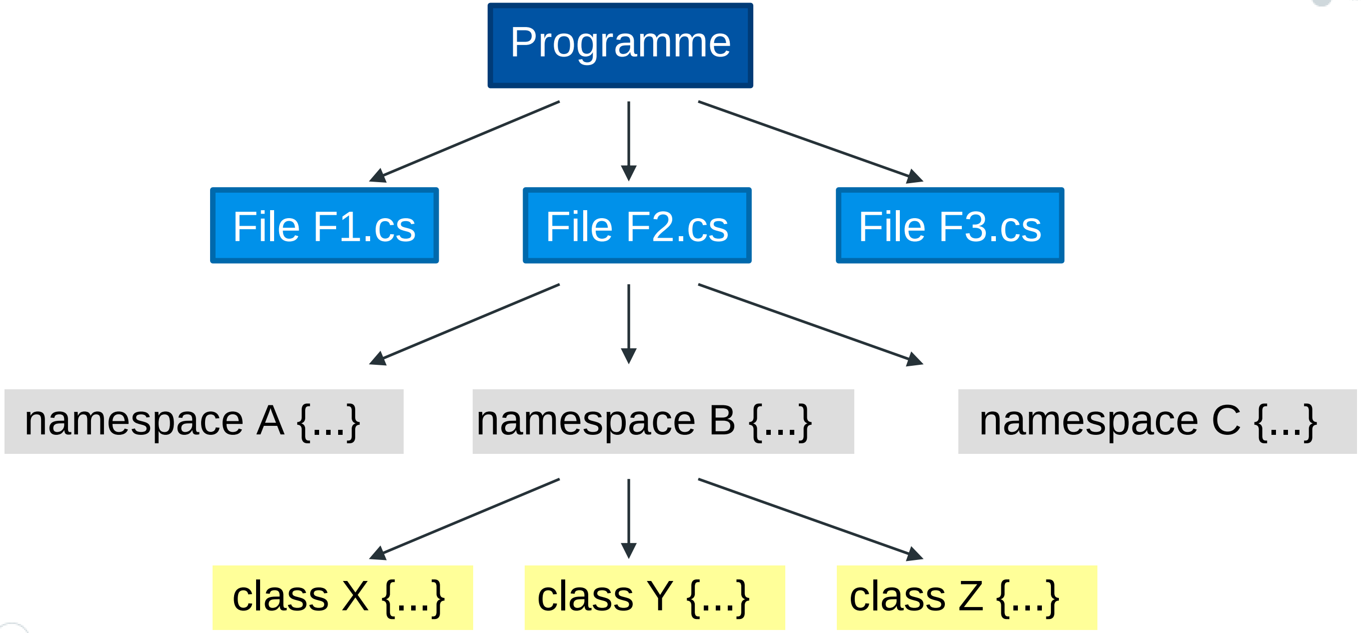
“Hello World” program



```
using System;

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

Structure of C# programs



2.
Variables

Declare variables

Declare a variable by using following syntax:

```
datatype identifier;
```

For example:

```
int i; // Explicitly typed
int j = 10; // Declare and initialize variable's value
int x = 10, y = 20; // Declare more than one variable
var k = 20; // Implicitly typed
```

```
int k = 20;
```

Variable Initialization rules

- The variable must be initialized. Otherwise, the compiler doesn’t have anything from which to infer the type.
- The initializer cannot be null.
- The initializer must be an expression.
- You can’t set the initializer to an object unless you create a new object in the initializer

Variable Scope

The *scope* of a variable is the region of code from which the variable can be accessed.

- A *field* (also known as a member variable) of a class is in scope for as long as a local variable of this type is in scope.
- A *local variable* is in scope until a closing brace indicates the end of the block statement or method in which it was declared.
- A local variable that is declared in a for, while, or similar statement is in scope in the body of that loop.

Variable Scope - Example

```
void foo(int a) {
  int b;
  if (...) {
       int b;          // error: b already declared in outer block
       int c;          // ok so far, but wait ...
       int d;
       ...
  } else {
       int a;          // error: a already declared in outer block
       int d;          // ok: no conflict with d from previous block
  }
  for (int i = 0; ...) {...}
  for (int i = 0; ...) {...} // ok: no conflict with i from previous loop
  int c;               // error: c already declared in this declaration space
}
```

Constants

A constant is a variable whose value cannot be changed throughout its lifetime.

Declaration syntax:

```
const datatype identifier;
```

For example:

```
const int MAX = 100;
```

Constants – Characteristics

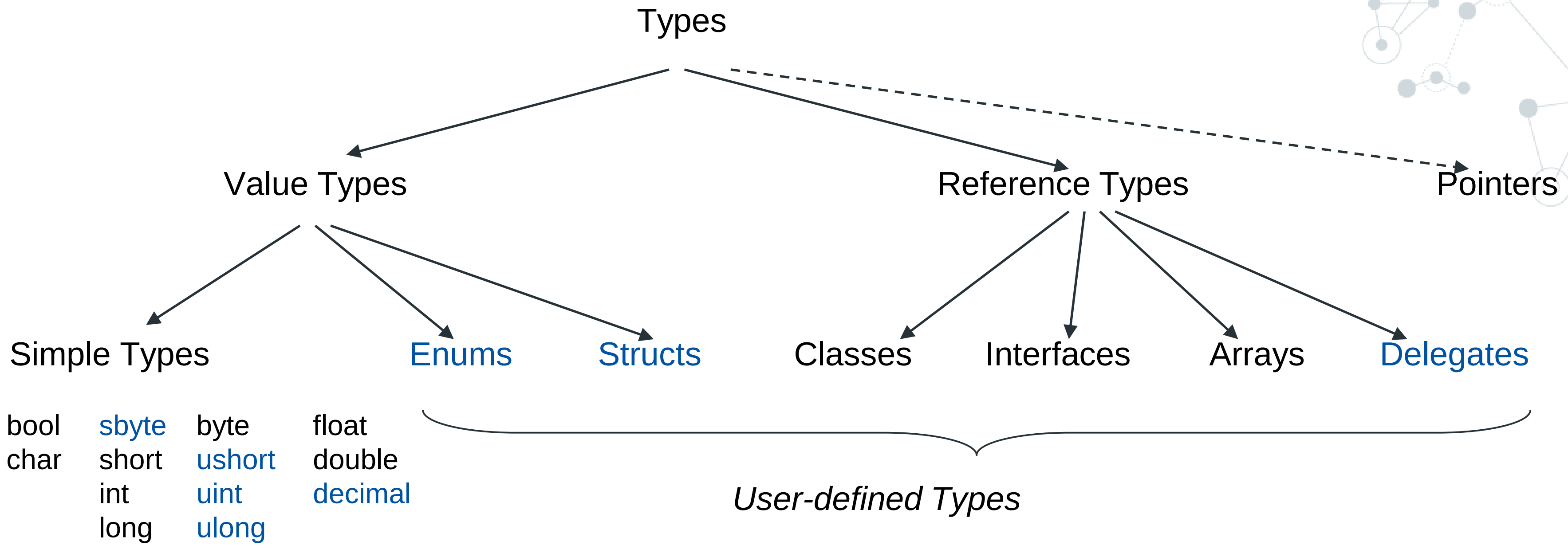
- Must be initialized when it is declared.
- The value of a constant must be computable at compile time => Can’t initialize a constant with a value taken from a variable.
- Constants are always implicitly static.

Constants – Advantages

- Replace magic numbers and strings with readable names.
- Make your programs easier to modify (avoid code duplication).
- Prevent mistakes if a constant is modified somewhere in program.

3.
Types

Unified Type System



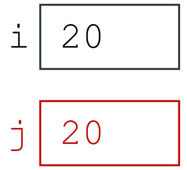
All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

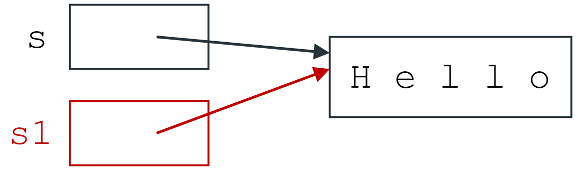
Value Types and Reference Types

	Value Types	Reference Types
Variable contains	**value**	**reference**
Stored on	**stack**	**heap**
Initialisation	**0, false, '\0'**	**null**
Assignment	**copies the value**	**copies the reference**

```
int i = 20;
int j = i;
```



```
string s = "Hello";
string s1 = s;
```



Value Types – Predefined Value Types: **Integer Types**

NAME	.NET TYPE	DESCRIPTION	RANGE (MIN:MAX)
sbyte	System.SByte	8-bit signed integer	−128:127 (−27:27−1)
short	System.Int16	16-bit signed integer	−32,768:32,767 (−215:215−1)
int	System.Int32	32-bit signed integer	−2,147,483,648:2,147,483,647 (−231:231−1)
long	System.Int64	64-bit signed integer	−9,223,372,036,854,775,808: 9,223,372,036,854,775,807 (−263:263−1)
byte	System.Byte	8-bit unsigned integer	0:255 (0:28−1)
ushort	System.UInt16	16-bit unsigned integer	0:65,535 (0:216−1)
uint	System.UInt32	32-bit unsigned integer	0:4,294,967,295 (0:232−1)
ulong	System.UInt64	64-bit unsigned integer	0:18,446,744,073,709,551,615 (0:264−1)

Value Types – Predefined Value Types: **Floating-Point Types**

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (MIN:MAX)
float	System.Single	32-bit, single-precision floating point	7	±1.5 × 10245 to ±3.4 × 1038
double	System.Double	64-bit, double-precision floating point	15/16	±5.0 × 102324 to ±1.7 × 10308

Value Types – Predefined Value Types: **Decimal Types**

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (MIN:MAX)
decimal	System.Decimal	128-bit, high-precision decimal notation	28	±1.0 × 10228 to ±7.9 × 1028

Value Types – Compatibility between Value Types

$$\begin{tikzcd}
\text{decimal} & \text{double} \arrow[l, dotted, "\text{only with type cast}"'] & \text{float} \arrow[l] & \text{long} \arrow[l] & \text{int} \arrow[l] & \text{short} \arrow[l] & \text{sbyte} \arrow[l] \\
 & & & \text{ulong} \arrow[ul] & \text{uint} \arrow[ul] \arrow[l] & \text{ushort} \arrow[ul] \arrow[l] & \text{byte} \arrow[ul] \arrow[l] \arrow[d] \\
 & & & & & & \text{char} \arrow[uul]
\end{tikzcd}$$

Value Types – Enumeration Types

Define a set of named constants of the underlying integral numeric type.

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Value Types – Structure Types

Encapsulate data and related functionality

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Value Types – Nullable value Types

- A *nullable value type* T? represents all values of its underlying value type T and an additional null value.

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

Value Types – Nullable value Types (cont.)

- Examination of an instance of a nullable value type.

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

Reference Types – Predefined Reference Types

NAME	.NET TYPE	DESCRIPTION
object	System.Object	The root type. All other types (including value types) are derived from object.
string	System.String	Unicode character string

4.
Statements

Simple statements

Empty statement

```
;                    // ; is a terminator, not a separator
```

Assigment

```
x = 3 * y + 1;
```

Method call

```
string s = "a,b,c";
string[] parts = s.Split(',’); // invocation of an object method
(non-static)
s = String.Join(" + ", parts); // invocation of a class method
(static)
```

Conditional Statements – if

```
string input;
input = Console.ReadLine();
if (input == "") {
  Console.WriteLine("You typed in an empty string.");
}
else if (input.Length < 5) {
  Console.WriteLine("The string had less than 5 characters.");
}
else {
  Console.WriteLine("The string had more than 5 Characters.");
}
```

Conditional Statements – switch

```
switch (integerA) {
case 1:
  Console.WriteLine("integerA = 1"); break;
case 2:
  Console.WriteLine("integerA = 2"); break;
case 3:
  Console.WriteLine("integerA = 3"); break;
default:
  Console.WriteLine("integerA is not 1, 2, or 3"); break;
}
```

Loop Statements

while

```
while (i < n) {
  sum += i;
  i++;
}
```

do … while

```
do {
  sum += a[i];
  i--;
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)
  sum += i;
```

Loop Statements – foreach

For iterating over collections and arrays

```
int[] a = {3, 17, 4, 8, 2, 29};
foreach (int x in a) sum += x;

string s = "Hello";
foreach (char ch in s)
  Console.WriteLine(ch);
```

5. Declarations

Declaration space

- The program area to which a declaration belongs
- **Entities can be declared in a ...**
  - **namespace**: Declaration of classes, interfaces, structs, enums, delegates
  - **class, interface, struct**: Declaration of fields, methods, properties, events, indexers, ...
  - **enum**: Declaration of enumeration constants
  - **block**: Declaration of local variables

Declaration space (cont.)

- **Scoping rules**
  - A name must not be declared twice in the same declaration space.
  - Declarations may occur in arbitrary order.
    Exception: local variables must be declared before they are used
- **Visibility rules**
  - A name is only visible within its declaration space
    (local variables are only visible after their point of declaration).
  - The visibility can be restricted by modifiers (private, protected, ...)

Namespaces

Equally named namespaces in different files constitute a single declaration space.

Nested namespaces constitute a declaration space on their own.

File X.cs

```
namespace A {
            ... Classes ...
            ... Interfaces ...
            ... Structs ...
            ... Enums ...
            ... Delegates ...
  namespace B {  // full name: A.B
            ...
  }
}
```

File Y.cs

```
namespace A {
            ...
  namespace B {...}

}
namespace C {...}
```

Using Other Namespaces

Color.cs

```
namespace Util {
    public enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    public class Rect {...}
    public class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    public class Triangle {...}
}
```

```
using Util.Figures;

class Test {
   Rect r;       // without qualification (because of using Util.Figures)
   Triangle t;
   Util.Color c;     // with qualification
}
```

Foreign namespaces

- must either be imported (e.g. using Util;)
- or specified in a qualified name (e.g. Util.Color)

Hands-on Exercise

Implement a console application:

- Print to console “Please enter your name”
- Read input
- Print “Hello {name}”

Hands-on Exercise

Integer operations:

- Read 2 integer numbers: x and y from the command line
- Print result of 4 operations: sum, subtract, multiply and divide
  - e.g.:

X + Y = {sum}

X – Y = {subtract}

X * Y = {multiply}

X / Y = {divide}

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com