

# Lecture 3 – Hello World & Simple Types

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

# Part I

Hello World

By longstanding tradition, the first program example when learning a new programming language is known as the **Hello World** program.

All it does is write the text “Hello World” to the display device.

It is a simple program; easily understood.

It takes very little time to write and is hard to get wrong.

Successful execution of it proves that the environment, compiler, display system, etc. are all present and working.

```
using System;

class HelloWorld
{
    static void Main( )
    {
        Console.WriteLine( "Hello World" );
    }
}
```

# Examining the Hello World Program

Let's examine each of the lines of this program.

At this point it is not expected that you will understand all the lines.

We'll say what they are, so more advanced students can understand the structure. If it's confusing to you, you will understand later.

What we are really interested in takes place between the inner set of { } braces.

# Examining the Hello World Program

```
using System;
```

This indicates the use of objects in the System namespace.

```
class HelloWorld
```

Declares an object named HelloWorld

```
static void Main()
```

This is the entry point for program execution.

```
Console.WriteLine("Hello World");
```

Tells the computer to output “Hello World” to the console.

In the Hello World program, we see the { and } braces twice each.

These define a **block**: a grouping of statements.

The { indicates the start of a block;

The } indicates the end of that block.

They must be balanced (otherwise it's a syntax error).

Examine: `Console.WriteLine("Hello World");`

Note the presence of a semicolon (;) at the end of that statement.

Often, a single statement appears on a line.

In C#, the semicolon denotes the end of many (but not all) statements.

The end of the line doesn't indicate the end of a statement.

This allows:

- A statement to be on more than one line; or
- More than one statement on a single line.

Forgetting a semicolon is a common syntax error.



All languages provide an entry point for program execution.

The **program entry point** tells the compiler where to find the first instruction of a program.

There can be only one program entry point at compilation time.

In C#, the simplest program entry point is defined as:

```
static void Main( ) { }
```

Our code goes in the block between the { and } braces after Main().

We can, for the moment, when creating a program, ignore all the other bits and just write our code inside this “main” block.

Later on, we'll understand all the parts.

# Writing & Compiling the Program

Programs are formed from source code: text written in a text file.

The text is written in a language the compiler understands (C#).

By convention, files containing C# code use the file extension `.cs`, such as `HelloWorld.cs`

In this course, we will use the command-line compiler provided with the Microsoft .NET Software Development Kit.

To compile the program at the command prompt, use the following command:

```
csc filename.cs
```

If we compile the Hello World program, what is our output?

On the assumption that there were no syntax errors, the compiler will run and there will be no error output.

You are left with a compiled, executable program.

If we made a mistake in writing the program, the compiler may identify it and report that to us.

[Demo: using the compiler; seeing successful/unsuccessful compiles]

If compilation did not succeed, we cannot run the program.  
Because no program was produced.

The compiler provides us with some error message(s) to identify why the compile was unsuccessful.

Read these messages carefully to find the error in the program.

Compiling again without fixing the error results in the same outcome.

After we correct the error, compile the program again.

If compilation succeeded, an executable file was generated and we can run this file.

On the command line, we start execution by using the name of the executable:

```
HelloWorld
```

When this program executes, it performs the following steps:

- 1 Starts executing instructions at the start of `Main( )`
- 2 Executes `Console.WriteLine("Hello World")` by displaying the “Hello World” text on the display
- 3 Finishes execution: returns control to the operating system

[Demo: running the program]

The implementation is straightforward in a high-level language.

It's easy for us to write, but for the computer it is complex.

It requires the execution of many machine language instructions.

Each character must be written to the screen at the right time.

Output statements equal many machine language instructions.



## Part II

# Simple Types

Flash back to high school algebra: variables.

Algebraic variables look like this:  $x$

We often **declare** a variable like so: Let  $x = 5$ .

Conceptually:  $x$  is a quantity and at the moment, the value of  $x$  is 5.

To conceptualize it visually, we have a box labelled  $x$ .  
The number in that box is 5.

Variables are changeable: later, we might say: Let  $x = 7$ .  
The number in the box changes to 7.

In algebra, you commonly saw expressions involving variables:  
 $y = x + 1$

We have a new box, labelled  $y$ , and we evaluate  $x + 1$ .  
Because  $x = 7$  and  $7 + 1 = 8$ ,  $y = 8$ .  
Write 8 into the box labelled  $y$ .

We have the same idea in programming: software variables.

Conceptually it's the same idea: we **declare** a variable.

When a variable is declared, the computer marks a memory location to represent that variable.

That memory location is accessible by the name of the variable.

This is like creating a box and labelling it  $x$  after we write “Let  $x = 5$ ”

When putting a value in the variable's box, we call that **assignment**.

Assignment can set or change the value of the variable.

Sometimes we can declare a variable to be **read-only**: we can set its value when we create it, but cannot change it after that.

In algebra, a variable  $x$  always contains a number (of some kind).

In some programming languages, when declaring a variable, it is necessary to specify a **Type**.

The type tells the compiler and computer what kind of value is stored in the box.

Example: declaring a variable as an **integer** (whole number).

Why does the type matter? Some possible answers:

Tell the compiler how much space it takes to store the value.

A big number, or one with high precision, takes more space to store.

Optimization: the compiler can translate your code into specialized instructions that operate on that data type rather than general ones.

Compile time checking: avoid assigning a “wrong” value to a variable.  
e.g., prevent assigning 1.5 to a variable that should be an integer.

(Some languages don't care & just figure things out at runtime.  
They do not benefit from this compile time checking.)

The equivalent to our algebraic statement “let  $x = 5$ ” in C# code:

```
int x = 5;
```

This statement follows the pattern *type identifier = value*;

`int` is the type; it's a language keyword indicating an integer variable.

`x` is the identifier; it's the name of the variable (and is case-sensitive).

`5` is a literal; it's the explicit value being stored in the variable `x`.



It's not strictly necessary to assign a value to `x` when it is declared.

```
int x;
```

is acceptable as a declaration.

At some later time, we can set the value of this variable with an assignment statement:

```
x = 3;
```

The first time a variable is assigned a value, we call that **initialization**.

It is often an error to use an uninitialized (not set) variable.

To declare a variable as read-only, we use the keyword **const**.

Example: `const int c = 0;`

If we mark a variable as `const`, it means that after assigning a value to that variable, we may not change it later.

A later attempt to assign `c = 1;` is a compile-time error.

C# has many different variable types.

Most of them refer to different types of numbers.

There are other values we can store in variables, but we'll examine numbers first.

Type	Description
byte	Unsigned 8-bit integer
ushort	Unsigned 16-bit integer
uint	Unsigned 32-bit integer
ulong	Unsigned 64-bit integer
sbyte	Signed 8-bit integer
short	Signed 16-bit integer
int	Signed 32-bit integer
long	Signed 64-bit integer
float	Signed 32-bit floating point
double	Signed 64-bit floating point
decimal	Signed 128-bit high precision floating point

# C# Built-In Numeric Types

Type	Epsilon Value (Min. Change)	Minimum Value	Maximum Value
byte	$\pm 1$	0	+255
ushort	$\pm 1$	0	+65,535
uint	$\pm 1$	0	+4,294,967,295
ulong	$\pm 1$	0	+18,446,744,073,709,551,615
sbyte	$\pm 1$	-128	+127
short	$\pm 1$	-32,768	+32,767
int	$\pm 1$	-2,147,483,648	+2,147,483,647
long	$\pm 1$	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807
float	$\pm 1.5 \times 10^{-45}$	$-3.4 \times 10^{+38}$	$+3.4 \times 10^{+38}$
double	$\pm 5.0 \times 10^{-324}$	$-1.7 \times 10^{+308}$	$+1.7 \times 10^{+308}$
decimal	$\pm 1.0 \times 10^{-28}$	$-7.9 \times 10^{+28}$	$+7.9 \times 10^{+28}$

When assigning the numeric types in code, we often use literals like 5.

Literals may need a letter (or two) after them to tell the compiler the format of the literal.

If you don't specify, the compiler will try to guess what you meant.

- Unsigned numbers are followed by U or u  
Examples: 5U, 5u, etc.
- Long numbers are followed by L or l  
Examples: 5L, 5l, etc.
- Unsigned long numbers are followed by UL or ul  
Examples: 5UL, 5ul, etc.
- Single-precision, floating point numbers are followed by F or f  
Examples: 362.5F, 362.5f, etc.
- Double-precision, floating point numbers are followed by D or d  
Examples: 362.5D, 362.5d, etc.
- Decimal numbers are followed by M or m  
Examples: 362.5M, 362.5m, etc.

We can also express numbers with powers of 10.

The double-precision floating point value for  $3.625 \times 10^2$  can be expressed as 3.625E2D or 3.625e2d.

Assignment statements would therefore look like:

```
long example1 = 9001;
```

```
float example2 = 365.25f;
```

```
double example3 = 75.5e4d;
```



# Too Many Number Types!

There are a lot of choices, but it turns out in practice you are likely to use just a small number of them.

Typically `int` and `double` are the most common choices for integer and real (floating point) numbers.

Other types have their applications, however.

Aside from numbers, there are 2 other simple types to examine today.

The `bool` (boolean variable); and  
The `char` (character).

A `bool` defines a Boolean variable.

A Boolean variable can only have one of two possible values:  
`true` or `false`.

Unlike C++, Boolean variables may not be assigned values of 0 or 1.

Example: `bool printingEnabled = true;`

A `char` defines a character variable.

A character is a single symbol from an alphabet, like 'a', 'z', or a Japanese kanji.

Characters are stored as 16-bit quantities in Unicode format.

A `char` literal is enclosed in single quotation characters (').

Some examples of valid `char` declarations:

```
char myChar1 = 'X'; - Character Literal  
char myChar2 = '\x0058'; - Hexadecimal  
char myChar3 = '\u0058'; - Unicode
```

```
using System;

class HelloWorld
{
    static void Main( )
    {
        char myChar1 = 'X';
        char myChar2 = '\x0058';
        char myChar3 = '\u0058';
        Console.WriteLine( myChar1 );
        Console.WriteLine( myChar2 );
        Console.WriteLine( myChar3 );
    }
}
```

[In-Class Demo: the output of this program]

This is the second time (after the Hello World) program that we've seen console output (`Console.WriteLine` statements).

In the next lecture, we'll look at how console input and output work.