C# Fundamentals

By: Mosh Hamedani

Primitive Types and Expressions

Variables and Constants

A variable is a name that we give to a storage location in memory. We use variables to store temporary values in memory.

A constant is a value that cannot be changed. We use constants in situations where we need to ensure that a value does not change. For example, if you're working on a program where you need to calculate the area of a circle, you need to use the Pi number (3.14). You can define Pi as a constant to ensure you don't accidentally change the value of Pi in computations.

To define a variable, we specify a type and an identifier:

```
int number;
```

Here, int represents the integer type, which takes 4 bytes of memory. You can find the most frequently used data types in the next section.

Number is the identifier for our variable. We can optionally set the value of a variable upon declaration. This is called “initializing a variable”:

```
int number = 5;
```

Remember: in C#, you cannot read the value of a variable unless you have set it before.

Primitive Types

Type	Bytes
byte	1
short	2
int	4
long	8
float	4
double	8
decimal	16
bool	1
char	2

These types have an equivalent type in .NET Framework. So when you compile your application, the compiler maps your types to the underlying type in .NET Framework.

C# Type	.NET Type
byte	Byte
short	Int16
int	Int32
long	Int64
float	Single
double	Double
decimal	Decimal
bool	Boolean
char	Char

The easy way to memorize Int* types is by remembering the number of bytes each type uses. For example, a "short" takes 2 bytes. We have 8 bits in each byte. So a "short" is 16 bytes, hence the underlying .NET type is Int16.

We also have a few non-primitive types (string, array, class, struct) that I'll discuss in the following sections.

Scope

Scope determines where a value has meaning and is accessible. A variable has a scope in the block it is defined and in any child blocks. But it is not accessible outside that block. A block is indicated by curly braces ({ }).

Overflowing

Each type, depending on the number of types allocated to it, can store a range of values. If we store a value in a variable, but that value exceeds the boundary of values for the underlying type, overflow happens. For example, we can store any values between 0 and 255 in a byte. If the value of a byte exceeds this boundary during computations, overflow happens. Here is an example:

```
byte b = 255;

b = b + 1;
```

As a result of the second line, the value of b will be 0.

Type Conversion

There are times that you need to temporarily convert the value of a variable to a different type. Note that this conversion does not impact the original variable since C# is a *statically-typed language*, which in simple term means: once you declare the type of a variable, you cannot change it. But you may need to convert the "value" of a variable as part of assigning that value to a variable of a different type.

There are a few conversion scenarios:

If types are compatible (e.g. integral numbers and real numbers) and the target type is bigger, you don't need to do anything. The value will be automatically converted by the runtime and stored in the target type.

```
byte b = 1;

int i = b;
```

Here because b is a byte and takes only 1 byte of memory, we can easily convert it to an int, which takes 4 bytes. So we don't need to do anything.

If the target type, however, is smaller than the source type, the compiler will generate an error. The reason for that is that overflow may happen as part of the conversion. For example, if you have an int with the value 1000, you cannot store it in a byte because the max value a byte can store is 255. In this case, some of the bits will be lost in memory. And that's the reason compiler

warns you about these scenarios. If you're sure that no bits will be lost as part of the conversion, you can tell the compiler that you're aware of the overflow and would still like the conversion to happen. In this case, you use a cast:

```
int i = 1;

byte b = (byte)i;
```

In this example, our int holds the value 1, which can perfectly be stored in a byte. So, we use a cast to tell the compiler to ignore the overflow. A cast means prefixing the variable with the target type. So here we are casting the variable i to a byte in the second line.

Finally, if the source and target type are not compatible (eg a string and a number), you need to use the Convert class.

```
string s = “1234”;

int i = Convert.ToInt32(s);
```

Convert class has a number of methods for converting values to various types.

Operators

In C# we have 4 types of operators:

- **Arithmetic**: used for computations
- **Comparison**: used for comparing values in boolean expressions
- **Logical**: represent logical AND, OR and NOT
- **Bitwise**: represent bitwise AND, OR and NOT

Arithmetic operators:

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Remainder of Division
++	Increment by 1
—	Decrement by 1

Comparison operators:

Operator	Description
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
.==	Equal
!=	Not equal

Logical operators:

Operator	Description
&&	Logical AND
ll	Logical OR
!	Logical NOT

Bitwise operators:

Operator	Description
&	Bitwise AND
l	Bitwise OR

Comments

A comment is text that we put in our code to improve readability and maintainability. There are two common styles to write comments in C# programs:

A single-line comment: prefix it with double slash:

```
// This is a one-line comment
```

A multi-line comment: use /* to start the comment and */ to finish it:

```
/* My comment starts here

   and finises here.

*/
```

As a rule for clean coding, use comments only to explain whys, not whats. Don't explain what the code is doing. Your code should be clean and self explanatory that doesn't need a comment. But if you had a reason for writing the code that way, or there were constraints at the time you were working on this piece of code, always explain them using comments.