Working with Data Types

Data typing is a feature of the strongly typed programming languages. It defines a context for every value within a given program. The data type of a value helps determine the set of operations we're able to do with it as well as its structure when the value is of a complex type. Strongly typed programming languages are strict and their code is validated before compilation not only for syntax errors but for semantic ones as well. That is why strongly typed programming languages help identify bugs at compilation.

Data types in C#

The C# language is a strongly typed language. The built-in types are described in the following table:

C# Type	Description	.NET Data Type
bool	**1 bit - a boolean value (either true or false)**	System.Boolean
byte	**8 bit – integer value from 0 to 255**	System.Byte
sbyte	**8 bit – signed integer value from -128 to 127**	System.SByte
char	**16 bit character**	System.Char
decimal	**A floating point value – from ±1.0 × 10^{-28} to ±7.9 × 10^{28}**	System.Decimal
double	**A floating point value – from ±5.0 × 10^{-324} to ±1.7 × 10^{308}**	System.Double
float	**A floating point value – from** -3.4 × 10^{38} to +3.4 × 10^{38}	System.Single
int	**32 bit integer value – from** -2,147,483,648 to 2,147,483,647	System.Int32
uint	**32 bit unsigned integer value – from** 0 to 4,294,967,295	System.UInt32
long	**64 bit integer value – from** –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	System.Int64
ulong	**64 bit unsigned integer value – from** 0 to 18,446,744,073,709,551,615	System.UInt64
short	**16 bit integer value – from** -32,768 to 32,767	System.Int16
ushort	**16 bit unsigned integer value – from** 0 to 65,535	System.UInt16
string	**A string of characters**	System.String

Data types in .NET Framework languages are coordinated with a set of rules that define how types are used within .NET Framework. This is crucial for supporting multiple programming languages within the framework. This set of rules is called the Common Type System. In order to facilitate the developers that are beginning with C# the language provides labels for the main built-in types. All labels are equivalent to their corresponding .NET types. You can find the .NET types as well as their corresponding labels in the table above.

Variables and constants

When working with a programming language we need a common method of identifying values, constants and other functional objects. That is handled by identifiers. An identifier is a sequence of letters and digits that identifies uniquely a given entity within the program. Identifiers in C# can contain any sequence of letters, digits and the underscore symbol. The only restriction is that an identifier can't start with a digit. The most common examples of identifiers are variables and constants.

A variable is a value container within the programming language. As a strongly typed language C# requires each variable to be associated with a data type thus the syntax of declaring a variable requires an explicit data type declaration. Each variable has its own memory block big enough to fit the maximum value for its data type (see the table above).

In programming a constant is an identifier whose value is assigned at compile time and can't be altered during the execution of the program.

Declaration and default values

Let's see how declaring variables works:

```
int a;
System.Int32 b;
int c = 10;
long d, e, f;
ushort g = 5, h, i = 0;
```

We have several ways of declaring a variable. The standard syntax is:

```
DataType VariableName;
```

In the first two declarations we have integer variables. As mentioned above int and System.Int32 are equivalent. When a variable is declared it is assigned the default value of the data type. For numeric types the default value is 0.

The third declaration includes explicit value assignment. The integer variable "c" now has value 10. You can declare multiple variables when they are of a common type as per declaration of "d", "e" and "f". When declaring multiple variables of a common type you can still assign values on declaration as per declaration of "g" and "i".

Let's see how declaring constants works:

```
const int a = 5;
```

The main difference between variable and constant declaration is that by syntax the constant needs to be assigned a value on declaration. That is because the value of the constant is assigned on compilation therefore it must happen in the declaration itself.

Assigning values

We have discussed variable declaration and assignment of values on declaration. In C# the assignment operator is "=". Depending on the values we have different ways to state them inside our code. We have three types of constant values – integers, floating point values and strings.

We have seen how integer values are stated within the code. This is done directly without any qualifiers by placing the value in the code as a sequence of digits without any separators or formatting.

Working with floating point values on the other hand is not that straight forward. The floating point in this value type is specified by the point symbol again no white spaces are permitted and no formatting is applied. By default a floating point value is of type double. The syntax in the example below is correct.

```
double a = 0.5;
```

If we do that for a variable of type float we will get a syntax error. In order to keep the correct syntax we must specify the type of the value by a letter after it. The letters that help identify the type of the value are “F” for float, “M” for decimal and “D” for double. Let’s have an example:

```
double a = 0.5D;
decimal b = 0.5M;
float c = 0.5F;
```

The third type that we’re going to discuss is of the character values. Characters are single Unicode symbols. Their underlying value is an unsigned 16 bit integer. The character values in C# are qualified with single quotation marks. That way C# does not confuse them with identifiers. Let’s have an example:

```
char a = 'A';
char b = '\u0041';
char c = '\x0041';
```

There are three ways to define a character value. All values that are based on characters (char and string) are called literals. The first and most common way to use a character in the code is by placing a symbol in single quotes. That is how we have assigned the value of the variable “a”. The second way is by specifying the Unicode code of the symbol. This is done in the declaration of the “b” variable. The third way is by giving the hexadecimal code of the symbol. This is done in the declaration of “c”. A table of all the Unicode codes is publicly available at http://www.unicode.org/charts/.

The last type that we’re discussing is of the string values. A string is a sequence of characters. Strings are qualified by semicolons for the same reason as the characters. Let’s have an example:

```
string a = "This is a string";
```

The main difficulty when using qualifiers is how to use the qualifier itself within the sting or as a character. Here are two examples:

```
char a = '\'';
string b = "\"This is quoted text\"";
```

The slash symbol is called an escape character. It’s used to give the following character a special meaning. Here is a table of the special characters that we can use within a string value:

C# Type	Description
\'	allows the usage of the ' character in a character literal
\"	allows the use of the " symbol in a string literal
\\	allows the use of the \ symbol in a string or character literal
\a	makes the speaker beep
\b	deletes the preceding character in a string literal
\n	new line
\r	carriage return
\t	horizontal tabulation

In the cases where we do not want the special character functionality within a string we use the “@” sign as a prefix to the string. For example:

```
string a = @"\This is quoted text\";
```

Will place the string “\This is a quoted text\” in the variable “a”. The “\T” will not be read as a horizontal tabulation. While in this mode we can use " sign within our string in the following way:

```
string a = @"""This is quoted text""";
```

Assigning a value to a variable can be done later in the code. As mentioned above when a variable value is not supplied at its declaration the variable is assigned the default value for the type. When dealing with numerical values the default is 0. Assigning values at a later point is done in the following way:

```
int a;
a = 5;
int b, c;
b = c = 10;
```

In this case right after the execution of the first line the variable’s value will be 0. After executing the second variable will hold the value 5. The next two lines illustrate how the assignment operator works. It assigns the value to the variable and then returns the new variable value as a result. This behavior enables us to use is in pipelines. After the declaration “b” and “c” both store the value 0. After the assignment call both “c” and “b” will hold the value 10.

Scope

The scope of a variable defines the extent of its visibility within the program. Whenever a variable is declared in a code block it is usually visible within that code block only. When a variable is out of scope it can’t be referenced thus can’t be used.

The term Scope applies to functions, properties, fields and other functional elements of programming languages. We will discuss it in latter chapters.

Type casting and converting between types

Values can be transformed between types. This process is called type casting. Type casting is applicable only between compatible data types. Let's take a look at the syntax:

```
DataType1 variable1 = value1;

DataType2 variable2 = (DataType2)Variable1;
```

Let's have a variable *variable1* of type *DataType1* with value *value1* and a variable *variable2* of type *DataType2*. We want to place *value1* into *variable2*. We can't do that directly because the two variables have different types. That is where type casting comes into place. By placing the destination data type in front of the variable whose value is to be casted we can transform it and place it in the desired destination. Let's have an example:

```
int a = 5;
decimal b = (decimal)a;
```

We're converting the int value to a decimal before placing it in the variable "b". In this example we're casting a value to a broader type (decimal can store both the minimum and maximum int value). Let's consider the opposite case where we're casting from decimal to integer:

```
decimal a = 5.5M;
int b = (int)a;
```

The problem here is that we're casting from a broader type to a more limited type. In this specific case we have values after the floating point that can't be stored by an integer variable. This is casting with loss of data. The values after the floating point will be lost. After the execution of the second line of code the "b" variable will hold the value 5.

There are two ways of type casting – explicit and implicit. Both examples were of explicit type casting. In this type of casting the destination type is specified explicitly in front of the value that is to be casted. The other method is applicable only when casting from a more limited type to a broader type (as in the first casting example). In this type of casting we do not need to specify the target type. For example the following code is correct:

```
int a = 5;
decimal b = a;
```

Here we leave the compiler to make the cast for us. This is allowed only because the destination type is broader than the original and there is no possible data loss. In all other cases the compiler will return an error and require the developer to verify the casting by specifying the destination type explicitly.

Let's end the subject with a counter-example – casting between incompatible types:

```
decimal a = 5.5M;
string b = (string)a;
```

Compiling this code will raise syntax error stating that you can't convert decimal to string.

While similar to type casting by its nature, converting between types is a very different process. The .NET Framework supplies a built-in tool for data conversion called Converter. Converter is used to transform data across incompatible types. Let’s start where we left off:

```
decimal a = 5.5M;
string b = Convert.ToString(a);
```

The second line in this example will convert the decimal value to a string and then assign it to the “b” variable. The Convert class has the ability to convert between the basic types. Conversion works by passing the value that is to be converted to the desired functionality (in the example above we’re passing the variable “a”). Here is a list of its basic functionality:

Method	Description
ToBoolean	converts the value to a bool
ToByte	converts the value to a byte
ToChar	converts the value to a char
ToDecimal	converts the value to a decimal
ToDouble	converts the value to a double
ToInt16	converts the value to a short
ToInt32	converts the value to an int
ToInt64	converts the value to a long
ToSByte	converts the value to a sbyle
ToSingle	converts the value to a float
ToString	converts the value to a string
ToUInt16	converts the value to an ushort
ToUInt32	converts the value to a uint
ToUInt64	converts the value to a ulong

Another way to convert between types is the Parse method. Every built-in numeric type has a method called Parse which converts any kind of value to a numeric. Let’s have an example:

```
string a = "100";
int b = int.Parse(a);
```

The string “a” is converted to an integer value by the Parse method of the int type.

The third way of type conversion that we’re going to discuss here is the ToString method. This method converts any value to a string and is present in every type in C#. Let’s see an example:

```
int a = 100;
string b = a.ToString();
```

This will convert to value of “a” to a string and then assign it to the variable “b”.

Formatting the output

Depending on the type the ToString method can accept formatting. Let’s see an example of the usage of ToString with numerical types:

```
decimal a = 100.50M;
string b = a.ToString("0.#");
Console.WriteLine(b);
```

In the formatting syntax the "#" symbol is replaced with the corresponding digit in the number if one is present. If no digit is present then no digit appears in the result string (tailing zeros in the fraction part are not considered).

The “0” symbol does a very similar thing – as “#” it is replaced by the corresponding digit in the number. The difference is that if no digit is present then “0” appears in the result string. These formatting symbols can be used to limit the number of outputted digits after the floating point. For example “0.##” will output only the first two digits after the floating point.

In our case the formatting will show us the integer part of the number and if there are any digits after the floating point the first one of them will be shown with a floating point separator. It will show any number of digits before the floating point and any number of digits after it. After the execution of line two the value of “b” is “100.5” and that’s what will be written to the standard output.

Let’s consider the case where we want to see only the first two digits after the floating point. If there are no present digits then we want to see zeros.

```
decimal a = 100.5M;
string b = a.ToString("0.00");
Console.WriteLine(b);
```

This example will output “100.50” to the standard output. Note that the second zero is added by the formatting because no digit was found for this position in the formatting string.

This type of formatting is called custom formatting. Custom formatting is not the only option in C#. There are standard number formats predefined in the language. Let’s have an example:

```
decimal a = 100.50M;
string b = a.ToString("F");
Console.WriteLine(b);
```

The “F” sign stands for floating point number formatting. Standard formatting also supports limitations of the number of outputted digits after the floating point. This is done by adding an integer value after the formatter as per the next example:

```
decimal a = 100.551M;
string b = a.ToString("F2");
Console.WriteLine(b);
```

This will output to the standard output the string “100.55” because the formatting explicitly states that only two digits should be outputted after the floating point.

Let’s take a look at the list of formatting options supported in C# for standard and custom formatting:

Standard formatting

Format specifier	Description	Example
"C" or "c"	Result: A currency value. Precision specifier: Number of decimal digits.	123.456 ("C") -> $123.46 -123.456 ("C3") -> ($123.456)
"D" or "d"	Result: Integer digits with optional negative sign. Precision specifier: Minimum number of digits.	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E" or "e"	Result: Exponential notation. Precision specifier: Number of decimal digits. Default precision specifier: 6.	1052.0329112756 ("E") -> 1.052033E+003 -1052.0329112756 ("e2") -> -1.05e+003
"F" or "f"	Result: Integral and decimal digits with optional negative sign. Precision specifier: Number of decimal digits. Default precision specifier: Defined	1234.567 ("F") -> 1234.57 1234 ("F1") -> 1234.0 -1234.56 ("F4") -> -1234.5600
"G" or "g"	Result: The most compact of either fixed-point or scientific notation. Precision specifier: Number of significant digits.	-123.456 ("G") -> -123.456 123.4546 ("G4") -> 123.5 -1.234567890e-25 ("G") -> -1.23456789E-25
"N" or "n"	Result: Integral and decimal digits, group separators, and a decimal separator with optional negative sign. Precision specifier: Desired number of decimal places.	1234.567 ("N") -> 1,234.57 1234 ("N1") -> 1,234.0 -1234.56 ("N3") -> -1,234.560
"P" or "p"	Result: Number multiplied by 100 and displayed with a percent symbol. Precision specifier: Desired number of decimal places.	1 ("P") -> 100.00 % -0.39678 ("P1") -> -39.7 %
"R" or "r"	Result: A string that can round-trip to an identical number.	123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.1234567
"X" or "x"	Result: A hexadecimal string. Precision specifier: Number of digits in the result string.	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF
Any other single character	Result: Throws a FormatException at run time.	

Custom Formatting

Format specifier	Description	Example
"0"	Replaces the zero with the corresponding digit if one is present; otherwise, zero appears in the result string.	1234.5678 ("00000") -> 01235 0.45678 ("0.00") -> 0.46
"#"	Replaces the "#" symbol with the corresponding digit if one is present; otherwise, no digit appears in the result string.	1234.5678 ("#####") -> 1235 0.45678 ("#.##") -> .46
"."	Determines the location of the decimal separator in the result string.	0.45678 ("0.00") -> 0.46
","	Serves as both a group separator and a number scaling specifier. As a group separator, it inserts a localized group separator character between each group. As a number scaling specifier, it divides a number by 1000 for each comma specified.	Group separator specifier: 2147483647 ("##,#") -> 2,147,483,647 Scaling specifier: 2147483647 ("#,#,,") -> 2,147
"%"	Multiplies a number by 100 and inserts a localized percentage symbol in the result string.	0.3697 ("%#0.00") -> %36.97 0.3697 ("##.0 %") -> 37.0 %
"‰"	Multiplies a number by 1000 and inserts a localized per mille symbol in the result string.	0.03697 ("#0.00‰") -> 36.97‰
"E0" "E+0" "E-0" "e0" "e+0" "e-0"	If followed by at least one 0 (zero), formats the result using exponential notation. The case of "E" or "e" indicates the case of the exponent symbol in the result string. The number of zeros following the "E" or "e" character determines the minimum number of digits in the exponent. A plus sign (+) indicates that a sign character always precedes the exponent. A minus sign (-) indicates that a sign character precedes only negative exponents.	987654 ("#0.0e0") -> 98.8e4 1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16 ("0.0e+00") -> 1.9e-16
\	Causes the next character to be interpreted as a literal rather than as a custom format specifier.	987654 ("\###00\#") -> #987654#
'*string*' "*string*"	Indicates that the enclosed characters should be copied to the result string unchanged.	68 ("# ' degrees'") -> 68 degrees 68 ("#' degrees'") -> 68 degrees
;	Defines sections with separate format strings for positive, negative, and zero numbers.	12.345 ("#0.0#;(#0.0#);-\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -> 0.0 -12.345 ("#0.0#;(#0.0#)") -> (12.35)
Other	The character is copied to the result string unchanged.	

Value types and reference types

Data types in C# are divided into two main branches – the value types and the reference types. They differ in the way they are instantiated in the code and the way the assignment operator works for them.

While the value types work with the value itself, the reference types work with a reference to the value (generally called instance) that they work with. This is where the assignment process changes – the value types make a copy of the value that they are assigned while reference types copy the reference to the value.

Let's have a small example illustrating how reference and value types work:

```
            int a;
            object b;
```

In the code sample above we define two variables.

The first – “a” is of type “int”. The “int” is a value type and as all built-in value types its variables are initialized on declaration. Right now “a” has the default value for our “int” type which is 0.

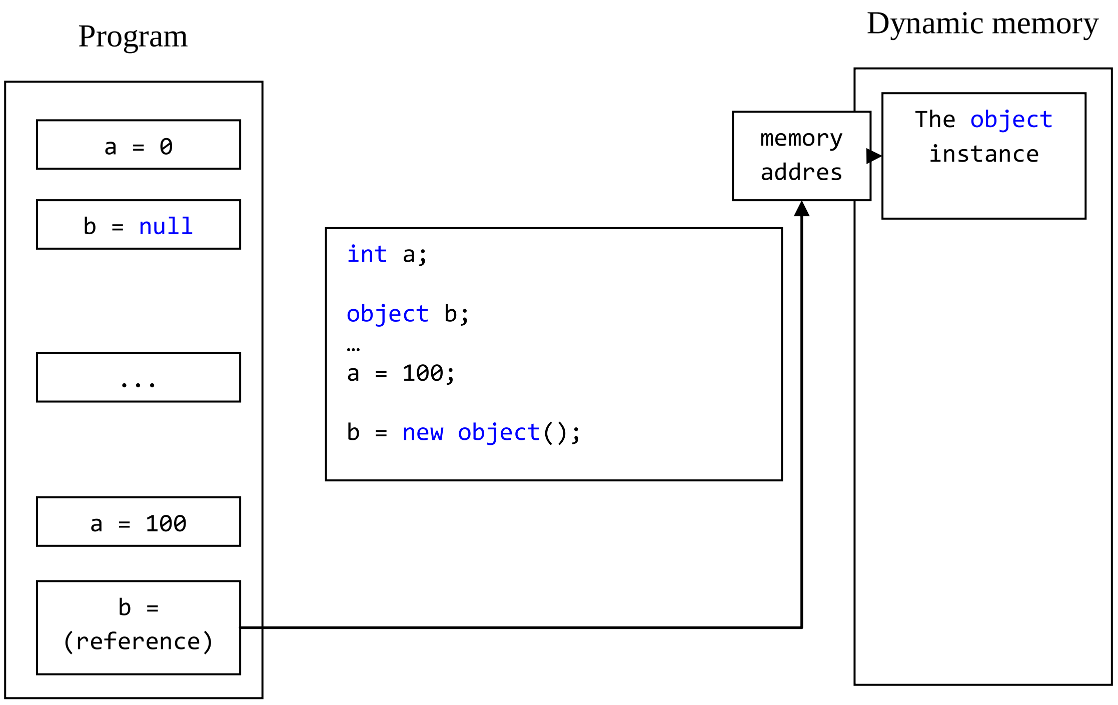
The second – “b” is of type “object”. The “object” is a reference type and as all reference types its variables need a value (an instance) to point to. Since “b” is not yet assigned it does not point anywhere in the memory thus has the service value “null”. This value is applied as a default value to all reference type variables that don’t point anywhere (don’t reference any instance).

Let’s extend our example:

```
            int a;
            object b;

            a = 100;
            b = new object();
```

We first declare the two variables – “a” with default value of 0 and “b” that still doesn’t point anywhere and has default value “null”. On the next line we assign “a” with 100. Next we create an instance of the type “object” and assign the reference to “b”.



The difference between reference types and value types will be discussed more at a latter chapter.