C# Notes

# 9.6.2017

DocTags

Int + long = long

Short + int = int

(primitive types – known by the language)

Integrals

|  |  |  |
| --- | --- | --- |
| Signed | sbyte | 8 |
|  | short | 16 |
|  | int | 32 |
|  | long | 64 |
| Unsigned | byte | 8 |
|  | ushort | 16 |
|  | uint | 32 |
|  | ulong | 64 |
| Floats | float | 4 |
|  | double | 8 |
|  | decimal | Money |
| String | char | 2 |
|  | bool | True/false |
|  | string |  |
|  |  |  |

Add ‘L’ to the end to set a variable type to a Long (456L)

DateTime

Timespan

GUID

(The previous three types are NOT primitive types. They are known by the framework). Notice they are capitalized, whereas the primitive types are not

Look up Type Aliasing

Math.Ceiling(5.75) – takes value and rounds up to next whole integer. This will equal 6.

Default(int) – gives the default for a type

Int hours;

Hours = default

(the previous code will default hours to 0)

{

};

Block statement. Defines scope.

(put semicolons only on statements, expressions or code blocks. Do not put them on functions)

Strings

String literals in C# are double quotes.

Single quotes are characters.

String name = “John”;

Add to a string (concatenate):

Name = name + “ Williams”;

Assign new value (string copy)

Name = “Hello”;

String is part of the .Net framework. It works the same in all .Net languages. But it does not work the same in all other languages (outside of the .Net Framework).

Unicode is how we represent all languages around the world.

Unicode is 2 bytes.

‘A’ = 0065

With Unicode, you can include any language in the world. The program will work as long as the OS hosting the application contains the included language being used.

Immutable (cannot be changed)

Strings are immutable. Once assigned a value, it can never be changed.

Strings don’t go away, they always point to the same location in memory. (look this up)

Lookup Call Stack

Call Stack – contains local variables

Primitives are stored on the stack.

Strings are stored on the stack, but reference the actual value elsewhere. String assignments work by moving a pointer to a new value; but, the original value remains in memory.

String variables are case sensitive.

Bool areEqual = name = “Hello”;

Bool areNotEqual = name != “Hello”;

Escape Sequence – inside of a string literal, escape sequences have special meaning.

\n = new line

\t = tab

\a = alarm

\x## - hexadecimal

File Paths (you have to escape the back slash in file paths).

“C:\\temp\\test.txt”

Verbatim Strings – no escape sequences needed or allowed. Predominate use is in paths.

@”C:\temp\test.txt”

No limit to number of concatenations you can do with strings.

//Option 1

String names = “John” + “ William” + “ Murphy” + “ Charles” + “ Henry”;

Inefficient, but works. This will create multiple memory spaces.

//Option 2

StringBuilder builder = new StringBuilder();

Builder.append(“John”);

Builder.apppend(“Williams”);

Builder.Append(“Murphy”);

String names2 = builder.ToString();

StringBuilder does NOT create multiple memory spaces. It maximizes memory.

//Option 3

String names = string.Concat(“John” + “ William” + “ Murphy” + “ Charles” + “ Henry”);

Pass each string as a parameter… behind the scenes, it will concatenate them together. Behind the scenes, it uses StringBuilder.

String.Concat is considered to be most readable and efficient for concatenating strings. It will take string assignments like option 1 and will convert it behind the scenes to string.concat.

String.concat is only efficient until you get to 6 strings. StringBuilder would be used for variables containing more than 6 strings.

To combine strings that is more than just string data, you run into string formatting.

String formatting

String name = “John”;

Int hours = 10;

//John worked 10 hours

String format1 = name + “ worked “ + hours + “ hours”;

(this won’t work because you are mixing in an integer. You can format the hours variable to a string by using toString().

String format1 = name + “ worked “ + hours.ToString() + “ hours”;

A **format string** is generally a string literal that uses place holders that get replaced by variables at run time.

String.format(“{0} worked for {1} hours”, name, hours);

Format string is more clear than using string concatenation. They are not compiled until run time!

Conditional If (More concise) (look up if it is evaluated at compile time or not)

The two expressions MUST BE THE SAME TYPE IN A CONDITIONAL IF. If they are not, you will have to type cast them.

Eb ? Et : Ef

isPassing ? “Pass” : “Fail”

String.Compare() – can remove syntax evaluations.

String.Compare(input, “A”, true); if value 0, then equal. If not 0, then not equal.

If (String.Compare(input, “A”, true) == 0) //this is true

Return ‘A’;

String.StartsWith(string) 🡪 bool

String.EndsWith(string) 🡪bool

String.Trim(string) 🡪 string

String.PadLeft(char, int) 🡪 string

String.PadRight(char, int) 🡪string

Look up rectangular select (select beginning, hold down alt, mouse and move to last location)

Different ways of checking for an empty string:

String str;

1. Str != String.Empty; (this is a string literal)
2. Str != “”;
3. Str.Length =- 0; (have to check first if it is null… so str != null && str.length != 0;)
4. String.IsNullOrEmpty(str); (checks for null like option 3)

ReadLine() returns a string

String.Format

String.Interpolation

TryParse(string) 🡪 evaluates and returns false if not able to parse

**Parameter Kinds**

Input Parameter [int param] – also referred to as ‘passed by value’. Doesn’t change original variable.

Input/Output Parameter [ref int param] – passed by reference. Changes original variable (rarely used)

Output Parameter – [out int param]

1. function must write to it
2. function can read after write
3. The Caller passes no value

Inline Variable Declaration –

Var is ONLY valid in local declarations. Cannot use it outside of function. Var is known as Type Inferencing. It simply looks at the right side assignment and set variable to that type. For example, Console.ReadLine() returns a string. So:

Var a = Console.ReadLine().

Variable ‘a’ will be a string.

CTS – Common Type System – The foundation that allows us to do a lot of stuff in .NET

Fundamental concept of this is that there is one base type that all other types derive from. That one type is Object.

Object is the .NET framework base type for everything. Strings, Ints, decimals, etc. all come from the Object type.

Every type is either a reference type or a value type. They have completely different purposes.

All primitives are value types. Enumerations are value types. **A Class is a reference type.** Structs are value types. Strings are reference types because they are immutable.

If you look at bools or ints, etc. variables, they will show as structs… which are value types.

What distinguishes value types from reference types – value types have the actual value stored. Reference Types have implementation details (pointers… but not called pointers in C#) stored instead of the actual value.

Value types are always designed to be small values. Almost always less than 64 bytes. This is to keep the stack small.

The call stack sits in memory. Value Types always have a default value of zero.

Reference types point to somewhere in memory. Refernce types always start out as Null… meaning they don’t point to anything. You must always handle null with reference types. Value types can never be null… so you don’t have to worry about checking for null for them.

You have to explicitly give a reference type a value. To do this, you use the New keyword. New requires the type. It allocates memory for the new object.

Equality works different on reference types than with value types. Value semantics mean it looks at the actual value you are storing. Reference Types follow Reference Semantics. Reference Semantics compare the reference value (the pointer). If they point to the same reference in memory, then they are equal. If not, they are never equal.