C# Basics Cheat Sheet (1 of 4)

Introduction to C#

The C# language was developed by Microsoft for the .NET framework. C# is a completely-rewritten language based on C Language and C++ Language. It is a general-purpose, object-oriented, type-safe platform-neutral language that works with the .NET Framework.

Visual Studio (VS)

Visual Studio Community 2017 is a free download from Microsoft. To create a new project, go to File ➤ New ➤ Project in Visual Studio. From there select the Visual C# template type in the left frame. Then select the Console App template in the right frame. At the bottom of the window configure the name and location of the project. Click OK and the project wizard will create your project.

C# Hello World (at the Console)

```
using System;
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
            /* this comment in C# is ignored by compiler */
            /* a multi-line comment
               that is ignored by the compiler*/
        }
    }
}
```

Ctrl+F5 will run the program without the debug mode. The reason why you do not want to choose the Start Debugging command (**F5**) here is because the console window will then close as soon as the program has finished executing, unless you use `Console.ReadKey();` at the end.

There are several methods and properties of console. You can change colors and put a Title on the console. Add this to the Main() to use your namespace, which may be your solution and project name also.

```
Type myType = typeof(Program);
Console.Title = myType.Namespace;
Console.ForegroundColor = ConsoleColor.Red;
Console.WindowWidth = 180;  // max might be 213 (180 is very wide)
```

A Few Code Snippets in VS

Code Snippet	Description
cw	`Console.WriteLine()`
prop	`public int MyProperty { get; set; }`
ctor	Constructor
Ctrl+K+C/Ctrl+K+U	Comment & un-comment a selected code block
F12	Go to Definition

ReSharper is a plug-in for Visual Studio that adds many code navigation and editing features. It finds compiler errors, runtime errors, redundancies, and code smells right as you type, suggesting intelligent corrections for them.

Common Language Runtime (CLR)

A core component of the .NET Framework is the CLR, which sits on top of the operating system and manages program execution. You use the .NET tools (VS, compiler, debugger, ASP and WCF) to produce compiled code that uses the Base Class Library (BCL) that are all used by the CLR. The compiler for a .NET language takes a source code (C# code and others) file and produces an output file called an assembly (EXE or DLL), which isn't native machine code but contains an intermediate language called the Common Intermediate Language (CIL), and metadata. The program's CIL isn't compiled to native machine code until it's called to run. At run time, the CLR checks security, allocates space in memory and sends the assembly's executable code to its just-in-time (JIT) compiler, which compiles portions of it to native (machine) code. Once the CIL is compiled to native code, the CLR manages it as it runs, performing such tasks as releasing orphaned memory, checking array bounds, checking parameter types, and managing exceptions. Compilation to native code occurs at run time. In summary, the steps are: C# code ➤assembly (exe or dll) & BCL ➤ CLR & JIT compiler ➤machine code ➤ operating system ➤ machine.

Variable Declaration and Assignment

In C#, a variable must be declared (created) before it can be used. To declare a variable, you start with the data type you want it to hold followed by a variable name. A value is assigned to the variable by using the equals sign, which is the assignment operator (**=**). The variable then becomes defined or initialized. Although seemingly simple, this concept becomes important to understand when we later talk about instantiating a class to create a concrete object with the new keyword.

Data Types

A primitive is a C# built-in type. A string is not a primitive type but it is a built-in type.

Primitive	Bytes	Suffix	Range	Sys Type
bool	1		True or False	Boolean
char	2		Unicode	Char
byte	1		0 to 255	Byte
sbyte	1		-128 to 127	SByte
short	2		-32,768 to 32,767	Int16
int	4		-2^{31} to $2^{31}-1$	Int32
long	8	L	-2^{63} to $2^{63}-1$	Int64
ushort	2		0 to $2^{16}-1$	UInt16
uint	4	U	0 to $2^{32}-1$	UInt32
ulong	8	UL	0 to $2^{64}-1$	UInt64
float	4	F	$+-1.5 \times 10^{-45}$ to $+-3.4 \times 10^{38}$	Single
double	8	D	$+-5.0 \times 10^{-324}$ to $+-1.7 \times 10^{308}$	Double
decimal	16	M	$+-1.0 \times 10^{-28}$ to $+-7.9 \times 10^{28}$	Decimal

The numeric suffixes listed in the preceding table explicitly define the type of a literal. By default, the compiler infers a numeric literal to be either of type double or an integral type:

• If the literal contains a decimal point or the exponential symbol (E), it is a double.

• Otherwise, the literal's type is the first type in this list that can fit the literal's value: int, uint, long, and ulong.

- Integral Signed (sbyte, short, int, long)
- Integral Unsigned (byte, ushort, uint, ulong)
- Real (float, double, decimal)

```
Console.WriteLine(2.6.GetType());  // System.Double
Console.WriteLine(3.GetType());  // System.Int32
```

Type	Default Value	Reference/Value
All numbers	0	Value Type
Boolean	False	Value Type
String	null	Reference Type
Char	'\0'	Value Type
Struct		Value Type
Enum	E(0)	Value Type
Nullable	null	Value Type
Class	null	Reference Type
Interface		Reference Type
Array		Reference Type
Delegate		Reference Type

Reference Types & Value Types

C# types can be divided into value types and reference types. Value types comprise most built-in types (specifically, all numeric types, the char type, and the bool type) as well as custom struct and enum types. There are two types of value types: structs and enumerations. Reference types comprise all class, array, delegate, and interface types. Value types and reference types are handled differently in memory. Value types are stored on the stack. Reference types have a reference (memory pointer) stored on the stack and the object itself is stored on the heap. With reference types, multiple variables can reference the same object, and object changes made through one variable will affect other variables that reference the same object. With value types, each variable will store its own value and operations on one will not affect another. Integers can be used with enum.

Strings

A string is a built-in non-primitive reference type that is an immutable sequence of Unicode characters. A string literal is specified between double quotes. The **+** operator concatenates two strings. A string preceded with the $ character is called an interpolated string which can include expressions inside braces **{ }** that can be formatted by appending a colon and a format string.

```
string s = $"255 in hex is {byte.MaxValue:X2}";
```

Interpolated strings must complete on a single line, unless you also specify the verbatim string operator. Note that the $ operator must come before @ as shown here:

```
int x = 2;
string s = $@"this spans {
x} lines in code but 1 on the console.";
```

Another example:

```
    string s = $@"this spans {x}
lines in code and 2 lines on the console."; // at left side of editor
```

`string` does not support < and > operators for comparisons. You must instead use string's CompareTo method, which returns a positive number, a negative number, or zero.

Char

C#'s char type (aliasing the System.Char type) represents a Unicode character and occupies two bytes. A char literal is specified inside single quotes.

```
char MyChar = 'A';
char[] MyChars = { 'A', 'B', 'C' };
Console.WriteLine(MyChar);
foreach (char ch in MyChars) { Console.Write(ch); }
```

C# Basics Cheat Sheet (2 of 4)

Escape Sequences

Escape sequences work with chars and strings, except for verbatim strings, which are proceeded by the @ symbol.

```
Console.WriteLine("Hello\nWorld");  // on two lines
Console.WriteLine("Hello\u000AWorld");  // on two lines
char newLine = '\n';
Console.WriteLine("Hi" + newLine + "World"); // on two lines
```

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code.

Char	Meaning	Value
\’	Single quote	0x0027
\”	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

Verbatim string literals. A verbatim string literal is prefixed with @ and does not support escape sequences.

```
string myPath = @"C:\temp\";
string myPath = "C:\\temp\\";
```

Constants

A local constant is much like a local variable, except that once it is initialized, its value can’t be changed. The keyword `const` is not a modifier but part of the core declaration and it must be placed immediately before the type. A constant is a static field whose value can never change. A constant is evaluated statically at compile time and the compiler literally substitutes its value whenever used (rather like a macro in C++). A constant can be any of the built-in numeric types, bool, char, string, or an enum type.

```
const int myNumber = 3;
```

Expressions

An expression essentially denotes a value. The simplest kinds of expressions are constants (such as 45) and variables (such as myInt). Expressions can be transformed and combined with operators. An operator takes one or more input operands to output a new expression.

Operators

Operators are used to operate on values and can be classed as unary, binary, or ternary, depending on the number of operands they work on (one, two, or three). They can be grouped into five types: arithmetic, assignment, comparison, logical and bitwise operators. The **arithmetic** operators include the four basic arithmetic operations, as well as the modulus operator (%) which is used to obtain the division remainder. The second group is the **assignment** operators. Most importantly, the assignment operator (=) itself, which assigns a value to a variable. The **comparison** operators compare two values and return either true or false. The **logical** operators are often used together with the comparison operators. Logical and (&&) evaluates to true if both the left and right side are true, and logical or (||) evaluates to true if either the left or right side is true. The logical not (!) operator is used for inverting a Boolean result. The **bitwise** operators can manipulate individual bits inside an integer. A few examples of Operators.

Symbol	Name	Example	Overloadable?
.	Member access	x.y	No
()	Function call	x()	No
[]	Array/index	a[x]	Via indexer
++	Post-increment	x++	Yes
--	Post-decrement	x--	Yes
new	Create instance	new Foo()	No
?.	Null-conditional	x?.y	No
!	Not	!x	Yes
++	Pre-increment	++x	Yes
--	Pre-decrement	--x	Yes
()	Cast	(int)x	No
==	Equals	x == y	Yes
!=	Not equals	x != y	Yes
&	Logical And	x & y	Yes
|	Logical Or	x | y	Yes
&&	Conditional And	x && y	Via &
||	Conditional Or	x || y	Via|
? :	Ternary	isTrue ? then this : elseThis	No
=	Assign	x = 23	No
*=	Multiply by self (and / + -)	x *= 3	Via *
=>	Lambda	x => x + 3	No

Note: The && and || operators are conditional versions of the & and | operators. The operation `x && y` corresponds to the operation `x & y`, except that y is evaluated only if x is not false. The right-hand operand is evaluated conditionally depending on the value of the left-hand operand. `x && y` is equivalent to `x ? y : false`

The `??` operator is the null coalescing operator. If the operand is non-null, give it to me; otherwise, give me a default value.

The `using` Directive

To access a class from another namespace, you need to specify its fully qualified name, however the fully qualified name can be shortened by including the namespace with a `using` directive. It is mandatory to place `using` directives before all other members in the code file. In Visual Studio, the editor will grey out any using statements that are not required.

StringBuilder

System.Text.StringBuilder

There are three Constructors

StringBuilder sb = new StringBuilder();

StringBuilder sb = new StringBuilder(myString);

StringBuilder sb = new StringBuilder(myString,capacity);

Capacity is initial size (in characters) of buffer.

The string class is immutable, meaning once you create a string object you cannot change its content. If you have a lot of string manipulations to do, and you need to modify it, use StringBuilder. Note that you cannot search your string. You do not have the following: IndexOf(), StartsWith(), LastIndexOf(), Contains() and so on. Instead you have methods for manipulating strings such as Append(), Insert(), Remove(), Clear() and Replace(). StringBuilder needs `using System.Text`. You can chain these methods together because each of these methods return a StringBuilder object.

```
static void Main(string[] args)
{
    var sbuild = new System.Text.StringBuilder("");
    sbuild.AppendLine("Title")
            .Append('=', 5)
            .Replace('=', '-')
            .Insert(0, new string('-', 5))
            .Remove(0, 4);
    Console.WriteLine(sbuild);
}
```

Arrays

An array is a fixed number of elements of the same type. An array uses square brackets after the element type. Square brackets also index the array, starting at zero, not 1.

```
static void Main(string[] args)
{
    int[] numArray = { 7, 2, 3 };
    int[] numArray2 = new int[3]; // default value is 0
    // below is 3 rows and 2 columns
    int[,] numArray3 = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
    char[] vowels = new char[] { 'a', 'e', 'i', 'o', 'u' };
    char[] vowels2 = { 'a', 'e', 'i', 'o', 'u' };  // simplified
    Array.Sort(numArray);
    foreach (int n in numArray) { Console.Write(n); }  // 237
    Console.WriteLine("First element is: " + numArray[0]);  // 2
}
```

An array itself is always a reference type object, regardless of element type. For integer types the default is zero and for reference types the default is null. For Boolean the default is False.

```
int[] a = null;  // this is legal since arrays themselves are ref tyes
```

Rectangular & Jagged Arrays

With rectangular arrays we use one set of square brackets with the number of elements separated by a comma. Jagged arrays are arrays of arrays, and they can have irregular dimensions. We use 2 sets of square brackets for jagged arrays.

```
static void Main(string[] args)
{
    // a jagged array with 3 rows
    string[][] a = new string[3][];
    a[0] = new string[1]; a[0][0] = "00";
    a[1] = new string[3]; a[1][0] = "10"; a[1][1] = "11";
                                          a[1][2] = "12";
    a[2] = new string[2]; a[2][0] = "20"; a[2][1] = "21";
    foreach (string[] b in a)
    {
        foreach (string c in b)
        {
            Console.Write(c + " ");
        }
    }
    Console.WriteLine("initialize them");
    string[][] e = { new string[] { "00" },
                     new string[] { "10", "11", "12" },
                     new string[] { "20", "21" } };

    foreach (string[] f in e)
    {
        foreach (string g in f)
        {
            Console.Write(g + " ");
        }
    }
}
```

C# Basics Cheat Sheet (3 of 4)

DateTime

DateTime is a struct and is therefore a value type.

```
var dateTime = new DateTime(2000, 1, 1);
var now = DateTime.Now;  // gets the current date & time
var today = DateTime.Today;  // gets the current date (no time)
var utcnow = DateTime.UtcNow;
Console.WriteLine($"The current hour is: {now.Hour}");
Console.WriteLine($"The current minute is:  {now.Minute}");
Console.WriteLine($"The current second is:  {now.Second}");
var tomorrow = now.AddDays(1);
var yesterday = now.AddDays(-1);
// AddDays, AddHours, AddMinutes, AddMonths, AddYears etc.
Console.WriteLine($"Tomorrow (yyyy-mm-dd): {tomorrow}");
Console.WriteLine(now.ToLongDateString());
Console.WriteLine(now.ToShortDateString());
Console.WriteLine(now.ToLongTimeString());
Console.WriteLine(now.ToShortTimeString());
Console.WriteLine(now.ToString());  // shows date and time
Console.WriteLine(now.ToString("yyyy-MM-dd"));  // format specifier
Console.WriteLine(now.ToString("yyyy-MMMM-dd"));  // format specifier
Console.WriteLine(now.ToString("dddd yyyy-MMMM-  dd"));
Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm:ss"));
Console.WriteLine(String.Format("today: {0:D}", now));
Console.WriteLine(String.Format("today: {0:F}", now));
// D F d f g G M m Y y t T s u U
```

TimeSpan

```
// Creating TimeSpan object - there are 3 ways.
var timeSpan = new TimeSpan(2, 1, 45);  // hours minutes second
// Creating TimeSpan object - there are 3 ways.
var timeSpan = new TimeSpan(2, 1, 45);  // hours minutes seconds
var timeSpan1 = new TimeSpan(3, 0, 0);  // 3 hours
// second way:
// easier to know it is one hour with FromHours()
var timeSpan2 = TimeSpan.FromHours(1);
// third way:
var now = DateTime.Now;
var end = DateTime.Now.AddMinutes(2);
var duration = end - now;
Console.WriteLine("Duration: " + duration);
// above result is: Duration: 00:02:00.00199797
var negativeduration = now - end;
Console.WriteLine("\"Negative Duration\": " + duration);  // positive
number

TimeSpan trueEnd = now.AddMinutes(2) - now; // subtract to get TimeSpa
n object
Console.WriteLine("True Duration: " + trueEnd);
// above output: True Duration: 00:02:00

// Properties
// timeSpan is two hours, one minutes and 45 seconds
Console.WriteLine("Minutes: " + timeSpan.Minutes);
Console.WriteLine("Total Minutes: " + timeSpan.TotalMinutes);
Console.WriteLine("Total Days: " + timeSpan.TotalDays);

// Add Method of TimeSpan
// Add 3 min to our original TimeSpan 2 hours 1 minutes 45 seconds
Console.WriteLine("Add 3 min: " + timeSpan.Add(TimeSpan.FromMinutes(3)
));
Console.WriteLine("Add 4 min: " + timeSpan.Add(new TimeSpan(0,4,0)));
// ToString method
Console.WriteLine("ToString: " + timeSpan.ToString());
// don't need ToString here:
Console.WriteLine("ToString not needed: " + timeSpan);
// Parse method
Console.WriteLine("Parse: " + TimeSpan.Parse("01:02:03"));
```

Formatting Numerics

Numbers fall into two categories: integral and floating point.

Format Specifier	Pattern	Value	Description
C or c	{0:C2}, 2781.29	$2781.29	Currency
D or d	{0:D5}, 78	00078	Must be integer
E or e	{0:E2}, 2781.29	2.78+E003	Must be floating point
F or f	{0:F2}, 2781.29	2781.29	Fixed point
G or g	{0:G5}, 2781.29	2781.2	General
N or n	{0:N1}, 2781.29	2,781.29	Inserts commas
P or p	{0:P3}, 4516.9	45.16%	Converts to percent
R or r	{0:R}, 2.89351	2.89315	Retains all decimal places (round-trip)
X or x	{0,9:X4}, 17	0011	Converts to Hex

```
Console.WriteLine("Value: {0:C}.", 447); // $447.00
int myInt = 447;
Console.WriteLine($"Value: {myInt:C}"); // $ is interpolation $447.00
```

The optional alignment specifier represents the minimum width of the field in terms of characters. It is separated from the index with a comma. It consists of a positive or negative integer. The sign represents either right (positive) or left (negative) alignment.

```
Console.WriteLine("Value: {0, 10:C}", myInt); // + right align
Console.WriteLine("Value: {0, -10:C}", myInt); // - left align
```

```
Value:     $447.00
Value: $447.00
```

```
Console.WriteLine($"Value: {myInt, 10:C}"); // interpolation
```

```
Value:     $447.00
```

```
Console.WriteLine("Percent: {0:P2}",0.126293); // 12.63 rounds
Console.WriteLine("{0:E2}", 12.6375);//2 decimal places 1.26E+001
```

Enumerated Type

It can be defined using the `enum` keyword directly inside a namespace, class, or structure.

```
public enum Score
{
    Touchdown = 6, FieldGoal = 3, Conversion = 1, Safety = 2,
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Score.Touchdown);// output: Touchdown
        int myInt = (int)Score.FieldGoal;
        Console.WriteLine(myInt); // output: 3
        Score myScore = (Score)6;
        Console.WriteLine(myScore); // output: Touchdown
        string teamscore = "Conversion";
        Enum.TryParse(teamscore, out Score myVar);
        Console.WriteLine(myVar);  // output: Conversion
        Console.WriteLine((int)myVar);  // output: 1
    }
}
```

Enumerations could just be a list of words. For example you could have a list of the days of the week: Monday, Tuesday and so on, without any values. You can use `IsDefined()` and `typeof()`.

```
if(Enum.IsDefined(typeof(Score), “Safety”)…
```

The Object Class

In .NET, everything is an object and the base of everything is the `Object` class. The available methods of an object are: `Equals`, `GetHashCode`, `GetType` and `ToString`.

Struct

You can define a custom value type with the `struct` keyword. A struct is a **value type**. Fields cannot have an initializer and cannot inherit from a class or be inherited. The explicit constructor must have a parameter.

```
struct Customer
{
    public string firstName;
    public string lastName;
    public string middleName;
    public int birthYear;
    //public string Name() => firstName + " " + middleName + " " + las
tName;
    public string Name() => firstName + " " +
                    (String.IsNullOrWhiteSpace(middleName) ? "" :
                                middleName + " ") + lastName;
    // Name() accesses firstName & lastName; uses lambda and ternary
    public string NameFunct()
    {
        string midN = String.IsNullOrWhiteSpace(middleName) ?
                                        "" : middleName + " ";
        return firstName + " "  + midN + lastName;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Customer myCustomer;
        myCustomer.firstName = "Sam";
        myCustomer.lastName = "Smith";
        myCustomer.middleName = "   ";  // note the spaces
        myCustomer.birthYear = 1960;
        Console.WriteLine($"{myCustomer.Name()} was born in {myCustome
r.birthYear}.");
        Console.WriteLine($"{myCustomer.NameFunct()} was born in {myCu
stomer.birthYear}.");
        // Output: Sam Smith was born in 1960.
        // Output: Sam Smith was born in 1960.
    }
}
```

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either implicit or explicit: implicit conversions happen automatically, whereas explicit conversions require a **cast**. One useful use of a conversion is when you are getting input from the user in a Console program, using **Convert()**.

```
Console.WriteLine("Enter your number: ");
double number = Convert.ToDouble(Console.ReadLine());
```

Here are some examples below using implicit and explicit conversions.

```
int x = 12345; // int is a 32-bit integer
long y = x; // implicit conversion to 64-bit long
short z = (short)x; // cast - explicit conversion to 16-bit int
Console.WriteLine(z);
byte b = (byte)x;  // data loss !!
Console.WriteLine(b);  // 57
// 12345 = 0011 0000 0011 1001
// 57 =            0011 1001
int myInt = 1_000_000;  // C# 7 allows underscores
Console.WriteLine(2.6.GetType());  // System.Double
Console.WriteLine(3.GetType());  // System.Int32
```

Conversions from integral types to real types are implicit, whereas the reverse must be explicit. Converting from a floating-point to an integral type truncates any fractional portion; to perform rounding conversions, use the static System.Convert class.

```
float f = 128.67F;
int d = Convert.ToInt32(f);  // rounds
// System.Int32 d is 129
Console.WriteLine(d.GetType() + " d is " + d);
```

C# Basics Cheat Sheet (4 of 4)

The Regex Class

The class is System.Text.RegularExpressions.Regex

Pattern	Desscription	Example
+	Matches one or more	ab+c matches abc, abbc
*****	Matches zero or more	ab*c matches ac, abbc
?	Matches zero or one	ab?c matches ac, abc
\d	Any digit from 0 to 9	\d\d matches 14, 98, 03
[0-9]	Any digit from 0 to 9	[0-9] matches 3, 8, 1, 0, 2
\d{3}	Any 3 digits from 0-9	\d{3} matches 123, 420
[0-9]{3}	Any 3 digits from 0-9	[0-9]{3} matches 123, 420

Comparison Operators

The comparison operators compare two values and return true or false. They specify conditions that evaluate to true or false (like a predicate):

== != > < >= <=

Conditional Statements

Syntax	Example

if (condition) {
  // statements
} else {
  // statements
}
	if (product == "H1")
	  price = 134.00M;// M decimal
	else if (product == "H2")
	  price = 516.00M;
	else price = 100.00M;

q ? a : b, if condition q is true, a is evaluated, else b is evaluated.	price = (product == "A1") ? 34 : 42; // ternary operator ? :

switch (expression)
{ case expression:
// statements
break / goto / return() case ...
default:
// statements
break / goto / return()
}
// expression may be integer, string, or enum
	switch (product)
	{
	  case "P1": price = 15; break;
	  case "P2": price = 16; break;
	  default: price = 10M; break;
	}

Loops

Syntax	Example

while (condition)
{ body }
	var i = 1;
	var total = 0;
	while (i <= 4)
	{ // 1 + 2 + 3 + 4 = 10
	  total = total + i;
	  i++;
	}

do { body }
while (condition);
	do
	{ // 1 + 2 + 3 + 4 + 5 = 15
	  total = total + i;
	  i++;
	}
	while (i <= 4);

for (initializer;
  termination condition;
  iteration;)
{ // statements }
	for (var i = 1; i < list.Count; i++)
	{
	  if (list[i] < min)
	  min = list[i];
	}

foreach (type identifier in *collection*)
{ // statements }
	int[] nums = new int[]{ 2, 5, 4};
	foreach (int num in nums)
	{
	  Console.WriteLine(num);
	}

Lists

Lists are covered in more detail in the Lists of Objects part in the Advanced section, but are here now due to their importance and popularity.

```
var numbers = new List<int>() {1,2,3,4};
numbers.Add(1);
numbers.AddRange(new int [3] {5,6,7});
foreach (var num in numbers) Console.Write(num + " ");
```

File IO

```
using System;
using System.IO;  // add this
namespace FileManipulation
{
  class Program
  {// File (statis) and FileInfo (instance)
      static void Main(string[] args)
      {
       var filenamewithpath = @"D:\myfile.txt";  // verbatim @
       using (File.Create(filenamewithpath))
       // without using you get Unhandled Exception
       // true will over-write existing file
       File.Copy(filenamewithpath, @"D:\myfile_2.txt", true);
       File.Copy(filenamewithpath, @"D:\myfile_3.txt", true);
       File.Delete(@"D:\myfile_3.txt");
       if(File.Exists(@"D:\myfile_2.txt"))
       {
        Console.WriteLine("File " + @"D:\myfile_2.txt" + " exists.");
       }
       string filecontent = File.ReadAllText(filenamewithpath);
       var fileInfo = new FileInfo(filenamewithpath);
       fileInfo.CopyTo(@"D:\myfile_4.txt", true);
       var fileInfo4 = new FileInfo(@"D:\myfile_4.txt");
       if (fileInfo4.Exists)  // Exists is a property
       {
           fileInfo4.Delete();  // takes no paramters
       }
       else
       {
        Console.WriteLine("Cannot delete file "
           + @"D:\myfile_4.txt" + " because it does not exist.");
       }
       // FileInfo does not have a ReadAllText method
       // need to call openread which returns a file string but
       // that is a little bit complex.
       Console.WriteLine("Press any key to continue...");
      }
  }
}
```

Use File for occasional usage and FileInfo for many operations because each time you use File the OS does security checks and that can slow down your app; with FileInfo you need to create an instance of it. Both are easy to use. StreamReader and StreamWriter are available. You can encode in ASCII, Unicode, BigEndianUnicode, UTF-8, UTF-7, UTF-32 and Default. Different computers can use different encodings as the default, but UTF-8 is supported on all the operating systems (Windows, Linux, and Max OS X) on which .NET Core applications run..

```
var filenamewithpath = @"D:\temp\A_ascii.txt";
File.WriteAllText(filenamewithpath, "A", Encoding.ASCII);
```

Directory IO

```
using System;
using System.IO;   // add this
namespace Directories
{
  class Program
  {
    static void Main(string[] args)
    {
     Directory.CreateDirectory(@"D:\temp\folder1");
     File.Create(@"D:\temp\folder1\mytext.txt");
     File.Create(@"D:\temp\folder1\mytext2.txt");
     string[] files = Directory.GetFiles(@"D:\temp\folder1", "*.*",
         SearchOption.AllDirectories);  // or TopDirectoryOnly

     foreach (var file in files){Console.WriteLine(file);}
     var directories = Directory.GetDirectories(@"D:\temp","*.*",
         SearchOption.AllDirectories);
     foreach (var dir in directories)
     {
         Console.WriteLine(dir);
     }
     var directoryInfo = new DirectoryInfo(@"D:\temp\folder1");
     var ct = directoryInfo.CreationTime;
     Console.WriteLine("Creation date and time: " + ct);
     }
  }
}
```

Debugging

To debug your code you first decide where in your code you suspect a problem and create a breakpoint by putting the cursor on that line and pressing **F9**. Press **F5** to run the program in debug mode. You can use multiple breakpoints if you want. We can either use **F10** to step over or perhaps **F11** to step into. Place your cursor over a variable and you should be able to see the data inside. If all looks good, go ahead and press F10 or perhaps F11. If you have another breakpoint, you can press F5 to run to the next breakpoint. Also, you can move the current position of execution backwards by dragging the yellow arrow at the left. When you are done you can press **Shift+F11** to step out. You can end the debugging with **Shift+F5**. You can run it without the debugger with **Ctrl+F5**. You can manage all your breakpoints with the Breakpoints window. Debug ➤ Windows ➤ Breakpoints.

It's a good idea to always check that the methods you write receive meaningful data. For example, if you expect a list of something, check that the list is not null. Users may not enter values you expect. It's important to think of these Edge Cases, which are uncommon scenarios, which is the opposite of the Happy Path.

NuGet Package Manager

NuGet is the package manager for .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers.. Packages are installed into a Visual Studio project using the Package Manager UI or the Package Manager Console. One interesting package is the HtmlAgilityPack that allows you to parse HTML, but there are lots of them.

Your Own Library (Assembly)

To create a class library using Visual Studio 2017 Community, in the menu select File ➤ New ➤ Project ➤ Installed ➤ Visual C# ➤ .NET Standard ➤ Class Library(.NET Standard) and give it a name and location and press OK. Write your library code. Switch to Release from Build. Press Ctrl+Shift+B to build the DLL. Note the location of the DLL (**bin\Release\netstandard2.0**). Within the project that uses the library, you need to give the compiler a **reference** to your assembly by giving its name and location. Select Solution Explorer ➤ Right-click the References folder ➤ Add Reference. Select the Browse tab, browse to the DLL file mentioned above. Click the OK button. For convenience you can now add a **`using`** statement at the top of your program. You should now have access to your library code. Nice!