C# Basics Cheat Sheet (1 of 4)

begincodingnow.com

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.

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 x 10^{-45} to +-3.4 x 10^{38}	Single
double	8	D	+-5.0 x 10^{-324} to +-1.7 x 10^{308}	Double
decimal	16	M	+-1.0 x 10^{-28} to +-7.9 x 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.

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)

begincodingnow.com

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)

begincodingnow.com

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)

begincodingnow.com

The Regex Class

The class is System.Text.RegularExpressions.Regex

Pattern	Description	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 section in the Advanced section, but are here now due to their importance.

```
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!

C# OOP Cheat Sheet (1 of 4)

begincodingnow.com

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that employs objects to encapsulate code. Objects consist of types and are called classes. A class is just a template for an object which is an instance of the class, which occupies memory. When we say that a class is instantiated, we mean that an object in memory has been created. Classes contain data and executable code. Everything in C# and .NET is an object. In the menu View ➤ Object Browser.

Programming Principles

DRY is an acronym for Don't Repeat Yourself. In OOP, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof: (1) A language mechanism for restricting direct access to some of the object's components. (2) A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data. In OOP, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Simple Class Declaration

The simplest class declaration is:

```
class Foo { }
```

```
[ public | protected | internal | private ]
[ abstract | sealed | static ]
class class_name [:class/interfaces inherited from ]
```

A `class` is a data structure that can store data and execute code. It contains data members and function members. The members can be any combination of nine possible member types. A local variable is a variable declared inside a function member. On the Internet are the StyleCop Rules Documentation the ordering of members in classes. Note: Whenever you have a class, such as our Customer, and inside that class you have a **List** of objects of any type, you should always initialize that list to an empty list.

Static

Static classes are meant to be consumed without instantiating them. Static classes can be used to group members that are to be available throughout the program. A static class must have all members marked as static as well as the class itself. The class can have a static constructor, but it cannot have an instance constructor. Static classes are implicitly sealed, meaning you cannot inherit from a static class. A non-static instantiable class can have static members which exist and are accessible even if there are no instances of the class. A static field is shared by all the instances of the class, and all the instances access the same memory location when they access the static field. Static methods exist. Static function members cannot access instance members but can access other static members. Static members, like instance members, can also be accessed from outside the class using dot-syntax notation. Another option to access the member doesn't use any prefix at all, if you have included a using static declaration for the specific class to which that member belongs:

```
Using static System.Console;
```

Abstract Classes

A class declared as abstract can never be instantiated. Instead, only its concrete subclasses can be instantiated. Abstract classes can define abstract members which are like virtual members, except they don't provide a default implementation. That implementation must be provided by the subclass, unless that subclass is also declared abstract.

Sealed Classes

A sealed class cannot be used as the base class for any other class. You use the sealed keyword to protect your class from the prying methods of a subclass. Static classes are implicitly sealed.

Instance Constructors

For classes, the C# compiler automatically generates a parameterless public constructor if and only if you do not define any constructors. However, as soon as you define at least one constructor, the parameterless constructor is no longer automatically generated, so you may need to write it yourself.

Instance constructors execute when the object is first instantiated. When an object is destroyed the destructor is called. Memory is freed up at this time. Constructors are called with the new keyword.

Constructors can be static. A static constructor executes once per type, rather than once per instance. A type can define only one static constructor, and it must be parameterless and have the same name as the type.

```
public class Customer
{
    public string Name;  // in real world these are private
    public int Id;  // in real world these are private
    public Customer() { }  // constructor (same name as class)
    public Customer(int id) // constructor
    {
        this.Id = id;  // set Id property
    }
    public Customer(int id, string name) // constructor
    {
        this.Id = id;  // 'this' references current object Customer
        this.Name = name;  // here we set Name property
    }
}
class Program
{
    static void Main(string[] args)
    {
        // ERROR: not contain constructor that takes zero arguments
        // unless we create OUR OWN parameterless constructor (we did)
        var customer = new Customer();
        customer.Id = 7;
        customer.Name = "John";
        Console.WriteLine(customer.Id);
        Console.WriteLine(customer.Name);
    }
}
```

Fields

A field is a variable that belongs to a class. It can be of any type, either predefined or user-defined. A field initializer is part of the field declaration and consists of an equal sign followed by an expression that evaluates to a value. The initialization value must be determinable at compile time. If no initializer is used, the compiler sets the value of a field to a default value, determined by the type of the field. The default value for each type is 0 and is false for bool. The default for reference types is null.

```
class Order
{
    public int Id;
}
class Customer
{
    public int Id;
    public string Name;
    public readonly List<Order> Orders = new List<Order>();
    // Note: no parameterless constructor for Customer
    public Customer(int id)
    {   // a constructor
        this.Id = id;  // the keyword this is redundant
    }
    public Customer(int id, string name) : this(id)
    {   // a constructor
        this.Name = name; // the keyword this is redundant
    }
    public void DoSomething() { } // just an example method

}
class Program
{
    static void Main(string[] args)
    {
        var customer = new Customer(3, "Bob");
        customer.Orders.Add(new Order());
        customer.Orders.Add(new Order() { Id = 7 });
        Console.WriteLine("Customer Id: " + customer.Id + " Name: "
            + customer.Name);
        Console.WriteLine("Num orders: " + customer.Orders.Count);
        foreach (var ord in customer.Orders) { Console.WriteLine("O
rder Id: " + ord.Id); }
    }
}
```

Here is the Console output of he above program. Notice that the first order Id below is zero because zero is the default.

```
Customer Id: 3 Name: Bob
Number of orders: 2
Order Id: 0
Order Id: 7
```

Generally, you would use private fields with public properties to provide encapsulation.

Methods

A method is a named block of code that is a function member of a class. You can execute the code from somewhere else in the program by using the method's name, provided you have access to it. Below is the simplest way to write a method inside a class.

```
class NotAnything { void DoNothingMethod() { } }
```

You can also pass data into a method and receive data back as output. A block is a sequence of statements between curly braces. It may contain local variables (usually for local computations), flow-of-control statements, method invocations, nested blocks or other methods known as local functions.

```
[access modifier]
[static|virtual|override|new|sealed|abstract]
method name (parameter list) { body }
```

C# allows for **optional** parameters which you can either include or omit when invoking the method. To specify that, you must include a default value for that parameter in the method declaration. Value types require the default value to be determinable at compile time, and reference types only if the default value is null. The declaration order must be all required (if any) – all optional (if any) – all params (if any).

C# OOP Cheat Sheet (2 of 4)

begincodingnow.com

Access Modifier	Description
public	Fully accessible. This is the implicit accessibility for members of an enum or interface.
private	Accessible only within the containing type. This is the default accessibility for members of a class or struct. Perhaps you have a method that is implementation detail that calculates something.
protected	Accessible only within the containing type or subclasses (derived classes). May be a sign of bad design.
internal	Accessible only from the same assembly. We create a separate class library and use internal. How? Right-click Solution ➤ Add ➤ New Project ➤ Class Library (DLL). We'll need to add a Reference (Project, Add Reference) and add using statement.
protected internal	Not used normally! Accessible only from the same assembly or any derived classes. The union of protected and internal.

virtual – method can be overridden in subclass.

override – overrides virtual method in base class.

new – hides non-virtual method in base class.

sealed – prevents derived class from inheriting.

abstract – must be implemented by subclass.

Below is an example of a method called MyMethod (()

```
public class MyClass
{
    public int MyMethod (int integer, string text)
    {
        return 0;
    }
}
```

Properties

A property is declared like a field, but with a get/set block added. Properties look like fields from the outside, but internally they contain logic, like methods do. You can set the values of a public field and a public property, no problem. Note that `-=` means subtract from self.

```
class Program
{
    static void Main(string[] args)
    {
        Item it = new Item();
        it.FieldPrice = 24.67M;
        it.PropertyPrice = 45.21M;
        Console.WriteLine(it.FieldPrice + " " + it.PropertyPrice);
        it.FieldPrice -= 1.00M;
        it.PropertyPrice -= 1.00M;
        Console.WriteLine(it.FieldPrice + " " + it.PropertyPrice);
    }
}
public class Item
{
    public decimal FieldPrice;
    public decimal PropertyPrice { get; set; }
}
```

Here is a public property Amount with its backing field, that can be simplified with auto implemented property with `{ get; set; }`.

```
private decimal _amount; // backing field
public decimal Amount  // public property
{
    get { return _amount; }
    set { _amount = value; } // notice the keyword value
}
```

The `get` and `set` denote property **accessors**. The `set` method could throw an exception if value was outside a valid range of values.

Object Initialization Syntax

C# 3.0 (.NET 3.5) introduced Object Initializer Syntax, a new way to initialize an object of a class or collection. Object initializers allow you to assign values to the fields or properties at the time of creating an object without invoking a constructor.

```
class Program
{
    public class Person
    {
        public int id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
    }
    static void Main(string[] args)
    {   // don't need to initialize all fields
        var p = new Person {FirstName = "J", LastName = "Smith"};
        Console.WriteLine("Last name is {0}", p.LastName);
        // OUTPUT: Last name is Smith
    }
}
```

If the class has a constructor that initializes a field, the field in the object initializer syntax wins. Below, the last name is Smith.

```
public class Person
{
    public int id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public Person()  // constructor
        { LastName = "Johnson"; }
}
static void Main(string[] args)
{   // don't need to initialize all fields
    var p = new Person {FirstName = "J", LastName = "Smith"};
    Console.WriteLine("Last name is {0}", p.LastName);
    // OUTPUT: Last name is Smith
}
```

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are like properties but are accessed via an index argument rather than a property name. The string class has an indexer that lets you access each of its char values via an int Index.

```
string s = "hello";
Console.WriteLine(s[0]); // 'h' zero-based
Console.WriteLine(s[1]); // 'e'
Console.WriteLine(s[99]); // IndexOutOfRangeException
Console.WriteLine(s[-1]); // IndexOutOfRangeException
```

The index argument(s) can be of any type(s), unlike arrays. You can call indexers null-conditionally by inserting a question mark before the square bracket as shown below.

```
string str = null;
Console.WriteLine(str?[0]); // Writes nothing; no error.
Console.WriteLine(str[0]); // NullReferenceException
```

To write an indexer, define a property called `this`, specifying the arguments in square brackets.

```
class Sentence
{
    string[] words = "The quick brown fox".Split(); //field
    public Sentence() { }  // default constructor
    public Sentence(string str) // constructor
            { words = str.Split(); }
    public int Length  // property
            { get { return words.Length; } }
    public string this[int wordNum] // indexer
    {
        get { return words[wordNum]; }
        set { words[wordNum] = value; }
    }
}
static void Main(string[] args)
{
    string s = "hello world";
    Console.WriteLine(s[0]); // 'h' zero-based
    Console.WriteLine(s[5]); // ' '
    string str = null;
    Console.WriteLine(str?[0]); // Writes nothing; no error.
    // Console.WriteLine(str[0]); // NullReferenceException

    Sentence sen = new Sentence();
    Console.WriteLine(sen[1]); // quick
    sen[3] = "wildebeest";  // replace the 4th word
    Console.WriteLine(sen[3]); // wildebeest
    for (int i=0;i<sen.Length;i++) { Console.Write(sen[i] + "|"); }
    // now use our constructor to use our sentence
    Sentence sent = new Sentence("The sleeping black cat");
    Console.WriteLine(sent[1]);  // sleeping
}
```

You have your own class `Customer` with fields `FirstName` and `LastName`. Instantiate it as Cust1. Get the first name and last name with Cust1.FirstName and Cust1.LastName. Indexers allow you to do the same with Cust1[0] and Cust1[1] respectively. An indexer is a pair of get and set accessors inside the code block of `ReturnType this [ Type param1, ... ]`. The `set` and `get` blocks use `switch`.

Inheritance

Inheritance is a type of relationship ("Is-A") between classes that allows one class to inherit members from the other (code reuse). A horse "is an" animal. Inheritance allows for polymorphic behaviour. In UML, the Animal is at the top with the Horse under it with an arrow pointing up to Animal. Another example of inheritance is where a Saving Bank Account and Chequing Bank Account inherit from a Bank Account.

```
public class BaseClass
{
    public int Amount { get; set; }
    public BaseClass() { Console.WriteLine("Base constr"); }
    public void BaseDo() { Console.WriteLine("Base's BaseDo."); }
    public virtual void Do() { Console.WriteLine("Base's Do"); }
}
public class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("Sub constr"); }
    public override void Do() { Console.WriteLine("Sub's Do");}
}
class Program
{
    static void Main(string[] args)
    {
        var bas = new BaseClass();
        var sub = new SubClass();
        sub.Amount = 1;  // Amount inherited from Base
        sub.Do(); // Sub's Do
    }
}
```

Output:

```
Base constr
Base constr
Sub constr
Sub's Do
```

Constructor Inheritance

When you instantiate a sub class, base class constructors are always executed first, then sub class constructors, as you can see in lines 2 and 3 from the output. Base class constructors are not inherited.

C# OOP Cheat Sheet (3 of 4)

begincodingnow.com

Composition (aka Containment)

Composition is a type of relationship (“has -a”) between two classes that allows one class to **contain** another. Inheritance is another type of relationship. Both methods give us code re-use. In our example, both the car and truck have an engine and the engine needs to send a message to the console. We use a private field in the **composite** class (car and truck) to achieve this. You use a member field to hold an object instance. Generally, inheritance results in a more tightly-couple relationship than composition and many developers prefer composition, but it depends on your project. Two things to remember: **private field** and **constructor**.

```
using System;
namespace CompositionGeneral
{
    class Car
    {
        private readonly Engine _engine;
        public Car(Engine engine)  // constructor
        {
            _engine = engine;
        }
        public void DriveCar()
        {
            float speed = 0.0F;
            _engine.EngineStatus("car starting engine");
            speed = 50.0F;


          _engine.EngineStatus($"speed of {speed} Km/hr");
            _engine.EngineStatus("car engine off");
        }
    }
    class Truck
    {
        private readonly Engine _engine;
        public Truck(Engine engine)  // constructor
        {
            _engine = engine;
        }
        public void DriveTruck() { //...
        }
    }
    class Engine  // the car and truck "Have An" engine
    {
        public void EngineStatus(string message)
        {
            Console.WriteLine("Engine status: " + message);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var e = new Engine();
            var sedan = new Car(e);
            sedan.DriveCar();
            var pickup = new Truck(new Engine());
            pickup.DriveTruck();

        }
    }
}
```

Instead of having the car and truck contain a concrete class, like Engine, what if we used an interface, like IEngine instead? Please see the section on **Interfaces & Extensibility** for an example of this.

Composition vs Inheritance

Designing classes needs to be done carefully. Be careful with designing your inheritance because it can result in large hierarchies that become fragile (difficult to modify due to tight coupling). You can always re-factor inheritance into composition. A horse and a fish are both animals, but they are quite different. Both eat and sleep (Animal class) but horses walk and fish swim. You could use composition and create a CanWalk and CanSwim class. The horse “has-a” CanWalk class. This is fine even though “has-a” may not make sense or sound correct in the real world. You don’t want to put a Walk() method in your Animal class unless you are certain all of your animals now and in the future can walk. If you have that, using inheritance, you may need a sub-class of Animal called mammal, and re-compile and re-deploy your code. Also, with composition we get an extra benefit that’s not possible with inheritance: **Interfaces**. We can replace our Animal class with an interface IAnimal. This is dependency injection and is covered later in the topic called Interfaces & Extensibility.

Method Overriding & Polymorphism

Method overriding is changing the implementation of an **inherited** method that came from the base class. Use the `virtual` keyword in the method of the base class and `override` in the derived class. `Virtual` is just the opportunity to `override`. You don’t have to override it. What is polymorphism? Poly means many and morph means form. Let’s use an example with classes called `BaseClass`, `ChildRed` and `ChildBlue`.

```
class MyBaseClass
{
    public int CommonProperty { get; set; }
    public virtual void WriteMessage() { }
}
class ChildRed : MyBaseClass
{
    public new int CommonProperty { get; set; }
    public override void WriteMessage()
    {
        CommonProperty = 46;
        Console.WriteLine("Red " + CommonProperty);    }
}
class ChildGreen : MyBaseClass
{
    public override void WriteMessage()
    {
        Console.WriteLine("Green " + CommonProperty);
    }
}
class Display
{
    public void WriteMyMessages(List<MyBaseClass> baseclasses)
    {
        foreach (var bc in baseclasses)
        {
            bc.WriteMessage();
        }
    }
}
```

When we call `WriteMessage()` above we have polymorphic behavior. We have a list of different colors, but the implementation is different for each colour. Red and Green overrode the base class’s method. Notice that the list is a list of the base class `MyBaseClass.`

```
class Program
{
    static void Main(string[] args)
    {
        var baseclasses = new List<MyBaseClass>();
        baseclasses.Add(new ChildRed() { CommonProperty = 1 });
        baseclasses.Add(new ChildGreen() { CommonProperty = 3 });
        var display = new Display();
        display.WriteMyMessages(baseclasses);
    }
}
```

You can assign a variable that is of a derived type to a variable of one of the base types. No casting is required for this. You can then call methods of the base class through this variable. This results in the implementation of the method in the derived class being called. You can cast a base type variable into a derived class variable and call the method of the derived class.

Interfaces

An interface is like a class, but it provides a specification rather than an implementation for its members. It’s a “contract”. Interface members are all implicitly abstract. A class (or struct) can implement multiple interfaces. In contrast, a class can inherit from only a single class, and a struct cannot inherit at all (aside from deriving from `System.ValueType`). The interface's members will be implemented by the classes and structs that implement the interface. By convention, interface names start with the capital letter “I”.

```
interface IInfo
{
    string GetName();
    string GetAge();
}
class CA : IInfo
{   // declare that CA implements the interface IInfo
    public string Name;
    public int Age;
    // implement two interface methods of IInfo:
    public string GetName() { return Name; }
    public string GetAge() { return Age.ToString(); }
}
class CB : IInfo
{   // declare that CB implements the interface
    public string First;
    public string Last;
    public double PersonsAge;
    public string GetName() { return First + " " + Last; }
    public string GetAge() { return PersonsAge.ToString(); }
}
class Program
{   // pass objects as references to the interface
    static void PrintInfo(IInfo item)
    {
        Console.WriteLine("Name: {0}  Age: {1}",  item.GetName() ,
                                                  item.GetAge() );
    }
    static void Main()
    {
        // instantiate using object initialization syntax
        CA a = new CA() { Name = "John Doe", Age = 35 };
        CB b = new CB() { First = "Jane", Last = "Smith",
                                           PersonsAge = 44.0 };
        // references to the objects are automatically
        // converted to references
        // to the interfaces they implement (in the code below)
        PrintInfo(a);
        PrintInfo(b);

        Type myType = typeof(Program);
        Console.Title = myType.Namespace;
    }
}
```

Output:

```
Name: John Doe  Age: 35
Name: Jane Smith  Age: 44
```

C# OOP Cheat Sheet (4 of 4)

begincodingnow.com

Interfaces & Extensibility

We create a constructor and inject a dependency. This is called **dependency injection**, which means that in the constructor we are specifying the dependencies of our class. The FileProcessor is not directly dependent on the ConsoleLogger. It doesn't care who implements ILogger. It could be a DatabaseLogger that does it.

```
// FilesProcessor is dependent on an interface.
public interface ILogger
{
    void LogError(string message);  // method
    void LogInfo(string message);  // method
}
public class ConsoleLogger : ILogger
{   // ConsoleLogger implements ILogger
    public void LogError(string message)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
        Console.ForegroundColor = ConsoleColor.White;
    }
    public void LogInfo(string message)
    {
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine(message);
        Console.ForegroundColor = ConsoleColor.White;
    }
}
public class FilesProcessor
{
    private readonly ILogger _logger;
    public FilesProcessor(ILogger logger) // constructor
    {
        _logger = logger;
    }
    public void Process()
    {
        try
        {   // we might employ the using keyword in the real world
            _logger.LogInfo($"Migrating started at {DateTime.Now}");
            _logger.LogInfo($"In middle of doing stuff...");
            int zero = 0;
            int myError = 1 / zero;
            _logger.LogInfo($"Migrating ended at {DateTime.Now}");
        }
        catch
        {
            _logger.LogError($"Opps! Error");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Our logger to sends to console
        var filesProcessor = new FilesProcessor(new ConsoleLogger());
        filesProcessor.Process();
        Console.WriteLine("done program.");
    }
}
```

Output:

```
Migrating started at 2019-02-07 10:05:43 AM
In middle of doing stuff...
Opps! Error
done program.
```

We can **extend** it. We can create more loggers other than ConsoleLogger, such as DatabaseLogger, FileLogger, EmailLogger SMSLogger and so on, and all we need to do is change the Main().

Interfaces & Testability

Using interfaces help with **unit testing**. This example builds on many topics in this cheat sheet. We'll use the Microsoft Test Runner. You get a new journal entry and then post it passing the entry to the Post() method of the JournalPoster class. Posting the entry requires the services of the checker. In order to make testing work, we need to use an interface for the checker.

```
class Program
{
    static void Main(string[] args)
    {
        var jp = new JournalPoster(new DrEqualsCrChecker());
        var je = new JournalEntry { DebitAmount = 120.50f,
                                    CreditAmount = -120.50f };
        jp.Post(je);
        Console.WriteLine("Posted? " + je.IsPosted);
        Console.WriteLine("Date posted: " +
            je.Posting.PostingDate.ToString("yyyy-MM-dd"));
        Console.WriteLine("Debit amount: {0:C}", je.DebitAmount);
        Console.WriteLine("Credit amount: {0:C}", je.CreditAmount);
        Console.WriteLine("JE Balance: {0:C}", je.Posting.Balance);
    }
}
```

Output:

```
Posted? True
Date posted: 2019-02-07
Debit amount: $120.50
Credit amount: -$120.50
JE Balance: $0.00
```

```
public class JournalEntry
{   // in the reaal world there is more than this
    public Posting Posting { get; set; }
    public float DebitAmount = 0f;
    public float CreditAmount = 0f;
    public DateTime DatePosted { get; set; }
    public bool IsPosted
    {
        get { return Posting != null; }
    }
}
public class Posting
{
    public float Balance { get; set; } // zero if Dr = Cr
    public DateTime PostingDate { get; set; }
}
public class JournalPoster
{   // this class does not even know about DrEqualsCrChecker
    private readonly IDrEqualsCrChecker _checker;
    public JournalPoster(IDrEqualsCrChecker checker)
        { _checker = checker; }
    public void Post(JournalEntry je)
    {
        if (je.IsPosted)
            throw new InvalidOperationException("Opps. Already posted!
");
        je.Posting = new Posting
        {
            Balance = _checker.CalcBalance(je),
            PostingDate = DateTime.Today.AddDays(1)
        };
    }
}
public interface IDrEqualsCrChecker
    { float CalcBalance(JournalEntry je); }
public class DrEqualsCrChecker : IDrEqualsCrChecker
{
    public float CalcBalance(JournalEntry je)
    {
        var balance = je.DebitAmount + je.CreditAmount;
        return balance;
    }
}
```

Here is our test. We need to test the JournalPoster's Post method. We need to isolate it so we can write code to test our code. Go to the Solution Explorer ➤ Right-Click Solution ➤ Add ➤ Project ➤ Visual C# ➤ Test ➤ Unit Test Project ➤ Name it after the Project and append **.UnitTests** to the name ➤ OK. You get the following by default. We'll change that.

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

Since we want to test the `JournalPoster()`,rename UnitTest1 to JournalPosterTests. Rename TestMethod1 following the naming convention of methodname_condition_expectation. We need to add a Reference to our Project in our UnitTest project. In the unit test project, Right-Click References ➤ Add Reference ➤ Projects ➤ click the check box of the project. We need to create a Fake debit equals credit checker. Why? We don't want to pass the original one to the JournalPoster. We need to pass a fake one that must always be working because our testing is focussing on the JournalPoster, not the checker.

```
[TestClass]
public class JournalPosterTests
{
    // need to add a Reference to our project
    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void JournalPoster_JEIsAlreadyPosted_ThrowsAnException()
    {  // naming convention: methodname_condition_expection
        var JournalPoster = new JournalPoster(new FakeDrEqualsCrChecke
r());
        var je = new JournalEntry { Posting = new Posting() };
        JournalPoster.Post(je);
    }
    [TestMethod]
    public void JournalPoster_JEIsNotPosted_ShouldSetPostedPropertyOfJ
ournalEntry()
    {
        var JournalPoster = new JournalPoster(new FakeDrEqualsCrChecke
r());
        var je = new JournalEntry();
        JournalPoster.Post(je);
        Assert.IsTrue(je.IsPosted);
        Assert.AreEqual(1, je.Posting.Balance);
        Assert.AreEqual(DateTime.Today.AddDays(1), je.Posting.PostingD
ate);
    }
}
public class FakeDrEqualsCrChecker : IDrEqualsCrChecker
{   // methods defined in an interface must be public
    public float CalcBalance(JournalEntry je)
        { return 1; } // simple and it will works
}
```

C# Advanced Cheat Sheet (1 of 4)

begincodingnow.com

Generics

C# has two separate mechanisms for writing code that is reusable across different types: inheritance and generics. Whereas inheritance expresses re-usability with a base type, generics express reusability with a "template" that contains "placeholder" types.

```
using System;
using System.Collections.Generic;
namespace Generics
{
    public class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            List<Customer> myCustomers = new List<Customer>(); //empty
            myCustomers.Add(new Customer() { Id = 1, Name = "Jack" });
            myCustomers.Add(new Customer() { Id = 2, Name = "Jill" });
            foreach (Customer cust in myCustomers) {
                                    Console.WriteLine(cust.Name); }
        }
    }
}
```

Lists of Objects

Lists were covered briefly in the Basics section of this cheat sheet series, but the example was only a list of integers. Here was have a list of our own objects based on our own class: Customer.

```
class Customer
{
    public int Id = 0;
    public string Name = "";
    public string Status { get; set; }
}
class Repository
{
    private static List<Customer> privatecust = new List<Customer>();
    public static IEnumerable<Customer> Customers {
        get { return privatecust; }
    }
    public static void AddCustomers(Customer customer) {
        privatecust.Add(customer);
    }
    public static int NumberOfCustomers {
        get { return privatecust.Count; }
    }
}
class Program
{
    static void Main(string[] args)
    {
        var cust1 = new Customer { Id = 1, Name = "Joe",
            Status = "Active" };
        var cust2 = new Customer { Id = 1, Name = "Sally",
            Status = "Active" };
        Repository.AddCustomers(cust1);
        Repository.AddCustomers(cust2);
        foreach (Customer cust in Repository.Customers)
        { Console.WriteLine($"Name: {cust.Name} Id: {cust.Id} " +
            $"Status: {cust.Status}"); }
        Console.WriteLine($"Number of customers: " +
            $"{Repository.NumberOfCustomers}");
    }
}
```

Here is another example of a list of objects, without the Repository.

```
class Program
{
    public class Customer
    {   // mix fields with a property just for demonstration
        public int Id = 0;
        public string Name = "";
        public string Status { get; set; }
    }
    static void Main(string[] args)
    {
        var customers = new List<Customer>
        {   // using object initialization syntax here
            new Customer { Id = 4, Name = "Jack", Status = "Active"},
            new Customer { Name = "Sally", Status = "Active"}
        };
        customers.Add(new Customer { Name = "Sam" });
        foreach (Customer cust in customers)
                    { Console.WriteLine(cust.Id + " " + cust.Name +
                                              " " + cust.Status); }
    }
}
```

Delegates

A delegate is an object that “holds” one or more methods. A delegate is a reference to a function or ordered list of functions with a specific signature. You can “execute” a delegate and it will execute the method or methods that it “contains” (points to). A delegate is a user-defined reference type, like a class. You can create your own delegate or use the generic ones: `Func<>` and `Action<>`. First, we’ll create our own.

```
class Program
{
    delegate int Multiplier(int x);  // type declaration
    static void Main()
    {
        Multiplier t = Cube; // Create delegate instance
        // by assigning a method to a delegate variable.
        int result = t(2); // Invoke delegate: t(3)
        Console.WriteLine(result); // 8
    }
    static int Cube(int x) => x * x * x;
}
```

Here is second example.

```
using System;
namespace ReturnValues
{
    // Illustrated C# 7 Fifth Edition page 361
    delegate int MyDel(); // Declare delegate with return value.
    class MyClass
    {
        private int IntValue = 5;
        public int Add2() { IntValue += 2; return IntValue; }
        public int Add3() { IntValue += 3; return IntValue; }
    }
    class Program
    {
        static void Main()
        {
            MyClass mc = new MyClass();
            MyDel mDel = mc.Add2; // Create initialize delegate.
            mDel += mc.Add3; // Add a method.
            mDel += mc.Add2; // Add a method.
            Console.WriteLine($"Value: { mDel() }");  // output 12
        }
    }
}
```

Here is a more realistic example of delegates. Here we create a multicast delegate. The consumer of our code is the method `Main()`. We have an object that we need to “process” with several methods in order, and we also want the code to be **extensible** so the consumer can add their own methods in `Main()` to the list of our methods.

```
class MyClass
{
    public string MyString { get; set; }
    public int MyInt { get; set; }

    public static MyClass MyClassDoMethod()
    {
        return new MyClass();  // we don't use these
    }
}
```

Our code has 3 methods that act upon the above class. They are: `AddOne()`, `DoubleIt()` and `AppendString()`.

```
class MyClassMethods
{
    public void AddOne(MyClass mc)
    {  // here we do something with the object mc
        mc.MyInt = mc.MyInt + 1;
        Console.WriteLine("AddOne: " + mc.MyString + " " + mc.MyInt);
    }
    public void DoubleIt(MyClass mc)
    {
        mc.MyInt = mc.MyInt * 2;
        Console.WriteLine("DoubleIt:" + mc.MyString + " " + mc.MyInt);
    }
    public void AppendString(MyClass mc)
    {
        mc.MyString = mc.MyString + " appending string now ";
        Console.WriteLine("AppendString: " + mc.MyString + " "
                                                   + mc.MyInt);
    }
}
class MyClassProcessor
{
    public int MyAmount { get; set; }
    public delegate void MyClassMethodHandler(MyClass myclass);

    public void Process(MyClassMethodHandler methodHandler)
    {   // methodHandler is a delegate
        // instantiate with object initialization syntax
        var myclass = new MyClass { MyString = "In Process method ",
                                        MyInt = 1 };
        methodHandler(myclass);
        // we do not define the methods we want to run here because
        // we are going to let the consumer define that.
    }
}
class Program
{
    static void Main(string[] args)
    {
        var myclassprocessor = new MyClassProcessor();
        var myclassmethods = new MyClassMethods();
        MyClassProcessor.MyClassMethodHandler
                           methodHandler = myclassmethods.AddOne;
        // MyClassMethodHandler is a delegate (multicast)
        // methodHandler is pointer to a group of functions (delegate)
        methodHandler += myclassmethods.DoubleIt;
        methodHandler += FromConsumerMinusThree;
        methodHandler += myclassmethods.AppendString;

        // Process() takes a delegate
        myclassprocessor.Process(methodHandler);
    }
    static void FromConsumerMinusThree(MyClass myC)
    {
        myC.MyInt = myC.MyInt - 3;
        Console.WriteLine("FromConsumerMinusThree: " + myC.MyString +
                                       myC.MyInt);
    }
}
```

Output:

```
AddOne: inside Process method  2
DoubleIt: inside Process method  4
FromConsumerMinusThree: inside Process method 1
AppendString: inside Process method  appending string now  1
```

Func<> and Action<>

In .NET we have 2 delegates that are generic: `Action<>` and `Func<>`. Each also come in a non-generic form. Modifying the above program requires us to use `Action<>` and introducing a new processor (we’ll call it `MyClassGenericProcessor`) and removing our custom delegate in there and adding `Action<>.` Also in the `Main()` program we need to change the first line and the third line of code.

C# Advanced Cheat Sheet (2 of 4)

begincodingnow.com

Func<> and Action<> continued…

```
class MyClassGenericProcessor
{
    public int MyAmount { get; set; }
    // public delegate void MyClassMethodHandler(MyClass myclass);

    public void Process(Action<MyClass> methodHandler)
    {   // methodHandler is a delegate
        // instantiate with object initialization syntax
        var myclass = new MyClass { MyString = "in Process method ",
                                                      MyInt = 1 };
        methodHandler(myclass);
        // we do not define the methods we want to run here because
        // we are going to let the consumer define that.
    }
}
```

Below is a partial listing of our `Main()` program showing the changes.

```
var myclassprocessor = new MyClassGenericProcessor(); // generics
var myclassmethods = new MyClassMethods();
Action<MyClass> methodHandler = myclassmethods.AddOne;
```

Anonymous Types

An anonymous type is a simple class created on the fly to store a set of values. To create an anonymous type, you use the **new** keyword followed by an object initializer **{ }**, specifying the properties and values the type will contain. Anonymous types are used in LINQ queries.

```
static void Main(string[] args)
{
    var person = new { Name = "Bob", Number = 32 };
    Console.WriteLine($"Name: {person.Name} " +
        $"Number: {person.Number}");
    // output: Name: Bob Number: 32
}
```

Here is another example.

```
class Program
{
    static void Main(string[] args)
    {
        var person = new
        {
            Name = "John",
            Age = 29,
            Major = "Computers"
        };
        Console.WriteLine($"{ person.Name }, Age { person.Age }, "
                          + $"Major: {person.Major}");
        // the code below produces the same results
        string Major = "Computers";
        var guy = new { Age = 29, Other.Name, Major };
        Console.WriteLine($"{guy.Name }, Age {guy.Age }, "
                            + $"Major: {guy.Major}");
        // John, Age 29, Major: Computers
    }
}
class Other
{
    // Name is a static field of class Other
    static public string Name = "John";
}
```

Lambda

A lambda expression is an unnamed method written in place of a delegate instance. A lambda expression is an anonymous method that has no access modifier, no name and no return statement. We have code below that we can re-factor using a lambda expression. The => is read as "goes to".

```
class Program
{
    delegate int MyDel(int InParameter); // custom delegate
    static void Main(string[] args)
    {
        MyDel AddTwo = x => x + 2;
        Func<int, int> AddThree = number => number + 3;
        Console.WriteLine(AddOne(0));
        Console.WriteLine(AddTwo(0));
        Console.WriteLine(AddThree(0));
    }
    static int AddOne(int number)
    {
        return number + 1;
    }
}
```

Here is another example.

```
static void Main(string[] args)
{
    Console.WriteLine(Square(3));  // 9
    Func<int, int> squareDel = Square;
    Console.WriteLine(squareDel(3));  // 9
    Func<int, int> squareLambda = m => m * m;
    Console.WriteLine(squareLambda(3));  // 9
    Func<int, int, long> multiplyTwoInts = (m, n) => m * n;
    Console.WriteLine(multiplyTwoInts(3,4));  // 12
}
static int Square(int number)
{
    return number * number;
}
```

Here is another example that is more realistic. Here we have a list of Products. We also have a repository of products. We use object initialization syntax to initialize the list with a series of products. `FindAll()` takes a predicate. A predicate is something that evaluates to true or false.

```
class Product
{
    public string Title { get; set; }
    public int Price { get; set; }
}

class ProductRepository
{
    public List<Product> GetProducts()
    {
        return new List<Book>
            {
                new Product () { Title ="product 1", Price = 5},
                new Product () { Title = "product 2", Price = 6 },
                new Product () { Title = "product 3", Price = 17 }
            };
    }
}
class Program
{
    static void Main(string[] args)
    {
        var products = new ProductRepository().GetProducts();
        List<Product> cheapProducts = products.FindAll(b =>
                                                 b.Price < 10);
        foreach (var product in cheapProducts)
        {
            Console.WriteLine(product.Title + " $" + product.Price);
        }
    }
}
```

You can use a lambda expression when argument requires a delegate.

Events

1. define a delegate (define signature) or use EventHandler<>
2. define an event based on that delegate (ItemProcessed in this case)
3. raise the event

Here is an example program that uses events.

```
public class Item
{
    public string Name { get; set; }  // a property
}
public class ItemEventArgs : EventArgs
{
    public Item Item { get; set; }
}
public class ItemProcessor
{
    // public delegate void ItemProcessedEventHandler(object source,
    //                                              ItemEventArgs args);
    public event EventHandler<ItemEventArgs> ItemProcessed;

    public void ProcessItem(Item item)
    {
        Console.WriteLine("Processing Item...");
        Thread.Sleep(1500); // delay 1.5 seconds
        OnItemProcessed(item);
    }
    protected virtual void OnItemProcessed(Item item)
    {
        ItemProcessed?.Invoke(this, new ItemEventArgs() { Item = item
});
        //  if (ItemProcessed != null)
        //  ItemProcessed(this, new ItemEventArgs() { Item = item });
    }
}
public class SubscriberOne
{
    public void OnItemProcessed(object source, ItemEventArgs args)
    {  // maybe send an email
        Console.WriteLine("SubscriberOne: " + args.Item.Name);
    }
}
class SubscriberTwo
{
    public void OnItemProcessed(object source, ItemEventArgs args)
    {  // maybe send SMS (text message)
        Console.WriteLine("SubscriberTwo: " + args.Item.Name);
    }
}
```

Here is the main program.

```
class Program
{
    static void Main(string[] args)
    {
        var item = new Item() { Name = "Item 1 name" };
        var itemProcessor = new ItemProcessor();  // publisher
        var subscriberOne = new SubscriberOne();  // subscriber
        var subscriberTwo = new SubscriberTwo();  // subscriber

        Console.WriteLine("Beginning program EventsExample...");

        // itemProcessed is a list of pointers to methods
        itemProcessor.ItemProcessed += subscriberOne.OnItemProcessed;
        itemProcessor.ItemProcessed += subscriberTwo.OnItemProcessed;

        itemProcessor.ProcessItem(item);
    }
}
```

Attributes

Attributes allow you to add metadata to a program's assembly. Attribute names use Pascal casing and end with the suffix Attribute. An attribute section consists of square brackets enclosing an attribute name and sometimes a parameter list. A construct with an attribute applied to it is said to be decorated, or adorned, with the attribute. Use the `[Obsolete]` attribute to mark the old method as obsolete and to display a helpful warning message when the code is compiled.

Preprocessor Directives

C# includes a set of preprocessor directives that are mainly used for conditional compilation. The directives `#region` and `#endregion` delimit a section of code that can be expanded or collapsed using the outlining feature of Visual Studio and can be nested within each other.

C# Advanced Cheat Sheet (3 of 4)

begincodingnow.com

Extension Methods

Extension methods allow an existing type to be extended with new methods, without altering the definition of the original type. An extension method is a static method of a static class, where the `this` modifier is applied to the first parameter. The type of the first parameter will be the type that is extended. Extension methods, like instance methods, provide a way to chain functions.

```
public static class MyStringExtensions
{
    public static string Shorten(this String str, int numberOfWords)
    {
        if (numberOfWords < 0) throw new
                ArgumentOutOfRangeException("must contain words");
        if (numberOfWords == 0) return "";
        string[] words = str.Split(' ');
        if (words.Length <= numberOfWords) return str;
        return string.Join(" ", words.Take(numberOfWords)) + "...";
    }
}
class Program
{
    static void Main(string[] args)
    {
        string senten = "A very very long sentence...";
        Console.WriteLine("Number of chars: " + senten.Length);
        var shortededSentence = senten.Shorten(10);
        var s2 = shortededSentence.ToUpper();
        var s3 = s2.PadRight(60);
        Console.WriteLine("[" + s3 + "]");
    }
}
```

LINQ

LINQ stands for Language Integrated Query and is pronounced "link." LINQ is an extension of the .NET Framework and allows you to query collections of data in a manner like using SQL to query databases. With LINQ you can query data from databases (LINQ to Entities), collections of objects in memory (LINQ to Objects), XML documents (LINQ to XML), and ADO.NET data sets (LINQ to Data Sets).

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQint
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 6, 47, 15, 68, 23 }; // Data source
            IEnumerable<int> bigNums = // Define & store the query.
                        from n in numbers
                        where n > 30
                        orderby n descending
                        select n;
            foreach (var x in bigNums) // Execute the query.
                Console.Write($"{ x }, ");  // output: 68, 47
        }
    }
}
```

Now let's use a more realistic example. First we'll show the code **without** LINQ, then with LINQ. We have a class of our objects called `Product` and we have a `ProductRepository`.

```
class Product
{
    public string Name { get; set; }
    public float Price { get; set; }
}
class ProductRepository
{
    public IEnumerable<Product> GetProducts()  // method
    {
        return new List<Product>
        {
            new Product() {Name = "P one", Price = 5},
            new Product() {Name = "P two", Price = 9.99f},
            new Product() {Name = "P three", Price = 12},
        };
    }
}
class Program
{
    static void Main(string[] args)
    {
        var products = new ProductRepository().GetProducts();
        var pricyProducts = new List<Product>();
        // ------without LINQ----------------------------
        foreach (var product in products)
        {
            if (product.Price > 10)
                pricyProducts.Add(product);
        }
        // ------without LINQ-----------------------------
        foreach (var product in pricyProducts)
           Console.WriteLine("{0} {1:C}",product.Name, product.Price);
    }
}
```

When you type product followed by the dot, Intelisense gives you a few methods and a long list of extension methods. One extension method is Where<>. Where is asking for a delegate. `Func<Product,bool> predicate.` It points to a method that gets a Product and returns a bool based on the predicate. Whenever we see Func<> as a delegate we can use a Lambda expression such as p => p.Price > 10. Here is the code **with** LINQ.

```
// -----with LINQ-----------------------------------------
var pricyProducts2 = products.Where(p => p.Price > 10);
// -----with LINQ-----------------------------------------
```

The LINQ extension methods can be chained. When we use `Select` in this case we get back a list of strings, not products.

```
// -----LINQ----------------------------------------
var pricyProducts2 = products.Where(p => p.Price > 8)
                             .OrderBy(p => p.Name)
                             .Select(p => p.Name);  // string
// -----LINQ----------------------------------------
foreach (var product in pricyProducts2)
    Console.WriteLine(product);
```

There are several LINQ extension methods beyond `Where()`. A few are listed in the C# comments below. If you only want one Product you can use `Single()` or `SingleOrDefault()`. `Single()` will throw an error InvalidOperationException if it can't find a match. The OrDefault will return null if it can't find a match, which is probably better.

```
var product = products.Single(p => p.Name == "P two");
var product2 = products.SingleOrDefault(p => p.Name == "P unknown");
Console.WriteLine(product.Name);  // P two
Console.WriteLine(product2 == null);  // output: True
var product3 = products.First();
Console.WriteLine(product3.Name); // P one
// FirstOrDefault() Last() LastOrDefult()
// Skip(2).Take(3) will skip the first 2 and take the next 3
// Count() Max() Min() Sum() Average()
// Average(p => p.Price)
```

Nullable Types

Reference types can represent a nonexistent value with a null reference. Normally value types cannot be null, however to represent null in a value type, you must use a special construct called a nullable type which is denoted with a value type immediately followed by the **?** symbol. An important use case for nullable types is when you have a database with a column like MiddleName or BirthDate which may have a null value.

```
static void Main(string[] args)
{
    // DateTime is a value type - cannot be null, but...
    System.Nullable<DateTime> d = null;
    DateTime? dt = null;
    Console.WriteLine("GetValueOrDefault: " + dt.GetValueOrDefault());
    Console.WriteLine("HasValue: " + dt.HasValue);  // property
    // below line causes InvalidOperationException when null
    // Console.WriteLine("Value: " + dt.Value);  // property
    Console.WriteLine(dt);

    // output: 0001-01-01 12:00:00 AM
    // output: False
    // output:
}
```

What about conversions and the null-coalescing operator?

```
// Conversions
DateTime? date = new DateTime(2019, 1, 1);
// DateTime date2 = date;  compiler says cannot convert
DateTime date2 = date.GetValueOrDefault();
Console.WriteLine("date2: " + date2);
DateTime? date3 = date2;
Console.WriteLine(date3.GetValueOrDefault());

// Null Coales Operator: ??
DateTime? date4 = null;
DateTime date5;
// if date has a value use that, otherwise use today
if (date4 != null)
    date5 = date4.GetValueOrDefault();
else
    date5 = DateTime.Today;
// null
date5 = date4 ?? DateTime.Today; // same as if block above
```

When working with nullable types, `GetValueOrDefault()` is the preferred way of doing things.

Dynamics

Programming languages are either static or dynamic. C# and Java are static, but Ruby, JavaScript and Python are dynamic. With static languages the types are resolved at compile time, not at run time. The CLR (.NET's virtual machine) takes compiled code (verified by the compiler) which is in Intermediate language (IL) and converts that to machine code at runtime. Runtime checking is performed by the CLR. Runtime type checking is possible because each object on the heap internally stores a little type token. You can retrieve this token by calling the `GetType` method of object (reflection). With C# dynamics and the keyword dynamic, we don't need to use reflection. Much cleaner code results. When converting from dynamic to static types, if the runtime type of the dynamic object can be implicitly converted to the target type we don't need to cast it.

```
dynamic name = "Bob";
name = 19;  // this works because name is dynamic!
name++;
Console.WriteLine(name);  // 20
dynamic a = 4, b = 5;
var c = a + b; // c becomes dynamic
Console.WriteLine(c);  // 9
int i = 7;
dynamic d = i;
long l = d;
Console.WriteLine(l);  //
```

C# Advanced Cheat Sheet (4 of 4)

Asynchronous continued…

Asynchronous

In the synchronous model the program executes line by line, but in the asynchronous model (e.g. media players, web browsers), responsiveness is improved. In .NET 4.5 (in 2012) Microsoft introduced a new a new asynchronous model, instead of multi-threading and call-backs. It uses `async` and `await` keywords. In our example we have a WPF program that has 2 blocking operations (downloading and writing). You can only use the `await` operator inside an `async` method. `Async` affects only what happens inside the method and has no effect on a method's signature or public metadata.

```
using System.IO;
using System.Net;
using System.Threading.Tasks;
using System.Windows;
namespace AysnchronousProgramming
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private async void Button_Click(object sender,
                                   RoutedEventArgs e)
        {
            await DownloadHtmlAsync("http://begincodingnow.com");
        }
        public async Task DownloadHtmlAsync(string url)
        {   // decorate method async, use Task, and only by convention
            // put "Async" at end of the method name.
            var webClient = new WebClient();
            // use TaskAsync not Async, and await is a compiler marker
            var html = await webClient.DownloadStringTaskAsync(url);
            using (var streamWriter = new
                          StreamWriter(@"c:\temp\result.html"))
            {   // use the Async one: WriteAsync and add await
                await streamWriter.WriteAsync(html);
            }
            MessageBox.Show("Finished downloading","Asynch Example");
        }
        public void DownloadHtml(string url)
        {   // NOT asynchronous! - just shown here for comparison
            var webClient = new WebClient();
            var html = webClient.DownloadString(url);

            using (var streamWriter = new
                           StreamWriter(@"c:\temp\result.html"))
            {
                streamWriter.Write(html);
            }
        }
    }
}
```

Now we will modify our program. The message box "Waiting…" executes **immediately**. We can execute other code here. Another message box executes **after** the blocking operation completes.

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    //await DownloadHtmlAsync("http://begincodingnow.com");
    // Note: if we use await we must use async in method definition.
    // var html = await GetHtmlAsync("http://begincodingnow.com");
    var getHtmlTask = GetHtmlAsync("http://begincodingnow.com");
    // executes immediately
    MessageBox.Show("Waiting for task to complete...");
    var html = await getHtmlTask;
    // executes after html is downloaded
    MessageBox.Show(html.Substring(0, 500));
}
public async Task<string> GetHtmlAsync(string url)
{
    var webClient = new WebClient();
    return await webClient.DownloadStringTaskAsync(url);
}
```

Exception Handling

We write exception handling code to avoid those Unhandled Exception messages when the program crashes. We can use a Try Catch block. The four keywords of exception handling are: try, catch, finally and throw. The first code example crashes with an unhandled exception. In the second example we handle the exception.

```
public class Calculator
{
    public int Divide(int numerator, int denominator)
    {
        return numerator / denominator;
    }
}
class Program
{
    static void Main(string[] args)
    {
        var calc = new Calculator();
        // Unhandled Exception: System.DivideByZeroException:
        // Attempted to divide by zero. CRASHES !!
        var result = calc.Divide(89, 0);
    }
}
```

Let's refactor our Main() method to use a try catch block.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error!"); // F9, F5
    }
}
```

To implement multiple catch blocks set a break point (with F9) and run it in debug mode (F5). Place your cursor on "ex" and click the small right-arrow icon in the pop-up to bring up more details. Properties have the wrench icon. Look at Message, Source (the DLL or assembly), StackTrace (sequence of method calls in the reverse order – click the magnifier icon), TarketSite (method where exception happened) and the others.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error! " +
            ex.Message);  // F9, F5
        // Unexpected error! Attempted to divide by zero.
    }
}
```

Multiple catch blocks example below.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
        // type DivideByZeroException and F12
        // for Object Browser to see inheritance
        // hierarchy & click parent (bottom right)
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("Cannot divide by zero. " + ex.Message);
    }
    catch (ArithmeticException ex)
    {
        Console.WriteLine("Arithmetic exception. " + ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error! " +
            ex.Message);  // F9, F5
        // Unexpected error! Attempted to divide by zero.
    }
    finally  // unmanaged resources are not handled by CLR
    {  } // we need to .Dispose() of those here, unless we employ
// the using statement.
}
class Program
{   // we need using System.IO;
    static void Main(string[] args)
    {
        try
        {   // using creates finally block in background
            using (var strmRdr = new StreamReader(@"c:\not.txt")) ;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Unexpected error!");
        }
    }
}
```

One of the new features in C# 6 was exception filters, which are not covered here. They give you more control over your catch blocks and further tailor how you handle specific exceptions.

Recursion

Recursion happens when a method or function calls itself. We must write a condition that checks that the termination condition is satisfied. Below is a program that tells you how many times a number is evenly divisible by a divisor.

```
public static int CountDivisions(double number, double divisor)
{
    int count = 0;
    if (number > 0 && number % divisor == 0)
    {
        count++;
        number /= divisor;
        return count += CountDivisions(number, divisor);
    }
    return count;
}
static void Main(string[] args)
{
    Console.WriteLine("Enter your number: ");
    double number = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter your divisor: ");
    double divisor = Convert.ToDouble(Console.ReadLine());
    int count = CountDivisions(number, divisor);
    Console.WriteLine($"Total number of divisions: {count}");
    Console.ReadKey();
}
```