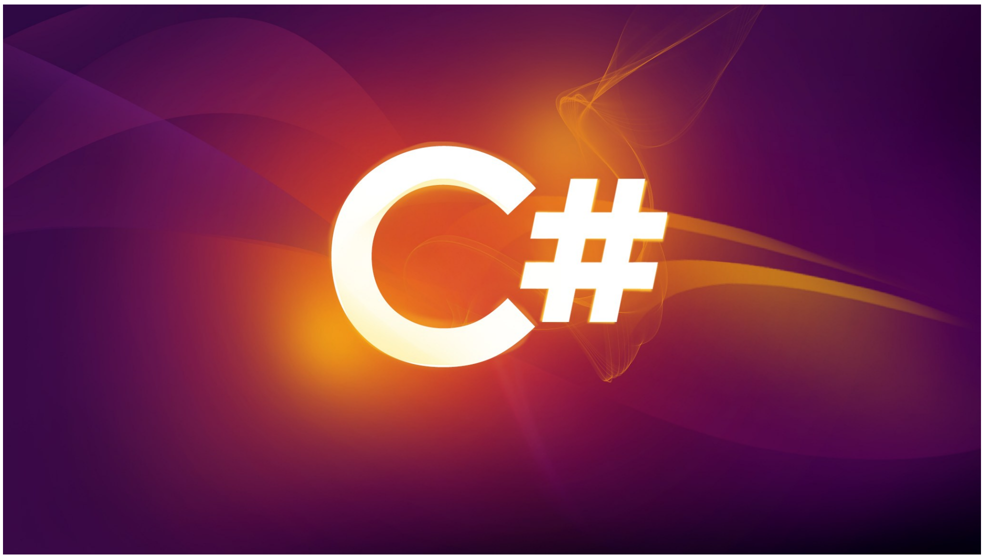
C# Cheatsheet



Index

Comments

```
// My comments about the class name could go here...

// My comments about the class name could go here...
// Add as many lines as you would like
// ...Seriously!

/*
My comments about the class name could go here...
Add as many lines of comments as you want
        ...and use indentation, if you want to!
*/
```

Commented Tasks highlighter

```
//TODO: Change "world" to "universe"

//HACK: Don't try this at home....

//NOTE: Don't try this at home....

//UNDONE: Don't try this at home....
```

back to top

Scopes

Block/Function Scoped

Class Scoped

```
using System;

namespace VariableScope
{
    class Program
    {
        private static string helloClass = "Hello, class!";

        static void Main(string[] args)
        {
            string helloLocal = "Hello, local!";
            Console.WriteLine(helloLocal); // Hello local
            Console.WriteLine(Program.helloClass); // Hello Class
            DoStuff();
        }

        static void DoStuff()
        {
            Console.WriteLine("A message from DoStuff: " + Program.helloClass); // Hello Class
        }
    }
}
```

back to top

Functions

Syntax

```
<visibility> <return type> <name>(<parameters>)
{
        <function code>
}
```

`If you don't define any, then the function will be private`

`void` means it returns nothing.

Example

```
public int AddNumbers(int number1, int number2)
{
    int result = number1 + number2;
    return result;
}
```

Params Keyword

We can create a function and pass parameters like this

```
static void GreetPersons(string[] names) { }
```

However, calling it would be a bit clumsy. In the shortest form, it would look like this:

```
GreetPersons(new string[] { "John", "Jane", "Tarzan" });
```

By Adding the keyword params we can call it like this

```
static void GreetPersons(params string[] names) { }
```

And call it like this,

```
GreetPersons("John", "Jane", "Tarzan");
```

Another advantage of using the params approach, is that you are allowed to pass `zero parameters` to it as well.

back to top

DataTypes

Type	Description	Examples
`int`	Integer (whole numbers)	103, 12, 5168
`double`	64 bit floating-point number	3.14, 3.4e38
`Float`	Floating-point number	3.14, 3.4e38
`Decimal`	Decimal number (higher precision)	1037.196543
`Bool`	Boolean	true, False
`String`	String	"Hello World"
`byte`	8-bit unsigned integer	0 to 255
`char`	16-bit Unicode character	"A"
`long`	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

back to top

Operators

Operator	Description	Examples
`=`	Assigns a value to a variable.	i=6
`+`	Adds a value or variable.	i=5+5
`-`	Subtracts a value or variable.	i=5-5
`*`	Multiplies a value or variable.	i=5*5
`/`	Divides a value or variable.	i=5/5
`+=`	Increments a variable.	i += 1
`-=`	Decrements a variable.	i -= 1
`==`	Equality. Returns true if values are equal.	if (i==10)
`!=`	Inequality. Returns true if values are not equal.	if (i!=10)
`<`	Less Than	if (i<10)
`<=`	Greater Than	if (i>10)
`>=`	Less Than or Equal to	if (i<=10)
`+`	Adding strings (concatenation).	"Hello " + "World"
`.`	Dot. Separate objects and methods.	DateTime.Hour
`()`	Parenthesis. Groups values.	(i+5)

Operator	Description	Examples
( )	Parenthesis. Passes parameters.	x=Add(i,5)
[ ]	Brackets. Accesses values in arrays or collections.	name[3]
!	Not. Reverses true or false.	if (!ready)
&&	Logical AND.	if (ready && clear)
`		`

back to top

Convert Data Types

Method	Description	Examples

AsInt(),
IsInt()
	Converts a string to an integer.	`if (myString.IsInt())`
		`{myInt=myString.AsInt();`

AsFloat(), IsFloat()	Converts a string to a floating-point number.	`if (myString.IsFloat())`
		`{myFloat=myString.AsFloat();}`

AsDecimal(), IsDecimal()	Converts a string to a decimal number..	`if (myString.IsDecimal())`
		`{myDec=myString.AsDecimal();}`

AsDateTime(), IsDateTime()	Converts a string to an ASP.NET DateTime type.	`myString="10/10/2012";`
		`myDate=myString.AsDateTime();`

AsBool(),
IsBool()
	Converts a string to a Boolean..	`myString="True";`
		`myBool=myString.AsBool();`

ToString()	Converts any data type to a string.	`myInt=1234;`
		`myString=myInt.ToString();`

back to top

Define Variables

int i, j, k; char c, ch; float f, salary; double d;

Type	Name
int	i, j, k;
char	c, ch;
float	f, salary;
double	d;

Initialise Varaiable

```
variable_name = value;
```

You can also initialize a varaible at the same time you define it.

```
int d = 3, f = 5;    /* initializing d and f. */
byte z = 22;         /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';        /* the variable x has the value 'x'. */
```

Constants

```
// const <data_type> <constant_name> = value;

const double pi = 3.14159;
```

back to top

Casting User input

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

? == Nullable Types

C# provides a special data types, the nullable types, to which you can assign normal range of values as well as null values.

```
// < data_type> ? <variable_name> = null;

int? num1 = null;
```

back to top

Classes

- A Class is a group of related methods and variables.
- On this object, you use the defined methods and variables.
- You can create as many instances of your class as you want

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare a variable of type Car
            Car car;

            // Creates an new instance
            car = new Car("Red");

            // New instance gives us method access
            Console.WriteLine(car.Describe());

            car = new Car("Green");
            Console.WriteLine(car.Describe());

            Console.ReadLine();

        }
    }

    class Car
    {
        private string color;

        public Car(string color)
        {
            this.color = color;
        }

        public string Describe()
        {
            return "This car is " + Color;
        }

        public string Color
        {
            get { return color; }
            set { color = value; }
        }
    }
}
```

- new class called Car
- It defines a single variable, called color, which of course is used to tell the color of our car.
- Our Car class defines a constructor
- It takes a parameter which allows us to initialize Car objects with a color.
- The Describe() method allows us to get a nice message
- Returns a string

back to top

Properties

- Properties allow you to control the accessibility of a class's variables

- Recommended way to access variables from the outside in an object oriented programming language like C#
- A property is much like a combination of a variable and a method - it can't take any parameters, but you are able to process the value before it's assigned to our returned variable
- A property consists of 2 parts, a get and a set method, wrapped inside the property:

```
private string color;

public string Color
{
    get { return color; }
    set { color = value; }
}
```

> The `get` method should `return` the variable
> The `set` method should `assign a value` to it.

However you can add logic and conditionals to the `Get` and `Set` methods.

```
public string Color
{
    get
    {
        return color.ToUpper();
    }
    set
    {
        if(value == "Red")
            color = value;
        else
            Console.WriteLine("This car can only be red!");
    }
}
```

back to top

Constructor

```
public Car()
{

}

// Constructor can take parameters
public Car(string color)
{
    // assigns value to `this` instance
    this.color = color;
}
```

back to top

Method Overloading

It allows the programmer to make one or several parameters optional, by giving them a default value. It's especially practical when adding functionality to existing code.

```
class SillyMath
{
    public static int Plus(int number1, int number2)
    {
        return Plus(number1, number2, 0);
    }

    public static int Plus(int number1, int number2, int number3)
    {
        return Plus(number1, number2, number3, 0);
    }

    public static int Plus(int number1, int number2, int number3, int number4)
    {
        return number1 + number2 + number3 + number4;
    }
}
```

back to top

Class Visability

Visability	Definition
public	the member can be reached from anywhere. This is the least restrictive visibility. Enums and interfaces are, by default, publicly visible
protected	members can only be reached from within the same class, or from a class which inherits from this class.
internal	members can be reached from within the same project only.
protected internal	the same as internal, except that classes which inherit from this class can reach its members; even from another project.
private	can only be reached by members from the same class. This is the most restrictive visibility. Classes and structs are by default set to private visibility.

> So for instance, if you have two classes: Class1 and Class2, private members from Class1 can only be used within Class1. You can't create a new instance of Class1 inside of Class2, and then expect to be able to use its private members.

If Class2 inherits from Class1, then only non-private members can be reached from inside of Class2.
back to top

Inheritance

- Is inheritance, the ability to create classes which inherits certain aspects from parent classes.

```
Classes:
Inheritance
One of the absolute key aspects of Object Oriented Programming (OOP), which is the concept that C# is built upon, is

This subject can be a bit difficult to comprehend, but sometimes it help with some examples, so let's start with a si



public class Animal
{
    public void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}

// Dog Inherits from Animal base classs
public class Dog : Animal
{

}
```

Then we can Create a new Animal and a new Dog. They will both have access to the Greet method though Inheritance.

```
Animal animal = new Animal();
animal.Greet();
Dog dog = new Dog();
dog.Greet();
```

back to top

Virtual and Override

Here we take the same principle as above however we override the Greet Method as defined in the Animal Base class. To allow this we have to add the keyword Virtual to the greet method. This then allows us to override that method in the child class dog.

```
public class Animal
{
    public virtual void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}

public class Dog : Animal
{
    public override void Greet()
    {
        Console.WriteLine("Hello, I'm a dog!");
    }
}
```

back to top

Base

In C#, you are not allowed to override a member of a class unless it's marked as `virtual` . If you want to, you can still access the inherited method, even when you override it, using the `base keyword` .

```
public override void Greet()
{
    /*
        you can still access the base class
        by using the base keyword
    */

    base.Greet();
    Console.WriteLine("Yes I am - a dog!");
}
```

back to top

Abstract Class

- Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy
- You `can't` create an instance of them.

```
// We can't new up an abstract class
abstract class FourLeggedAnimal
    {
        public virtual string Describe()
        {
            return "Not much is known about this four legged animal!";
        }
    }
 // We can inherit from it !
    class Dog : FourLeggedAnimal
    {

    }
```

back to top

Abstract methods

- Abstract methods are only allowed within abstract classes.
- We define them as abstract but without any code
- Then in our inherited class we override that method description .

```
  abstract class FourLeggedAnimal
    {
        /*
           Define the abstract method definition
           from within the abstract class
        */
        public abstract string Describe();
    }


    class Dog : FourLeggedAnimal
    {
         /*
            Override the abstract method definition
            from within the sub class and add
            code block
        */
        public override string Describe()
        {
            return "I'm a dog!";
        }
    }
```

back to top

Interfaces

- Interfaces are similar to Abstract Classes
- No Instances are created.

> NO METHODS ARE ALLOWED AT ALL

- All Interfaces are Public

> There are NO access modifiers (public, private, protected etc.),
> because they are not allowed.

- You can implement as many interfaces as you want to into a single class

```
    class Program
    {
        static void Main(string[] args)
        {
            List<Dog> dogs = new List<Dog>();
            dogs.Add(new Dog("Fido"));
            dogs.Add(new Dog("Bob"));
            dogs.Add(new Dog("Adam"));
            dogs.Sort();
            foreach(Dog dog in dogs)
                Console.WriteLine(dog.Describe());
            Console.ReadKey();
        }
    }

interface IAnimal
    {
        string Describe();

        string Name
        {
            get;
            set;
        }
    }

    class Dog : IAnimal, IComparable
    {
        private string name;

        public Dog(string name)
        {
            this.Name = name;
        }

        public string Describe()
        {
            return "Hello, I'm a dog and my name is " + this.Name;
        }


        // This method comes from the IComparable interface
        public int CompareTo(object obj)
        {
            if(obj is IAnimal)
                return this.Name.CompareTo((obj as IAnimal).Name);
            return 0;
        }

        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
```

back to top

Partial Classes

- A class can be broken down into several smaller files.
- These are within the same namespace and they are called with the same name.

```
// PartialClass1.cs

using System;

namespace PartialClasses
{
   public partial class PartialClass
   {
       public void HelloWorld()
       {
           Console.WriteLine("Hello, world!");
       }
   }
}
```

```
// PartialClass2.cs

using System;

namespace PartialClasses
{
  public partial class PartialClass
  {
      public void HelloUniverse()
      {
          Console.WriteLine("Hello, universe!");
      }
  }
}
```

- We can call them as one class as if they are in one file.

```
// Program.cs

using System;

namespace PartialClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a new instance
            PartialClass pc = new PartialClass();
            pc.HelloWorld();
            pc.HelloUniverse();
        }
    }
}
```

back to top

Collections

Arrays

1. An array stores a fixed-size sequential collection of elements of the same type.

Common Array Methods

Method
length
forEach
map
filter
reduce
some
every
find
findIndex
includes
sort
concat

Declaration

```
// datatype[] arrayName;
double[] balance;
```

Initialise

```
double[] balance = new double[10];
```

Assign Values

```
double[] balance = new double[10];
balance[0] = 4500.0;

double[] balance = { 2340.0, 4523.69, 3421.0};

int [] marks = new int[5]  { 99,  98, 92, 97, 95};

int [] marks = new int[]  { 99,  98, 92, 97, 95};

// You can copy an array variable into another target array variable. In such case, both the target and source point

int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

Accessing Elements in Array

```
double salary = balance[9];
```

back to top

ForLoop

```
 int []  n = new int[10]; /* n is an array of 10 integers */
         int i;

         /* initialize elements of array n */
         for ( i = 0; i < 10; i++ ) {
            n[ i ] = i + 100;
         }
```

ForEach

```
 int []  n = new int[10]; /* n is an array of 10 integers */
         int i;

          /* output each array element's value */
         foreach (int j in n ) {
            int i = j-100;
            Console.WriteLine("{0}", j);
         }
```

back to top

Lists

- C# implements the IList Interface
- The most popular is the generic list `List<T>`
- `List<T>` is type-safe.
- List is much like an ArrayList class
- List can do a lot of the same stuff as an Array (which also implements the IList interface by the way).
- List is simpler and easier to work with

> No preset size
> Automatically resizes.

Create a list

```
// Creates an Empty List
List<string> listOfStrings = new List<string>();
```

Add

```
listOfStrings.Add("a string");
```

Initialize a List with items

```
List<string> listOfNames = new List<string>()
{
    "John Doe",
    "Jane Doe",
    "Joe Doe"
};
```

Insert

```
List<string> listOfNames = new List<string>()
{
    "Joe Doe"
};
// Insert at the top (index 0)
listOfNames.Insert(0, "John Doe");
// Insert in the middle (index 1)
listOfNames.Insert(1, "Jane Doe");
```

AddRange of Items

```
listOfNames.AddRange(new string[]
            {
                "Jenna Doe",
                "Another Doe"
            });
```

Remove

```
List<string> listOfNames = new List<string>()
{
    "John Doe",
    "Jane Doe",
    "Joe Doe",
    "Another Doe"
};

listOfNames.Remove("Joe Doe");
```

Others

Tables	Are
RemoveAt()	listOfNames.RemoveAt(0);
using count	listOfNames.RemoveAt(listOfNames.Count - 1);
RemoveAll()	are neat
Sort()	listOfNames.Sort();

Iterate over lists with:

- Loops
- forEach

back to top

Dictionaries

- Dictionaries in C# all implement the `IDictionary interface.`
- The most common used is the `generic Dictionary`
- Dictionary<TKey, TValue>

Difference between Lists vr Dictionaries

Lists

- Items in a `specific order`
- Accessed by `numerical Index` list[1]

Dictionaries

- Stored as `Key Value Pairs`

- Accessed by a `Unique Key`.

Create

```
Dictionary<string, int> users = new Dictionary<string, int>();
```

Add

```
users.Add("John Doe", 42);
```

Add Many

```
Dictionary<string, int> users = new Dictionary<string, int>()
{
    { "John Doe", 42 },
    { "Jane Doe", 38 },
    { "Joe Doe", 12 },
    { "Jenna Doe", 12 }
};
```

ContainsKey

```
if(users.ContainsKey(key))
    Console.WriteLine("John Doe is " + users[key] + " years old");
```

Print Dictionary

```
Dictionary<string, int> users = new Dictionary<string, int>()
{
    { "John Doe", 42 },
    { "Jane Doe", 38 },
    { "Joe Doe", 12 },
    { "Jenna Doe", 12 }
};
foreach (KeyValuePair<string, int> user in users)
{
    Console.WriteLine(user.Key + " is " + user.Value + " years old");
}
```

Order by

```
Dictionary<string, int> users = new Dictionary<string, int>()
{
    { "John Doe", 42 },
    { "Jane Doe", 38 },
    { "Joe Doe", 12 },
    { "Jenna Doe", 12 }
};
foreach (KeyValuePair<string, int> user in users.OrderBy(user => user.Value))
{
    Console.WriteLine(user.Key + " is " + user.Value + " years old");
}
```

back to top

Strings

Create a String

```
 //from string literal and string concatenation
         string fname, lname;
         fname = "Rowan";
         lname = "Atkinson";

//or by using the string constructor
 string greetings = new string(letters);
```

Common String Methods

Returns :

`0` = true

`1` = false

Method	Code	Comments
`Clone()`	firstname.Clone()	Make clone of string.
`CompareTo()`	firstname.CompareTo(lastname)	Compare two strings and returns integer value as output. It returns 0 for true and 1 for false.
`Contains()`	firstname.Contains("ven")	The C# Contains method checks whether specified character or string is exists or not in the string value.
`EndsWith()`	firstname.EndsWith("n")	This EndsWith Method checks whether specified character is the last character of string or not.
`Equals()`	firstname.Equals(lastname)	The Equals Method in C# compares two string and returns Boolean value as output.
`GetHashCode()`	firstname.GetHashCode()	This method returns HashValue of specified string.
`GetType()`	firstname.GetType()	t returns the System.Type of current instance.
`IndexOf()`	firstname.IndexOf("e")	Returns the index position of first occurrence of specified character.
`ToLower()`	firstname.ToLower()	Converts String into lower case based on rules of the current culture.
`ToUpper()`	firstname.ToUper()	Converts String into upper case based on rules of the current culture.
`Insert()`	firstname.Insert(0, "Hello")	Insert the string or character in the string at the specified position.
`IsNormalized()`	firstname.IsNormalized()	This method checks whether this string is in Unicode normalization form C.
`LastIndexOf()`	firstname.LastIndexOf("e")	This method checks whether this string is in Unicode normalization form C.
`Length`	firstname.Length	It is a string property that returns length of string.
`Remove()`	firstname.Remove(5)	This method deletes all the characters from beginning to specified index position.
`Replace()`	firstname.Replace('e','i')	This method replaces the character.
`Split()`	string[] split = firstname.Split(new char[] { 'e' });	This method splits the string based on specified value.
`StartsWith()`	firstname.StartsWith("S")	It checks whether the first character of string is same as specified character.
`Substring()`	firstname.Substring(2,5)	This method returns substring.
`ToCharArray()`	firstname.ToCharArray()	Converts string into char array.
`Trim()`	firstname.Trim()	It removes extra whitespaces from beginning and ending of string.

back to top

Structs or Structures

Structs are different from Classes because

1. classes are reference types and structs are value types
2. structures do not support inheritance
3. structures cannot have default constructor.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

Title Author Subject Book ID

Defining a struct

```
struct Books {
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

Use Struct

```
public class testStructure {
   public static void Main(string[] args) {
      Books Book1;   /* Declare Book1 of type Book */


      /* book 1 specification */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;



      /* print Book1 info */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}", Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);



      Console.ReadKey();
   }
}
```

back to top

Enums

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

Declaration

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

int WeekdayStart = (int)Days.Mon;
int WeekdayEnd = (int)Days.Fri;

Console.WriteLine("Monday: {0}", WeekdayStart); // Monday: 1
Console.WriteLine("Friday: {0}", WeekdayEnd);  // Friday: 5
```

back to top

Read File

```
using (var sr = File.OpenText(path))
                {
                    string s;
                    while ((s = sr.ReadLine()) != null) Console.WriteLine(s);
                    {
                    }
                }
```

Write File

```
var path = @"C:\Users\escap\Source\Repos\ConsoleApp1\ConsoleApp1\Main.cs";

            if (!File.Exists(path))
                using (var sw = File.CreateText(path))
                {
                    sw.WriteLine("for (int i = 0; i<length; i++)");
                }
```

Create File