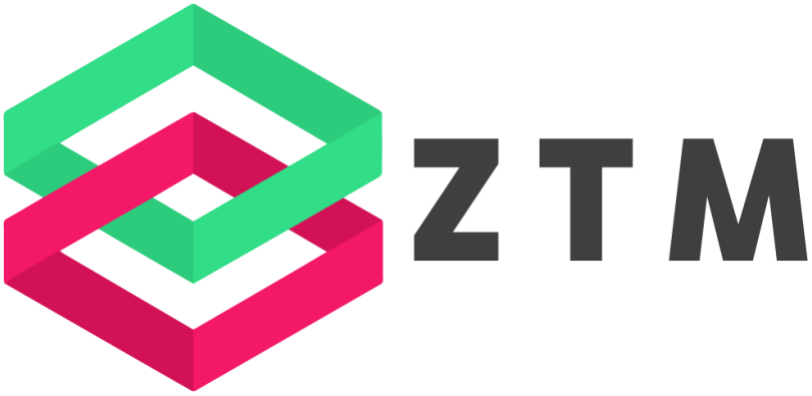

C#
CHEAT SHEET

CLAUDIO BERNASCONI

V1.01

HEEELLLOOOOO!

I'm Andrei Neagoie, Founder and Lead Instructor of the Zero To Mastery Academy.

After working as a Senior Software Developer over the years, I now dedicate 100% of my time to teaching others in-demand skills, help them break into the tech industry, and advance their careers.

In only a few years, **over 1,000,000 students** around the world have taken Zero To Mastery courses and many of them are now working at top tier companies like Apple, Google, Amazon, Tesla, IBM, Facebook, and Shopify, just to name a few.

This cheat sheet, created by our C#/.NET instructor (Claudio Bernasconi) provides you with the key C# syntax and concepts you need to know.

If you want to not only learn C# but also get the exact steps to build your own projects and get hired as a Developer, then check out our Career Paths.

Happy Learning!
Andrei

Founder & Lead Instructor, Zero To Mastery
Andrei Neagoie

C# Cheat Sheet

1. Basic Structure

Every C# program uses this foundational structure. This forms the skeleton of your application. The Program.cs file is the foundation of most C# project types.

```
using System; // 'using' allows for easier access to types in a namespace.

namespace YourNamespace // Namespaces organize code and prevent naming collisions.
{
    class YourClass // A 'class' defines the blueprint of objects.
    {
        static void Main(string[] args) // 'Main' is where the program starts execution.
        {
            Console.WriteLine("Hello, World!"); // Displays text in the console.
        }
    }
}
```

Starting with .NET 5, top-level statements make it possible to simplify the content of the Program.cs file by only having the code within the static void Main method definition:

```
Console.WriteLine("Hello, World");
```

2. Data Types

Every variable and constant in C# has a type, determining the values it can hold and operations that can be performed on it.

- **Value Types**: Directly store the data. Once you assign a value, it holds that data.
  - `int`, `char`, `float` are just a few examples.
- **Reference Types**: Store a memory address. They point to the address of the value.
  - `string`, `class`, `array` are commonly used.

3. Variables

Variables are symbolic names for values. They play a central role in programming.

```
int num = 5; // Declares an integer variable and assigns it the value 5.
string word = "Hello"; // Declares a string variable and assigns it the value "Hello".
```

Hint: Local variables usually start with a lowercase letter (camelCase) following the C# naming convention guidelines.

If the C# compiler can infer (evaluate) the type of a variable, for example, when it is a string literal, we can use **var** instead of explicitly using the variable type.

```
// We can use var because the compiler knows that "..." is a string
var name = "Peter";

// We can use var because we state that we create a new object of type Person.
var person = new Person();
```

The var keyword helps us keep the code short and improves readability.

4. Constants

Constants are immutable. Once their value is set, it can't be changed, ensuring some data remains consistent throughout its use.

```
const int ConstNum = 5; // The value of 'ConstNum' will always be 5.
```

Hint: Constants usually start with an uppercase letter (PascalCase) following the C# naming convention guidelines.

5. Conditional Statements

Allows you to branch your code based on conditions, making your programs dynamic.

```
if (condition) { /*...*/ } // Executes if 'condition' is true.
else if (condition) { /*...*/ } // Additional conditions if the above ones fail.
else { /*...*/ } // Executes if no conditions are met.

switch (variable) // Useful when comparing a single variable to many values.
{
    case value1:
        // Code for value1
        break; // Exits the switch statement.
```

```
    // ... other cases ...
    default:
        // Executes if no other case matches.
        break;
}
```

Hint: If you don't use break or return at the end of a switch case, the next case will also be executed.

6. Loops

Useful for performing repetitive tasks.

```
for (int i = 0; i < 10; i++) { /*...*/ } // Loops 10 times, incrementing 'i' each time.
foreach (var item in collection) { /*...*/ } // Loops over each item in 'collection'.
while (condition) { /*...*/ } // Continues looping as long as 'condition' remains true.
do { /*...*/ } while (condition); // Executes once before checking 'condition'.
```

7. Arrays

Fixed-size collections that hold elements of the same type.

```
int[] numbers = new int[5] {1, 2, 3, 4, 5}; // Declares an array of integers.
```

8. Lists

Like arrays, but can dynamically change in size.

```
using System.Collections.Generic; // Required namespace for Lists.
List<int> list = new List<int>() {1, 2, 3, 4, 5}; // Initializes a List with 5 integers.

list.Add(6); // Add the number 6 to the list
list.Remove(2) // Remove the element at index 2 (0-based) from the list.
```

9. Dictionaries

Associative containers that store key-value pairs.

```
using System.Collections.Generic;
Dictionary<string, int> dict = new Dictionary<string, int>()
{
    {"one", 1}, // "one" is the key, 1 is its associated value.
    {"two", 2}
};

dict.Add("key", "value"); // Add a new item to the dictionary.
dict["key"] // Access the value of the dictionary at the key 'key'.
```

10. Methods

Methods encapsulate logic for reuse and clarity.

```
public returnType MethodName(parameters) { /*...*/ } // A method's signature.

// A Sum method returning int with two int parameters.
public int Sum(int a, int b)
{
    return a + b;
}
```

11. Classes & Objects

Central to object-oriented programming, classes define blueprints for objects.

```
public class MyClass
{
    public string PropertyName { get; set; } // Properties store data for an object.
    public void MethodName() { /*...*/ } // Methods define actions an object can take.
}

MyClass obj = new MyClass(); // Creates a new object of type 'MyClass'.
```

Hint: Auto-properties (as used in the example above), tell the compiler to create a backing field. We do not have to create the backing field and fill it within the Set method or get the value within the Get method. However, we can still implement custom logic when required.

12. Exception Handling

A mechanism to handle runtime errors gracefully, ensuring the program continues or fails gracefully.

```
try
{
    // Code that might throw an exception.
}
catch (SpecificException ex) // Catches specific exceptions, allowing tailored responses.
{
    // Handle this specific error.
}
finally
{
    // Cleanup code. Executes regardless of whether an exception was thrown.
}
```

Hint: It's best practice to catch an exception with a specific type (e.g. SqlException) and have a single general fallback try-catch to catch exceptions of type Exception.

13. Delegates, Events & Lambdas

Essential for event-driven programming and for treating methods as first-class citizens.

```
// A type that represents methods with a specific signature.
public delegate void MyDelegate();

// An event that can be triggered when certain actions occur.
event MyDelegate MyEvent;


Func<int, int, int> add = (a, b) => a + b; // Lambda expression. A concise way to define m
ethods.
```

14. LINQ (Language-Integrated Query)

Introduces native query capabilities into C#, making data manipulation simpler and more readable.

```
using System.Linq; // Required for LINQ queries.
var result = from s in list where s.Contains("test") select s; // Sample query to filter d
```

```
ata.
```

15. Attributes

Add metadata to your code. This metadata can be inspected at runtime.

```
[Obsolete("This method is deprecated.")] // An attribute warning developers about deprecat
ed methods.
public void OldMethod() { /*...*/ }
```

16. Async/Await

Modern mechanism for writing non-blocking (asynchronous) code, especially useful for I/O-bound operations.

```
public async Task<returnType> MethodName()
{
    return await OtherAsyncMethod(); // 'await' pauses method execution until the awaited
task completes.
}
```

17. Miscellaneous

- `enum`: Defines a set of named constants.
- `interface`: A contract that classes can choose to implement.
- `class`: An implementation of an interface (explicit, or implicit inheriting from System.Object). Stored on the heap.
- `record`: A class or struct that provides special syntax and behavior for working with data models.
- `struct`: Lightweight class-like structures, useful for small data structures. Stored on the stack.
- `dynamic`: Allows runtime type resolution (skips compile-time type checking). Uncommon in modern .NET code.
- `is`, `as`: Useful for type checking and conversions.
- `var`: Compiler figures out the type for you, based on the assigned value.

- `nameof`: Returns the name of code elements like variables, types, or members.

18. String Manipulation

Given the ubiquity of string operations in programming, C# provides powerful string manipulation tools.

```
string.Concat(); // Combines multiple strings.
string.Join(); // Joins elements with a separator.
str.Split(); // Splits a string based on delimiters.
str.ToUpper(); // Converts to uppercase.
str.ToLower(); // Converts to lowercase.
str.Trim(); // Removes leading and trailing whitespace.
str.Substring(); // Extracts a portion of the string.
```

19. File I/O

Interactions with the filesystem are common tasks. C# makes reading from and writing to files straightforward.

```
using System.IO; // Necessary for most File I/O operations.

File.ReadAllText(path); // Reads the entire content of a file into a string.
File.WriteAllText(path, content); // Writes a string to a file, creating or overwriting i
t.
File.Exists(path); // Checks if a file exists at the specified path.
```

20. Date & Time

Handling dates and times is integral to many applications. C# offers rich date and time manipulation capabilities.

```
DateTime current = DateTime.Now; // Gets the current date and time.
current.AddDays(1); // Adds a day to the current date.
current.ToShortDateString(); // Converts the date to a short date string format.
```

21. Generics

Generics allow for type-safe data structures without compromising type integrity or performance.

```
public class MyGenericClass<T> // 'T' is a placeholder for any type.
{
    private T item;
    public void UpdateItem(T newItem)
    {
        item = newItem;
    }
}
```

22. Nullables

C# allows value types to be set to `null` using nullable types.

```
int? nullableInt = null; // '?' makes the int nullable.
bool hasValue = nullableInt.HasValue; // Checks if nullable type has a value.
```

23. Attributes & Reflection

Attributes provide metadata about program entities, while reflection allows introspection into types at runtime.

```
[MyCustomAttribute] // Custom attribute applied to a class.
public class MyClass
{
    //...
}

// Using reflection to get the attribute:
Attribute[] attrs = Attribute.GetCustomAttributes(typeof(MyClass));
```

24. Extension Methods

Add new methods to existing types without altering them.

```
public static class StringExtensions
{
    public static bool IsNullOrEmpty(this string str)
    {
        return string.IsNullOrEmpty(str);
```

```
    }
}
```

Hint: Extension Methods are only accessible if you import (using keyword) their namespace. Extension Methods should primarily be used when you cannot alter the type you extend. For example, when extending a framework or third-party library type.

25. Dependency Injection

A software design pattern that facilitates loosely coupled code, improving maintainability and testability.

```
public interface IService
{
    void DoSomething();
}

public class MyService : IService
{
    public void DoSomething()
    {
        // Implementation here
    }
}

public class Consumer
{
    private readonly IService _service;

    public Consumer(IService service)
    {
        _service = service; // Dependency injection through constructor
    }
}

//Register services to the internal dependency injection container in the Program.cs file.
builder.Services.AddScoped<IService, MyService>();
```

26. Partial Classes

Allows the splitting of a single class definition over multiple files or sections in the same file.

```
public partial class MyClass
{
    public void MethodOne() { /*...*/ }
}

public partial class MyClass
{
    public void MethodTwo() { /*...*/ }
}
```

Hint: Many .NET frameworks, such as Blazor or ASP.NET Core make use of the partial keyword.

27. Interoperability

C# allows interoperability with other languages, particularly with legacy Win32 API functions.

```
using System.Runtime.InteropServices; // Required for DllImport.

[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr h, string m, string c, int type);
```

28. Anonymous Types

Allows creation of unnamed types with automatic property definition.

```
var anon = new { Name = "John", Age = 25 }; // The type of 'anon' is inferred at compile t
ime.
```

29. Tuples

Tuples are data structures that have a specific number and sequence of elements.

```
var person = Tuple.Create("John", 25); // Creates a tuple with two items.
```

30. Pattern Matching

Introduced in later versions of C#, pattern matching simplifies certain programming tasks.

```
object obj = "hello";
if (obj is string str)
{
    Console.WriteLine(str); // str is "hello"
}
```

31. Local Functions

Functions can be defined within methods, allowing for encapsulation of logic without polluting class or namespace scopes.

```
public void MyMethod()
{
    void LocalFunction()
    {
        // Implementation
    }

    LocalFunction(); // Calling the local function
}
```

32. Records

Records provide a concise syntax for reference types with value semantics for equality. They're immutable and are great for data structures.

```
public record Person(string Name, int Age);
```

Hint: We use PascalCase for the arguments because they define (public) properties and not fields.

33. `with` Expressions

Used with records to create a non-destructive mutation.

```
var john = new Person("John", 30);
var jane = john with { Name = "Jane" }; // jane is now ("Jane", 30)
```

34. Indexers and Ranges

Allow for more flexible data access, especially useful with strings, arrays, and lists.

```
int[] arr = {0, 1, 2, 3, 4, 5};
var subset = arr[1..^1]; // Grabs elements from index 1 to the second to last.
```

35. `using` Declaration

A more concise way to ensure IDisposable objects are properly disposed.

```
using var reader = new StreamReader("file.txt"); // reader is disposed of at the end of th
e enclosing scope.
```

36. Nullable Reference Types (NRTs)

A feature to help avoid null reference exceptions by being explicit about reference type nullability.

```
#nullable enable
string? mightBeNull; // Reference type that might be null.
```

Hint: You can enable nullable reference types for specific parts of your .NET application using the `#nullable` pragma, as shown in the example. Or you can enable it for the whole .NET project within the .csproj file like this: `<Nullable>enable</Nullable>`.

37. Pattern-Based Using

Allows for more patterns in the `using` statement without implementing IDisposable.

```
public ref struct ResourceWrapper
{
    public void Dispose()
    {
```

```
        // Cleanup
    }
}

using var resource = new ResourceWrapper();
```

38. Property Patterns

Facilitates deconstruction of objects in pattern matching.

```
if (obj is Person { Name: "John", Age: var age })
{
    Console.WriteLine($"John's age is {age}");
}
```

39. Default Interface Implementations

Interfaces can now provide default method implementations.

```
public interface IPerson
{
    void Display();
    void Greet() => Console.WriteLine("Hello!"); // Default implementation.
}
```

40. Dynamic Binding

Makes use of the DLR (Dynamic Language Runtime) for runtime type resolution.

```
dynamic d = 5;
d = "Hello"; // No compile-time type checking.
```

Additional Resources

- The C#/.NET Bootcamp course
- C# Documentation