



C# CODE STRUCTURE

.NET

Don't start jumping into design, creating folders as they come, adding features when you think about it. Sit down for a minute, think clearly about what resources you will need, which technologies or languages you will use, and how to structure all this.

- JULIEN RIO

Procedural Programming

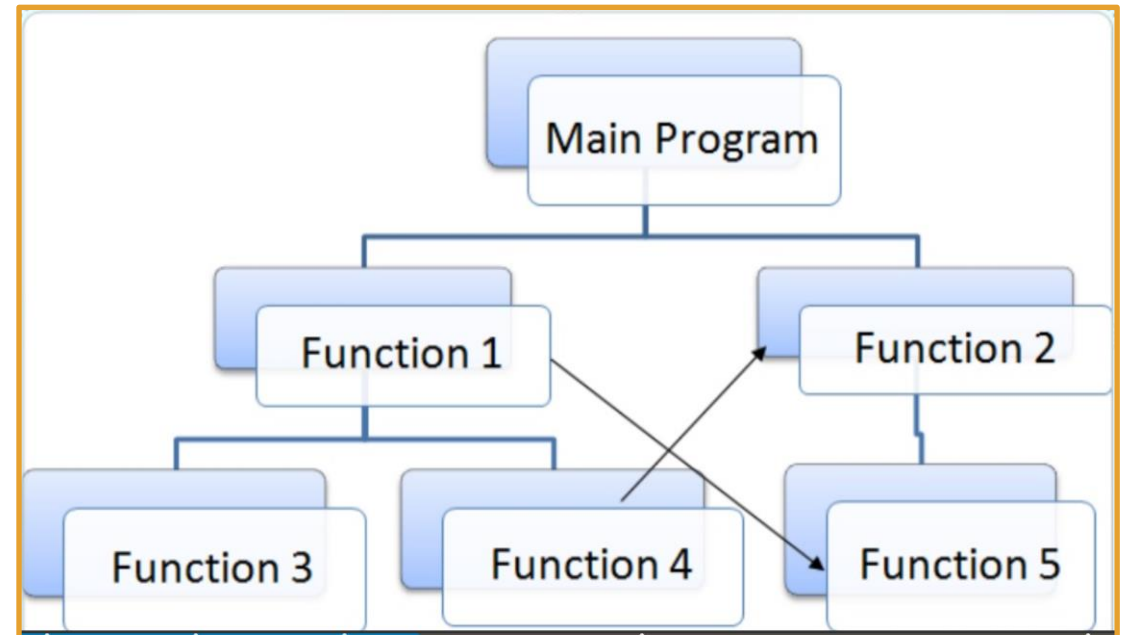
https://en.wikipedia.org/wiki/Procedural_programming

Procedural programming is a programming paradigm derived from 'structured' programming.

It's based on the concept of the procedure call.

Procedures (routines, subroutines, functions, methods) contain a series of computational steps to be carried out.

A method can be called at any point during a program's execution by other methods or by itself (recursion).



C# Program Structure

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-structure>

The key organizational concepts in C# are programs, namespaces, types, members, and assemblies. C# programs consist of one or more source files.

- Programs declare *namespaces*.
- *Namespaces* contain *types* (*classes/interfaces*).
- *Types* contain *members* (*fields, methods, Properties, events*)

When C# programs are compiled, they're packaged into **assemblies** with file extensions **.exe** (Executable) or **.dll** (Dynamic Link Library).

C# Program Structure

```
using System;

namespace Sample
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello, world");
        }
    }
}
```

C# uses braces { } to specify scopes of things

Program on the left contains

1. **namespace** Sample
2. **class** Program
3. **method** Main

Your first .NET program

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/#hello-world>

Hello World

- To create this program, first download and install the [.NET Core SDK](#).
- Then, execute the command `dotnet new console -o hello` to create a new program in a file names `hello` with a build script.
- The program and build script are in the files *Program.cs* and *hello.csproj*, respectively.
- In the command line, type `cd hello`.
- Build and run the application with the run command: `dotnet run`

C# - Hello, World

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/#hello-world>

- *Hello, World* starts with an **using** directive referencing the **System** namespace which contains the **Console** class, I/O, Collections, and others.
- **Namespaces** provide a hierarchical means of organizing C# programs and libraries. **Namespaces** contain **types** and other **namespaces**.
- “**using System**” means *Hello, World* can use **Console.WriteLine** as shorthand for **System.Console.WriteLine**.
- **System** namespace is provided by the **Base Class Library**.
- A static method named **Main** serves as the entry point of any .NET program.

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```


C# Program Structure

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-structure>

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

References to .NET
Namespaces.

Namespace. A logical
grouping of classes.

```
namespace MyFirstProgram  
{
```

Class. An object and scope
for logically related
functionality.

```
    class Program  
    {
```

static void Main Method
The entrypoint to a C#
program.

```
        static void Main(string[] args)  
        {
```

Data type and
variable name.

```
            string howdy = "Hello World!";  
            Console.WriteLine($"{howdy} \tthere \npardn'r!");
```

A value that the data type
holds.

WriteLine() takes an arg
to write to the console.

```
        }  
    }  
}
```

Generic Host

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-5.0>

- ASP.NET Core templates create a .NET Core **Generic Host** called **HostBuilder**.
- **Host** is an object that encapsulates an apps resources (Dependency Injection, Logging, Configuration).
- When a host starts, it calls **IHostedService.StartAsync** on each implementation of **IHostedService** that it finds in the DI container
- The reason for including all app resources in one object is lifetime management. This controls startup and shutdown of each resource.

The **host** is typically (in all but console applications) configured, built, and run by code in **Program.cs**.

Main() calls a **CreateHostBuilder** method to

- *create* a *builder* object and
- *configure* a *builder* object.

Then, it calls **Build()** and **Run()** on the **HostBuilder** object.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```


Generic Host

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-5.0>

The ***GenericHost*** has default builder settings like:

- Sets the content ***root*** to the path returned by ***GetCurrentDirectory***.
- Enables scope validation and dependency validation when the environment is Development.

Adds the following logging providers:

Console
Debug
EventSource
EventLog (only when running on Windows)

Loads app configuration from:

appsettings.json.
appsettings.{Environment}.json.
Secret Manager when the app runs in the Development environment.
Environment variables.
Command-line arguments.

Loads *host* configuration from:

Environment variables prefixed with DOTNET.
Command-line arguments.

C# Program Structure

A First Look

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-structure>

The fully qualified name of this class is **Acme.Collections.Stack**.

Class ***Stack*** contains several *members*:

A field: <ul style="list-style-type: none">•top,	Two methods: <ul style="list-style-type: none">•Push•Pop	•A <u>nested</u> class: <ul style="list-style-type: none">•Entry.
--	---	---

The ***Entry*** class further contains three *members*

Two fields: <ul style="list-style-type: none">•next•data	A parameterized constructor
---	-----------------------------

Will this code compile and run?

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

C# Structure – Classes

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects>

- Classes are the most fundamental of C#'s types.
- A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit.
- A class provides a template for *instances* of the class, known as objects.
- New classes are created using class declarations.
- `Point myPoint = new Point(1,2);`

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

C# Structure – Classes

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects>

A **class** declaration starts with a header that specifies

- the attributes and modifiers of the **class**,
- the name of the **class**,
- the base **class**, and
- the interfaces implemented by the **class**.

The header is followed by the **class** body, which consists of a list of member declarations written between the delimiters { and }.

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

C# Structure - Classes

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-structure>

The fully qualified name of this class is Acme.Collections.Stack.

Class **Stack** contains several *members*:

A field: <ul style="list-style-type: none">•top,	Two methods: <ul style="list-style-type: none">•Push•Pop	A <u>nested</u> class: <ul style="list-style-type: none">•Entry.
--	---	--

The **Entry** class contains three *members*

Two fields: <ul style="list-style-type: none">•Next•data	A parameterized constructor
---	-----------------------------

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

C# Structure – Data Types

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables>

C# supports two kinds of variable *types*:

Value types	Reference types
<p>These are the built-in primitive data types, such as char, int, and float, as well as user-defined types declared with struct. These types directly contain their data.</p>	<p>Classes and other complex data types that are constructed from the primitive types. These types contain a reference to a location in memory where the data is directly held.</p>
<pre>int i = 123;</pre>	<pre>Child child3 = new Child();</pre>

C# Structure – Expressions

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/expressions>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>

- Expressions are constructed from *operands* and *operators*.
- Operators are: **+**, **-**, *****, **/**, **new**
- *Operands are what the operators act upon: literals, fields, local variables, expressions*
- [Precedence](#) of the operators controls the order in which the individual operators are evaluated. (Basically PEMDAS).

```
var a = 2 + 2 * 2;  
Console.WriteLine(a); // output: 6
```

```
var a = (2 + 2) * 2;  
Console.WriteLine(a); // output: 8
```

```
int a = 13 / 5 / 2;  
int b = 13 / (5 / 2);  
Console.WriteLine($"a = {a}, b = {b}");  
// output: a = 1, b = 6
```

C# Structure – Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

The actions of any program are expressed using ***statements***. C# uses various ***statement*** types.

- A **block** consists of a list of statements written between the delimiters `{ }`.
- **Declaration** used to create variables and constants.
- **Expression** statements are used to evaluate actions taken on values.
- **Selection** statements are used for “flow control”.
- **Iteration** statements include the **while**, **do**, **for**, and **foreach** statements. They repeat until a predetermined condition is met.
- **Jump** statements are used to transfer control. This group contains the **break**, **continue**, **goto**, **throw**, **return**, and **yield** statements.

C# Structure – Block and Declaration Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

A **block** permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{}`.

Declaration statements are used to declare local variables and constants.

Local Variable Declaration

```
static void Declarations(string[] args)
{
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

Local Constant Declaration

```
static void ConstantDeclarations(string[] args)
{
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

C# Structure - Expression Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

Expression statements are used to evaluate expressions: method invocations, object allocations using the **new** operator, assignments using **=**, compound assignment operators, increment (**++**) and decrement (**--**) operations and **async/await** expressions.

```
static void Expressions(string[] args)
{
    int i;
    i = 123;           // Expression statement
    Console.WriteLine(i); // Expression statement
    i++;              // Expression statement
    Console.WriteLine(i); // Expression statement
}
```

C# Structure - Selection Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. This group contains the *if* and *switch* statements.

```
static void IfStatement(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine("No arguments");
    }
    else
    {
        Console.WriteLine("One or more arguments");
    }
}
```

```
static void SwitchStatement(string[] args)
{
    int n = args.Length;
    switch (n)
    {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine($"{n} arguments");
            break;
    }
}
```

C# Structure - Iteration Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

Iteration statements are used to execute repeatedly an embedded statement. This group contains the *while*, *do*, *for*, and *foreach* statements.

```
static void WhileStatement(string[] args)
{
    int i = 0;
    while (i < args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

```
static void ForEachStatement(string[] args)
{
    foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```

```
static void ForStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```

```
static void DoStatement(string[] args)
{
    string s;
    do
    {
        s = Console.ReadLine();
        Console.WriteLine(s);
    } while (!string.IsNullOrEmpty(s));
}
```


C# Structure - Jump Statements

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/statements>

Jump statements are used to transfer control. This group contains the **break**, **continue**, **goto**, **throw**, **return**, and **yield** statements.

```
static void BreakStatement(string[] args)
{
    while (true)
    {
        string s = Console.ReadLine();
        if (string.IsNullOrEmpty(s))
            break;
        Console.WriteLine(s);
    }
}
```

```
static void ContinueStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[i].StartsWith("/"))
            continue;
        Console.WriteLine(args[i]);
    }
}
```

```
static int Add(int a, int b)
{
    return a + b;
}
static void ReturnStatement(string[] args)
{
    Console.WriteLine(Add(1, 2));
    return;
}
```

C# Structure – Methods

<https://docs.microsoft.com/en-us/dotnet/csharp/methods>

A method (procedure, function) is a code block that contains a series of statements. A program calls the method and includes any required arguments. Every C# command is executed within a method.

The **Main** method is the entry point for every C# application.

Methods are declared in a **class** or **struct** by specifying a method signature that contains:

- (optional) access level
- (optional) modifiers
- Return value
- Method name
- Method parameters

```
// Anyone can call this.
public void StartEngine() { /* Method statements here */ }

// Only derived classes can call this.
protected void AddGas(int gallons) { /* Method statements here */ }

// Derived classes can override the base class implementation.
public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }
```

C# Structure – Method Invocation

<https://docs.microsoft.com/en-us/dotnet/csharp/methods#method-invocation>

There are two types of methods:

Instance and *Static*

Instance methods

Require an object be instantiated to be called –
myClassInstance.doWork();

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

C# Structure – Method Invocation

<https://docs.microsoft.com/en-us/dotnet/csharp/methods#method-invocation>

There are two types of methods:

Instance and *Static*

Static methods

Can be called without
instantiating an object –
myClassName.doWork();

```
public class Example
{
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}
```