

C#

Detailed Study of C#

Yash Lathiya

|  |  |  |  |
| --- | --- | --- | --- |
| Sr No. | Topic | Date | Page No. |
|  |  |  |  |
| 1 | Introduction to C# | 17/08/2023 | 2 |
| 2 | Structure of C# | 18/08/2023 | 3 |
| 3 | Working with code files, projects & Solutions | 18/08/2023 | 4 |
| 4 | Datatypes & Variables | 19/08/2023 | 5 |
| 5 | Operators & expressions | 19/08/2023 | 6 |
| 6 | Statements | 22/08/2023 | 8 |
| 7 | Understanding Arrays | 21/08/2023 | 10 |
| 8 | Define & Calling Methods | 22/08/2023 | 12 |
| 9 | OOP Concepts | 24/08/2023 | 12 |
| 10 | Scope & Accessibility Modifier | 27/08/2023 | 13 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

Introduction to C#

* Object-Oriented Programming language
* Type-safe programming language
* Similar to C, C++, Java, Javascript
* Automatic garbage collection
* Exception handling
* Nullable types are assigned to variable which are not assigned values
* Support of asynchronous operations.
* Supports generic methods & types.

Execution of C#

* C# programs run on .NET.
* CLR (Common Language Runtime) is a virtual library which is implementation of CLI (Common Language Infrastructure)..
* Source code written in C# is compiled to IL (Intermediate Language) that conforms CLI specifications.
* IL code and other resources (bitmaps, strings,..) are stored in assembly with .dll extension.
* When C# is executed, assembly is loaded in CLR & CLR performs compilation (JIT – Just in Time) to convert IL code to native machine instructions.
* CLR also provides automatic garbage collection, exception handling, resource management, etc..
* IL code (Compiled code of C#) can interact with .NET, C++ and other languages for which CTS (Common Type Specifications) is allowed.

Structure of C#

// A skeleton of a C# program

using System; //System is namespace & Console is class of that namespace

// C# program starts here:

Console.WriteLine("Hello world!");

namespace YourNamespace //It can contain class, struct, interface, delegate, enum or nested namespace

{

class YourClass

{

}

struct YourStruct // encapsulate data and related functionality

{

}

interface IYourInterface //Can’t be instanstiated directly, It’s members are implemented by class or structs implements interface

{

}

delegate int YourDelegate();

enum YourEnum // constants which has integral numeric type

{

}

namespace YourNestedNamespace

{

struct YourStruct

{

}

}

}

Working with code files, projects & solutions



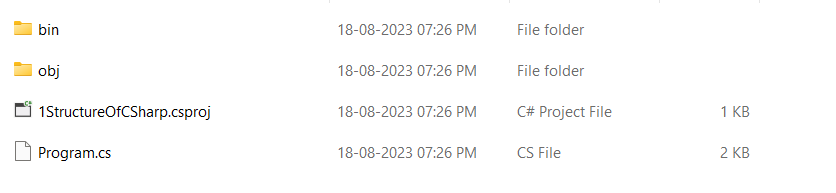
(Main Folder)

(Contains Solution Folder)



(Configuration file..& contains info about compilation & etc.)

(bin folder contains binary [machine-readable] data of the project)



Project File C# file

Datatypes & Variables

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| bool | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter, surrounded by single quotes |
| string | 2 bytes per character | Stores a sequence of characters, surrounded by double quotes |

Type Casting

* **Implicit Casting** (automatically) - converting a smaller type to
* a larger type size  
  char -> int -> long -> float -> double
* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char

Operators & Expressions

Operators

|  |  |
| --- | --- |
| **Operators** | **Category or name** |
| [x.y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#member-access-expression-), [f(x)](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#invocation-expression-), [a[i]](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#indexer-operator-), [x?.y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators" \l "null-conditional-operators--and-), [x?[y]](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and-), [x++](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#increment-operator-), [x--](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#decrement-operator---), [new](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/new-operator), [typeof](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast" \l "typeof-operator), [checked](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked), [unchecked](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked), [default](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/default),/ [sizeof](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/sizeof), [stackalloc](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/stackalloc) | Primary |
| [+x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#unary-plus-and-minus-operators), [-x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#unary-plus-and-minus-operators), [!x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-negation-operator-), [~x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#bitwise-complement-operator-), [++x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#increment-operator-), [--x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#decrement-operator---), [^x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#index-from-end-operator-), [(T)x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression), [await](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await), [&x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/pointer-related-operators#address-of-operator-), [\*x](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/pointer-related-operators#pointer-indirection-operator-), [true and false](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/true-false-operators) | Unary |
| [x..y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#range-operator-) | Range |
| [switch](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/switch-expression), [with](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/with-expression) | switch and with expressions |
| [x \* y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#multiplication-operator-), [x / y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#division-operator-), [x % y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#remainder-operator-) | Multiplicative |
| [x + y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#addition-operator-), [x – y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#subtraction-operator--) | Additive |
| [x << y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#left-shift-operator-), [x >> y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#right-shift-operator-), [x >>> y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#unsigned-right-shift-operator-) | Shift |
| [x < y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/comparison-operators#less-than-operator-), [x > y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/comparison-operators#greater-than-operator-), [x <= y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/comparison-operators#less-than-or-equal-operator-), [x >= y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/comparison-operators#greater-than-or-equal-operator-), [is](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#is-operator), [as](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#as-operator) | Relational and type-testing |
| [x == y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/equality-operators#equality-operator-), [x != y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/equality-operators#inequality-operator-) | Equality |
| x & y | [Boolean logical AND](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-and-operator-) or [bitwise logical AND](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#logical-and-operator-) |
| x ^ y | [Boolean logical XOR](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-exclusive-or-operator-) or [bitwise logical XOR](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#logical-exclusive-or-operator-) |
| x | y | [Boolean logical OR](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-or-operator-) or [bitwise logical OR](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#logical-or-operator-) |
| [x && y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-and-operator-) | Conditional AND |
| [x || y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-or-operator-) | Conditional OR |
| [x ?? y](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator) | Null-coalescing operator |
| [c ? t : f](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator) | Conditional operator |

Expression

* Interpolated String Expressions

string firstName = "Yash";

string lastName = "Lathiya";

var age = 21;

Console.WriteLine($"First Name : {firstName} , Last Name : {lastName}, age : {age}");

* Lambda Expressions

Numbers is array of int.

var maximumCube = numbers.Max( x => x\*x\*x );

Console.WriteLine(maximumCube);

* Query Expressions

int[] values = { 1, 2, 3, 4, 5, 6 };

IEnumerable<int> query = from value in values where value > 3 orderby value select value;

Console.WriteLine(string.Join(" ", query));

Statements

|  |  |
| --- | --- |
| **Category** | **C# keywords / notes** |
| [Declaration statements](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#declaration-statements) | A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required. |
| [Expression statements](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#expression-statements) | Expression statements that calculate a value must store the value in a variable. |
| Selection statements | Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics:   * [if](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements#the-if-statement) * [switch](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement) |
| Iteration statements | Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics:   * [do](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-do-statement) * [for](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-for-statement) * [foreach](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement) * [while](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-while-statement) |
| Jump statements | Jump statements transfer control to another section of code. For more information, see the following topics:   * [break](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-break-statement) * [continue](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-continue-statement) * [goto](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-goto-statement) * [return](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-return-statement) |
| Exception-handling statements | Exception-handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics:   * [throw](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-throw-statement) * [try-catch](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-catch-statement) * [try-finally](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-finally-statement) * [try-catch-finally](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-catch-finally-statement) |
| [checked and unchecked](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked) | The checked and unchecked statements enable you to specify whether integral-type numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. |
| The yield return statement | An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the [yield return](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield) statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.  For more information, see [Iterators](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators). |
| The fixed statement | The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see [fixed](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/fixed). |
| The [empty statement](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#the-empty-statement) | The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed. |

Understanding Arrays

* Array Declaration

int[] array1 = { 1, 2, 3 };

int[] array2 = new int[] { 1, 2, 3 };

int[] array3 = new int[3];

int[,] multiDimensionalArray1 = { { 1, 2, 3 }, { 4, 5, 6 } };

int[,] multiDimensionalArray2 = new int[2, 3];

string[][] jaggedArray = new string[2][];

jaggedArray[0] = new string[3] { "a", "b", "c" };

jaggedArray[1] = new string[4];

jaggedArray[1][0] = "a";

jaggedArray[1][1] = "b";

jaggedArray[1][2] = "c";

jaggedArray[1][3] = "d";

int[][] anotherJaggedArray =

{

new int[] { 1, 2, 3 },

new int[] { 4, 5,},

new int[] { 6 }

};

7

* Array Methods

//Length of Array

jaggedArray.Length

Console.WriteLine("multiDimensionalArray2.GetLength(0) : "+multiDimensionalArray2.GetLength(0));

Console.WriteLine("multiDimensionalArray2.GetLength(1) : "+multiDimensionalArray2.GetLength(1));

//Rank of array

Console.WriteLine("multiDimensionalArray1 Rank : " + multiDimensionalArray1.Rank);

//Reverse array

Array.Reverse(array1);

* Array as object

var objects = new[]

{

new

{

firstName = "Yash",

lastName = "Lathiya"

},

new

{

firstName = "Sachin",

lastName = "Tendulkar"

}

};

Define & Calling Methods

* Define & Calling methods consists similar implementation as Java.

OOP Concepts

* Class contains constructor, variables & method.
* Any object can be created by class reference.
* Object defines state (variables & values) and behaviour (methods & logic).

Encapsulation

* Process of wrapping code and data together into single unit.
* For protection of data
* Implemented by class modifiers (public, private, protected, internal)

Inheritance

* Subclass inherits all properties of baseClass.
* Single Inheritance
* Multi-Level Inheritance
* Hierarchical Inheritance
* Multiple Inheritance (Directly not supported)
* Hybrid Inheritance

Polymorphism

* One interface – multiple implementation
* Method Overloading
* Method Overriding

Abstraction

* Process of hiding the implementation details and showing only functionalities to the user.
* Abstract keyword is used.
* (0 to 100 % abstraction can be achieved in abstract class)
* Can’t create instance of abstract class.

Interface

* (100 % interface)
* Can implement multiple inheritance.
* Only contains method declaration without method body.

Scope & Accessibility Modifiers

Scope

* Other programming languages contains global scope and local scope.
* But in object oriented programming like C#, we should not categorize as global & local scope..
* It should be defined by class & defined by methods.

Accessibility Modifiers

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Caller's location** | **public** | **protected internal** | **protected** | **internal** | **private protected** | **private** |
| Within the class | ✔️️ | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ |
| Derived class (same assembly) | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ | ❌ |
| Non-derived class (same assembly) | ✔️ | ✔️ | ❌ | ✔️ | ❌ | ❌ |
| Derived class (different assembly) | ✔️ | ✔️ | ✔️ | ❌ | ❌ | ❌ |
| Non-derived class (different assembly) | ✔️ | ❌ | ❌ | ❌ | ❌ | ❌ |