What is C#? And What is the latest version of C#?

C# is a computer programming language. Microsoft developed C# in 2000 to provide a modern general-purpose programming language that can be used to develop all kinds of software targeting various platforms, including Windows, Web, and Mobile, using just one programming language. Today, C# is one of the most popular programming languages in the world. Millions of software developers use C# to build all kinds of software.

C# is the primary language for building Microsoft .NET software applications. Developers can build almost every kind of software using C#, including Windows UI apps, console apps, backend services, cloud APIs, Web services, controls and libraries, serverless applications, Web applications, native iOS and Android apps, AI and machine learning software, and blockchain applications.

C# provides rapid application development with the help of Visual Studio IDE. C# is a modern, object-oriented, simple, versatile, and performance-oriented programming language. C# is developed based on the best features and use cases of several programming languages, including C++, Java, Pascal, and SmallTalk.

C# syntaxes are like C++. .NET, and the C# library is similar to Java. C# supports modern object-oriented programming language features, including Abstraction, Encapsulation, Polymorphism, and Inheritance. C# is a strongly typed language. Most of the types in .NET are inherited from the Object class.

C# supports concepts of classes and objects. Classes have members such as fields, properties, events, and methods. Here is a detailed article on C# and OOP.

C# is versatile and modern and supports modern programming needs. Since its inception, C# language has gone through various upgrades. The latest version of C# is C# 11

## What is an object in C#?

C# language is an object-oriented programming language. Classes are the foundation of C#. A class is a template that defines a data structure and how data will be stored, managed, and transferred. A class has fields, properties, methods, and other members.

While classes are concepts, objects are real. Objects are created using class instances. A class defines the type of an object. Objects store real values in computer memory.

Any real-world entity with certain characteristics or that can perform some work is called an Object. This object is also called an instance, i.e., a copy of an entity in a programming language. Objects are instances of classes.

For example, we need to create a program that deals with cars. We need to create entities for cars. Let’s call it a class, Car. A car has four properties, i.e., model, type, color, and size.

To represent a car in programming, we can create a class Car with four properties, Model, Type, Color, and Size. These are called members of a class. A class has several types of members, constructors, fields, properties, methods, delegates, and events. A class member can be private, protected, or public. In addition, since these properties may be accessed outside the class, these can be public.

An object is an instance of a class. A class can have as many instances as needed. For example, Honda Civic is an instance of a Car. In real programming, Honda Civic is an object. Therefore, Honda Civic is an instance of the class Car. The Model, Type, Color, and Size properties of the Honda Civic are Civic, Honda, Red, and 4, respectively. BMW 330, Toyota Carolla, Ford 350, Honda CR4, Honda Accord, and Honda Pilot are some more examples of objects of Car.

To learn more about real-world examples of objects and instances, please read [Object Oriented Programming with Real World Scenario](https://www.c-sharpcorner.com/UploadFile/cda5ba/object-oriented-programming-with-real-world-scenario/).

# What is Roslyn C#

When we compile a C# program in Visual Studio, it calls the C# compiler also known as Roslyn. Today we will write an application that targets .NET framework 5.0 and directly compile it using Roslyn and then run it from the command prompt.

What is Boxing and Unboxing in C#?

Boxing and Unboxing are both used for type conversions.

Converting from a value type to a reference type is called boxing. Boxing is an implicit conversion. Here is an example of boxing in C#.

// Boxing

int anum = 123;

Object obj = anum;

Console.WriteLine(anum);

Console.WriteLine(obj);

C#

Copy

Converting from a reference type to a value type is called unboxing. Here is an example of unboxing in C#.

// Unboxing

Object obj2 = 123;

int anum2 = (int)obj;

Console.WriteLine(anum2);

Console.WriteLine(obj);

What is Object Type?

 The object data type in C# is the base type for all other types. It is a reference type that can hold values of any other type, including value types and reference types. The object data type is primarily used when you need to work with heterogeneous collections or when the specific type of an object is not known at compile time.

1. **object**: The object data type represents an instance of any type. It allows you to store values of different types in a single variable. Since all types in C# implicitly derive from object, you can assign any value to an object variable.

# Choosing Between object and dynamic in C#: Navigating Type Uncertainty for Flexible Programming

In C#, when dealing with situations where the type is not known at compile time, you have two main options: object and dynamic. Each has its own characteristics and use cases, and the choice depends on the specific requirements of your scenario.

### object:

* **Type Safety:**object is a type that can hold values of any data type, but it lacks compile-time type safety.Casting and type-checking are necessary when retrieving values, which may lead to runtime errors if not handled carefully.
* **Compile-Time Checking:**The compiler does not perform type checking on object at compile time.This can lead to potential errors that might only be discovered at runtime.
* **Examples:**

object myObject = 10; // Boxing of int int myInt = (int)myObject; // Unboxing

### dynamic:

* **Type Safety:**dynamic is a type that provides late binding, allowing you to defer type resolution until runtime.Type checking is done at runtime, offering more flexibility compared to object.
* **Compile-Time Checking:**The compiler performs minimal checking on dynamic, deferring most of it until runtime.This allows for dynamic member access without compiler errors.
* **Examples:**

dynamic myDynamic = 10; var result = myDynamic + 5; // Resolved at runtime

## What is an enum in C#?

An enum is a value type with a set of related named constants, often called an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type that is user-defined.

An enum type can be an integer (float, int, byte, double, etc.). But if you use it beside int, it has to be cast.

An enum is used to create numeric constants in the .NET framework. All the members of the enum are enum type. Therefore, there must be a numeric value for each enum type.

The underlying default type of the enumeration element is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

enum Dow {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

C#

Copy

Some points about enum,

* Enums are enumerated data types in c#.
* Enums are not for the end-user. They are meant for developers.
* Enums are strongly typed constant. They are strongly typed, i.e., an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members is the same.
* Enumerations (enums) make your code much more readable and understandable.
* Enum values are fixed. Enum can be displayed as a string and processed as an integer.
* The default type is int, and the approved types are byte, sbyte, short, ushort, uint, long, and ulong.
* Every enum type automatically derives from System.Enum, and thus, we can use System.Enum methods on enums.
* Enums are value types created on the stack, not the heap.

## What is the difference between “continue” and “break” statements in C#?

Using a break statement, you can 'jump out of a loop,' whereas using a continue statement, you can 'jump over one iteration' and resume your loop execution.

## 

## What is the difference between constant and readonly in C#?

**Example**

We have a Test Class in which we have two variables, one is readonly, and the other is a constant.

class Test {

readonly int read = 10;

const int cons = 10;

public Test() {

read = 100;

cons = 100;

}

public void Check() {

Console.WriteLine("Read only : {0}", read);

Console.WriteLine("const : {0}", cons);

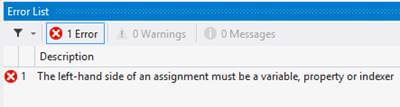
}

}

C#

Copy

Here, I was trying to change the value of both the variables in the constructor, but when I try to change the constant, it gives an error to change their value in the block that I have to call at run time.



Finally, remove that line of code from the class and call this Check() function like in the following code snippet:

class Program {

static void Main(string[] args) {

Test obj = new Test();

obj.Check();

Console.ReadLine();

}

}

class Test {

readonly int read = 10;

const int cons = 10;

public Test() {

read = 100;

}

public void Check() {

Console.WriteLine("Read only : {0}", read);

Console.WriteLine("const : {0}", cons);

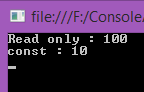
}

}

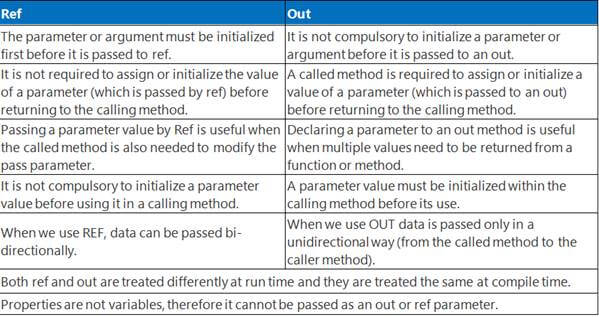
C#

Copy

**Output**



## What is the difference between ref and out keywords?



The ref keyword passes arguments by reference. Therefore, any changes made to this argument in the method will be reflected in that variable when the control returns to the calling method.

The out keyword passes arguments by reference. This is very similar to the ref keyword.

## Can “this” be used within a static method?

We can't use 'this' in a static method because the keyword 'this' returns a reference to the current instance of the class containing it. Static methods (or any static member) do not belong to a particular instance. They exist without creating an instance of the class and are called with the name of a class, not by instance, so we can’t use this keyword in the body of static Methods. However, in the case of Extension Methods, we can use the parameters of the method.

Let’s have a look at the “this” keyword.

The "this" keyword in C# is a special type of reference variable implicitly defined within each constructor and non-static method as the first parameter of the type class in which it is defined.

## What is the use of a delegate in C#?

A Delegate is an abstraction of one or more function pointers (as existed in C++; the explanation about this is out of the scope of this article). The .NET has implemented the concept of function pointers in the form of delegates. With delegates, you can treat a function as data. Delegates allow functions to be passed as parameters, returned from a function as a value, and stored in an array. Delegates have the following characteristics:

* Delegates are derived from the System.MulticastDelegate class.
* They have a signature and a return type. A function that is added to delegates must be compatible with this signature.
* Delegates can point to either static or instance methods.
* Once a delegate object has been created, it may dynamically invoke the methods it points to at runtime.
* Delegates can call methods synchronously and asynchronously.

The delegate contains a couple of useful fields. The first one holds a reference to an object, and the second holds a method pointer. When invoking the delegate, the instance method is called on the contained reference. However, if the object reference is null, then the runtime understands this to mean that the method is a static method. Moreover, invoking a delegate syntactically is the same as calling a regular function. Therefore, delegates are perfect for implementing callbacks.

**Why Do We Need Delegates?**

Historically, the Windows API made frequent use of C-style function pointers to create callback functions. Using a callback, programmers were able to configure one function to report back to another function in the application. So the objective of using a callback is to handle button-clicking, menu-selection, and mouse-moving activities. But the problem with this traditional approach is that the callback functions were not type-safe. In the .NET framework, callbacks are still possible using delegates with a more efficient approach. However, delegates maintain three important pieces of information:

* The parameters of the method.
* The address of the method it calls.
* The return type of the method.

A delegate is a solution for situations where you want to pass methods around to other methods. You are so accustomed to passing data to methods as parameters that the idea of passing methods as an argument instead of data might sound strange. However, there are cases in which you have a method that does something, for instance, invoking some other method. You do not know at compile time what this second method is. That information is available only at runtime. Hence Delegates are the device to overcome such complications.

## What are the Arrays in C#?

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. Therefore, the position of the last item on an array will total the number of items - 1. So if an array has ten items, the previous 10th item is in the 9th position.

In C#, arrays can be declared as fixed-length or dynamic.

A *fixed-length* array can store a predefined number of items.

A *dynamic array* does not have a predefined size. Instead, the size of a *dynamic array* increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined.

Let's take a look at simple declarations of arrays in C#. The following code snippet defines the simplest dynamic array of integer types with no fixed size.

*int[] intArray;*

*As you can see from the above code snippet, the declaration of an array starts with a type of array followed by a square bracket ([]) and the name of the array.*

The following code snippet declares an array that can store five items only, starting from index 0 to 4.

int[] intArray;

intArray = new int[5];

C#

Copy

 The following code snippet declares an array that can store 100 items from index 0 to 99.

int[] intArray;

intArray = new int[100];

**Creating an Indexer**

< modifier > <

return type > this[argument list] {

get {

// your get block code

}

set {

// your set block code

}

}

C#

Copy

**In the above code,**

*<modifier>*

It can be private, public, protected, or internal.

*<return type>*

It can be any valid C# type.

To learn more about indexers in C#, visit [Indexers in C#](https://www.c-sharpcorner.com/uploadfile/puranindia/indexers-in-C-Sharp/).

How to use Nullable<> Types in C#?

A nullable type is a data type that contains the defined data type or the null value.

This nullable type concept is not compatible with "var."

Any data type can be declared nullable type with the help of the operator "?".

For example, the following code declares the int 'i' as a null.

int? i = null;

C#

Copy

As discussed in the previous section, "var" is incompatible with nullable types. So, if you declare the following, you will get an error.

var? i = null;

**Why use access modifiers?**

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement the encapsulation of OOP. In addition, access modifiers allow you to define who does or doesn't have access to certain features.

In C#, there are six different types of Access Modifiers:

|  |  |
| --- | --- |
| Modifier | Description |
| public | There are no restrictions on accessing public members. |
| private | Access is limited to within the class definition. This is the default access modifier type if none is formally specified |
| protected | Access is limited to within the class definition and any class that inherits from the class |
| internal | Access is limited exclusively to classes defined within the current project assembly |
| protected internal | Access is limited to the current assembly and types derived from the containing class. All members in the current project and the derived class can access the variables. |
| private protected | Access is limited to the containing class or types derived from the containing class within the current assembly. |

## What are Value types and Reference types in C#?

In C#, data types can be of two types, value types, and reference types. Value-type variables contain their object (or data) directly. If we copy one value type variable to another, we make a copy of the thing for the second variable. Both of them will independently operate on their values, Value type data types are stored on a stack, and reference data types are stored on a heap.

In C#, basic data types include int, char, bool, and long, which are value types. In addition, classes and collections are reference types.

## How do you use the “using” statement in C#?

There are two ways to use the using keyword in C#. One is as a directive, and the other is as a statement. Let's explain!

1. using Directive

Generally, we use the using keyword to add namespaces in code-behind and class files. Then it makes available all the classes, interfaces, and abstract classes and their methods and properties on the current page. Adding a namespace can be done in the following two ways:

1. Using Statement

This is another way to use the using keyword in C#. It plays a vital role in improving performance in Garbage Collection.

What is a Jagged Array in C#?

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays."

A special type of array is introduced in C#. A Jagged Array is an array of an array in which the length of each array index can differ.

**Example**

int[][] jagArray = new int[5][];

C#

Copy

In the above declaration, the rows are fixed in size. But columns are not specified as they can vary.

Declaring and initializing a jagged array.

int[][] jaggedArray = new int[5][];

jaggedArray[0] = new int[3];

jaggedArray[1] = new int[5];

jaggedArray[2] = new int[2];

jaggedArray[3] = new int[8];

jaggedArray[4] = new int[10];

jaggedArray[0] = new int[] { 3, 5, 7, };

jaggedArray[1] = new int[] { 1, 0, 2, 4, 6 };

jaggedArray[2] = new int[] { 1, 6 };

jaggedArray[3] = new int[] { 1, 0, 2, 4, 6, 45, 67, 78 };

jaggedArray[4] = new int[] { 1, 0, 2, 4, 6, 34, 54, 67, 87, 78 };

### **What are the features of C# language?**

* **Object-Oriented:** C# supports object-oriented programming principles, allowing for encapsulation, inheritance, and polymorphism.
* **Type-Safe:** It enforces strong type checking, enhancing type safety and preventing type-related errors.
* **Interoperability:** C# can interoperate with other languages and libraries, facilitating integration with existing systems.
* **Memory Management:** C# uses automatic garbage collection to manage memory, reducing the risk of memory leaks.
* **Exception Handling:** It provides robust error-handling mechanisms, enabling developers to handle exceptions gracefully.
* **Modern Language Features:** C# includes features like properties, events, and delegates for efficient and expressive coding.

### **How is C# different from the C programming language?**

**C# and C differ in several key aspects:**

|  |  |
| --- | --- |
| **C#** | **C** |
| C# is primarily object-oriented, emphasizing classes and objects. | C is procedural, focusing on functions and structures. |
| C# has automatic memory management (garbage collection), reducing the risk of memory leaks. | C requires manual memory allocation and deallocation. |
| C# is designed for the .NET framework, making it platform-independent. | C code needs to be recompiled for different platforms. |
| C# syntax is more modern and user-friendly, incorporating features like properties, events, and delegates. | These are not present in C. |
| C# offers a rich standard library for various tasks, simplifying development. | C has a smaller standard library, requiring more external libraries for complex tasks |

### **What is Common Language Runtime (CLR)?**

The Common Language Runtime (CLR) is a key component of the .NET framework. It provides a runtime environment for executing managed code, enabling language interoperability and automatic memory management. CLR manages program execution, ensuring code safety and security, handling exceptions, and supporting features like garbage collection. It compiles source code into an intermediate language (IL) and translates it to machine code at runtime, allowing programs written in different languages to run on any platform with CLR support, enhancing software development efficiency and flexibility.

### **What are indexers in C# .NET?**

In [**C#**](https://www.scholarhat.com/tutorial/csharp/csharp-best-practices-for-writing-clean-and-efficient-code) .NET, indexers allow objects to be indexed like arrays. They enable instances of a class to be accessed using array-like syntax. Indexers are defined using special [**methods**](https://www.scholarhat.com/tutorial/csharp/methods-in-csharp) within a class and provide a way to get or set values based on an index, enhancing the flexibility of custom data structures

### **What is Garbage Collection in C#?**

Garbage collection in C# is an automatic memory management process. It identifies and deallocates unused [**objects**](https://www.scholarhat.com/tutorial/csharp/objects-and-classes-in-csharp), preventing memory leaks and optimizing memory usage. Developers don't need to manually release memory, enhancing code reliability and simplicity.

* **Automatic Memory Management:** C# uses a garbage collector to automatically manage memory.
* **Identifies Unused Objects:** The garbage collector identifies and deallocates memory occupied by objects no longer in use.
* **Prevents Memory Leaks:** Ensures unused objects are removed, preventing memory leaks and improving application stability.
* **Efficient Memory Usage:** Optimizes memory usage by reclaiming memory from discarded objects.
* **No Manual Cleanup:** Developers don't need to manually release memory, reducing the risk of bugs and improving developer productivity.

### **What is Datatype in C#?**

In C#, a [**datatype**](https://www.scholarhat.com/tutorial/csharp/datatype-in-csharp) specifies the type of data a variable can hold. C# supports various datatypes, including:

**Value Types:** Represents data with a specific value.

* Integers: int, long, short
* Floating-Point Numbers: float, double
* Characters: char
* Booleans: bool

**Reference Types:** Store references to the memory location of objects.

* Classes: class
* Interfaces: interface
* Arrays: Array
* Delegates: delegate
* Strings: string

### **What is the difference between value types and reference types in C#?**

|  |  |
| --- | --- |
| **Value Type** | **Reference Type** |
| Store data directly in memory. | Store references to the memory location. |
| Derived from System.ValueType. | Derived from System.Object. |
| Examples include int, float, and char. | Examples include classes, interfaces, and [**arrays**](https://www.scholarhat.com/tutorial/csharp/array-in-csharp). |
| Allocated on the stack. | Allocated on the heap. |
| Memory management is efficient. | Memory management is complex. |
| Copying creates independent copies. | Copying creates new references pointing to the same object. |

### **Mention all the advantages of C#.**

* Simplicity: Easy-to-learn syntax enhances readability and reduces complexity.
* Versatility: Supports various programming paradigms, from procedural to object-oriented and component-oriented.
* Interoperability: Seamless integration with other languages and technologies, facilitating code reuse.
* Robustness: Strongly typed language with automatic garbage collection ensures memory management and prevents common errors.
* Scalability: Ideal for developing scalable applications and web services, accommodating growth.
* Security: Provides robust security features like code access security and role-based security

### **What are namespaces in C#?**

In C#, namespaces are used to organize and categorize code elements, such as classes, interfaces, and methods, into logical groups. They prevent naming conflicts by providing a way to uniquely identify types within an application. Namespaces improve code readability, maintainability, and reusability by creating a hierarchical organization of classes and other code elements, making it easier to manage large and complex projects.

Can you name the types of comments in C#?

Single Line//

MultiLine/\*\*/

Can you tell us the extension of a C# language file?

.cs is used to save files

Difference between Indexers and Properties

|  |  |
| --- | --- |
| Indexers | Properties |
| Indexers are created with this keyword. | Properties don't require this keyword. |
| Indexers are identified by signature. | Properties are identified by their names. |
| Indexers are accessed using indexes. | Properties are accessed by their names. |
| Indexer are instance member, so can't be static. | Properties can be static as well as instance members. |
| A get accessor of an indexer has the same formal parameter list as the indexer. | A get accessor of a property has no parameters. |
| A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter. | A set accessor of a property contains the implicit value parameter. |