Contents

[.NET Stack 6](#_Toc94620838)

[The history behind .NET: from the .NET Framework first release to .NET 6 6](#_Toc94620839)

[The .NET Framework era 6](#_Toc94620840)

[The .NET CORE era 6](#_Toc94620841)

[The .NET 5 and .NET 6 era 7](#_Toc94620842)

[What is .NET development platform 10](#_Toc94620843)

[.NET framework 10](#_Toc94620844)

[.NET Core 10](#_Toc94620845)

[Xamarin 11](#_Toc94620846)

[Universal Windows Platform (UWP) 11](#_Toc94620847)

[.NET Standard 11](#_Toc94620848)

[Common language runtime (CLR) 12](#_Toc94620849)

[Supported languages 12](#_Toc94620850)

[C# - Arrays 14](#_Toc94620851)

[Array Declaration and Initialization 14](#_Toc94620852)

[Accessing Array Elements 15](#_Toc94620853)

[Accessing Array using for Loop 15](#_Toc94620854)

[Accessing Array using foreach Loop 16](#_Toc94620855)

[LINQ Methods 16](#_Toc94620856)

[Passing Array as Argument 16](#_Toc94620857)

[C# - Multidimensional Arrays 18](#_Toc94620858)

[C# - Jagged Arrays: An Array of Array 21](#_Toc94620859)

[C# - Collections 24](#_Toc94620860)

[Generic Collections 24](#_Toc94620861)

[Non-generic Collections 24](#_Toc94620862)

[C# - Array List 25](#_Toc94620863)

[Create an ArrayList 25](#_Toc94620864)

[Adding Elements in ArrayList 25](#_Toc94620865)

[Accessing an ArrayList 26](#_Toc94620866)

[Iterate an ArrayList 27](#_Toc94620867)

[Insert Elements in ArrayList 27](#_Toc94620868)

[Remove Elements from ArrayList 28](#_Toc94620869)

[Check Element in ArrayList 29](#_Toc94620870)

[ArrayList Class 29](#_Toc94620871)

[ArrayList Properties 30](#_Toc94620872)

[ArrayList Methods 31](#_Toc94620873)

[C# - List 31](#_Toc94620874)

[List<T> Characteristics 32](#_Toc94620875)

[Creating a List 32](#_Toc94620876)

[Adding an Array in a List 33](#_Toc94620877)

[Accessing a List 33](#_Toc94620878)

[Accessing a List using LINQ 33](#_Toc94620879)

[Insert Elements in List 34](#_Toc94620880)

[Remove Elements from List 34](#_Toc94620881)

[Check Elements in List 35](#_Toc94620882)

[List<T> Class Hierarchy 35](#_Toc94620883)

[List<T> Class Properties and Methods 36](#_Toc94620884)

[C# - SortedList<TKey, TValue> 37](#_Toc94620885)

[SortedList Characteristics 38](#_Toc94620886)

[Creating a SortedList 38](#_Toc94620887)

[Accessing SortedList 40](#_Toc94620888)

[Remove Elements from SortedList 41](#_Toc94620889)

[SortedList Class Hierarchy 42](#_Toc94620890)

[C# - Dictionary<TKey, TValue> 42](#_Toc94620891)

[Dictionary Characteristics 42](#_Toc94620892)

[Creating a Dictionary 43](#_Toc94620893)

[Access Dictionary Elements 43](#_Toc94620894)

[Update Dictionary 44](#_Toc94620895)

[Remove Elements in Dictionary 44](#_Toc94620896)

[Dictionary Class Hierarchy 45](#_Toc94620897)

[C# - Hashtable 46](#_Toc94620898)

[Hashtable Characteristics 46](#_Toc94620899)

[Creating a Hashtable 46](#_Toc94620900)

[Update Hashtable 47](#_Toc94620901)

[Remove Elements in Hashtable 47](#_Toc94620902)

[Hashtable Class Hierarchy 48](#_Toc94620903)

[C# - Stack<T> 50](#_Toc94620904)

[Stack<T> Characteristics 50](#_Toc94620905)

[Creating a Stack 50](#_Toc94620906)

[Stack<T> Properties and Methods: 51](#_Toc94620907)

[Pop() 51](#_Toc94620908)

[Peek() 51](#_Toc94620909)

[Contains() 52](#_Toc94620910)

[C# - Queue<T> 52](#_Toc94620911)

[Queue<T> Characteristics 52](#_Toc94620912)

[Creating a Queue 53](#_Toc94620913)

[Queue<T> Properties and Methods 53](#_Toc94620914)

[Retrieve Elements from a Queue 54](#_Toc94620915)

[Contains() 54](#_Toc94620916)

[C# - HashSet 55](#_Toc94620917)

[How to create a HashSet? 55](#_Toc94620918)

[How to remove elements from the HashSet? 57](#_Toc94620919)

[Set Operations 58](#_Toc94620920)

[C# - Type Conversions 63](#_Toc94620921)

[Implicit Type Casting / Automatic Type Conversion 63](#_Toc94620922)

[Explicit Type Casting 64](#_Toc94620923)

[C# - Serialization 67](#_Toc94620924)

[How serialization works 67](#_Toc94620925)

[Namespaces involved 67](#_Toc94620926)

[Uses for serialization 68](#_Toc94620927)

[JSON serialization 68](#_Toc94620928)

[Binary and XML serialization 68](#_Toc94620929)

[Making an object serializable 68](#_Toc94620930)

[Basic and custom serialization 69](#_Toc94620931)

[Example 1: Binary Serialization 69](#_Toc94620932)

[Explanation 70](#_Toc94620933)

[Example 2: Class / Object Serialization 70](#_Toc94620934)

[Example 3: Array Serialization 72](#_Toc94620935)

[Example 4: Serialize Class / Object 73](#_Toc94620936)

[Code Explanation: 74](#_Toc94620937)

[Serialize and Deserialize JSON 75](#_Toc94620938)

[Namespaces 75](#_Toc94620939)

[Serialize (Write) .NET objects as JSON 75](#_Toc94620940)

[Serialize JSON Serialization 75](#_Toc94620941)

[Serialize JSON to a File 76](#_Toc94620942)

[Serialize JSON to a File Asynchronously 77](#_Toc94620943)

[Serialize JSON using Generics 78](#_Toc94620944)

[Serialization of Class with Collections and User Defined Types 78](#_Toc94620945)

[Deserialize (Read) JSON as .NET Objects 80](#_Toc94620946)

[Deserialize a JSON String 80](#_Toc94620947)

[Deserialize JSON from a File 82](#_Toc94620948)

[Deserialize JSON from a File Asynchronously 82](#_Toc94620949)

[Serialize to Formatted JSON 84](#_Toc94620950)

[Include fields 85](#_Toc94620951)

[HttpClient and HttpContent extension methods 86](#_Toc94620952)

[Serialize and Deserialize XML 88](#_Toc94620953)

[Serialize (Read) Data from an XML File 88](#_Toc94620954)

[Compiling the Code 88](#_Toc94620955)

[Robust Programming 88](#_Toc94620956)

[Deserialize (Write) Data to an XML File 89](#_Toc94620957)

[Compiling the Code 89](#_Toc94620958)

[Robust Programming 89](#_Toc94620959)

[A Better Example 90](#_Toc94620960)

[Serializing XML to C# Object 90](#_Toc94620961)

[Deserializing a C# Object in XML 91](#_Toc94620962)

[Another XML Example 91](#_Toc94620963)

[Verification 92](#_Toc94620964)

[Serializing a Class that Contains a Field Returning a Complex Object 92](#_Toc94620965)

[Serializing an Array of Objects 93](#_Toc94620966)

[Serializing a Class that Implements the ICollection Interface 93](#_Toc94620967)

[Purchase Order Example - Exercise 95](#_Toc94620968)

# .NET Stack

## The history behind .NET: from the .NET Framework first release to .NET 6

Microsoft started working on the .NET framework in the late 90s. The idea was to create a platform based on so-called managed code, code that can be executed under a runtime environment. This was needed to improve the development experience and relieve engineers from handling security operations, active memory management, and other low-level efforts that C/C++ developers had to bother with.

## The .NET Framework era

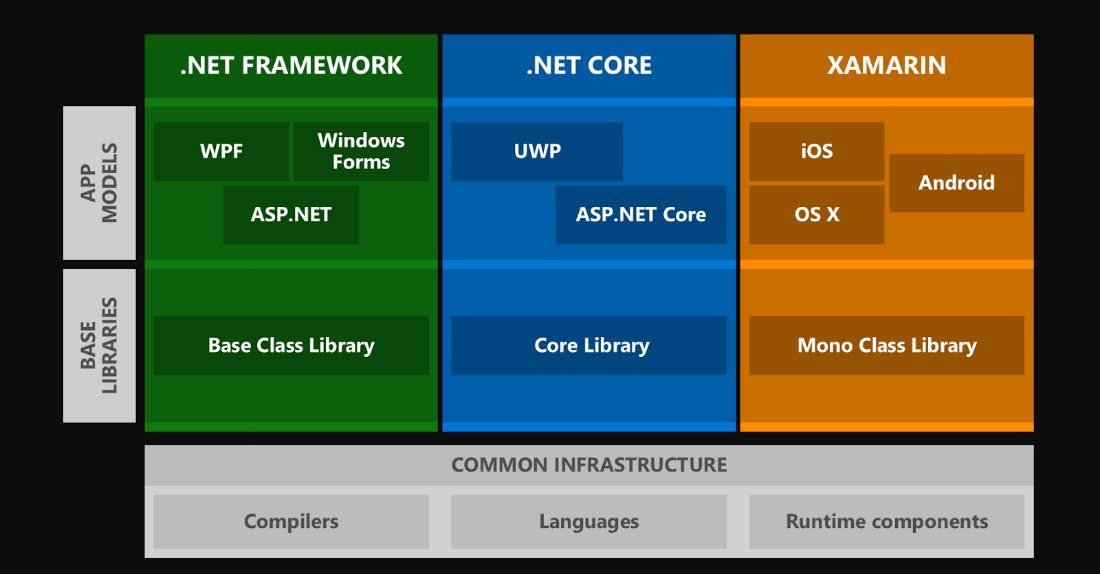
The first release of .NET Framework in 2002 introduced C#, a language for writing managed code that had a design similar to C++. The framework itself was aimed at Windows-based computers and servers. It had WinForms, a GUI library for desktop applications; ASP.NET, a framework for Web; and ADO.NET for data access. All these elements were driven by Common Language Runtime (CLR) to compile and execute managed code.

To unite various functions, .NET offered a framework class library (FCL) that included the base class library (BCL), network library, a numerics library, and others.

Since that time, the framework has undergone multiple iterations spanning runtime updates, new desktop graphical systems (WPF), APIs for service-oriented applications (WCF), and more.

## The .NET CORE era

In 2014, Microsoft announced a dramatic shift in the way .NET exists by presenting .NET Core, a new cross-platform, cloud-friendly, and open-source version of the framework. .NET Core made it to a release in 2016, becoming the main technology to consider for new .NET projects. Gradually, Microsoft started porting existing services to work with Core. Some that didn’t receive official ports, like Windows Communication Foundation (WCF), were substituted by alternatives sourced from the community.

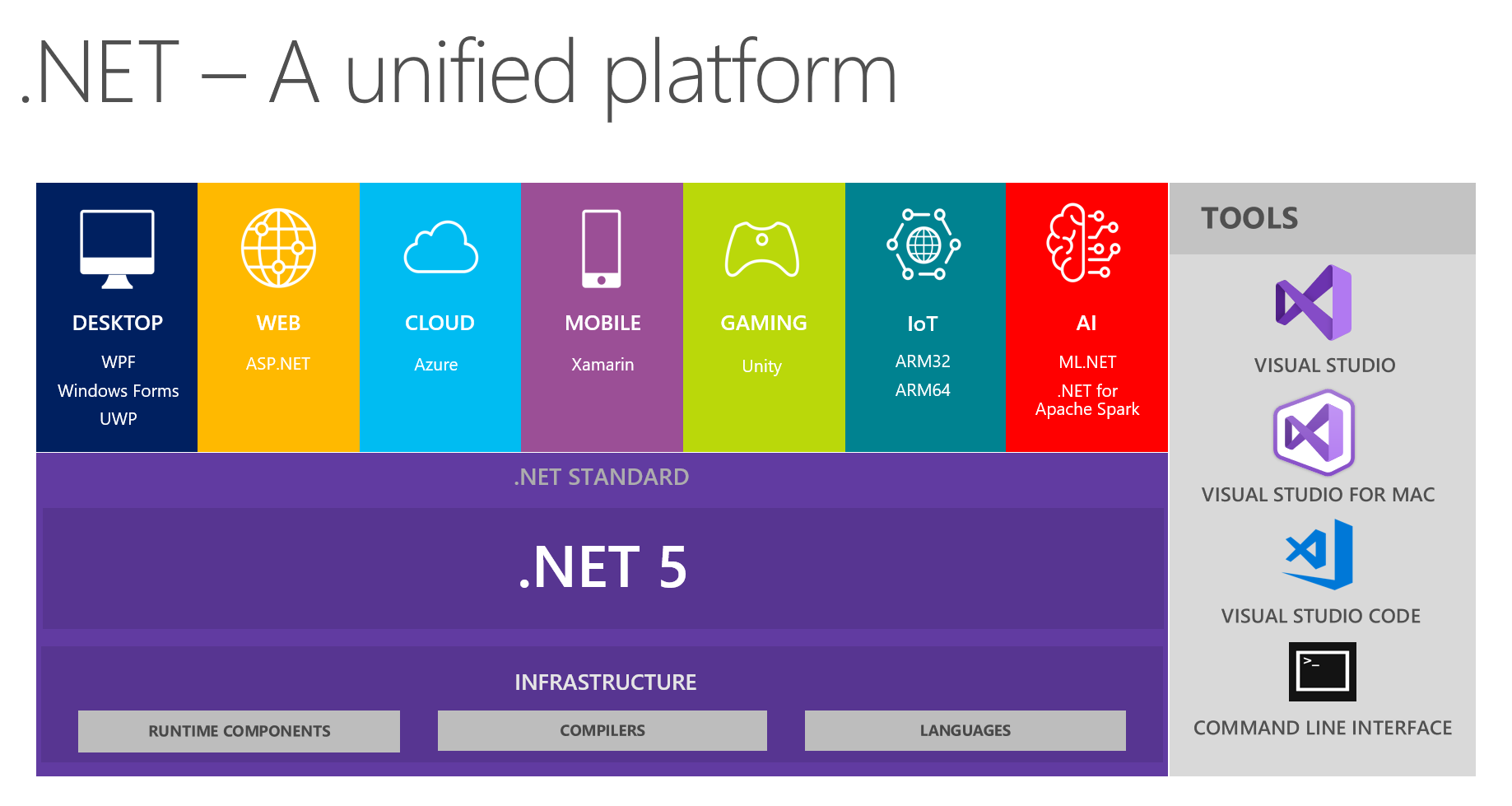


Also, in 2016, Microsoft acquired Xamarin, previously a proprietary technology for cross-platform mobile development, making it open source as well.

Microsoft continued moving towards “transparency between the product team and the community,” and open-sourced Windows Presentation Foundation (WPF), Windows Forms, and WinUI frameworks in December 2018.

## The .NET 5 and .NET 6 era

In May 2019, the company announced the big release that would tie the ecosystem together: All .NET elements were supposed to be bundled in the .NET 5 development platform. While changes were made to the schedule because of COVID-19, the .NET 5 unified development platform was finally introduced in November 2020. The successor to .NET Core 3.1 and .NET Framework 4.8, .NET 5 puts order into the fragmentation of the .NET world and provides a lot of features to build applications on Windows, Linux, macOS, iOS, watchOS, Android, tvOS, or using WebAssembly. The platform comes with new APIs, language features, and runtime capabilities. Also, .NET 5 includes ASP.NET Core, Xamarin, Entity Framework Core, WPF, WinForms, and ML.NET.



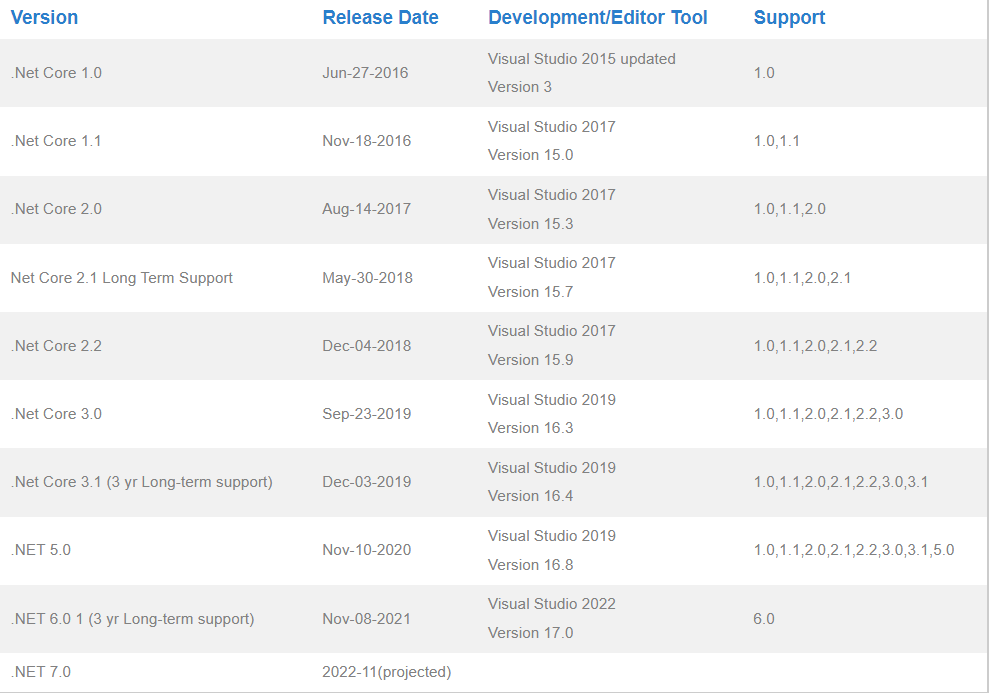
While .NET 5 set the unification foundations, the newest .NET 6 version delivered the final parts of it in November 2021, with Visual Studio 2022 released the same day. This is a unified platform for building projects across cloud, browser, IoT, mobile, and desktop environments, enabling all to use the same .NET libraries, SDK, and runtime.



One of the most prominent .NET 6 features is .NET MAUI (Multi-platform App UI) that acts as the cross-platform framework for developing native desktop and mobile apps with C# and XAML.

Apart from being the final step of the unification, .NET 6 can boast of:

* Better performance with decreased project execution time, latency time, and memory use.
* APIs for HTTP/3, JSON processing, mathematics, and direct memory manipulation.
* Improved security with support for OpenSSL 3 and ChaCha20Poly1305 encryption scheme.
* Stable version and long-term support for 3 years.



Besides all that, it’s the very first release that has native support for Apple Silicon (mac OS Arm64) and improved support for Windows Arm64.

# What is .NET development platform

.NET comes in four flavors: .NET Framework, .NET Core, Xamarin, and Universal Windows Platform (UWP).

These implementations combined are called the .NET development platform. Each of them contains frameworks and libraries to build various applications.

## .NET framework

The .NET Framework released back in 2002 is the first and oldest implementation of the platform. It includes three main application models – WPF, Windows Forms, ASP.NET Forms – and Base Class Library.

Windows Presentation Foundation (WPF) is a UI framework used for creating graphical interfaces primarily for desktop client applications on Windows OS. WPF uses the capabilities of Extensible Application Markup Language (XAML).

Windows Forms is a GUI class library within .NET Framework. Windows Forms are used to develop desktop applications with rich graphics that are easy to update and deploy.

ASP.NET. While the previous two components are designed for desktop engineering ASP.NET is used to develop dynamic websites and web applications. There is the Common Language Runtime (CLR) in its core that gives developers the opportunity to write ASP.NET code using different .NET languages that we discuss below.

Base Class Library (BCL) provides the most common functionality like classes in namespaces and is the core of the Framework Class Library (FCL), a set of reusable interfaces, classes, and value types that are closely integrated with the Common Language Runtime (CLR). The combination of FCL and CLR constitutes the .NET Framework. The base class library also includes ADO.NET, data access technology used by developers to access databases.

As .NET Framework supports only Windows-based devices, there was a need for a cross-platform package.

## .NET Core

.NET Core was released in 2016. It’s a cross-platform re-build of .NET Framework. Unlike the old version, engineers can now use the product on Linux and macOS and create applications that aren’t necessarily tied to the Windows family. The new system aims at conquering the cloud space as some providers like Digital Ocean are Linux-driven. Not only is .NET Core cross-platform but its different versions can also be installed side by side on the same device. .NET Core includes ASP.NET Core and Universal Windows Platform (UWP).

ASP.NET Core is a rebuild of ASP.NET that happened to be a more modular framework than its predecessor. ASP.NET Core allows you to build the mobile backend, web apps, and services. It’s also cross-platform and runs on OS X, Windows, and Linux.

In .NET 6, ASP.NET Core counts many new features including Hot Reload that enables the application of changes to C#, Visual Basic, and CSS source files without having to restart or rebuild the app (the app is running while you are updating the code).

## Xamarin

The third implementation is called Xamarin and is used for mobile applications and Mac products. Originally, Xamarin was designed independently from Microsoft and was a proprietary product. Then Microsoft acquired it in 2016 making it a fully open-source branch of the .NET platform. Xamarin uses the Mono runtime and a version of the .NET Framework adjusted to work with APIs for iOS, Android, and Xamarin.Mac.

All runtimes use a common infrastructure that makes the entire ecosystem work. It provides runtime components, languages, and compilers.

## Universal Windows Platform (UWP)

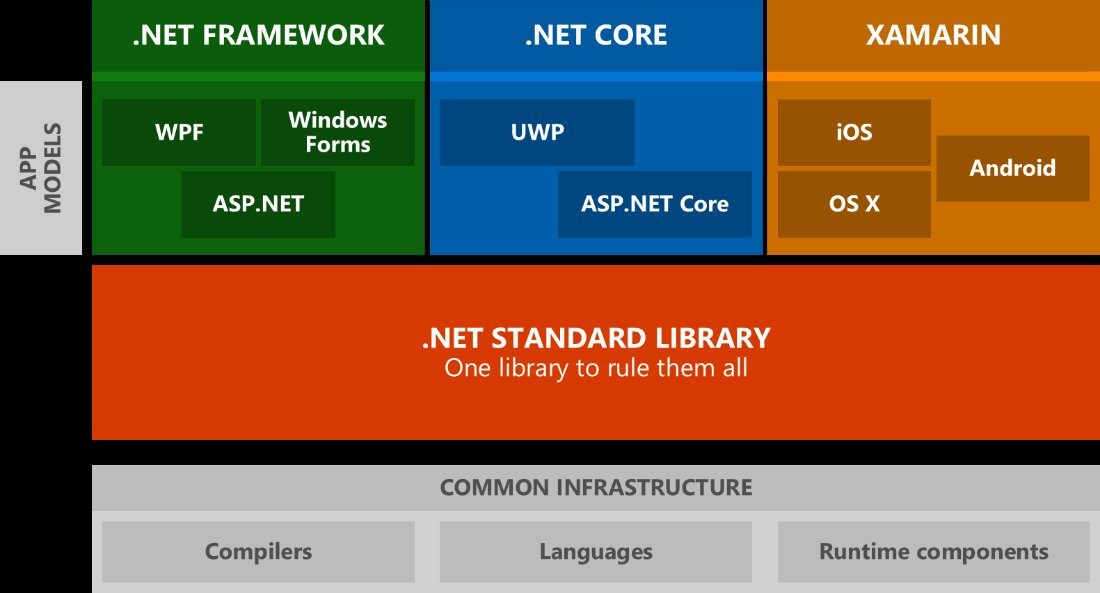
UWP provides a common type system, APIs, and application model for all devices running on Windows 10. So, UWP enables the development of universal apps for PC, tablet, Xbox, Surface Hub, HoloLens, or Internet of Things (IoT) devices.

UWP app developers get access to the Microsoft store that charges only 15 percent for non-gaming subscription-based apps, unlike Google Play Store and App Store. Other services include an execution environment (AppContainer) and Extension SDKs to call specialized APIs for different devices.

Unfortunately, .NET 5 and .NET 6 will not be coming to UWP project types and there will not be an update on UWP in this regard.

## .NET Standard

In 2016 Microsoft also introduced .NET Standard, a library that combines APIs from .NET Framework, .NET Core, and Xamarin allowing engineers to use a single base-class library rather than mastering three different ones related to each .NET implementation. This step unified the ecosystem and brought a higher consistency to reusing components across different platforms.



.NET development platform is best served with Visual Studio IDE used for building, debugging, and publishing applications across all platforms and devices.

## Common language runtime (CLR)

Common Language Runtime (CLR) is the heart of .NET, an application virtual machine that manages memory, implements code access security, verifies code safety, and provides execution of threads and code. CLR is what makes the .NET code a managed one.

As we mentioned, the idea of CLR is to make the developer’s life easier. Besides, it allows engineers to design systems with multiple languages, as CLR enables them to communicate and integrate their behaviors. The runtime checks the needed versions of applied services to ensure that all dependencies are intact and the code works as intended.

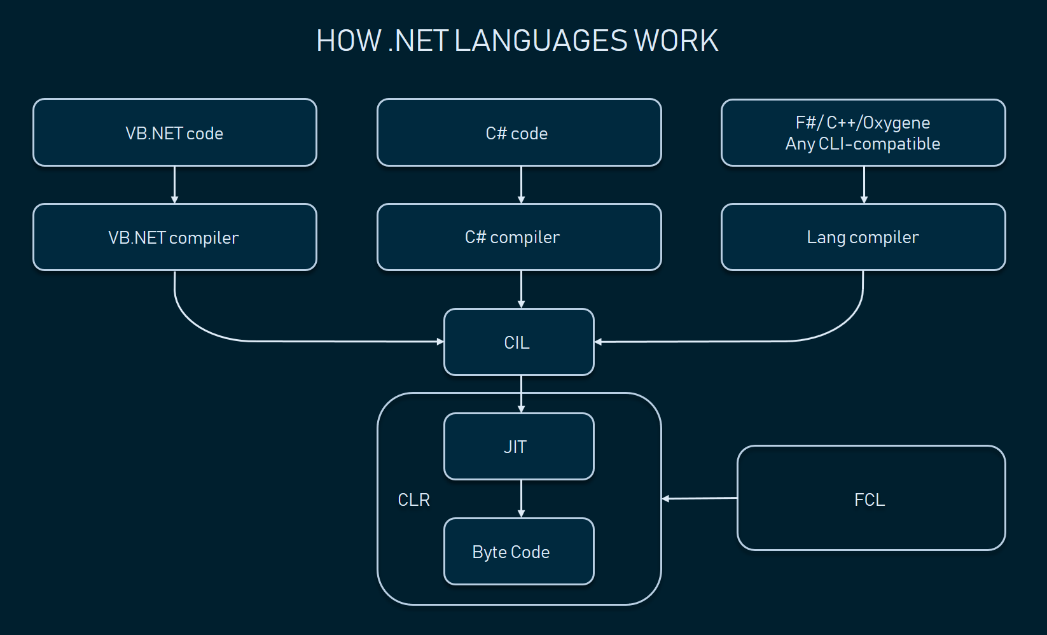
In .NET Core, an open-source CoreCLR is used. While nearly identical to CLR in .NET Framework, CoreCLR is adjusted to the .NET Core cross-platform makeup.

## Supported languages

The languages that you can use with .NET can be ultimately divided into two major groups: 1) the main officially supported languages by Microsoft, and 2) the rest of the languages that are CLI-compliant.

Main languages. Most of the .NET development happens with C#, F#, and Visual Basic. All three saw great improvements within the .NET 6 release. C# 10 and F# 6 were introduced, both aiming at simplifying code and making it more performant.

The rest of CLI-compliant languages. CLI means common language infrastructure. It’s a technical standard for high-level languages that can be compiled into a common intermediate language (CIL) and further compiled into a byte-code. Besides those three mentioned above, there are about 25 active CLI-compliant languages, including C++/CLI, IronPython, Oxygene, Phalanger, and more. There’s also a number of languages that are no longer used, like IronRuby.



# C# - Arrays

A variable is used to store a literal value, whereas an array is used to store multiple literal values.

An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.

In C#, an array can be of three types: single-dimensional, multidimensional, and jagged array. Here you will learn about the single-dimensional array.

The following figure illustrates an array representation.

[Table

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/array.png)

## Array Declaration and Initialization

An array can be declared using by specifying the type of its elements with square brackets.

Example: Array Declaration

int[] evenNums; // integer array

string[] cities; // string array

The following declares and adds values into an array in a single statement.

Example: Array Declaration & Initialization

int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

string[] cities = new string[3]{ "Mumbai", "London", "New York" };

Above, evenNums array can store up to five integers. The number 5 in the square brackets new int[5] specifies the size of an array. In the same way, the size of cities array is three. Array elements are added in a comma-separated list inside curly braces { }.

Arrays type variables can be declared using [var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) without square brackets.

Example: Array Declaration using var

var evenNums = new int[]{ 2, 4, 6, 8, 10};

var cities = new string[]{ "Mumbai", "London", "New York" };

If you are adding array elements at the time of declaration, then size is optional. The compiler will infer its size based on the number of elements inside curly braces, as shown below.

Example: Short Syntax of Array Declaration

int[] evenNums = { 2, 4, 6, 8, 10};

string[] cities = { "Mumbai", "London", "New York" }

The following example demonstrate invalid array declarations.

Example: Invalid Array Creation

//must specify the size

int[] evenNums = new int[];

//number of elements must be equal to the specified size

int[] evenNums = new int[5] { 2, 4 };

//cannot use var with array initializer

var evenNums = { 2, 4, 6, 8, 10};

It is not necessary to declare and initialize an array in a single statement. You can first declare an array then initialize it later on using the new operator.

Example: Late Initialization

int[] evenNums;

evenNums = new int[5];

// or

evenNums = new int[]{ 2, 4, 6, 8, 10 };

## Accessing Array Elements

Array elements can be accessed using an index. An index is a number associated with each array element, starting with index 0 and ending with array size - 1.

The following example add/update and retrieve array elements using indexes.

Example: Access Array Elements using Indexes

int[] evenNums = new int[5];

evenNums[0] = 2;

evenNums[1] = 4;

//evenNums[6] = 12; //Throws run-time exception IndexOutOfRange

Console.WriteLine(evenNums[0]); //prints 2

Console.WriteLine(evenNums[1]); //prints 4

Note that trying to add more elements than its specified size will result in IndexOutOfRangeException.

## Accessing Array using for Loop

Use the for loop to access array elements. Use the length property of an array in conditional expression of the for loop.

Example: Accessing Array Elements using for Loop

int[] evenNums = { 2, 4, 6, 8, 10 };

for(int i = 0; i < evenNums.Length; i++)

Console.WriteLine(evenNums[i]);

for(int i = 0; i < evenNums.Length; i++)

evenNums[i] = evenNums[i] + 10; // update the value of each element by 10

## Accessing Array using foreach Loop

Use foreach loop to read values of an array elements without using index.

Example: Accessing Array using foreach Loop

int[] evenNums = { 2, 4, 6, 8, 10};

string[] cities = { "Mumbai", "London", "New York" };

foreach(var item in evenNums)

Console.WriteLine(item);

foreach(var city in cities)

Console.WriteLine(city);

## LINQ Methods

All the arrays in C# are derived from an abstract base class System.Array.

The Array class implements the IEnumerable interface, so you can LINQ extension methods such as Max(), Min(), Sum(), reverse(), etc.

Example: LINQ Methods

int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

nums.Max(); // returns 16

nums.Min(); // returns 6

nums.Sum(); // returns 55

nums.Average(); // returns 55

The System.ArraySystem.Array class also includes methods for creating, manipulating, searching, and sorting arrays.

Example: Array Methods

int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

Array.Sort(nums); // sorts array

Array.Reverse(nums); // sorts array in descending order

Array.ForEach(nums, n => Console.WriteLine(n)); // iterates array

Array.BinarySearch(nums, 5);// binary search

## Passing Array as Argument

An array can be passed as an argument to a method parameter. Arrays are reference types, so the method can change the value of the array elements.

Example: Passing Array as Argument

public static void Main(){

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

UpdateArray(nums);

foreach(var item in nums)

Console.WriteLine(item);

}

public static void UpdateArray(int[] arr)

{

for(int i = 0; i < arr.Length; i++)

arr[i] = arr[i] + 10;

}

i = 0; i < arr.Length; i++)

arr[i] = arr[i] + 10;

}

# C# - Multidimensional Arrays

C# supports multidimensional arrays up to 32 dimensions. The multidimensional array can be declared by adding commas in the square brackets. For example, [,] declares two-dimensional array, [, ,] declares three-dimensional array, [, , ,] declares four-dimensional array, and so on. So, in a multidimensional array, no of commas = No of Dimensions - 1.

The following declares multidimensional arrays.

Example: Multidimensional Arrays

int[,] arr2d; // two-dimensional array

int[, ,] arr3d; // three-dimensional array

int[, , ,] arr4d ; // four-dimensional array

int[, , , ,] arr5d; // five-dimensional array

Let's understand the two-dimensional array. The following initializes the two-dimensional array.

Example: two-dimensional Array

int[,] arr2d = new int[3,2]{

{1, 2},

{3, 4},

{5, 6}

};

// or

int[,] arr2d = {

{1, 2},

{3, 4},

{5, 6}

};

[,] arr2d = {

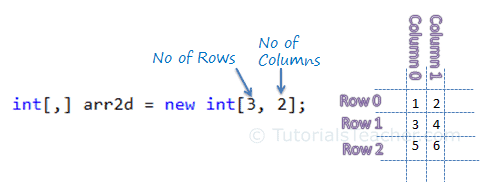
{1, 2},

{3, 4},

{5, 6}

};

In the above example of a two-dimensional array, [3, 2] defines the no of rows and columns. The first rank denotes the no of rows, and the second rank defines no of columns. The following figure illustrates the two-dimensional array divided into rows and columns.

[](https://www.tutorialsteacher.com/Content/images/csharp/twodimensional-array.PNG)

The following access values of the two-dimensional array.

Example: Access two-dimensional Array

int[,] arr2d = new int[3,2]{

{1, 2},

{3, 4},

{5, 6}

};

arr2d[0, 0]; //returns 1

arr2d[0, 1]; //returns 2

arr2d[1, 0]; //returns 3

arr2d[1, 1]; //returns 4

arr2d[2, 0]; //returns 5

arr2d[2, 1]; //returns 6

//arr2d[3, 0]; //throws run-time error as there is no 4th row

In the above example, the value of a two-dimensional array can be accessed by index no of row and column as [row index, column index]. So, [0, 0] returns the value of the first row and first column and [1, 1] returns the value from the second row and second column.

Now, let's understand the three-dimensional array. The following declares and initializes three-dimensional arrays.

Example: Three-dimensional Array

int[, ,] arr3d1 = new int[1, 2, 2]{

{ { 1, 2}, { 3, 4} }

};

int[, ,] arr3d2 = new int[2, 2, 2]{

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

};

int[, ,] arr3d3 = new int[2, 2, 3]{

{ { 1, 2, 3}, {4, 5, 6} },

{ { 7, 8, 9}, {10, 11, 12} }

};

arr3d2[0, 0, 0]; // returns 1

arr3d2[0, 0, 1]; // returns 2

arr3d2[0, 1, 0]; // returns 3

arr3d2[0, 1, 1]; // returns 4

arr3d2[1, 0, 0]; // returns 5

arr3d2[1, 0, 1]; // returns 6

arr3d2[1, 1, 0]; // returns 7

arr3d2[1, 1, 1]; // returns 8

As you can see in the above example, [1, 2, 2] of arr3d1 specifies that it will contain one row of two-dimensional array [2, 2]. arr3d2 specifies dimensions [2, 2, 2], which indicates that it includes two rows of two-dimensional array of [2, 2]. Thus, the first rank indicates the number of rows of inner two-dimensional arrays.

Now, consider the following four-dimensional array.

Example: Four-dimensional Array

int[,,,] arr4d1 = new int[1, 1, 2, 2]{

{

{ { 1, 2}, { 3, 4} }

}

};

arr4d1[0, 0, 0, 0]; // returns 1

arr4d1[0, 0, 0, 1]; // returns 2

arr4d1[0, 0, 1, 0]; // returns 3

arr4d1[0, 0, 1, 1]; // returns 4

int[,,,] arr4d2 = new int[1, 2, 2, 2]{

{

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

}

};

arr4d2[0, 0, 0, 0]; // returns 1

arr4d2[0, 0, 0, 1]; // returns 2

arr4d2[0, 0, 1, 0]; // returns 3

arr4d2[0, 0, 1, 1]; // returns 4

arr4d2[0, 1, 0, 0]; // returns 5

arr4d2[0, 1, 0, 1]; // returns 6

arr4d2[0, 1, 1, 0]; // returns 7

arr4d2[0, 1, 1, 1]; // returns 8

In the above example, the four-dimensional array arr4d1 specifies [1, 1, 2, 2], which indicates that it includes one row of the three-dimensional array.

In the same way, you can declare and initialize five-dimensional, six-dimensional array, and up to 32-dimensional arrays in C#.

# C# - Jagged Arrays: An Array of Array

A jagged array is an array of array. Jagged arrays store arrays instead of literal values.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

The following example declares jagged arrays.

Example: Jagged Arrays

int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays

int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays

II

n the above example, jArray1 can store up to two single-dimensional arrays. jArray2 can store up to three two-dimensional, arrays [,] specifies the two-dimensional array.

Example: Jagged Array

int[][] jArray = new int[2][];

jArray[0] = new int[3]{1, 2, 3};

jArray[1] = new int[4]{4, 5, 6, 7 };

You can also initialize a jagged array upon declaration like the below.

Example: Jagged Array

int[][] jArray = new int[2][]{

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

new int[4]{4, 5, 6, 7}

};

jArray[0][0]; //returns 1

jArray[0][1]; //returns 2

jArray[0][2]; //returns 3

jArray[1][0]; //returns 4

jArray[1][1]; //returns 5

jArray[1][2]; //returns 6

jArray[1][3]; //returns 7

You can access a jagged array using two for loops, as shown below.

Example: Jagged Array

int[][] jArray = new int[2][]{

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

new int[4]{4, 5, 6, 7}

};

for(int i=0; i<jArray.Length; i++)

{

for(int j=0; j < (jArray[i]).Length; j++)

Console.WriteLine(jArray[i][j]);

}

Console.WriteLine(jArray[i][j]);

}

The following jagged array stores two-dimensional arrays where the second bracket [,] indicates the two-dimensional array.

Example: Jagged Array

int[][,] jArray = new int[2][,];

jArray[0] = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };

jArray[1] = new int[2, 2] { { 7, 8 }, { 9, 10 } };

jArray[0][1, 1]; //returns 4

jArray[1][1, 0]; //returns 9

jArray[1][1, 1]; //returns 10

If you add one more bracket then it will be array of array of arry.

Example: Jagged Array

int[][][] intJaggedArray = new int[2][][]

{

new int[2][]

{

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

new int[2] { 4, 5}

},

new int[1][]

{

new int[3] { 7, 8, 9}

}

};

Console.WriteLine(intJaggedArray[0][0][0]); // 1

Console.WriteLine(intJaggedArray[0][1][1]); // 5

Console.WriteLine(intJaggedArray[1][0][2]); // 9

In the above example of a jagged array, three brackets [][][] means an array of array of array. So, intJaggedArray will contain two elements, which means two arrays. Now, each of these arrays also contains an array (single-dimension). intJaggedArray[0][0][0] points to the first element of first inner array. intJaggedArray[1][0][2] points to the third element of the second inner array. The following figure illustrates this.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/jagged-array.png)

# C# - Collections

C# includes specialized classes that store series of values or objects are called collections.

There are two types of collections available in C#: non-generic collections and generic collections.

The System.Collections namespace contains the non-generic collection types and System.Collections.Generic namespace includes generic collection types.

In most cases, it is recommended to use the generic collections because they perform faster than non-generic collections and also minimize exceptions by giving compile-time errors.

## Generic Collections

C# includes the following generic collection classes in the System.Collections.Generic namespace.

| Generic Collections | Description |
| --- | --- |
| List<T> | Generic List<T> contains elements of specified type. It grows automatically as you add elements in it. |
| Dictionary<TKey,TValue> | Dictionary<TKey,TValue> contains key-value pairs. |
| SortedList<TKey,TValue> | SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default. |
| Queue<T> | Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. |
| Stack<T> | Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. |
| Hashset<T> | Hashset<T> contains non-duplicate elements. It eliminates duplicate elements. |

## Non-generic Collections

| Non-generic Collections | Usage |
| --- | --- |
| ArrayList | ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically. |
| SortedList | SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection. |
| Stack | Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack. |
| Queue | Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue. |
| Hashtable | Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys. |
| BitArray | BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0). |

## C# - Array List

In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same as Array except that its size increases dynamically.

An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

### Create an ArrayList

The ArrayList class included in the System.Collections namespace. Create an object of the ArrayList using the new keyword.

Example: Create an ArrayList

using System.Collections;

ArrayList arlist = new ArrayList();

// or

var arlist = new ArrayList(); // recommended

## Adding Elements in ArrayList

Use the Add() method or object initializer syntax to add elements in an ArrayList.

An ArrayList can contain multiple null and duplicate values.

Example: Adding Elements in ArrayList

// adding elements using ArrayList.Add() method

var arlist1 = new ArrayList();

arlist1.Add(1);

arlist1.Add("Bill");

arlist1.Add(" ");

arlist1.Add(true);

arlist1.Add(4.5);

arlist1.Add(null);

// adding elements using object initializer syntax

var arlist2 = new ArrayList()

{

2, "Steve", " ", true, 4.5, null

};

};

Use the AddRange(ICollection c) method to add an entire Array, HashTable, SortedList, rrayList, BitArray, Queue, and Stack in the ArrayList.

Example: Adding Entire Array/ArrayList into ArrayList

var arlist1 = new ArrayList();

var arlist2 = new ArrayList()

{

1, "Bill", " ", true, 4.5, null

};

int[] arr = { 100, 200, 300, 400 };

Queue myQ = new Queue();

myQ.Enqueue("Hello");

myQ.Enqueue("World!");

arlist1.AddRange(arlist2); //adding arraylist in arraylist

arlist1.AddRange(arr); //adding array in arraylist

arlist1.AddRange(myQ); //adding Queue in arraylist

### Accessing an ArrayList

The ArrayList class implements the IList interface. So, elements can be accessed using indexer, in the same way as an array. Index starts from zero and increases by one for each subsequent element.

An explicit casting to the appropriate types is required, or use the var variable.

Example: Accessing Elements of ArrayList

var arlist = new ArrayList()

{

1,

"Bill",

300,

4.5f

};

//Access individual item using indexer

int firstElement = (int) arlist[0]; //returns 1

string secondElement = (string) arlist[1]; //returns "Bill"

//int secondElement = (int) arlist[1]; //Error: cannot cover string to int

//using var keyword without explicit casting

var firstElement = arlist[0]; //returns 1

var secondElement = arlist[1]; //returns "Bill"

//var fifthElement = arlist[5]; //Error: Index out of range

//update elements

arlist[0] = "Steve";

arlist[1] = 100;

//arlist[5] = 500; //Error: Index out of range

### Iterate an ArrayList

The ArrayList implements the ICollection interface that supports iteration of the collection types. So, use the foreach and the for loop to iterate an ArrayList. The Count property of an ArrayList returns the total number of elements in an ArrayList.

Example: Iterate ArrayList

ArrayList arlist = new ArrayList()

{

1,

"Bill",

300,

4.5F

};

foreach (var item in arlist)

Console.Write(item + ", "); //output: 1, Bill, 300, 4.5,

for(int i = 0 ; i < arlist.Count; i++)

Console.Write(arlist[i] + ", "); //output: 1, Bill, 300, 4.5,

### Insert Elements in ArrayList

Use the Insert() method to insert an element at the specified index into an ArrayList.

Signature: *void Insert(int index, Object value)*

Example: Insert Element in ArrayList

ArrayList arlist = new ArrayList()

{

1,

"Bill",

300,

4.5f

};

arlist.Insert(1, "Second Item");

foreach (var val in arlist)

Console.WriteLine(val);

Use the InsertRange() method to insert a collection in an ArrayList at the specfied index.

Signature: *Void InsertRange(int index, ICollection c)*

Example: Insert Collection in ArrayList

ArrayList arlist1 = new ArrayList()

{

100, 200, 600

};

ArrayList arlist2 = new ArrayList()

{

300, 400, 500

};

arlist1.InsertRange(2, arlist2);

foreach(var item in arlist1)

Console.Write(item + ", "); //output: 100, 200, 300, 400, 500, 600,

### Remove Elements from ArrayList

Use the Remove(), RemoveAt(), or RemoveRange methods to remove elements from an ArrayList.

Example: Remove Elements from ArrayList

ArrayList arList = new ArrayList()

{

1,

null,

"Bill",

300,

" ",

4.5f,

300,

};

arList.Remove(null); //Removes first occurance of null

arList.RemoveAt(4); //Removes element at index 4

arList.RemoveRange(0, 2);//Removes two elements starting from 1st item (0 index)

### Check Element in ArrayList

Use the Contains() method to determine whether the specified element exists in the ArrayList or not. It returns true if exists otherwise returns false.

Example: Check for Elements

ArrayList arList = new ArrayList()

{

1,

"Bill",

300,

4.5f,

300

};

Console.WriteLine(arList.Contains(300)); // true

Console.WriteLine(arList.Contains("Bill")); // true

Console.WriteLine(arList.Contains(10)); // false

Console.WriteLine(arList.Contains("Steve")); // false

It is not recommended to use the ArrayList class due to performance issue. Instead, use List<object> to store heterogeneous objects. To store data of same data type, use Generic List<T>.

### ArrayList Class

The following diagram illustrates the ArrayList class.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/arraylist.png)

### ArrayList Properties

| Properties | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the ArrayList can contain. |
| Count | Gets the number of elements actually contained in the ArrayList. |
| IsFixedSize | Gets a value indicating whether the ArrayList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the ArrayList is read-only. |
| Item | Gets or sets the element at the specified index. |

### ArrayList Methods

| Methods | Description |
| --- | --- |
| Add()/AddRange() | Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList. |
| Insert()/InsertRange() | Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList. |
| Remove()/RemoveRange() | Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList. |
| RemoveAt() | Removes the element at the specified index from the ArrayList. |
| Sort() | Sorts entire elements of the ArrayList. |
| Reverse() | Reverses the order of the elements in the entire ArrayList. |
| Contains | Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false. |
| Clear | Removes all the elements in ArrayList. |
| CopyTo | Copies all the elements or range of elements to compitible Array. |
| GetRange | Returns specified number of elements from specified index from ArrayList. |
| IndexOf | Search specified element and returns zero based index if found. Returns -1 if element not found. |
| ToArray | Returns compitible array from an ArrayList. |

## C# - List

The List<T> List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the generic version of the ArrayList that comes under System.Collection.GenericSystem.Collection.Generic namespace.

### List<T> Characteristics

* List<T> List<T> equivalent of the ArrayList, which implements IList<T>.
* It comes under System.Collections.Generic namespace.
* List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
* Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
* List<T> performs faster and less error-prone than the ArrayList.

### Creating a List

The List<T> List<T> is a generic collection, so you need to specify a type parameter for the type of data it can store. The following example shows how to create list and add elements.

Example: Adding elements in List

List<int> primeNumbers = new List<int>();

primeNumbers.Add(1); // adding elements using add() method

primeNumbers.Add(3);

primeNumbers.Add(5);

primeNumbers.Add(7);

var cities = new List<string>();

cities.Add("New York");

cities.Add("London");

cities.Add("Mumbai");

cities.Add("Chicago");

cities.Add(null);// nulls are allowed for reference type list

//adding elements using collection-initializer syntax

var bigCities = new List<string>()

{

"New York",

"London",

"Mumbai",

"Chicago"

};

};

In the above example, List<int> primeNumbers = new List<int>(); creates a list of int type. In the same way, cities and bigCities are string type list. You can then add elements in a list using the Add() method or the collection-initializer syntax.

You can also add elements of the custom classes using the collection-initializer syntax. The following adds objects of the Student class in the List<Student>.

Example: Add Custom Class Objects in List

var students = new List<Student>() {

new Student(){ Id = 1, Name="Bill"},

new Student(){ Id = 2, Name="Steve"},

new Student(){ Id = 3, Name="Ram"},

new Student(){ Id = 4, Name="Abdul"}

};

};

### Adding an Array in a List

Use the AddRange() method to add all the elements from an array or another collection to List.

AddRange() signature: void AddRange(IEnumerable<T> collection)

Example: Add Arrays in List

string[] cities = new string[3]{ "Mumbai", "London", "New York" };

var popularCities = new List<string>();

// adding an array in a List

popularCities.AddRange(cities);

var favouriteCities = new List<string>();

// adding a List

favouriteCities.AddRange(popularCities);

favouriteCities.AddRange(popularCities);

### Accessing a List

A list can be accessed by an index, a for/foreach loop, and using LINQ queries. Indexes of a list start from zero. Pass an index in the square brackets to access individual list items, same as array. Use a foreach or for loop to iterate a List<T> collection.

Example: Accessing List

List<int> numbers = new List<int>() { 1, 2, 5, 7, 8, 10 };

Console.WriteLine(numbers[0]); // prints 1

Console.WriteLine(numbers[1]); // prints 2

Console.WriteLine(numbers[2]); // prints 5

Console.WriteLine(numbers[3]); // prints 7

// using foreach LINQ method

numbers.ForEach(num => Console.WriteLine(num + ", "));//prints 1, 2, 5, 7, 8, 10,

// using for loop

for(int i = 0; i < numbers.Count; i++)

Console.WriteLine(numbers[i]);

### Accessing a List using LINQ

The List<T> implements the IEnumerable interface. So, we can query a list using LINQ query syntax or method syntax, as shown below.

Example: LINQ Query on List

var students = new List<Student>() {

new Student(){ Id = 1, Name="Bill"},

new Student(){ Id = 2, Name="Steve"},

new Student(){ Id = 3, Name="Ram"},

new Student(){ Id = 4, Name="Abdul"}

};

//get all students whose name is Bill

var result = from s in students

where s.Name == "Bill"

select s;

foreach(var student in result)

Console.WriteLine(student.Id + ", " + student.Name);

### Insert Elements in List

Use the Insert() method inserts an element into the List<T> collection at the specified index.

Insert() signature:void Insert(int index, T item);

Example: Insert elements into List

var numbers = new List<int>(){ 10, 20, 30, 40 };

numbers.Insert(1, 11);// inserts 11 at 1st index: after 10.

foreach (var num in numbers)

Console.Write(num);

### Remove Elements from List

Use the Remove() method to remove the first occurrence of the specified element in the List<T> collection. Use the RemoveAt() method to remove an element from the specified index. If no element at the specified index, then the ArgumentOutOfRangeException will be thrown.

Remove() signature: bool Remove(T item)

RemoveAt() signature: void RemoveAt(int index)

Example: Remove elements from List

var numbers = new List<int>(){ 10, 20, 30, 40, 10 };

numbers.Remove(10); // removes the first 10 from a list

numbers.RemoveAt(2); //removes the 3rd element (index starts from 0)

//numbers.RemoveAt(10); //throws ArgumentOutOfRangeException

foreach (var el in intList)

Console.Write(el); //prints 20 30

### Check Elements in List

Use the Contains() method to determine whether an element is in the List<T> or not.

Example: Contains()

var numbers = new List<int>(){ 10, 20, 30, 40 };

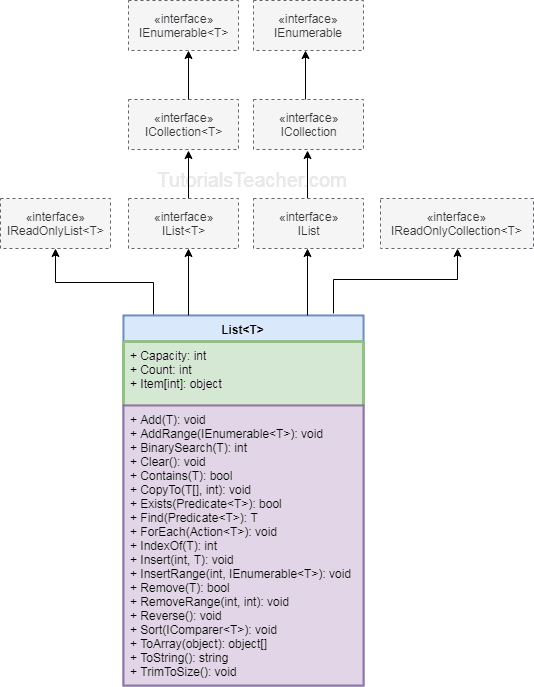
numbers.Contains(10); // returns true

numbers.Contains(11); // returns false

numbers.Contains(20); // returns true

### List<T> Class Hierarchy

The following diagram illustrates the List<T> hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/list.png)

### List<T> Class Properties and Methods

The following table lists the important properties and methods of List<T> class:

| Property | Usage |
| --- | --- |
| Items | Gets or sets the element at the specified index |
| Count | Returns the total number of elements exists in the List<T> |

| Method | Usage |
| --- | --- |
| Add | Adds an element at the end of a List<T>. |
| AddRange | Adds elements of the specified collection at the end of a List<T>. |
| BinarySearch | Search the element and returns an index of the element. |
| Clear | Removes all the elements from a List<T>. |
| Contains | Checks whether the specified element exists or not in a List<T>. |
| Find | Finds the first element based on the specified predicate function. |
| Foreach | Iterates through a List<T>. |
| Insert | Inserts an element at the specified index in a List<T>. |
| InsertRange | Inserts elements of another collection at the specified index. |
| Remove | Removes the first occurrence of the specified element. |
| RemoveAt | Removes the element at the specified index. |
| RemoveRange | Removes all the elements that match the supplied predicate function. |
| Sort | Sorts all the elements. |
| TrimExcess | Sets the capacity to the actual number of elements. |
| TrueForAll | Determines whether every element in the List<T> matches the conditions defined by the specified predicate. |

## C# - SortedList<TKey, TValue>

The SortedList<TKey, TValue>, and SortedList are collection classes that can store key-value pairs that are sorted by the keys based on the associated IComparer implementation. For example, if the keys are of primitive types, then sorted in ascending order of keys.

C# supports generic and non-generic SortedList. It is recommended to use generic SortedList<TKey, TValue> because it performs faster and less error-prone than the non-generic SortedList.

### SortedList Characteristics

* SortedList<TKey, TValue> SortedList<TKey, TValue> is an array of key-value pairs sorted by keys.
* Sorts elements as soon as they are added. Sorts primitive type keys in ascending order and object keys based on IComparer<T>.
* Comes under System.Collection.Generic namespace.
* A key must be unique and cannot be null.
* A value can be null or duplicate.
* A value can be accessed by passing associated key in the indexer mySortedList[key]
* Contains elements of type KeyValuePair<TKey, TValue>
* It uses less memory than SortedDictionary<TKey,TValue>.
* It is faster in the retrieval of data once sorted, whereas SortedDictionary<TKey, TValue> is faster in insertion and removing key-value pairs.

### Creating a SortedList

The following example demonstrates how to create a generic SortedList<TKey, TValue>, and add key-value pairs in it.

Example: Create a SortedList and Add Elements

//SortedList of int keys, string values

SortedList<int, string> numberNames = new SortedList<int, string>();

numberNames.Add(3, "Three");

numberNames.Add(1, "One");

numberNames.Add(2, "Two");

numberNames.Add(4, null);

numberNames.Add(10, "Ten");

numberNames.Add(5, "Five");

//The following will throw exceptions

//numberNames.Add("Three", 3); //Compile-time error: key must be int type

//numberNames.Add(1, "One"); //Run-time exception: duplicate key

//numberNames.Add(null, "Five");//Run-time exception: key cannot be null

In the above example, a generic SortedList<TKey, TValue> object is created by specifying the type of keys and values it is going to store. The SortedList<int, string> will store keys of int type and values of string type.

The Add() method is used to add a single key-value pair in a SortedList. Keys cannot be null or duplicate. If found, it will throw a run-time exception. Values can be duplicate and null if the type is nullable.

Use the collection-initializer syntax to initialize a SortedList with multiple key-value pairs at the time of instantiating, as shown below.

//Creating a SortedList of string keys, string values

//using collection-initializer syntax

SortedList<string,string> cities = new SortedList<string,string>()

{

{"London", "UK"},

{"New York", "USA"},

{ "Mumbai", "India"},

{"Johannesburg", "South Africa"}

};

};

The SortedList rearranges key-value pairs in the ascending order of keys as soon as a key-value pair added. The following example displays all the keys and values using foreach loop.

Example: SortedList Elements Default Sorting Order

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{5, "Five"},

{1, "One"}

};

Console.WriteLine("---Initial key-values--");

foreach(KeyValuePair<int, string> kvp in numberNames)

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

numberNames.Add(6, "Six");

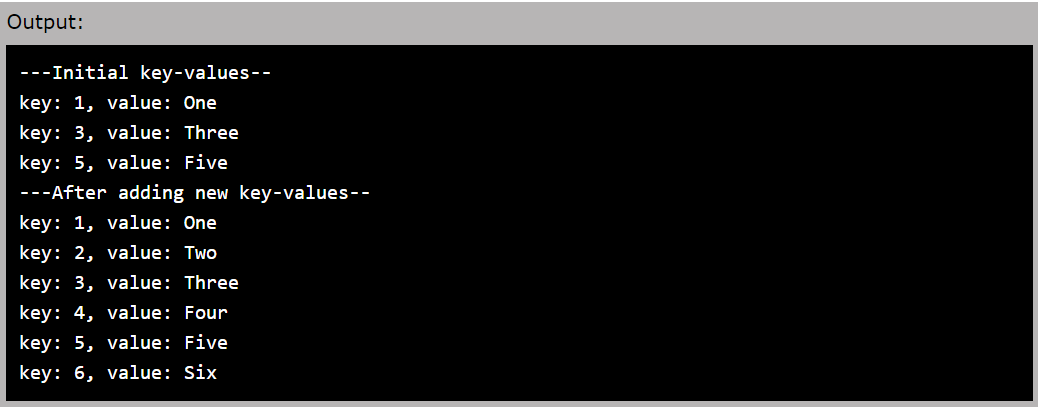
numberNames.Add(2, "Two");

numberNames.Add(4, "Four");

Console.WriteLine("---After adding new key-values--");

foreach(var kvp in numberNames)

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

Microsoft To Launch Cross-Platform Email Client Codenamed Project Monarch

### Accessing SortedList

Specify a key in the indexer sortedList[key], to get or set a value in the SortedList.

Example: Access SortedList Values

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

Console.WriteLine(numberNames[1]); //output: One

Console.WriteLine(numberNames[2]); //output: Two

Console.WriteLine(numberNames[3]); //output: Three

//Console.WriteLine(numberNames[10]); //run-time KeyNotFoundException

numberNames[2] = "TWO"; //updates value

numberNames[4] = "Four"; //adds a new key-value if a key does not exists

Above, numberNames[10] will throw a KeyNotFoundException because specified key 10 does not exist in a sortedlist. To prevent this exception, use ContainsKey() or TryGetValue() methods, as shown below.

Example: ContainsKey() and TryGetValue()

SortedList<int, string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

if(numberNames.ContainsKey(4)){

numberNames[4] = "four";

}

int result;

if(numberNames.TryGetValue(4, out result))

Console.WriteLine("Key: {0}, Value: {1}", 4, result);

Output:

Key:4, Value: Four

Use Keys and Values properties if you want to iterate a SortedList using a for loop.

Example: Iterate SortedList using For Loop

SortedList<int, string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

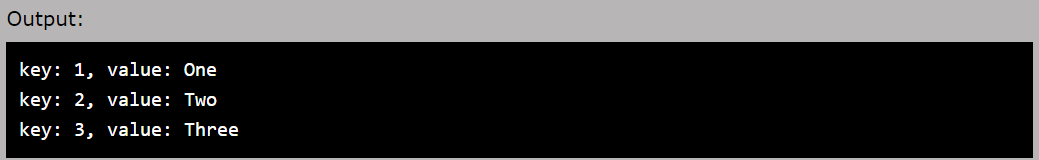
for (int i = 0; i < numberNames.Count; i++)

{

Console.WriteLine("key: {0}, value: {1}", numberNames.Keys[i], numberNames.Values[i]);

}

}



### Remove Elements from SortedList

Use the Remove(key) and RemoveAt(index) methods to remove key-value pairs from a SortedList.

Example: Remove Elements

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"},

{5, "Five"},

{4, "Four"}

};

numberNames.Remove(1);//removes key 1 pair

numberNames.Remove(10);//removes key 1 pair, no error if not exists

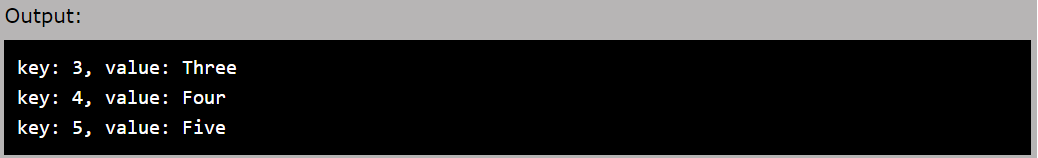
numberNames.RemoveAt(0);//removes key-value pair from index 0

//numberNames.RemoveAt(10);//run-time exception: ArgumentOutOfRangeException

foreach(var kvp in numberNames)

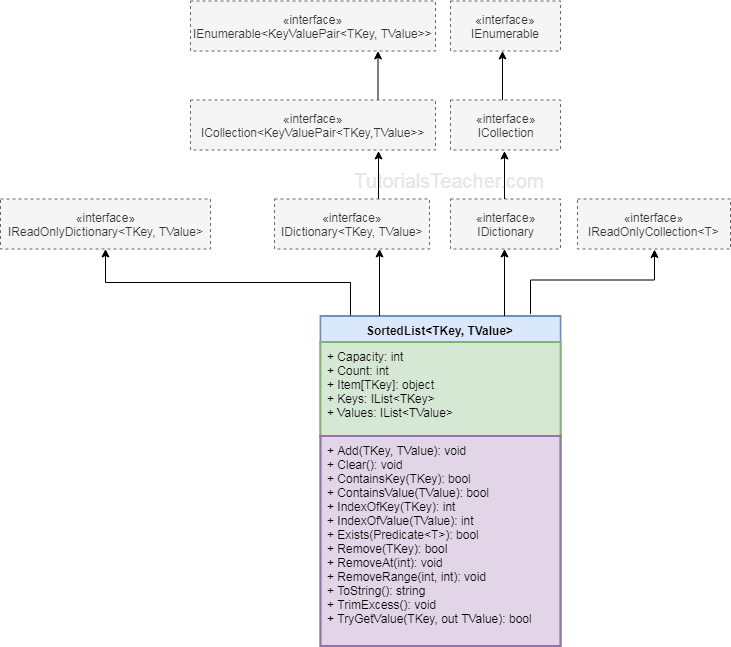
Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );



### SortedList Class Hierarchy

The following diagram illustrates the SortedList hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/sortedlist-generic.png)

## C# - Dictionary<TKey, TValue>

The Dictionary<TKey, TValue> is a generic collection that stores key-value pairs in no particular order.

### Dictionary Characteristics

* Dictionary<TKey, TValue>Dictionary<TKey, TValue> stores key-value pairs.
* Comes under System.Collections.Generic namespace.
* Implements IDictionary<TKey, TValue> interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g., myDictionary[key]
* Elements are stored as KeyValuePair<TKey, TValue> objects.

### Creating a Dictionary

You can create the Dictionary<TKey, TValue> object by passing the type of keys and values it can store. The following example shows how to create a dictionary and add key-value pairs.

Example: Create Dictionary and Add Elements

IDictionary<int, string> numberNames = new Dictionary<int, string>();

numberNames.Add(1,"One"); //adding a key/value using the Add() method

numberNames.Add(2,"Two");

numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.

//numberNames.Add(3, "Three");

foreach(KeyValuePair<int, string> kvp in numberNames)

Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

//creating a dictionary using collection-initializer syntax

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

foreach(var kvp in cities)

Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

In the above example, numberNames is a Dictionary<int, string> type dictionary, so it can store int keys and string values. In the same way, cities is a Dictionary<string, string> type dictionary, so it can store string keys and string values. Dictionary cannot include duplicate or null keys, whereas values can be duplicated or null. Keys must be unique otherwise, it will throw a runtime exception.

### Access Dictionary Elements

The Dictionary can be accessed using indexer. Specify a key to get the associated value. You can also use the ElementAt() method to get a KeyValuePair from the specified index.

Example: Access Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

Console.WriteLine(cities["UK"]); //prints value of UK key

Console.WriteLine(cities["USA"]);//prints value of USA key

//Console.WriteLine(cities["France"]); // run-time exception: Key does not exist

//use ContainsKey() to check for an unknown key

if(cities.ContainsKey("France")){

Console.WriteLine(cities["France"]);

}

//use TryGetValue() to get a value of unknown key

string result;

if(cities.TryGetValue("France", out result))

{

Console.WriteLine(result);

}

//use ElementAt() to retrieve key-value pair using index

for (int i = 0; i < cities.Count; i++)

{

Console.WriteLine("Key: {0}, Value: {1}",

cities.ElementAt(i).Key,

cities.ElementAt(i).Value);

}

cities.ElementAt(i).Key,

cities.ElementAt(i).Value);

}

### Update Dictionary

Update the value of a key by specifying a key in the indexer. It will throw the KeyNotFoundException if a key does not exist in the dictionary, therefore use the ContainsKey() method before accessing unknown keys.

Example: Update Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities["UK"] = "Liverpool, Bristol"; // update value of UK key

cities["USA"] = "Los Angeles, Boston"; // update value of USA key

//cities["France"] = "Paris"; //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){

cities["France"] = "Paris";

}

}

### Remove Elements in Dictionary

The Remove() method deletes an existing key-value pair from a dictionary. The Clear() method deletes all the elements of the dictionary.

Example: Remove Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities.Remove("UK"); // removes UK

//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it

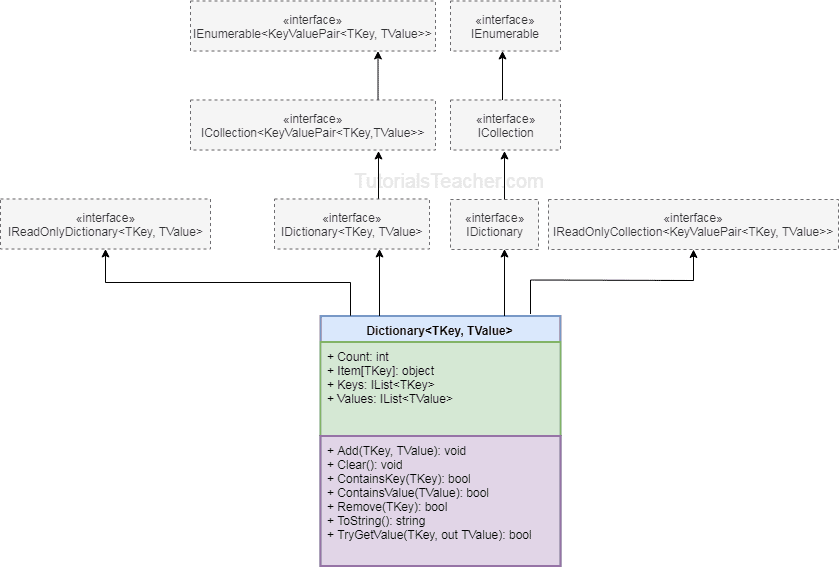
cities.Remove("France");

}

cities.Clear(); //removes all elements

### Dictionary Class Hierarchy

The following diagram illustrates the generic Dictionary class hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/generic-dictionary.png)

## C# - Hashtable

The Hashtable Hashtable is a non-generic collection that stores key-value pairs, similar to generic Dictionary<TKey, TValue> collection. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

### Hashtable Characteristics

* HashtableHashtable stores key-value pairs.
* Comes under System.Collection namespace.
* Implements IDictionary interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g. myHashtable[key]myHashtable[key]
* Elements are stored as DictionaryEntry objects.

### Creating a Hashtable

The following example demonstrates creating a Hashtable and adding elements.

Example: Create and Add Elements

Hashtable numberNames = new Hashtable();

numberNames.Add(1,"One"); //adding a key/value using the Add() method

numberNames.Add(2,"Two");

numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.

//numberNames.Add(3, "Three");

foreach(DictionaryEntry de in numberNames)

Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);

//creating a Hashtable using collection-initializer syntax

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

foreach(DictionaryEntry de in cities)

Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);

The HashtableHashtable collection can include all the elements of Dictionary, as shown below.

Example: Add Dictionary in Hashtable

Dictionary<int, string> dict = new Dictionary<int, string>();

dict.Add(1, "one");

dict.Add(2, "two");

dict.Add(3, "three");

Hashtable ht = new Hashtable(dict);

### Update Hashtable

You can retrieve the value of an existing key from the Hashtable by passing a key in indexer. The Hashtable is a non-generic collection, so you must type cast values while retrieving it.

Example: Update Hashtable

//creating a Hashtable using collection-initializer syntax

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

string citiesOfUK = (string) cities["UK"]; //cast to string

string citiesOfUSA = (string) cities["USA"]; //cast to string

Console.WriteLine(citiesOfUK);

Console.WriteLine(citiesOfUSA);

cities["UK"] = "Liverpool, Bristol"; // update value of UK key

cities["USA"] = "Los Angeles, Boston"; // update value of USA key

if(!cities.ContainsKey("France")){

cities["France"] = "Paris";

}

}

### Remove Elements in Hashtable

The Remove() method removes the key-value that match with the specified in the Hashtable. It throws the KeyNotfoundException if the specified key not found in the Hashtable, so check for an existing key using the ContainsKey() method before removing.

Use the Clear() method to remove all the elements in one shot.

Example: Remove Elements from Hashtable

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities.Remove("UK"); // removes UK

//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it

cities.Remove("France");

}

cities.Clear(); //removes all elements

### Hashtable Class Hierarchy

The following diagram illustrates the Hashtable class hierarchy.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/hashtable.png)

## C# - Stack<T>

StackStack is a special type of collection that stores elements in LIFO style (Last In First Out). C# includes the generic Stack<T> and non-generic Stack collection classes. It is recommended to use the generic Stack<T> collection.

Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.

### Stack<T> Characteristics

* Stack<T>Stack<T> is Last In First Out collection.
* It comes under System.Collection.Generic namespace.
* Stack<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Push() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Pop() and the Peek() methods. It does not support an indexer.

### Creating a Stack

You can create an object of the Stack<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Stack<T> using the Push() method. Stack allows null (for reference types) and duplicate values.

Example: Create and Add Elements in Stack

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

foreach (var item in myStack)

Console.Write(item + ","); //prints 4,3,2,1,

You can also create a Stack from an array, as shown below.

Example: Create and Add Elements in Stack

int[] arr = new int[]{ 1, 2, 3, 4};

Stack<int> myStack = new Stack<int>(arr);

foreach (var item in myStack)

Console.Write(item + ","); //prints 4,3,2,1,

### Stack<T> Properties and Methods:

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Stack. |

| Method | Usage |
| --- | --- |
| Push(T) | Inserts an item at the top of the stack. |
| Peek() | Returns the top item from the stack. |
| Pop() | Removes and returns items from the top of the stack. |
| Contains(T) | Checks whether an item exists in the stack or not. |
| Clear() | Removes all items from the stack. |

### Pop()

The Pop() method returns the last element and removes it from a stack. If a stack is empty, then it will throw the InvalidOperationException. So, always check for the number of elements in a stack before calling the Pop() method.

Example: Access Stack using Pop()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

Console.Write("Number of elements in Stack: {0}", myStack.Count);

while (myStack.Count > 0)

Console.Write(myStack.Pop() + ",");

Console.Write("Number of elements in Stack: {0}", myStack.Count);



### Peek()

The Peek() method returns the lastly added value from the stack but does not remove it. Calling the Peek() method on an empty stack will throw the InvalidOperationException. So, always check for elements in the stack before retrieving elements using the Peek() method.

Example: Retrieve Elements usign Peek()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

Console.Write("Number of elements in Stack: {0}", myStack.Count);// prints 4

if(myStack.Count > 0){

Console.WriteLine(myStack.Peek()); // prints 4

Console.WriteLine(myStack.Peek()); // prints 4

}

Console.Write("Number of elements in Stack: {0}", myStack.Count);// prints 4

### Contains()

The Contains() Contains() method checks whether the specified element exists in a Stack collection or not. It returns true if it exists, otherwise false.

Example: Contains()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

myStack.Contains(2); // returns true

myStack.Contains(10); // returns false

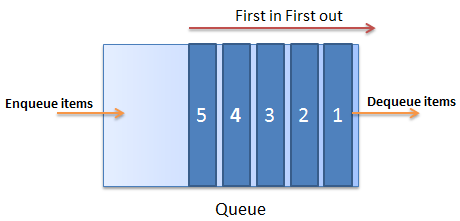
## C# - Queue<T>

QueueQueue is a special type of collection that stores the elements in FIFO style (First In First Out), exactly opposite of the Stack<T> collection. It contains the elements in the order they were added. C# includes generic Queue<T> and non-generic Queue collection. It is recommended to use the generic Queue<T> collection.

### Queue<T> Characteristics

* Queue<T>Queue<T> is FIFO (First In First Out) collection.
* It comes under System.Collection.Generic namespace.
* Queue<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Enqueue() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Dequeue() and the Peek() methods. It does not support an indexer.

The following figure illustrates the Queue collection:

[](https://www.tutorialsteacher.com/Content/images/csharp/csharp-queue.png)

### Creating a Queue

You can create an object of the Queue<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Queue<T> using the Enqueue() method. A Queue collection allows null (for reference types) and duplicate values.

Example: Create and Add Elements in the Queue

Queue<int> callerIds = new Queue<int>();

callerIds.Enqueue(1);

callerIds.Enqueue(2);

callerIds.Enqueue(3);

callerIds.Enqueue(4);

foreach(var id in callerIds)

Console.Write(id); //prints 1234

### Queue<T> Properties and Methods

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Queue. |

| Method | Usage |
| --- | --- |
| Enqueue(T) | Adds an item into the queue. |
| Dequeue | Returns an item from the beginning of the queue and removes it from the queue. |
| Peek(T) | Returns an first item from the queue without removing it. |
| Contains(T) | Checks whether an item is in the queue or not |
| Clear() | Removes all the items from the queue. |

### Retrieve Elements from a Queue

The Dequeue() and the Peek() method is used to retrieve the first element in a queue collection. The Dequeue() removes and returns the first element from a queue because the queue stores elements in FIFO order. Calling the Dequeue() method on an empty queue will throw the InvalidOperation exception. So, always check that the total count of a queue is greater than zero before calling it.

Example: Reading Queue

Queue<string> strQ = new Queue<string>();

strQ.Enqueue("H");

strQ.Enqueue("e");

strQ.Enqueue("l");

strQ.Enqueue("l");

strQ.Enqueue("o");

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

while (strQ.Count > 0)

Console.WriteLine(strQ.Dequeue()); //prints Hello

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 0

The Peek() method always returns the first item from a queue collection without removing it from the queue. Calling the Peek() method on an empty queue will throw a run-time exception InvalidOperationException.

Example: Peek()

Queue<string> strQ = new Queue<string>();

strQ.Enqueue("H");

strQ.Enqueue("e");

strQ.Enqueue("l");

strQ.Enqueue("l");

strQ.Enqueue("o");

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

if(strQ.Count > 0){

Console.WriteLine(strQ.Peek()); //prints H

Console.WriteLine(strQ.Peek()); //prints H

}

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

### Contains()

The Contains() method checks whether an item exists in a queue or not. It returns true if the specified item exists, otherwise returns false.

Contains() Signature: bool Contains(object obj);

Example: Contains()

Queue<int> callerIds = new Queue<int>();

callerIds.Enqueue(1);

callerIds.Enqueue(2);

callerIds.Enqueue(3);

callerIds.Enqueue(4);

callerIds.Contains(2); //true

callerIds.Contains(10); //false

## C# - HashSet

In C#, HashSet is an unordered collection of unique elements. This collection is introduced in *.NET 3.5*. It supports the implementation of sets and uses the hash table for storage. This collection is of the generic type collection and it is defined under *System.Collections.Generic* namespace. It is generally used when we want to prevent duplicate elements from being placed in the collection. The performance of the HashSet is much better in comparison to the list.

Important Points:

* The HashSet class implements the ICollection, IEnumerable, IReadOnlyCollection, ISet, IEnumerable, IDeserializationCallback, and ISerializable interfaces.
* In HashSet, the order of the element is not defined. You cannot sort the elements of HashSet.
* In HashSet, the elements must be unique.
* In HashSet, duplicate elements are not allowed.
* Is provides many mathematical set operations, such as intersection, union, and difference.
* The capacity of a HashSet is the number of elements it can hold.
* A HashSet is a dynamic collection means the size of the HashSet is automatically increased when the new elements are added.
* In HashSet, you can only store the same type of elements.

### How to create a HashSet?

The HashSet class provides*7 different types of constructors* which are used to create a HashSet, here we only use *HashSet()*, constructor.

HashSet(): It is used to create an instance of the HashSet class that is empty and uses the default equality comparer for the set type.

Step 1: Include System.Collections.Generic namespace in your program with the help of using keyword:

using System.Collections.Generic;

Step 2: Create a HashSet using the HashSet class as shown below:

HashSet<Type\_of\_hashset> Hashset\_name = new HashSet<Type\_of\_hashset>();

Step 3: If you want to add elements in your HashSet, then use *Add()* method to add elements in your HashSet. And you can also store elements in your HashSet using collection initializer.

Step 4: The elements of HashSet is accessed by using a *foreach*loop. As shown in the below example.

**Example:**

// C# program to illustrate how to

// create hashset

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

Console.WriteLine("Elements of myhash1:");

// Accessing elements of HashSet

// Using foreach loop

foreach(var val in myhash1)

{

Console.WriteLine(val);

}

// Creating another HashSet

// using collection initializer

// to initialize HashSet

HashSet<int> myhash2 = new HashSet<int>() {10,

100,1000,10000,100000};

// Display elements of myhash2

Console.WriteLine("Elements of myhash2:");

foreach(var value in myhash2)

{

Console.WriteLine(value);

}

}

}

**Output**:

Elements of myhash1:

C

C++

C#

Java

Ruby

Elements of myhash2:

10

100

1000

10000

100000

### How to remove elements from the HashSet?

In HashSet, you are allowed to remove elements from the HashSet. HashSet<T> class provides three different methods to remove elements and the methods are:

* Remove(T): This method is used to remove the specified element from a HashSet object.
* RemoveWhere(Predicate): This method is used to remove all elements that match the conditions defined by the specified predicate from a HashSet collection.
* Clear: This method is used to remove all elements from a HashSet object.

**Example 1:**

// C# program to illustrate how to

// remove elements of HashSet

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash.Add("C");

myhash.Add("C++");

myhash.Add("C#");

myhash.Add("Java");

myhash.Add("Ruby");

// Before using Remove method

Console.WriteLine("Total number of elements present (Before Removal)"+

" in myhash: {0}", myhash.Count);

// Remove element from HashSet

// Using Remove method

myhash.Remove("Ruby");

// After using Remove method

Console.WriteLine("Total number of elements present (After Removal)"+

" in myhash: {0}", myhash.Count);

// Remove all elements from HashSet

// Using Clear method

myhash.Clear();

Console.WriteLine("Total number of elements present"+

" in myhash:{0}", myhash.Count);

}

}

**Output:**

Total number of elements present in myhash: 5

Total number of elements present in myhash: 4

Total number of elements present in myhash:0

### Set Operations

HashSet class also provides some methods that are used to perform different operations on sets and the methods are:

* [UnionWith(IEnumerable)](https://www.geeksforgeeks.org/c-union-of-two-hashset/): This method is used to modify the current HashSet object to contain all elements that are present in itself, the specified collection, or both.  
  **Example**:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using UnionWith method

myhash1.UnionWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C

C++

C#

Java

Ruby

PHP

Perl

* [IntersectWith(IEnumerable)](https://www.geeksforgeeks.org/c-intersection-of-two-hashsets/): This method is used to modify the current HashSet object to contain only elements that are present in that object and in the specified collection.  
  **Example**:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using IntersectWith method

myhash1.IntersectWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C++

Java

* [ExceptWith(IEnumerable)](https://www.geeksforgeeks.org/c-remove-all-elements-in-a-collection-from-a-hashset/): This method is used to remove all elements in the specified collection from the current HashSet object.

**Example**:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using ExceptWith method

myhash1.ExceptWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C

C#

Ruby

Let us see an example to remove duplicate strings using C# HashSet.

**Example:**

using System;

using System.Collections.Generic;

using System.Linq;

class Program {

   static void Main() {

      string[] arr1 = {"Table","Chair","Pen","Clip","Table"};

      Console.WriteLine(string.Join(",", arr1));

      // HashSet

      var h = new HashSet<string>(arr1);

      // eliminates duplicate words

      string[] arr2 = h.ToArray();

      Console.WriteLine(string.Join(",", arr2));

   }

}

# C# - Type Conversions

Type conversion happens when we assign the value of one data type to another. If the data types are compatible, then C# does Automatic Type Conversion. If not comparable, then they need to be converted explicitly which is known as Explicit Type conversion. For example, assigning an int value to a long variable.

## Implicit Type Casting / Automatic Type Conversion

It happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type.

For Example, in C#, the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other. Before converting, the compiler first checks the compatibility according to the following figure and then it decides whether it is alright or there some error.

**Following table shows the implicit types of conversion that is supported by C# :**

|  |  |
| --- | --- |
| Convert from Data Type | **Convert to Data Type** |
| byte | short, int, long, float, double |
| short | int, long, float, double |
| int | long, float, double |
| long | float, double |
| float | double |

**Example**:

// C# program to demonstrate the

// Implicit Type Conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

int i = 57;

// automatic type conversion

long l = i;

// automatic type conversion

float f = l;

// Display Result

Console.WriteLine("Int value " +i);

Console.WriteLine("Long value " +l);

Console.WriteLine("Float value " +f);

}

}

}

**Output**:

Int value 57

Long value 57

Float value 57

## Explicit Type Casting

There may be compilation error when types not compatible with each other. For example, assigning double value to int data type:

// C# program to illustrate incompatible data

// type for explicit type conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

double d = 765.12;

// Incompatible Data Type

int i = d;

// Display Result

Console.WriteLine("Value of i is ", +i);

}

}

}

**Error**:

prog.cs(14,21): error CS0266: Cannot implicitly convert type `double' to `int'.

An explicit conversion exists (are you missing a cast?)

So, if we want to assign a value of larger data type to a smaller data type we perform explicit type casting.

* This is useful for incompatible data types where automatic conversion cannot be done.
* Here, target-type specifies the desired type to convert the specified value to.
* Sometimes, it may result into the lossy conversion.

**Example**:

// C# program to demonstrate the

// Explicit Type Conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

double d = 765.12;

// Explicit Type Casting

int i = (int)d;

// Display Result

Console.WriteLine("Value of i is " +i);

}

}

}

**Output**:

Value of i is 765

**Explanation**:   
Here due to lossy conversion, the value of i becomes 765 and there is a loss of 0.12 value.

C# provides built-in methods for Type-Conversions as follows:

|  |  |  |
| --- | --- | --- |
| **Method** |  | **Description** |
| ToBoolean |  | It will converts a type to Boolean value |
| ToChar |  | It will converts a type to a character value |
| ToByte |  | It will converts a value to Byte Value |
| ToDecimal |  | It will converts a value to Decimal point value |
| ToDouble |  | It will converts a type to double data type |
| ToInt16 |  | It will converts a type to 16-bit integer |
| ToInt32 |  | It will converts a type to 32 bit integer |
| ToInt64 |  | It will converts a type to 64 bit integer |
| ToString |  | It will converts a given type to string |
| ToUInt16 |  | It will converts a type to unsigned 16 bit integer |
| ToUInt32 |  | It will converts a type to unsigned 32 bit integer |
| ToUInt64 |  | It will converts a type to unsigned 64 bit integer |

**Example**:

// C# program to demonstrate the

// Built- In Type Conversion Methods

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

int i = 12;

double d = 765.12;

float f = 56.123F;

// Using Built- In Type Conversion

// Methods & Displaying Result

Console.WriteLine(Convert.ToString(f));

Console.WriteLine(Convert.ToInt32(d));

Console.WriteLine(Convert.ToUInt32(f));

Console.WriteLine(Convert.ToDouble(i));

Console.WriteLine("AjaySingala");

}

}

}

**Output**:

56.123

765

56

12

AjaySingala

# C# - Serialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object to be able to recreate it when needed. The reverse process is called deserialization.

Often, we need to store objects to a physical storage so it can be read back and converted back to an object. The process of storing an object to a physical storage is called serialization. The process of reading a serialized object back into memory is deserialization.

**Note**: Serialization and Deserialization are also known as Marshal and Unmarshal respectively.

In simple words serialization in C# is a process of storing the object instance to a persistent storage. Serialization stores state of objects i.e., member variable values to persistent storage such as a disk. Deserialization is reverse of serialization. It is a process of reading objects from a file where they have been stored. In this code sample we will see how to serialize and deserialize objects using C#.

## How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

## Namespaces involved

Following namespaces are involved in serialization process,

* System.Runtime.Serialization
* System.Runtime.Serialization.Formatter
* System.Runtime.Serialization.Formatters.Binary
* System.Text.Json
* System.Text.Json.Serialization
* System.Xml.Serialization

## Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

* Sending the object to a remote application by using a web service
* Passing an object from one domain to another
* Passing an object through a firewall as a JSON or XML string
* Maintaining security or user-specific information across applications

## JSON serialization

The System.Text.Json namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to the RFC 8259 JSON specification. To control the way JsonSerializer serializes or deserializes an instance of the class:

* Use a JsonSerializerOptions object
* Apply attributes from the System.Text.Json.Serialization namespace to classes or properties
* Implement custom converters

## Binary and XML serialization

The System.Runtime.Serialization namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

**Warning:** Binary serialization can be dangerous.

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. System.Xml.Serialization contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the XmlSerializer serializes or deserializes an instance of the class.

## Making an object serializable

For binary or XML serialization, you need:

* The object to be serialized
* A stream to contain the serialized object
* A System.Runtime.Serialization.Formatter instance

Apply the SerializableAttribute attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the SerializableAttribute attribute.

To prevent a field from being serialized, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

## Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

## Example 1: Binary Serialization

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize) b.Deserialize(s);

Console.WriteLine(c.name);

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

public int age = 100;

public string name = "bipin";

}

### Explanation

Here we have our own class named ClassToSerialize. This class has two public valiables name and age with some default values. We will write this class to a disk file (temp.dat) using SerializeTest class.

SerializeTest class has two methods SerializeNow() and DeSerializeNow() which perform the task of serialization and deserialization respectively.

The general steps for serializing are,

* Create an instance of File that will store serialized object.
* Create a stream from the file object.
* Create an instance of BinaryFormatter.
* Call serialize method of the instance passing it stream and object to serialize.

The steps for de-serializing the object are similar. The only change is that you need to call deserialize method of BinaryFormatter object.

## Example 2: Class / Object Serialization

Now, let us see an example where we have used 'real' class with public and shared members and properties to encapsulate them. The class also uses another supporting class. This is just to make clear that if your class contains further classes, all the classes in the chain will be serialized.

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize c = new ClassToSerialize();

c.Name = "bipin";

c.Age = 26;

ClassToSerialize.CompanyName = "xyz";

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize) b.Deserialize(s);

Console.WriteLine("Name :" + c.Name);

Console.WriteLine("Age :" + c.Age);

Console.WriteLine("Company Name :" + ClassToSerialize.CompanyName);

Console.WriteLine("Company Name :" + c.GetSupportClassString());

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

private int age;

private string name;

static string companyname;

SupportClass supp = new SupportClass();

public ClassToSerialize() {

supp.SupportClassString = "In support class";

}

public int Age {

get {

return age;

}

set {

age = value;

}

}

public string Name {

get {

return name;

}

set {

name = value;

}

}

public static string CompanyName {

get {

return companyname;

}

set {

companyname = value;

}

}

public string GetSupportClassString() {

return supp.SupportClassString;

}

}

public class SupportClass {

public string SupportClassString;

}

## Example 3: Array Serialization

This example shows how to serialize array of objects.

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize[] c = new ClassToSerialize[3];

c[0] = new ClassToSerialize();

c[0].Name = "bipin";

c[0].Age = 26;

c[1] = new ClassToSerialize();

c[1].Name = "abc";

c[1].Age = 75;

c[2] = new ClassToSerialize();

c[2].Name = "pqr";

c[2].Age = 50;

ClassToSerialize.CompanyName = "xyz";

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize[] c;

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize[]) b.Deserialize(s);

Console.WriteLine("Name :" + c[2].Name);

Console.WriteLine("Age :" + c[2].Age);

Console.WriteLine("Company Name :" + ClassToSerialize.CompanyName);

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

private int age;

private string name;

static string companyname;

public int Age {

get {

return age;

}

set {

age = value;

}

}

public string Name {

get {

return name;

}

set {

name = value;

}

}

public static string CompanyName {

get {

return companyname;

}

set {

companyname = value;

}

}

}

## Example 4: Serialize Class / Object

using System;

using System.IO;

using System.Linq;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

[Serializable]

class Tutorial

{

public int ID;

public String Name;

static void Main(string[] args)

{

Tutorial obj = new Tutorial();

obj.ID = 1;

obj.Name = ".Net";

IFormatter formatter = new BinaryFormatter();

Stream stream = new FileStream(@"E:\ExampleNew.txt",FileMode.Create,FileAccess.Write);

formatter.Serialize(stream, obj);

stream.Close();

stream = new FileStream(@"E:\ExampleNew.txt",FileMode.Open,FileAccess.Read);

Tutorial objnew = (Tutorial)formatter.Deserialize(stream);

Console.WriteLine(objnew.ID);

Console.WriteLine(objnew.Name);

Console.ReadKey();

}

}

}

### Code Explanation:

1. The class which needs to be serialized needs to have the [Serializable] attribute. This is a keyword in C#. This keyword is then attached to the Tutorial class. If you don’t mention this attribute, you will get an error when you try to serialize the class.
2. Next is the definition of the class which will be serialized. Here we are defining a class called “Tutorial” and providing 2 properties, one is “ID” and the other is “Name.”
3. First, we create an object of the Tutorial class. We then assign the value of “1” to ID and “.net” to the name property.
4. We then use the formatter class which is used to serialize or convert the object to a binary format. The data in the file in serialization is done in binary format. Next, we create a file stream object. The file stream object is used to open the file Example.txt for writing purposes. The keywords FileMode.Create and FileMode.Write is used to specifically mention that the file should be opened for writing purposes.
5. Finally, we use the Serialize method to transfer the binary data to the file. We then close the stream since the write operation is complete.
6. Ensure that the data is present in the file, we use deserialization to deserialize the object from the file.
7. We create the object “stream” to open the file Example.txt in reading only mode.
8. We then use the formatter class which is used to deserialize the object, which is stored in the Example.txt file. The object returned is set to the object objnew.
9. Finally, we display the properties of the object “objnew” to the console using the “ID” and “name” properties.

# Serialize and Deserialize JSON

Let’s see how to use the System.Text.Json namespace to serialize to and deserialize from JavaScript Object Notation (JSON).

The code samples here:

* Use the library directly, not through a framework such as ASP.NET Core.
* Use the JsonSerializer class with custom types to serialize from and deserialize into.
* Use the WriteIndented option to format the JSON for human readability when that is helpful.

For production use, you would typically accept the default value of false for this setting, since adding unnecessary whitespace may incur a negative impact on performance and bandwidth usage.

## Namespaces

The System.Text.Json namespace contains all the entry points and the main types. The System.Text.Json.Serialization namespace contains attributes and APIs for advanced scenarios and customization specific to serialization and deserialization. The code examples shown in this article require using directives for one or both namespaces:

using System.Text.Json;

using System.Text.Json.Serialization;

**Important:** Attributes from the **System.Runtime.Serialization** namespace aren't supported in System.Text.Json.

## Serialize (Write) .NET objects as JSON

To write JSON to a string or to a file, call the JsonSerializer.Serialize method.

### Serialize JSON Serialization

The following example creates JSON as a string:

using System;

using System.Text.Json;

namespace SerializeBasic

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static void Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot"

};

string jsonString = JsonSerializer.Serialize(weatherForecast);

Console.WriteLine(jsonString);

}

}

}

// output:

//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}

The JSON output is minified (whitespace, indentation, and new-line characters are removed) by default.

### Serialize JSON to a File

The following example uses synchronous code to create a JSON file:

using System;

using System.IO;

using System.Text.Json;

namespace SerializeToFile

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static void Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot"

};

string fileName = "WeatherForecast.json";

string jsonString = JsonSerializer.Serialize(weatherForecast);

File.WriteAllText(fileName, jsonString);

Console.WriteLine(File.ReadAllText(fileName));

}

}

}

// output:

//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}

### Serialize JSON to a File Asynchronously

The following example uses asynchronous code to create a JSON file:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

namespace SerializeToFileAsync

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static async Task Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot"

};

string fileName = "WeatherForecast.json";

using FileStream createStream = File.Create(fileName);

await JsonSerializer.SerializeAsync(createStream, weatherForecast);

await createStream.DisposeAsync();

Console.WriteLine(File.ReadAllText(fileName));

}

}

}

// output:

//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}

### Serialize JSON using Generics

The preceding examples use type inference for the type being serialized. An overload of Serialize() takes a generic type parameter:

using System;

using System.Text.Json;

namespace SerializeWithGenericParameter

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static void Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot"

};

string jsonString =

JsonSerializer.Serialize<WeatherForecast>(weatherForecast);

Console.WriteLine(jsonString);

}

}

}

// output:

//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}

### Serialization of Class with Collections and User Defined Types

Here's an example showing how a class that contains collection properties and a user-defined type is serialized:

using System;

using System.Collections.Generic;

using System.Text.Json;

namespace SerializeExtra

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

public string SummaryField;

public IList<DateTimeOffset> DatesAvailable { get; set; }

public Dictionary<string, HighLowTemps> TemperatureRanges { get; set; }

public string[] SummaryWords { get; set; }

}

public class HighLowTemps

{

public int High { get; set; }

public int Low { get; set; }

}

public class Program

{

public static void Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot",

SummaryField = "Hot",

DatesAvailable = new List<DateTimeOffset>()

{ DateTime.Parse("2019-08-01"), DateTime.Parse("2019-08-02") },

TemperatureRanges = new Dictionary<string, HighLowTemps>

{

["Cold"] = new HighLowTemps { High = 20, Low = -10 },

["Hot"] = new HighLowTemps { High = 60 , Low = 20 }

},

SummaryWords = new[] { "Cool", "Windy", "Humid" }

};

var options = new JsonSerializerOptions { WriteIndented = true };

string jsonString = JsonSerializer.Serialize(weatherForecast, options);

Console.WriteLine(jsonString);

}

}

}

// output:

//{

// "Date": "2019-08-01T00:00:00-07:00",

// "TemperatureCelsius": 25,

// "Summary": "Hot",

// "DatesAvailable": [

// "2019-08-01T00:00:00-07:00",

// "2019-08-02T00:00:00-07:00"

// ],

// "TemperatureRanges": {

// "Cold": {

// "High": 20,

// "Low": -10

// },

// "Hot": {

// "High": 60,

// "Low": 20

// }

// },

// "SummaryWords": [

// "Cool",

// "Windy",

// "Humid"

// ]

//}

## Deserialize (Read) JSON as .NET Objects

A common way to deserialize JSON is to first create a class with properties and fields that represent one or more of the JSON properties. Then, to deserialize from a string or a file, call the JsonSerializer.Deserialize method. For the generic overloads, you pass the type of the class you created as the generic type parameter. For the non-generic overloads, you pass the type of the class you created as a method parameter. You can deserialize either synchronously or asynchronously. Any JSON properties that aren't represented in your class are ignored.

### Deserialize a JSON String

The following example shows how to deserialize a JSON string:

using System;

using System.Collections.Generic;

using System.Text.Json;

namespace DeserializeExtra

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

public string SummaryField;

public IList<DateTimeOffset> DatesAvailable { get; set; }

public Dictionary<string, HighLowTemps> TemperatureRanges { get; set; }

public string[] SummaryWords { get; set; }

}

public class HighLowTemps

{

public int High { get; set; }

public int Low { get; set; }

}

public class Program

{

public static void Main()

{

string jsonString =

@"{

""Date"": ""2019-08-01T00:00:00-07:00"",

""TemperatureCelsius"": 25,

""Summary"": ""Hot"",

""DatesAvailable"": [

""2019-08-01T00:00:00-07:00"",

""2019-08-02T00:00:00-07:00""

],

""TemperatureRanges"": {

""Cold"": {

""High"": 20,

""Low"": -10

},

""Hot"": {

""High"": 60,

""Low"": 20

}

},

""SummaryWords"": [

""Cool"",

""Windy"",

""Humid""

]

}

";

WeatherForecast weatherForecast =

JsonSerializer.Deserialize<WeatherForecast>(jsonString);

Console.WriteLine($"Date: {weatherForecast.Date}");

Console.WriteLine($"TemperatureCelsius: {weatherForecast.TemperatureCelsius}");

Console.WriteLine($"Summary: {weatherForecast.Summary}");

}

}

}

// output:

//Date: 8/1/2019 12:00:00 AM -07:00

//TemperatureCelsius: 25

//Summary: Hot

### Deserialize JSON from a File

To deserialize from a file by using synchronous code, read the file into a string, as shown in the following example:

using System;

using System.IO;

using System.Text.Json;

namespace DeserializeFromFile

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static void Main()

{

string fileName = "WeatherForecast.json";

string jsonString = File.ReadAllText(fileName);

WeatherForecast weatherForecast =

await JsonSerializer.Deserialize<WeatherForecast>(jsonString);

Console.WriteLine($"Date: {weatherForecast.Date}");

Console.WriteLine($"TemperatureCelsius: {weatherForecast.TemperatureCelsius}");

Console.WriteLine($"Summary: {weatherForecast.Summary}");

}

}

}

// output:

//Date: 8/1/2019 12:00:00 AM -07:00

//TemperatureCelsius: 25

//Summary: Hot

### Deserialize JSON from a File Asynchronously

To deserialize from a file by using asynchronous code, call the DeserializeAsync method:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

namespace DeserializeFromFileAsync

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static async Task Main()

{

string fileName = "WeatherForecast.json";

using FileStream openStream = File.OpenRead(fileName);

WeatherForecast weatherForecast =

await JsonSerializer.DeserializeAsync<WeatherForecast>(openStream);

Console.WriteLine($"Date: {weatherForecast.Date}");

Console.WriteLine($"TemperatureCelsius: {weatherForecast.TemperatureCelsius}");

Console.WriteLine($"Summary: {weatherForecast.Summary}");

}

}

}

// output:

//Date: 8/1/2019 12:00:00 AM -07:00

//TemperatureCelsius: 25

//Summary: Hot

**Tip**

If you have JSON that you want to deserialize, and you don't have the class to deserialize it into, you have options other than manually creating the class that you need:

* Deserialize into a **JSON DOM (document object model)** and extract what you need from the DOM.
* The DOM lets you navigate to a subsection of a JSON payload and deserialize a single value, a custom type, or an array. For information about the **JsonNode** DOM in .NET 6, see **Deserialize subsections of a JSON payload**. For information about the **JsonDocument** DOM, see **How to search a JsonDocument and JsonElement for sub-elements**.
* Use the **Utf8JsonReader** directly.
* Use Visual Studio 2019 to automatically generate the class you need:
  + Copy the JSON that you need to deserialize.
  + Create a class file and delete the template code.
  + Choose Edit > Paste Special > Paste JSON as Classes. The result is a class that you can use for your deserialization target.

## Serialize to Formatted JSON

To pretty-print the JSON output, set JsonSerializerOptions.WriteIndented to true:

using System;

using System.Text.Json;

namespace SerializeWriteIndented

{

public class WeatherForecast

{

public DateTimeOffset Date { get; set; }

public int TemperatureCelsius { get; set; }

public string Summary { get; set; }

}

public class Program

{

public static void Main()

{

var weatherForecast = new WeatherForecast

{

Date = DateTime.Parse("2019-08-01"),

TemperatureCelsius = 25,

Summary = "Hot"

};

var options = new JsonSerializerOptions { WriteIndented = true };

string jsonString = JsonSerializer.Serialize(weatherForecast, options);

Console.WriteLine(jsonString);

}

}

}

// output:

//{

// "Date": "2019-08-01T00:00:00-07:00",

// "TemperatureCelsius": 25,

// "Summary": "Hot"

//}

If you use JsonSerializerOptions repeatedly with the same options, don't create a new JsonSerializerOptions instance each time you use it. Reuse the same instance for every call

## Include fields

Use the JsonSerializerOptions.IncludeFields global setting or the [JsonInclude] attribute to include fields when serializing or deserializing, as shown in the following example:

using System;

using System.Text.Json;

using System.Text.Json.Serialization;

namespace Fields

{

public class Forecast

{

public DateTime Date;

public int TemperatureC;

public string Summary;

}

public class Forecast2

{

[JsonInclude]

public DateTime Date;

[JsonInclude]

public int TemperatureC;

[JsonInclude]

public string Summary;

}

public class Program

{

public static void Main()

{

var json =

@"{""Date"":""2020-09-06T11:31:01.923395"",""TemperatureC"":-1,""Summary"":""Cold""} ";

Console.WriteLine($"Input JSON: {json}");

var options = new JsonSerializerOptions

{

IncludeFields = true,

};

var forecast = JsonSerializer.Deserialize<Forecast>(json, options);

Console.WriteLine($"forecast.Date: {forecast.Date}");

Console.WriteLine($"forecast.TemperatureC: {forecast.TemperatureC}");

Console.WriteLine($"forecast.Summary: {forecast.Summary}");

var roundTrippedJson =

JsonSerializer.Serialize<Forecast>(forecast, options);

Console.WriteLine($"Output JSON: {roundTrippedJson}");

var forecast2 = JsonSerializer.Deserialize<Forecast2>(json);

Console.WriteLine($"forecast2.Date: {forecast2.Date}");

Console.WriteLine($"forecast2.TemperatureC: {forecast2.TemperatureC}");

Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

roundTrippedJson = JsonSerializer.Serialize<Forecast2>(forecast2);

Console.WriteLine($"Output JSON: {roundTrippedJson}");

}

}

}

// Produces output like the following example:

//

//Input JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}

//forecast.Date: 9/6/2020 11:31:01 AM

//forecast.TemperatureC: -1

//forecast.Summary: Cold

//Output JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}

//forecast2.Date: 9/6/2020 11:31:01 AM

//forecast2.TemperatureC: -1

//forecast2.Summary: Cold

//Output JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}

To ignore read-only fields, use the JsonSerializerOptions.IgnoreReadOnlyFields global setting.

## HttpClient and HttpContent extension methods

Serializing and deserializing JSON payloads from the network are common operations. Extension methods on HttpClient and HttpContent let you do these operations in a single line of code. These extension methods use web defaults for JsonSerializerOptions.

The following example illustrates use of HttpClientJsonExtensions.GetFromJsonAsync and HttpClientJsonExtensions.PostAsJsonAsync:

using System;

using System.Net.Http;

using System.Net.Http.Json;

using System.Threading.Tasks;

namespace HttpClientExtensionMethods

{

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public string Username { get; set; }

public string Email { get; set; }

}

public class Program

{

public static async Task Main()

{

using HttpClient client = new()

{

BaseAddress = new Uri("https://jsonplaceholder.typicode.com")

};

// Get the user information.

User user = await client.GetFromJsonAsync<User>("users/1");

Console.WriteLine($"Id: {user.Id}");

Console.WriteLine($"Name: {user.Name}");

Console.WriteLine($"Username: {user.Username}");

Console.WriteLine($"Email: {user.Email}");

// Post a new user.

HttpResponseMessage response = await client.PostAsJsonAsync("users", user);

Console.WriteLine(

$"{(response.IsSuccessStatusCode ? "Success" : "Error")} - {response.StatusCode}");

}

}

}

// Produces output like the following example but with different names:

//

//Id: 1

//Name: Tyler King

//Username: Tyler

//Email: Tyler @contoso.com

//Success - Created

There are also extension methods for System.Text.Json on HttpContent.

# Serialize and Deserialize XML

## Serialize (Read) Data from an XML File

This example reads object data that was previously written to an XML file using the XmlSerializer class.

**Example:**

public class Book

{

public String title;

}

public void ReadXML()

{

// First write something so that there is something to read ...

var b = new Book { title = "Serialization Overview" };

var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));

var wfile =

new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");

writer.Serialize(wfile, b);

wfile.Close();

// Now we can read the serialized book ...

System.Xml.Serialization.XmlSerializer reader =

new System.Xml.Serialization.XmlSerializer(typeof(Book));

System.IO.StreamReader file = new System.IO.StreamReader(

@"c:\temp\SerializationOverview.xml");

Book overview = (Book)reader.Deserialize(file);

file.Close();

Console.WriteLine(overview.title);

}

### Compiling the Code

Replace the file name "c:\temp\SerializationOverview.xml" with the name of the file containing the serialized data. For more information about serializing data, see [How to write object data to an XML file (C#)](#_Deserializing_a_C#) below.

The class must have a public constructor without parameters.

Only public properties and fields are deserialized.

### Robust Programming

The following conditions may cause an exception:

* The class being serialized does not have a public, parameterless constructor.
* The data in the file does not represent data from the class to be deserialized.
* The file does not exist (IOException).

## Deserialize (Write) Data to an XML File

This example writes the object from a class to an XML file using the XmlSerializer class.

**Example:**

public class XMLWrite

{

static void Main(string[] args)

{

WriteXML();

}

public class Book

{

public String title;

}

public static void WriteXML()

{

Book overview = new Book();

overview.title = "Serialization Overview";

System.Xml.Serialization.XmlSerializer writer =

new System.Xml.Serialization.XmlSerializer(typeof(Book));

var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + "//SerializationOverview.xml";

System.IO.FileStream file = System.IO.File.Create(path);

writer.Serialize(file, overview);

file.Close();

}

}

### Compiling the Code

The class being serialized must have a public constructor without parameters.

### Robust Programming

The following conditions may cause an exception:

* The class being serialized does not have a public, parameterless constructor.
* The file exists and is read-only (IOException).
* The path is too long (PathTooLongException).
* The disk is full (IOException).

## A Better Example

### Serializing XML to C# Object

Let's understand how to convert an XML file into a C# object. Take note of the below small XML file to demonstrate.

<?xml version="1.0" encoding="utf-8"?>

<Company xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <Employee name="x" age="30" />

  <Employee name="y" age="32" />

</Company>

To convert this XML into an object, first you need to create a similar class structure in C#.

[XmlRoot(ElementName = "Company")]

**public** **class** Company

{

**public** Company()

    {

        Employees = **new** List<Employee>();

    }

    [XmlElement(ElementName = "Employee")]

**public** List<Employee> Employees { **get**; **set**; }

**public** Employee **this**[**string** name]

    {

**get** { **return** Employees.FirstOrDefault(s => **string**.Equals(s.Name, name, StringComparison.OrdinalIgnoreCase)); }

    }

}

**public** **class** Employee

{

    [XmlAttribute("name")]

**public** **string** Name { **get**; **set**; }

    [XmlAttribute("age")]

**public** **string** Age { **get**; **set**; }

}

Your XML and C# objects are ready. Let's see the final step of converting XML into a C# object. To do that, you need to use System.Xml.Serialization.XmlSerializer to serialize it.

**public** T DeserializeToObject<T>(**string** filepath) where T : **class**

{

    System.Xml.Serialization.XmlSerializer ser = **new** System.Xml.Serialization.XmlSerializer(**typeof**(T));

**using** (StreamReader sr = **new** StreamReader(filepath))

    {

**return** (T)ser.Deserialize(sr);

    }

}

Use the XML file path and use this function. You should see that the XML is converted into a company object with two employee objects.

### Deserializing a C# Object in XML

Create a C# object, such as a company with a few employees, and then convert it into an XML file.

var company = **new** Company();

company.Employees = **new** List<Employee>() { **new** Employee() { Name = "o", Age = "10"}};

SerializeToXml(company, xmlFilePath);

**public** **static** **void** SerializeToXml<T>(T anyobject, **string** xmlFilePath)

{

    XmlSerializer xmlSerializer = **new** XmlSerializer(anyobject.GetType());

**using** (StreamWriter writer = **new** StreamWriter(xmlFilePath))

    {

        xmlSerializer.Serialize(writer, anyobject);

    }

}

The output should look like the below text after converting it into XML.

<?xml version="1.0" encoding="UTF-8"?>

<Company xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <Employee age="10" name="o"/>

</Company>

## Another XML Example

using System;

public class clsPerson

{

public string FirstName;

public string MI;

public string LastName;

}

class class1

{

static void Main(string[] args)

{

clsPerson p=new clsPerson();

p.FirstName = "Jeff";

p.MI = "A";

p.LastName = "Price";

System.Xml.Serialization.XmlSerializer x = new System.Xml.Serialization.XmlSerializer(p.GetType());

x.Serialize(Console.Out, p);

Console.WriteLine();

Console.ReadLine();

}

}

### Verification

To verify that your project works, press CTRL+F5 to run the project. A clsPerson object is created and populated with the values that you entered. This state is serialized to XML. The console window shows the following code:

<?xml version="1.0" encoding="IBM437"?>

<clsPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<FirstName>Jeff</FirstName>

<MI>A</MI>

<LastName>Price</LastName>

</clsPerson>

## Serializing a Class that Contains a Field Returning a Complex Object

If a property or field returns a complex object (such as an array or a class instance), the XmlSerializer converts it to an element nested within the main XML document. For example, the first class in the following code example returns an instance of the second class.

public class PurchaseOrder

{

public Address MyAddress;

}

public class Address

{

public string FirstName;

}

The serialized XML output might resemble the following.

<PurchaseOrder>

<MyAddress>

<FirstName>George</FirstName>

</MyAddress>

</PurchaseOrder>

## Serializing an Array of Objects

You can also serialize a field that returns an array of objects, as shown in the following code example.

public class PurchaseOrder

{

public Item [] ItemsOrders;

}

public class Item

{

public string ItemID;

public decimal ItemPrice;

}

The serialized class instance might resemble the following, if two items are ordered.

<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<ItemsOrders>

<Item>

<ItemID>aaa111</ItemID>

<ItemPrice>34.22</ItemPrice>

</Item>

<Item>

<ItemID>bbb222</ItemID>

<ItemPrice>2.89</ItemPrice>

</Item>

</ItemsOrders>

</PurchaseOrder>

## Serializing a Class that Implements the ICollection Interface

You can create your own collection classes by implementing the ICollection interface, and use the XmlSerializer to serialize instances of these classes. Note that when a class implements the ICollection interface, only the collection contained by the class is serialized. Any public properties or fields added to the class will not be serialized. The class must include an Add method and an Item property (C# indexer) to be serialized.

using System;

using System.Collections;

using System.IO;

using System.Xml.Serialization;

public class Test {

static void Main(){

Test t = new Test();

t.SerializeCollection("coll.xml");

}

private void SerializeCollection(string filename){

Employees Emps = new Employees();

// Note that only the collection is serialized -- not the

// CollectionName or any other public property of the class.

Emps.CollectionName = "Employees";

Employee John100 = new Employee("John", "100xxx");

Emps.Add(John100);

XmlSerializer x = new XmlSerializer(typeof(Employees));

TextWriter writer = new StreamWriter(filename);

x.Serialize(writer, Emps);

}

}

public class Employees:ICollection {

public string CollectionName;

private ArrayList empArray = new ArrayList();

public Employee this[int index]{

get{return (Employee) empArray[index];}

}

public void CopyTo(Array a, int index){

empArray.CopyTo(a, index);

}

public int Count{

get{return empArray.Count;}

}

public object SyncRoot{

get{return this;}

}

public bool IsSynchronized{

get{return false;}

}

public IEnumerator GetEnumerator(){

return empArray.GetEnumerator();

}

public void Add(Employee newEmployee){

empArray.Add(newEmployee);

}

}

public class Employee {

public string EmpName;

public string EmpID;

public Employee(){}

public Employee(string empName, string empID){

EmpName = empName;

EmpID = empID;

}

}

## Purchase Order Example - Exercise

You can cut and paste the following example code into a text file renamed with a .cs file name extension. Use the C# to compile the file. Then run it using the name of the executable.

This example uses a simple scenario to demonstrate how an instance of an object is created and serialized into a file stream using the Serialize method. The XML stream is saved to a file, and the same file is then read back and reconstructed into a copy of the original object using the Deserialize method.

In this example, a class named PurchaseOrder is serialized and then deserialized. A second class named Address is also included because the public field named ShipTo must be set to an Address. Similarly, an OrderedItem class is included because an array of OrderedItem objects must be set to the OrderedItems field. Finally, a class named Test contains the code that serializes and deserializes the classes.

The CreatePO method creates the PurchaseOrder, Address, and OrderedItem class objects, and sets the public field values. The method also constructs an instance of the XmlSerializer class that is used to serialize and deserialize the PurchaseOrder. Note that the code passes the type of the class that will be serialized to the constructor. The code also creates a FileStream that is used to write the XML stream to an XML document.

The ReadPo method is a little simpler. It just creates objects to deserialize and reads out their values. As with the CreatePo method, you must first construct an XmlSerializer, passing the type of the class to be deserialized to the constructor. Also, a FileStream is required to read the XML document. To deserialize the objects, call the Deserialize method with the FileStream as an argument. The deserialized object must be cast to an object variable of type PurchaseOrder. The code then reads the values of the deserialized PurchaseOrder. Note that you can also read the PO.xml file that is created to see the actual XML output.

using System;

using System.IO;

using System.Xml;

using System.Xml.Serialization;

// The XmlRoot attribute allows you to set an alternate name

// (PurchaseOrder) for the XML element and its namespace. By

// default, the XmlSerializer uses the class name. The attribute

// also allows you to set the XML namespace for the element. Lastly,

// the attribute sets the IsNullable property, which specifies whether

// the xsi:null attribute appears if the class instance is set to

// a null reference.

[XmlRoot("PurchaseOrder", Namespace="http://www.cpandl.com",

IsNullable = false)]

public class PurchaseOrder

{

public Address ShipTo;

public string OrderDate;

// The XmlArray attribute changes the XML element name

// from the default of "OrderedItems" to "Items".

[XmlArray("Items")]

public OrderedItem[] OrderedItems;

public decimal SubTotal;

public decimal ShipCost;

public decimal TotalCost;

}

public class Address

{

// The XmlAttribute attribute instructs the XmlSerializer to serialize the

// Name field as an XML attribute instead of an XML element (XML element is

// the default behavior).

[XmlAttribute]

public string Name;

public string Line1;

// Setting the IsNullable property to false instructs the

// XmlSerializer that the XML attribute will not appear if

// the City field is set to a null reference.

[XmlElement(IsNullable = false)]

public string City;

public string State;

public string Zip;

}

public class OrderedItem

{

public string ItemName;

public string Description;

public decimal UnitPrice;

public int Quantity;

public decimal LineTotal;

// Calculate is a custom method that calculates the price per item

// and stores the value in a field.

public void Calculate()

{

LineTotal = UnitPrice \* Quantity;

}

}

public class Test

{

public static void Main()

{

// Read and write purchase orders.

Test t = new Test();

t.CreatePO("po.xml");

t.ReadPO("po.xml");

}

private void CreatePO(string filename)

{

// Creates an instance of the XmlSerializer class;

// specifies the type of object to serialize.

XmlSerializer serializer =

new XmlSerializer(typeof(PurchaseOrder));

TextWriter writer = new StreamWriter(filename);

PurchaseOrder po=new PurchaseOrder();

// Creates an address to ship and bill to.

Address billAddress = new Address();

billAddress.Name = "Teresa Atkinson";

billAddress.Line1 = "1 Main St.";

billAddress.City = "AnyTown";

billAddress.State = "WA";

billAddress.Zip = "00000";

// Sets ShipTo and BillTo to the same addressee.

po.ShipTo = billAddress;

po.OrderDate = System.DateTime.Now.ToLongDateString();

// Creates an OrderedItem.

OrderedItem i1 = new OrderedItem();

i1.ItemName = "Widget S";

i1.Description = "Small widget";

i1.UnitPrice = (decimal) 5.23;

i1.Quantity = 3;

i1.Calculate();

// Inserts the item into the array.

OrderedItem [] items = {i1};

po.OrderedItems = items;

// Calculate the total cost.

decimal subTotal = new decimal();

foreach(OrderedItem oi in items)

{

subTotal += oi.LineTotal;

}

po.SubTotal = subTotal;

po.ShipCost = (decimal) 12.51;

po.TotalCost = po.SubTotal + po.ShipCost;

// Serializes the purchase order, and closes the TextWriter.

serializer.Serialize(writer, po);

writer.Close();

}

protected void ReadPO(string filename)

{

// Creates an instance of the XmlSerializer class;

// specifies the type of object to be deserialized.

XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));

// If the XML document has been altered with unknown

// nodes or attributes, handles them with the

// UnknownNode and UnknownAttribute events.

serializer.UnknownNode+= new

XmlNodeEventHandler(serializer\_UnknownNode);

serializer.UnknownAttribute+= new

XmlAttributeEventHandler(serializer\_UnknownAttribute);

// A FileStream is needed to read the XML document.

FileStream fs = new FileStream(filename, FileMode.Open);

// Declares an object variable of the type to be deserialized.

PurchaseOrder po;

// Uses the Deserialize method to restore the object's state

// with data from the XML document. \*/

po = (PurchaseOrder) serializer.Deserialize(fs);

// Reads the order date.

Console.WriteLine ("OrderDate: " + po.OrderDate);

// Reads the shipping address.

Address shipTo = po.ShipTo;

ReadAddress(shipTo, "Ship To:");

// Reads the list of ordered items.

OrderedItem [] items = po.OrderedItems;

Console.WriteLine("Items to be shipped:");

foreach(OrderedItem oi in items)

{

Console.WriteLine("\t"+

oi.ItemName + "\t" +

oi.Description + "\t" +

oi.UnitPrice + "\t" +

oi.Quantity + "\t" +

oi.LineTotal);

}

// Reads the subtotal, shipping cost, and total cost.

Console.WriteLine(

"\n\t\t\t\t\t Subtotal\t" + po.SubTotal +

"\n\t\t\t\t\t Shipping\t" + po.ShipCost +

"\n\t\t\t\t\t Total\t\t" + po.TotalCost

);

}

protected void ReadAddress(Address a, string label)

{

// Reads the fields of the Address.

Console.WriteLine(label);

Console.Write("\t"+

a.Name +"\n\t" +

a.Line1 +"\n\t" +

a.City +"\t" +

a.State +"\n\t" +

a.Zip +"\n");

}

protected void serializer\_UnknownNode

(object sender, XmlNodeEventArgs e)

{

Console.WriteLine("Unknown Node:" + e.Name + "\t" + e.Text);

}

protected void serializer\_UnknownAttribute

(object sender, XmlAttributeEventArgs e)

{

System.Xml.XmlAttribute attr = e.Attr;

Console.WriteLine("Unknown attribute " +

attr.Name + "='" + attr.Value + "'");

}

}

The XML output might resemble the following:

<?xml version="1.0" encoding="utf-8"?>

<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.cpandl.com">

<ShipTo Name="Teresa Atkinson">

<Line1>1 Main St.</Line1>

<City>AnyTown</City>

<State>WA</State>

<Zip>00000</Zip>

</ShipTo>

<OrderDate>Wednesday, June 27, 2001</OrderDate>

<Items>

<OrderedItem>

<ItemName>Widget S</ItemName>

<Description>Small widget</Description>

<UnitPrice>5.23</UnitPrice>

<Quantity>3</Quantity>

<LineTotal>15.69</LineTotal>

</OrderedItem>

</Items>

<SubTotal>15.69</SubTotal>

<ShipCost>12.51</ShipCost>

<TotalCost>28.2</TotalCost>

</PurchaseOrder>