# Compile & Execute C# Programs Using Command-Line

You can compile the "Hello World!" program by using the command line instead of the Visual Studio IDE.

**To compile and run from a command prompt**

1. Paste the code from the preceding procedure into any text editor, and then save the file as a text file. Name the file *Hello.cs*.
2. Perform one of the following steps to open a command-prompt window:
   * In Windows 10, on the **Start** menu, search for Developer Command Prompt, and then tap or choose **Developer Command Prompt for VS 2017**.
   * In Windows 7, open the **Start** menu, expand the folder for the current version of Visual Studio, open the shortcut menu for **Visual Studio Tools**, and then choose **Developer Command Prompt for VS 2017**.

**Note**: For how to enable command-line builds from a standard Command Prompt window, check [How to Set Environment Variables for the Visual Studio Command Line](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/how-to-set-environment-variables-for-the-visual-studio-command-line).

1. In the command-prompt window, navigate to the folder that contains your *Hello.cs* file.
2. Enter the following command to compile *Hello.cs*.

$ csc Hello.cs

If your program has no compilation errors, an executable file that is named *Hello.exe* is created.

1. In the command-prompt window, enter the following command to run the program:

$ Hello

# Coding Convention

|  |  |  |
| --- | --- | --- |
| **Type** | **Convention** | **Example** |
| Class | Use PascalCase | public partial class About : Page  {  // ...  } |
| Interface | Use PascalCase  Use Prefix “I” | public interface IUser  {  // ...  } |
| Namespace | Use PascalCase  Notes:   * Should name the namespace the same as its project (same as DLL, EXE). * Should NOT name a class the same as its namespace. [Why](https://stackoverflow.com/a/18731452)? | using System.Windows.Forms; |
| Methods | Use PascalCase | File.Create(path); |
| Arguments and Local Variable | Use camelCase  Don't use Hungarian  Don't use underscore ( \_ ) | // Good  int numberOfPost = 0;  // Bad  int iNumberOfPost = 0; |
| Private member variable | Use camelCase  Use prefix with underscore ( \_ ). | private int \_salary = 100; |
| Public and Protected member variable | Use PascalCase | public int Salary = 100; |
| Property | Use PascalCase  Don’t use Get and Set as prefix | public int Salary  {  get { return \_salary; }  set { \_salary = value; }  } |
| Native data types | Always use native datatype instead of .NET CTS type | // Good  private int \_salary = 100;    // Bad  private Int16 \_salary = 100;  private String \_firstName;  private Boolean \_isSaved; |
| Implicit type | Use var for local variable declarations.  Exception: primitive types (int, string, double, etc). | var stream = File.Create(path);  var customers = new Dictionary();    // Exceptions  int index = 100;  string timeSheet;  bool isCompleted; |
|  |  |  |

**Some conventions I don’t follow:**

|  |  |  |
| --- | --- | --- |
| **Types** | **Convetions** | **Examples** |
| Constants | Don’t use Screaming Caps  **Why**: consistent with the Microsoft's .NET Framework | // Correct  public static const string ShippingType = "DropShip";    // Avoid  public static const string SHIPPING\_TYPE = "DropShip"; |
| Enum | Don’t use suffix enum names with “Enum”  **Why**: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers. | // Don't  public enum CoinEnum  {  Penny,  Nickel,  }    // Correct  public enum Coin  {  Penny,  Nickel,  } |
|  |  |  |

**Standard Abbreviation for Controls:**

|  |  |
| --- | --- |
| **Abbreviations** | **Standard Control** |
| btn | Button |
| cb | CheckBox |
| cbl | CheckBoxList |
| ddl | DropDownList |
| fu | FileUpload |
| hdn | HiddenField |
| hlk | Hyperlink |
| img | Image |
| lbl | Label |
| lbtn | LinkButton |
| mv | MultiView |
| pnl | Panel |
| txt | TextBox |
| DataGrid | dtg |
| imb | ImageButton |
| lst | ListBox |
| dtl | DataList |
| rep | Repeater |
| rdo | RadioButton |
| rdl | RadioButtonList |
| phd | Placeholder |
| tbl | Table |
| gv | GridView |
| dtv | DetailView |
| fv | FormView |

# Data Types

In C#, variables are categorized into following types:

* Value types
* Reference types
* Pointer types

Trick: The simplest test for reference type vs. value type is that reference types can be null, but value types cannot. Except for nullable value types, which are nullable (you can set the value to null, which means the null value for the type rather than a null reference).

## Value Types

Value type variables can be assigned a value directly. They are derived from the class System.ValueType.

The value types are stored in **Stack**.

The following table lists the available value types in C# 2010:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

## Reference Types

The reference types don’t contain the actual data stored in a variable, but a reference to variables. In particular, they **refer to a memory location**. If the data in the memory location is changed by one of the variables, other variables automatically reflects this change in value.

The referenced types are stored in **Heap**.

Example of built-in reference types are: object, dynamic, string, etc.

### Object Type

The object type is the **base class for all data types** in C# Common Type System (CTS). It can be assigned values of any other types: value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

The object keyword is an alias for System.Object class.

#### Boxing

When a value type is converted to object type, it is called **boxing**. Boxing is an implicit conversion.

object obj = 20; // boxing

#### Unboxing

When an object type is converted to a value type, it is called **unboxing**. Unboxing is an explicit conversion.

object obj = 20;

int i = (int)obj; // Unboxing

// or

// int i = obj as int;

### Dynamic Type

You can store any type of value in the dynamic data type.

dynamic d = 20;

The dynamic type is similar to the object type, except that type checking for object types takes place at compile time, whereas that for **the dynamic type takes place at run time**.

Read more: [When to use the dynamic type?](https://stackoverflow.com/a/2690661)

Note: The dynamic type does not support IntelliSense in visual studio

### String Type

The string type allows you to assign any string values to a variable. The string keyword is an alias for the System.String class, which is derived from object type. The value for a string type can be assigned using string literals in two forms:

string str1 = "C:\\Users\\Rich"; // regular string literal

string str2 = @" C:\Users\Rich"; // verbatim string literal

**Note**: With verbatim string literal, we don’t need to use escape characters. The compiler automatically adds appropriate escape characters for us.

### Array

In C#, System.Array is the abstract base type of all array types.

// Value type

int val = 0;

// Reference type because 10 integers are allocated on the HEAP

int[] val = new int[10];

Other ways: <https://www.dotnetperls.com/initialize-array>

### User-Defined Reference Types

They are **class**, [**interface**](#_Interfaces), or [**delegate**](#_Delegates). We will discuss these types in later chapter.

## Pointer Types

A pointer type stores the memory address of another type. Pointers in C# have the same syntax and capabilities as pointers in C or C++.

char\* cptr;

int\* iptr;

We will discuss pointer types in the chapter [Unsafe Codes](https://www.tutorialspoint.com/csharp/csharp_unsafe_codes.htm).

# var

It declares a type **based on what it’s assigned to in the initialization** (same as auto in C++ 11).

Eg:

var foo = "bar"; // implicitly typed

// same as

string foo = "bar"; // explicitly typed

var n = 2;

// same as

int n = 2;

var n = 2l;

// same as

long n = 2;

// Similarly

var i = null; // not allowed as type can't be inferred.

var j = (string)null; // allowed as “(string) null” has both type and value.

The var has only two uses:

* It requires less typing to declare variables, especially when declaring a variable as a nested generic type or class data type.
* It must be used when storing a reference to an object of an anonymous type, because the type name cannot be known in advance: var foo = new { Bar = "bar" };

**Note**: var is only used for **local variables**. **CANNOT be used to declare field/property/parameter/return types**.

# Type Casting

Type Casting (or Type Conversion) is the process of converting one type of data to another type. In C#, type casting has two forms:

* **Implicit type conversion**: These conversions are performed by C# in a type-safe manner. For example, conversions from *int* to *double*, or from derived classes to base classes.
* **Explicit type conversion**: These conversions are done explicitly by users using the pre-defined functions. For example, conversions from *double* to *int*. Explicit conversions **require a cast operator**.

The following example is explicit type conversion:

[Live Demo](http://tpcg.io/JzfLhC)

using System;

namespace TypeConversionApplication

{

class ExplicitConversion

{

static void Main(string[] args)

{

double d = 56.74;

int i;

// Cast double to int

i = (int)d;

Console.WriteLine(i);

}

}

}

Output:

56

## Built-In Type Conversion Methods

C# provides the following built-in type conversion methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| ToBoolean | Converts a type to a Boolean value, where possible. |
| ToByte | Converts a type to a byte. |
| ToChar | Converts a type to a single Unicode character, where possible. |
| ToDateTime | Converts a type (integer or string type) to date-time structures. |
| ToDecimal | Converts a floating point or integer type to a decimal type. |
| ToDouble | Converts a type to a double type. |
| ToInt16 | Converts a type to a 16-bit integer. |
| ToInt32 | Converts a type to a 32-bit integer. |
| ToInt64 | Converts a type to a 64-bit integer. |
| ToSbyte | Converts a type to a signed byte type. |
| ToSingle | Converts a type to a small floating-point number. |
| ToString | Converts a type to a string. |
| ToType | Converts a type to a specified type. |
| ToUInt16 | Converts a type to an unsigned int type. |
| ToUInt32 | Converts a type to an unsigned long type. |
| ToUInt64 | Converts a type to an unsigned big integer. |

The following example converts various value types to string type:

[Live Demo](http://tpcg.io/Wjo68S)

using System;

namespace TypeConversionApplication

{

class StringConversion

{

static void Main(string[] args)

{

int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;

Console.WriteLine(i.ToString());

Console.WriteLine(f.ToString());

Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

Console.ReadKey();

}

}

}

Output:

75

53.005

2345.7652

True

## The 'as' and 'is' Keyword

<https://www.geeksforgeeks.org/c-sharp-as-operator-keyword/>

<https://www.geeksforgeeks.org/is-vs-as-operator-keyword-in-c-sharp/>

# Functions

## Parameters: Passing by Reference

**Note:**

As mentioned in [this section](#_Reference_Types), object types (such as string, array, etc.) are always passed by reference by default. So, **we don't need to pass object types by reference** anymore.

A return statement can return only one value from a function. However, using output parameters, we can return more values from a function. In C#, we have out and ref keyword to denote output parameters.

### The 'out' Keyword

The out keyword tells the compiler that the object will be initialized inside the function. This means the value is not already set, and therefore must be set before calling. Therefore, out is **only one way, which is out.**

For example:

[Live Demo](http://tpcg.io/DFRvav)

using System;

namespace CalculatorApplication

{

class NumberManipulator

{

public void GetValue(out int x)

{

x = 5; // x must be initialized

}

static void Main(string[] args)

{

NumberManipulator n = new NumberManipulator();

int a = 10;

Console.WriteLine("Before, value of a: {0}", a);

n.GetValue(out a);

Console.WriteLine("After, value of a: {0}", a);

}

}

}

Output:

Before, value of a: 10

After, value of a: 5

### The 'ref' Keyword

The ref keyword tells the compiler that the object is initialized before entering the function. This means the value is already set, the method can therefore read it and modify it. So, ref is **two ways, both in and out.**

For example:

[Live Demo](http://tpcg.io/DFRvav)

using System;

namespace CalculatorApplication

{

class NumberManipulator

{

public void GetValue(ref int x)

{

x = 5; // optionally initialize x

}

static void Main(string[] args)

{

NumberManipulator n = new NumberManipulator();

int a = 10;

Console.WriteLine("Before, value of a: {0}", a);

n.GetValue(ref a);

Console.WriteLine("After, value of a: {0}", a);

}

}

}

Output:

Before, value of a: 10

After, value of a: 5

Better example about the difference between out and ref: <https://stackoverflow.com/a/1683423>

# Properties

In C#, properties are special methods called “accessors”, because they offer a way to get and set a field if you have a private field. So, they **enable a class to expose a public way of getting and setting values, while hiding implementation or verification code**.

## What Are They?

If you have had any dealing with C# you will likely have seen variables declared in the following format:

public class MyClass

{

public string MyProperty { get; set; }

}

The above implementation is known in C# as an **Auto-Implemented Property**. **The compiler will automatically create a private backing field that can only be accessed through the property's get and set accessors**, like this:

public class MyClass

{

private string \_myProperty;    // Also known as a 'backing field'

public string MyProperty

{

get

{

return \_myProperty;

}

set

{

\_myProperty = value; // [details about ‘value’ keyword](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value)

}

}

}

Basically speaking, it’s a shorthand single-line version of the more verbose implementation directly above. It’s typically **used when no extra logic is required** when getting or setting the value of a variable.

The values can then be accessed and modified from an external class like so:

var myClass = new MyClass();

string foo = myClass.MyProperty;    // getting

myClass.MyProperty = "John";     // setting

## Mutable vs Immutable Properties

The property in the above example is mutable; client code can change MyProperty after it’s created. In complex classes that contain significant behavior (methods) and data, it is often necessary to have public properties.

However, for small classes or structs that just encapsulate a set of data and have little or no behaviors, you should either **make the objects immutable by declaring the set accessor as private** (immutable to client code) **or by declaring only a get accessor** (immutable everywhere, except the constructor).

In summary:

// All classes can get and set the value of \_myProperty.

// Called ‘read-write’

public string MyProperty { get; set; }

// All classes can get the value of \_myProperty,

// but CANNOT set it, **even the class declaring myProperty** (except its constructor).

// Called ‘read-only’

public string MyProperty { get; }

// All classes can get \_myProperty,

// But only the class declaring \_myProperty can set it.

public string MyProperty { get; private set; }

// SYNTAX ERROR: auto-implemented properties must have get accessor.

public string MyProperty { set; }

// Correct syntax, but completely USELESS in practical.

private string MyProperty { get; set; }

// AVOID doing this

public string \_myProperty;

// In C# 6, it is possible to declare the auto-properties just as a field:

public string MyProperty { get; set; } = "John";

## Properties with Backing Fields

One basic pattern for implementing a property involves using a private backing field for setting and getting the property value. The get accessor returns the value of the private field, and the set accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

Example:

using System;

class TimePeriod

{

private double \_seconds;

public double Hours

{

get { return \_seconds / 3600; }

set {

if (value < 0 || value > 24)

throw new ArgumentOutOfRangeException(

$"{nameof(value)} must be between 0 and 24.");

\_seconds = value\*3600;

}

}

}

class Program

{

static void Main()

{

TimePeriod t = new TimePeriod();

// The property assignment causes the 'set' accessor to be called.

t.Hours = 24;

// Retrieving the property causes the 'get' accessor to be called.

Console.WriteLine($"Time in hours: {t.Hours}");

}

}

Output:

Time in hours: 24

# Indexer

An indexer allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves like a virtual array. You can then access the instance of this class using the array access operator ([ ]).

## Syntax

A one-dimensional indexer has the following syntax:

[access\_modifier] [return\_type] this [argument\_list]

{

get

{

// get block code

}

set

{

// set block code

}

}

In the above syntax:

* access\_modifier: It can be public, private, protected or internal.
* return\_type: It can be any valid C# type.
* this: It is the keyword which points to the object of the current class.
* argument\_list: This specifies the parameter list of the indexer.
* get{ } and set{ }: These are the accessors.

## Use of Indexers

The definition of an indexer is to some extent similar to a property; you use get and set accessors for defining an indexer. Their main differences are:

* Properties return or set a specific data member, whereas indexers return or set a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.
* Defining a property involves providing a property name. Indexers are not defined with names, but with the this keyword, which refers to the object instance.

The following example demonstrates the concept of indexers:

using System;

namespace IndexerApplication

{

class IndexedNames

{

private string[] namelist = new string[size];

static public int size = 5;

public IndexedNames()

{

for (int i = 0; i < size; i++) {

namelist[i] = "N.A.";

}

}

public string this[int index]

{

set

{

if (index >= 0 && index <= size - 1) {

namelist[index] = value;

}

}

get

{

if (index >= 0 && index <= size - 1) {

return namelist[index];

}

else {

return "";

}

}

}

static void Main(string[] args)

{

IndexedNames names = new IndexedNames();

names[0] = "Zara";

names[1] = "Riz";

names[2] = "Nuha";

for (int i = 0; i < IndexedNames.size; i++) {

Console.WriteLine(names[i]);

}

}

}

}

Output:

Zara

Riz

Nuha

N.A.

N.A.

## Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers:

using System;

namespace IndexerApplication

{

class IndexedNames

{

private string[] namelist = new string[size];

static public int size = 5;

public IndexedNames()

{

for (int i = 0; i < size; i++) {

namelist[i] = "N.A.";

}

}

public string this[int index]

{

set

{

if (index >= 0 && index <= size - 1) {

namelist[index] = value;

}

}

get

{

if (index >= 0 && index <= size - 1) {

return namelist[index];

}

else {

return "";

}

}

}

public int this[string name]

{

get

{

int index = 0;

while (index < size)

{

if (namelist[index] == name) {

return index;

}

index++;

}

return index;

}

}

static void Main(string[] args)

{

IndexedNames names = new IndexedNames();

names[0] = "Zara";

names[1] = "Riz";

names[2] = "Nuha";

// using the first indexer with int parameter

for (int i = 0; i < IndexedNames.size; i++) {

Console.WriteLine(names[i]);

}

// using the second indexer with the string parameter

Console.WriteLine("The index of Nuha is: {0}", names["Nuha"]);

}

}

}

Output:

Zara

Riz

Nuha

N.A.

N.A.

The index of Nuha is: 2

## Expression Body Definitions

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/>

# Constant Class

<https://softwareengineering.stackexchange.com/a/231270>

# Arrays

## Declaring Arrays

Syntax:

data-type[] array-name;

Where,

* data-type is used to specify the type of elements in the array.
* [] specifies the rank of the array. The rank specifies the size of the array.
* array-name specifies the name of the array.

For example:

double[] balance;

## Initializing Arrays

Array is a [**reference type**](#_Reference_Types), so you need to use the new keyword to create an instance of the array. For example:

double[] balance = new double[10];

C# compiler implicitly initializes each array element to a default value depending on the array type. For example, for a *double* array, all elements are initialized to 0.0D.

## Assigning Values to Arrays

You can assign values to individual array elements, by using the index number:

double[] balance = new double[10];

balance[0] = 4500.0;

You can assign values to the array at the time of declaration:

double[] balance = { 2340.0, 4523.69, 3421.0 };

You can create and initialize an array:

int[] marks = new int[5] { 99, 98, 92, 97, 95 };

You may omit the size of the array, as shown:

int[] marks = new int[] { 99, 98, 92, 97, 95 };

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location:

int[] marks = new int[] { 99, 98, 92, 97, 95 };

int[] score = marks;

## Other Array Concepts in C#

There are following few important concepts related to array which should be clear to a C# programmer:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| [Multi-dimensional arrays](https://www.tutorialspoint.com/csharp/csharp_multi_dimensional_arrays.htm) | C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| [Jagged arrays](https://www.tutorialspoint.com/csharp/csharp_jagged_arrays.htm) | C# supports multidimensional arrays, which are arrays of arrays. |
| [Passing arrays to functions](https://www.tutorialspoint.com/csharp/csharp_passing_arrays_to_functions.htm) | You can pass to the function a pointer to an array by specifying the array's name without an index. |
| [Param arrays](https://www.tutorialspoint.com/csharp/csharp_param_arrays.htm) | This is used for passing unknown number of parameters to a function. |
| [The Array Class](https://www.tutorialspoint.com/csharp/csharp_array_class.htm) | Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays. |

# Struct

A structure is a composite data type which consists of many elements of other types. In C#, structure is a **value type**. Its instances or objects are created in stack. It can contain fields, methods, constants, constructors, properties, indexers, operators and even other structure types.

## Structure Declaration & Object Creation

For example:

struct MyStruct

{

public int x;

public int y;

}

We can use an **access modifier** (private, public or internal) for structs like properties.

The objects of a struct can be created by:

// Using the new operator. An appropriate constructor is called.

MyStruct ms = new MyStruct();

// Or without using the new operator

MyStruct ms;

// But in this case all fields of the struct will remain unassigned

// and **the object can't be used until all of the fields are initialized**.

// Also, NO constructor called.

The members of a struct can be accessed by using the dot (.) operator:

ms.x = 10;

ms.y = 20;

## Structs & Fields

Fields in a struct can be declared as private, public, internal. Remember that we **cannot initialize a field inside a struct**. However, we can use constructor to initialize the structure fields (below section).

struct MyStruct

{

int x = 20;     // ERROR

}

However, a struct can contain **static fields, which can be initialized inside the struct**. For example:

using System;

struct MyStruct

{

public **static** int x = 25;

public **static** int y = 50;

}

class MyClient

{

public **static** void Main() {

int sum = MyStruct.x + MyStruct.y;

Console.WriteLine("The sum is {0}", sum);

}

}

## Structs & Constructors

A C# struct **can declare constructor, but they must take parameters**. It is an error to define a default (parameterless) constructor for a struct. Also remember that C# struct **do not support destructors**.

using System;

struct MyStruct

{

int x;

public MyStruct(int i) {

x = i;

}

public void ShowXY() {

Console.WriteLine("x = {0}", x);

}

}

class MyClient

{

public static void Main() {

MyStruct ms = new MyStruct(30);

ms.ShowXY();

}

}

## Struct & Methods

A C# struct can contain methods, either static or non-static. In case of static methods, they can access only other static members.

using System;

struct MyStruct

{

**static** int x = 25;

**static** int y = 50;

public void SetXY(int i, int j) {

x = i;

y = j;

}

public **static** void ShowSum() {

int sum = x + y;

Console.WriteLine("The sum is {0}", sum);

}

}

class MyClient

{

public **static** void Main() {

MyStruct ms = new MyStruct();

ms.SetXY(100, 200);

MyStruct.ShowSum();

}

}

## Structs & Properties

The properties can be declared inside a struct as shown below.

using System;

class MyStruct

{

private int \_x;

public int X

{

get { return \_x; }

set { \_x = value; }

}

}

class MyClient

{

public static void Main()

{

MyStruct ms = new MyStruct();

ms.X = 10;

int xVal = ms.X;

Console.WriteLine(xVal); // Displays 10

}

}

## Structs & Indexers

The indexers can be used with a C# struct. An example is shown below.

using System;

using System.Collections;

struct MyStruct

{

public string[] data;

public string this[int index]

{

get { return data[index]; }

set { data[index] = value; }

}

}

class MyClient

{

public static void Main()

{

MyStruct ms = new MyStruct();

ms.data = new string[5];

ms[0] = "Rajesh";

ms[1] = "A3-126";

ms[2] = "Snehadara";

ms[3] = "Irla";

ms[4] = "Mumbai";

Console.WriteLine("{0},{1},{2},{3},{4}", ms[0], ms[1], ms[2], ms[3], ms[4]);

}

}

## Structs & Inheritance

Since structs **doesn't support inheritance**. So, a struct cannot inherit from another struct or be the base of another struct. However, it can inherit from the base class Object. We **can't use the keywords** virtual**,** override**,** abstract**, etc. with a struct method**.

Also, the declared accessibility of a struct member cannot be protected or protected internal. Since all struct types are implicitly inherited from object class, it is possible to override the methods of the object class inside a struct by using the keyword override. Remember that this is special case in C# structs.

## Structs & Interfaces

Just like classes, a **C# struct can implement interfaces**. For example:

using System;

interface IInterface

{

void Method();

}

struct Complex : IInterface

{

public void Method() {

Console.WriteLine("Struct Method");

}

}

class MyClient

{

public static void Main() {

Complex c1 = new Complex();

c1.Method();

}

}

## Structs vs Classes

In C#, structs share many similarities to classes. But they are two entirely different aspects of the language. Following is some of the most major differences:

|  |  |
| --- | --- |
| **Struct** | **Class** |
| A value type. Inherits from System.Value Type. | A reference type. Inherits from the System.Object Type. |
| Usually used for smaller amounts of data. | Usually used for large amounts of data. |
| Can’t be inherited to other types. | Can be inherited to other class. |
| Can't be abstract. | Can be abstract type. |
| No need to create object by new keyword. | Must use new keyword. |
| Can’t create any default constructor. | Can create a default constructor. |
| Objects are always created on stack. | Objects are always created on heap. |

## Why Structs Should Be Immutable?

<https://stackoverflow.com/a/441357>

<https://stackoverflow.com/a/25172054>

# Collections

<https://www.tutorialspoint.com/csharp/csharp_collections.htm>

## ArrayList

<https://www.tutorialspoint.com/csharp/csharp_arraylist.htm>

## Dictionary

[C# Dictionary (tutorialsteacher.com)](https://www.tutorialsteacher.com/csharp/csharp-dictionary)

[C# Dictionary Examples - Dot Net Perls](https://www.dotnetperls.com/dictionary)

## Queue

<https://www.tutorialspoint.com/csharp/csharp_queue.htm>

# Preprocessors

# Delegates

In C#, **a delegate is a reference type variable that holds the reference to a method**. The reference can be changed at runtime. C# delegates are **similar to function pointers** in C/C++, but are type-safe.

Delegates are **especially used for implementing events and callback methods**. All delegates are implicitly derived from the System.Delegateclass.

## Example

**Delegate Declaration**

In the below statement, we defined a delegate named *MyDelegate*:

public delegate int MyDelegate(string s);

*MyDelegate* can refer to any method that has a single string parameter and returns an int variable. This means **any method that matches *MyDelegate*‘s signature can be assigned to *MyDelegate***.

Note: The definiton of delegates is implemented in System.Delegate. In our applications, we DON’T NEED to re-define them.

**Delegate Instantiation**

Delegate is actually a class. So, delegate objects must be created with the newkeyword and be associated with a particular method. When creating a delegate, the argument passed to the new expression is written similar to a method call, but without the arguments to the method. For example:

public delegate void printString(string s);

...

printString ps = new printString(WriteToScreen);

**Delegate Usage**

Following example demonstrates declaration, instantiation, and use of a delegate:

[Live Demo](http://tpcg.io/0MSErR)

using System;

namespace DelegateApp

{

class TestDelegate

{

// Declare a delegate

public delegate int NumberChanger(int n);

static int num = 2;

public static int AddNum(int n)

{

num += n;

return num;

}

public static int MultNum(int n)

{

num \*= n;

return num;

}

public static int GetNum()

{

return num;

}

static void Main(string[] args)

{

// Create delegate instances

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

// Calling the methods using the delegate objects

nc1(8);

Console.WriteLine("Value of Num: {0}", GetNum());

nc2(5);

Console.WriteLine("Value of Num: {0}", GetNum());

}

}

}

Output:

Value of Num: 10

Value of Num: 50

## Multicasting of Delegates

Delegate objects can be composed using the "+" operator. A composed delegate calls two or more delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this property of delegates, you can create an invocation list of methods that will be called when a delegate is invoked.

[Live Demo](http://tpcg.io/TE74JO)

using System;

delegate int NumberChanger(int n);

namespace DelegateAppl

{

class TestDelegate

{

static int num = 2;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

// Create delegate instances

NumberChanger nc;

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

// Multicast delegates

nc = nc1 + nc2;

// Call multicast

nc(5);

Console.WriteLine("Value of Num: {0}", getNum());

Console.ReadKey();

}

}

}

Output:

Value of Num: 35

## Passing Delegates as Arguments

Delegates to be passed as arguments to regular functions. The following example print the same message content defined in a function to both the console and text file.

using System;

using System.IO;

namespace DelegateApp

{

class StringPrinting

{

static FileStream fs;

static StreamWriter sw;

// Delegate declaration

public delegate void PrintString(string s);

// This method prints to the console

public static void WriteToScreen(string str)

{

Console.WriteLine("The String is: {0}", str);

}

// This method prints to a file

public static void WriteToFile(string s)

{

fs = new FileStream("c:\\message.txt",

FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

sw.WriteLine(s);

sw.Flush();

sw.Close();

fs.Close();

}

// This method takes the delegate as parameter and uses it to

// call the methods as required

public static void SendString(PrintString ps)

{

ps("Hello World");

}

static void Main(string[] args)

{

PrintString ps1 = new PrintString(WriteToScreen);

PrintString ps2 = new PrintString(WriteToFile);

// Print the same content in SendString() to both console and file.

SendString(ps1);

SendString(ps2);

}

}

}

Output:

The String is: Hello World

## Benefits of Delegates

* Multicast delegates are used extensively in **event handling** because multiple methods can be called on a single event.
* Delegates are extensively used in threading
* Delegates are also used for generic class libraries, which have generic functionality, defined.

## Anonymous Delegate Methods

You can create a delegate, but there is no need to declare the method associated with it. You do not have to explicitly define a method prior to using the delegate. Such a method is referred to as anonymous. In other words, if a delegate itself contains its method definition, it is known as **anonymous delegate method**.

You need NOT specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

For example:

using System;

delegate void NumberChanger(int n);

namespace DelegateApp

{

class TestDelegate

{

static int num = 10;

public static void AddNum(int p)

{

num += p;

Console.WriteLine("Named Method: {0}", num);

}

static void Main(string[] args)

{

//create delegate instances using anonymous method

NumberChanger nc = delegate(int x) {

Console.WriteLine("Anonymous Method: {0}", x);

};

// Call the delegate using the anonymous method

nc(10);

// Instantiate the delegate using the named methods

nc = new NumberChanger(AddNum);

// Call the delegate using the named methods

nc(5);

}

}

}

Output:

Anonymous Method: 10

Named Method: 15

# Callback

<https://www.brainbell.com/tutors/C_Sharp/Using_Callback_Functions_with_C.htm>

<https://www.c-sharpcorner.com/UploadFile/1c8574/delegate-used-for-callback-operation/>

<https://docs.microsoft.com/en-us/dotnet/framework/interop/how-to-implement-callback-functions>

# Events

## What Are Events?

Events are **user actions** such as key presses, mouse clicks, screen touches, mouse movements, etc., or some **occurrence** such as system generated notifications. Applications need to respond to events when they occur.

Events play an important part in user interfaces and programming notifications. When something happens in one class or one part of the code, and other part of the code needs a notification, events are used.

## Delegates and Events

Events and delegate work together. **An event is a reference to a delegate**, i.e. when an event is raised, a delegate is called. In C# terms, events are a special form of delegates.

A C# event is a class member that is activated whenever the event it was designed for occurs. It starts with a class that declares an event. Any class, including the same class that the event is declared in, may register one of its methods for the event. This occurs through a delegate, which specifies the signature of the method that is registered for the event. The event keyword is a delegate modifier. It must always be used in connection with a delegate.

**A delegate can be a pre-defined .NET delegate, or one you declare yourself**. Whichever is suitable, you **assign the delegate to the event**, which effectively registers the method that will be called when the event fires.

## How to Use Events in C#?

### Declaring Events

To declare an event, first must declare a delegate type for the event. For example:

public delegate string MyDelegate(string str);

Next, the event itself is declared, using the event keyword:

public event MyDelegate MyEvent;

### Instantiating Events

Once an event is declared, it must be associated with one or more event handlers before it can be raised. An event handler is nothing but a method that is called using a delegate. Use the += operator to associate an event with an instance of a delegate that already exists.

Example:

MyEvent += new MyDelegate(WelcomeUser);

An event has the value null if it has no registered listeners.

## Examples

EX1: Program for creating a custom **singlecast** delegate and event:

using System;

namespace SampleApp

{

class EventProgram

{

public delegate string MyDelegate(string str);

public event MyDelegate MyEvent;

public EventProgram()

{

MyEvent += new MyDelegate(WelcomeUser);

}

public string WelcomeUser(string username)

{

return "Welcome " + username;

}

static void Main(string[] args)

{

EventProgram obj = new EventProgram();

string result = obj.MyEvent("Tutorials Point");

Console.WriteLine(result);

}

}

}

Output:

Welcome Tutorials Point

EX2: Program for creating custom a **multiplecast** delegate and event:

using System;

namespace SampleApp

{

class EventProgram

{

public delegate void MyDelegate(int a, int b);

public event MyDelegate MyEvent;

public void RaiseEvent(int a, int b)

{

MyEvent(a, b);

Console.WriteLine("Event Raised");

}

public void Add(int x, int y)

{

Console.WriteLine("Add Method {0}", x + y);

}

public void Subtract(int x, int y)

{

Console.WriteLine("Subtract Method {0}", x - y);

}

static void Main(string[] args)

{

EventProgram obj = new EventProgram();

obj.MyEvent += new MyDelegate(obj.Add);

obj.MyEvent += new MyDelegate(obj.Subtract);

obj.RaiseEvent(20, 10);

}

}

}

Output:

Add Method 30

Subtract Method 10

Event Raised

Must read more:

<https://www.tutorialsteacher.com/csharp/csharp-event>

<https://docs.microsoft.com/en-us/dotnet/standard/events/>

# OOP

## Inheritance

## Encapsulation

Encapsulation, in OOP, prevents access to implementation details.

Abstraction and encapsulation are related features in OOP. Abstraction allows making relevant information visible and encapsulation enables a programmer to implement the desired level of abstraction.

Encapsulation is implemented by using **access specifiers** (or **access modifiers**) which are used to define the scope and visibility of a class member. C# supports the following access specifiers:

* Public
* Private
* Protected
* Internal
* Protected internal

### Internal

The internal access specifier allows the types or members can be accessed by any code in the same assembly, but not from another assembly.

It is useful when you want to declare a member or type inside a DLL, not outside that. Normally, when you declare a member as public you can access that from other DLLs.

### Protected Internal

The protected internal access specifier allows the types or members can be accessed by any code in the assembly in which it is declared, OR from within a derived class in another assembly. Access from another assembly must take place within a class declaration that derives from the class in which the protected internal element is declared, and it must take place through an instance of the derived class type.

Note: protected internal means "protected OR internal" (any class in the same assembly, or any derived class - even if it is in a different assembly).

## Abstract Classes

### Declaration and Implementation

// Abstract class declaration

abstract class Log

{

// Note: access modifiers are allowed in abstract classes.

// With ‘abstract’, no implement here. So derived classes must override this method.

public abstract void PrintLog(string message);

protected int \_logCount; // abstract classes can have fields

// Without ‘abstract’, must implement here. So derived classes can’t override.

public int GetLogCount()

{

return \_logCount; // methods in abstract classes can have implementaton

}

// With ‘virtual’, must implement here.

// Derived classes are allowed to choose to whether override it or not.

public virtual void DeleteLog() { }

}

Now, different classes can implement Log by providing an implementation of the PrintLog() method. For example:

// Implement abstract class

class ConsoleLog : Log

{

public override void PrintLog(string message)

{

Console.WriteLine(message);

}

public override void DeleteLog()

{

Console.WriteLine("Log is deleted");

}

}

Now, you can instantiate an object of either the ConsoleLog class and use its methods:

// Instantiate object

static void Main(string[] args)

{

Log log = new ConsoleLog();

log.PrintLog("abc");

log.DeleteLog();

}

// NOTE: If ‘ConsoleLog log = new ConsoleLog()’, we still can ‘log.GetLogCount()’.

// Same for ‘log.DeleteLog()’ even when there is no ‘public override void DeleteLog()’

### Keep in Mind

* Unlike interfaces, abstract classes **can implement non-abstract methods**. So they can have fields, properties, constants, etc.
* Unlike interface, abstract classes **require access modifiers** for each of their members.
* Unlike interfaces, abstract class **cannot support multiple inheritance**:

interface ILog { }

class Log : ILog { }

abstract class ConsoleLog : Log { } // We can't inherit like this

* Abstract classes **can inherit from one or more classes and interfaces**.

abstract class ConsoleLog : ILog, Log { }

* It is a **syntax error** to use the virtual or abstract modifier on a static property or private method.

## Interfaces

An interface in C# contains **only the declaration** of the methods, properties, and events, but NOT the implementation which is left to the class that inherit the interface. Interface makes it easy to maintain a program.

### Declaration and Implicit Implementation

// Interface declaration

interface ILog

{

void PrintLog(string message); // Note: No access modifier here

}

Now, different classes can implement ILog by providing an implementation of the PrintLog() method. For example:

// Implement interface

class ConsoleLog : ILog // Note: No access modifier here

{

public void PrintLog(string message)

{

Console.WriteLine(message);

}

}

Now, you can instantiate an object of the ConsoleLog class and use its methods:

// Instantiate object

ILog log = new ConsoleLog();

log.PrintLog("abc");

// Or

ConsoleLog log = new ConsoleLog();

log.PrintLog("abc");

### Explicit Implementation

You can implement interface explicitly by prefixing interface name with method name, as below:

// Implement interface explicitly

class ConsoleLog: ILog

{

// Note: Unlike implicit implemenation, NO access modifer here.

void ILog.PrintLog(string message)

{

Console.WriteLine(message);

}

}

Now, you can instantiate an object of the ConsoleLog class and use it methods:

// Instantiate object

ConsoleLog log = new ConsoleLog();

log.PrintLog("abc");

// Note: If use ‘ILog log = new ConsoleLog()’, CANNOT use ‘log.PrintLog("abc")’

Explicit implementation is **useful when class is implementing multiple interface; thereby, it is more readable and eliminates the confusion**. It is also useful if interfaces have same method name coincidently.

### Keep in Mind

* Interfaces **cannot have *private* members**. By default, all the members of interface are public and abstract.
* Interface **cannot contain fields** because they represent a particular implementation of data.
* Interface **cannot initialize** a property. That means always “{get; set;}”. You should use abstract class instead.
* **Multiple inheritance** is possible with the help of Interfaces but not with classes.

## Partial Classes/Methods – Sealed Classed/Methods

<https://www.geeksforgeeks.org/partial-classes-in-c-sharp/>

<https://www.geeksforgeeks.org/partial-methods-in-c-sharp/>

<https://www.geeksforgeeks.org/c-sharp-sealed-class/>

# Generics

<https://www.tutorialspoint.com/csharp/csharp_generics.htm>

# Memory Management in C# (.NET)

## Managed vs Unmanaged Resources?

**Managed resources** are created and managed under scope of CLR and by runtime. Anything that lies **within .NET libraries**, including all .NET framework classes (e.g. string, int, bool variables), are referred to as managed code. Managed resources are directly **under the control of the garbage collector**.

**Unmanaged resources** are created **outside .NET libraries** and NOT managed by CLR. Examples of unmanaged code are COM objects, file streams, pointers, registries, connection objects, Interop objects. (basically, third party libraries that are referred in .NET code). Unmanaged resources are NOT directly **under the control of the garbage collector**.

## The Garbage Collector

The garbage collector knows about all managed resources; at some point in time, it will come along and clean up all the memory occupied by managed resources.

However, although the GC knows the lifetime of object which uses unmanaged objects, it doesn’t know how to release them (e.g. if SQL connection is open at the time of destroying object, then GC doesn't know how to close the connection. It simply removes the object from the HEAP). As a result, **we have to release unmanaged resources explicitly**when we finished using them. Or else, we will end up with memory leaks and locked resources.

## Cleaning Up Unmanaged Resources

There are different ways to clean up unmanaged resources:

### 1. Implementing *Dispose()* Method– The Dispose Pattern

There is a couple of ways to implement Dispose() using the Dispose Pattern:

**a) Use *SafeHandle* class**

It is only useful when you deal with **Win32** Interop calls or any unamanaged resource represented as IntPtr. In Win32, most things are represented by "handles", including Windows, Mutexes, etc. The .NET SafeHandle uses the disposable pattern to ensure these Win32 handles are properly closed. If you can't find a SafeHandle that is suitable for releasing your unmanaged resource, consider making your own SafeHandle-like class inheriting from [CriticalFinalizerObject](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.constrainedexecution.criticalfinalizerobject?view=netframework-4.8).

Using this approach, you implement the IDisposable.Dispose() and an additional Dispose(Boolean). This is recommended because it doesn't require overriding the Object.Finalize(). Why?

🡪 Because Object.Finalize() is handled asynchronously, so unmanaged resources might not be freed in a timely fashion. This is especially important for large and expensive unmanaged resources (e.g. bitmaps or database connections). Using SafeHandle class, resources are reclaimed as soon as Dispose() is called and the object doesn’t have to be queued for finalization. You want almost all of your finalizable objects being disposed and not finalized.

Exampe 1 – For the base class:

using Microsoft.Win32.SafeHandles;

using System;

using System.Runtime.InteropServices;

namespace SampleApp

{

class BaseClass : IDisposable

{

// Flag to check if object is already disposed

bool isDisposed = false;

// Create an object of SafeHandle class. E.g., SafeFileHandle

SafeHandle safeHandle = new SafeFileHandle(IntPtr.Zero, true);

// Dispose method (public) - called by the client

public void Dispose()

{

Dispose(true);

// Prevent the GC from running the finalizer (Object.Finalize())

// because managed and unmanaged resources have been explicitly released.

GC.SuppressFinalize(this);

}

// Dispose method (protected)

// isFromDisposeMethod: true if calling comes from a Dispose method

// isFromDisposeMethod: false if calling from a finalizer

protected virtual void Dispose(bool isFromDisposeMethod)

{

if (isDisposed) return;

if (isFromDisposeMethod)

{

safeHandle.Dispose();

// Free any other managed objects here.

}

// Free any unmanaged objects here.

isDisposed = true;

}

}

}

Exampe 2 – For the derived class:

using Microsoft.Win32.SafeHandles;

using System;

using System.Runtime.InteropServices;

namespace SampleApp

{

class DerivedClass : BaseClass

{

// Flag to check if object is already disposed

bool isDisposed = false;

// Create an object of SafeHandle class. E.g., SafeFileHandle

SafeHandle safeHandle = new SafeFileHandle(IntPtr.Zero, true);

// Dispose method (protected)

// isFromDisposeMethod: true if calling comes from a Dispose method

// isFromDisposeMethod: false if calling from a finalizer

protected virtual void Dispose(bool isFromDisposeMethod)

{

if (isDisposed) return;

if (isFromDisposeMethod)

{

safeHandle.Dispose();

// Free any other managed objects here.

}

// Free any unmanaged objects here.

isDisposed = true;

// Call base class implementation.

base.Dispose(isFromDisposeMethod);

}

}

}

Note: For detailed example about implementing Dispose using SafeHandle class, check [here](https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose#SafeHandles).

**b) Override *Object.Finalize* method**

You implement the IDisposable.Dispose() and an additional Dispose(Boolean). You also override the Object.Finalize() to ensure that unmanaged resources are disposed if your IDisposable.Dispose() is not called by a client of your type.

When the GC finds that that object has not been referenced for a long time, it calls Object.Finalize() just before it destroys the object completely. **In C#, the finalize method is called automatically in destructors**.

Example 1 – For the base class:

using System;

namespace SampleApp

{

class BaseClass : IDisposable

{

// Flag to check if object is already disposed

bool isDisposed = false;

// Dispose method (public) - called by the client

public void Dispose()

{

Dispose(true);

// Prevent the GC from running the finalizer (Object.Finalize())

// because managed and unmanaged resources have been explicitly released.

GC.SuppressFinalize(this);

}

// Dispose method (protected)

// isFromDisposeMethod: true if calling comes from a Dispose method

// isFromDisposeMethod: false if calling from a finalizer

protected virtual void Dispose(bool isFromDisposeMethod)

{

if (isDisposed) return;

if (isFromDisposeMethod)

{

// Free any other managed objects here.

}

// Free any unmanaged objects here.

isDisposed = true;

}

~BaseClass()

{

// In C#, override Object.Finalize() by defining a destructor

Dispose(false);

}

}

}

Notes:

* As said, the destructor of BaseClass class implicitly calls Object.Finalize(). Therefore, the previous destructor code is implicitly translated to below code:

protected override void Finalize()

{

try

{

// Cleanup statements...

}

finally

{

base.Finalize();

}

}

* If you reference any static variables or methods in finalize-time Dispose code, make sure you check the Environment.HasShutdownStarted property. If your object is thread-safe, be sure to take whatever locks are necessary for cleanup.

protected virtual void Dispose(bool isFromDisposeMethod)

{

if (isFromDisposeMethod)

{

// dispose-time code

}

// finalize-time code

CloseHandle();

if (!Environment.HasShutDownStarted)

{

// Debug.Write or Trace.Write – static methods

}

isDisposed = true;

}

Example 2 – For the derived class:

using System;

namespace SampleApp

{

class DerivedClass : BaseClass

{

// Flag to check if object is already disposed

bool isDisposed = false;

// Dispose method (protected)

// isFromDisposeMethod: true if calling comes from a Dispose method

// isFromDisposeMethod: false if calling from a finalizer

protected virtual void Dispose(bool isFromDisposeMethod)

{

if (isDisposed) return;

if (isFromDisposeMethod)

{

// Free any other managed objects here.

}

// Free any unmanaged objects here.

isDisposed = true;

// Call base class implementation.

base.Dispose(isFromDisposeMethod);

}

~DerivedClass()

{

// In C#, override Object.Finalize by defining a destructor

Dispose(false);

}

}

}

### 2. Use 'using' Statement

C# provides a special using statement to call IDisposable.Dispose() explicitly. In which, we instantiate an object in the statement. At the end of the statement block, the Dispose() is automatically called.

This statement provides some unique features:

* Manage scope: It manages the scope of the object.
* Instantiate the object as read-only: Object instantiates in the statement are read-only and you cannot modify or reassigned the object. This feature ensures Dispose() is called on the object which is instantiated.
* Ensures disposal of objects: The statement ensures Dispose() will always be called whether any exception was occurred or not.

**Example**

using(SqlHelper sqlHelper = new SqlHelper())

{

// use sqlHelper object

} // automatically calls Dispose method

Notes:

* sqlHelper object is not available outside the using statement.
* You can only use object in using statement which implements the IDisposable interface.

Here is how the statement works:

It converts your code block to try… finally block internally:

SqlHelper sqlHelper = new SqlHelper();

try

{

// use sqlHelper object

}

finally

{

if (sqlHelper != null)

{

((IDisposable)sqlHelper).Dispose();

}

}

**Nested “using” Statements**

You can use nested using statements. For example:

using (Customer customer = new Customer())

{

using (Order order = customer.Order)

{

Console.WriteLine(order.ItemName);

}

}

## Keep In Mind

**1.** **Removing reference doesn’t mean removing memory**: When client is destroying the object, it doesn't have the option of releasing the resources the object is currently holding. When client assigns null to an object, it’s only removing reference of the object from the STACK. Object is NOT removed from the HEAP.

**2. Don’t implement Finalize()** **unless required**: Implementing a finalizer on classes that do not require it adds load to the finalizer thread and the GC, as well as affects the performance of the application. That’s because the Dispose() is fast and frees the object instantly, which makes it not affect the performance cost. By contrast, the Finalize() is slower and does not free the object instantly.

**Use a finalizer only on objects that hold unmanaged resources across client calls**. For example, if your object has only one method named GetData() that opens a connection, fetches data from an unmanaged resource, closes the connection, and returns data, then there is no need to implement a finalizer. However, if your object also exposes an Open() method in which a connection to an unmanaged resource is made, and then data is fetched using a separate GetData() method, it is possible for the connection to be maintained to the unmanaged resource across calls. In this case, you should provide a Finalize() method to clean up the connection to the unmanaged resource, and in addition use the Dispose Pattern to give the client the ability to explicitly release the resource after it is finished.

**3. Move the finalization burden to the leaves of object graphs**: If you have an object graph with an object referencing other objects (leaves) that hold unmanaged resources, you should implement the finalizers in the leaf objects instead of in the root object. Moving the finalization burden to leaf objects results in the promotion of only the relevant ones to the finalization queue, which helps optimize the finalization process.

**4. If you implement Finalize(), implement Dipose()**: In this way, the calling code has an explicit way to free resources by calling the Dispose() method. You should still implement a finalizer along with Dispose because you cannot assume that the calling code always calls Dispose(). Even though this is a costly method, the finalizer implementation ensures that resources are released.

**5. If you implement Finalize() and Dispose(), use the Dispose Pattern**

**6. Suppress finalization in your Dispose() method**: If the calling code calls Dispose(), you don’t want the GC to call Finalize()because the unmanaged resources will have already been returned to the operating system. You must prevent the GC from calling the finalizer by using GC.SuppressFinalization() in your Dispose().

**7. Allow Dispose() to be called multiple times**: Calling code should be able to safely call Dispose() multiple times without causing exceptions. After the first call, subsequent calls should do nothing and not throw an ObjectDisposedException for subsequent calls.

**8. Keep finalizer code simple to prevent blocking**: Finalizer code should be simple and minimal. The finalization happens on a dedicated, single finalizer thread. Do not issue calls that could block the calling thread. If the finalizer does block, resources are not freed and the application leaks memory. Also, do not use thread local storage or any other technique that requires thread affinity because the finalizer method is called by a dedicated thread, separate from your application’s main thread.

**9. Provide thread safe cleanup code only if your type is thread safe**: If your type is thread safe, make sure your cleanup code is also thread safe. For example, if your thread safe type provides both Close() and Dispose() methods to clean up resources, ensure you synchronize threads calling Close() and Dispose() simultaneously.

# Memory Leaks in .NET Applications

[Find, Fix, and Avoid Memory Leaks in C# .NET](https://michaelscodingspot.com/find-fix-and-avoid-memory-leaks-in-c-net-8-best-practices/)

[Best Practices No 5: Detecting .NET application memory leaks](https://www.c-sharpcorner.com/UploadFile/shivprasadk/best-practices-no-5-detecting-net-application-memory-leaks/)

Tools

* [Windbg](https://snede.net/hunting-net-memory-leaks-with-windbg/)
* [.NET Memory Profiler Extension](https://marketplace.visualstudio.com/items?itemName=SciTechSoftware.NETMemoryProfiler)
* [Deleaker](https://www.deleaker.com/)

# Lambda Expression

# LINQ

<https://csharp.net-tutorials.com/linq/introduction/>

# Reflection

<https://stackoverflow.com/questions/1458256/why-is-the-use-of-reflection-in-net-recommended>

<https://stackify.com/what-is-c-reflection/>

<https://www.codeproject.com/Articles/3737/TraceListeners-and-Reflection>

# DLLs

1. [How to create and use DLL (Class Library)?](https://www.c-sharpcorner.com/UploadFile/1e050f/creating-and-using-dll-class-library-in-C-Sharp/)

2. [How to set build order of projects in solution?](https://stackoverflow.com/a/12912167)

3. [How to share code between projects/solutions without copying the code](https://stackoverflow.com/a/3382768)? (Common files)