**C# Class and Object**

In this tutorial, you will learn about the concept of classes and objects in C# with the help of examples.

C# is an object-oriented program. In object-oriented programming(OOP), we solve complex problems by dividing them into objects.

To work with objects, we need to perform the following activities:

* create a class
* create objects from the class

**C# Class**

Before we learn about objects, we need to understand the working of classes. Class is the blueprint for the object.

We can think of the class as a **sketch (prototype) of a house**. It contains all the details about the floors, doors, windows, etc. We can build a house based on these descriptions. **House** is the object.

Like many houses can be made from the sketch, we can create many objects from a class.

**Create a class in C#**

We use the class keyword to create an object. For example,

class ClassName {

}

Here, we have created a class named ClassName. A class can contain

* **fields** - variables to store data
* **methods** - functions to perform specific tasks

Let's see an example,

class Dog {

//field

string breed;

//method

public void bark() {

}

}

In the above example,

* Dog - class name
* breed - field
* bark() - method

**Note**: In C#, fields and methods inside a class are called members of a class.

**C# Objects**

An object is an instance of a class. Suppose, we have a class Dog. Bulldog, German Shepherd, Pug are objects of the class.

**Creating an Object of a class**

In C#, here's how we create an object of the class.

ClassName obj = new ClassName();

Here, we have used the new keyword to create an object of the class. And, obj is the name of the object. Now, let us create an object from the Dog class.

Dog bullDog = new Dog();

Now, the bullDog object can access the fields and methods of the Dog class.

**Access Class Members using Object**

We use the name of objects along with the . operator to access members of a class. For example,

using System;

namespace ClassObject {

class Dog {

string breed;

public void bark() {

Console.WriteLine("Bark Bark !!");

}

static void Main(string[] args) {

// create Dog object

Dog bullDog = new Dog();

// access breed of the Dog

bullDog.breed = "Bull Dog";

Console.WriteLine(bullDog.breed);

// access method of the Dog

bullDog.bark();

Console.ReadLine();

}

}

}

**Output**

Bull Dog

Bark Bark !!

In the above program, we have created an object named bullDog from the Dog class. Notice that we have used the object name and the . (dot operator) to access the breed field

// access breed of the Dog

bullDog.breed = "Bull Dog";

and the bark() method

// access method of the Dog

bullDog.bark();

**Creating Multiple Objects of a Class**

We can create multiple objects from the same class. For example,

using System;

namespace ClassObject {

class Employee {

string department;

static void Main(string[] args) {

// create Employee object

Employee sheeran = new Employee();

// set department for sheeran

sheeran.department = "Development";

Console.WriteLine("Sheeran: " + sheeran.department);

// create second object of Employee

Employee taylor = new Employee();

// set department for taylor

taylor.department = "Content Writing";

Console.WriteLine("Taylor: " + taylor.department);

Console.ReadLine();

}

}

}

**Output**

Sheeran: Development

Taylor: Content Writing

In the above example, we have created two objects: sheeran and taylor from the Employee class.

Here, you can see both the objects have their own version of the department field with different values.

**Creating objects in a different class**

In C#, we can also create an object of a class in another class. For example,

For example,

using System;

namespace ClassObject {

class Employee {

public string name;

public void work(string work) {

Console.WriteLine("Work: " + work);

}

}

class EmployeeDrive {

static void Main(string[] args) {

// create Employee object

Employee e1= new Employee();

Console.WriteLine("Employee 1");

// set name of the Employee

e1.name="Gloria";

Console.WriteLine("Name: " + e1.name);

//call method of the Employee

e1.work("Coding");

Console.ReadLine();

}

}

}

**Output**

Employee 1

Name: Gloria

Work: Coding

In the above example, we have two classes: Employee and EmployeeDrive. Here, we are creating an object e1 of the Employee class in the EmployeeDrive class.

We have used the e1 object to access the members of the Employee class from EmployeeDrive. This is possible because the members in the Employee class are public.

Here, public is an access specifier that means the class members are accessible from any other classes. To learn more, visit [C# Access Modifiers](https://www.programiz.com/csharp-programming/access-modifiers).

**Why Objects and Classes?**

Objects and classes help us to divide a large project into smaller sub-problems.

Suppose you want to create a game that has hundreds of enemies and each of them has fields like health, ammo, and methods like shoot() and run().

With OOP we can create a single Enemy class with required fields and methods. Then, we can create multiple enemy objects from it.

Each of the enemy objects will have its own version of health and ammo fields. And, they can use the common shoot() and run() methods.

Now, instead of thinking of projects in terms of variables and methods, we can think of them in terms of objects.

This helps to manage complexity as well as make our code reusable.

# C# Method

In this tutorial, we will learn about the C# method with the help of examples.

A method is a block of code that performs a specific task. Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

* a method to draw the circle
* a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

## Declaring a Method in C#

Here's the syntax to declare a method in C#.

returnType methodName() {

// method body

}

Here,

* **returnType** - It specifies what type of value a method returns. For example, if a method has an int return type then it returns an int value.

If the method does not return a value, its return type is void.

* **methodName** - It is an identifier that is used to refer to the particular method in a program.
* **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }

Let's see an example,

void display() {

// code

}

Here, the name of the method is display(). And, the return type is void.

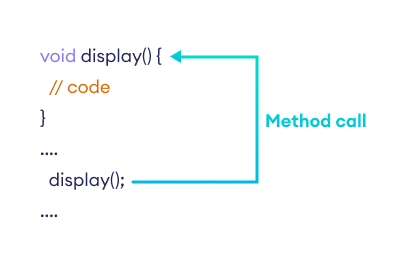
## Calling a Method in C#

In the above example, we have declared a method named display(). Now, to use the method, we need to call it.

Here's how we can call the display() method.

// calls the method

display();

Working of C# method call

### Example: C# Method

using System;

namespace Method {

class Program {

// method declaration

public void display() {

Console.WriteLine("Hello World");

}

static void Main(string[] args) {

// create class object

Program p1 = new Program();

//call method

p1.display();

Console.ReadLine();

}

}

}

**Output**

Hello World

In the above example, we have created a method named display(). We have created an object p1 of the Program class.

Notice the line,

p1.display();

Here, we are using the object to call the display() method.

## C# Method Return Type

A C# method may or may not return a value. If the method doesn't return any value, we use the void keyword (shown in the above example).

If the method returns any value, we use the return statement to return any value. For example,

int addNumbers() {

...

return sum;

}

Here, we are returning the variable sum. One thing you should always remember is that the return type of the method and the returned value should be of the same type.

In our code, the return type is int. Hence, the data type of sum should be of int as well.

### Example: Method Return Type

using System;

namespace Method {

class Program {

// method declaration

static int addNumbers() {

int sum = 5 + 14;

return sum;

}

static void Main(string[] args) {

// call method

int sum = addNumbers();

Console.WriteLine(sum);

Console.ReadLine();

}

}

}

**Output**

19

In the above example, we have a method named addNumbers() with the int return type.

int sum = addNumbers();

Here, we are storing the returned value from the addNumbers() to sum. We have used int data type to store the value because the method returns an int value.

**Note**: As the method is static we do not create a class object before calling the method. The static method belongs to the class rather than the object of a class.

## C# Methods Parameters

In C#, we can also create a method that accepts some value. These values are called method parameters. For example,

int addNumber(int a, int b) {

//code

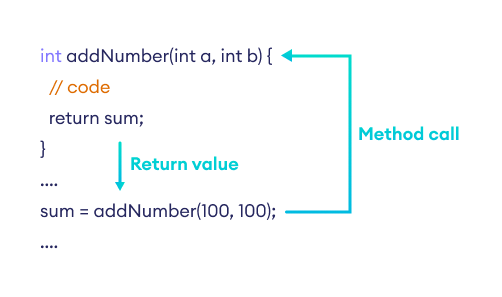
}

Here, a and b are two parameters passed to the addNumber() function.

If a method is created with parameters, we need to pass the corresponding values(arguments) while calling the method. For example,

// call the method

addNumber(100, 100);

Representation of the C# method returning a value

Here, We have passed 2 arguments (100, 100).

### Example 1: C# Methods with Parameters

using System;

namespace Method {

class Program {

int addNumber (int a, int b) {

int sum = a + b;

return sum;

}

static void Main(string[] args) {

// create class object

Program p1 = new Program();

//call method

int sum = p1.addNumber(100,100);

Console.WriteLine("Sum: " + sum);

Console.ReadLine();

}

}

}

**Output**

Sum: 200

### C# Methods with Single Parameter

In C#, we can also create a method with a single parameter. For example,

using System;

namespace Method {

class Program {

string work(string work) {

return work;

}

static void Main(string[] args) {

// create class object

Program p1 = new Program();

//call method

string work = p1.work("Cleaning"); ;

Console.WriteLine("Work: " + work);

Console.ReadLine();

}

}

}

**Output**

Work: Cleaning

Here, the work() method has a single parameter work.

## Built-in methods

So far we have defined our own methods. These are called **user-defined methods**.

However, in C#, there are various methods that can be directly used in our program. They are called **built-in methods**. For example,

* Sqrt() - computes the square root of a number
* ToUpper() - converts a string to uppercase

### Example: Math.Sqrt() Method

using System;

namespace Method {

class Program {

static void Main(string[] args) {

// Built in method

double a = Math.Sqrt(9);

Console.WriteLine("Square root of 9: " + a);

}

}

}

**Output**

Square root of 9: 3

In the above program, we have used

double a = Math.Sqrt(9);

to compute the square root of 9. Here, the Sqrt() is a built-in method that is defined inside the Math class.

We can simply use built-in methods in our program without writing the method definition. To learn more, visit C# built-in methods.

## Method Overloading in C#

In C#, we can create two or more methods with the same name. It is known as method overloading. For example,

using System;

namespace MethodOverload {

class Program {

// method with one parameter

void display(int a) {

Console.WriteLine("Arguments: " + a);

}

// method with two parameters

void display(int a, int b) {

Console.WriteLine("Arguments: " + a + " and " + b);

}

static void Main(string[] args) {

Program p1 = new Program();

p1.display(100);

p1.display(100, 200);

Console.ReadLine();

}

}

}

**Output**

Arguments: 100

Arguments: 100 and 200

In the above example, we have overloaded the display() method. It is possible because:

* one method has one parameter
* another has two parameter

# C# Access Modifiers

In this tutorial, we will learn about the public, private, protected, and internal access modifiers in C# with the help of examples.

In C#, access modifiers specify the accessibility of types (classes, interfaces, etc) and type members (fields, methods, etc). For example,

class Student {

public string name;

private int num;

}

Here,

* name - public field that can be accessed from anywhere
* num - private field can only be accessed within the Student class

**Types of Access Modifiers**

In C#, there are 4 basic types of access modifiers.

* public
* private
* protected
* internal

## 1. public access modifier

When we declare a type or type member public, it can be accessed from anywhere. For example,

using System;

namespace MyApplication {

class Student {

public string name = "Sheeran";

public void print() {

Console.WriteLine("Hello from Student class");

}

}

class Program {

static void Main(string[] args) {

// creating object of Student class

Student student1 = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Console.ReadLine();

}

}

}

**Output**

Name: Sheeran

Hello from Student class

In the above example, we have created a class named Student with a field name and a method print().

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Since the field and method are public, we are able to access them from the Program class.

**Note**: We have used the object student1 of the Student class to access its members. To learn more, visit the C# class and objects.

## 2. private access modifier

When we declare a type member with the private access modifier, it can only be accessed within the same class or struct. For example,

using System;

namespace MyApplication {

class Student {

private string name = "Sheeran";

private void print() {

Console.WriteLine("Hello from Student class");

}

}

class Program {

static void Main(string[] args) {

// creating object of Student class

Student student1 = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Console.ReadLine();

}

}

}

In the above example, we have created a class named Student with a field name and a method print().

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Since the field and method are private, we are not able to access them from the Program class. Here, the code will generate the following error.

**Error** CS0122 'Student.name' is inaccessible due to its protection level

**Error** CS0122 'Student.print()' is inaccessible due to its protection level

## 3. protected access modifier

When we declare a type member as protected, it can only be accessed from the same class and its derived classes. For example,

using System;

namespace MyApplication {

class Student {

protected string name = "Sheeran";

}

class Program {

static void Main(string[] args) {

// creating object of student class

Student student1 = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

Console.ReadLine();

}

}

}

In the above example, we have created a class named Student with a field name. Since the field is protected, we are not able to access it from the Program class.

Here, the code will generate the following error.

**Error** CS0122 'Student.name' is inaccessible due to its protection level

Now, let's try to access the protected member from a derived class.

using System;

namespace MyApplication {

class Student {

protected string name = "Sheeran";

}

// derived class

class Program : Student {

static void Main(string[] args) {

// creating object of derived class

Program program = new Program();

// accessing name field and printing it

Console.WriteLine("Name: " + program.name);

Console.ReadLine();

}

}

}

**Output**

Name: Sheeran

In the above example, we have created a class Student with a protected field name. Notice that we have inherited the Program class from the Student class.

// accessing name field and printing it

Console.WriteLine("Name: " + program.name);

Since the protected member can be accessed from derived classes, we are able to access name from the Program class.

## 4. internal access modifier

When we declare a type or type member as internal, it can be accessed only within the same assembly.

An assembly is a collection of types (classes, interfaces, etc) and resources (data). They are built to work together and form a logical unit of functionality.

That's why when we run an assembly all classes and interfaces inside the assembly run together. To learn more, visit the [C# Assembly](https://docs.microsoft.com/en-us/dotnet/standard/assembly/).

### Example: internal within the same Assembly

using System;

namespace Assembly {

class Student {

internal string name = "Sheeran";

}

class Program {

static void Main(string[] args) {

// creating object of Student class

Student theStudent = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + theStudent.name);

Console.ReadLine();

}

}

}

**Output**

Name: Sheeran

In the above example, we have created a class named Student with a field name. Since the field is internal, we are able to access it from the Program class as they are in the same assembly.

If we use internal within a single assembly, it works just like the public access modifier.

### Example: internal in different Assembly

Let's create one assembly first.

// Code on Assembly1

using System;

namespace Assembly1 {

public class StudentName {

internal string name = "Sheeran";

}

class Program {

static void Main(string[] args) {

}

}

}

Here, this code is in **Assembly1**. We have created an internal field name inside the class StudentName. Now, this field can only be accessed from the same assembly **Assembly1**.

Now, let's create another assembly.

// Code on Assembly2

using System;

// access Assembly1

using Assembly1;

namespace Assembly2 {

class Program {

static void Main(string[] args) {

StudentName student = new StudentName();

// accessing name field from Assembly1

Console.Write(student.name);

Console.ReadLine();

}

}

}

Here, this code is in **Assembly2**. We are trying to access the name field of the StudentName class(**Assembly1**).

To access fields from **Assembly1**, we first need to set the reference of **Assembly1** in **Assembly2**. Now the code

using Assembly1;

allows us to use the code from **Assembly1** to **Assembly2**.

Here, when we try to access the name field from **Assembly2**, we get an error.

**Error** CS0122 'StudentName.name' is inaccessible due to its protection level

This is because name is an internal field present in **Assembly1**.

## 5. protected internal access modifier

The protected internal is a combination of protected and internal access modifiers.

When we declare a member protected internal, it can be accessed from the same assembly and the derived class of the containing class from any other assembly.

// Code on Assembly1

using System;

namespace Assembly1 {

public class Greet {

protected internal string msg="Hello";

}

class Program {

static void Main(string[] args) {

Greet greet = new Greet();

Console.WriteLine(greet.msg);

Console.ReadLine();

}

}

}

**Output**

Hello

The above code is in **Assembly1**.

In the above example, we have created a class named Greet with a field msg. Since the field is protected internal, we are able to access it from the Program class as they are in the same assembly.

Let's derive a class from Greet in another assembly and try to access the protected internal field msg from it.

// Code on Assembly2

using System;

// access Assembly1

using Assembly1;

namespace Assembly2 {

// derived class of Greet

class Program: Greet {

static void Main(string[] args) {

Program greet = new Program();

// accessing name field from Assembly1

Console.Write(greet.msg);

Console.ReadLine();

}

}

}

**Output**

Hello

The above code is in **Assembly2**.

In the above example, we have inherited the Program class from the Greet class(from **Assembly1**).

// accessing name field from Assembly1

Console.Write(greet.msg);

We are able to access the msg from the Greet class of **Assembly1** from **Assembly2**.

This is because the msg is a protected internal field and we are trying to access it from the child class of Greet.

## 6. private protected access modifier

The private protected access modifier is a combination of private and protected. It is available from the C# version 7.2 and later.

When we declare a member private protected, it can only be accessed within the same class, and its derived class within the same assembly. For example,

// Code in Assembly1

using System;

namespace Assembly1 {

public class StudentName {

private protected string name = "Sheeran";

}

//derived class of StudentName class

class Program1 : StudentName {

static void Main(string[] args) {

Program1 student = new Program1();

// accessing name field from base class

Console.Write(student.name);

Console.ReadLine();

}

}

}

**Output**

Sheeran

The above code is in **Assembly1**

In the above example, we have created a class StudentName with a private protected field name.

Notice that we have inherited the Program1 class from the StudentName class.

Since the private protected member can be accessed from derived classes within the same assembly, we are able to access name from the Program1 class.

Let's derive a class from StudentName in another assembly and try to access the private protected field name from it. For example,

// Code in Assembly2

using System;

//access Assembly1

using Assembly1;

namespace Assembly2 {

//derived class of StudentName

class Program : StudentName {

static void Main(string[] args) {

Program student = new Program();

// accessing name field from Assembly1

Console.Write(student.name);

Console.ReadLine();

}

}

}

The above code is in **Assembly2**

In the above example, when we try to access the name field from the derived class of StudentName, we get an error.

**Error** CS0122 'StudentName.name' is inaccessible due to its protection level

This is because the name field is in **Assembly1** and the derived class is in **Assembly2**.

**Note**: We can also use access modifiers with types (classes, interface, etc). However, we can only use types with public and internal access modifiers.

**C# Variable Scope**

In this tutorial, you will learn about variable scope in C# with the help of examples.

A variable scope refers to the availability of variables in certain parts of the code.

In C#, a variable has three types of scope:

* Class Level Scope
* Method Level Scope
* Block Level Scope

**C# Class Level Variable Scope**

In C#, when we declare a variable inside a class, the variable can be accessed within the class. This is known as **class level variable scope**.

Class level variables are known as fields and they are declared outside of methods, constructors, and blocks of the class. For example,

using System;

namespace VariableScope {

class Program {

// class level variable

string str = "Class Level";

public void display() {

Console.WriteLine(str);

}

static void Main(string[] args) {

Program ps = new Program();

ps.display();

Console.ReadLine();

}

}

}

**Output**

Class Level

In the above example, we have initialized a variable named str inside the Program class.

Since it is a class level variable, we can access it from a method present inside the class.

public void display() {

Console.WriteLine(str);

}

This is because the class level variable is accessible throughout the class.

**Note**: We cannot access the class level variable through static methods. For example, suppose we have a static method inside the Program class.

static void display2() {

// Access class level variable

// Cause an Error

Console.WriteLine(str);

}

**Method Level Variable Scope**

When we declare a variable inside a method, the variable cannot be accessed outside of the method. This is known as **method level variable scope**. For example,

using System;

namespace VariableScope {

class Program {

public void method1() {

// display variable inside method

string str = "method level";

}

public void method2() {

// accessing str from method2()

Console.WriteLine(str);

}

static void Main(string[] args) {

Program ps = new Program();

ps.method2();

Console.ReadLine();

}

}

}

In the above example, we have created a variable named str inside method1().

// Inside method1()

string str = "method level";

Here, str is a method level variable. So, it cannot be accessed outside method1().

However, when we try to access the str variable from the method2()

// Inside method2

Console.WriteLine(str); // Error code

we get an error.

Error CS0103 The name 'str' does not exist in the current context

This is because method level variables have scope inside the method where they are created. For example,

using System;

namespace VariableScope {

class Program {

public void display() {

string str = "inside method";

// accessing method level variable

Console.WriteLine(str);

}

static void Main(string[] args) {

Program ps = new Program();

ps.display();

Console.ReadLine();

}

}

}

**Output**

inside method

Here, we have created the str variable and accessed it within the same method display(). Hence, the code runs without any error.

**Block Level Variable Scope in C#**

When we declare a variable inside a block ([for loop](https://www.programiz.com/csharp-programming/for-loop), [while loop](https://www.programiz.com/csharp-programming/do-while-loop), [if..else](https://www.programiz.com/csharp-programming/if-else-statement)), the variable can only be accessed within the block. This is known as **block level variable scope**. For example,

using System;

namespace VariableScope {

class Program {

public void display() {

for(int i=0;i<=3;i++) {

}

Console.WriteLine(i);

}

static void Main(string[] args) {

Program ps = new Program();

ps.display();

Console.ReadLine();

}

}

}

In the above program, we have initialized a block level variable i inside the for loop.

for(int i=0;i<=3;i++) {

}

Since i is a block level variable, when we try to access the variable outside the for loop,

// Outside for loop

Console.WriteLine(i);

we get an error.

Error CS0103 The name 'i' does not exist in the current context

# C# Constructor

In this tutorial, we will learn about the C# constructors and their types with the help of examples.

In C#, a constructor is similar to a method that is invoked when an object of the class is created.

However, unlike methods, a constructor:

* has the same name as that of the class
* does not have any return type

## Create a C# constructor

Here's how we create a constructor in C#

class Car {

// constructor

Car() {

//code

}

}

Here, Car() is a constructor. It has the same name as its class.

**Call a constructor**

Once we create a constructor, we can call it using the new keyword. For example,

new Car();

In C#, a constructor is called when we try to create an object of a class. For example,

Car car1 = new Car();

Here, we are calling the Car() constructor to create an object car1. To learn more about objects, visit [C# Class and Objects](https://www.programiz.com/csharp-programming/class-objects).

## Types of Constructors

There are the following types of constructors:

* Parameterless Constructor
* Parameterized Constructor
* Default Constructor

## 1. Parameterless Constructor

When we create a constructor without parameters, it is known as a parameterless constructor. For example,

using System;

namespace Constructor {

class Car {

// parameterless constructor

Car() {

Console.WriteLine("Car Constructor");

}

static void Main(string[] args) {

// call constructor

new Car();

Console.ReadLine();

}

}

}

**Output**

Car Constructor

In the above example, we have created a constructor named Car().

new Car();

We can call a constructor by adding a new keyword to the constructor name.

## 2. C# Parameterized Constructor

In C#, a constructor can also accept parameters. It is called a parameterized constructor. For example,

using System;

namespace Constructor {

class Car {

string brand;

int price;

// parameterized constructor

Car(string theBrand, int thePrice) {

brand = theBrand;

price = thePrice;

}

static void Main(string[] args) {

// call parameterized constructor

Car car1 = new Car("Bugatti", 50000);

Console.WriteLine("Brand: " + car1.brand);

Console.WriteLine("Price: " + car1.price);

Console.ReadLine();

}

}

}

**Output**

Brand: Bugatti

Price: 50000

In the above example, we have created a constructor named Car(). The constructor takes two parameters: theBrand and thePrice.

Notice the statement,

Car car1 = new Car("Bugatti", 50000);

Here, we are passing the two values to the constructor.

The values passed to the constructor are called arguments. We must pass the same number and type of values as parameters.

## 3. Default Constructor

If we have not defined a constructor in our class, then the C# will automatically create a default constructor with an empty code and no parameters. For example,

using System;

namespace Constructor {

class Program {

int a;

static void Main(string[] args) {

// call default constructor

Program p1 = new Program();

Console.WriteLine("Default value of a: " + p1.a);

Console.ReadLine();

}

}

}

**Output**

Default value of a: 0

In the above example, we have not created any constructor in the Program class. However, while creating an object, we are calling the constructor.

Program p1 = new Program();

Here, C# automatically creates a default constructor. The default constructor initializes any uninitialized variable with the default value.

Hence, we get **0** as the value of the int variable a.

**Note**: In the default constructor, all the numeric fields are initialized to 0, whereas string and object are initialized as null.

## 4. Copy Constructor in C#

We use a copy constructor to create an object by copying data from another object. For example,

using System;

namespace Constructor {

class Car {

string brand;

// constructor

Car (string theBrand) {

brand = theBrand;

}

// copy constructor

Car(Car c1) {

brand = c1.brand;

}

static void Main(string[] args) {

// call constructor

Car car1 = new Car("Bugatti");

Console.WriteLine("Brand of car1: " + car1.brand);

// call the copy constructor

Car car2 = new Car(car1);

Console.WriteLine("Brand of car2: " + car2.brand);

Console.ReadLine();

}

}

}

**Output**

Brand of car1: Bugatti

Brand of car2: Bugatti

In the above program, we have used a copy constructor.

Car(Car c1) {

brand = c1.brand;

}

Here, this constructor accepts an object of Car as its parameter. So, when creating the car2 object, we have passed the car1 object as an argument to the copy constructor.

Car car2 = new Car(car1);

Inside the copy constructor, we have assigned the value of the brand for car1 object to the brand variable for car2 object. Hence, both objects have the same value of the brand.

## 5. Private Constructor

We can create a private constructor using the private [access specifier](https://www.programiz.com/csharp-programming/access-modifiers). This is known as a private constructor in C#.

Once the constructor is declared private, we cannot create objects of the class in other classes.

### Example 1: Private Constructor

using System;

namespace Constructor {

class Car {

// private constructor

private Car () {

Console.WriteLine("Private Constructor");

}

}

class CarDrive {

static void Main(string[] args) {

// call private constructor

Car car1 = new Car();

Console.ReadLine();

}

}

}

In the above example, we have created a private constructor Car(). Since private members are not accessed outside of the class, when we try to create an object of Car

// inside CarDrive class

Car car1 = new Car();

we get an error

error CS0122: 'Car.Car()' is inaccessible due to its protection level

**Note**: If a constructor is private, we cannot create objects of the class. Hence, all fields and methods of the class should be declared static, so that they can be accessed using the class name.

## 6. C# Static Constructor

In C#, we can also make our constructor static. We use the static keyword to create a static constructor. For example,

using System;

namespace Constructor {

class Car {

// static constructor

static Car () {

Console.WriteLine("Static Constructor");

}

// parameterless constructor

Car() {

Console.WriteLine("Default Constructor");

}

static void Main(string[] args) {

// call parameterless constructor

Car car1 = new Car();

// call parameterless constructor again

Car car2 = new Car();

Console.ReadLine();

}

}

}

In the above example, we have created a static constructor.

static Car () {

Console.WriteLine("Static Constructor");

}

We cannot call a static constructor directly. However, when we call a regular constructor, the static constructor gets called automatically.

Car car1 = new Car();

Here, we are calling the Car() constructor. You can see that the static constructor is also called along with the regular constructor.

**Output**

Static Constructor

Default Constructor

Default Constructor

The static constructor is called only once during the execution of the program. That's why when we call the constructor again, only the regular constructor is called.

**Note**: We can have only one static constructor in a class. It cannot have any parameters or access modifiers.

## C# Constructor Overloading

In C#, we can create two or more constructor in a class. It is known as constructor overloading. For example,

using System;

namespace ConstructorOverload {

class Car {

// constructor with no parameter

Car() {

Console.WriteLine("Car constructor");

}

// constructor with one parameter

Car(string brand) {

Console.WriteLine("Car constructor with one parameter");

Console.WriteLine("Brand: " + brand);

}

static void Main(string[] args) {

// call constructor with no parameter

Car car = new Car();

Console.WriteLine();

// call constructor with parameter

Car car2 = new Car("Bugatti");

Console.ReadLine();

}

}

}

**Output**

Car constructor

Car constructor with one parameter

Brand: Bugatti

In the above example, we have overloaded the Car constructor:

* one constructor has one parameter
* another has two parameter

Based on the number of the argument passed during the constructor call, the corresponding constructor is called.

Here,

* Object car - calls constructor with one parameter
* Object car2 - calls constructor with two parameter

# C# this Keyword

In this article, we will learn about this keyword in C# with the help of examples.

In C#, this keyword refers to the current instance of a class. For example,

using System;

namespace ThisKeyword {

class Test {

int num;

Test(int num) {

// this.num refers to the instance field

this.num = num;

Console.WriteLine("object of this: " + this);

}

static void Main(string[] args) {

Test t1 = new Test(4);

Console.WriteLine("object of t1: " + t1);

Console.ReadLine();

}

}

}

**Output**

object of this: ThisKeyword.Test

object of t1: ThisKeyword.Test

In the above example, we have created an object named t1 of the class Test. We have printed the name of the object t1 and this keyword of the class.

Here, we can see the name of both t1 and this is the same. This is because this keyword refers to the current instance of the class which is t1.

Here are some of the major uses of this keyword in C#.

## C# this with Same Name Variables

We cannot declare two or more variables with the same name inside a scope (class or method). However, instance variables and parameters may have the same name. For example,

using System;

namespace ThisKeyword {

class Test {

int num;

Test(int num) {

num = num;

}

static void Main(string[] args) {

Test t1 = new Test(4);

Console.WriteLine("value of num: " + t1.num);

Console.ReadLine();

}

}

}

**Output**

0

In the above program, the instance variable and the parameter have the same name: num. We have passed 4 as a value to the constructor.

However, we are getting **0** as an output. This is because the C# gets confused because the names of the instance variable and the parameter are the same.

We can solve this issue by using this.

### Example: this with Same Name Variables

using System;

namespace ThisKeyword {

class Test {

int num;

Test(int num) {

// this.num refers to the instance field

this.num = num;

}

static void Main(string[] args) {

Test t1 = new Test(4);

Console.WriteLine("value of num: " +t1.num);

Console.ReadLine();

}

}

}

**Output**

value of num: 4

Now, we are getting the expected output that is **4**. It is because this.num refers to the instance variable of the class.

So, there is no confusion between the names of the instance variable and the parameter.

## Invoke Constructor of the Same Class Using this

While working with constructor overloading, we might have to invoke one constructor from another constructor. In this case, we can use this keyword. For example,

using System;

namespace ThisKeyword {

class Test {

Test(int num1, int num2) {

Console.WriteLine("Constructor with two parameter");

}

// invokes the constructor with 2 parameters

Test(int num) : this(33, 22) {

Console.WriteLine("Constructor with one parameter");

}

public static void Main(String[] args) {

Test t1 = new Test(11);

Console.ReadLine();

}

}

}

**Output**

Constructor with two parameter

Constructor with one parameter

In the above example, we have used : followed by this keyword to call constructor Test(int num1, num2) from the constructor Test(int num).

When we call the Test(int num) constructor the Test(int num1, int num2) constructor executes first.

**Note**: Calling one constructor from another constructor is known as constructor chaining.

## C# this as an object argument

We can use this keyword to pass the current object as an argument to a method. For example,

using System;

namespace ThisKeyword {

class Test {

int num1;

int num2;

Test() {

num1 = 22;

num2 = 33;

}

// method that accepts this as argument

void passParameter(Test t1)

{

Console.WriteLine("num1: " + num1);

Console.WriteLine("num2: " + num2);

}

void display() {

// passing this as a parameter

passParameter(this);

}

public static void Main(String[] args) {

Test t1 = new Test();

t1.display();

Console.ReadLine();

}

}

}

**Output**

num1: 22

num2: 33

In the above program, we have a method passParameter(). It accepts the object of the class as an argument.

passParameter(this);

Here, we have passed this to the passParameter() method. As this refers to the instance of the class, we are able to access the value of num1 and num2.

## this to declare a C# indexer

Indexers allow objects of a class to be indexed just like arrays. We use this keyword to declare an indexer in C#. For example,

using System;

namespace ThisKeyword {

class Student {

private string[] name = new string[3];

// declaring an indexer

public string this[int index] {

// returns value of array element

get {

return name[index];

}

// sets value of array element

set {

name[index] = value;

}

}

}

class Program {

public static void Main() {

Student s1 = new Student();

s1[0] = "Ram";

s1[1] = "Shyam";

s1[2] = "Gopal";

for (int i = 0; i < 3; i++) {

Console.WriteLine(s1[i] + " ");

}

}

}

}

**Output**

Ram

Shyam

Gopal

In the above program, we have created an indexer using this keyword.

The array name[] is private. So, we cannot access it from the Program class.

Now, to access and set the value of the array, we use an indexer.

Student s1 = new Student();

s1[0] = "Ram";

s1[1] = "Shyam";

s1[2] = "Gopal";

As we have used this to create an indexer, we must use the object of the Student class to access the indexer.

# C# static Keyword

In this tutorial, we will learn about the static keyword in C# with the help of examples.

In C#, if we use a static keyword with class members, then there will be a single copy of the type member.

And, all objects of the class share a single copy instead of creating individual copies.

## C# Static Variables

If a variable is declared static, we can access the variable using the class name. For example,

using System;

namespace StaticKeyword {

class Student {

// static variable

public static string department = "Computer Science";

}

class Program {

static void Main(string[] argos) {

// access static variable

Console.WriteLine("Department: " + Student.department);

Console.ReadLine();

}

}

}

**Output**

Department: Computer Science

In the above example, we have created a static variable named department. Since the variable is static, we have used the class name Student to access the variable.

### Static Variables Vs Instance Variables

In C#, every object of a class will have its own copy of instance variables. For example,

class Student {

// instance variable

public string studentName;

}

class Program {

static void Main(string[] args) {

Student s1 = new Student();

Student s2 = new Student();

}

}

Here, both the objects s1 and s2 will have separate copies of the variable studentName. And, they are different from each other.

However, if we declare a variable static, all objects of the class share the same static variable. And, we don't need to create objects of the class to access the static variables.

### Example: C# Static Variable Vs. Instance Variable

using System;

namespace StaticKeyword {

class Student {

static public string schoolName = "Programiz School";

public string studentName;

}

class Program {

static void Main(string[] args) {

Student s1 = new Student();

s1.studentName = "Ram";

// calls instance variable

Console.WriteLine("Name: " + s1.studentName);

// calls static variable

Console.WriteLine("School: " + Student.schoolName);

Student s2 = new Student();

s2.studentName = "Shyam";

// calls instance variable

Console.WriteLine("Name: " + s2.studentName);

// calls static variable

Console.WriteLine("School: " + Student.schoolName);

Console.ReadLine();

}

}

}

**Output**

Name: Ram

School: Programiz School

Name: Shyam

School: Programiz School

In the above program, the Student class has a non-static variable named studentName and a static variable named schoolName.

Inside the Program class,

* s1.studentName / s2.studentName - calls the non-static variable using objects s1 and s2 respectively
* Student.schoolName - calls the static variable by using the class name

Since the schoolName is the same for all students, it is good to make the schoolName static. It saves memory and makes the program more efficient.

## C# Static Methods

Just like static variables, we can call the static methods using the class name.

class Test {

public static void display() {....}

}

class Program {

static void Main(string[] args) {

Test.display();

}

}

Here, we have accessed the static method directly from Program classes using the class name.

When we declare a method static, all objects of the class share the same static method.

### Example: C# Static and Non-static Methods

using System;

namespace StaticKeyword {

class Test {

public void display1() {

Console.WriteLine("Non static method");

}

public static void display2() {

Console.WriteLine("Static method");

}

}

class Program {

static void Main(string[] args) {

Test t1 = new Test();

t1.display1();

Test.display2();

Console.ReadLine();

}

}

}

**Output**

Non static method

Static method

In the above program, we have declared a non-static method named display1() and a static method named display2() inside the class Test.

Inside the Program class,

* t1.display1() - access the non-static method using s1 object
* Test.display2() - access the static method using the class name Test

**Note**: In C#, the Main method is static. So, we can call it without creating the object.

## C# Static Class

In C#, when we declare a class as static, we cannot create objects of the class. For example,

using System;

namespace StaticKeyword {

static class Test {

static int a = 5;

static void display() {

Console.WriteLine("Static method");

}

static void Main(string[] args) {

// creating object of Test

Test t1 = new Test();

Console.WriteLine(a);

display();

}

}

}

In the above example, we have a static class Test. We have created an object t1 of the class Test.

Since we cannot make an object of the static class, we get the following error:

error CS0723: Cannot declare a variable of static type 'Test'

error CS0712: Cannot create an instance of the static class

Notice the field and method of the static class are also static because we can only have static members inside the static class.

**Note**: We cannot inherit a static class in C#. For example,

static class A {

...

}

// Error Code

class B : A {

...

}

## Access static Members within the Class

If we are accessing the static variables and methods inside the same class, we can directly access them without using the class name. For example,

using System;

namespace StaticKeyword {

class Test {

static int age = 25;

public static void display() {

Console.WriteLine("Static method");

}

static void Main(string[] args) {

Console.WriteLine(age);

display();

Console.ReadLine();

}

}

}

**Output**

25

Static method

Here, we are accessing the static field age and static method display() without using the class name.

# C# String

In this tutorial, we will learn about C# string and its methods with the help of examples.

In C#, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use the string keyword to create a string. For example,

// create a string

string str = "C# Programming";

Here, we have created a string named str and assigned the text "C# Programming". We use double quotes to represent strings in C#.

### Example: Create string in C#

using System;

namespace CsharpString {

class Test {

public static void Main(string [] args) {

// create string

string str1 = "C# Programming";

string str2 = "Programiz";

// print string

Console.WriteLine(str1);

Console.WriteLine(str2);

Console.ReadLine();

}

}

}

**Output**

C# Programming

Programiz

In the above example, we have created two strings named str1 and str2 and printed them.

**Note**: A string variable in C# is not of primitive types like int, char, etc. Instead, it is an object of the String class.

## String Operations

C# string provides various methods to perform different operations on strings. We will look into some of the commonly used string operations.

### 1. Get the Length of a string

To find the length of a string, we use the Length property. For example,

using System;

namespace CsharpString {

class Test {

public static void Main(string [] args) {

// create string

string str = "C# Programming";

Console.WriteLine("string: " + str);

// get length of str

int length = str.Length;

Console.WriteLine("Length: "+ length);

Console.ReadLine();

}

}

}

**Output**

string: C# Programming

Length: 14

In the above example, the Length property calculates the total number of characters in the string and returns it.

### 2. Join two strings in C#

We can join two strings in C# using the Concat() method. For example,

using System;

namespace CsharpString {

class Test {

public static void Main(string [] args) {

// create string

string str1 = "C# ";

Console.WriteLine("string str1: " + str1);

// create string

string str2 = "Programming";

Console.WriteLine("string str2: " + str2);

// join two strings

string joinedString = string.Concat(str1, str2);

Console.WriteLine("Joined string: " + joinedString);

Console.ReadLine();

}

}

}

**Output**

string str1: C#

string str2: Programming

Joined string: C# Programming

In the above example, we have created two strings named str1 and str2. Notice the statement,

string joinedString = string.Concat(str1, str2);

Here, the Concat() method joins str1 and str2 and assigns it to the joinedString variable.

We can also join two strings using the + operator in C#. To learn more, visit C# string Concat.

### 3. C# compare two strings

In C#, we can make comparisons between two strings using the Equals() method. The Equals() method checks if two strings are equal or not. For example,

using System;

namespace CsharpString {

class Test {

public static void Main(string [] args) {

// create string

string str1 = "C# Programming";

string str2 = "C# Programming";

string str3 = "Programiz";

// compare str1 and str2

Boolean result1 = str1.Equals(str2);

Console.WriteLine("string str1 and str2 are equal: " + result1);

//compare str1 and str3

Boolean result2 = str1.Equals(str3);

Console.WriteLine("string str1 and str3 are equal: " + result2);

Console.ReadLine();

}

}

}

**Output**

string str1 and str2 are equal: True

string str1 and str3 are equal: False

In the above example, we have created 3 strings named str1, str2, and str3. Here, we are using the Equals() method to check if one string is equal to another.

## Immutability of String Objects

In C#, strings are immutable. This means, once we create a string, we cannot change that string.

To understand it, consider an example:

// create string

string str = "Hello ";

Here, we have created a string variable named str. The variable holds the string "Hello ".

Now suppose we want to change the string str.

// add another string "World"

// to the previous string example

str = string.Concat(str, "World");

Here, we are using the Concat() method to add the string "World" to the previous string str.

But how are we able to modify the string when they are immutable?

Let's see what has happened here,

1. C# takes the value of the string "Hello ".
2. Creates a new string by adding "World" to the string "Hello ".
3. Creates a new string object, gives it a value "Hello World", and stores it in str.
4. The original string, "Hello ", that was assigned to str is released for garbage collection because no other variable holds a reference to it.

## String Escape Sequences

The escape character is used to escape some of the characters present inside a string. In other words, we use escape sequences to insert special characters inside the string.

Suppose we need to include double quotes inside a string.

// include double quote

string str = "This is the "String" class";

Since strings are represented by double quotes, the compiler will treat "This is the " as the string. And the above code will cause an error.

To solve this issue, we use the escape character \" in C#. For example,

// use the escape character

string str = "This is the \"String\" class.";

Now by using \ before double quote ", we can include it in the string.

Some of the escape sequences in C# are as follows:

|  |  |
| --- | --- |
| Escape Sequence | Character Name |
| \' | single quote |
| \" | double quote |
| \\ | backslash |
| \0 | null |
| \n | new line |
| \t | horizontal tab |

String interpolation

In C#, we can use string interpolation to insert variables inside a string. For string interpolation, the string literal must begin with the $ character. For example,

using System;

namespace CsharpString {

class Test {

public static void Main(string [] args) {

// create string

string name = "Programiz";

// string interpolation

string message = $"Welcome to {name}";

Console.WriteLine(message);

Console.ReadLine();

}

}

}

**Output**

Welcome to Programiz

In the above example, we are using the name variable inside the message string.

string message = $"Welcome to {name}";

Notice that,

* the string literal starts with $
* the name variable is placed inside the curly braces {}

## Methods of C# string

There are various string methods in C#. Some of them are as follows:

|  |  |
| --- | --- |
| Methods | Description |
| Format() | returns a formatted string |
| Split() | splits the string into substring |
| Substring() | returns substring of a string |
| Compare() | compares string objects |
| Replace() | replaces the specified old character with the specified new character |
| Contains() | checks whether the string contains a substring |
| Join() | joins the given strings using the specified separator |
| Trim() | removes any leading and trailing whitespaces |
| EndsWith() | checks if the string ends with the given string |
| IndexOf() | returns the position of the specified character in the string |
| Remove() | returns characters from a string |
| ToUpper() | converts the string to uppercase |
| ToLower() | converts the string to lowercase |
| PadLeft() | returns string padded with spaces or with a specified Unicode character on the left |
| PadRight() | returns string padded with spaces or with a specified Unicode character on the right |
| StartsWith() | checks if the string begins with the given string |
| ToCharArray() | converts the string to a char array |
|  |  |
| LastIndexOf() | returns index of the last occurrence of a specified string |