**Access Specifiers**

* Access Specifiers are a special kind of modifiers using which we can define the scope of a type and its members
* C# Supports 5 different access specifiers they are

1- private

2- protected

3- internal

4- protected internal

5- public

* Each access modifier has a different scope
* Public can be accessed from anywhere

**Note**: the important point to remember here is that a member of a class defined with any scope is always accessible within the class. If there are any restrictions these restrictions will start when we call the members form out their class

See the example below where we defined 5 different method with different access specifiers.

public class Program

{

private void Test1()

{

Console.WriteLine("Private Method");

}

protected void Test2()

{

Console.WriteLine("protected Method");

}

internal void Test3()

{

Console.WriteLine("Internal Method");

}

protected internal void Test4()

{

Console.WriteLine("protected Internal Method");

}

public void Test5()

{

Console.WriteLine("public Method");

}

We can call all the 5 method from their class without any difference.

static void Main(string[] args)

{

Program p=new Program();

p.Test1();

p.Test2();

p.Test3();

p.Test4();

p.Test5();

}

}

Now let’s add a new class in the same project and make it inherent from Program class

namespace AccessDemo1

{

class ClassTwo:Program

{

static void Main(string[] args)

{

ClassTwo obj= new ClassTwo();

obj.Test2();

obj.Test3();

obj.Test4();

obj.Test5();

}

}

}

In the above code

* The new class called ClassTwo is inheriting from the Program class
* Inside the Main() method of the ClassTwo we created an instance of this class called obj
* As inheritance concept suggests, the ClassTwo will have all the members of the parent class, but you will notice that using the obj we can only access four method of the parent class and we can’t access the private method, because this method is a private method and it can be accessed only within its class

**Private access specifier**

* **Remember**: the **default** **scope** of every member of a class is **private**.
* The private members will be accessed only within the class in which it was defined
* The child class can’t access the private members of its parent class
* We can’t declare a class as a private (we can only use public or internal. If we don’t specify the access specifier of a class then it’s going to be internal by default)

Now let’s add one more class to the project and here we will not use inheritance, but we will access the members of the other class by creating an instance of that class

namespace AccessDemo1

{

class ClassThree

{

static void Main(string[] args)

{

Program obj = new Program();

obj.Test3();

obj.Test4();

obj.Test5();

}

}

}

In the above code

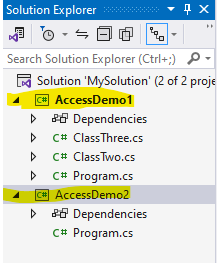
* We created an instance of the Program class called obj using which we can access the members of the Program class.
* As you can notice, using the instance of the class, we can only access three methods (the public, internal, and protected Internal), **but we can’t access the private and protected** methods.

**Protected access specifier**

* Members declared as protected are accessed only from their classes and from classes that inherent that class. **They are not accessible form non-child classes**
* We can’t declare a class as protected class

**Now** to learn the other two access specifiers (**internal** and **internal protected**) we need to create a new project as following:

Under the same solution of the first project add a new project by right clicking on the solution name and select add then new project. Give it a name as your wish I will name it AccessDemo2



**Note**

In order to be able to access the classes of the first project, you need to add a project reference in the other project (AccessDemo2) by right clicking the project name (AccessDemo2) and then select add then select project reference. Select the .dll file of the other project (AccessDemo1)

Now create a new class that inherent the Program class of the first project and create an instance of this class as following

namespace AccessDemo2

{

class ClassFour:AccessDemo1.Program

{

static void Main(string[] args)

{

ClassFour obj= new ClassFour();

obj.Test2();

obj.Test4();

obj.Test5();

}

}

}

From the above code you can see that we can only access three methods of the program class (protected, protected internal, and public), but we can’t call the private and internal methods.

**Internal access specifier**

* If you declare a class or a member of a class as internal, then it can be accessed only within the same project either from child or non-child classes
* The default scope of the class is internal

**Protected internal access specifier:**

* It’s combination of both protected and internal
* If any one of these protected or internal is accessible then the protected internal will also be accessible otherwise it will not be accessible(this means that if protected and internal are both not accessible then protected internal will not be accessible as well.
* In the following example we will add a new class called ClassFive and inside its Main() method we will try to access the Program class of the other project by creating an instance of the Program class

using AccessDemo1;

namespace AccessDemo2

{

class ClassFive

{

static void Main(string[] args)

{

Program obj=new Program();

obj.Test5();

}

}

}

As you can see from the code above we can only access one method which is Test5() that was declared as public in Program class

The table below is the summary table of all access specifiers discussed here

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access specifier** | **Accessing the members from same project** | | **Accessing the members from another project** | |
| From child class | From non-child class | From child class | From non-child class |
| Public |  |  |  |  |
| Private | X | X | X | X |
| Protected |  | X |  | X |
| Internal |  |  | X | X |
| Protected internal |  |  |  | X |

**Different kinds of Variables in a class**

* A variable in C# class can be one of the following types

1. Static
2. Non-Static
3. Constant
4. Readonly

**Static and non-static variables**

* If the variable is explicitly declared using static keyword, or if the variable declared inside a static block, then it’s called as a static variable. The rest of others are non-static

class Program

{

static int x; // static variable

int y; // non-static variable

static void Main(string[] args)

{

int z; // static variable

}

}

* In the above example, we can directly access the static variable x from the Main() method which is a static block without the need to create an instance of the class. This is because the initialization of the static variable will be done immediately when the execution of the class starts. But for non-static variable the memory will be allocated only after we create an instance of the class.

class Program

{

static int x=100; // static variable

int y=200; // non-static variable

static void Main(string[] args)

{

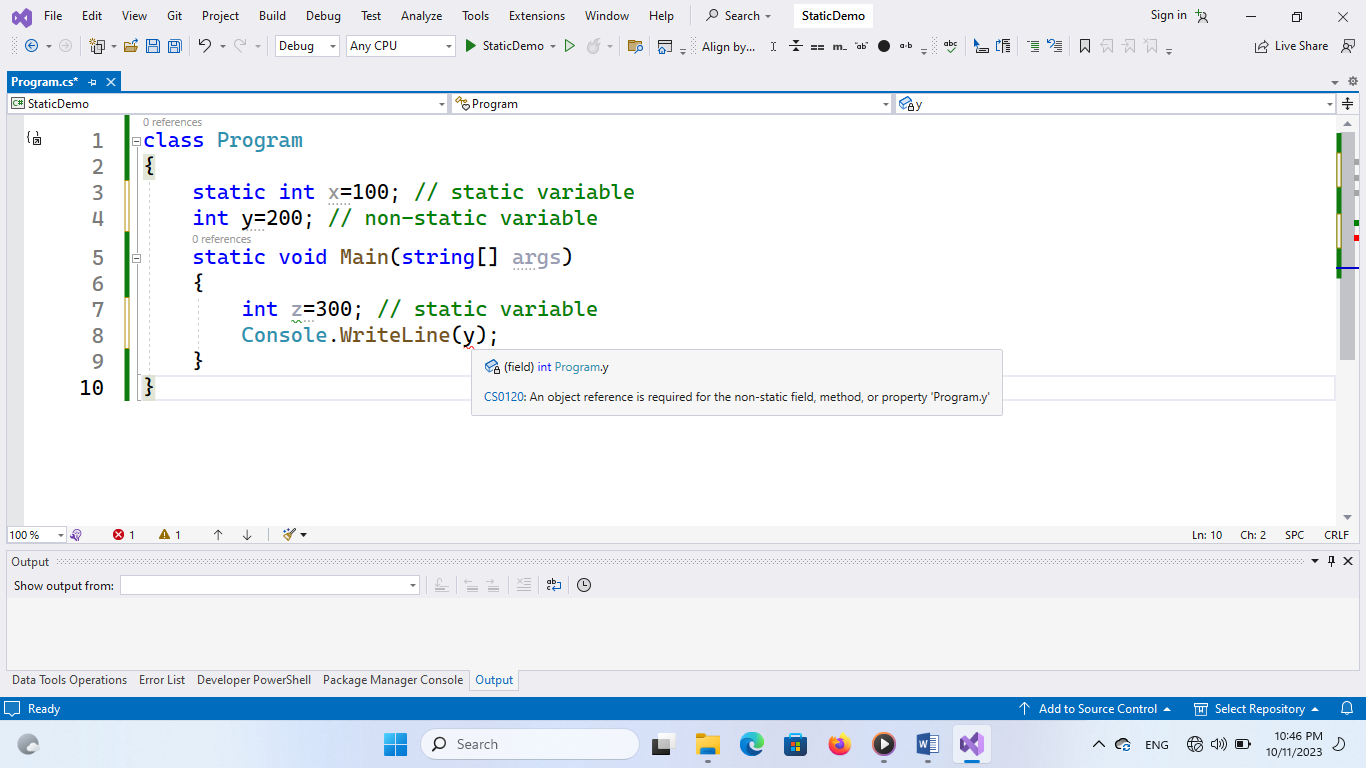
int z=300; // static variable

Console.WriteLine(x);

}

}

* As you can see in the above code example we are able to print the static variable x without the need of creating an instance from the Program class.
* If you try to access the non-static variable “y” from the Main() method, you will get an error states that you require an object reference.



* Every time when we create an instance of the class, the memory for the non-static variable will be allocated see the code and the figure below

class Program

{

static int x=100; // static variable

int y=200; // non-static variable

static void Main(string[] args)

{

int z=300; // static variable

Console.WriteLine(x);

Console.WriteLine(z);

//first instance of the class is p1

Program p1 = new Program();

Console.WriteLine(p1.y);

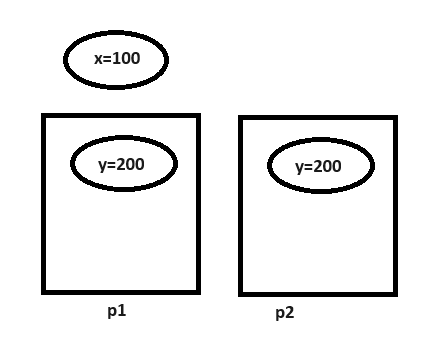
//second instance of the class is p2

Program p2 = new Program();

Console.WriteLine(p2.y);

}

}



* In the life cycle of the class, the static variables are initialized one and only one time. Whereas non-static variable, they are initialized zero time if no instance of the class is created and n times if n instances are created.
* Initialization of non-static variables is associated with instance creation and constructor calling, so these variables will initialized through constructors also.

class Program

{

static int x=100; // static variable

int y=200; // non-static variable

//constructor

public Program(int y)

{

this.y = y;

}

static void Main(string[] args)

{

int z=300; // static variable

Console.WriteLine(x);

Console.WriteLine(z);

//first instance of the class is p1

Program p1 = new Program(50);

Console.WriteLine(p1.y);

//second instance of the class is p2

Program p2 = new Program(80);

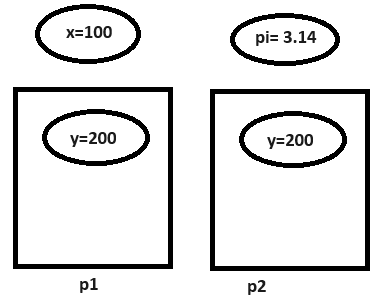
Console.WriteLine(p2.y);

}

}

**Constant Variables**

* If the variable is declared using the **“const”** keyword, then it’s called “constant variable”. These constant variable can’t be modified after their declaration.
* It’s must to initialize the constant variables at the time of declaration only.
* **The behavior of the constant variable** will be **similar** to the b**ehavior of the static variables**. This means the constant variables will be initialized one and only one in the life cycle of the class and doesn’t require the instance of the class for accessing or initializing them.



* **The only difference** between static variable and constant variable is that static variable can be modified whereas constant variables can’t be modified.

**Readonly variables**

* If a variable is declared using the **readonly** keyword, then we call that variable as readonly variable.
* **These variable can’t be modified like constant variable**, but only after initialization. This means that **it’s not compulsory to initialize the readonly variable at the time of declaration**. And they can also be initialized in the constructor also.

class Program

{

static int x=100; // static variable

int y=200; // non-static variable

const float pi = 3.14f; // constant variable

readonly bool flag; // readonly variable

//constructor

public Program(int y)

{

this.y = y;

this.flag = true;

}

static void Main(string[] args)

{

int z=300; // static variable

Console.WriteLine(x);

Console.WriteLine(z);

//first instance of the class is p1

Program p1 = new Program(50);

Console.WriteLine(p1.flag);

}

}

* The **behavior of readonly variables will be similar to the behavior of the non-static** variables. This means that the readonly variable will be initialized only at the time of creating the instance of the class and once for each instance of the class creation.

class Program

{

static int x=100; // static variable

int y=200; // non-static variable

const float pi = 3.14f; // constant variable

readonly bool flag; // readonly variable

//constructor

public Program(int y, bool flag)

{

this.y = y;

this.flag = flag;

}

static void Main(string[] args)

{

int z=300; // static variable

Console.WriteLine(x);

Console.WriteLine(z);

//first instance of the class is p1

Program p1 = new Program(50, true);

Console.WriteLine(p1.y);

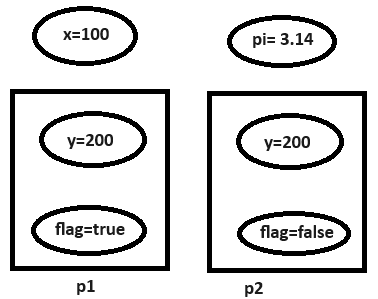
//second instance of the class is p2

Program p2 = new Program(80, false);

Console.WriteLine(p2.y);

}

}



* **The only difference** between readonly variables and non-static variables is that not-static variables can be modified whereas readonly variable can’t be modified.
* Constant variable is fixed for the whole class whereas readonly is a fixed variable specific to an instance of the class.