

The this reference

The “this” keyword in C# is used to refer to the current instance of the class. It is also used to differentiate between the method parameters and class fields if they both have the same name. Another usage of “this” keyword is to call another constructor from a constructor in the same class.

Here, for an example, we are showing a record of Students i.e: id, Name, Age, and Subject. To refer to the fields of the current class, we have used the “this” keyword in C#:

```
public Student(int id, String name, int age, String subject) {  
    this.id = id;  
    this.name = name;  
    this.subject = subject;  
    this.age = age;  
}
```

Let us see the complete example to learn how to work with the “this” keyword in C#:

```
using System;
```

```
class Student {  
    public int id, age;  
    public String name, subject;  
  
    public Student(int id, String name, int age, String subject) {  
        this.id = id;  
        this.name = name;  
        this.subject = subject;  
        this.age = age;  
    }  
  
    public void showInfo() {  
        Console.WriteLine(id + " " + name + " " + age + " " + subject);  
    }  
}  
  
class StudentDetails {  
    public static void Main(string[] args) {
```

```

        Student std1 = new Student(001, "Jack", 23, "Maths");
        std1.showInfo();
    }
}

```

Output: 001 Jack 23 Maths

Properties

Properties look like fields from the outside, but internally they contain logic, like methods do. Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors(get and set)** through which the values of the private fields can be read, written or manipulated.

Usually, inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

```

using System;
class MyClass
{
    private int x;
    public void SetX(int i)
    {
        x = i;
    }
    public int GetX()
    {
        return x;
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.SetX(10);
        int xVal = mc.GetX();
        Console.WriteLine(xVal);
    }
}

```

Output: 10

Automatic Properties

The most common implementation for a property is a getter and/or setter that simply reads and writes to a private field of the same type as the property. An automatic property declaration instructs the compiler to provide this implementation. We can improve the first example in this section by declaring `CurrentPrice` as an automatic property:

```
public class Stock
{
    ...
    public decimal CurrentPrice { get; set; }
}
```

```
class Chk {
    public int a { get; set; }
    public int b { get; set; }
    public int sum {
        get { return a + b; }
    }
}

class Test {
    static void Main() {
        Chk obj = new Chk();
        obj.a = 10; obj.b = 5;
        Console.WriteLine("Sum of "+obj.a+" and "+obj.b+" = "+obj.sum);
        Console.ReadKey();
    }
}
```

Output:

Sum of 10 and 5 = 15

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index argument rather than a property name.

The string class has an indexer that lets you access each of its char values via an int index:

```
string s = "hello";  
Console.WriteLine (s[0]); // 'h'  
Console.WriteLine (s[3]); // 'l'
```

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s).

C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Defining an indexer allows you to create a class like that can allows its items to be accessed an array. Instances of that class can be accessed using the [] array access operator.

```
<modifier> <return type> this [argument list]  
{  
    get  
    {  
        // your get block code  
    }  
    set  
    {  
        // your set block code  
    }  
}
```

In the above code:

<modifier>

can be private, public, protected or internal.

<return type>

can be any valid C# types.

this

this is a special keyword in C# to indicate the object of the current class.

[argument list]

The formal-argument-list specifies the parameters of the indexer. Following program demonstrates how to use an indexer.

```

class Program
{
    class IndexerClass
    {
        private string[] names = new string[10];
        public string this[int i]
        {
            get
            {
                return names[i];
            }
            set
            {
                names[i] = value;
            }
        }
    }
    static void Main(string[] args)
    {
        IndexerClass Team = new IndexerClass();
        Team[0] = "Rocky";
        Team[1] = "Teena";
        Team[2] = "Ana";
        Team[3] = "Victoria";
        Team[4] = "Yani";
        Team[5] = "Mary";
        Team[6] = "Gomes";
        Team[7] = "Arnold";
        Team[8] = "Mike";
        Team[9] = "Peter";
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(Team[i]);
        }
        Console.ReadKey();
    }
}

```

Difference between Indexers and Properties

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

Static Classes

A C# static class is a class that can't be instantiated. The sole purpose of the class is to provide blueprints of its inherited classes. A static class is created using the "static" keyword in C#. A static class can contain static members only. You can't create an object for the static class.

Advantages of Static Classes

1. If you declare any member as a non-static member, you will get an error.
2. When you try to create an instance to the static class, it again generates a compile time error, because the static members can be accessed directly with its class name.
3. The static keyword is used before the class keyword in a class definition to declare a static class.
4. A static class members are accessed by the class name followed by the member name.

Syntax of static class

```
static class classname
{
    //static data members
    //static methods
}
```

Static members of a class

If we declare any members of a class as static we can access it without creating object of that class.

Example

```
class MyCollege
{
    //static fields
    public static string CollegeName;
    public static string Address;

    //static constructor
    static MyCollege()
    {
        CollegeName = "ABC College of Technology";
        Address = "Hyderabad";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyCollege.CollegeName);
        Console.WriteLine(MyCollege.Address);
        Console.Read();
    }
}
```

Example of static class

```
// Creating static class
// Using static keyword
static class Author {

    // Static data members of Author
    public static string A_name = "Ankita";
    public static string L_name = "CSharp";
    public static int T_no = 84;

    // Static method of Author
    public static void details()
    {
        Console.WriteLine("The details of Author is:");
    }
}

// Driver Class
public class GFG {

    // Main Method
    static public void Main()
    {

        // Calling static method of Author
        Author.details();

        // Accessing the static data members of Author
        Console.WriteLine("Author name : {0} ", Author.A_name);
        Console.WriteLine("Language : {0} ", Author.L_name);
        Console.WriteLine("Total number of articles : {0} ",
                           Author.T_no);
    }
}
```

Output

The details of Author is:

Author name : Ankita

Language : CSharp

Total number of articles : 84

Finalizers

Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the ~ symbol:

```
class Class1
{
    ~Class1()
    {
        ...
    }
}
```

Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allow to **inherit** the features (**fields and methods**) of another class. The process by which one class acquires the properties (data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship. Inheritance is used in java for the following:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name : Superclass-name
{
    //methods and fields
}
```

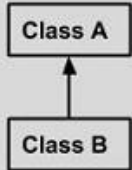
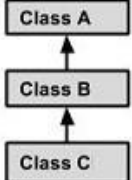
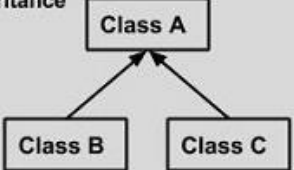
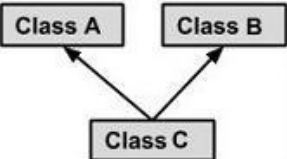
The : indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of C#, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in C#

On the basis of class, there can be **three** types of inheritance in java: **single, multilevel and hierarchical**.

In C# programming, **multiple and hybrid inheritance** is supported through interface only.

Single Inheritance  <pre>graph BT B[Class B] --> A[Class A]</pre>	<pre>public class A { } public class B extends A { }</pre>
Multi Level Inheritance  <pre>graph BT C[Class C] --> B[Class B] B --> A[Class A]</pre>	<pre>public class A {} public class B extends A {.....} public class C extends B {.....}</pre>
Hierarchical Inheritance  <pre>graph BT B[Class B] --> A[Class A] C[Class C] --> A</pre>	<pre>public class A {} public class B extends A {.....} public class C extends A {.....}</pre>
Multiple Inheritance  <pre>graph BT C[Class C] --> A[Class A] C --> B[Class B]</pre>	<pre>public class A {} public class B {.....} public class C extends A,B { } // Java does not support multiple Inheritance</pre>

Note: Codes are written in Java, So please change the syntax of the program according to C# language.

1. Single Inheritance

Single Inheritance refers to a child and parent class relationship where a class extends the another

```
class class A
```

```
{
    public int a=10, b=5;
}
```

```
class B : A
```

```
{
    int a=30, b=5;
    public void test()
    {
        Console.WriteLine("Value of a is: "+a);
        Console.WriteLine("Value of a is: " + base.a);
    }
}
```

```
//driver class
```

```
class Inherit
```

```
{
    static void Main(string[] args) {
        B obj = new B();
        obj.test();
        Console.ReadLine();
    }
}
```

2. Multilevel Inheritance

Multilevel inheritance refers to a child and parent class relationship where a class extends the child class. For example, class C extends class B and class B extends class A.

```
class A5 {
```

```
    public int a, b, c;
    public void ReadData(int a, int b) {
        this.a = a;
```

```

        this.b = b;
    }
    public void Display(){
        Console.WriteLine("Value of a is: "+a);
        Console.WriteLine("Value of b is: " + b);
    }
}
class A6 : A5
{
    public void Add() {
        base.c = base.a + base.b;
        Console.WriteLine("Sum="+base.c);
    }
}
class A7 : A6
{
    public void Sub() {
        base.c = base.a - base.b;
        Console.WriteLine("Difference=" + base.c);
    }
}
class Level {
    static void Main() {
        A7 obj = new A7();
        obj.ReadData(20,5);
        obj.Display();
        obj.Add();
        obj.Sub();
        Console.ReadLine();
    }
}

```

3. Hierarchical Inheritance

Hierarchical inheritance refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```
class Polygon {
    public int dim1,dim2;
    public void ReadDimension(int dim1, int dim2) {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }
}

class Rectangle : Polygon {
    public void AreaRec() {
        base.ReadDimension(10,5);
        int area = base.dim1 * base.dim2;
        Console.WriteLine("Area of Rectangle="+area);
    }
}

class Traingle : Polygon
{
    public void AreaTri()
    {
        base.ReadDimension(10,5);
        double area = 0.5*base.dim1 * base.dim2;
        Console.WriteLine("Area of Triangle=" + area);
    }
}

//driver class
class Hier {
    static void Main() {
        Traingle tri = new Traingle();
        //tri.ReadDimension(10,5);
        tri.AreaTri();
        Rectangle rec = new Rectangle();
    }
}
```

```

        //rec.ReadDimension(10,7);
        rec.AreaRec();

        Console.ReadLine();
    }
}

```

4. Multiple Inheritance

When one class extends more than one classes then this is called multiple inheritance. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance.

C# doesn't allow multiple inheritance. We can use **interfaces** instead of **classes** to achieve the same purpose.

```

interface IA {
    // doesn't contain fields
    int CalculateArea();
    int CalculatePerimeter();
}

class CA {
    public int l, b;
    public void ReadData(int l, int b) {
        this.l = l;
        this.b = b;
    }
}

class BB : CA, IA {
    public int CalculateArea() {
        ReadData(10,5);
        int area=l*b;
        return area;
    }

    public int CalculatePerimeter() {
        ReadData(15, 10);
        int peri = 2*(l+b);
    }
}

```

```
        return peri;
    }
}

//driver class
class Inter {
    static void Main(string[] args) {
        BB obj = new BB();
        //int area=obj.CalculateArea();
        //int peri=obj.CalculatePerimeter();
        Console.WriteLine("Area of Rectangle=" + obj.CalculateArea());
        Console.WriteLine("Perimeter of Rectangle=" + obj.CalculatePerimeter());
        Console.ReadKey();
    }
}
```

Interface in C#

An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties. The reason interfaces only provide declarations is because they are inherited by structs and classes, that must provide an implementation for each interface member declared.

Like a class, an interface can have methods and properties, but the methods declared in interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

Why do we use interface?

- It is used to achieve total abstraction.
- Since C# does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.