



Krishna Nigalye

Software Developer

Learning C-Sharp Interface

WHAT IS AN INTERFACE WHEN TO USE?

- By convention, name of the interface should start with 'I'.

```
public interface IRemote
{
    ...
}
```

- Interface should always be defined as 'public' or 'internal'. If you declare it as a private, protected, protected internal or private protected, then compiler will show you an error.

```
namespace c_sharp_interface
{
    private interface IRemote
    {
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
}
```

interface c_sharp_interface.IRemote

Elements defined in a namespace cannot be explicitly declared as private, protected, protected internal, or private protected

- Interface are not allowed to have constructors.

```
public interface IRemote
{
    public IRemote()
    {
    }
}
```

IRemote.IRemote()

Interfaces cannot contain instance constructors

- Like abstract classes, you cannot create an instance of the interface.

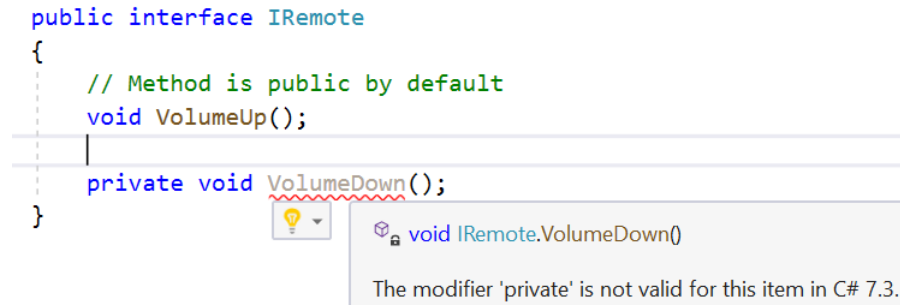
```
class Program
{
    static void Main(string[] args)
    {
        IRemote remote = new IRemote();
        Console.ReadLine();
    }
}
```

interface c_sharp_interface.IRemote

Cannot create an instance of the abstract class or interface 'IRemote'

- You don't need to specify any access modifiers for members defined in an interface. They are all 'Public' by default but you should not explicitly define it as 'public' otherwise you will get a compiler error.

```
public interface IRemote
{
    // Method is public by default
    void VolumeUp();
    private void VolumeDown();
}
```

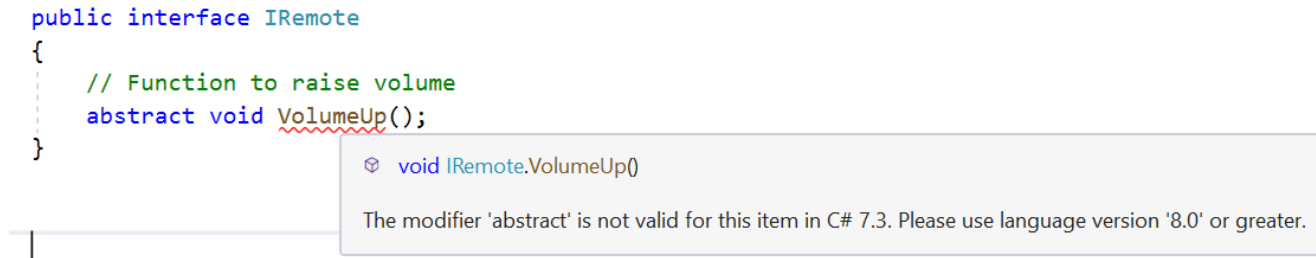


The screenshot shows a code editor with a C# interface definition. The method `VolumeDown()` is marked as `private`, which is invalid for an interface. A tooltip displays the error: "The modifier 'private' is not valid for this item in C# 7.3."

If you declare a method as 'private', compiler will show you an error as shown in the picture.

- If you define something as 'abstract', compiler will show you an error.

```
public interface IRemote
{
    // Function to raise volume
    abstract void VolumeUp();
}
```



The screenshot shows a code editor with a C# interface definition. The method `VolumeUp()` is marked as `abstract`, which is invalid for an interface. A tooltip displays the error: "The modifier 'abstract' is not valid for this item in C# 7.3. Please use language version '8.0' or greater."

So, no access modifiers or no static keywords works inside an interface.

Now, we have following interface. As you can see Interfaces provide no implementation just definitions. Unlike Abstract classes, you can not provide default behaviour of properties or methods in an interface.

```
public interface IRemote
{
    int MaxVolume { get; }

    // Function to raise volume
    void AdjustVolume();
}
```

- Interface is nothing but a contract which enforces derived classes to implement all the functionality defined in that interface. Any class or a struct implementing an interface must implement all defined parts of the interface.

```
namespace c_sharp_interface
{
    class Tv : IRemote
    {
        public int MaxVolume => 60;

        public void AdjustVolume()
        {
            Console.WriteLine("Adjusting Volume using volume button.\n");
        }
    }
}
```

```
namespace c_sharp_interface
{
    class MusicPlayer : IRemote
    {
        public int MaxVolume => 100;

        public void AdjustVolume()
        {
            Console.WriteLine("Adjusting Volume using voice commands.\n");
        }
    }
}
```

As you can see, both classes 'TV' and 'MusicPlayer' are implementing 'IRemote' interface. Both classes are providing their own implementation of properties and functions that are defined in that interface.

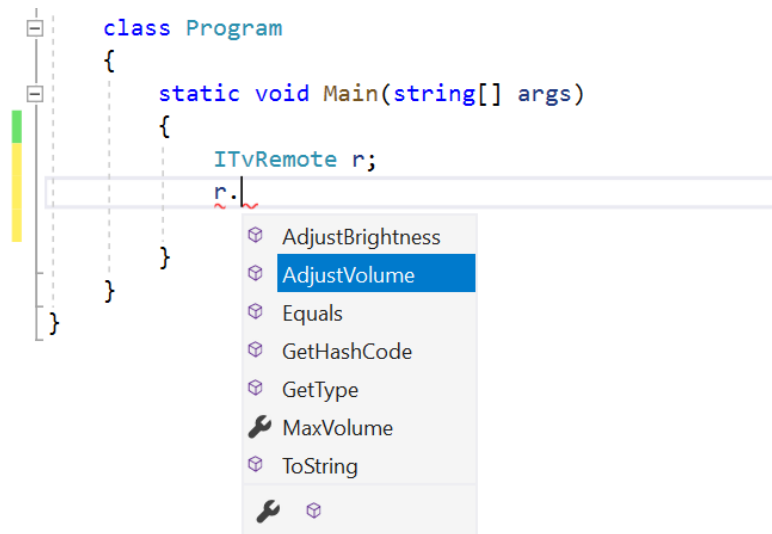
- Interfaces supports Inheritance.

```
public interface IRemote
{
    int MaxVolume { get; }

    // Function to raise volume
    void AdjustVolume();
}

public interface ITvRemote: IRemote
{
    void AdjustBrightness();
}
```

You can have another interface 'ITvRemote' which inherits from 'IRemote' interface. You will still be able to access methods of 'IRemote' interface.



Now, I will make changes to my Tv class. Tv class will implement 'ITvRemote' instead of 'IRemote'.

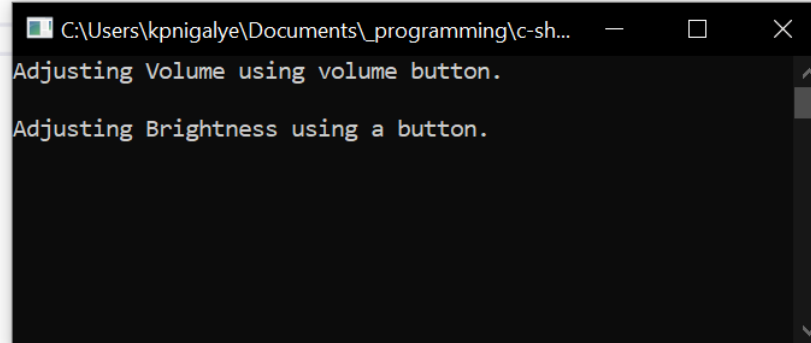
```
class Tv : ITvRemote
{
    public int MaxVolume => 60;

    public void AdjustBrightness()
    {
        Console.WriteLine("Adjusting Brightness using a button.\n");
    }

    public void AdjustVolume()
    {
        Console.WriteLine("Adjusting Volume using volume button.\n");
    }
}
```

As you can see, 'Tv' class implements properties and functions defined in 'ITvRemote' as well as 'IRemote' as 'ITvRemote' inherits from 'IRemote'.

```
namespace c_sharp_interface
{
    class Program
    {
        static void Main(string[] args)
        {
            Tv tv = new Tv();
            tv.AdjustVolume();
            tv.AdjustBrightness();
        }
    }
}
```



A screenshot of a Windows console window. The title bar shows the file path 'C:\Users\kpnigalye\Documents_programming\c-sh...'. The console output consists of two lines: 'Adjusting Volume using volume button.' followed by a blank line, and then 'Adjusting Brightness using a button.'.

But 'MediaPlayer' class still shows only the method defined in 'IRemote' interface. Compiler will not recognize the 'AdjustBrightness' method as 'MediaPlayer' is not inheriting from 'ITvRemote' interface.

Check out all the function calls related these classes below.

```
namespace c_sharp_interface
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tv Remote");
            Console.WriteLine("=====\n");
            Tv tv = new Tv();
            tv.AdjustVolume();
            tv.AdjustBrightness();

            Console.WriteLine();

            Console.WriteLine("Music Player Remote");
            Console.WriteLine("=====\n");
            MediaPlayer musicPlayer = new MediaPlayer();
            musicPlayer.AdjustVolume();
        }
    }
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp\c-sh...
Tv Remote
=====

---> Adjusting Volume using volume button.

---> Adjusting Brightness using a button.

Music Player Remote
=====

---> Adjusting Volume using voice commands.
```

```
class Program
{
    static void Main(string[] args)
    {
        MediaPlayer musicPlayer = new MediaPlayer();
        musicPlayer.AdjustVolume();

        musicPlayer.AdjustBrightness();

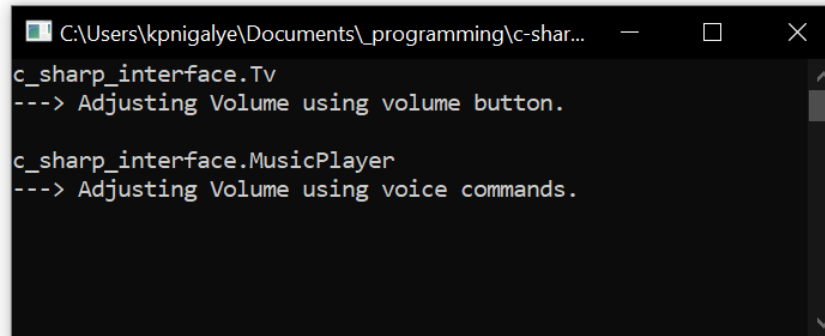
        Console.ReadLine();
    }
}
```

You can create a function taking 'IRemote' as a parameter and pass any class object which inherits from 'IRemote'.

```
namespace c_sharp_interface
{
    class Program
    {
        static void Main(string[] args)
        {
            ChangeVolume(new Tv());
            ChangeVolume(new MusicPlayer());

            Console.ReadLine();
        }

        private static void ChangeVolume(IRemote remote)
        {
            // Check Type of the object using Reflection
            Console.WriteLine(remote.GetType());
            remote.AdjustVolume();
        }
    }
}
```



```
C:\Users\kpnigalye\Documents\_programming\c-shar...
c_sharp_interface.Tv
---> Adjusting Volume using volume button.

c_sharp_interface.MusicPlayer
---> Adjusting Volume using voice commands.
```


But if you declare 'ITvRemote' as a parameter and pass 'MusicPlayer' class object, then compiler will not allow you to run your code as MusicPlayer does not implement 'ITvRemote' interface.

```
namespace c_sharp_interface
{
    class Program
    {
        private static void ChangeVolume(ITvRemote remote)
        {
            // Check Type of the object using Reflection
            Console.WriteLine(remote.GetType());
            remote.AdjustVolume();
        }

        static void Main(string[] args)
        {
            ChangeVolume(new Tv());
            ChangeVolume(new MusicPlayer());
            Console.ReadLine();
        }
    }
}
```

MusicPlayer.MusicPlayer()

Argument 1: cannot convert from 'c_sharp_interface.MusicPlayer' to 'c_sharp_interface.ITvRemote'

[Show potential fixes](#) (Alt+Enter or Ctrl+.)

- C-Sharp supports multiple inheritance of interfaces.

Let's define an interface named 'ITuner'.

```
public interface ITuner
{
    void EnableAudioTuner();
}
```

You can make 'MusicPlayer' class extend 'ITuner' interface as shown below. So now 'MusicPlayer' class implements two interface 'IRemote' and 'ITuner' as shown below.

```
namespace c_sharp_interface
{
    class MusicPlayer : IRemote, ITuner
    {
        public int MaxVolume => 100;

        public void AdjustVolume()
        {
            Console.WriteLine("---> Adjusting Volume using voice commands.\n");
        }

        public void EnableAudioTuner()
        {
            Console.WriteLine("---> Audio tuner Enabled.\n");
        }
    }
}
```

You can define 'AdjustVolume' method of 'IRemote' interface as 'virtual' in the derived class 'MusicPlayer' implementation. Now you can have a class named 'PremiumMusicPlayer' which inherits from 'MusicPlayer'.

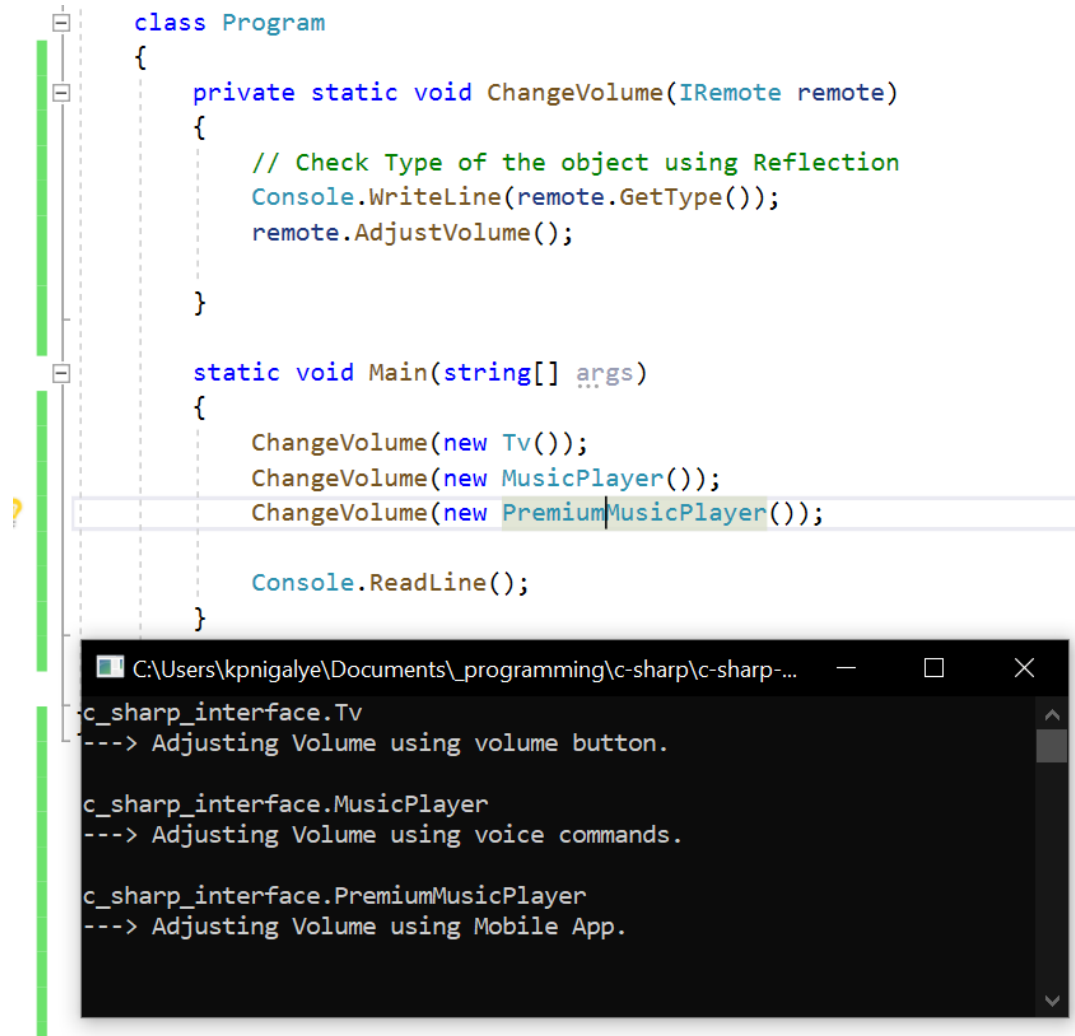
```
namespace c_sharp_interface
{
    class MusicPlayer : IRemote, ITuner
    {
        public int MaxVolume => 100;

        public virtual void AdjustVolume()
        {
            Console.WriteLine("---> Adjusting Volume using voice commands.\n");
        }

        public void EnableAudioTuner()
        {
            Console.WriteLine("---> Audio tuner Enabled.\n");
        }
    }

    class PremiumMusicPlayer: MusicPlayer
    {
        public override void AdjustVolume()
        {
            Console.WriteLine("---> Adjusting Volume using Mobile App.\n");
        }
    }
}
```

Here is the output of this function.



The image shows a C# program in a code editor and its output in a console window. The code defines a `Program` class with a `ChangeVolume` method that uses reflection to check the type of an `IRemote` object and call its `AdjustVolume` method. The `Main` method calls `ChangeVolume` for `Tv`, `MusicPlayer`, and `PremiumMusicPlayer` objects. The console output shows the type of each object and the corresponding volume adjustment message.

```
class Program
{
    private static void ChangeVolume(IRemote remote)
    {
        // Check Type of the object using Reflection
        Console.WriteLine(remote.GetType());
        remote.AdjustVolume();
    }

    static void Main(string[] args)
    {
        ChangeVolume(new Tv());
        ChangeVolume(new MusicPlayer());
        ChangeVolume(new PremiumMusicPlayer());

        Console.ReadLine();
    }
}
```

C:\Users\kpnigalye\Documents_programming\c-sharp\c-sharp-...
c_sharp_interface.Tv
---> Adjusting Volume using volume button.

c_sharp_interface.MusicPlayer
---> Adjusting Volume using voice commands.

c_sharp_interface.PremiumMusicPlayer
---> Adjusting Volume using Mobile App.

- If two interfaces have the method with same name, you can explicitly specify 'interface' along with method name to avoid ambiguous implementations.

```
public interface IRemote
{
    int MaxVolume { get; }
    string Device { get; }

    event Action<string> OnVolumeChange;

    void AdjustVolume();
    void PowerOn();
    void PowerOff();

    void CommonFunction();
}
```

```
public interface ITuner
{
    void EnableAudioTuner();

    // Common function
    void CommonFunction();
}
```

These methods will be implemented as shown in the figure below.

```
#region CommonFunctions
void ITuner.CommonFunction()
{
}

void IRemote.CommonFunction()
{
}
#endregion
```