When To Use Abstract Class and Interface In Real Projects

[Download Free .NET & JAVA Files API](https://www.c-sharpcorner.com/AdRedirector.aspx?AdId=133&target=https://www.e-iceblue.com/download/free-products.html?aff_id=100" \t "_blank)

**Introduction**This is now an important question asked by an interviewer in the interview. I have found many solutions on the Internet regarding "**When to use interface and abstract method?**". Lately, I thought about writing an article on simplifying my experience and what I learned. If you check both the things, they seem to be very similar, which makes a confusion in the mind while answering these questions.  
  
Here, in this article, I will try to explain these things.  
  
**Abstract Class**  
  
Before giving the introduction about abstract class, I will try to explain the abstract method, first.  
  
**Abstract Method**  
A method without any particular body is known as an abstract method. These methods contain only declaration of the method.To declare an abstract method, we have to use abstract modifier on the method.The class, under which these abstract methods are defined, is referred to as an abstract class, and this also has to be declared using abstract modifier.The implementation of abstract method is done by a derived class. When the abstract class inherits the derived class, the derived class must implement the abstract methods using override keyword in it.

**Abstract Class** An abstract class is a special class that contains both abstract and non-abstract members in it.

**Example**

1. **public** **abstract** **class** Cars
2. {
4. //Abstract Methods
5. **public** **abstract** **double** price();
6. **public** **abstract** **int** getTotalSeat();
7. }

 Here are some points regarding abstract class.

1. Abstract class can contain abstract members as well as non-abstract members in it.
2. A class can only inherit from one abstract Class.
3. We cannot create object of an abstract class.

**Interface**  
  
It is also user defined type like a class which only contains abstract members in it. These abstract members should be given the implementation under a child class of an interface. A class can be inherited from a class or from an interface.

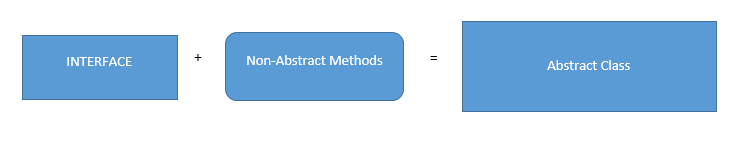
**Points to remember**

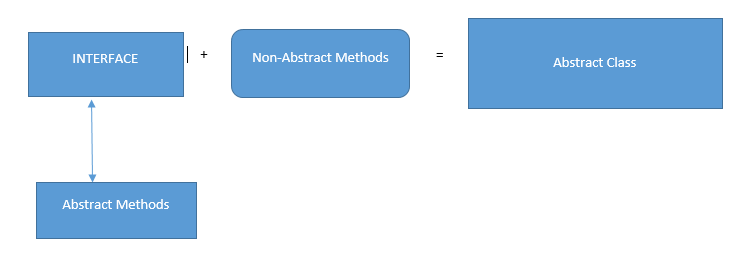
1. Interface contains only abstract methods.
2. We cannot create object of an interface.
3. The default scope for a member in Interface is Public. So, no need to use the Public access specifier in the program.

**NOTE - I**n case of multiple inheritance, use Interface.

From both the definitions, it gets concluded that,

**Figure 1**

  
**Figure 2**

   
Now, I have  class like Hyundai and Toyota which are derived from a parent class called Car. Here, I have the car methods as follow:

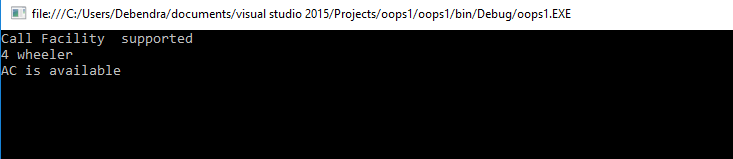
1. **public**  **class** Cars
2. {
4. **public** **string** Wheel()
5. {
6. **return** "4 wheeler";
8. }
9. **public** **string** CheckAC()
10. {
11. **return** "AC is available";
12. }
13. **public** **string** CallFacility()
14. {
15. **return** "Call Facility  supported";
16. }
18. }

Here, I have 2 types of cars which are inherited from Cars.

1. **using** oops1;
2. **using** System;
3. **using** System.Collections.Generic;
4. **using** System.Linq;
5. **using** System.Text;
6. **using** System.Threading.Tasks;
8. **namespace** oops1
9. {
10. **public** **class** Hyundai : Cars
11. {
13. **static** **void** Main(**string**[] args)
14. {
15. Hyundai dust = **new** Hyundai();
17. Console.WriteLine(dust.CallFacility());
18. Console.WriteLine(dust.Wheel());
19. Console.WriteLine(dust.CheckAC());
20. Console.ReadLine();

23. }
24. }

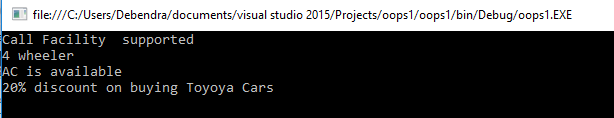
27. }

Now, it runs as expected and gives the following output .  
  
   
  
Similarly, as Toyota is a car and it also inherits from Cars class.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** oops1
8. {
9. **public** **class** Toyota : Cars
10. {
11. **public** **string** DiscountPrice()
12. {
13. **return** "20% discount on buying Toyoya Cars";
14. }

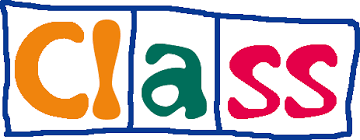
17. **static** **void** Main(**string**[] args)
18. {
19. Toyota Toy = **new** Toyota();
21. Console.WriteLine(Toy.CallFacility());
22. Console.WriteLine(Toy.Wheel());
23. Console.WriteLine(Toy.CheckAC());
24. Console.WriteLine(Toy.DiscountPrice());
26. Console.ReadLine();

29. }
30. }
31. }

  
  
We need some methods which are common to both the classes but their implementation is different.

* **PRICE**()- Both have price but different price.
* **TotalSeat**()- Both have total seats but different no. of seats.
* **colors**()- Both cars are of different colour.

So, here are the options for implementing these methods in both classes.

* Can I go for a normal class?
* Should I use interface here?
* Can I use an abstract class?  
    
  
* If we are taking class, then we can only write normal methods having common implementation there. But, this will not satisfy our requirement because we need separate implementations in both the classes. Thus, we will not go forward with Class.  
  

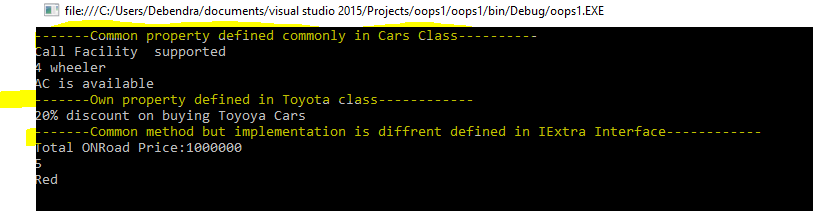
If we go for interface, we can achieve our goal but it will be like this. First, I will declare n interface, as follows.

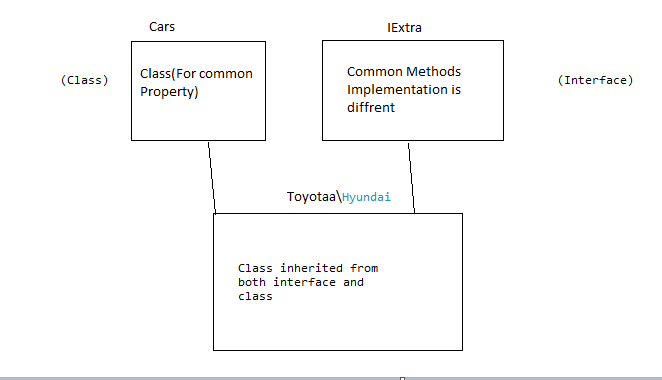
1. **interface** IExtra
2. {
3. **double** price();
4. **int** getTotalSeat();
5. **string** colors();
6. }

Now, let me first inherit the Toyota class from Cars class and IExtra interface class to achieve our goal.  
  
The code is given below.

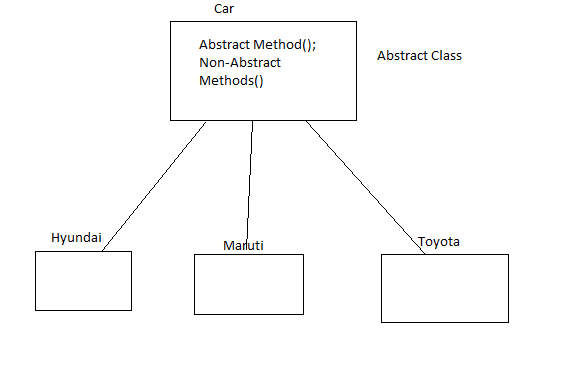
1. **namespace** oops1
2. {
3. **public** **class** Toyota : **Cars**,**IExtra**
4. {
5. **public** **string** DiscountPrice()
6. {
7. **return** "20% discount on buying Toyoya Cars";
8. }
9. **public**  **double** price()
10. {
11. **return** 1000000.00;
12. }
13. **public**  **int** getTotalSeat()
14. {
15. **return** 5;
16. }
17. **public** **string** colors()
18. {
19. **return** "Red";
20. }
22. **static** **void** Main(**string**[] args)
23. {
24. Toyota Toy = **new** Toyota();
26. Console.WriteLine(Toy.CallFacility());
27. Console.WriteLine(Toy.Wheel());
28. Console.WriteLine(Toy.CheckAC());
29. Console.WriteLine(Toy.DiscountPrice());
30. Console.WriteLine("Total ONRoad Price:"+ Toy.price());
31. Console.WriteLine(Toy.getTotalSeat());
32. Console.WriteLine(Toy.colors());
34. Console.ReadLine();

37. }
38. }

    
  
So, the sketch diagram of this implementation will be like this.



**Abstract Class**  
   
Now, we will see here how we can solve our problem using Abstract Class.  
  
1. Define an abstract class as Car.   
  
2. Put all the common functionality in simple methods and all the methods whose implementation is different but name is same. Make them  Abstract method.

Here is my Car abstract class.  
  


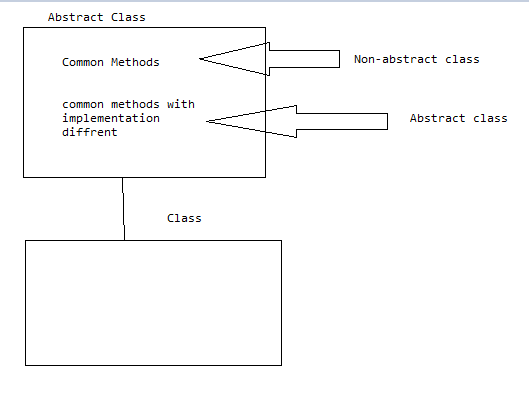
1. **using** System;
2. **using** System.Collections;
3. **using** System.Collections.Generic;
5. **namespace** oops1
6. {
7. **public** **abstract**  **class** Cars
8. {
10. //put all the common functions but diffrent implementation in  abstract method.
11. **public** **abstract** **double** price();
12. **public** **abstract** **int** getTotalSeat();
13. **public** **abstract** **string** colors();
15. //put all the common property in normal class
16. **public** **string** Wheel()
17. {
18. **return** "4 wheeler";
20. }
21. **public** **string** CheckAC()
22. {
23. **return** "AC is available";
24. }
25. **public** **string** CallFacility()
26. {
27. **return** "Call Facility  supported";
28. }
29. }
30. }

Now, here is my Toyota class which is derived from Cars abstract class.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** oops1
8. {
9. **public** **class** Toyota : **Cars**
10. {
11. **public** **string** DiscountPrice()
12. {
13. **return** "20% discount on buying Toyoya Cars";
14. }
15. **public** **override** **double** price()
16. {
17. **return** 1000000.00;
18. }
19. **public** **override** **int** getTotalSeat()
20. {
21. **return** 5;
22. }
23. **public** **override** **string** colors()
24. {
25. **return** "Red";
26. }
28. **static** **void** Main(**string**[] args)
29. {
30. Toyota Toy = **new** Toyota();
31. Console.WriteLine("-------Common property defined commonly in Cars Class----------");
33. Console.WriteLine(Toy.CallFacility());
34. Console.WriteLine(Toy.Wheel());
35. Console.WriteLine(Toy.CheckAC());
36. Console.WriteLine("-------Own property defined in Toyota class------------");
37. Console.WriteLine(Toy.DiscountPrice());
38. Console.WriteLine("-------Common method but implementation is diffrent defined in IExtra Interface------------");
39. Console.WriteLine("Total ONRoad Price:"+ Toy.price());
40. Console.WriteLine(Toy.getTotalSeat());
41. Console.WriteLine(Toy.colors());
43. Console.ReadLine();

46. }
47. }

50. }

And thus, the result will be the same.  
  
   
**Conclusion**When we have the requirement of a class that contains some common properties or methods with some common properties whose implementation is different for different classes, in that situation, it's better to use Abstract Class then Interface.  
  
**Abstract classes** provide you the flexibility to have certain concrete methods and some other methods that the derived classes should implement. On the other hand, if you use interfaces, you would need to implement all the methods in the class that extends the interface. An abstract class is a good choice if you have plans for future expansion.

Now, I will explain the second use of Abstract Class here.

Imagine, we have taken a normal parent class as Cars where we have only defined the common methods. And, my Toyota class is derived from the Cars class. This will look like the below code.

1. **public** **class** Cars
2. {
4. **public** **string** Wheel()
5. {
6. **return** "4 wheeler";
8. }
9. **public** **string** CheckAC()
10. {
11. **return** "AC is available";
12. }
13. **public** **string** CallFacility()
14. {
15. **return** "Call Facility  supported";
16. }
18. }

 And, here is the Toyota class after inheriting from Cars.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** oops1
8. {
9. **public** **class** Toyota : Cars
10. {
11. **public** **string** DiscountPrice()
12. {
13. **return** "20% discount on buying Toyoya Cars";
14. }

17. **static** **void** Main(**string**[] args)
18. {

21. Console.ReadLine();

24. }
25. }

28. }

Here, users know only the Toyota class, its methods, and properties. The user can access the Cars class property and method by creating the object of Toyota class because it is a child class.  
  
Now, the main demerit of this kind of Implementation is that it allows the user to create the object of the Parent class (which the user should not be aware of and this is not his strategy). Let's see what happens when the user creates the object.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** oops1
8. {
9. **public** **class** Toyota : Cars
10. {
11. **public** **string** DiscountPrice()
12. {
13. **return** "20% discount on buying Toyoya Cars";
14. }

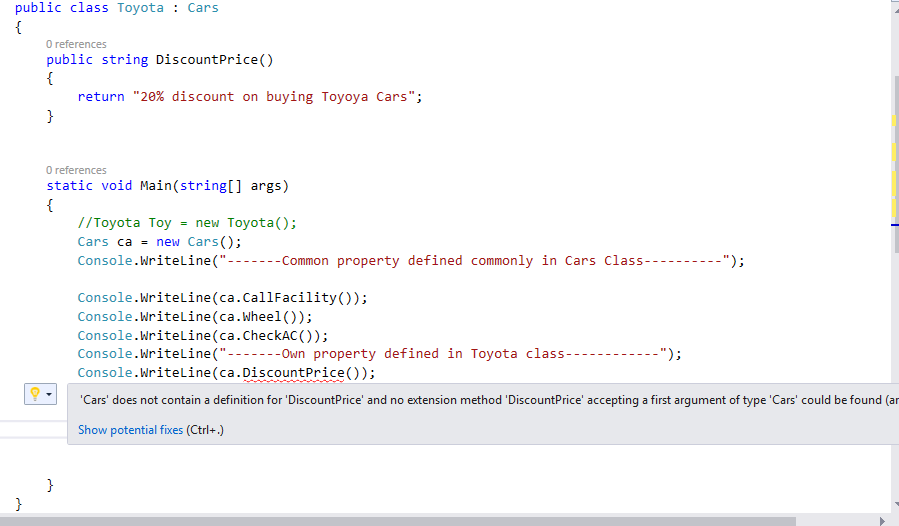
17. **static** **void** Main(**string**[] args)
18. {
19. //Toyota Toy = new Toyota();
20. Cars ca = **new** Cars();
21. Console.WriteLine("-------Common property defined commonly in Cars Class----------");
23. Console.WriteLine(ca.CallFacility());
24. Console.WriteLine(ca.Wheel());
25. Console.WriteLine(ca.CheckAC());
27. Console.WriteLine("-------Own property defined in Toyota class------------");
29. Console.WriteLine(ca.DiscountPrice());

32. Console.ReadLine();

35. }
36. }

39. }

As we know, we won't be able to access the child class methods using parent class object, this Cars object will not access the DiscountPrice() Method because it is a child class method. This implementation will not satisfy our requirement. So, even if the user will create an object of Parent class, we need to check them because using that object, he can't able to access the child class method.

   
  
So, the main problem we face here is,  
  
As the user is unaware of parent class, he is able to create the object of parent class and unable to access the child class method using parent class. So, to restrict this, we have to do the following things-

* We should not give the permission to create the object of parent class in child class
* We can only allow the user to create the child class object to access both parent class methods and child class methods.  
    
  Simple to do this Create the base class as an Abstract class and that will solve our problems as follow.  
  1. **using** System;
  2. **using** System.Collections;
  3. **using** System.Collections.Generic;
  5. **namespace** oops1
  6. {
  7. **public** **abstract**  **class** Cars
  8. {

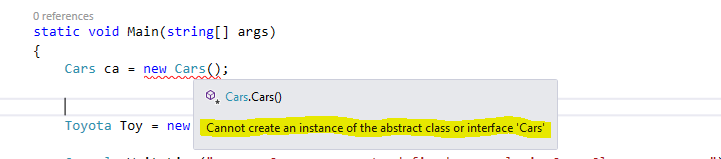
  11. **public** **string** Wheel()
  12. {
  13. **return** "4 wheeler";
  15. }
  16. **public** **string** CheckAC()
  17. {
  18. **return** "AC is available";
  19. }
  20. **public** **string** CallFacility()
  21. {
  22. **return** "Call Facility  supported";
  23. }




  29. }

  32. }

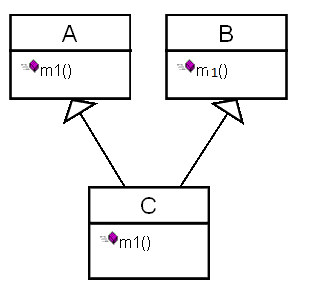
Even if you want to create the object of parent class, it shows the following error.

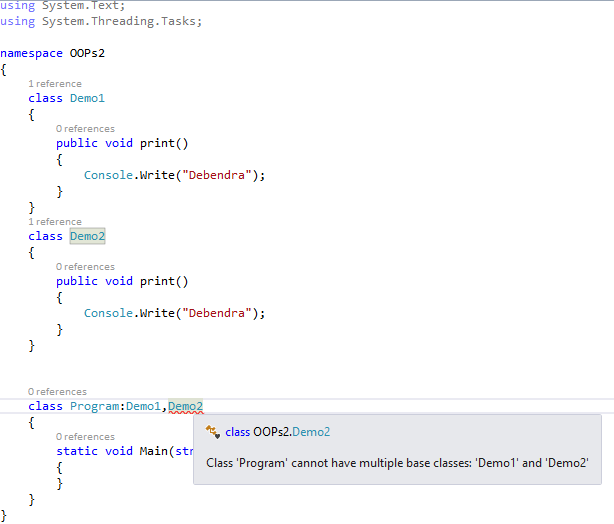
  
  
So, this will solve our problem.  
  
**Conclusion**So, in those cases, we will use abstract class where we want to restrict the user from creating the object of parent class because by creating object of parent class, you can't call child class methods. So, the developer has to restrict accidental creation of parent class object by defining it as abstract class.  
  
So, I think, in these ways, we can use abstract class in our real time project.  
  
**When To Use Interface In Real Time Project?**

**INTERFACE**  
It is also user-defined type like a **class**which only contains abstract members in it and these abstract members should be given implementation under a child class of an interface. A class can inherit from a class or from an interface.

1. **interface** I1
2. {
3. **void** MethodToImplement();
4. }

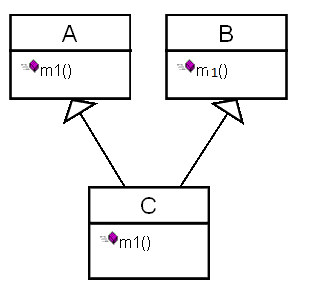
In your daily programming, you are using a lot of interfaces knowingly or unknowingly. If you are using a List, Dictionary, or Hash Table etc in your project, then you are indirectly using interface in your project because all collections are derived from an Interface, **IEnumerable.**  
Here, I will not discuss about the uses of Interface. I will discuss when can we use our interface in C#.  
  
      Before explaining in details, I will ask you to focus on the following points.

* In C++, we have a concept of Multiple Inheritance where we find a serious problem called Diamond Problem.
* Thus, in C#, multiple inheritance is not supported. When there is a situation like multiple inheritance, use Interface.  
    
  

  
  
The solution of the multiple inheritance can be provided by Interface. So, we can do this example using Interface as follows. In this way, the diamond problem of Inheritance is solved.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** OOPs2
8. {
9. **interface** Demo1
10. {
11. **void** print();
13. }
14. **interface** Demo2
15. {
16. **void** print();
18. }

21. **class** Program:Demo1,Demo2
22. {
23. **void** Demo1.print()
24. {
25. Console.WriteLine("Debendra");
26. }
27. **void** Demo2.print()
28. {
29. Console.WriteLine("Debendra");
30. }
31. **static** **void** Main(**string**[] args)
32. {
33. Program p = **new** Program();
34. ((Demo2)p).print();
35. Console.ReadLine();
37. }
38. }
39. }



Now, I will narrate a real time scenario where we can use interface. I will go for the same example that I have done earlier.

1. **using** System;
2. **using** System.Collections;
3. **using** System.Collections.Generic;
5. **namespace** oops1
6. {
7. **public**   **class** Cars
8. {
10. **public** **string** Wheel()
11. {
12. **return** "4 wheeler";
14. }
15. **public** **string** CheckAC()
16. {
17. **return** "AC is available";
18. }
19. **public** **string** CallFacility()
20. {
21. **return** "Call Facility  supported";
22. }




28. }

31. }

Now, here is my Toyota class which is derived from Cars.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** oops1
8. {
9. **public** **class** Toyota : Cars
10. {

13. **static** **void** Main(**string**[] args)
14. {

17. Toyota Toy = **new** Toyota();


21. Console.WriteLine(Toy.CallFacility());
22. Console.WriteLine(Toy.Wheel());
23. Console.WriteLine(Toy.CheckAC());


27. Console.ReadLine();

30. }
31. }

34. }

Here is my Hyundai class which is derived from Cars.

1. **using** oops1;
2. **using** System;
3. **using** System.Collections.Generic;
4. **using** System.Linq;
5. **using** System.Text;
6. **using** System.Threading.Tasks;
8. **namespace** oops1
9. {
10. **public** **class** Hyundai:Cars
11. {

14. **static** **void** Main(**string**[] args)
15. {
16. Hyundai dust = **new** Hyundai();
18. Console.WriteLine(dust.CallFacility());
19. Console.WriteLine(dust.Wheel());
20. Console.WriteLine(dust.CheckAC());

23. Console.ReadLine();

26. }
27. }

30. }

A new feature for the Hyundai car is Introduced called GPS which is not supported in Toyota cars. Then, how can we implement and which way we can implement these?  
  
Here, we have certain options like

* Go for a new class defining the GPS method and inherit it to the Hyundai Class.
* Go for an abstract class and define GPS method and inherit it on Hyundai class and implement the GPS method there.
* Directly create a method in Hyundai class and consume it.
* Go for Interface  
  .  
  **CASE 1 - By Using simple class**  
  Let's find what will happen if we use a class there, and declare a method as GPS and try to inherit in Hyundai class.  
  Created a new class as "NewFeatures, as shown below.
  1. **class** NewFeatures
  2. {
  3. **public** **void** GPS()
  4. {
  5. Console.WriteLine("GPS supported");
  6. }
  7. }

Now, inherit the Hyundai class from NewFeatures. So, if we check this class, it was previously inherited from Cars class and for now, again inherits from NefFeatures class. So, it gives the following error when you try to run the program.

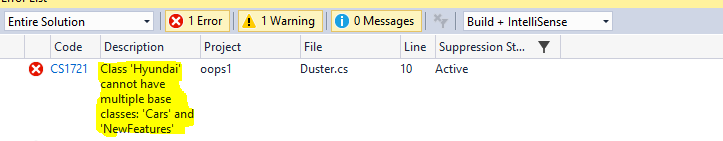
* 1. **using** oops1;
  2. **using** System;
  3. **using** System.Collections.Generic;
  4. **using** System.Linq;
  5. **using** System.Text;
  6. **using** System.Threading.Tasks;
  8. **namespace** oops1
  9. {
  10. **public** **class** Hyundai:Cars,NewFeatures
  11. {

  14. **static** **void** Main(**string**[] args)
  15. {
  16. Hyundai hun = **new** Hyundai();
  18. Console.WriteLine(hun.CallFacility());
  19. Console.WriteLine(hun.Wheel());
  20. Console.WriteLine(hun.CheckAC());

  23. Console.ReadLine();

  26. }
  27. }

  30. }

Now, run the program and find out the error.  
  
  
  
This is simple because C# does not support multiple inheritance.

* **CASE 2 - By using Abstract class**  
    
  Now, go for abstract class and see what happens.
  1. **public** **abstract** **class**  NewFeatures
  2. {
  3. **abstract** **public** **void** GPS();
  5. }

Now, let's try to inherit from abstract class.

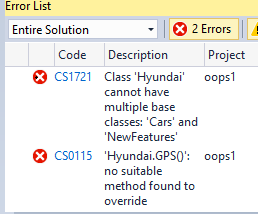
* 1. **using** oops1;
  2. **using** System;
  3. **using** System.Collections.Generic;
  4. **using** System.Linq;
  5. **using** System.Text;
  6. **using** System.Threading.Tasks;
  8. **namespace** oops1
  9. {
  10. **public** **class** Hyundai:Cars,NewFeatures
  11. {
  12. **public**  **override**  **void** GPS()
  13. {
  14. Console.WriteLine("GPS supported.");
  16. }

  19. **static** **void** Main(**string**[] args)
  20. {
  21. Hyundai hun = **new** Hyundai();
  23. Console.WriteLine(hun.CallFacility());
  24. Console.WriteLine(hun.Wheel());
  25. Console.WriteLine(hun.CheckAC());

  28. Console.ReadLine();

  31. }
  32. }

  35. }

So, here is the error I got.  
  


* **CASE 3 - Direct creating a method called GPS() inside Hyundai class**  
  This is very relevant way to use a unique method but it has a small problem. If for any reason we forget to create such common method, then it will not ask to write methods.Today, we are using one unique method so that is OK we can remember and write the method. Suppose, we have hundreds of such common methods and we forget to  write 2 of them then it will run but not give the expected output,so we have to skip the way and go for the fourth one by defining interface.
* **CASE 4 - By defining Interface**  
  This is the most important case and we will see how it will solve our problem.Lets define  the  method in an Interface
  1. **interface  INewFeatures**
  2. **{**
  3. **void GPS();**
  5. **}**

Now inherit the Interface from Cars and see without implementing the GPS() method.

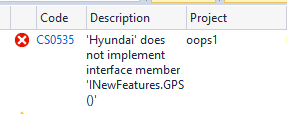
* 1. **using oops1;**
  2. **using System;**
  3. **using System.Collections.Generic;**
  4. **using System.Linq;**
  5. **using System.Text;**
  6. **using System.Threading.Tasks;**
  8. **namespace oops1**
  9. **{**
  10. **public class Hyundai:Cars, INewFeatures**
  11. **{**

  14. **static void Main(string[] args)**
  15. **{**
  16. **Hyundai hun = new Hyundai();**
  18. **Console.WriteLine(hun.CallFacility());**
  19. **Console.WriteLine(hun.Wheel());**
  20. **Console.WriteLine(hun.CheckAC());**

  23. **Console.ReadLine();**

  26. **}**
  27. **}**

  30. **}**

As we have not implemented the method it will show the following error as follow.  
  
****So, the problems which happen in case 3 can be easily solved by using Interface. Now, let's implement the method and see will it solve the problem which arises in Case1 and case2 as follows.

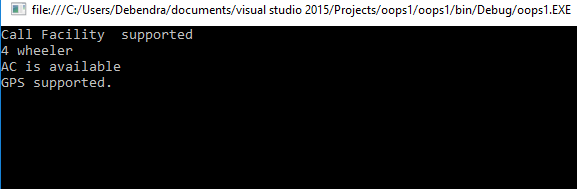
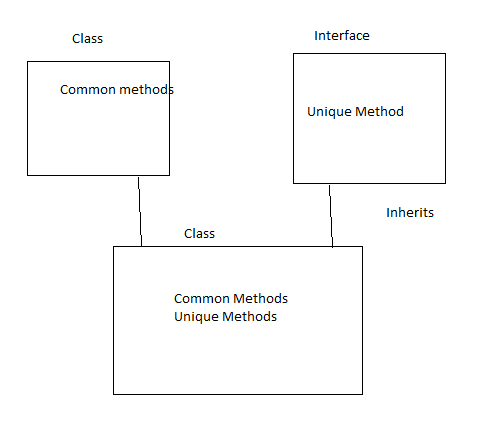
* 1. **using oops1;**
  2. **using System;**
  3. **using System.Collections.Generic;**
  4. **using System.Linq;**
  5. **using System.Text;**
  6. **using System.Threading.Tasks;**
  8. **namespace oops1**
  9. **{**
  10. **public class Hyundai:Cars,INewFeatures**
  11. **{**
  12. **public  void GPS()**
  13. **{**
  14. **Console.WriteLine("GPS supported.");**
  16. **}**

  19. **static void Main(string[] args)**
  20. **{**
  21. **Hyundai hun = new Hyundai();**
  23. **Console.WriteLine(hun.CallFacility());**
  24. **Console.WriteLine(hun.Wheel());**
  25. **Console.WriteLine(hun.CheckAC());**
  26. **hun.GPS();**

  29. **Console.ReadLine();**

  32. **}**
  33. **}**

  36. **}**

****  
  
Thus, it resolves the demerits of all the above 3 cases. So, in this situation, we have to use Interface. Here, I have shown you the sketch diagram.  
****

In this way, we can use abstract class and interface in our project depending on the condition.  
  
Hope this article gives you an idea about when to use these 2 important things.If you have any suggestions, please feel free to comment so that I can include that in this article.