Lesson 8: interface and Abstraction in C#

What is an interface?

These objects implement the interface IPowerPlug

So they can be used with PowerSocket objects

Interface -*system that unrelated entities use to interact*

- So in this example, the PowerSocket doesn't know anything else about the other objects. The objects all depend on Power provided by the PowerSocket, so they implement IPowerPlug, and in so doing they can connect to it.

- **Interfaces** are useful because they provide contracts that objects can use to work together without needing to know anything else about each other.

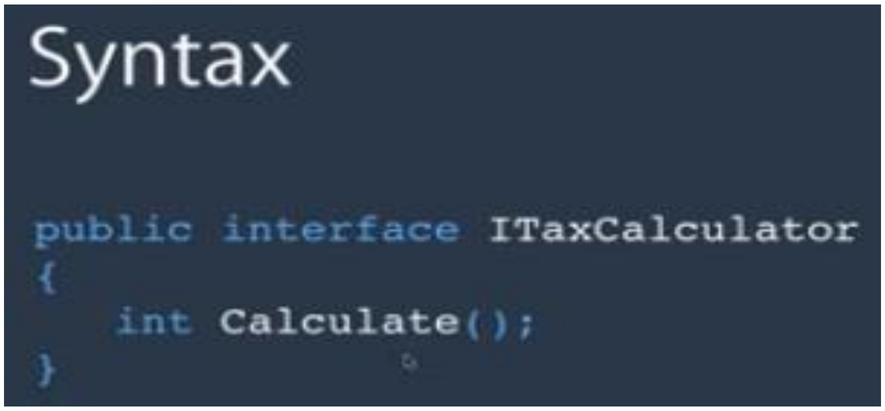
    - **Interfaces -** *are a kind of code contract (Agreement/Template), which must be implemented by a concrete class*.
    - An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):
    - an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract *(method without body).*

- An interface can contain declarations of methods/signatures, properties, indexers, and events. However, it cannot contain instance fields.

Note: Interfaces can contain properties and methods, but not fields

By default, members of an interface are `abstract` and `public`.

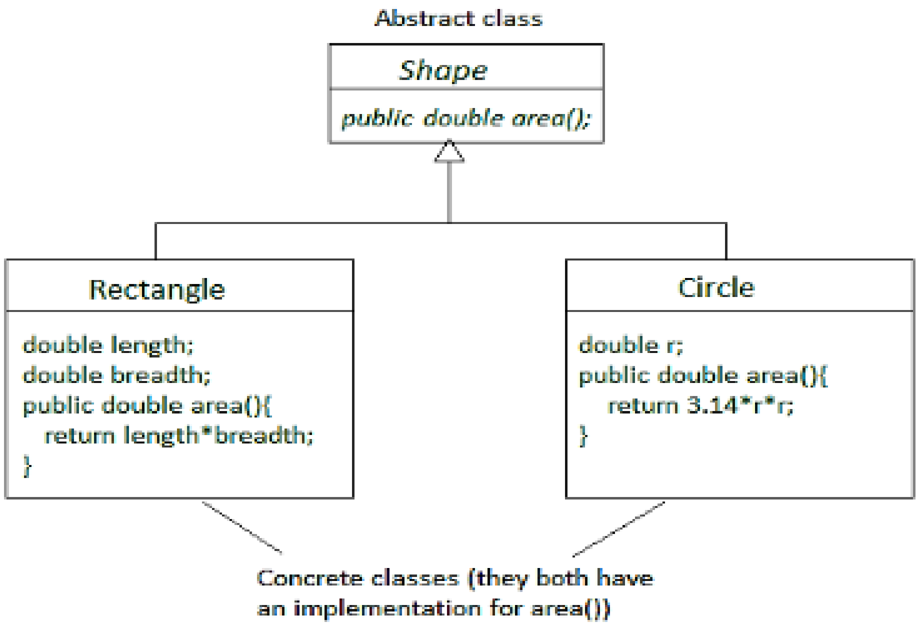
It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.



Syntax

```
public interface ITaxCalculator
{
    int Calculate();
}
```

Unlike classes interfaces have no implementation
Only method called calculate but no //body,no
// cury braces

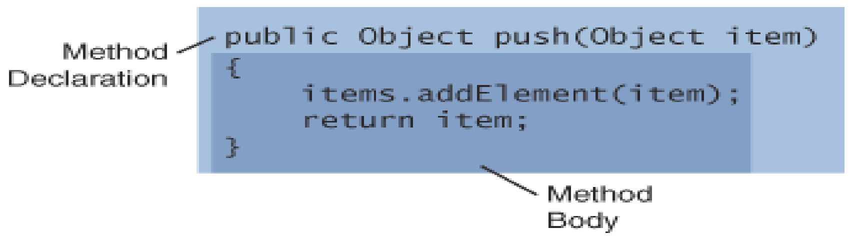


- **Abstract classes** are similar to normal classes, with the difference that they can include abstract methods, which are methods without a body. Abstract classes cannot be instantiated.

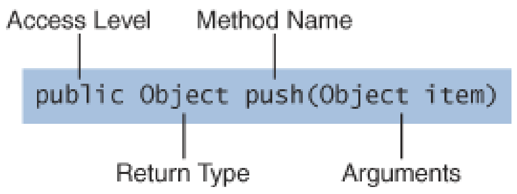
Why use interface?

we need an Interface for the following reasons:

a) **Total Abstraction** - An interface only stores the method signature and not the method definition. *Method Signatures make an Interface achieve complete Abstraction by hiding the method implementation from the user.*



```
public Object push(Object item)
{
    items.addElement(item);
    return item;
}
```



```
public Object push(Object item)
```

b) **Multiple Inheritance** –

Without Interface, the process of multiple inheritances is impossible as the conventional way of inheriting multiple parent classes results in profound ambiguity. *This type of ambiguity is known as the Diamond problem. Interface resolves this issue.*

c) Loose-Coupling-

Loose coupling, on the other hand, means that the components are less dependent on each other and can operate more independently.
Which is better, tight or loose coupling?

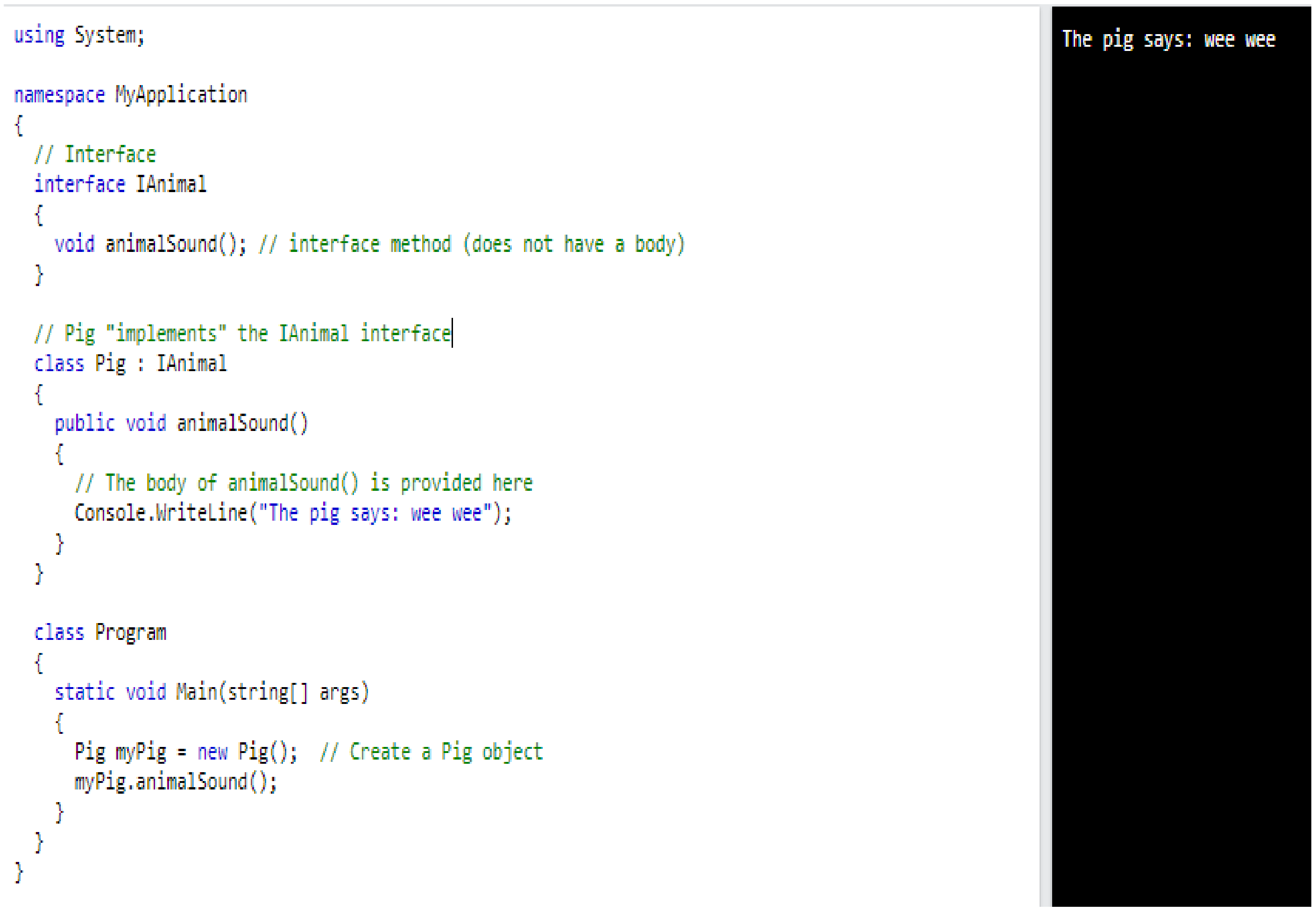
As with everything else in software development, there's not an approach that is "better".
Generally,
loose coupling is preferred for larger or more complex systems, where flexibility, scalability, and maintainability are more important,

while **tight coupling** *is better suited for simpler systems, where the goal is to keep complexity low.*

d) **Flexibility** – swappable components
-Helps implementing design patterns and principles
e) Adaptable to changes
-welcome to changes without breaking
f) **Extensible** –extensible applications using plugins or modules
g) **Readable**- looking at interface you can figure out high level.

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - *the body is provided by the "implement" class*
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

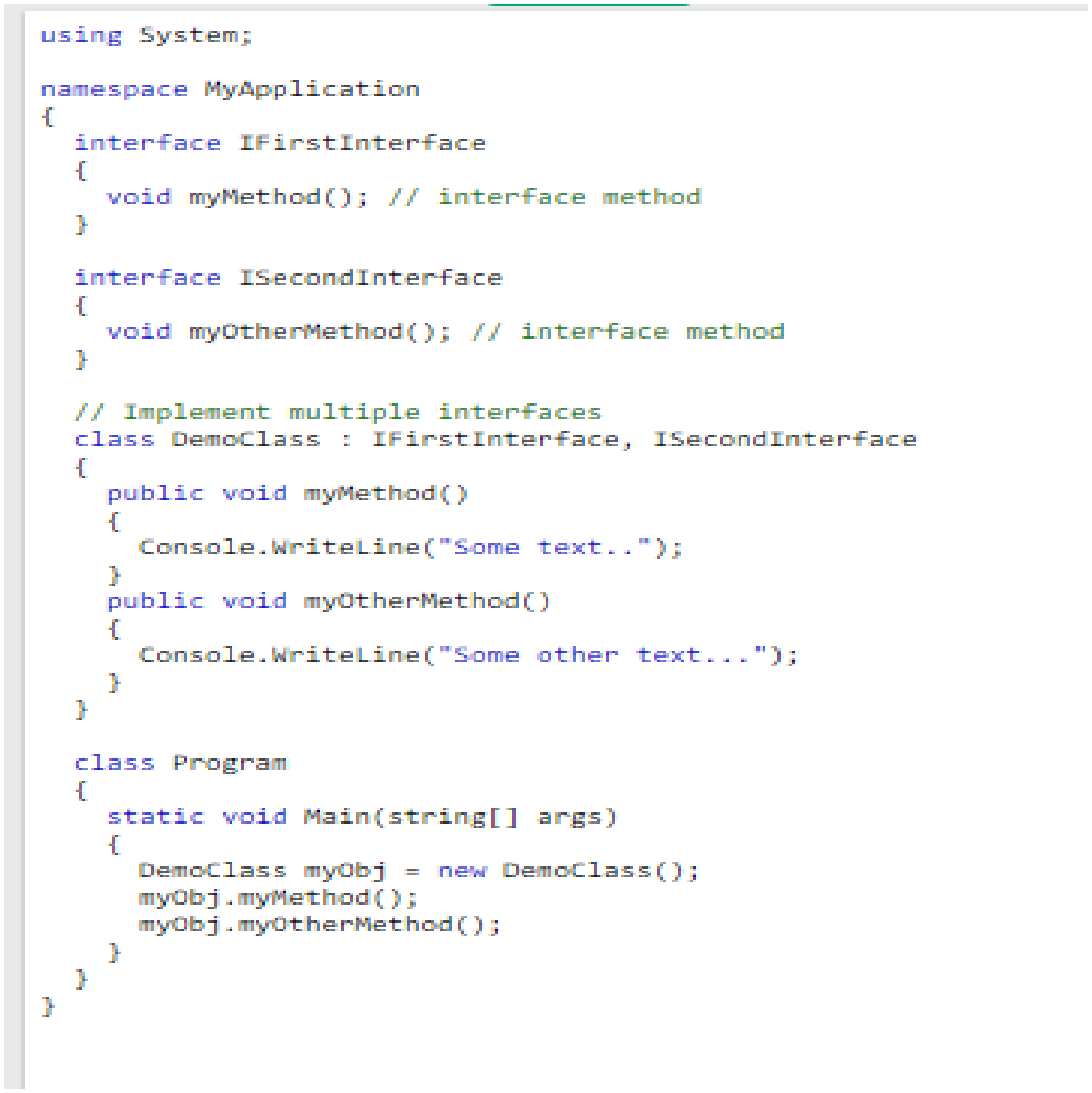


```
using System;

namespace MyApplication
{
  // Interface
  interface IAnimal
  {
    void animalSound(); // interface method (does not have a body)
  }

  // Pig "implements" the IAnimal interface
  class Pig : IAnimal
  {
    public void animalSound()
    {
      // The body of animalSound() is provided here
      Console.WriteLine("The pig says: wee wee");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Pig myPig = new Pig();  // Create a Pig object
      myPig.animalSound();
    }
  }
}
```

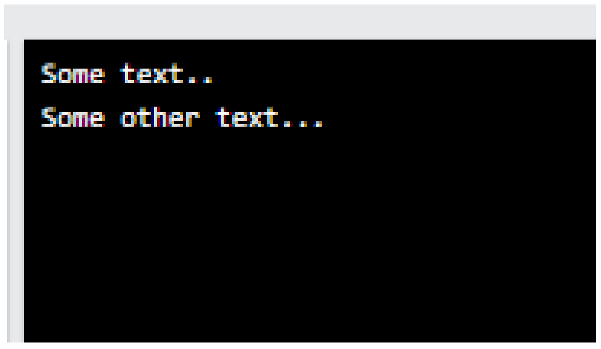
```
The pig says: wee wee
```

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:



```
using System;

namespace MyApplication
{
  interface IFirstInterface
  {
    void myMethod(); // interface method
  }

  interface ISecondInterface
  {
    void myOtherMethod(); // interface method
  }

  // Implement multiple interfaces
  class DemoClass : IFirstInterface, ISecondInterface
  {
    public void myMethod()
    {
      Console.WriteLine("Some text..");
    }
    public void myOtherMethod()
    {
      Console.WriteLine("Some other text...");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      DemoClass myObj = new DemoClass();
      myObj.myMethod();
      myObj.myOtherMethod();
    }
  }
}
```



```
Some text..
Some other text...
```

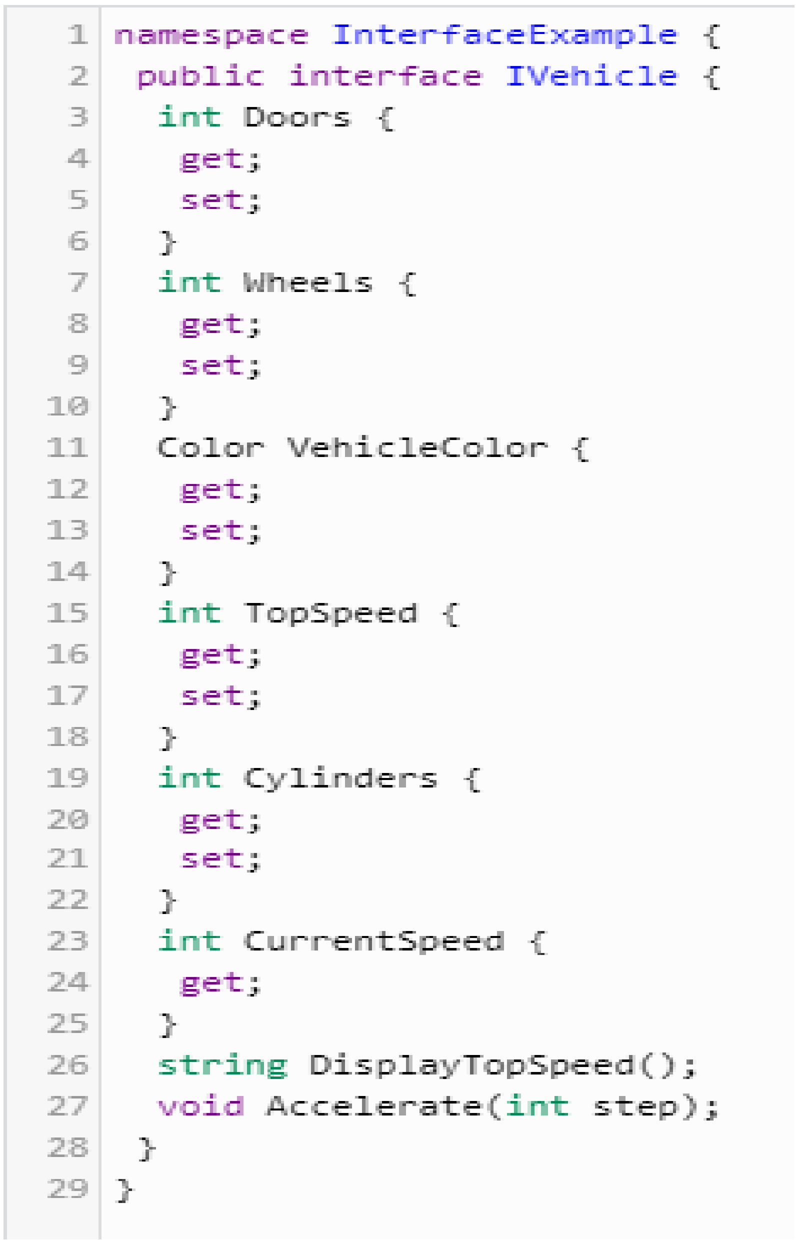
Personal Revision case study

Case study

For example, let's take a vehicle. All vehicles have similar items but are different enough that we could design an interface that holds all the common items of a vehicle. Some vehicles have two wheels, some have four wheels, and some even have one wheel. Though these are differences, they all have things in common: they're all movable, they all have some sort of engine, they all have doors, but each of these items may vary. So we can create an interface of a vehicle that has these properties, then we inherit from that interface to implement it.

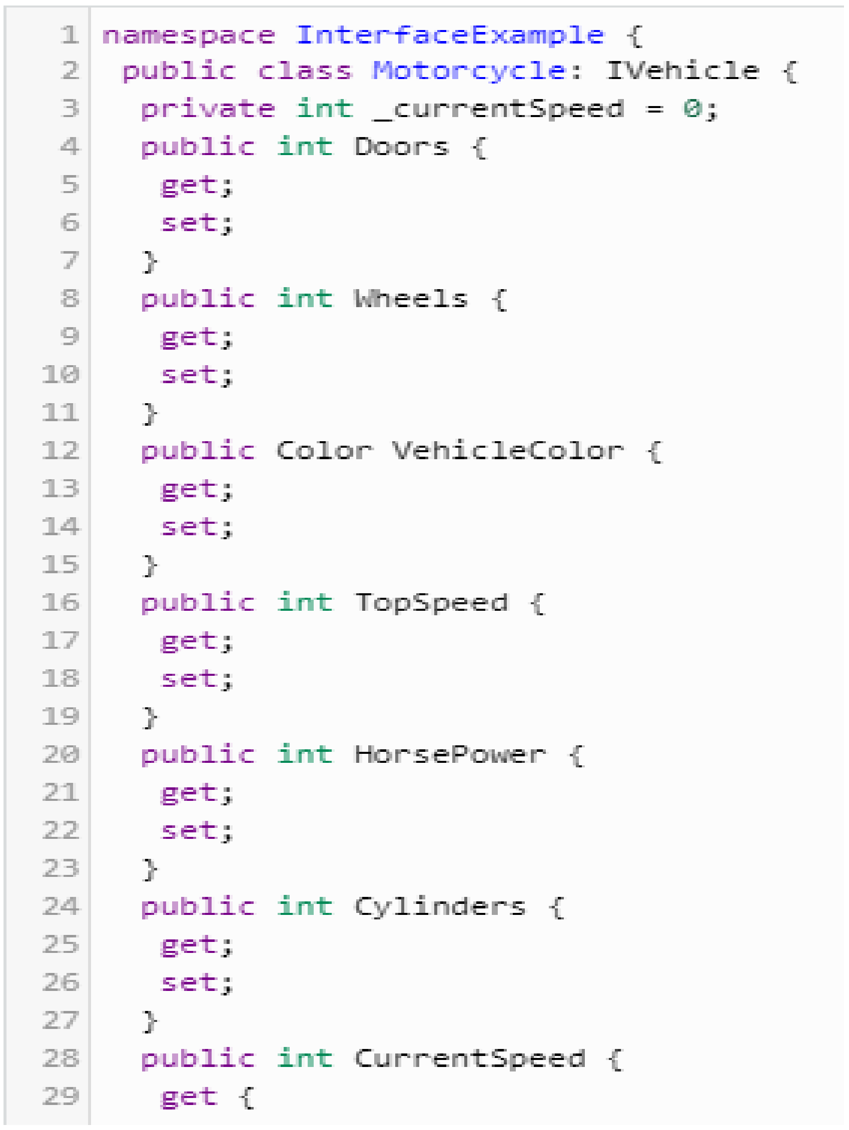
While wheels, doors, and engines are different they all rely on the same interface (I sure hope this is making sense). Interfaces allow us to create nice layouts for what a class is going to implement. Because of the guarantee that the interface gives us, when many components use the same interface it allows us to easily interchange one component for another which is using the same interface. Dynamic programs begin to form easily from this. An interface is a contract that defines the signature of some piece of functionality.

So here's a simple example of an interface and how to implement it. From the above example, we're created an IVehicle interface that looks like this

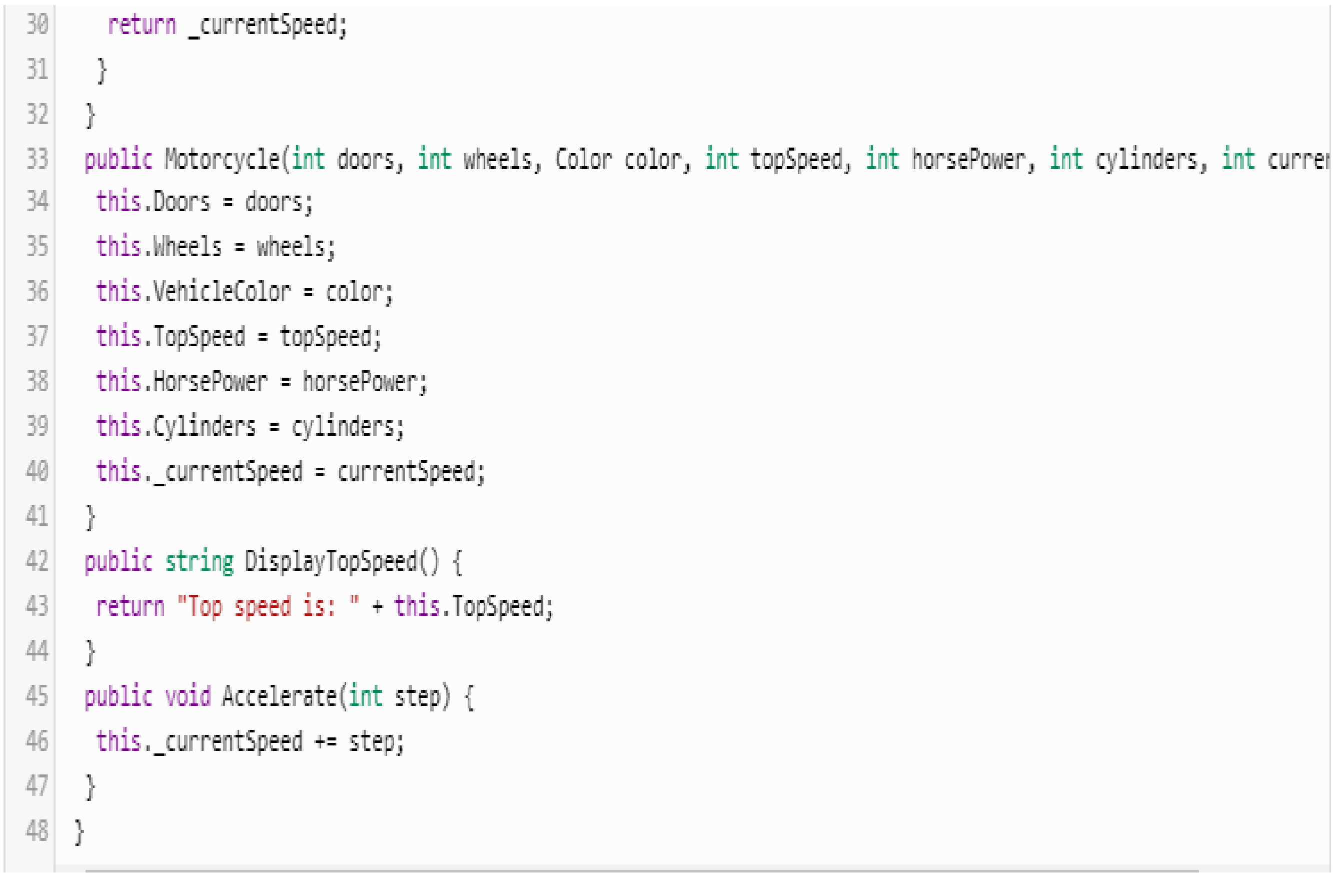


```
namespace InterfaceExample {
 public interface IVehicle {
  int Doors {
   get;
   set;
  }
  int Wheels {
   get;
   set;
  }
  Color VehicleColor {
   get;
   set;
  }
  int TopSpeed {
   get;
   set;
  }
  int Cylinders {
   get;
   set;
  }
  int CurrentSpeed {
   get;
  }
  string DisplayTopSpeed();
  void Accelerate(int step);
 }
}
```

Now we have our vehicle blueprint, and all classes that implement it must implement the items in our interface, whether it be a motorcycle, car, or truck class we know that all of them will contain the same functionality. Now for a sample implementation, in this example, we'll create a Motorcycle class that implements our IVehicle class. This class will contain everything we have defined in our interface.



```
namespace InterfaceExample {
 public class Motorcycle: IVehicle {
  private int _currentSpeed = 0;
  public int Doors {
   get;
   set;
  }
  public int Wheels {
   get;
   set;
  }
  public Color VehicleColor {
   get;
   set;
  }
  public int TopSpeed {
   get;
   set;
  }
  public int HorsePower {
   get;
   set;
  }
  public int Cylinders {
   get;
   set;
  }
  public int CurrentSpeed {
   get {
```



```
    return _currentSpeed;
   }
  }
  public Motorcycle(int doors, int wheels, Color color, int topSpeed, int horsePower, int cylinders, int curre
   this.Doors = doors;
   this.Wheels = wheels;
   this.VehicleColor = color;
   this.TopSpeed = topSpeed;
   this.HorsePower = horsePower;
   this.Cylinders = cylinders;
   this._currentSpeed = currentSpeed;
  }
  public string DisplayTopSpeed() {
   return "Top speed is: " + this.TopSpeed;
  }
  public void Accelerate(int step) {
   this._currentSpeed += step;
  }
 }
```

C# 16 - Interface Basics

https://www.youtube.com/watch?v=cjboaM6-PQs