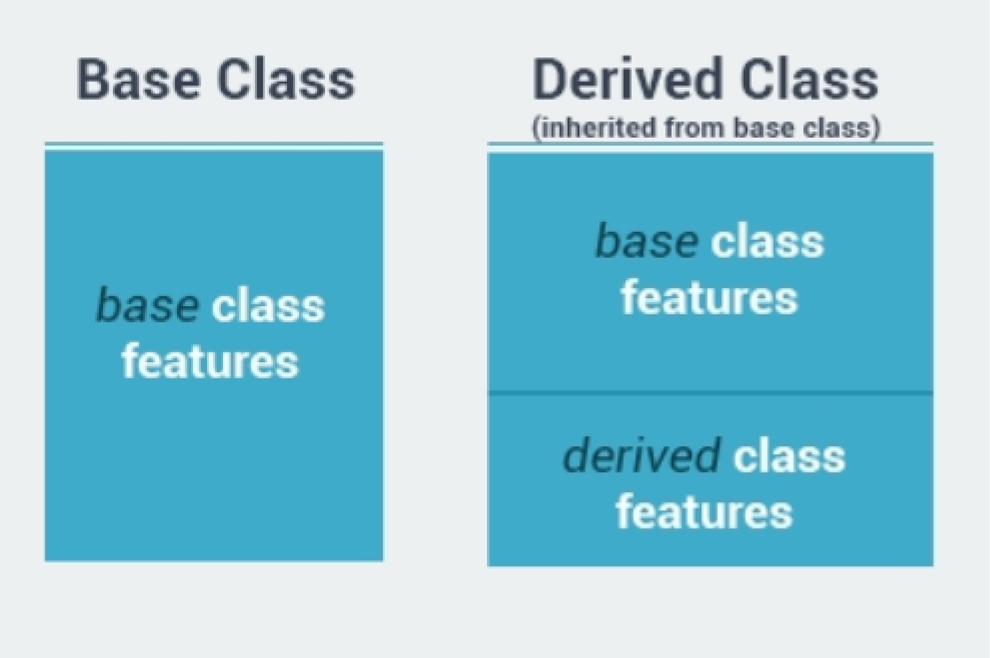
Inheritance & Polymorphism

Inheritance

Inheritance allows us to define a class based on another class. This makes creating and maintaining an application easy.
The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class.
For example, base class **Animal** can be used to derive **Cat** and **Dog** classes.
The derived class inherits all the features from the base class, and can have its own additional features.



> Inheritance allows us to define a class based on another class.

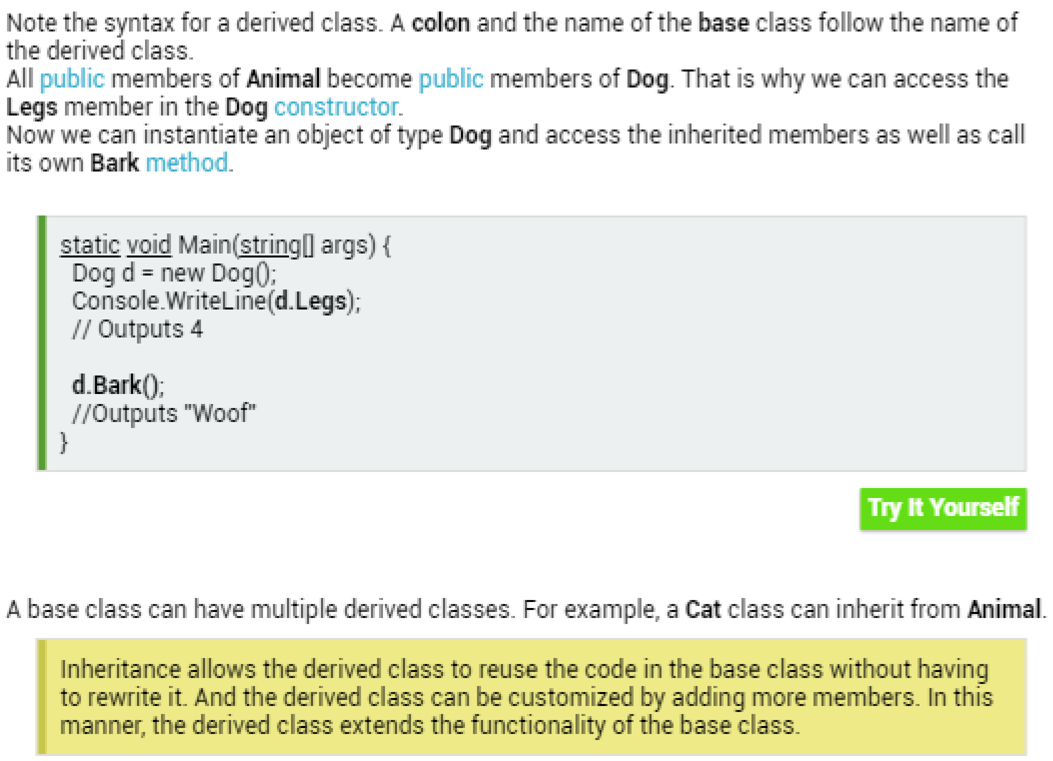
Inheritance

Let's define our base class **Animal**:

```
class Animal {
  public int Legs {get; set;}
  public int Age {get; set;}
}
```

Now we can derive class **Dog** from it:

```
class Dog : Animal {
  public Dog() {
    Legs = 4;
  }
  public void Bark() {
    Console.Write("Woof");
  }
}
```



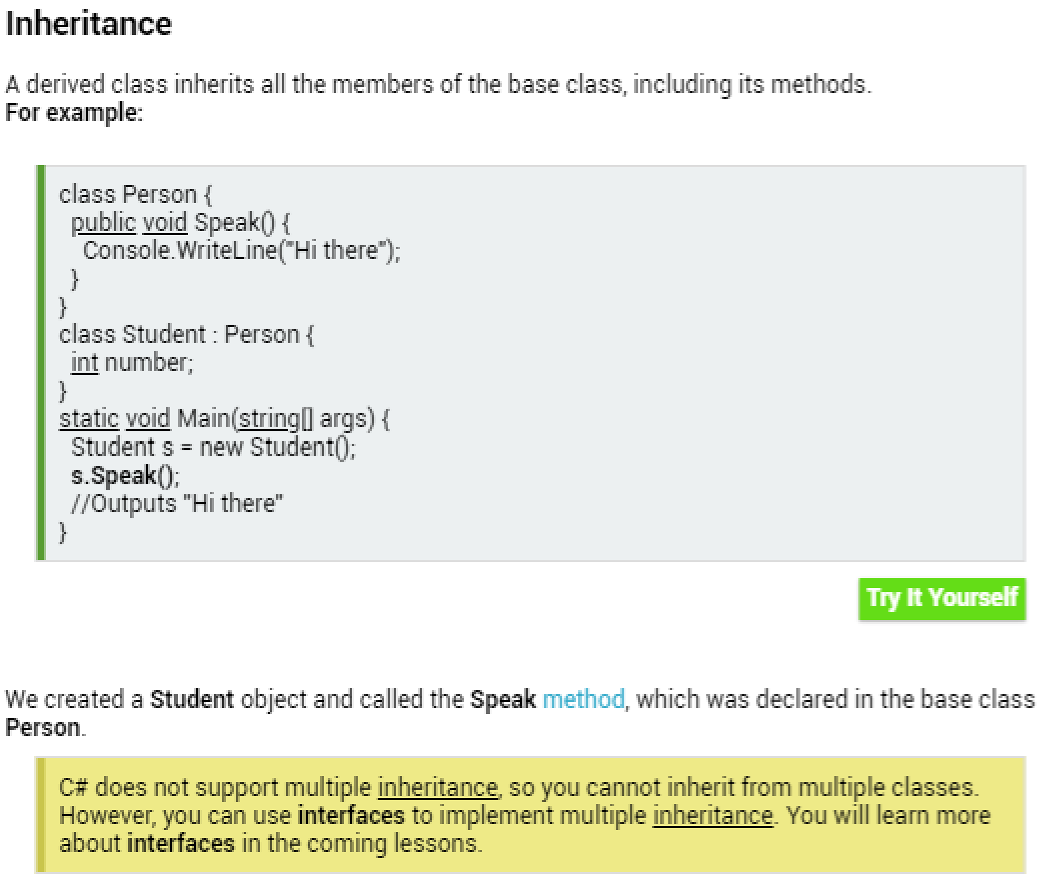
Note the syntax for a derived class. A **colon** and the name of the **base** class follow the name of the derived class.
All public members of **Animal** become public members of **Dog**. That is why we can access the **Legs** member in the **Dog** constructor.
Now we can instantiate an object of type **Dog** and access the inherited members as well as call its own **Bark** method.

```
static void Main(string[] args) {
  Dog d = new Dog();
  Console.WriteLine(d.Legs);
  // Outputs 4

  d.Bark();
  //Outputs "Woof"
}
```

A base class can have multiple derived classes. For example, a **Cat** class can inherit from **Animal**.

> Inheritance allows the derived class to reuse the code in the base class without having to rewrite it. And the derived class can be customized by adding more members. In this manner, the derived class extends the functionality of the base class.



Inheritance

A derived class inherits all the members of the base class, including its methods.
For example:

```
class Person {
  public void Speak() {
    Console.WriteLine("Hi there");
  }
}
class Student : Person {
  int number;
}
static void Main(string[] args) {
  Student s = new Student();
  s.Speak();
  //Outputs "Hi there"
}
```

We created a **Student** object and called the **Speak** method, which was declared in the base class **Person**.

> C# does not support multiple inheritance, so you cannot inherit from multiple classes. However, you can use **interfaces** to implement multiple inheritance. You will learn more about **interfaces** in the coming lessons.

protected

Up to this point, we have worked exclusively with **public** and **private** access modifiers.
Public members may be accessed from anywhere outside of the class, while access to private members is limited to their class.
The **protected** access modifier is very similar to **private** with one difference; it can be accessed in the derived classes. So, a **protected** member is accessible only from derived classes.
For example:

```
class Person {
  protected int Age {get; set;}
  protected string Name {get; set;}
}
class Student : Person {
  public Student(string nm) {
    Name = nm;
  }
  public void Speak() {
    Console.Write("Name: "+Name);
  }
}
static void Main(string[] args) {
  Student s = new Student("David");
  s.Speak();
  //Outputs "Name: David"
}
```

Try It Yourself

As you can see, we can access and modify the **Name** property of the base class from the derived class.
But, if we try to access it from outside code, we will get an error:

```
static void Main(string[] args) {
  Student s = new Student("David");
  s.Name = "Bob"; //Error
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

sealed

A class can prevent other classes from inheriting it, or any of its members, by using the **sealed** modifier.
For example:

```
sealed class Animal {
  //some code
}
class Dog : Animal { } //Error
```

Try It Yourself

In this case, we cannot derive the Dog class from the Animal class because Animal is **sealed**.

> The **sealed** keyword provides a level of protection to your class so that other classes cannot inherit from it.

Inheritance

Constructors are called when objects of a class are created. With inheritance, the base class constructor and destructor are not inherited, so you should define constructors for the derived classes.
However, the base class constructor and destructor are being invoked automatically when an object of the derived class is created or deleted.
Consider the following example:

```
class Animal {
 public Animal() {
  Console.WriteLine("Animal created");
 }
 ~Animal() {
  Console.WriteLine("Animal deleted");
 }
}
class Dog: Animal {
 public Dog() {
  Console.WriteLine("Dog created");
 }
 ~Dog() {
  Console.WriteLine("Dog deleted");
 }
}
```

We have defined the Animal class with a constructor and destructor and a derived Dog class with its own constructor and destructor.

> So what will happen when we create an object of the derived class? Tap next to find out!

Inheritance

Let's create a Dog object:

```
static void Main(string[] args) {
 Dog d = new Dog();
}
/*Outputs
Animal created
Dog created
Dog deleted
Animal deleted
*/
```

Try It Yourself

Note that the base class constructor is called first and the derived class constructor is called next.
When the object is destroyed, the derived class destructor is invoked and then the base class destructor is invoked.

> You can think of it as the following: The derived class needs its base class in order to work, which is why the base class constructor is called first.

Polymorphism

The word **polymorphism** means "having many forms".
Typically, polymorphism occurs when there is a hierarchy of classes and they are related through inheritance from a common base class.
Polymorphism means that a call to a member method will cause a different implementation to be executed depending on the **type** of object that invokes the method.

> Simply, polymorphism means that a single method can have a number of different implementations.

Polymorphism

Consider having a program that allows users to draw different shapes. Each shape is drawn differently, and you do not know which shape the user will choose.
Here, polymorphism can be leveraged to invoke the appropriate **Draw** method of any derived class by overriding the same method in the base class. Such methods must be declared using the **virtual** keyword in the base class.
For example:

```
class Shape {
  public virtual void Draw() {
    Console.Write("Base Draw");
  }
}
```

The **virtual** keyword allows methods to be overridden in derived classes.

> Virtual methods enable you to work with groups of related objects in a uniform way.

Polymorphism

Now, we can derive different shape classes that define their own **Draw** methods using the **override** keyword:

```
class Circle : Shape {
  public override void Draw() {
    // draw a circle...
    Console.WriteLine("Circle Draw");
  }
}
class Rectangle : Shape {
  public override void Draw() {
    // draw a rectangle...
    Console.WriteLine("Rect Draw");
  }
}
```

The virtual Draw method in the Shape base class can be overridden in the derived classes. In this case, Circle and Rectangle have their own Draw methods.
Now, we can create separate Shape objects for each derived type and then call their Draw methods:

```
static void Main(string[] args) {
  Shape c = new Circle();
  c.Draw();
  //Outputs "Circle Draw"

  Shape r = new Rectangle();
  r.Draw();
  //Outputs "Rect Draw"
}
```

Try It Yourself

> As you can see, each object invoked its own **Draw** method, thanks to polymorphism.

Polymorphism

To summarize, **polymorphism** is a way to call the same method for different objects and generate different results based on the object type. This behavior is achieved through virtual methods in the base class.
To implement this, we create objects of the base type, but instantiate them as the derived type:

```
Shape c = new Circle();
```

Shape is the base class. **Circle** is the derived class.
So why use polymorphism? We could just instantiate each object of its type and call its method, as in:

```
Circle c = new Circle();
c.Draw();
```

The polymorphic approach allows us to treat each object the same way. As all objects are of type Shape, it is easier to maintain and work with them. You could, for example, have a list (or array) of objects of that type and work with them dynamically, without knowing the actual derived type of each object.

> Polymorphism can be useful in many cases. For example, we could create a game where we would have different Player types with each Player having a separate behavior for the Attack method.
> In this case, Attack would be a virtual method of the base class Player and each derived class would override it.

Abstract Classes

As described in the previous example, polymorphism is used when you have different derived classes with the same method, which has different implementations in each class. This behavior is achieved through **virtual** methods that are **overridden** in the derived classes.
In some situations there is no meaningful need for the virtual method to have a separate definition in the base class.

These methods are defined using the **abstract** keyword and specify that the derived classes must define that method on their own.
You cannot create objects of a class containing an abstract method, which is why the class itself should be abstract.
We could use an abstract method in the Shape class:

```
abstract class Shape {
  public abstract void Draw();
}
```

As you can see, the **Draw** method is **abstract** and thus has no body. You do not even need the curly brackets; just end the statement with a semicolon.
The Shape class itself must be declared **abstract** because it contains an **abstract** method.
Abstract method declarations are only permitted in abstract classes.

> Remember, **abstract** method declarations are only permitted in **abstract** classes.
> Members marked as **abstract**, or included in an abstract class, must be implemented by classes that derive from the abstract class. An abstract class can have multiple abstract members.

Abstract Classes

An abstract class is intended to be a base class of other classes. It acts like a template for its derived classes.
Now, having the abstract class, we can derive the other classes and define their own **Draw**() methods:

```
abstract class Shape {
 public abstract void Draw();
}
class Circle : Shape {
 public override void Draw() {
  Console.WriteLine("Circle Draw");
 }
}
class Rectangle : Shape {
 public override void Draw() {
  Console.WriteLine("Rect Draw");
 }
}
static void Main(string[] args) {
 Shape c = new Circle();
 c.Draw();
 //Outputs "Circle Draw"
}
```

Try It Yourself

Abstract classes have the following features:
- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

> It is not possible to modify an **abstract** class with the **sealed** modifier because the two modifiers have opposite meanings. The **sealed** modifier prevents a class from being inherited and the **abstract** modifier requires a class to be inherited.

Interfaces

An **interface** is a completely abstract class, which contains **only** abstract members.
It is declared using the **interface** keyword:

```
public interface IShape
{
  void Draw();
}
```

All members of the interface are by default abstract, so no need to use the abstract keyword.
Also, all members of an interface are always public, and no access modifiers can be applied to them.

> It is common to use the capital letter **I** as the starting letter for an interface name.
> Interfaces can contain properties, methods, etc. but **cannot** contain fields (variables).

Interfaces

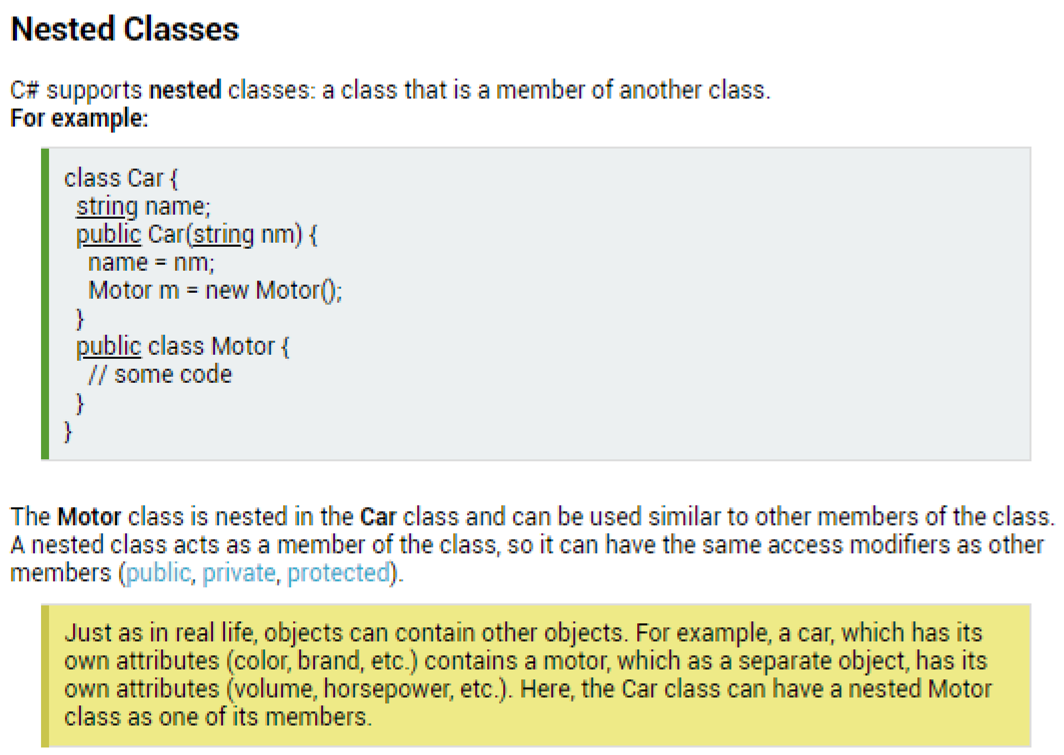
When a class **implements** an interface, it must also implement, or define, all of its methods.
The term **implementing an interface** is used (opposed to the term "inheriting from") to describe the process of creating a class based on an interface. The interface simply describes what a class should do. The class implementing the interface must define how to accomplish the behaviors.
The syntax to implement an interface is the same as that to derive a class:

```
public interface IShape {
  void Draw();
}
class Circle : IShape {
  public void Draw() {
    Console.WriteLine("Circle Draw");
  }
}
static void Main(string[] args) {
  IShape c = new Circle();
  c.Draw();
  //Outputs "Circle Draw"
}
```

Try It Yourself

Note, that the **override** keyword is not needed when you implement an interface.

> But why use interfaces rather than abstract classes?
> A class can inherit from just one base class, but it can implement **multiple** interfaces!
> Therefore, by using interfaces you can include behavior from multiple sources in a class.
> To implement multiple interfaces, use a comma separated list of interfaces when creating the class: **class A: IShape, IAnimal, etc.**



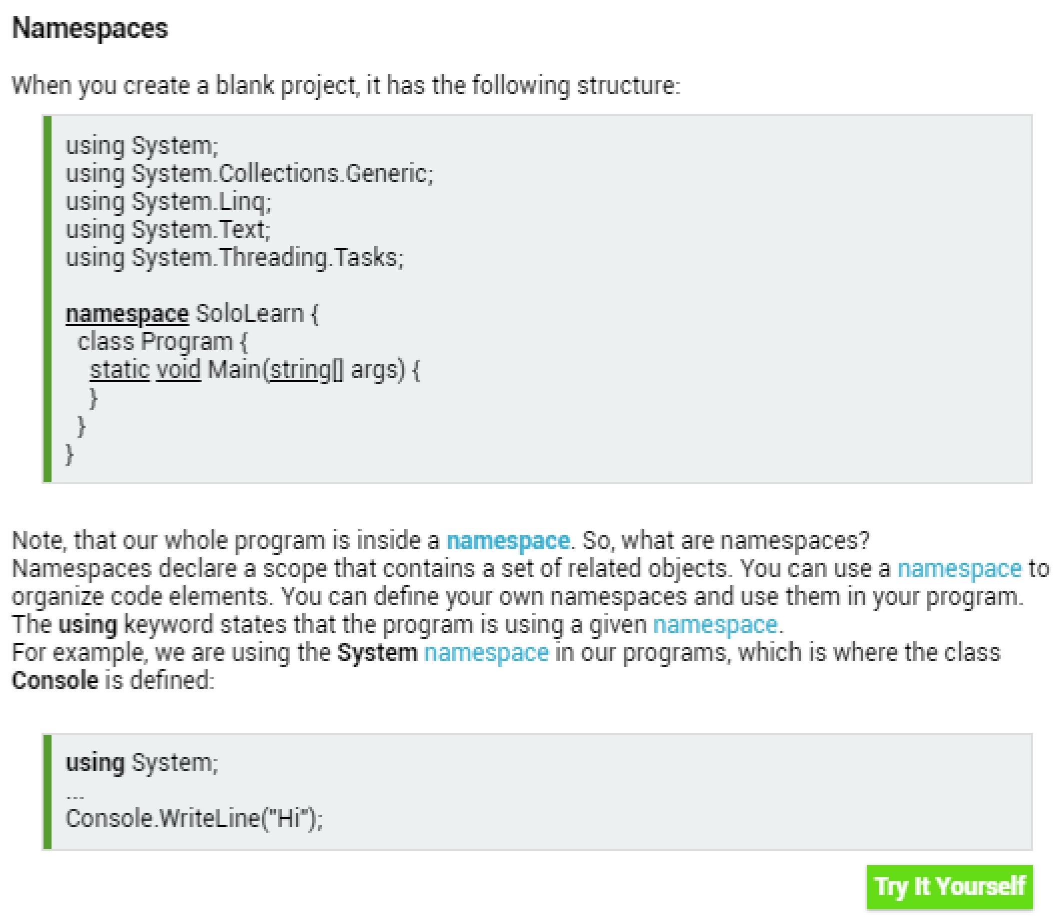
Nested Classes

C# supports **nested** classes: a class that is a member of another class.
For example:

```
class Car {
 string name;
 public Car(string nm) {
  name = nm;
  Motor m = new Motor();
 }
 public class Motor {
  // some code
 }
}
```

The **Motor** class is nested in the **Car** class and can be used similar to other members of the class.
A nested class acts as a member of the class, so it can have the same access modifiers as other members (public, private, protected).

> Just as in real life, objects can contain other objects. For example, a car, which has its own attributes (color, brand, etc.) contains a motor, which as a separate object, has its own attributes (volume, horsepower, etc.). Here, the Car class can have a nested Motor class as one of its members.



Namespaces

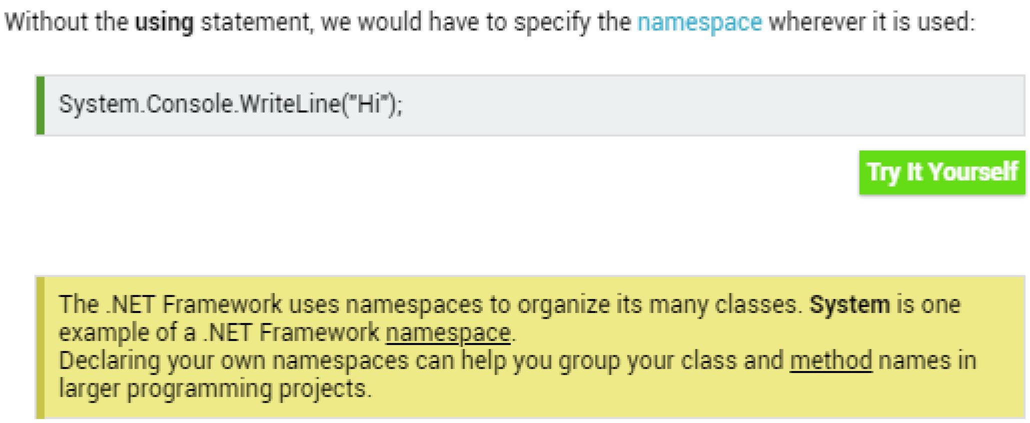
When you create a blank project, it has the following structure:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SoloLearn {
 class Program {
  static void Main(string[] args) {
  }
 }
}
```

Note, that our whole program is inside a **namespace**. So, what are namespaces?
Namespaces declare a scope that contains a set of related objects. You can use a namespace to organize code elements. You can define your own namespaces and use them in your program.
The **using** keyword states that the program is using a given namespace.
For example, we are using the **System** namespace in our programs, which is where the class **Console** is defined:

```
using System;
...
Console.WriteLine("Hi");
```



Without the **using** statement, we would have to specify the namespace wherever it is used:

```
System.Console.WriteLine("Hi");
```

The .NET Framework uses namespaces to organize its many classes. **System** is one example of a .NET Framework namespace.
Declaring your own namespaces can help you group your class and method names in larger programming projects.

End.