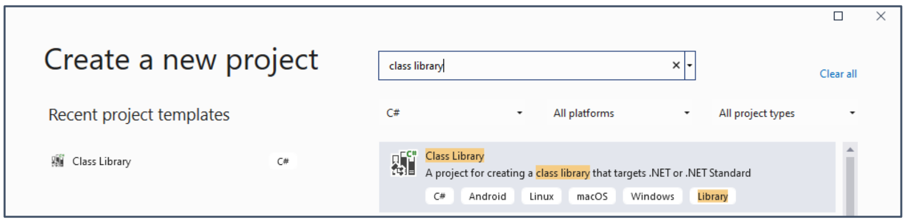
Interfaces

The objective of this exercise is to consolidate your understanding of interfaces.

1 Create a new Visual Studio project of type Class Library called **InterfaceProject**.



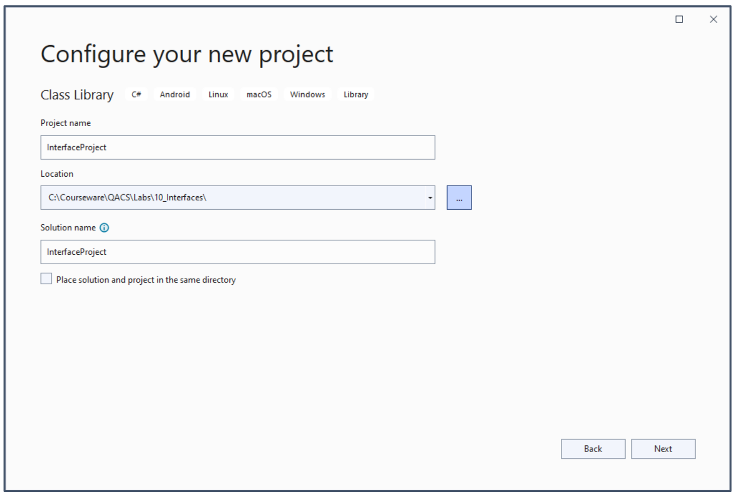
Create a new project

Search: class library

Recent project templates: Class Library C#

Filters: C# · All platforms · All project types · Clear all

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library



Configure your new project

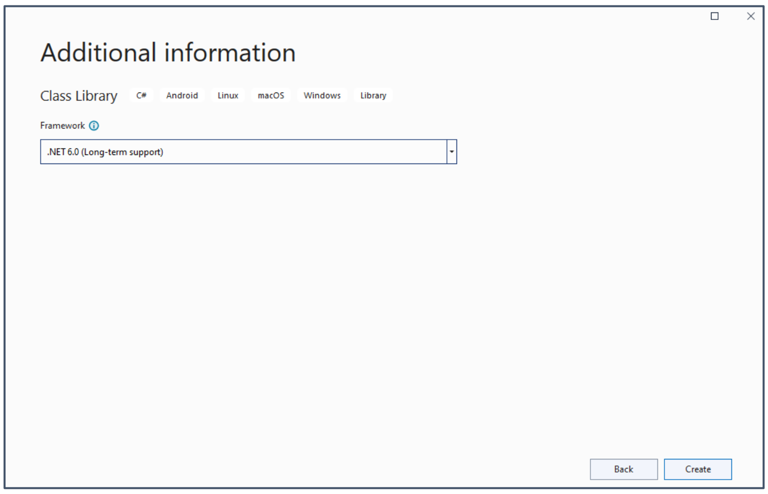
Class Library C# Android Linux macOS Windows Library

Project name: InterfaceProject

Location: C:\Courseware\QACS\Labs\10_Interfaces\

Solution name: InterfaceProject

☐ Place solution and project in the same directory



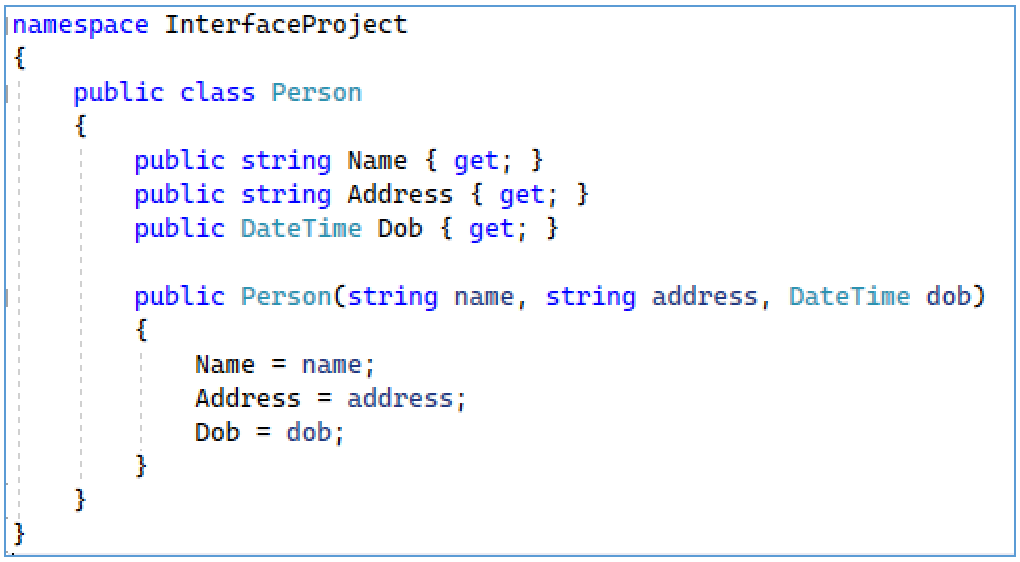
Additional information

Class Library C# Android Linux macOS Windows Library

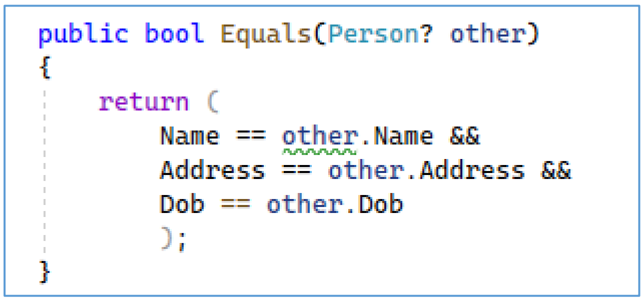
Framework: .NET 6.0 (Long-term support)

Delete **Class1**.

Add to this solution an XUnit Test project called **TestProject**.

2	We have provided a **Person** class in the Assets folder. Drag this file onto your **InterfaceProject**.
	
	
	`namespace InterfaceProject`
	`{`
	`    public class Person`
	`    {`
	`        public string Name { get; }`
	`        public string Address { get; }`
	`        public DateTime Dob { get; }`
	
	`        public Person(string name, string address, DateTime dob)`
	`        {`
	`            Name = name;`
	`            Address = address;`
	`            Dob = dob;`
	`        }`
	`    }`
	`}`

3	Replace the provided Test1() with the two tests in **IEquatable.txt** from the **Assets** folder and resolve the red squiggles.
4	Run the tests. The first test fails, the second test passes by accident. The problem is that the compiler doesn't know how to figure out if two Persons are the same person. For example, if your bank account has a balance of £100 and so does mine – does that mean they are the same account?

5	Make Person implement the interface **IEquatable<Person>**
	
	This will make you implement the **Equals** method.
	
	As far as we are concerned, if the Name, Address, and DateOfBirth are the same, it is the same person. Put in this code and ensure both tests now pass.
	
	
	`public bool Equals(Person? other)`
	`{`
	`    return (`
	`        Name == other.Name &&`
	`        Address == other.Address &&`
	`        Dob == other.Dob`
	`        );`
	`}`

6 You will now compare some people, the Spice Girls, by putting them in order based on their DateOfBirth.

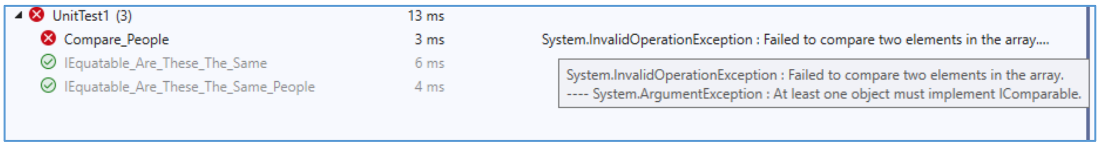
We'll put these in DateOfBirth order.

Copy in the test in the Assets folder called **Compare_People** and resolve the red squiggles.

```
[Fact]
public void Compare_People()
{
    List<Person> spiceGirls = new List<Person>() {
        new Person("Baby", "Finchley", dob:new DateTime(1976, 1, 21) ),
        new Person("Posh", "Harlow", dob:new DateTime(1974, 4, 17) ),
        new Person("Ginger", "Watford", dob:new DateTime(1972, 8, 6) ),
        };

    spiceGirls.Sort();
    Assert.Equal("Ginger", spiceGirls[0].Name);
    Assert.Equal("Posh", spiceGirls[1].Name);
    Assert.Equal("Baby", spiceGirls[2].Name);
}
```

Run the test. It will fail. Look at the error message:



```
UnitTest1 (3)                              13 ms
  Compare_People                            3 ms    System.InvalidOperationException : Failed to compare two elements in the array....
  IEquatable_Are_These_The_Same             6 ms
  IEquatable_Are_These_The_Same_People      4 ms    System.InvalidOperationException : Failed to compare two elements in the array.
                                                    ---- System.ArgumentException : At least one object must implement IComparable.
```

It fails because it does not know how to sort Persons into order because they are not comparable.

7 Implement the interface **IComparable<Person>**

Notice the **CompareTo** method returns an int.

We want to rank Person by **DateOfBirth**, so:

CompareTo should return **+1** if DoB is greater than the compared to object's' DoB.

-1 if smaller

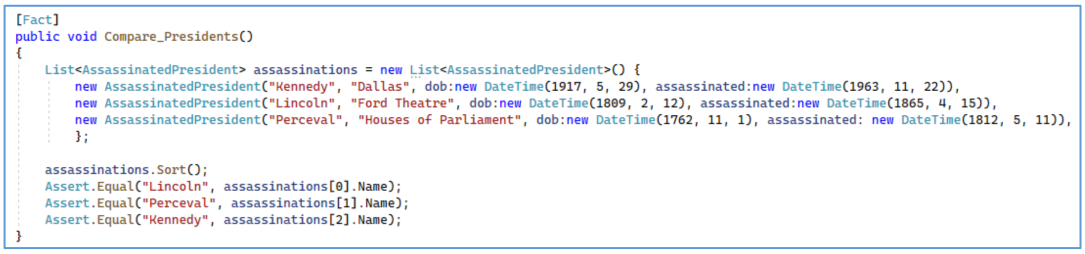
Otherwise return **0**.

Enter the code and run the test to ensure it now passes.

8

In the Assets folder is a class **AssassinatedPresident**. Drag this onto your **InterfaceProject**.

Add the test **Compare_Presidents**.



```
[Fact]
public void Compare_Presidents()
{
    List<AssassinatedPresident> assassinations = new List<AssassinatedPresident>() {
        new AssassinatedPresident("Kennedy", "Dallas", dob:new DateTime(1917, 5, 29), assassinated:new DateTime(1963, 11, 22)),
        new AssassinatedPresident("Lincoln", "Ford Theatre", dob:new DateTime(1809, 2, 12), assassinated:new DateTime(1865, 4, 15)),
        new AssassinatedPresident("Perceval", "Houses of Parliament", dob:new DateTime(1762, 11, 1), assassinated: new DateTime(1812, 5, 11)),
        };

    assassinations.Sort();
    Assert.Equal("Lincoln", assassinations[0].Name);
    Assert.Equal("Perceval", assassinations[1].Name);
    Assert.Equal("Kennedy", assassinations[2].Name);
}
```

Now implement **IComparable** for **AssassinatedPresident** where we want the sort order to be by the *month* in which they were *assassinated*.

Run the Test and confirm it passes.

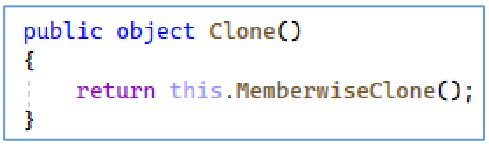
9

You will now experiment with cloning objects.

Copy in the **Clone_A_Spice_Girl** test.

The interface you need is **ICloneable**. Make Person implement the interface **ICloneable**.

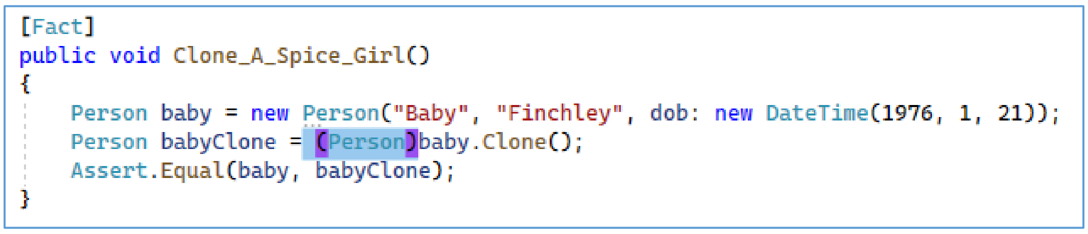
Conveniently, there is a protected method on the Object class which does this for us:



```
public object Clone()
{
    return this.MemberwiseClone();
}
```

But you'll get a red squiggly in the test.

This is because **Clone** method returns an object, so we have to cast the result to **Person** in the test:



```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = (Person)baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

Re-run the test. It should now pass.

10

This works, but it's not ideal.

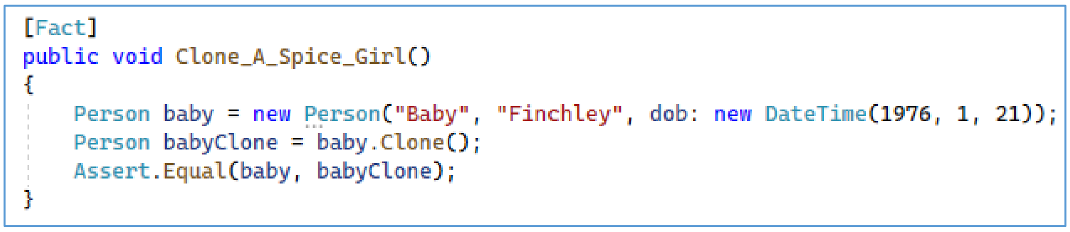
It’s a pity that the framework does not offer a generic version of **ICloneable.**

Have a go at creating your own generic **ICloneable<T>** interface.

Hint: Do an F12 on the generic interfaces **IEquatable** and **IComparable** to see how they are created.

Implement your generic version of **ICloneable<T>** in Person instead of the non-generic **ICloneable**. You can perform the explicit cast in the implemented **Clone** method.

Your test should no longer require the explicit cast and should be as follows:

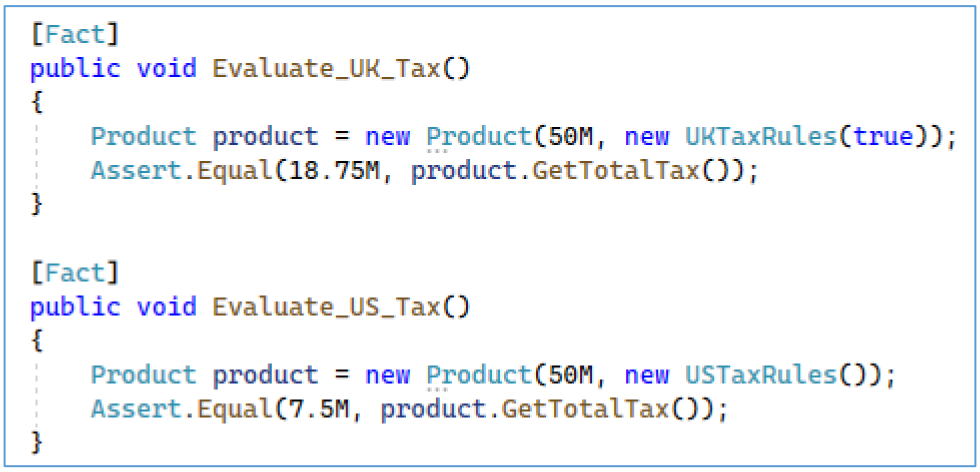


```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

Run the Tests. All tests should now pass.

11 You will now explore another interface **ITaxRules** and see how it is implemented in different classes.

Into your **TestProject**, copy the **Tax Tests** from the **Assets** folder:



```
[Fact]
public void Evaluate_UK_Tax()
{
    Product product = new Product(50M, new UKTaxRules(true));
    Assert.Equal(18.75M, product.GetTotalTax());
}

[Fact]
public void Evaluate_US_Tax()
{
    Product product = new Product(50M, new USTaxRules());
    Assert.Equal(7.5M, product.GetTotalTax());
}
```

We have provided all the code for you.

Copy the **Product** folder from the **Assets** folder and drop it onto your **InterfaceProject**.

	Have a look at the **Product** class. See that it is passed in an **ITaxRules** in its constructor but it doesn’t know or care what sort of tax rules they, are as long as they implement **ITaxRules.** Review the code and confirm all tests pass.

Explicit interfaces

Because one class can implement many interfaces, it's possible that there could be a clash of method name. In the example coming up, we create an **AmphibiousVehicle** (this is the DUKW – pronounced Duck), and it was actually the US Army coding for such a vehicle.

D=1942

U=Utility

K=all-wheel drive

W=2 powered rear axles

This remarkable vehicle is truly a water vehicle and truly a land vehicle. In other technologies we might describe this by using multiple inheritance. .NET outlaws multiple inheritance, so we would achieve this by implementing multiple interfaces.

1	Drag the folder **DUKW** onto your **InterfaceProject** and have a look at the three files. In particular, notice that both interfaces have a **Brake** method. A land vehicle brakes by squeezing the disc pads. A water vehicle brakes by putting the propeller into reverse. Therefore, we need two methods.
2	Open the **AmphibiousVehicle** class and implement the interface **ILandVehicle**. Build the project to confirm there are no errors. You will see it gives you one method and both interfaces are satisfied. However, it gives you no chance to have two methods. Delete the **Brake** method.
3	Now implement the **IWaterVehicle** interface *explicitly* by placing the cursor on **IWaterVehicle** in **AmphibiousVehicle** and using **Ctrl+dot** -> **Implement all members explicitly**.

Delete the **throw NotImplementedException**.

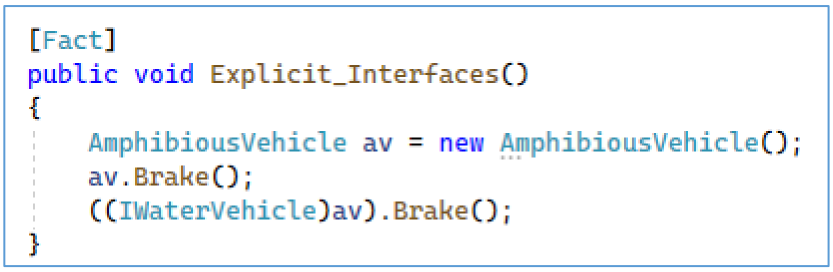
Now place the cursor on **ILandVehicle** and use **Ctrl+dot** -> **Implement interface**.

You get the Brake method.

Delete the **throw NotImplementedException**.

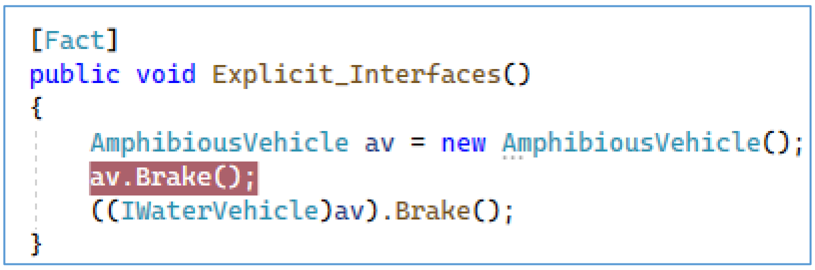
4 You will now use a test to call the implicitly implemented brake method and the explicitly implemented break method.

Add this test to **TestProject**:



```
[Fact]
public void Explicit_Interfaces()
{
    AmphibiousVehicle av = new AmphibiousVehicle();
    av.Brake();
    ((IWaterVehicle)av).Brake();
}
```

Add a breakpoint:



```
[Fact]
public void Explicit_Interfaces()
{
    AmphibiousVehicle av = new AmphibiousVehicle();
    av.Brake();
    ((IWaterVehicle)av).Brake();
}
```

Right-click in the test and choose **Debug Tests.** Step into the code using and **Step Into (F11)** to confirm that both brake methods are called.