11. What are default implementations in interfaces?

Brief summary: Starting with C# 8, we can provide methods implementations in interfaces. This feature was mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code.

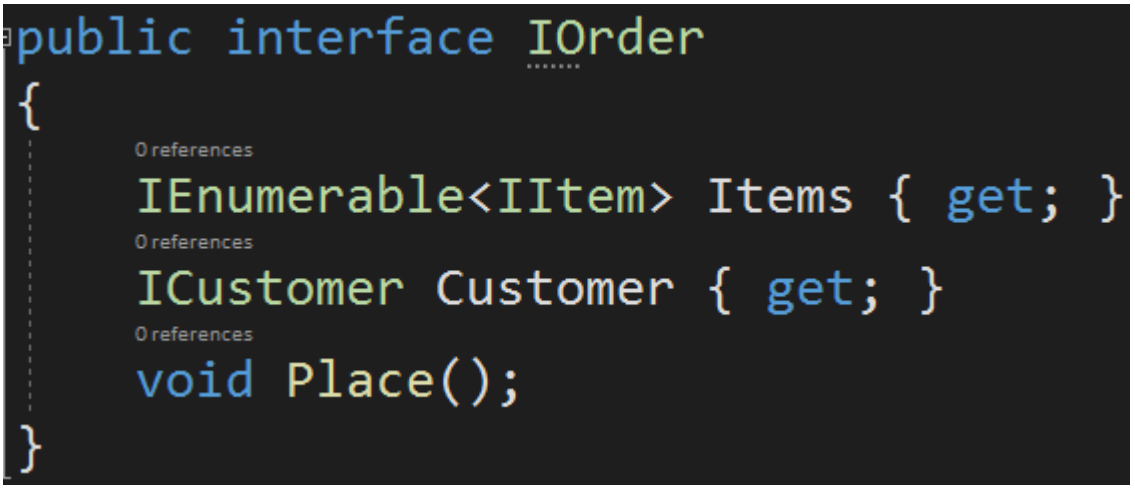
If I asked you "What are the characteristics of interfaces in C#?" you would probably be able to list them pretty easily:

- they can only contain methods, properties, indexers, and events declarations. They can't have fields.
- the methods declared in the interface can't have implementations. In other words, they are methods with no bodies.
- the methods can't be declared abstract or virtual (because they are implicitly virtual).
- all its members are by default public and using any other access modifier leads to a compilation error.
- they can't have static methods.

This is all perfectly correct… or rather was, until C# 8 was introduced in 2019. With this version of C# new feature was introduced: **default implementations in interfaces**. In short, it means that interfaces can now contain methods with bodies. Because of that a couple of other changes must have been introduced too - for example, methods in the interface can be private now, it can contain fields, etc.

If you (like me) are used to the "old" interfaces, you are probably quite surprised now. This change goes against everything we knew about them - that they are an abstraction over behavior, or that they only define a contract a class must fulfill. When I learned about this change my first thought was "so what will be the difference between an interface and abstract class now?". It seemed to me (correctly) that they will be very similar concepts from now on, so I was asking myself why did Microsoft decide to introduce this change.

So, before we dive into details, let's understand **why**. I will explain it by showing you an example. Let's say we develop some library that other people or companies will use. We will publish it as a NuGet package. Let's say this is one of the interfaces we define in our library, and we expect our customers to provide their own implementations:

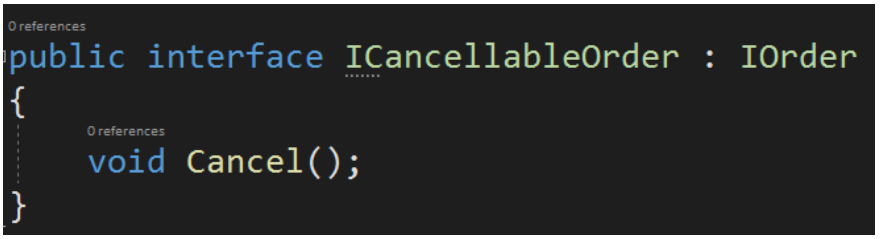


Our library is meant for e-commerce, and this interface defines what functionality should the classes representing orders contain.

We finish our work and we release our library. The release is a roaring success, and more and more people download the package with NuGet. It is widely used and highly rated.

One year later we are almost ready to release version 2.0. There is one problem, though. We would like to add “void Cancel();” method to the IOrder interface. But if we do so, and our customers upgrade the version of the library, they all will suddenly see compilation errors all-around their codebases. The classes they defined to implement the IOrder interface do not provide the implementation of the Cancel method. We will force our customers to adjust to this breaking change, and this may not be easy. Some of them may be stuck with development for days, and their business may be impacted by it.

One solution could be to extend the existing interface like this:



We will create a new interface extending the old one. Our customers can gradually start using it in their codebase, and everyone is happy.

But there is a problem with this solution. As our library evolves, we may want to add more and more methods to the IOrder interface. This will lead to creating new interfaces, and soon it will become hard to maintain. No one will wrap their heads

around what is what in the application, as everywhere we will see things like IOrder, ICancellableOrder, ICancellableOrderWithDeliveryDelay, ICancellableOrderWithDeliveryDelayAndDiscout, etc.

And this is where default implementations in interfaces can help. We can add new methods to an existing interface, and provide default implementations, so we won't break our customers' code. If they want to, they can provide their own implementation, but until then the default implementation will be used. Let's see this in code:

```
public interface IOrder
{
    1 reference
    IEnumerable<IItem> Items { get; }
    1 reference
    ICustomer Customer { get; }
    1 reference
    void Place();

    1 reference
    public void DelayDeliveryByDays(int days)
    {
        Console.WriteLine("DelayDeliveryByDays from interface");
    }
}
```

As you can see the DelayDeliveryByDays method has a body, which was impossible before C# 8.

Now, let's define a class implementing this interface:

```
class CustomOrder : IOrder
{
    1 reference
    public IEnumerable<IItem> Items { get; } = new List<IItem>();

    1 reference
    public ICustomer Customer {get;}

    1 reference
    public void Place()
    {
        Console.WriteLine("Placing an order");
    }
}
```

And let's see if we can call the DelayDeliveryByDays method on an object of this class:

```
CustomOrder order = new CustomOrder();
order.DelayDeliveryByDays(1);
```

Well, that doesn't compile. We can't use the default interface implementations on variables of a concrete type. We must use it via the interface:

```
IOrder orderByInterface = new CustomOrder();
orderByInterface.DelayDeliveryByDays(1);
```

Of course, if we provide the implementation of this method in the concrete class, it will be used instead of the implementation from the interface. Let's add another class implementing the IOrder interface:

```
class CustomOrderWithDelay : IOrder
{
    1 reference
    public IEnumerable<IItem> Items { get; } = new List<IItem>();

    1 reference
    public ICustomer Customer { get; }

    1 reference
    public void Place()
    {
        Console.WriteLine("Placing an order");
    }

    2 references
    public void DelayDeliveryByDays(int days)
    {
        Console.WriteLine("DelayDeliveryByDays from CustomOrderWithDelay");
    }
}
```

And now, let's see what this code will print:

```
IOrder order = new CustomOrder();
order.DelayDeliveryByDays(1);

IOrder orderWithDelay = new CustomOrderWithDelay();
orderWithDelay.DelayDeliveryByDays(1);
```

And the result is:

```
DelayDeliveryByDays from interface
DelayDeliveryByDays from CustomOrderWithDelay
```

As you can see, if the non-default implementation is provided, it will be used.

All right. Since we now can define methods in interfaces, we may need some other things that typically are used in methods, like:

- other, private methods that can enclose some piece of logic.
- static methods to do the same, if they don't use any non-static members of the interface.
- private fields.

Also, if an interface is derived from another interface, we may want to use the members of the parent. This means we can declare interface methods or fields as protected. The protected methods or fields are only available in derived interfaces, not classes implementing the interface.

We can also have virtual methods in the interface, but they can only be overridden by derived interfaces, not the classes implementing the interface. Also, if we declare a virtual method in the interface, it must contain a body. So for example, this method is defined in the base interface:

```
public interface IOrder
{
    2 references
    public virtual void VirtualMethodFromInterface()
    {
        Console.WriteLine("IOrder interface");
    }
```

And we can override it in the derived interface like this:

```
public interface ICancellableOrder : IOrder
{
    2 references
    void IOrder.VirtualMethodFromInterface()
    {
        Console.WriteLine("ICancellableOrder interface");
    }
```

All right. Let's summarize the topic of default implementations in interfaces.

This feature was added in C# 8. It's mostly designed to make it easier to add new methods to interfaces without breaking the existing code. Also, they make it possible for C# to work with APIs targeting Android (written in Java) and iOS (written in Swift) as those languages support similar features. They also enable using something called Traits, which is beyond this course's level; if you are curious, you can read about it here:
https://en.wikipedia.org/wiki/Trait_(computer_programming).
https://dlang.org/spec/traits.html
https://stackoverflow.com/questions/59547812/c-sharp-interface-with-default-method-vs-traits

As you can see this was a huge change and it completely changed what was true about interfaces in C#. This feature received a lot of criticism, and to be honest I can see why. The line between interfaces and abstract classes is very blurry now. In practice, it's hard to provide a default implementation that brings any value. I recommend this extensive article pointing out some problems with the default implementations in interfaces:
https://jeremybytes.blogspot.com/2019/09/interfaces-in-c-8-are-bit-of-mess.html

My recommendation is as follows: be aware that something like the default implementation in interfaces exists. Still, I think it's best if you use interfaces as they were meant to be used before C# 8 unless you are 100% sure you know what you are doing and equally sure that this will bring value to your application.

Bonus questions:

- "**What can be the reason for using default implementations in interfaces?**"
  Default implementations in interfaces are mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code. Without it, if we add a method to an interface we release it as a public library, we will force everyone who updates this library to provide the implementation immediately - otherwise, their code will not build.