C# Intermediate: Classes, Interfaces and OOP

INTERFACES

Introduction

- An interface is a language construct that is similar to a class (in terms of syntax) but is fundamentally different.
- An interface is simply a declaration of the capabilities (or services) that a class should provide.

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

- This interface states a class that wants to play the role of a tax calculator, should provide a method called Calculate() that takes no parameters and returns an int. The implementation of this class might look like this:

```
public class TaxCalculator : ITaxCalculator
{
       public void Calculate() { … }
}
```

- So an interface is purely a declaration. Members of an interface do not have implementation.
- An interface can only declare methods and properties, but not fields (because fields are about implementation detail).
- Members of an interface do not have access modifiers.

- Interfaces help building loosely coupled applications. We reduce the coupling between two classes by putting an interface between them. This way, if one of these classes changes, it will have no impact on the class that is dependent on that (as long as the interface is kept the same).

Interfaces and Testability

- Unit testing is part of the automated practice which helps improve the quality of our code. With automated testing, we write code to test our own code. This helps catching bugs early on as we change the code.
- In order to unit test a class, we need to isolate it. This means: we need to assume that every other class in our application is working properly and see if the class under test is working as expected.
- A class that has tight dependencies to other classes cannot be isolated.
- To solve this problem, we use an interface. Here is an example:

```
public class OrderProcessor
{
	private IShippingCalculator _calculator;

	public Customer(IShippingCalculator calculator)
	{
		_calculator = calculator;
	}

	…
}
```

- So here, OrderProcessor is not dependent on the ShippingCalculator class. It's only dependent on an interface (IShippingCalculator). If we change the code inside the ShippingCalculator (eg add a new method or change the method implementations) it will have no impact on OrderProcessor (as long as the interface is kept the same).

Interfaces and Extensibility

- We can use interfaces to change our application's behaviour by "extending" its code (rather than changing the existing code).
- If a class is dependent on an interface, we can supply a different implementation of that interface at runtime. This way, the behaviour of the application changes without any impact on that class.
- For example, let's assume our **DbMigrator** class is dependent on an **ILogger** interface. At runtime, we can supply a **ConsoleLogger** to log the messages on the console. Later, we may decide to log the messages in a file (or a database). We can simply create a new class that implements the **ILogger** interface and inject it into **DbMigrator**.

Interfaces and Inheritance

- One of the common misconceptions about interfaces is that they are used to implement multiple inheritance in C#. This is fundamentally wrong, yet many books and videos make such a false claim.
- With inheritance, we write code once and re-use it without the need to type all that code again.
- With interfaces, we simply declare the members the implementing class should contain. Then we need to type all that declaration along with the actual implementation in that class. So, code is not inherited, even the declaration of the members!