Lesson 27 - Inheritance and Polymorphism III

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:

```
public 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.

> 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.

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:

```
public class Circle : Shape
{
```

```
        public override void Draw()
                => Console.WriteLine("Circle Draw");
}

public class Rectangle : Shape
{
        pubilc override void Draw()
                => Console.WriteLine("Rectangle Draw");
}
```

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

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.

Interface

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. Interfaces can have public (by default), private and protected members.

> 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).

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 class Circle : IShape
{
        public void Draw()
                => Console.WriteLine("Circle Draw");
}

public class Rectangle : IShape
{
        publc void Draw()
                => Console.WriteLine("Rectangle Draw");
}
```

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.

Default Implementation

Default implementation in interfaces allows to write an implementation of any method. This is useful when there is a need to provide a single implementation for common functionality.

Let's suppose we need to add new **common** functionality to our already existing interface, which is implemented by many classes. Without default implementation, this operation would create errors, because the method we have added isn't implemented in the classes, and we would need to implement the same operation one by one in each class. Default implementation in interface solves this problem.

```
public interface IShape
{
        void Draw();
        void Finish()
                => Console.WriteLine("Done!");
}

public class Circle : IShape
{
        public void Draw()
                => Console.WriteLine("Circle Draw");
}
```

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

We added the Finish() method with default implementation to our IShape interface and called it without implementing it inside the Circle class.

> Methods with default implementation can be freely overridden inside the class which implements that interface.