Lesson 26 - Inheritance & Polymorphism II

Inheritance

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

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

```
Dog d = new();
```

```
OUTPUT:
Animal created
Dog created
```

> 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.

What if there is no empty base Constructor?

You can call a base constructor explicitly by using `base` keyword.

```
class Animal {
  public string Name { get; set; }
  public Animal(string name) {
    Console.WriteLine("Animal created");
    Name = name;
```

```
  }
}
class Dog: Animal {
  public Dog(string name) : base(name) {
    Console.WriteLine("Dog created");
  }
}
```

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.

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.

```
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.

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:

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

Shape r = new Rectangle();
r.Draw();
```

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

```
Shape shape1 = new Circle();
Shape shape2 = new Rectangle();
Shape shape3 = new Circle();
Shape shape4 = new Rectangle();
Shape shape5 = new Circle();

Shape[] shapes = { shape1, shape2, shape3, shape4, shape5 };
```

```
foreach (Shape shape in shapes)
{
    shape.Draw();
}
```