Part 10 - Polymorphism, or Inherited Methods

Part 10 - Polymorphism, or Inherited Methods

> **Definition: Polymorphism**
>
> The ability for a new object to implement the base functionality of a parent object in a new way.

Two keywords are used to make Polymorphism happen: `virtual` and `override`.

You need to describe a method as `virtual` if you want the ability to `override` its capabilities.

Polymorphism with Rectangle and Square

```
class Rectangle:
    def constructor(width as single, height
as single):
        _width = width
        _height = height

    virtual def GetArea():
        return _width * _height

    _width as single
    _height as single

class Square(Rectangle):
    def constructor(width as single):
        super(width, width)

    override def GetArea():
        return _width * _width

r = Rectangle(4.0, 6.0)
s = Square(5.0)

print r.GetArea()
print s.GetArea()
print cast(Rectangle, s).GetArea()
```

Note that, because Rectangle does **not** have a default, parameterless constructor, we must invoke Rectangle's constructor in the **first** line of Square's constructor (the `super(width, width)` part). Once again: when calling a superclass's constructor, it must be done on the **first line**.

Output

```
24.0
25.0
25.0
```

Even when `casted` to a `Rectangle`, s's `.GetArea()` functioned like if it were a `Square`.

An easier example to see is this:

Simplified Polymorphism Example

```
class Base:
    virtual def Execute():
        return 'From Base'

class Derived(Base):
    override def Execute():
        return 'From Derived'

b = Base()
d = Derived()

print b.Execute()
print d.Execute()
print cast(Base, d).Execute()
```

Output

```
From Base
From Derived
From Derived
```

If I were to leave out the `virtual` and {{override} keywords,

Output w/o virtual

```
From Base
From Derived
From Base
```

This happens because unless the base method is `virtual` or `abstract`, the derived method cannot be declared as `override`.

> ✅ **Recommendation**
>
> Although you do not have to explicitly declare a method as `override` when inheriting from a `virtual` method, you should anyway, in case the signatures of the `virtual` and `overriding` methods do not match.

In order to `override`, the base function must be declared as `virtual` or `abstract`, have the same return type, and accept the same arguments.

Polymorphism is very handy when dealing with multiple types derived from the same base.

Another Polymorphism Example

```
interface IAnimal:
    def MakeNoise()

class Dog(IAnimal):
    def MakeNoise():
        print 'Woof'

class Cat(IAnimal):
    def MakeNoise():
        print 'Meow'

class Hippo(IAnimal):
    def MakeNoise():
        print '*Noise of a Hippo*'

list = []
list.Add(Dog())
list.Add(Cat())
list.Add(Hippo())

for animal as IAnimal in list:
    animal.MakeNoise()
```

Output w/o virtual

```
Woof
Meow
*Noise of a Hippo*
```

Very handy.

Exercises

1. Figure out an exercise

Go on to **Part 11 - Structs**