[0630] Polymorphism and Abstract Classes

In its simplest form, **Polymorphism is simply a fancy name for method overriding/hiding** which happens when a subclass has a method with the same name as a method in its superclass. Here is what a "typical" use of polymorphism looks like:

```
class Bird
{
   public virtual void MakeSound()
   {
      Console.WriteLine("Cheep, cheep...");
   }
}

class Eagle : Bird
{
   public override void MakeSound()
   {
      Console.WriteLine("SCREECH!!");
   }
}

Bird jonathan = new Bird();
Eagle sam = new Eagle();
jonathan.MakeSound();    // writes "Cheep, cheep..."
sam.MakeSound();   // writes "SCREECH!!"
```

In this example, the Eagle class' MakeSound() method is used when called on an Eagle object. When MakeSound() is called on a regular Bird object, the Bird's version of the method is used.

Note that the "virtual" keyword is used by the class designer to say "It's OK if a subclass overrides this method. They don't have to, but if they do, I'll allow it." If the virtual keyword isn't there, overriding the method will result in a compiler error.

The subclass' counterpart to the "virtual" keyword is the "override" keyword. The "override" key is used in front of methods that are supposed to override their superclass' "virtual" methods. **Remember - "override" sits on top of "virtual"!**

Polymorphism is Resolved at Run-Time

Based on the code above, you may be thinking that the compiler determines which version of MakeSound() is used by jonathan and sam. This isn't true. The program determines that at run-time. That means that polymorphism is extremely flexible. A more realistic use of polymorphism is below.

```
class Bird
{
   public virtual void MakeSound()
   {
```

```
      Console.WriteLine("Cheep, cheep...");
   }
}

class Eagle : Bird
{
   public override void MakeSound()
   {
      Console.WriteLine("SCREECH!!");
   }
}

class Hummingbird: Bird
{
   public override void MakeSound()
   {
      Console.WriteLine("Hummmmm....");
   }
}

Bird[] aviary = { new Eagle(), new Hummingbird(), new Bird() };

foreach (Bird b in aviary)
{
   b.MakeSound();
}

// Output:
// SCREECH!!
// Hummmmm....
// Cheep, cheep...
```

Note that a different version of MakeSound() is called each time through the for loop! Also, note that we can store Eagles and Hummingbirds in variables with a datatype of Bird due to the fact they are superclasses of the Bird class.

Abstract Classes and Abstract Methods

It is very common for base classes to be so general purpose that they actually aren't very useful by themselves. These classes have a bunch of methods and abilities, but they don't have everything they need to make complete, useful objects. Instead, they need more information - gained via subclassing - to provide more context before everything will work. These "half-baked" classes are called "Abstract Classes" and, because they have parts missing from them, they cannot be directly instantiated. Until an abstract class is subclassed and the missing parts added, it is useless.

Think of an abstract class as a cookie-cutter (i.e. a template) that is missing some parts. Until those missing parts are added (by a subclass), you can't make cookies!

Inside of an abstract class, the missing methods are also marked "abstract" and, until a subclass provides compatible versions of those missing methods, things won't compile.

Let's go back to our simple Bird example. Thinking about it more, we might decide that having a generic bird go "Cheep, Cheep" when "MakeSound()" is called, is not really correct. In reality, we have no idea what noise a generic bird makes and therefore we can't implement MakeSound() in the Bird class. We really should have this instead:

```
abstract class Bird
{
   public abstract void MakeSound();
}

class Eagle : Bird
{
   public override void MakeSound()
   {
      Console.WriteLine("SCREECH!!");
   }
}

Bird jonathan = new Bird();   // Compile Time Error!  We don't have enough info to create a generic Bird
Eagle sam = new Eagle();
sam.MakeSound();                 // writes "SCREECH!!"
```

That feels more intellectually honest to hard-squawking birds everywhere. Now each kind of bird can get its own unique noise when the Bird class is subclassed. The trade-off is that we can no longer create a generic Bird object like we could before.

Abstract Virtual Methods

Note that the abstract method MakeSound() in the abstract Bird class has no method body at all. It just has a method signature followed by a semi-colon. That means the class designer cannot think of a good default behavior for the method. The subclass programmer MUST come up with a method to fill in the gap before things will work. This is how Java works too.

C# developers being C# developers, they weren't content with that situation. "What if the class designer *did* know of a pretty good default behavior for a method? Some behavior that works say 60% of the time? We MUST create some way for them to do that!" Enter Abstract Virtual Methods (or Virtual Abstract Methods, I'm not sure that order matters).

```
abstract class Bird
{
   public virtual void MakeSound()
   {
      Console.WriteLine("Cheep, cheep...");
   }
}

class Eagle : Bird
{
   public override void MakeSound()
   {
```

```
      Console.WriteLine("SCREECH!!");
   }
}

class Chicken : Bird
{
    // This class deliberately left empty
}

Bird jonathan = new Bird(); // Compile Time Error! We don't have enough info to create a generic Bird
Eagle sam = new Eagle();
Chicken henny = new Chicken();
sam.MakeSound();   // writes "SCREECH!!"
henny.MakeSound();    // writes "Cheep, cheep..."
```

I can hear the complaints already - "Now wait a second Chip! That's what we had before!" Well, yes and no. By changing "abstract" into "virtual" in the MakeSound() method in the Bird class, we are now allowed to provide default code for the method. BUT THE CLASS ITSELF IS STILL ABSTRACT! Which means that we still cannot create a Bird object without first subclassing it (sorry Jonathan).

Generally, to me, Abstract Virtual Methods are rarely used. I'm used to abstract methods being empty. (In reality, I'm used to Interfaces - which are similar but better. We'll get to them soon.)

Overriding vs Shadowing

If you use the keyword "new" instead of the keyword "override" in your subclass, you will "shadow" the base class' method instead of overriding it. "Shadowing" is also called "Hiding" in C# literature. I have looked for hours (literally 5 hours) to try a find a reason why you'd want to do that. I have concluded that there is no reason. None. The book claims it can happen with 3rd party libraries. I doubt it. Let's move on and never speak of shadowed methods again.