Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-five)

5.4: Abstract Methods and Interfaces

There is another, better way to prevent bare Shapes from being created: give the Shape class an *abstract method*. An abstract method is one that has no method body.

```
public abstract class Shape
{
    // other stuff ...

    public abstract void draw();
}
```

Instead of curly braces and code, the method signature is followed by a semicolon, and both the method and the class must be marked "abstract." The abstract method Shape.draw( ) means this:

"A Shape, in general, can draw itself. Exactly how this is accomplished varies between different kinds of Shapes, so I can't provide any code. Each kind of Shape must describe how it is to be drawn for itself."

The Java compiler will now require every subclass of Shape to provide a void draw( ) method. Furthermore, any attempt by any code to create a naked Shape object will fail:

```
Shape s = new Shape();
```

Compiler Says:

```
Test.java:8: class Shape is an abstract class. It can't be instantiated.
Shape s = new Shape();
          ^
```

Objects of abstract types can never be created! On the other hand, Shape variables are still OK:

```
Shape s = new Circle(3.0);
```

This ability (to state that a subclass should provide the body for a method) is so powerful that Java has another important construct to make it more widely usable: the *interface*. An interface is precisely a class that contains nothing but abstract methods and constants:

```
public interface Floatable
{
   public void float();
}

public interface Flyable
{
   public void fly();
}
```

```
public interface Driveable
{
   public void drive();
}
```

A class is allowed to *implement* as many interfaces as it pleases. (Recall that a class can only extend one other class.) (See Footnote below)

```
public class Seaplane implements Floatable, Flyable {...}
public class Sailboat implements Floatable {...}

public class Airliner implements Driveable, Flyable {...}
```

The ellipses would in each case need to be replaced with implementations of the appropriate methods for each type of plane: note how a variable of type Flyable could refer to either a Seaplane or an Airplane, while a Floatable could be a Yacht or a Seaplane, but not an Airliner. This allows us to treat similar things similarly, as appropriate. Read on as I make this excruciatingly clear.

Footnote

Basically, what the rules about extending classes and implementing interfaces boil down to is that any given class can inherit code directly from *at most* one other class. Code inheritance from multiple classes (or more simply put, "multiple inheritance") has long been a nightmare for C++ programmers and implementers of C++ compilers alike. It is extremely hard to understand the rules governing multiple inheritance in C++, and extremely unlikely that any compiler will ever implement all the rules correctly, for the formal specification contains some unresolved ambiguities. The creators of Java have very wisely avoided this entire mess by insisting on only single inheritance of code, allowing multiple inheritance only of abstract methods via interfaces.