Week 05 Research Assignment

Note: All answers are a synthesis of what I've learned from the class materials, unless a source is linked specifically.

Prompt: What are the differences between abstract classes and interfaces? When should you use one over the other?

Response:

To start, abstract classes and interfaces are actually quite similar: they are both much like a standard Java class, except that Java Objects cannot be instantiated directly from either. Instead, they act as templates from which a standard Java class can borrow its structure. In order for this to work, the interface and/or abstract class must be filled with methods that have a signature and a return type, but no associated code (the function body is empty). In addition, any declared properties won't be initialized with a value in either an interface or an abstract class. In order to actually create objects from an interface or an abstract class, a standard Java class must be defined that either "extends" an abstract class or "implements" an interface. Those are the actual keywords used:

```
class myExtendingClass extends myAbstractClass {}

// or

class myImplementingClass implements myInterface {}
```

The major difference between abstract classes and interfaces is that interfaces strictly require any implementing class to contain overriding definitions of all their methods (using the `@Override` keyword), and those methods cannot be defined in the interface. Abstract classes, on the other hand, allow any extending class to have only those methods defined that the programmer deems necessary for the class to function, and some of those methods could already have been defined in the abstract class. A class derived from an interface can benefit from multiple inheritance: it can implement more than one interface, as long as all methods and properties of all interfaces it implements are present and defined within its code. A class derived from an abstract class, however, can only extend a single abstract class: multiple inheritance will throw an exception. This is what gives us one situation in which using interfaces over abstract classes is useful: if we want to create a class derived from multiple interfaces, we can. Of course, we might prefer an abstract class in the situation that we don't want or need to implement all its methods, and we know some of those methods will be the same across all extending classes (and can therefore define them on the original abstract class).

Prompt: What is the relationship between a class and an Object?

Response:

A class acts as a "template" for objects, which "inherit" properties and methods from a class. Any methods/properties defined in a class--unless they're declared with the `static` keyword--can't be called or referenced on the class itself, but rather on a unique Object that is instantiated from the class. Classes are generally useful when the programmer knows they will be using an Object of generally the same structure in different situations, but that Object will probably contain different data than other Objects that are members of the same class. For example:

```
public class myClass {

    public int myInt;

    public void myVoid(){
        System.out.println("Welcome to my void!")
    }

    //the class constructor (the method that will execute when we use the "new" keyword):

    public myClass(int anInt){
        this.myInt = anInt;
    }

    public static void myStaticMethod(){
        System.out.println("I can't hear you, too much static");
    }
}

// and, in another file:

public class myApp {
    public void main(String[] args){

        // call the constructor with an int value for the "myInt" property included
        myClass myClassInstance = new myClass(5);

        myClassInstance.myVoid();
        // will print "Welcome to my void!" to the console

        System.out.println(myClassInstance.myInt);
        // will print 5 to the console.
```

```
        System.out.println(myClass.myStaticMethod());
        // will print "I can't hear you, too much static" to the console

        System.out.println(myClassInstance.myStaticMethod());
        /* will throw an error. You can't call a class's static methods
        on instances of that class: they must be called on the class itself. */
    }
}
```