COM6516 Object oriented programming and software design: Practical session 4

The three aims of this practical are to work though the material on abstract classes and interfaces that we covered in the last lecture. You should work through this worksheet on your own.

Please do not introduce packages in your code (Java expects them to be placed in directories with names matching package names and CLASSPATH be set to point to that directory).

Part 1: Review example code
Make sure you have completed all the programming exercises from last week. Look at the sample solutions at Blackboard to understand them.

Part 2: Inheritance programming task
Implement a superclass called `Person`, with two subclasses called `Student` and `Tutor` that inherit from `Person`. A `Person` should have a name and date of birth. A `Student` should have a course, and a `Tutor` should have an office (you are free to implement office number as a `String` or an `int`). For each class write a constructor, and a `toString` method which displays the instance fields. Write and implement a test program to create instances of the `Person`, `Student`, and `Tutor` class. Within the `Student` and `Tutor` class, write a `toString` method that overrides the `toString` method in the `Person` class (for example by displaying different information).

See the Oracle tutorial on overriding if you are not sure how to do this –
http://docs.oracle.com/javase/tutorial/java/IandI/override.html
What is the advantage of using the `@Override` annotation?

Part 3: Polymorphism
Review the slides on polymorphism from the lecture, and also take a look at the Oracle tutorial –
http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Using the example code provided for the `Animal`, `Cow`, and `Pig` classes, write a new class called `Sheep` with a `talk` method that returns "`Baaa!`". Write a new test class `AnimalTest` that creates an array of `Animal` objects, which refer to `Cow`, `Pig`, and `Sheep`. Implement a loop, which invokes the `talk` method on each object in the array.

Part 4: Abstract classes
Implement the `Animal` class as an abstract class. What problem does this create for the `AnimalTest` class? How can you adapt `AnimalTest` so that it still exploits polymorphism? Is it sensible to define the `Animal` class as abstract?

Part 5: Interfaces
Write a `Drawable` interface for the `Shape`, `Circle`, `Rectangle` and `ShapeDemo` classes provided as examples. Note that `ShapeDemo` uses the `EasyGraphics` class, which is part of the sheffield package. `Drawable` should specify that implementing classes must contain a `draw` method. Should the `Drawable` interface be implemented by the `Shape` class or the `Circle` and `Rectangle` classes?

By sub-classes

Part 6: Abstract classes or Interfaces – something to think of rather than implement

An interface can be thought of as an extreme case of an abstract class, but there are some important consequences that follow from using an interface rather than an abstract class. The following example was originally published by Sun as a JDC Tech Tip in October 2001, but seems to have disappeared from the Oracle documentation. The following material is taken from a digest of this example available at http://users.cs.cf.ac.uk/O.F.Rana/jdc/november7-01.txt

Consider the code to implement the following classes, together with a test class to create `Time` and `HoursMinutes` objects, and to display the output of the `getMinutes` method.

```
abstract class Time {
    public abstract int getMinutes();
}

class Days extends Time {
    private int days;
    public int getMinutes() {
        return days * 24 * 60;
    }
}

class HoursMinutes extends Time {
    private int hours;
    private int minutes;
    public int getMinutes() {
        return hours * 60 + minutes;
    }
}
```

Now think how it changes if `iTime` is an interface rather than an abstract class (turn to next page). The use of a small 'i' to indicate interface (e.g. `iTime`) in this example is a convention that is sometimes used, but is often regarded as unhelpful. It is usually better to give an interface an adjective as a name (e.g. `Drawable`).

```
interface iTime {
    int getMinutes();
}

class iDays implements iTime {
    private final int days;
    public iDays(int days) {
        this.days = days;
    }
    public int getMinutes() {
        return days * 24 * 60;
    }
}

class iHoursMinutes implements iTime {
    private final int hours;
    private final int minutes;
    public iHoursMinutes(int hours, int minutes) {
        this.hours = hours;
        this.minutes = minutes;
    }
    public int getMinutes() {
        return hours * 60 + minutes;
    }
}
```

So what's the difference between using abstract classes and interfaces in the example above? One difference is that an abstract class is easier to evolve over time. Suppose that you want to add a method:

```
public int getSeconds();
```

to `Time`. If `Time` is an abstract class, you can say:

```
public int getSeconds() {
    return getMinutes() * 60;
}
```

In other words, you provide a partial implementation of the abstract class. Doing it this way means that subclasses of the abstract class do not need to provide their own implementation of `getSeconds` unless they want to override the default version. If `Time` is an interface, you can say:

```
interface Time {
    int getMinutes();
    int getSeconds();
}
```

But you're not allowed to implement `getSeconds` within the interface. This means that all classes that implement `Time` are now broken, unless they are fixed to include a `getSeconds` method. So if you want to use an interface in this situation, you need to be absolutely sure that you've got it right the first time. That way you don't have to add to the interface at a later time, thereby invalidating all the classes that use the interface.