Problems relating to Unit 4: Interfaces

1. Terminology of interfaces:

(a) Explain briefly how an **interface** differs from a concrete class. An interface is a single or collection of abstract methods where a concrete class in a class that models something

(b) Java supports **multiple interface inheritance**. Explain briefly what this means. Give any suitable example you like to support your answer. No code is required here.

You can implement ActionListener and WindowListener at the same time.

(c) Give *one* way in which an interface differs from an abstract class.

An interface is implemented and an abstract class is extended

1. The Shape interface. Copy the files **Shape.java**, **Point.java** and **Circle.java** to your H: drive. Open the interface file **Shape.java** and answer the following questions in relation to it:
2. The interface defines no attributes of its own. Could it? Try it out by attempting to give the Shape interface a **private** String attribute called “dummy”. What does this tell you about interfaces? Interfaces cannot have attributes
3. Change the access mode on the attribute to **public** and recompile. What happens now? Initialize the value of the attribute to “test” and recompile. What does this tell you about interface attributes? A Variable declared in an interface must be initiated and it can then be used as a constant
4. Recall that it is possible with classes to set them up so that they cannot be derived from, through the **final** keyword. Do you think it should be possible to do the same with interfaces? Alter the source code to determine if it is possible. If an interface is declared final you cannot implement the method as it cannot be changed
5. The methods inside the interface are all public and all abstract. Do you think they could be made private or protected rather than public? Try out these possibilities.

Interface methods have to be public so when the interface is implemented these methods are available

1. Is it possible for an interface to contain any methods that have an implementation? Try it out by adding the following method to the interface:

public double tester(){ return 2.5;}

No an interface method cannot have a body

1. Write an interface Enemy which would be appropriate for a game system, containing the methods you think would be common to all the kinds of enemies a game might feature. Make sure it compiles. Then write any class you like which implements Enemy, and a tester which creates one or two Enemy objects.
2. Take a look at **Point.java** and answer the following questions based on it:
3. The class Point, by its class definition header, “implements” the interface called Shape. What does it actually mean for a class to “implement” an interface. Can a class that implements an interface get away with only implementing a few of the abstract methods contained in the interface? Try it out by commenting out the area() method in the Point class. No if an interface is implemented then all the methods of that interface have to be implemented
4. Notice that the Point class overrides the methods defined in the Shape interface by providing an implementation for them. Could the overridden methods in Point be made private or protected, even though they are public (by default) in the interface? Try it out. No the method needs to be public
5. Examine **Circle.java** and answer the following questions:
6. Is Circle a base class or a derived class? A Derived Class
7. Notice in the Circle constructor that the super reference is used as follows:

super(a,b);

in order to set up the coordinates of the center of the circle. Would it have been possible to use the following code instead? Explain.

setPoint(a,b);

Yes because Point is extended all the methods of Point are available

Would it have been possible to use the following code instead? Explain.

x=a;

y=b;

No as x and y are private attributes of the Point class

1. Note that the call to super() takes place as the first line of the Circle constructor. Could it take place elsewhere in the constructor? Could it be called from any method other than the constructor? The call to super always has to be the first statement in the constructor it cannot be called from any where else

25. Examine the driver program **ShapeTester.java** and answer the following questions:

1. Explain very briefly the purpose of the line of code

Shape arrayOfShapes[] = new Shape[2];

An interface cannot be instantiated but the new keyword appears here. Why does this not violate the rule? This is not an instance of shape but an array of shapes

1. The driver contains **polymorphic** **behaviour**. Explain where this behaviour exists and how it operates. Where the array of shapes is created each shape can be of different type with the same initial attributes
2. The driver contains **dynamic method binding**. Give any example of where dynamic method binding exists in the driver. Dynamic binding occurs here == shapes = new Circle();

26. Adding a class to a hierarchy: Derive a subclass Cylinder from Circle: it will have a radius and an ‘x’ and ‘y’ as before, but also a height. Look up the formula for the volume of a Cylinder, then add a volume() method to the Cylinder class. Add some code to ShapeTester to make sure your Cylinder class works as expected.

27. Writing all the classes for a hierarchy, including an interface and an abstract class: Consider the following **inheritance hierarchy diagram**

### **SecondaryPupi**l

## PrimaryPupil

## Pupil

## Child

Write classes/interfaces to implement this hierarchy, assuming the following:

**Child** is an **interface** that defines the following **abstract methods:** *getName()*, *getAddress()*, *getDateOfBirth(), setName(), setAddress(), setDateOfBirth().*

**Pupil** is an **abstract class** that contains the attributes *name, address, dateOfBirth, school, className*, implements Child, and has the following methods of its own: mutator and accessor method for class and school, an abstract accessor method called getCategory() which will be implemented by its subclasses, a **multi-argument constructor**, a **no-argument constructor** and a **toString**() method for displaying the state of a generic Pupil object (use the accessors to refer to the 6 attributes (must include getCategory() here) **indirectly**).

**PrimaryPupil** is a concrete class that inherits from Pupil as well as defining a teacherName attribute of its own, with mutator and accessor, a multi-argument constructor (which invokes the Pupil constructor), a no-argument constructor, an implementation of getCategory() which returns the String “Primary Pupil” and toString().

**SecondaryPupil** is a concrete class that inherits from Pupil as well as defining an attribute of its own: Subjects (a LinkedList of Strings), with accessor and mutator. It should have a multi-argument constructor, a no-argument constructor that initializes a SecondaryPupil object with a set of default values, an implementation of getCategory() that returns the String “Secondary Pupil” and a toString() method that can be used to display the values of all the features associated with a SecondaryPupil. .

Once all your classes and interfaces compile, write a tester which tests all the functionality of the two concrete classes, and includes some polymorphic behaviour involving Pupil and Child to ensure that your hierarchy is correctly implemented.

28. Design a hierarchy of classes for modeling the invaders and the defenders in a game situation. For example, an abstract ‘Invader’ class which contains any attributes common to all kinds of enemy, and two concrete implementations such as terrorist and extra-terrestrial (choose your own); also an abstract ‘AbstractDefender’ class and two concrete implementations like footsoldier and tankdriver (again, choose your own). Show the hierarchy as class diagrams, and consider whether your invader class should implement the Enemy interface (see Q22), and whether it would be a good idea to include a ‘Defender’ interface. Write the upper levels of the hierarchy, making sure they compile, then one concrete invader implementation class and one concrete defender class, and an application which will let you add some defenders and some invaders, and some buttons to allow you to pit them against each other. This exercise is intentionally left very open so that you can approach it in different ways.