Exercise 1:

Examine the following codes and draw the class diagram.

```java
abstract public class Animal {
   abstract public void greeting();
}

public class Cat extends Animal {
   @Override
   public void greeting() {
      System.out.println("Meow!");
   }
}

public class Dog extends Animal {
   @Override
   public void greeting() {
      System.out.println("Woof!");
   }
   public void greeting(Dog another) {
      System.out.println("Wooooo0ooof!");
   }
}

public class BigDog extends Dog {
   @Override
   public void greeting() {
      System.out.println("Woow!");
   }
   @Override
   public void greeting(Dog another) {
      System.out.println("Woooooo0000ww!");
   }
}
```

Explain the outputs (or error) for the following test program.

```java
public class TestAnimal {
   public static void main(String[] args) {
      // Using the subclasses
      Cat cat1 = new Cat();
      cat1.greeting();
      Dog dog1 = new Dog();
      dog1.greeting();
      BigDog bigDog1 = new BigDog();
      bigDog1.greeting();
      // Using Polymorphism
      Animal animal1 = new Cat();
      animal1.greeting();
      Animal animal2 = new Dog();
      animal2.greeting();
      Animal animal3 = new BigDog();
      animal3.greeting();
      Animal animal4 = new Animal();
      // Downcast
      Dog dog2 = (Dog)animal2;
      BigDog bigDog2 = (BigDog)animal3;
      Dog dog3 = (Dog)animal3;
      Cat cat2 = (Cat)animal2;
      dog2.greeting(dog3);
      dog3.greeting(dog2);
      dog2.greeting(bigDog2);
      bigDog2.greeting(dog2);
      bigDog2.greeting(bigDog1);
   }
}
```
Exercise 2:
Suppose that we have a set of objects with some common behaviors: they could move up, down, left or right. The exact behaviors (such as how to move and how far to move) depend on the objects themselves. One common way to model these common behaviors is to define an interface called Movable, with abstract methods moveUp(), moveDown(), moveLeft() and moveRight(). The classes that implement the Movable interface will provide actual implementation to these abstract methods.

Let's write two concrete classes - MovablePoint and MovableCircle - that implement the Movable interface.

```
public interface Movable { // saved as "Movable.java"
  public void moveUp();
  ......
}

public class MovablePoint implements Movable { // saved as "MovablePoint.java"
  // instance variables
  int x, y, xSpeed, ySpeed; // package access

  // Constructor
  public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
    this.x = x;
    ......
  }
  ......

  // Implement abstract methods declared in the interface Movable
  @Override
  public void moveUp() {
    y -= ySpeed; // y-axis pointing down for 2D graphics
  }
  ......}
```

The code for the interface Movable is straightforward.

For the MovablePoint class, declare the instance variable x, y, xSpeed and ySpeed with package access as shown with ‘~’ in the class diagram (i.e., classes in the same package can access these variables directly). For the MovableCircle class, use a MovablePoint to represent its center (which contains four variable x, y, xSpeed and ySpeed). In other words, the MovableCircle composes a MovablePoint, and its radius.

```
public class MovableCircle implements Movable { // saved as "MovableCircle.java"
  // instance variables
  private MovablePoint center; // can use center.x, center.y directly because they are package accessible
  private int radius;

  // Constructor
  public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {
    center = new MovablePoint(x, y, xSpeed, ySpeed);
    ......
  }
  ......

  // Implement abstract methods declared in the interface Movable
  @Override
  public void moveUp() {
    center.y += center.ySpeed;
  }
  ......
```
public class MovableCircle implements Movable { // saved as "MovableCircle.java"

    // instance variables
    private MovablePoint center;  // can use center.x, center.y directly
                                    // because they are package accessible
    private int radius;

    // Constructor
    public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {
        // Call the MovablePoint's constructor to allocate the center instance.
        center = new MovablePoint(x, y, xSpeed, ySpeed);
        .......
    }
    .......

    // Implement abstract methods declared in the interface Movable
    @Override
    public void moveUp() {
        center.y -= center.ySpeed;
    }
    .......
}

Write a test program and try out these statements:

Movable m1 = new MovablePoint(5, 6, 10);  // upcast
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);

Movable m2 = new MovableCircle(2, 1, 2, 20); // upcast
System.out.println(m2);
m2.moveRight();
System.out.println(m2);

Write a new class called MovableRectangle, which composes two MovablePoints (representing the top-left and bottom-right corners) and implementing the Movable Interface. Make sure that the two points has the same speed.

![Diagram of Movable, MovablePoint, and MovableRectangle classes]

What is the difference between an interface and an abstract class?

Reference
http://www.ntu.edu.sg/home/ehchua/programming/java/J3f_OOPExercises.html - zz-4.2