Exercise Set: Type Hierarchy, Polymorphism and Dispatching

In this exercise set, we have marked questions we think are harder than others with a **[‡]**. We have also marked questions for which solutions are provided at the end of the set (**[SP]**). To check solutions for other questions than those marked with **[SP]**, ask one of the instructors or TAs or post a question to Piazza!

1. Answer the following questions with true or false and explain your choice in one sentence. **[SP]**

a) A supertype extends the behaviour of a subtype.

b) The apparent type of the variable `aList` defined in the following statement is `List<String>`.

```
List<String> aList = new ArrayList<String>();
```

c) The actual type of the variable `aList` defined in the statement of question 1.b) is `List<String>`.

d) A class in Java can inherit from only one superclass, but can implement multiple interfaces.

e) A class extending another class has to override all methods of the superclass.

f) A class (that is not abstract) implementing an interface has to provide implementations for each method defined in the interface.

2. Assume there is a class `Animal` and a class `Dolphin` that is a subclass of `Animal`. `Dolphin` defines a method `eat()` that is not defined in `Animal`. Is the following code correct and why or why not? **[‡]**, **[SP]**

```
Animal anAnimal = new Dolphin();
anAnimal.eat();
```

3. Assume you are implementing a drawing program that allows a user to draw a variety of figures/objects. The figures/objects that the user should be allowed to draw are Line, Circle, Square, ColouredLine, LineWithArrow, Rectangle, Ellipse and ColouredEllipse. All of these types are subtypes of the supertype Figure.

a) Think about which type hierarchy would work well and draw a picture of it. Briefly explain how you chose the type hierarchy.**[‡]**

b) Why do you use inheritance in this example instead of creating each type as a stand-alone class?

4. Given the classes `Figure` and `SmallFigure` as defined below.

```
public class Figure{
	...
	//pre-condition: width and height are positive integers
	//post-condition: returns a Graphics object that has the figure drawn
onto it
	public Graphics drawFigure(int width, int height) {...}
	...
}

public class SmallFigure extends Figure{
	...
	//pre-condition: 0 < width < 100 and 0 < height < 250
	//post-condition: returns a Graphics object that has the small figure
drawn onto it
	public Graphics drawFigure(int width, int height) {...}
	...
}
```

a) is an object of type `SmallFigure` substitutable for an object of the supertype `Figure`? Explain why or why not! **[SP]**

For the remaining parts of this question, assume that method `drawFigure` in class `SmallFigure` does not have a call to the superclass method.

b) When you run the following code, the `drawFigure` method of which class is being executed? (explain why)

```
Figure aFigure = new SmallFigure();
aFigure.drawFigure(40, 38);
```

c) When you run the following code, the `drawFigure` method of which class is being executed? (explain why)

```
SmallFigure aSmallFigure = new SmallFigure();
aSmallFigure.drawFigure(40, 38);
```

5. What does it mean to override a method and why can it be useful to allow the overriding of methods? (Explain briefly.)

6. What is the difference between overloading and overriding of a method? (Explain briefly.) **[SP]**

SOLUTIONS:

1.
a) False.
A subtype extends the behaviour of a supertype.

b) True.
The apparent type is the declared type of the variable, which is `List` in this case.

c) False.
The actual type of the variable is `ArrayList<String>`.

d) True.
In Java, multiple inheritance is not allowed, however by implementing multiple interfaces, different parts of a Java software system can use an object from different perspectives.

e) False.
A class can override methods of its supertype but does not have to.

f) True.
If a class implements an interface, it means that it provides an implementation for the supertype; as an interface does not provide an implementation for any of the methods it defines, the implementing class has to provide them.

2. No, the code is not correct. As the method `eat` is only defined on objects of the class `Dolphin` and the variable `anAnimal` is of the apparent type `Animal`, the method `eat` cannot be called on the object `anAnimal`.

4a. An object of type `SmallFigure` is not substitutable for an object of the supertype `Figure`. This is because the precondition on the overridden `drawFigure` method in the `SmallFigure` class has been strengthened as compared to the corresponding precondition in the superclass. Note that if the precondition on the `drawFigure` method in the supertype is true, this does not imply that the precondition on the `drawFigure` method in the subtype is true – hence the precondition has been strengthened.

Consider the following illustration of the problem. The code below is written to the specification of the `Figure` class:

```
Figure myFigure = new Figure();
myFigure.drawFigure(200, 400);      // meets precondition
                                    // specified in Figure class
```

Now imagine that we replace the `Figure` object with a `SmallFigure` object:

```
Figure myFigure = new SmallFigure();
myFigure.drawFigure(200, 400);      // Oops! Fails to meet precondition
                                    // specified in SmallFigure class
```

If a `SmallFigure` object were to be substitutable for a `Figure` object, the code above should work. However, we cannot expect that it will work because the parameters passed to `drawFigure` violate the precondition specified in the `SmallFigure` class. Hence, the behaviour of `drawFigure` is not specified.

6. Method overloading occurs when two methods in the same class have the same name but different parameter lists. We use method overloading when we want to provide implementations of the same behaviour for different input parameters. Method overriding occurs when a subclass provides a method of the same name and same signature as a method in a superclass. We use method overriding when we want to extend, or provide a different implementation of, a behaviour inherited from a supertype.