Behavioral Subtyping and Behavioral Contracts

This recitation has two parts. In the first part your goal is to better understand behavioral subtyping, which provides a way to characterize good inheritance hierarchies. In the second part, you will learn about Java's `java.lang.Object` behavioral contracts, particularly the `equals(Object obj)` method.

Behavioral Subtyping

The following must hold for a class `B` to be a behavioral subtype of class `A`:

1. `B` must have the same or stronger invariants than `A`.
2. Overridden methods in `B` must have weaker or the same preconditions than in `A`.
3. Overridden methods in `B` must have the same or stronger postconditions than in `A`.

For each of the supertype/subtype pairs below, is the subtype a behavioral subtype?

1. Consider a `Beverage` which can contain additional ingredients:

```
public class Beverage {
    //@invariant ingredients != null;
    protected List<Ingredient> ingredients;

    //@ensures ingredients != null;
    public Beverage() {
        ingredients = new ArrayList<>();
    }

    //@requires a != null;
    //@ensures ingredients.contains(a);
    public void addIngredient(Ingredient a)
    {
        ingredients.add(a);
    }
}

public interface Ingredient {
    String getDescription();
}
```

```
public class Lemon implements Ingredient {
    public String getDescription() {
        return "lemon";
    }
}

public class Milk extends Beverage {
    //Milk will curdle with lemons!
    //@requires a != null;
    //@ensures ingredients.contains(a) ||
                (a instanceof Lemon);
    public void addIngredient(Ingredient a) {
        if(!(a instanceof Lemon)) {
            super.addIngredient(a);
        }
    }
}
```

   Is `Milk` a behavioral subtype of `Beverage`? Why or why not?

2. Consider a `Point` and `ColorPoint` class which can draw themselves to a `Canvas` below:

```
public class Point {
    //@invariant px > 0 && py > 0;
    private int px, py;

    //@requires x > 0 && y > 0;
    //@ensures px == x && py == y;
    public Point(int x, int y) {
        px = x;
        py = y;
    }

    //@requires canvas != null;
    public void draw(Canvas canvas) {
        /* ... */
    }
}
```

```
public class ColorPoint extends Point {
    //@invariant px > 0 && py > 0 &&
                 pc != null;
    private Color pc;

    //@requires x > 0 && y > 0 && c != null;
    //@ensures px == x && py == y && pc == c;
    public ColorPoint(int x, int y, Color c)
    {
        super(x, y);
        pc = c;
    }

    //@requires canvas != null &&
                pc != Color.WHITE;
    public void draw(Canvas canvas)
    { /* ... */ }
}
```

Is `ColorPoint` a behavioral subtype of `Point`? Why or why not?

3. Consider the following *immutable* implementations of a `Rectangle` and `Square` class, noting that these immutable implementations are different from the `Rectangle` and `Square` we saw in lecture:

```
public class Rectangle {
    //@ invariant w > 0 && h > 0;
    protected final int w, h;

    //@ requires aw > 0 && ah > 0;
    //@ ensures w == aw && h == ah;
    public Rectangle(int aw, int ah) {
        w = aw;
        h = ah;
    }

    //@ requires factor >= 2;
    //@ ensures \result.w == w+factor &&
                \result.h == h+factor;
    public Rectangle scale(int factor) {
        return new Rectangle(w+factor,
                             h+factor);
    }

    //@ requires a >= 0;
    //@ ensures \result.w == w+a &&
                \result.h == h;
    public Rectangle wider(int a) {
        return new Rectangle(w+a,h);
    }
}
```

```
public class Square extends Rectangle {
    //@ invariant w > 0;
    //@ invariant w == h;

    //@ requires aw > 0;
    //@ ensures h == w && w == aw;
    public Square (int aw) {
        super(aw, aw);
    }

    //@ requires factor >= 0;
    //@ ensures \result.w == w+factor &&
                \result.h == h+factor;
    public Square scale(int factor) {
        return new Square(w+factor);
    }
}
```

Is `Square` a behavioral subtype of `Rectangle`? Why or why not?

`java.lang.Object` Behavioral Contracts

All Java objects inherit from `java.lang.Object`. Some of the commonly-used/overriden methods include `String toString()`, `boolean equals(Object obj)`, `int hashCode()`, and `Object clone()`.

It is common practice for Java programmers to override these methods to suit the behavior of their own classes. However, the behavioral contracts for each method must be observed. Today, let us consider the contracts for `.equals` and `.hashCode`.

The `.equals(Object obj)` contract:

1. An equivalence relation
   (a) Reflexive: $\forall$ `x` `x.equals(x)`
   (b) Symmetric: $\forall$ `x,y` `x.equals(y)` if and only if `y.equals(x)`
   (c) Transitive: $\forall$ `x,y,z` `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`
2. Consistent. Invoking `x.equals(y)` repeatedly returns the same value unless `x` or `y` is modified.
3. `x.equals(null)` is always false.
4. `.equals()` always terminates and is side-effect free (does not change state of program).

The `.hashCode()` contract:

1. Consistent. Invoking `x.hashCode()` repeatedly returns the same value unless `x` is modified.
2. `x.equals(y)` implies `x.hashCode() == y.hashCode()`.

Because the `.hashCode()` behavioral contract depends on `.equals()`, you must always override `.hashCode()` when you override `.equals()`, and vice versa.

Exercise: Implement equality-checking for a `Point` class

Task: Write all the code needed to correctly implement equality-checking for this simple `Point` class.

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Consider this subclass of `Point` called `ColorPoint`:

```
public class ColorPoint extends Point {
    private final Color color;
    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint)) return false;
        ColorPoint cp = (ColorPoint) obj;
        return this.color.equals(cp.color) && super.equals(obj);
    }

    public int hashCode() {
        return 31 * super.hashCode() + color.hashCode();
    }
}
```

For the following questions, consider this use of the `Point` and `ColorPoint` classes. You may assume that `Color.equals()` correctly meets the behavioral contracts of the `java.lang.Object` class.

```
Point p = new Point(2, 42);
ColorPoint cp1 = new ColorPoint(2, 42, Color.BLUE);
ColorPoint cp2 = new ColorPoint(2, 42, Color.MAUVE);
```

(a) Is `p.equals(p)` true?

(b) Is `p.equals(cp1)` true?

(c) Is `cp1.equals(p)` true?

(d) Is `p.equals(cp2)` true?

(e) Is `cp1.equals(cp2)` true?

Question: Based on the behavioral contract of `java.lang.Object`, is the `Point` class a behavioral subtype of the `Object` class? If not, explain why not and change your `Point` implementation so it is.

Question: Based on the behavioral contract of `java.lang.Object`, is the `ColorPoint` class a behavioral subtype of the `Object` class? If not, explain why not.