CS2030S Programming Methodology II
Semester 2 2021/2022

26 & 27 January 2022
Problem Set #1 Suggested Guidance

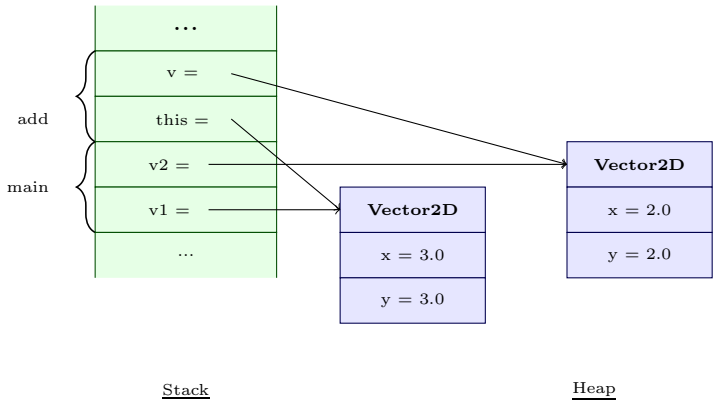
1. Consider the following definition of a `Vector2D` class:

```
class Vector2D {
    private double x;
    private double y;

    Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    void add(Vector2D v) {
        this.x = this.x + v.x;
        this.y = this.y + v.y;
        // line A
    }
}
```

   (a) Suppose that the following program fragment is in a `main` method, show the content of the stack and the heap when the execution reaches the line labelled `A` above.

```
Vector2D v1 = new Vector2D(1, 1);
Vector2D v2 = new Vector2D(2, 2);
v1.add(v2);
```

   Label your variables and the values they hold clearly. You can use arrows to indicate object references. Draw boxes around the stack frames of the methods `main` and `add`, and label them.



Unlike languages like C, Java has automatic memory management. The garbage collector "cleans up" or reclaims memory taken up by unreferenced objects in the heap.

(b) Suppose that the representation of `x` and `y` have been changed to a double array:

```
class Vector2D {
    private double[] coord2D;

    ...
}
```

i. What changes do you need for the other parts of class `Vector2D`

```
class Vector2D {
    private double[] coord2D;

    Vector2D(double x, double y) {
        this.coord2D = new double[]{x, y};
    }

    void add(Vector2D v) {
        coord2D = new double[] {
            this.coord2D[0] + v.coord2D[0],
            this.coord2D[1] + v.coord2D[1]};
    }
}
```

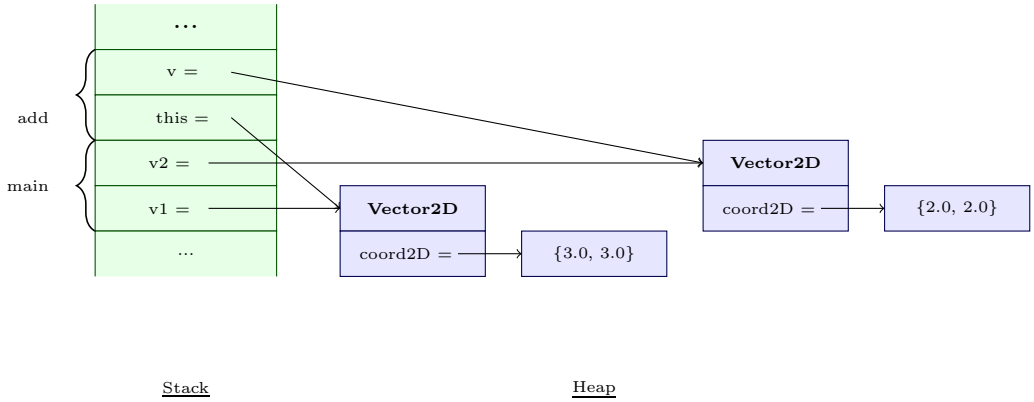
Note that in this example we created a new double array in the add method. You could have also just assigned the elements of the double array in place:

```
this.coord2D[0] = this.coord2D[0] + v.coord2D[0];
this.coord2D[1] = this.coord2D[1] + v.coord2D[1];
```

ii. Would the program fragment in 1a above be valid?

Yes, the program fragment is still valid. The lower-level implementation of how the x and y coordinates are stored and operated on in `Vector2D` *is encapsulated from other clients.*

Show the content of the stack and the heap when the execution reaches the line labelled `A` again.



2. Study the following `Point` and `Circle` classes.

```
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

public class Circle {

    private Point centre;
    private int radius;

    public Circle(Point centre, int radius) {
        this.centre = centre;
        this.radius = radius;
    }

    @Override
    public boolean equals(Object obj) {
        System.out.println("equals(Object) called");
        if (obj == this) {
            return true;
        }
        if (obj instanceof Circle) {
            Circle circle = (Circle) obj;
            return (circle.centre.equals(centre) && circle.radius == radius);
        } else {
            return false;
        }
    }

    public boolean equals(Circle circle) {
        System.out.println("equals(Circle) called");
        return circle.centre.equals(centre) && circle.radius == radius;
    }
}
```

Given the following program fragment,

```
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = new Circle(new Point(0, 0), 10);
Object o1 = c1;
Object o2 = c2;
```

what is the output of the following statements?

(a) `o1.equals(o2);`

(b) `o1.equals((Circle) o2);`

(c) `o1.equals(c2);`

(d) `o1.equals(c1);`

(e) `c1.equals(o2);`

(f) `c1.equals((Circle) o2);`

(g) `c1.equals(c2);`

(h) `c1.equals(o1);`

```
jshell> o1.equals(o2)
equals(Object) called
$.. ==> false

jshell> o1.equals((Circle) o2)
equals(Object) called
$.. ==> false

jshell> o1.equals(c2)
equals(Object) called
$.. ==> false

jshell> o1.equals(c1)
equals(Object) called
$.. ==> true

jshell> c1.equals(o2)
equals(Object) called
$.. ==> false

jshell> c1.equals((Circle) o2);
equals(Circle) called
$.. ==> false

jshell> c1.equals(c2)
equals(Circle) called
$.. ==> false

jshell> c1.equals(o1)
equals(Object) called
$.. ==> true
```

Invoking the `equals` *method through a variable of compile-time type* `Object` *would execute the* `equals(Object)` *method of* `Object`*. This method can be overridden by the overriding method of the same name in the sub-class* `Circle`*.*

The only time that the overloaded method `equals(Circle)` *can be executed is when the method is invoked through a target with compile-time type* `Circle`*, and run-time type is also* `Circle` *(as can be seen in the output of the code excerpt* `c1.equals(c2)`*).*

The dynamic binding process to determine which method gets invoked happened in two steps as can be seen in the notes.

The output of `true` *or* `false` *largely depends on the presence of an overriding* `equals` *method in the* `Point` *class.*