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

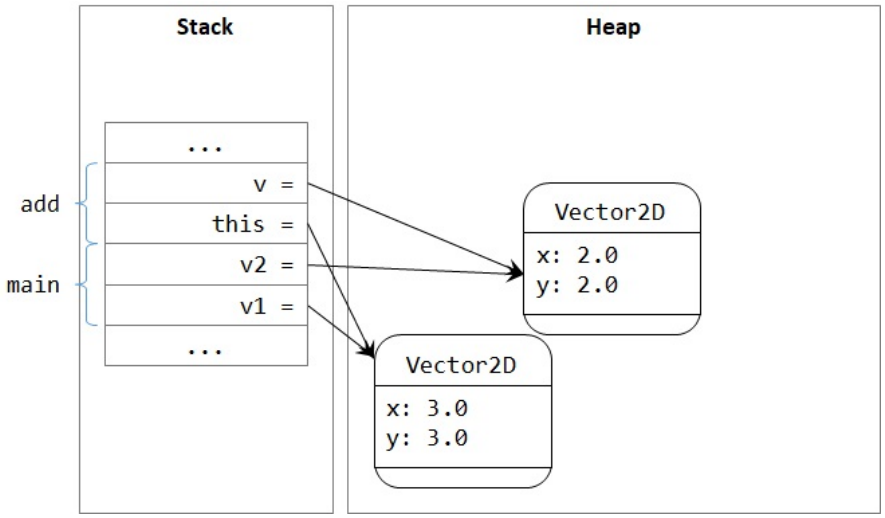
```
1  class Vector2D {
2    private double x;
3    private double y;
4
5    Vector2D(double x, double y) {
6      this.x = x;
7      this.y = y;
8    }
9
10   void add(Vector2D v) {
11     this.x = this.x + v.x;
12     this.y = this.y + v.y;
13     // line A
14   }
15 }
```

(a) Suppose 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.

```
1  Vector2D v1 = new Vector2D(1, 1);
2  Vector2D v2 = new Vector2D(2, 2);
3  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.

Suggested Guide:



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 the representation of `x` and `y` have been changed to a `double` array.

```
1  class Vector2D {
2      private double[] coord2D;
3      // code omitted
4  }
```

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

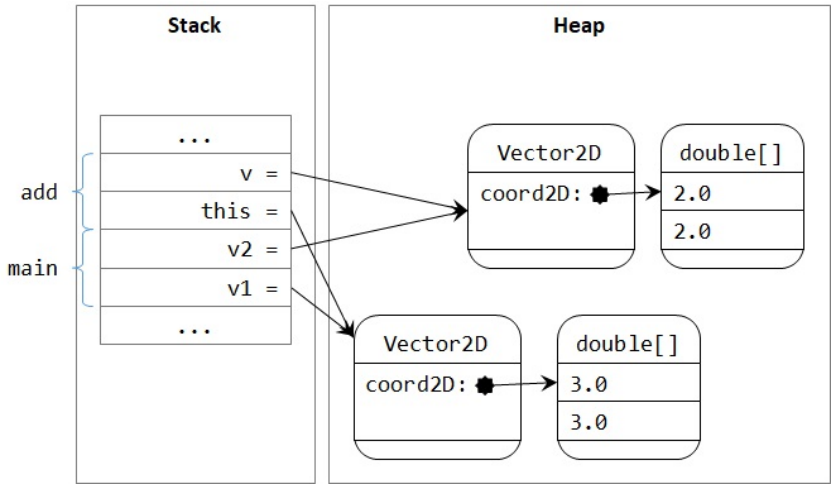
Suggested Guide:

```
1  class Vector2D {
2    private double[] coord2D;
3
4    Vector2D(double x, double y) {
5      this.coord2D = new double[]{x, y};
6    }
7
8    void add(Vector2D v) {
9      this.coord2D = new double[] {
10         this.coord2D[0] + v.coord2D[0],
11         this.coord2D[1] + v.coord2D[1]
12     };
13     /* Alternatively:
14     this.coord2D[0] = this.coord2D[0] + v.coord2D[0];
15     this.coord2D[1] = this.coord2D[1] + v.coord2D[1];
16     */
17   }
18 }
```

ii. Would the program fragment in 1a above be valid? Show the content of the stack and the heap when the execution reaches the line labelled `A` again.

Suggested Guide:

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.



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

```
1  public class Point {
2    private double x;
3    private double y;
4
5    public Point(double x, double y) {
6      this.x = x;
7      this.y = y;
8    }
9  }
10
11 public class Circle {
12   private Point centre;
13   private int radius;
14
15   public Circle(Point centre, int radius) {
16     this.centre = centre;
17     this.radius = radius;
18   }
19
20   @Override
21   public boolean equals(Object obj) {
22     System.out.println("equals(Object) called");
23     if (obj == this) {
24       return true;
25     }
26     if (obj instanceof Circle) {
27       Circle circle = (Circle) obj;
28       return (circle.centre.equals(centre)
29               && circle.radius == this.radius);
30     } else {
31       return false;
32     }
33   }
34
35   public boolean equals(Circle circle) {
36     System.out.println("equals(Circle) called");
37     return (circle.centre.equals(centre)
38             && circle.radius == this.radius);
39   }
40 }
```

Given the following program fragment,

```
1 Circle c1 = new Circle(new Point(0, 0), 10);
2 Circle c2 = new Circle(new Point(0, 0), 10);
3 Object o1 = c1;
4 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);`

Suggested Guide:

```
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 subclass `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 a 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.