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

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

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

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

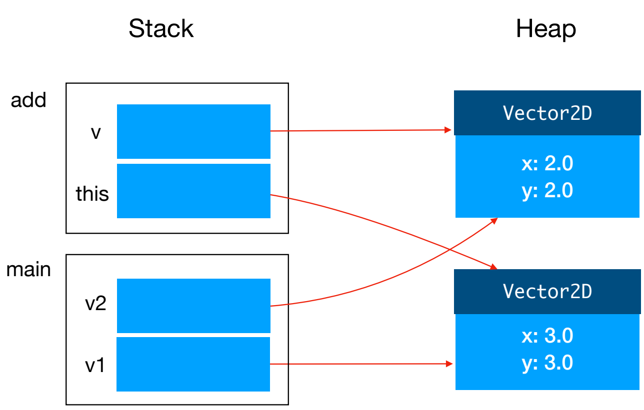
Suppose that the following program fragment is in a `main` method,

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

(a) Show the content of the stack and the heap when the execution reaches the line labeled `A` above. 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.

Comments:

The stack and heap look something like this:



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

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

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

Comments:

We can change it to either:

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

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

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

or

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

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

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

The difference is that the former allocates the new array while the latter just updates the 2D array in place.

Would the program fragment above still be valid?

Comments:

Yes, the program fragment, which is the client of `Vector2D`, is still valid. This is possible because the implementation details (how x and y coordinates are stored and operated on) of `Vector2D` is hidden behind the abstraction barrier and thus one can switch to a different implementation without affecting the existing code written by the clients.

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;
  }
}
```

```java
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,

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

(a) What is the return value of `c1.equals(c2)`? Explain.

> **Comments:**
>
> It returns `false`. Even though both `c1` and `c2` are circles with the same radius and the same center, the `Point` class does not override the `equals` method. As such, when comparing the two centers, `Object::equals` is invoked and the comparison returns `false`.
>
> Without an implementation of `Point::equals`, we need to initialize the circles as follows for `c1.equals(c2)` to return `true`.
>
> ```java
> Point p = new Point(0, 0);
> Circle c1 = new Circle(p, 10);
> Circle c2 = new Circle(p, 10);
> ```

(b) For each of the statement below, trace through the two-step dynamic binding process to show which `equals` method is invoked during run-time.

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

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

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

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

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

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

Comments:

For (i) to (iii), the invocation target is `o1`, with a compile-time type of `Object`. The only method named `equals` the compiler can find in the class `Object` is `bool equals(Object)`. Thus, this method descriptor will be stored in the generated bytecode. During run time, Java determines that the run-time type of `o1` is `Circle`. It thus looks for an accessible method in the class `Circle` with matching method descriptor `bool equals(Object)`.

In this question, there is an implementation of `bool Circle::equals(Object)` that overrides `Object::equals`. Thus, `bool Circle::equals(Object)` is invoked for (i) to (iii).

For (iv) to (vi), the invocation target `c1` has a compile-time type of `Circle`. Now the compiler finds two (overloaded) methods named `equals` in the class `Circle`. In this case, it determines the more specific, invocable, methods between the two.

- For (iv), the parameter has a compile-time type of `Object`. Since we can't pass a `Object` instance into a method expecting a `Circle`, the only correctly invocable method is `bool Circle::equals(Object)`. Similar to (i) to (iii), the method descriptor `bool equals(Object)` is stored in the generated binaries. The run-time decision is the same as (i) to (iii) since the run-time type of the target `c1` is also a `Circle`.
- For (v) and (vi), the parameter has a compile-time type of `Circle`. Now, both `bool Circle::equals(Object)` and `bool Circle::equals(Circle)` are invocable. Between the two, `bool Circle::equals(Circle)` is the more specific one and thus the descriptor `bool equals(Circle)` is stored in the generated binaries. During run-time, `bool Circle::equals(Circle)` will be invoked.