Written Test 2

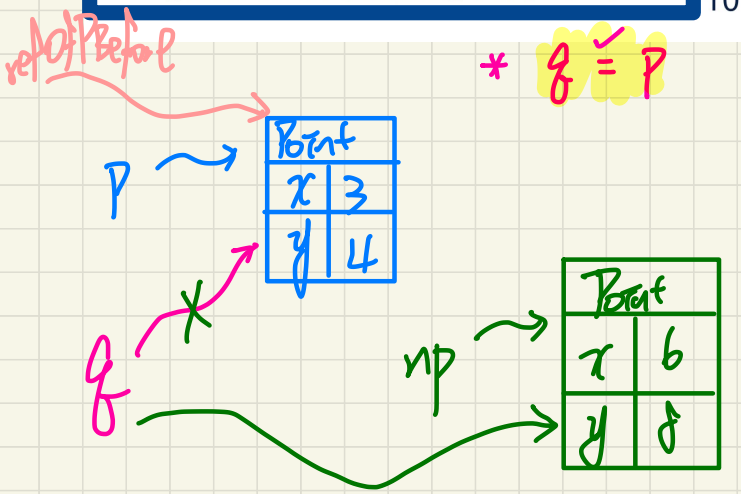
Review Q&A

Call by Value, Caller vs. Callee
assertSame vs. ==
Modelling Diagram of Aggregation
Catch-or-Specify Requirement
Short-Circuit Evaluation

Call by Value: Re-Assigning Reference Parameter

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByRef_1() {
3    Util u = new Util();
4    Point p = new Point(3, 4);
5    Point refOfPBefore = p;
6    u.reassignRef(p);
7    assertTrue(p == refOfPBefore);
8    assertTrue(p.getX() == 3);
9    assertTrue(p.getY() == 4);
10 }
```




```
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x += x; }
}
```

Caller vs. Callee

```
class Course{
    String name;
}
```

```
class Student{
    Course[] cs;
    int noc;
    void register(Course c){...}}
```

callee : Student.register
caller : SMS.registerAll

```
class SMS{
    Student[] ss;
    void registerAll(Course c){
        for(int i=0; i< nos ; i++){ ss[i].register(c)}
    }
}
```

Student

Testing Equality of Points in JUnit: Default vs. Overridden

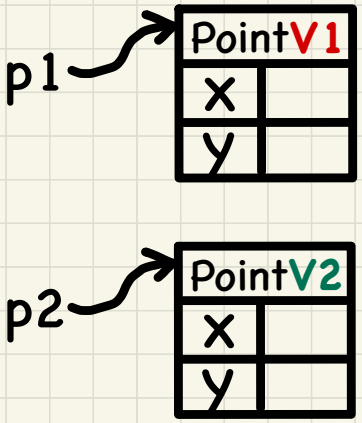
```java
@Test
public void testEqualityOfPointV1andPointv2() {
  PointV1 p1 = new PointV1(3, 4); PointV2 p2 = new PointV2(3, 4);
  /* These two assertions do not compile because p1 and p2 are of different types. */
  /* assertFalse(p1 == p2); assertFalse(p2 == p1); */
  /* assertSame can take objects of different types and fail. */
  /* assertSame(p1, p2); */ /* compiles, but fails */
  /* assertSame(p2, p1); */ /* compiles, but fails */
  /* version of equals from Object is called */
  assertFalse(p1.equals(p2));
  /* version of equals from PointP2 is called */
  assertFalse(p2.equals(p1));
}
```

— == —

LHS and RHS compatible types.

assertSame(p1, p2) → "p1 == p2"

```java
public class Object {
  ...
  public boolean equals(Object obj) {
    return this == obj;
  }
}
```



extends

extends

```java
public class PointV1 {
  private double x;
  private double y;
  public PointV1 (double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

```java
public class PointV2 {
  private int x; private int y;
  public PointV2 (int x, int y) { ... }
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false }
    PointV2 other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

(1)

(obj1 == obj2)

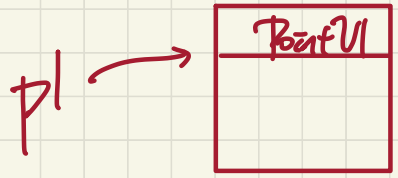
==

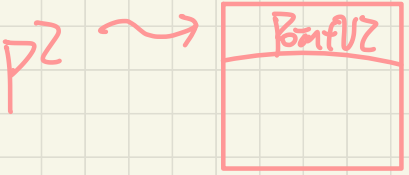
(obj2 == obj1)

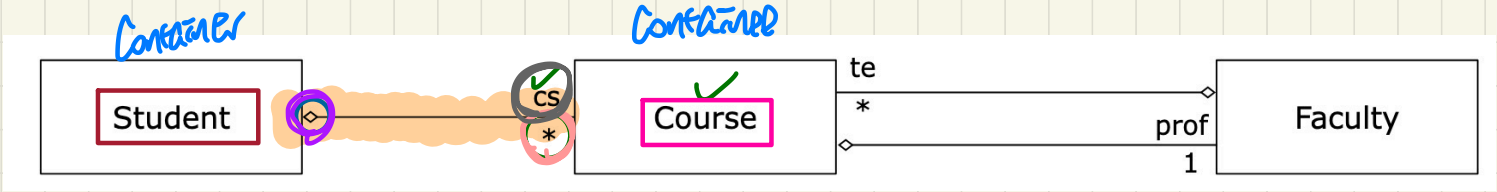
true.

(2) p1.equals(p2) → version depends on DT of p1

p2.equals(p1) → version depends on DT of p2







Student, by aggregation, has zero or more Course objects as their cs

3 aggregation relations.

```java
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  PersonCollector other = (PersonCollector) obj;
  boolean equal = false;
  if(this.nop == other.nop) {
    equal = true;
    for(int i = 0; equal && i < this.nop; i ++) {
      equal = this.persons[i].equals(other.persons[i]);
    }
  }
  return equal;
}
```

Person[]

Person[]

Person

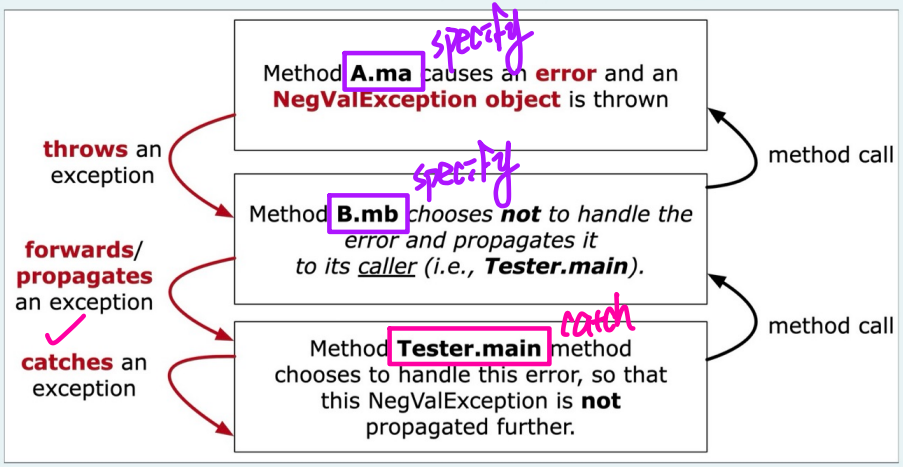
Person

residing/context class (PersonCollector)

this.persons[i].firstName.equals(other.persons[i].firstName)

String version

Consider the following call stack where method ma from class A **throws** a NegValException:



In the above call stack, upon satisfying the catch-or-specify requirement, how many methods opt for the **specify** option? Your answer must be an **integer** value.

Answer: 2.

Correct: 0 <= i && i < ns.length && ns[i] % 2 == 1

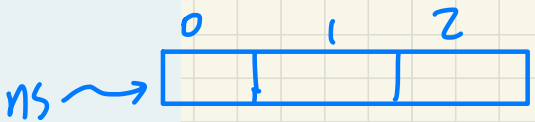
Assume a non-empty integer array **ns** of length 3 and an integer variable **i**.

Consider the following fragment of code:

```
if(0 <= i && ns[i] % 2 == 1 && i < ns.length) {
   System.out.println("Outcome 1");
}
else {
   System.out.println("Outcome 2");
}
```

When executing the above program, which of the following value or values of variable **i** will result in an **ArrayIndexOutOfBoundsException**?

- [ ] a. -2
- [ ] b. -1
- [ ] c. 0
- [ ] d. 1
- [ ] e. 2
- [x] f. 3
- [x] g. 4
- [ ] h. None of the listed answers is correct.



Correct order: gc1 && gc2 && ns[i] ..

should be guarded.

0 <= i

i < ns.length.

0 <= i && ns[i] % 2 == 1 && i < ns.length

any value of i that's negative will cause this exp. to eval to false, and the short-circuit evaluation will skip the rest

guard cond. meant for "too-large" i value but this gc. is misplaced