WrittenTest1 Practice: Q1

For each of the following descriptions, choose the **best** matching kind of methods (constructors, accessors, mutators).

Implementation is meant to initialize values of some attributes.	constructors
Calls to this kind of method should be used as value expressions (e.g., RHS of variable assignments, values to be printed to the console).	accessors
Name must match that of the class.	constructors
Calls to this kind of method must stand alone and cannot be used as value expressions (e.g., RHS of variable assignments, values be printed to the console).	mutators
Return type is always void.	mutators
Name can be arbitrary and implementation must contain at least one return statement.	accessors
Calls to this kind of method must be associated with the **new** keyword.	constructors
Name can be arbitrary and implementation cannot contain any return statement.	mutators
Return type can be either primitive or reference.	accessors

accessor → getter

p.getX();

mutators → setter

void

p.setX(3); ✓

int i = p.setX(3) ✗

WrittenTest1 Practice: Q2

Assume a `Person` class declared with: a string attribute `name` and a constructor initializing that string attribute using the input parameter.

Now consider the following fragment code which implements the `main` method of some console application class:

```
Person p1 = new Person("Alan");
Person p2 = new Person("Mark");
Person p3 = new Person("Alan");
Person p4 = p2;
p2 = p1;
p1 = p4;
p4 = p3;
p3 = p1;
System.out.println("Done!");
```

Now say we place a breakpoint at the last line of the above fragment of code and debug it as Java Application. For the following list of statements, choose **all** which are **false**.

- [x] a. Addresses stored in p2 and p4 are the same.
- [ ] b. The `name` attribute value of p1 is the same as that of p3.
- [x] c. Addresses stored in p2 and p3 are the same.
- [x] d. The `name` attribute value of p1 is the same as that of p4.
- [x] e. Addresses stored in p1 and p2 are the same.
- [x] f. The `name` attribute value of p1 is the same as that of p2.
- [x] g. The `name` attribute value of p2 is the same as that of p3.
- [ ] h. The `name` attribute value of p2 is the same as that of p4.
- [ ] i. Addresses stored in p1 and p3 are the same.
- [x] j. The `name` attribute value of p3 is the same as that of p4.
- [x] k. Addresses stored in p3 and p4 are the same.
- [x] l. Addresses stored in p1 and p4 are the same.

WrittenTest1 Practice: Q3

Consider the following class:

```
public class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void moveUp(double units) {
    this.y = this.y + units;
  }
  public double getDistanceFromOrigin() {
    return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
  }
}
```

call by value

double units = 24.8 ✓

double

Now say we have the following variable declared and initialized:

```
Point p = new Point(3.4, 5.7);
```

From the following independent lines of code, chose those which compile (i.e., without any syntax or type error).

- ✗ a. int dist = p.getDistanceFromOrigin();
- ✗ b. double dist = p.moveUp(24.8);
- ✓ c. System.out.println(p.getDistanceFromOrigin());
- ✓ d. p.getDistanceFromOrigin();
- ✓ e. double dist = p.getDistanceFromOrigin();
- ✓ f. p.moveUp(24.8);
- ✗ g. System.out.println(p.moveUp(24.8));

int = double ✗

returns void

int i = 23;
double d = 46.23;

i = d; ✗

double = int

i = (int) d; ✓

i = 46

d = i; ✓

i = 23.0

WrittenTest1 Practice: Q4

Consider the following Java class defining a template for points on a 2-dimensional plane, each of which characterized by its **position**: x and y co-ordinates.

```
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Point(char axis, int dist) {
    if(axis == 'X') {
      this.x = dist;
      this.y = 0;
    }
    else if(axis == 'Y') {
      this.y = dist;
      this.x = 0;
    }
  }
  public void move(char direction, int dist) {
    if(direction == 'U') {
      this.y = this.y + dist;
    }
    else if(direction == 'D') {
      this.y = this.y - dist;
    }
    else if(direction == 'L') {
      this.x = this.x - dist;
    }
    else if(direction == 'R') {
      this.x = this.x + dist;
    }
  }
}
```

Consider the following fragment of code testing the above class:

```
Point p1 = new Point('X', 5);
Point p2 = new Point('Y', 5);
Point p3 = new Point(3, 0);
Point p4 = new Point(0, 1);
p4.move('D', 5);
p1.move('L', 1);
p3.move('D', 2);
p2.move('L', 2);
p1.move('U', 2);
p3.move('R', 1);
p4.move('L', 6);
p2.move('D', 2);
```

After executing the above lines of code creating and manipulating point objects, what are the **positions** of the four points (p1, p2, p3, p4)?

p3 (4, -2)

p4 (-6, -4)

p2 (-2, 3)

p1 (4, 2)

WrittenTest1 Practice: Q5

Assume that a Person class is already defined, and it has an attribute name, a constructor that initializes the person's name from the input string, and an accessor `getName` returning the person's name. Consider the following fragment of Java code (inside some main method):

```
Person p0 = new Person("Suyeon");
Person p1 = new Person("Yuna");
Person p2 = new Person("Sunhye");
Person p3 = new Person("Jihye");
p0 = p2;
p1 = p3;
Person[] persons1 = {p0, p1, p2, p3};
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons2.length; i ++) {
    persons2[i] = persons1[persons2.length - i - 1];
}
```

i	persons2.length - i - 1
0	4 - 0 - 1 = 3
1	4 - 1 - 1 = 2
2	1
3	0

var changed = var used;

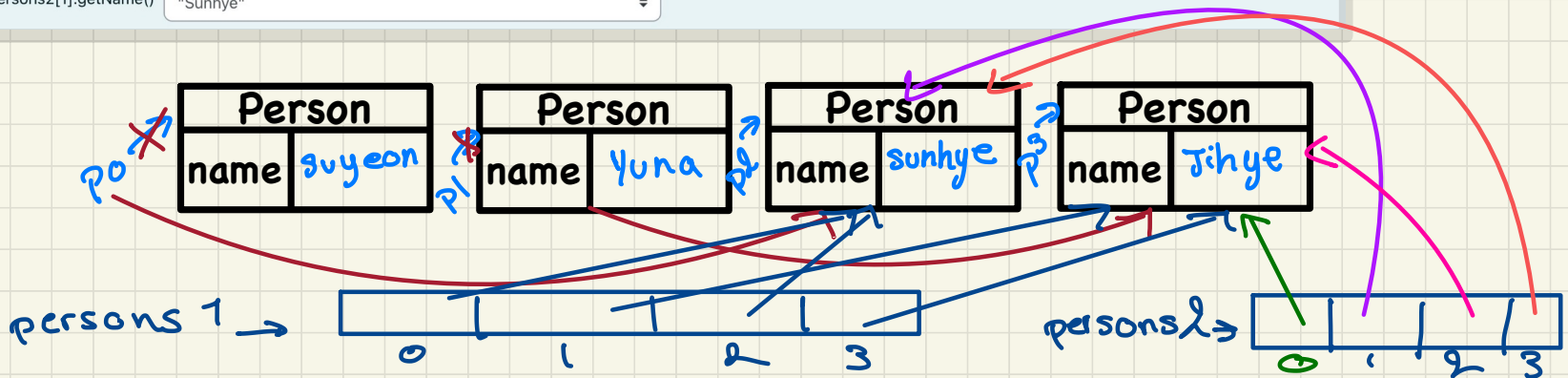
Executing the above fragment of code, after exiting from the loop, indicate the value of each of the following expressions.

persons2[0].getName() "Jihye"

persons2[3].getName() "Sunhye"

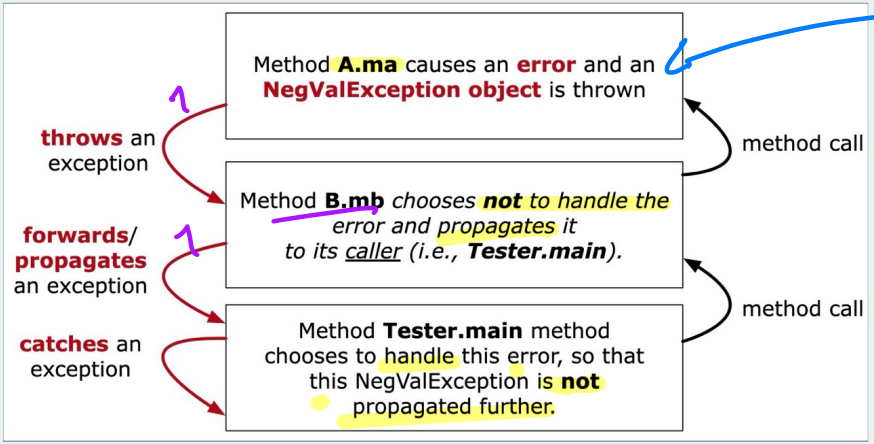
persons2[2].getName() "Jihye"

persons2[1].getName() "Sunhye"



WrittenTest1 Practice: Q6

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

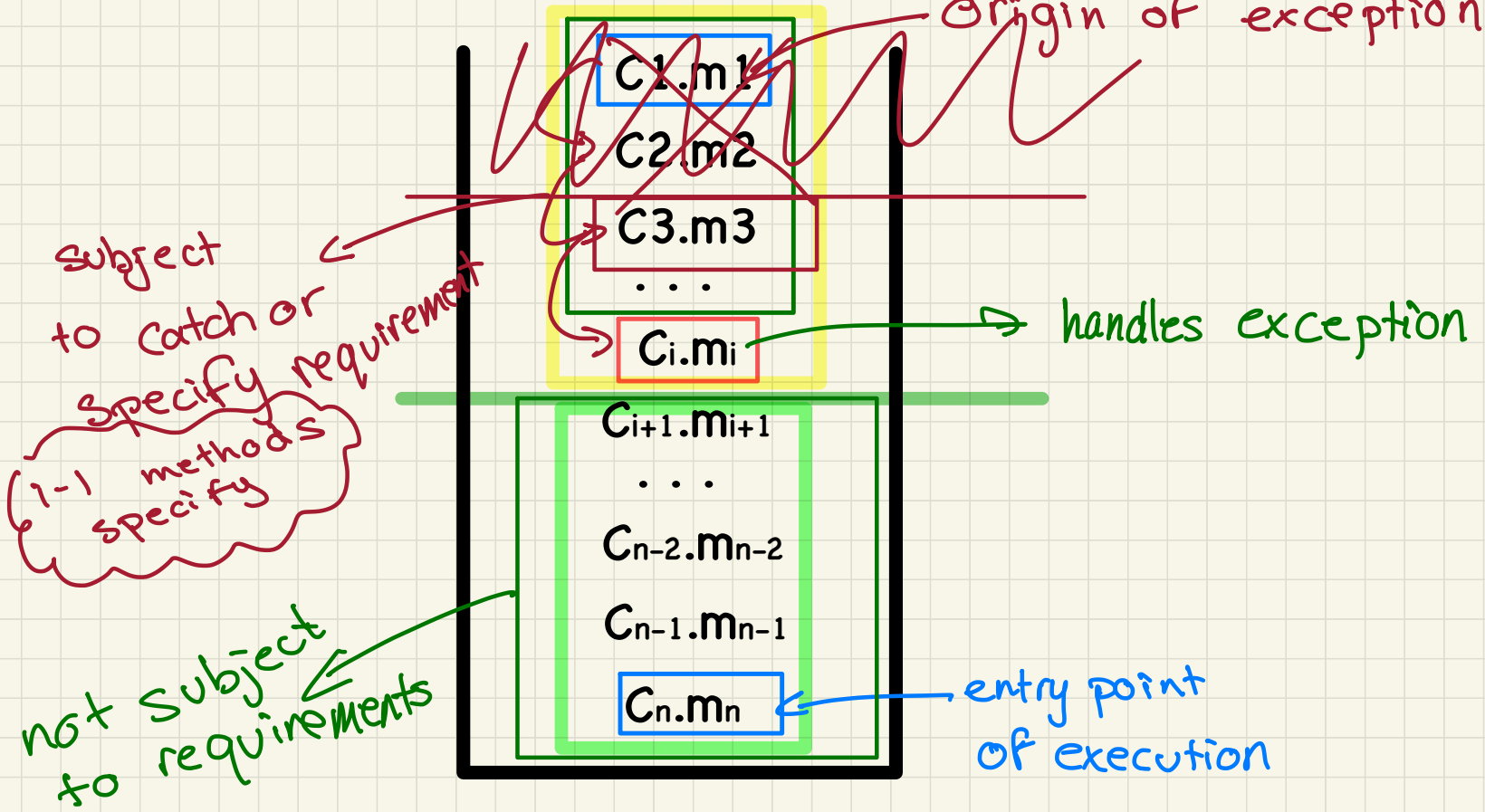


In the above call stack, upon satisfying the catch-or-specify requirement, how many methods opt for the **specify** option? Your answer must be an **integer** value.

Answer: 2

Catch-or-Specify Requirement: Call Stack

point .X

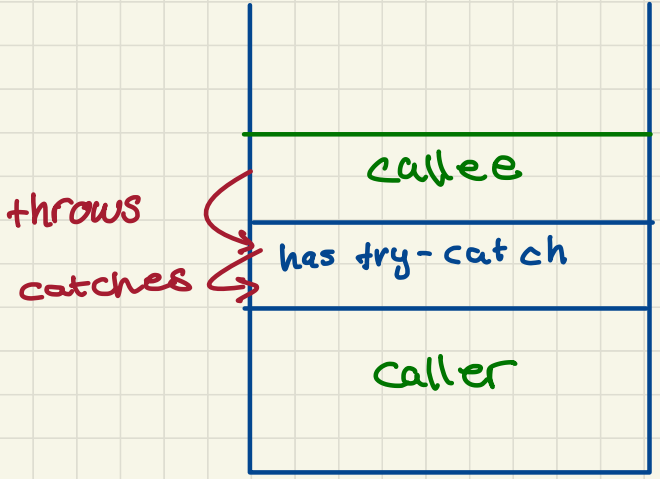


WrittenTest1 Practice: Q7

At a runtime call stack, if a method implements a try-catch block to handle a *NegValException* that may be thrown from its **callee**, then this method's **caller** is still obliged to either catch or specify that *NegValException*.

Select one:

- ○ True
- ◉ False



WrittenTest1 Practice: Q8

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown ");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeException thrown.");
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

call	counter value
c.inc	0 → 1 (0<3)
c.inc	1 → 2 (1<3)
c.inc	2 → 3 (2<3)
c.inc	3 X (3≤3)

Say the method `increment` is implemented correctly as explained above.

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any):	L3 of CounterTester2
2nd line to execute (if any):	L4 of CounterTester2
3rd line to execute (if any):	L6 of CounterTester2
4th line to execute (if any):	L7 of CounterTester2
5th line to execute (if any):	L9 of CounterTester2
6th line to execute (if any):	L13 of CounterTester2
7th line to execute (if any):	Execution Terminated

WrittenTest1 Practice: Q9

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown.");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeException thrown.");
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

call	counter value
c.inc	0 → 1 (1>3)
c.inc	1 → 2 (2>3)
c.inc	2 → 3 (3>3)
c.inc	3 → 4

Say the *increment* method is implemented **incorrectly** as follows:

```
public void increment() throws ValueTooLargeException {
if(value > Counter.MAX_VALUE) {
 throw new ValueTooLargeException("value is " + value);
}
else { value ++; }
}
```

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any):	L3 of CounterTester2
2nd line to execute (if any):	L4 of CounterTester2
3rd line to execute (if any):	L6 of CounterTester2
4th line to execute (if any):	L7 of CounterTester2
5th line to execute (if any):	L9 of CounterTester2
6th line to execute (if any):	L10 of CounterTester2
7th line to execute (if any):	Execution Terminated

WrittenTest1 Practice: Q10

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown.");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeException thrown.");
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

Say the *increment* method is implemented **incorrectly** as follows:

```
public void increment() throws ValueTooLargeException {
if(value < Counter.MAX_VALUE) {
  throw new ValueTooLargeException("value is " + value);
}
else { value ++; }
}
```

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any): L3 of CounterTester2

2nd line to execute (if any): L4 of CounterTester2

3rd line to execute (if any): L6 of CounterTester2

4th line to execute (if any): L17 of CounterTester2

5th line to execute (if any): Execution Terminated

6th line to execute (if any): Execution Terminated

7th line to execute (if any): Execution Terminated

WrittenTest1 Practice: Q11

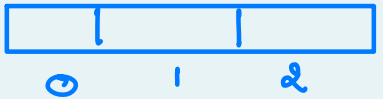
Assume a non-empty integer array **ns** of length 3 and an integer variable **i**.

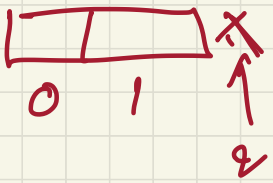
Consider the following fragment of code:

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



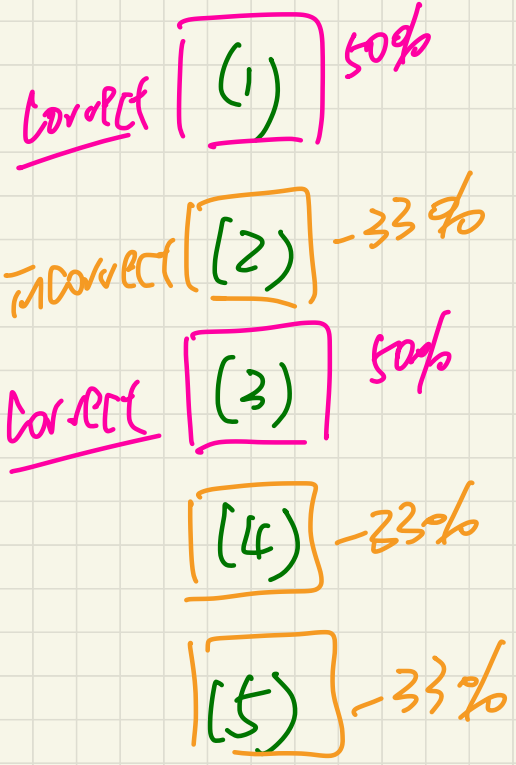


✓

p || q && r

3 + (4 * 5)

3 + 32 = 35


Correct
(1)
50%
Incorrect
(2)
-33%
Correct
(3)
50%
(4)
-33%
(5)
-33%