EECS2030 Advanced Object-Oriented Programming (Fall 2021)

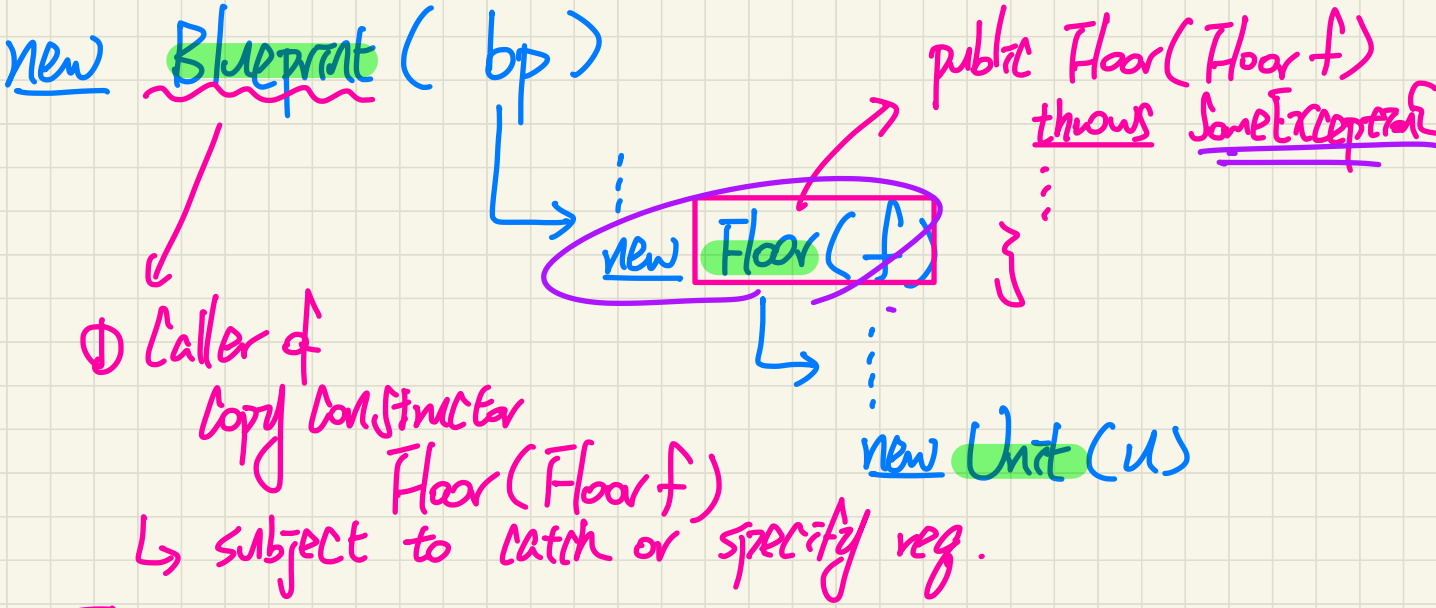
Q&A - Lecture 4

Thursday, October 28

Announcement

- Lecture W7 (released: Oct. 25) — inheritance
- Lab3 (released: Oct. 20; due: Nov 1) — 2pm EST.
- Written Test 2 (due: Oct. 28/29)
- Programming Test 2

What if a copy constructor throws an exception



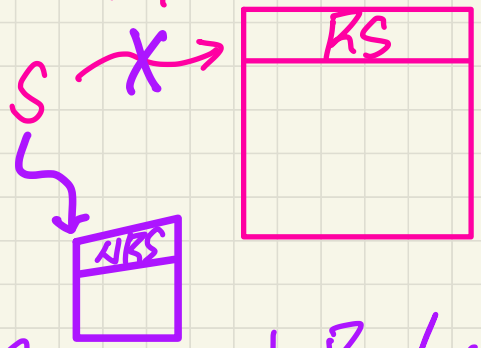
WT: intuition of polymorphism

Student s = new ResidentStudent("Jim");

declared type (static type)
↳ never changes
↳ determines the range of attributes/methods callable on "s".

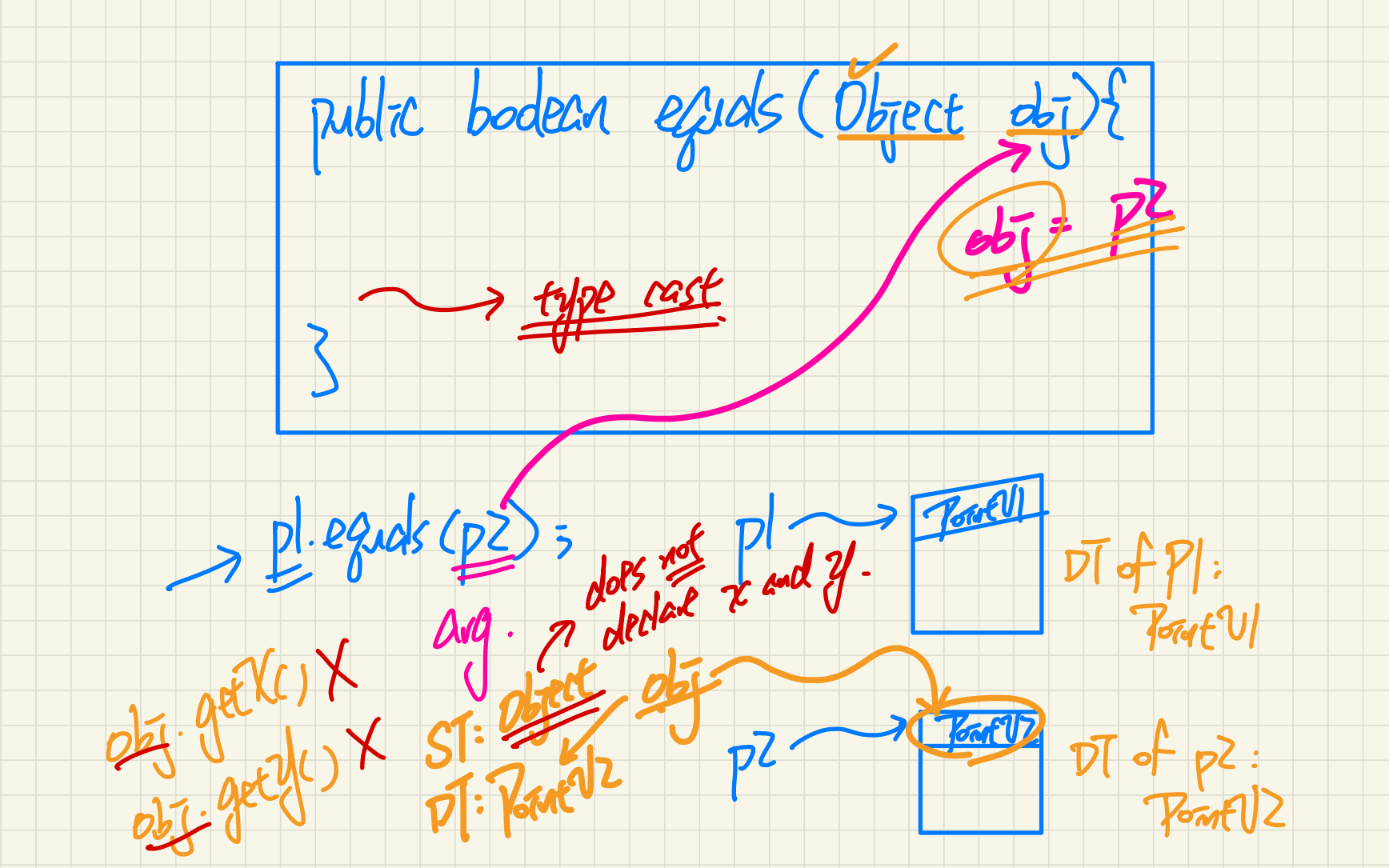
expectation

D.T: RS



s = new NonResidentStudent("Alex");

D.T: NRS


public boolean equals(Object obj){
obj = p2
type cast
}
p1.equals(p2);
arg.
does not declare x and y.
p1
PointV1
DT of p1: PointV1
obj.getX() X
obj.getY() X
ST: Object
DT: PointV2
obj
p2
PointV2
DT of p2: PointV2

Written Test 2: Practice Questions

Consider the following classes:

```
public class Student {
  private String id;
  private Course[] cs;
}
```

```
public class Course {
  private String title;
  private Faculty prof;
}
```

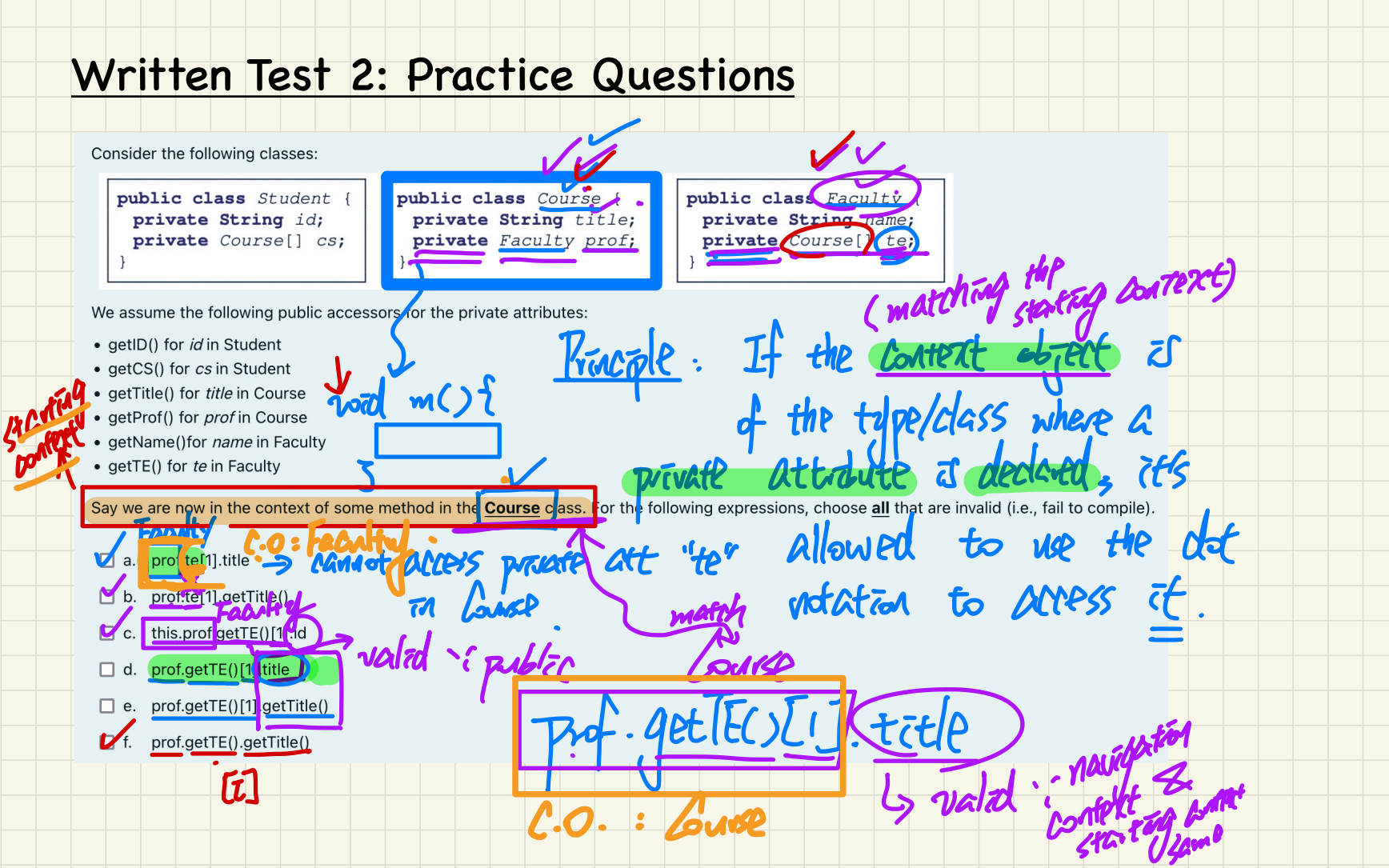
```
public class Faculty {
  private String name;
  private Course[] te;
}
```

We assume the following public accessors for the private attributes:

- getID() for *id* in Student
- getCS() for *cs* in Student
- getTitle() for *title* in Course
- getProf() for *prof* in Course
- getName()for *name* in Faculty
- getTE() for *te* in Faculty

Say we are now in the context of some method in the **Course** class. For the following expressions, choose **all** that are invalid (i.e., fail to compile).

- [x] a. prof.te[1].title
- [ ] b. prof.te[1].getTitle()
- [x] c. this.prof.getTE()[1].id
- [ ] d. prof.getTE()[1].title
- [x] e. prof.getTE()[1].getTitle()
- [x] f. prof.getTE().getTitle()



Written Test 2: Practice Questions

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 : 3

2nd line : 4

3rd line : 6

4th line : 7

5th line : 9

6th line : 10

7th line : Execution Te.

8th line Execution Te.

Written Test 2: Practice Questions

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 */
```

3
4
6
17
Exec. Terminated.

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.

Written Test 2: Practice Questions

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 */
```

3
4
6
7
9
13

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.

Written Test 2: Practice Questions

Consider the following two classes for representing 2D points (where the equals method is overridden in PointV2):

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

```
public class PointV2 {
  private int x; private 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;
  }
}
```

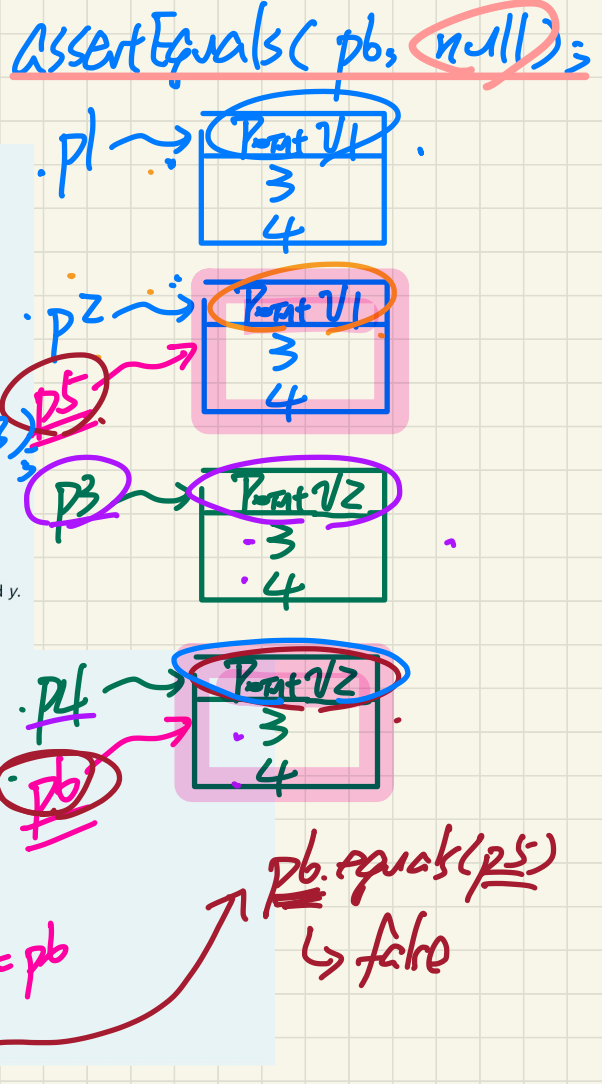
For the above PointV2 class, assume that there is a constructor, like in PointV1, which initializes the values of attributes *x* and *y*.

Let's now assume the following object creations:

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
PointV2 p4 = new PointV2(3, 4);
PointV1 p5 = p2;
PointV2 p6 = p4;
```

For the following assertions, consider each in isolation and choose **all** those that will **fail**.

- [ ] a. assertNotSame(p4, p6);
- [ ] b. assertSame(p4, p6);
- [ ] c. assertEquals(p2, p5);
- [ ] d. assertSame(p1, p2);
- [ ] e. assertEquals(p5, p6);
- [ ] f. assertNotSame(p1, p2);
- [ ] g. assertEquals(p1, p2);
- [ ] h. assertEquals(p3, p4);
- [ ] i. assertNotEquals(p3, p4);
- [ ] j. assertEquals(p6, p5);



Written Test 2: Practice Questions

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