

EECS2030 Advanced Object-Oriented Programming
(Fall 2021)

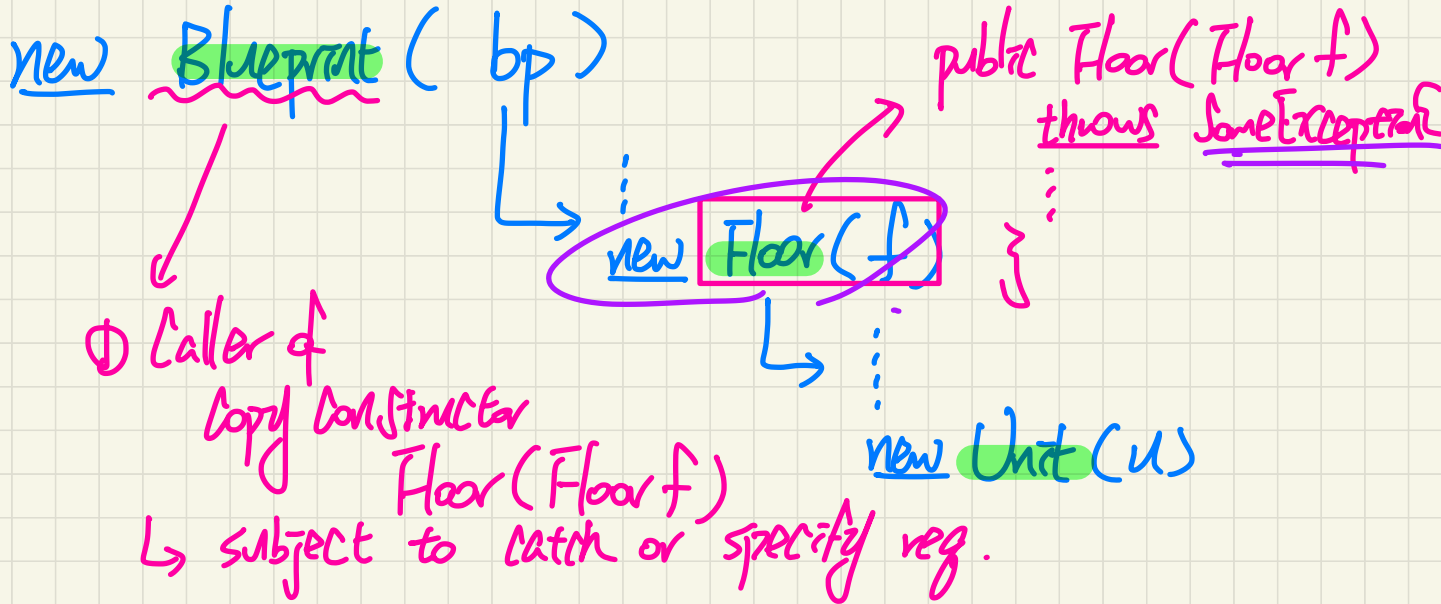
Q&A - Lecture 4

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



② public Blueprint(Blueprint bp) throws SomeException catch (SE e)

③ public Blueprint(Blueprint bp) { ... try { ... new Floor(f) } }

WT: intuition of polymorphism

Student

S = new ResidentStudent("Jim")

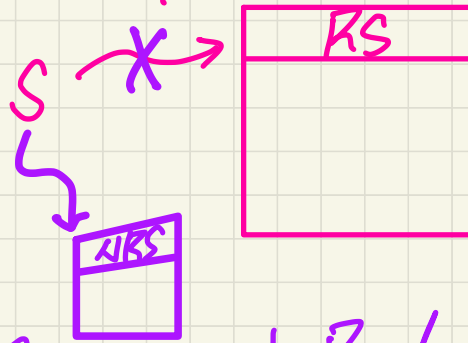
dedared type

(static type)

↳ never changes

↳ determines the range of the callable methods on "S". *expectation*

D.T: KS



S = new NonResidentStudent("Alan")

DT: NRS

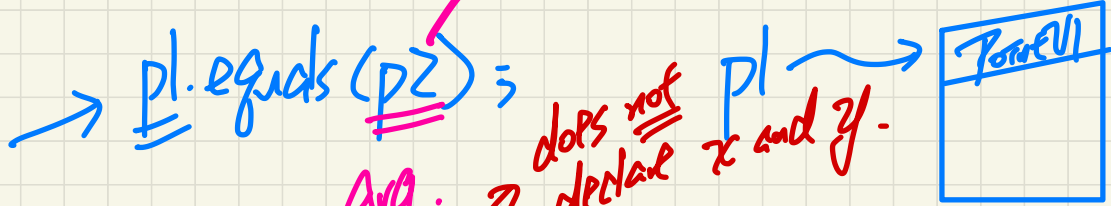
```

public boolean equals(Object obj) {
    // ...
}

```

type cast

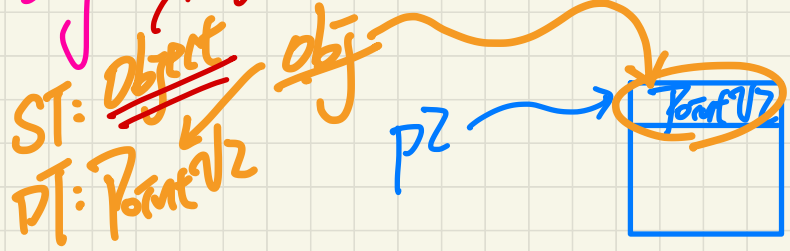
obj = p2



DT of pl: PointV1

obj.getx() X
obj.gety() X

arg. → does not declare x and y.



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

Principle: If the Context object is of the type/class where a private attribute is declared, it's (matching the starting context)

word m() {
[]

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).

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

Starting context

Faculty

C.O.: Faculty
cannot access private att "te" in Course

Faculty

valid - is public

match Course

`prof.getTE()[1].title`

C.O.: Course

title

valid - navigation context starting from same

[1]

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 Ter.

8th line : Execution Ter.

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
b
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.

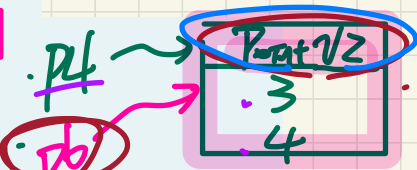
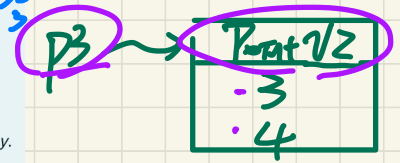
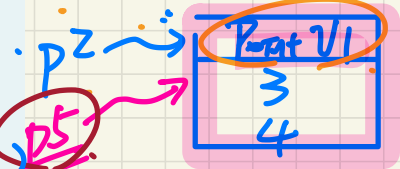
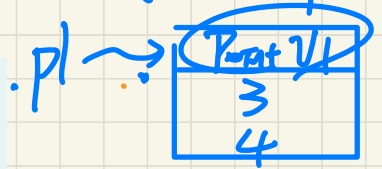
Let's now assume the following object creations:

- 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);`

assertEquals(p6, null);



Annotations:
 - Red arrow pointing to `if(obj == null) { return false; }` with text "null → NPE" and a red 'X'.
 - Red box around `if(this.getClass() != obj.getClass()) { return false; }` with text "p4.equals(p3)".

Handwritten notes:
 - "p4.equals(p3)" with a blue arrow pointing to the boxed code.
 - "p5" circled in red with an arrow pointing to the `PointV1 p5 = p2;` line.
 - "p3" circled in purple with an arrow pointing to the `PointV2 p3 = new PointV2(3, 4);` line.

assertEquals(p4, p3);

Handwritten notes:
 - "p2.equals(p5)" with an arrow pointing to the `assertEquals(p2, p5);` assertion.
 - "p2 == p5" below it.

p6 circled in red with an arrow pointing to the `assertEquals(p6, p5);` assertion.

Handwritten notes:
 - "p6.equals(p5)" with an arrow pointing to the `assertEquals(p6, p5);` assertion.
 - "↳ false" below it.

Handwritten notes:
 - "p1.equals(p2)" with an arrow pointing to the `assertSame(p1, p2);` assertion.
 - "↳ p1 == p2" below it.

Handwritten notes:
 - "p5.equals(p6)" with an arrow pointing to the `assertEquals(p5, p6);` assertion.
 - "↳ PointV1" below it.
 - "p5 == p6" below that.

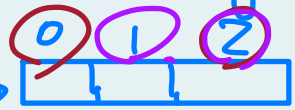
Written Test 2: Practice Questions

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");  
}
```

$ns.length == 3$



`ns`

$0 \leq i < 3$
 $29.4. : 3, 4, 5.$
 \rightarrow should be guarded

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

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

placed after the indexing notation that's supposed to be guarded.

$0 \leq 3 \leq 4$
 $0 \leq \bar{i} \leq 2$
 $ns[\bar{i}] \% 2 == 1$
 $i < ns.length$
 $(-1) (F)$