Lecture 14 - Oct. 31

Aggregation

Call by Value: Primitive vs. Reference Aggregations

Announcements

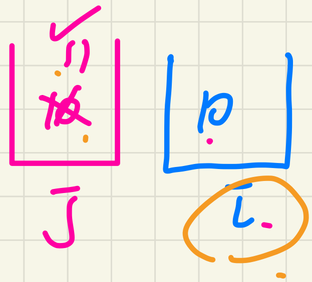
- ProgTest1: Visit office hours to discuss your solution
- Lab3 released (equals & copy constructor)
- WrittenTest2 tomorrow (guide & practice)

Call by Value: Re-Assigning Primitive Parameter

```java
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); } }
```

call by value

```java
1  @Test
2  public void testCallByVal() {
3    Util u = new Util();
4    int i = 10;
5    assertTrue(i == 10);
6    u.reassignInt(i);
7    assertTrue(i == 10);
8  }
```



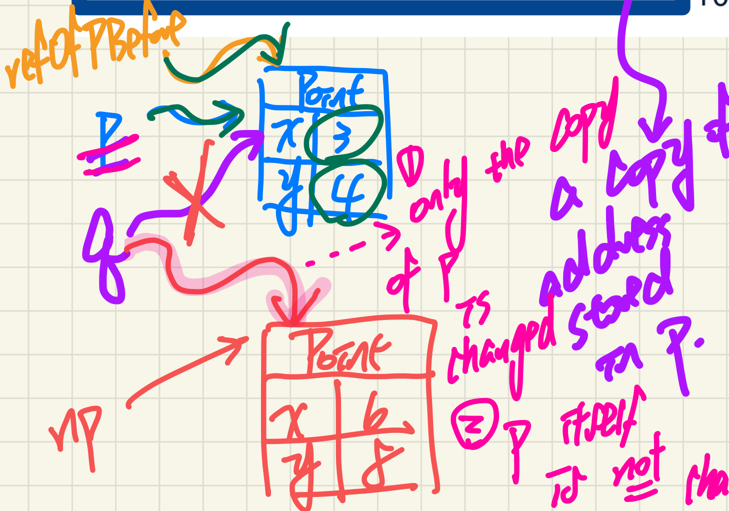
from the caller's point of view, arg. i is not changed

Call by Value: Re-Assigning Reference Parameter

```java
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); } }
```

```java
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 }
```

call by value



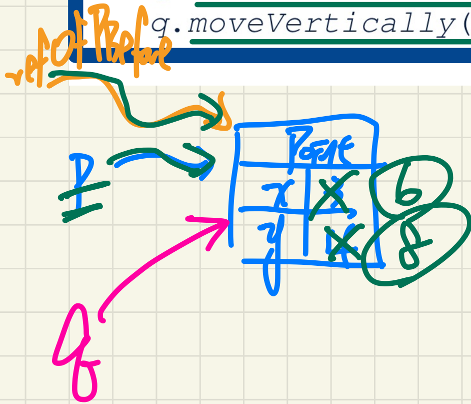

```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x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

Call by Value: Calling Mutator on Reference Parameter

```java
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); } }
```

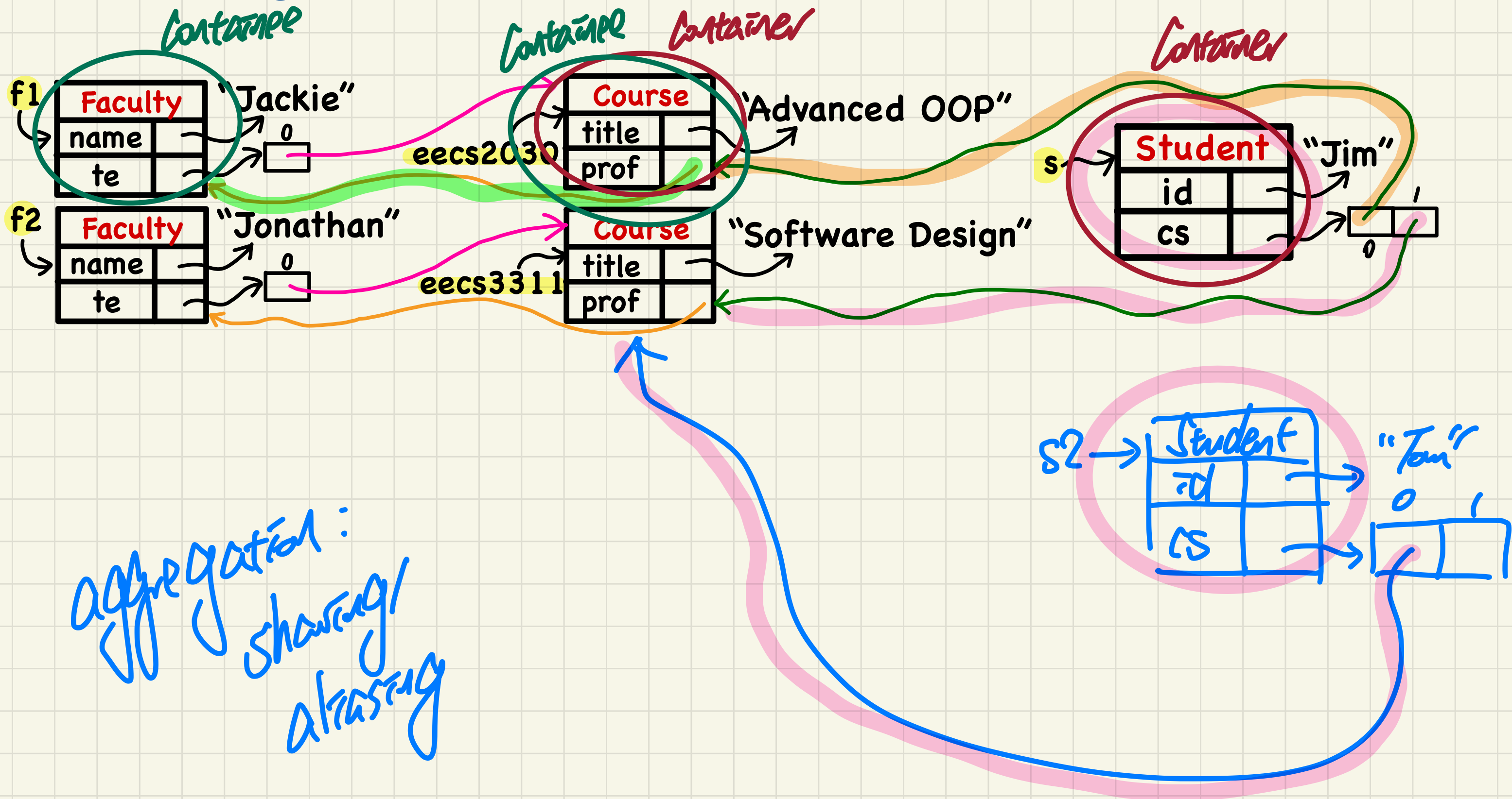
```java
1  @Test
2  public void testCallByRef_2() {
3    Util u = new Util();
4    Point p = new Point(3, 4);
5    Point refOfPBefore = p;
6    u.changeViaRef(p);
7    assertTrue(p == refOfPBefore);
8    assertTrue(p.getX() == 6);
9    assertTrue(p.getY() == 8);
10 }
```

call by value



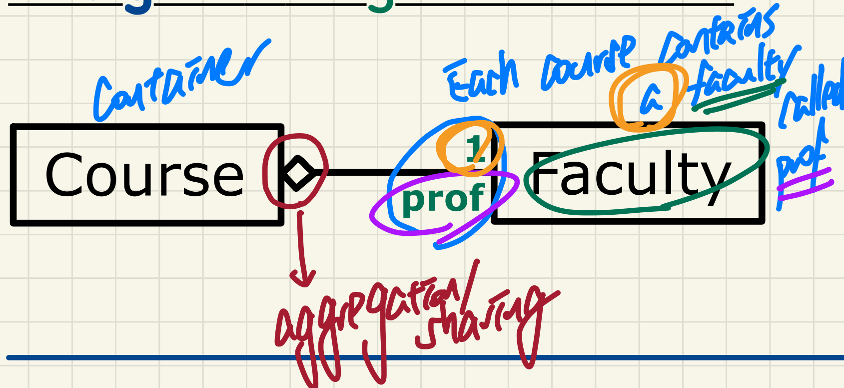
```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x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

Terminology: Container vs. Containee



Aggregation: Design

Design 1: Single Containee

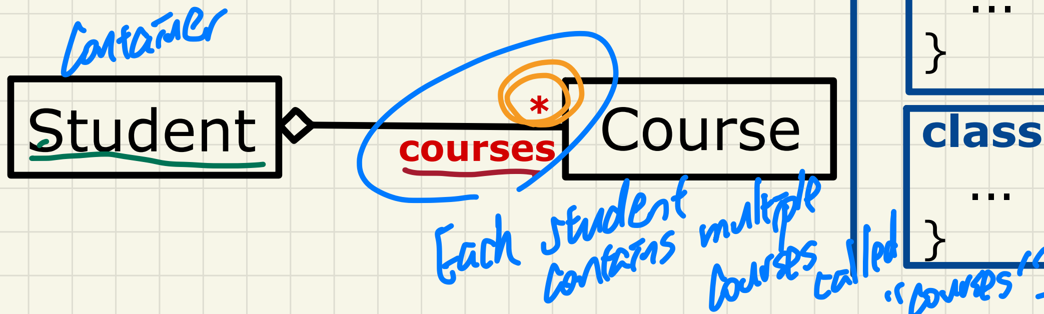


Java Implementation

```java
class Course {
  Faculty prof;
  ...
}
```

```java
class Faculty {
  ...
}
```

Design 2: Multiple Containees




```java
class Student {
  Course[] courses;
  ...
}
```

```java
class Course {
  ...
}
```

Aggregation (1)

Course	
title	
prof	

Faculty	
name	

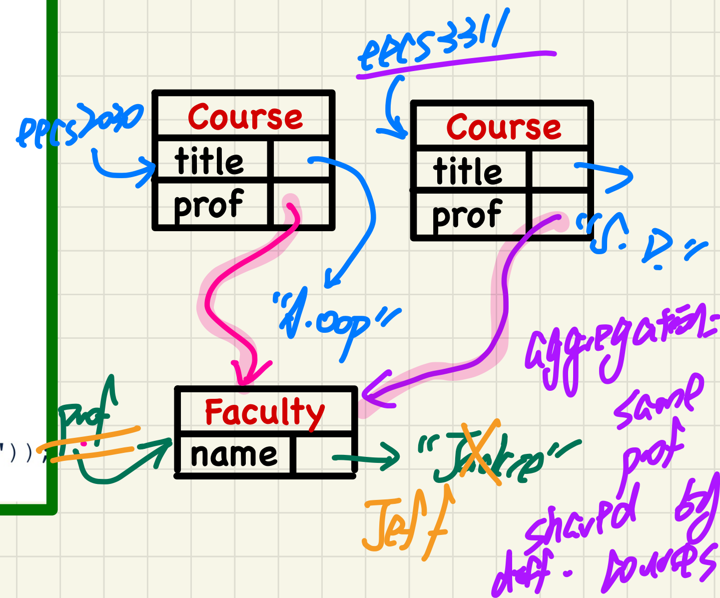
```java
class Course {
  String title;
  Faculty prof;
  Course(String title) {
    this.title = title;
  }
  void setProf(Faculty prof) {
    this.prof = prof;
  }
  Faculty getProf() {
    return this.prof;
  }
}
```

```java
class Faculty {
  String name;
  Faculty(String name) {
    this.name = name;
  }
  void setName(String name) {
    this.name = name;
  }
  String getName() {
    return this.name;
  }
}
```

```java
@Test
public void testAggregation1() {
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  Faculty prof = new Faculty("Jackie");
  eecs2030.setProf(prof);
  eecs3311.setProf(prof);
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  /* aliasing */
  prof.setName("Jeff");
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));

  Faculty prof2 = new Faculty("Jonathan");
  eecs3311.setProf(prof2);
  assertTrue(eecs2030.getProf() != eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));
  assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
```

exercise



Aggregation (2)

Student	
id	
cs	

Faculty	
name	
te	

Course	
title	
prof	

(Exercise)

```
public class Student {
  private String id; Course[] cs; int noc; /* # of courses */
  public Student(String id) { ... }
  public void addCourse(Course c) { ... }
  public Course[] getCS() { ... }
}
```

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

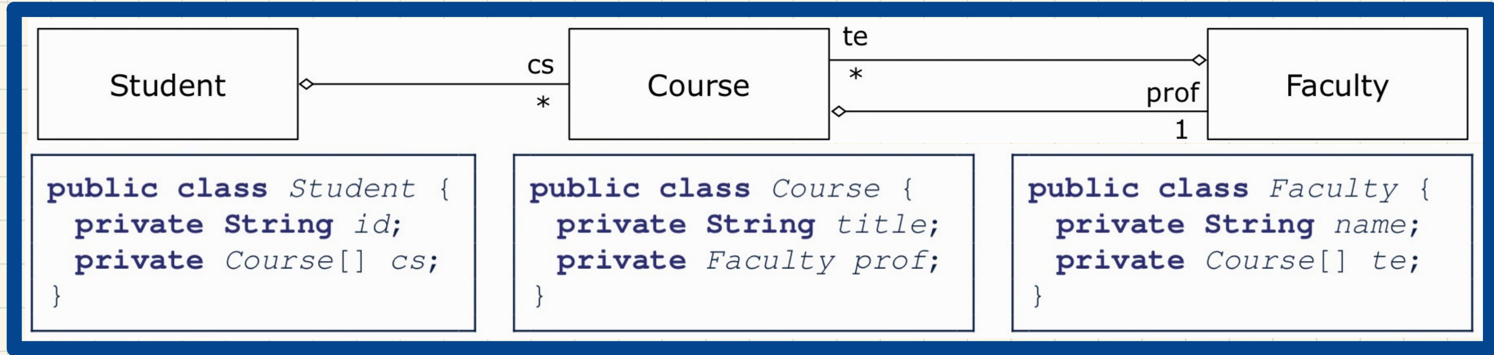
```
public class Faculty {
  private String name; Course[] te; int not; /* # of teaching */
  public Faculty(String name) { ... }
  public void addTeaching(Course c) { ... }
  public Course[] getTE() { ... }
}
```

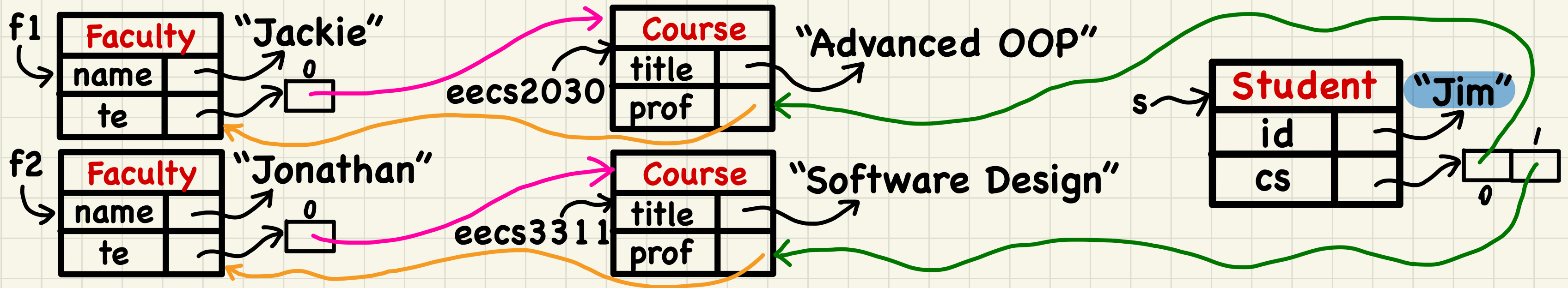
```
@Test
public void testAggregation2() {
  Faculty p = new Faculty("Jackie");
  Student s = new Student("Jim");
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  eecs2030.setProf(p);
  eecs3311.setProf(p);
  p.addTeaching(eecs2030);
  p.addTeaching(eecs3311);
  s.addCourse(eecs2030);
  s.addCourse(eecs3311);

  assertTrue(eecs2030.getProf() == s.getCS()[0].getProf());
  assertTrue(s.getCS()[0].getProf()
             == s.getCS()[1].getProf());
  assertTrue(eecs3311 == s.getCS()[1]);
  assertTrue(s.getCS()[1] == p.getTE()[1]);
}
```

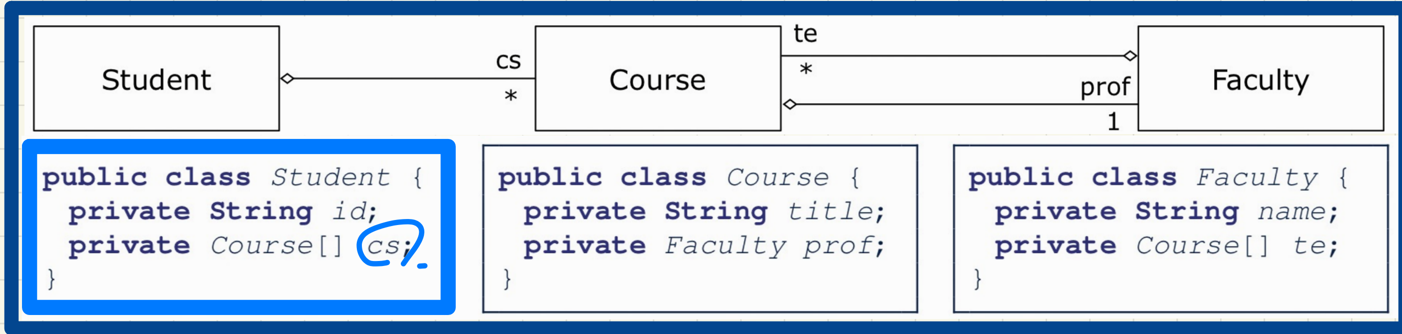
adding a container (e.g. addPoint)

Runtime Object Structure: Student, Course, Faculty





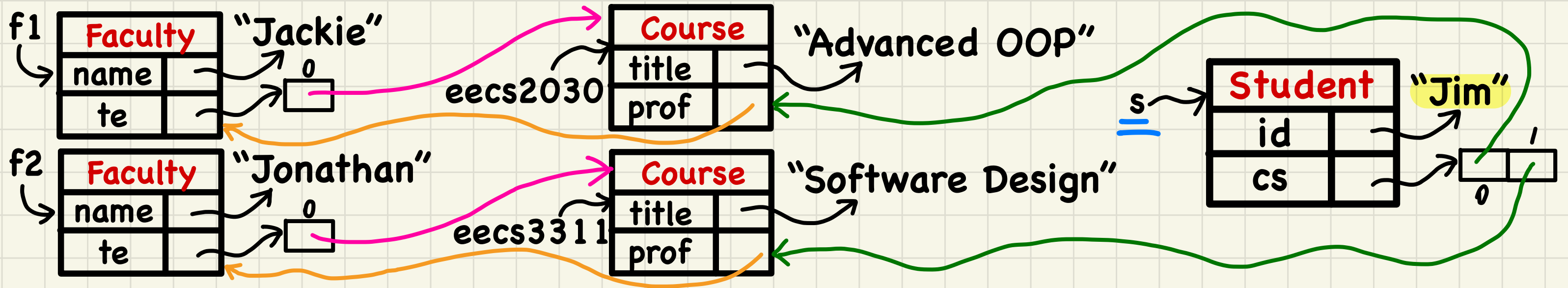
Dot Notation for Navigating Classes (1)



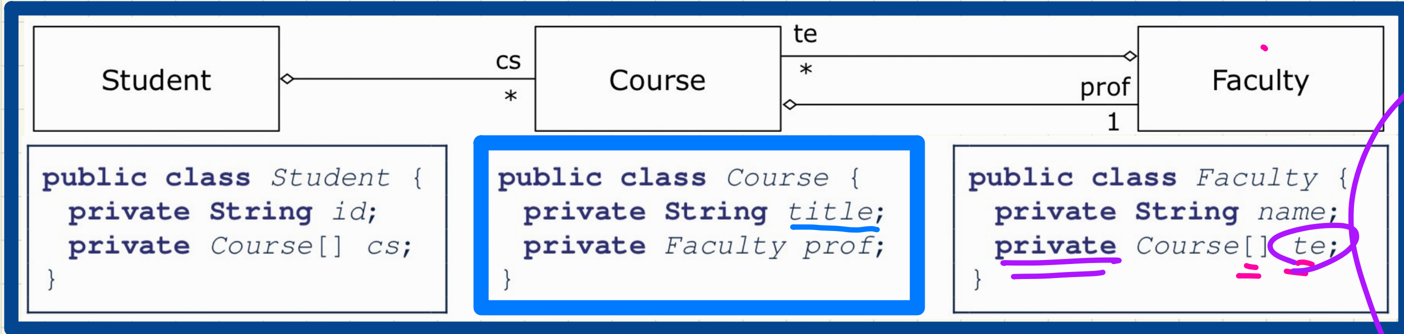

```
/* Get the student's id.
 */
String getID() {
    return this.id;
}
```

```
/* Title of ith course
 */    s.getTitle(1)
String getTitle(int i) {
    return this.cs[i].getTitle()
}
```

```
/* Name of
 * ith course's instructor
 */   s.getName(1)
String getName(int i) {
    return this.cs[i].getProf().getName()
}
```



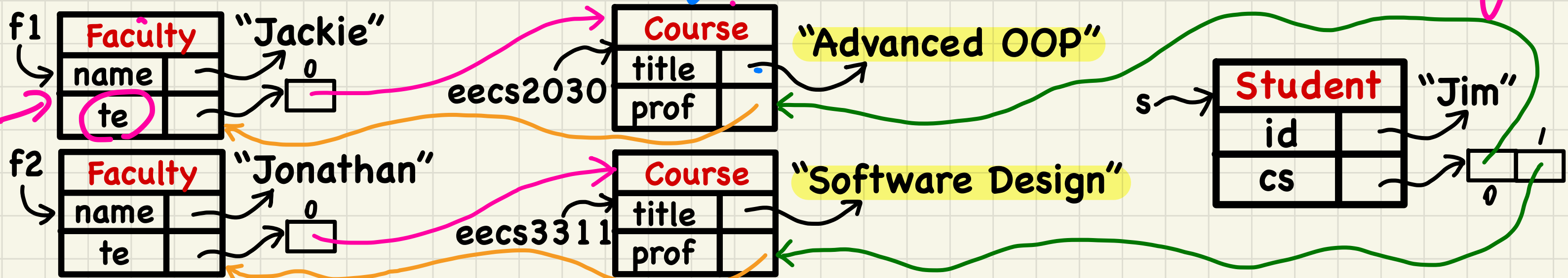
Dot Notation for Navigating Classes (2)



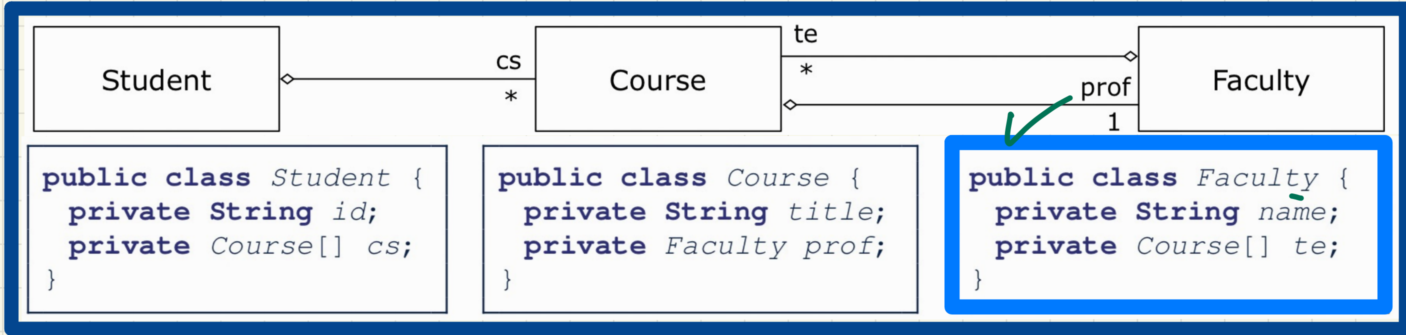

```
/* Get course's title.
 */
String getTitle() {
  return this.title;
}
```

```
/* Name of instructor
 */
String getName() {
  return this.getProf().getName()
}
```

```
/* Title of instructor's
 * ith teaching course
 */ eecs3311.getTitle(0);
String getTitle(int i) {
  return this.getProf().getTe()[i].getTitle();
}
```

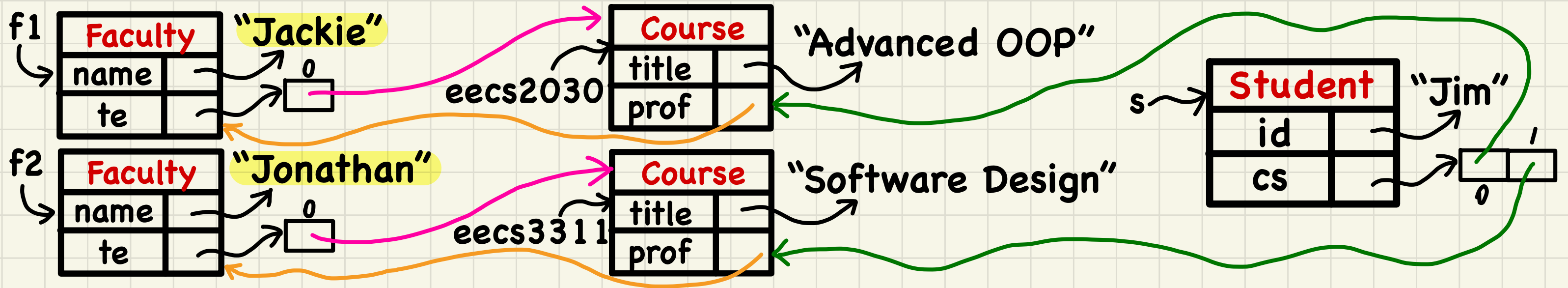


Dot Notation for Navigating Classes (3)




```
/* Name of instructor
 */
String getName() {
    return this.getName();
}
```

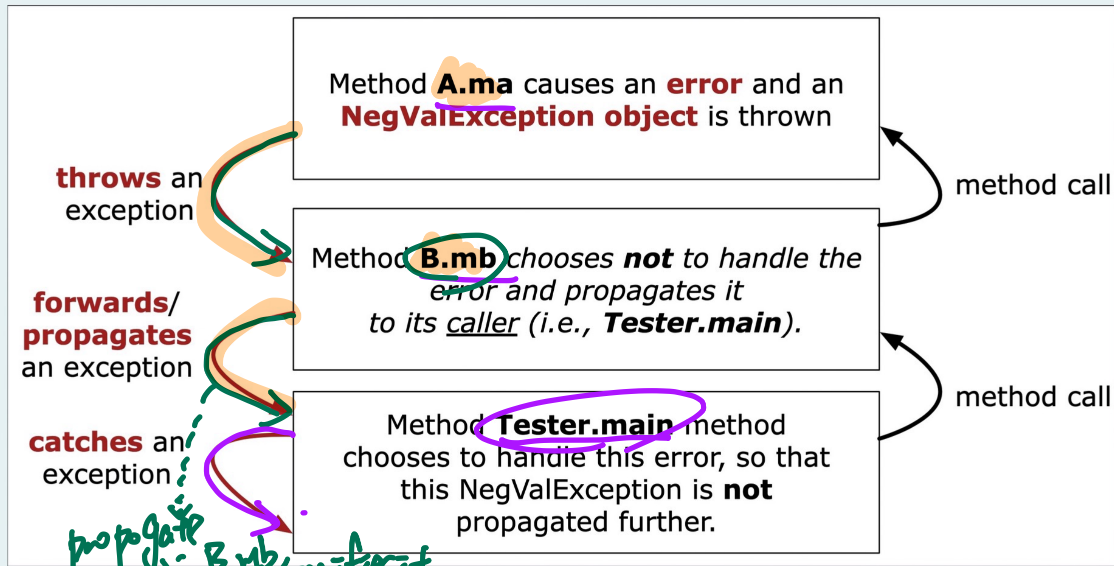
```
/* Title of instructor's
 * ith teaching course
 */
String getTitle(int i) {
    return this.getTe()[i].getTitle();
}
```



Practice Written Test 2

Q1

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:

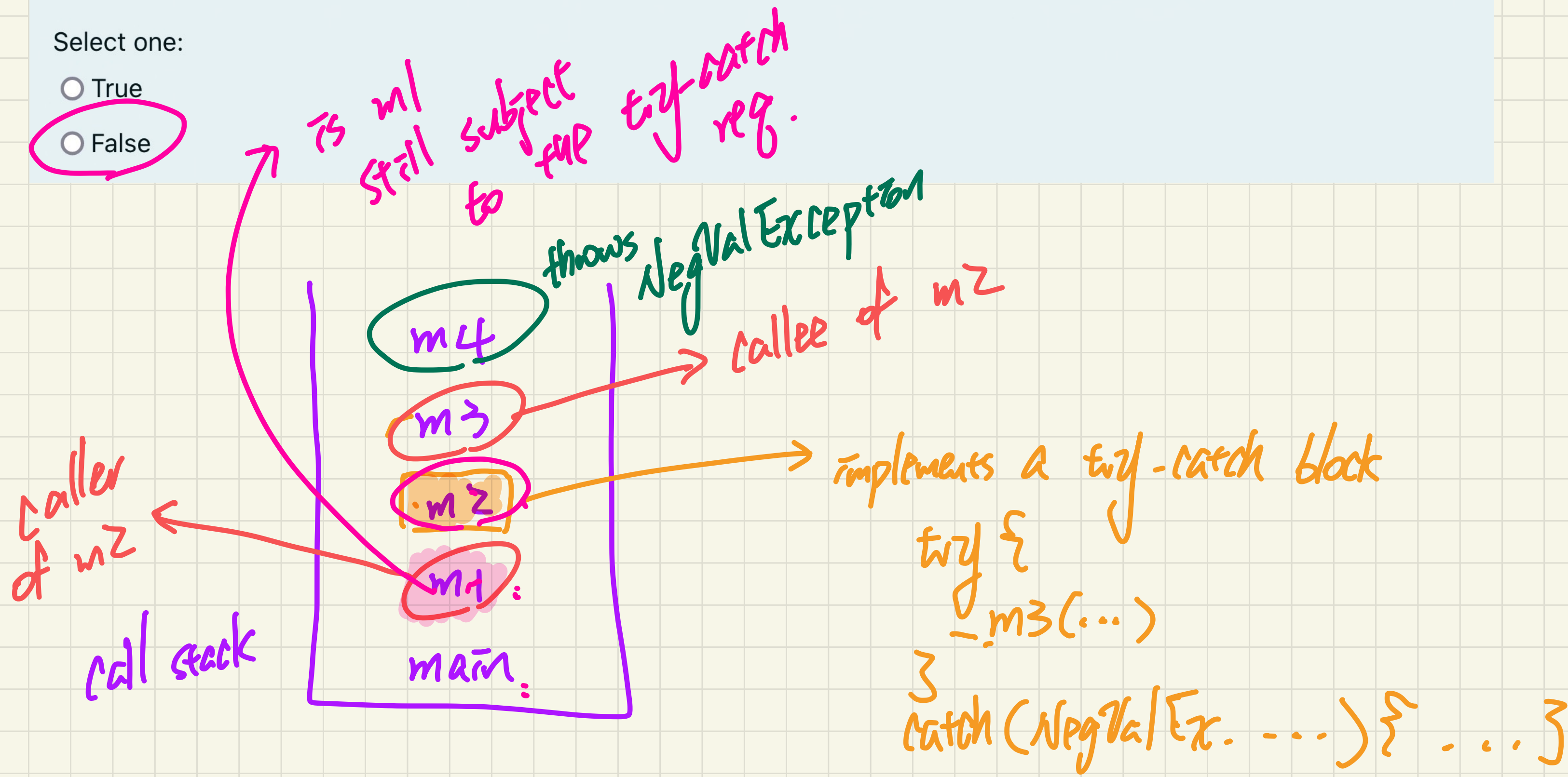
Practice Written Test 2

Q2

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



Practice Written Test 2

Q3

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 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):

2nd line to execute (if any):

3rd line to execute (if any):

4th line to execute (if any):

5th line to execute (if any):

6th line to execute (if any):

7th line to execute (if any):

L3 of CounterTester2	L4 of CounterTester2	L6 of CounterTester2	L7 of CounterTester2	L9 of CounterTester2	L10 of CounterTester2
L13 of CounterTester2	L17 of CounterTester2	Execution Terminated			

Practice Written Test 2

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: ValueTooLargeExcept
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException
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): ____

2nd line to execute (if any): ____

3rd line to execute (if any): ____

4th line to execute (if any): ____

5th line to execute (if any): ____

6th line to execute (if any): ____

7th line to execute (if any): ____

L3 of CounterTester2 | L4 of CounterTester2 | L6 of CounterTester2 | L7 of CounterTester2 | L9 of CounterTester2 | L10 of CounterTester2

L13 of CounterTester2 | L17 of CounterTester2 | Execution Terminated

Practice Written Test 2

Q5

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: ValueTooLargeExce
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeExcepti
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): [ ]

2nd line to execute (if any): [ ]

3rd line to execute (if any): [ ]

4th line to execute (if any): [ ]

5th line to execute (if any): [ ]

6th line to execute (if any): [ ]

7th line to execute (if any): [ ]

[L3 of CounterTester2] [L4 of CounterTester2] [L6 of CounterTester2] [L7 of CounterTester2] [L9 of CounterTester2] [L10 of CounterTester2]

[L13 of CounterTester2] [L17 of CounterTester2] [Execution Terminated]

Practice Written Test 2

Q6

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, whi

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

Practice Written Test 2

Q7

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
- ☐ f. 3
- ☐ g. 4
- ☐ h. None of the listed answers is correct.