EECS2030 (Sec. F) Fall 2024
Advanced Object-Oriented Programming

Lecture Notes

Instructor: Jackie Wang

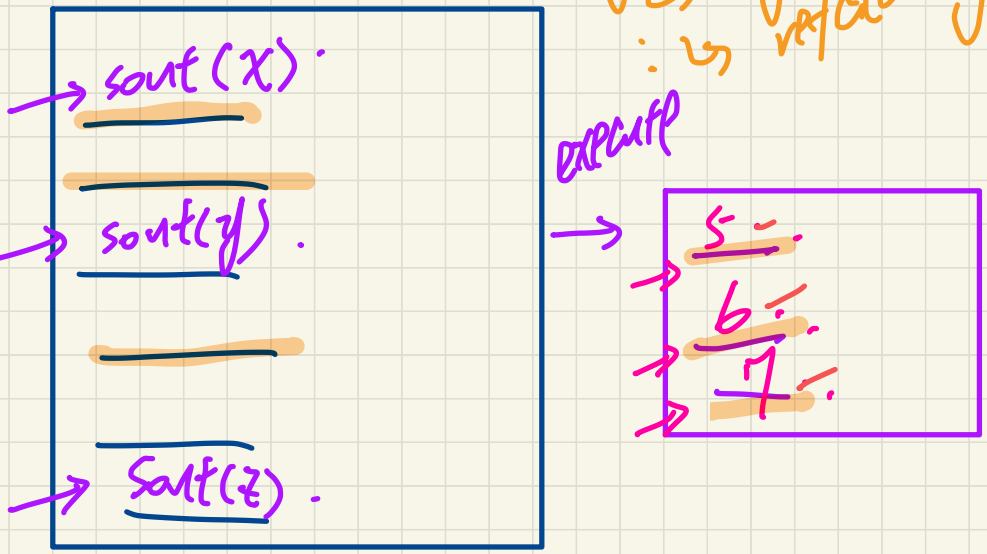
Lecture 1 - Sep. 5

Syllabus & Introduction

Professional Engineers: Code of Ethics
OOP Review Question: Attribute Types

Console Tester vs. JUnit Testing

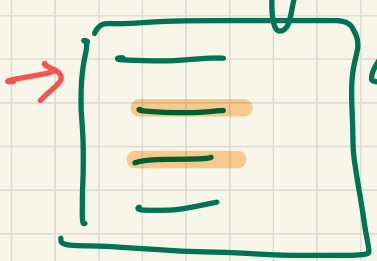
Console Tester

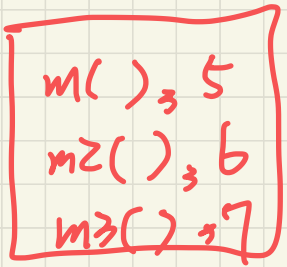


change
↳ bug fix
↳ refactoring.
⋮

expected

Unit Testing





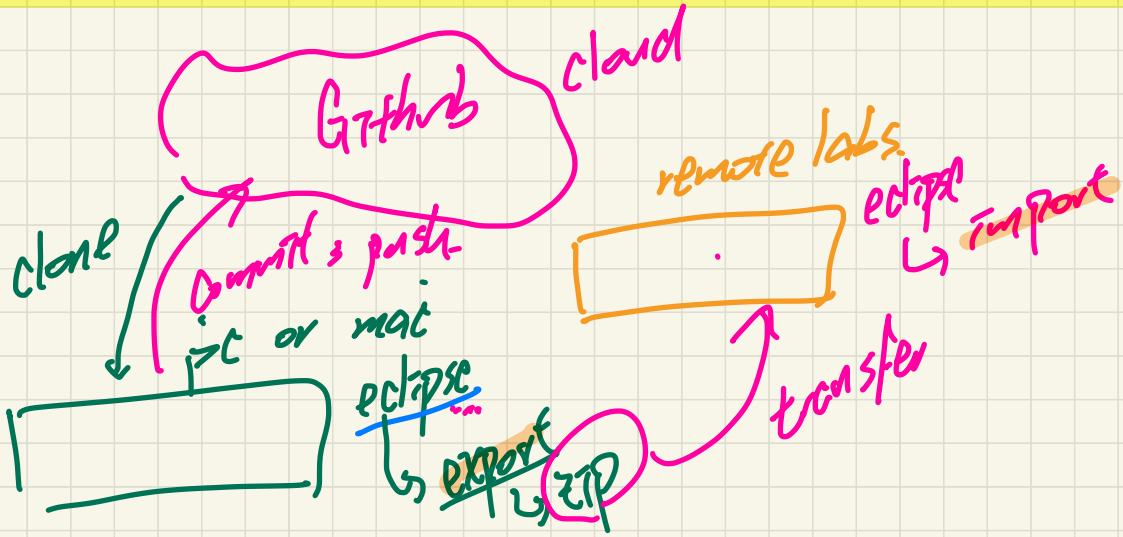
test.

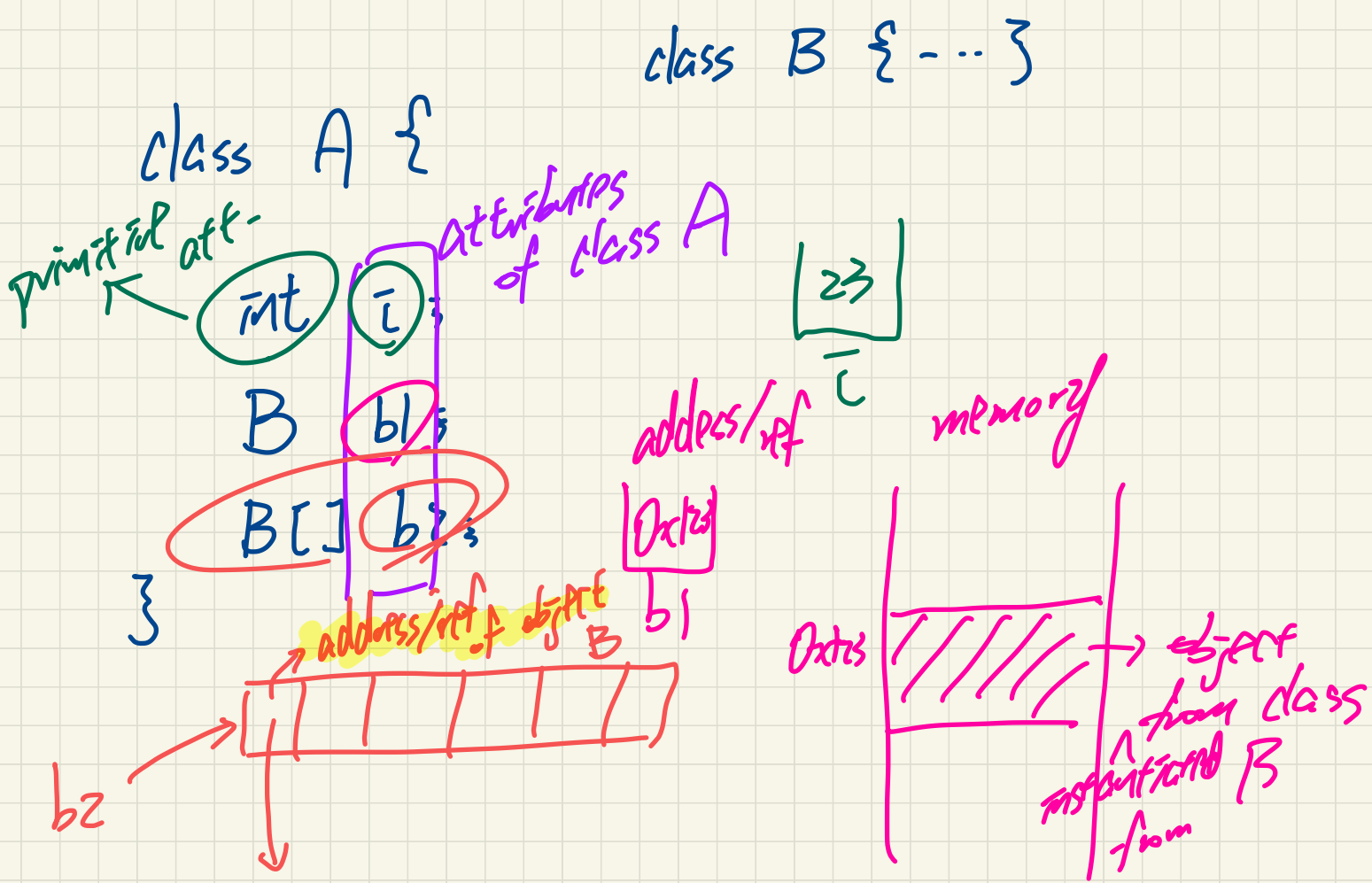
Course Learning Outcomes (CLOs)

CLO1	Implement an Application Programming Interface (API).
CLO2	Test the implementation.
CLO3	Document the implementation.
CLO4	Implement aggregations and compositions.
CLO5	Implement inheritance.
CLO6	Use recursion.
CLO7	Implement linked lists.
CLO8	(Informally) prove that recursive algorithms are correct and terminate.
CLO9	(Informally) analyse the running time of (recursive) algorithms.

Synchronization: Laptop, Github, RemoteLabs

- Cloning Github repositories may not work on a remotelab terminal
- Instead:
  + Develop Java projects on your laptop.
  + **Incremental** commit & push to keep the Github cloud space up to date.
  + To test if the finished project works on Linux, **export** & transfer an archive file to remotelab.
  + On the remotelab, **import** the archive file and re-run all JUnit tests.
  + If everyone works, submit the archive file!




class B {...}
class A {
primitive attr.
int
i;
attributes of class A
23
i
B
b1;
B[]
b2;
}
address/ref
memory
0x123
b1
address/ref of object B
b2
0x123
object from class B
instantiated from

Lecture 2 - Sep. 10

Review of OOP

Observe-Model-Execute Process
Java Data Types
NullPointerException
Parameters vs. Arguments
Scope of Variables

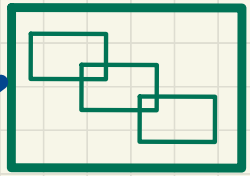
Lab 01 !

Lab 02 → reference-typed attributes

Lab Wed
↳ helper methods.

Separation of Concerns

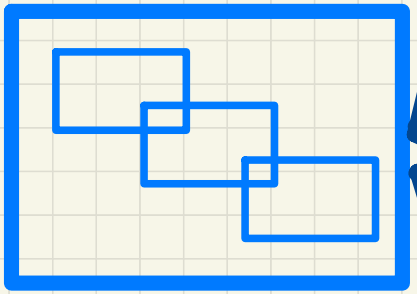
junit_tests



- Expected vs. Actual Values
- Methods
  * calling methods from model
  * assertions
  * containing **no** print statements

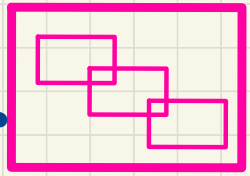
use

model



use

console_apps



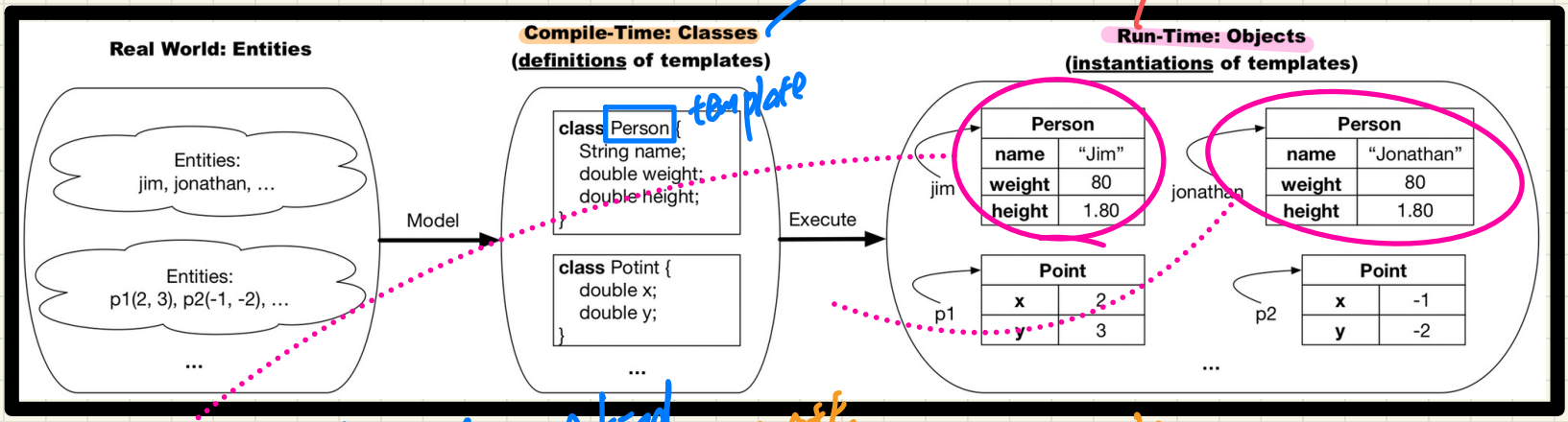
- Classes & Methods
- Methods
  * constructors
  * accessors: return statements
  * mutators: **no** return statements
  * containing **no** print statements

- main method (entry point of execution)
  * reading inputs from keyboard
  * calling methods from model
  * producing outputs to console (print)
  * containing **no** return statements

Observe-Model-Execute Process

development of code in editor

1. compiles
2. executes



template

h. 1.8

different entities of the same kind

h. 1.7

↳ common attr.
common behaviour ↳ change weight

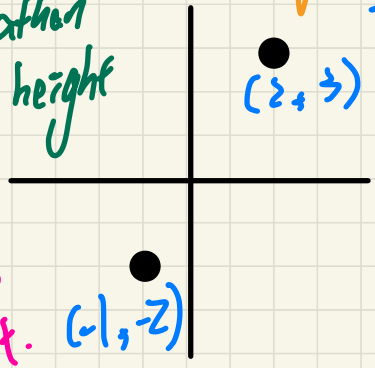
Entities: jim, jonathan

Attributes: weight, height

Changes:

Inquiries: instance-specific

Template: Person. values for att.



Exercise.

Entities:

Attributes:

Changes:

Inquiries:

Template:

Object Oriented Programming (OOP)

- Templates (compile-time Java classes)
  + attributes (common around instances)
  + methods
    * constructors
    * accessors/getters
    * mutators/setters
  + Eclipse: Refactoring

change
↳ - rename variables
- rename methods

- Instances/Entities (runtime objects)
  + instance-specific attribute values
  + calling constructor to create objects
  + using the "dot notation", with the right contexts, to:
    * get attribute values
    * call accessors or mutators

obj.m()
obj.m1().m2()....

Modelling: from Entities to Classes

Identify Critical Nouns & Verbs

Example 1

Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axises. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

Example 2

A person is a being, such as a human, that has certain attributes and behaviour constituting personhood: a person ages and grows on their heights and weights.

OO Thinking: Templates vs. Instances

(Exercise)!

Templates

Person

Common
Attribute Definitions
(Types)

double weight
double height

Common
Behaviour Definitions
(Headers/API)

double getWeight()
void gainWeight(double w)
double getBMI()

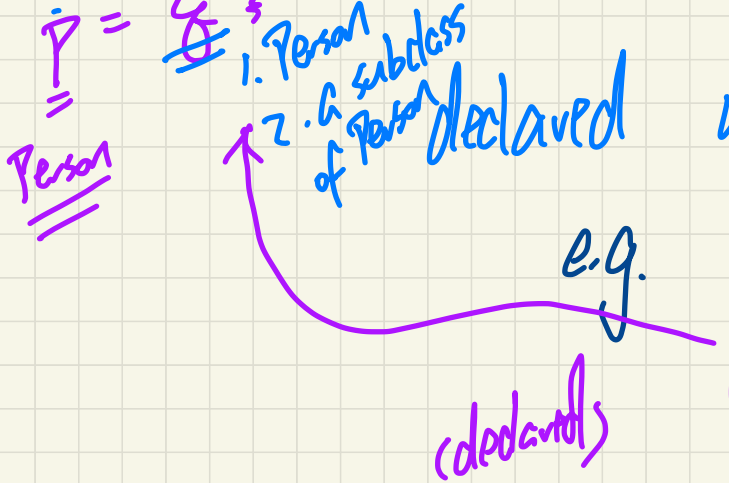
Instance-Specific
Attribute Values

object 1: W: 46.3
object 2: W: 70.7

Instance-Specific
Behaviour Occurrence

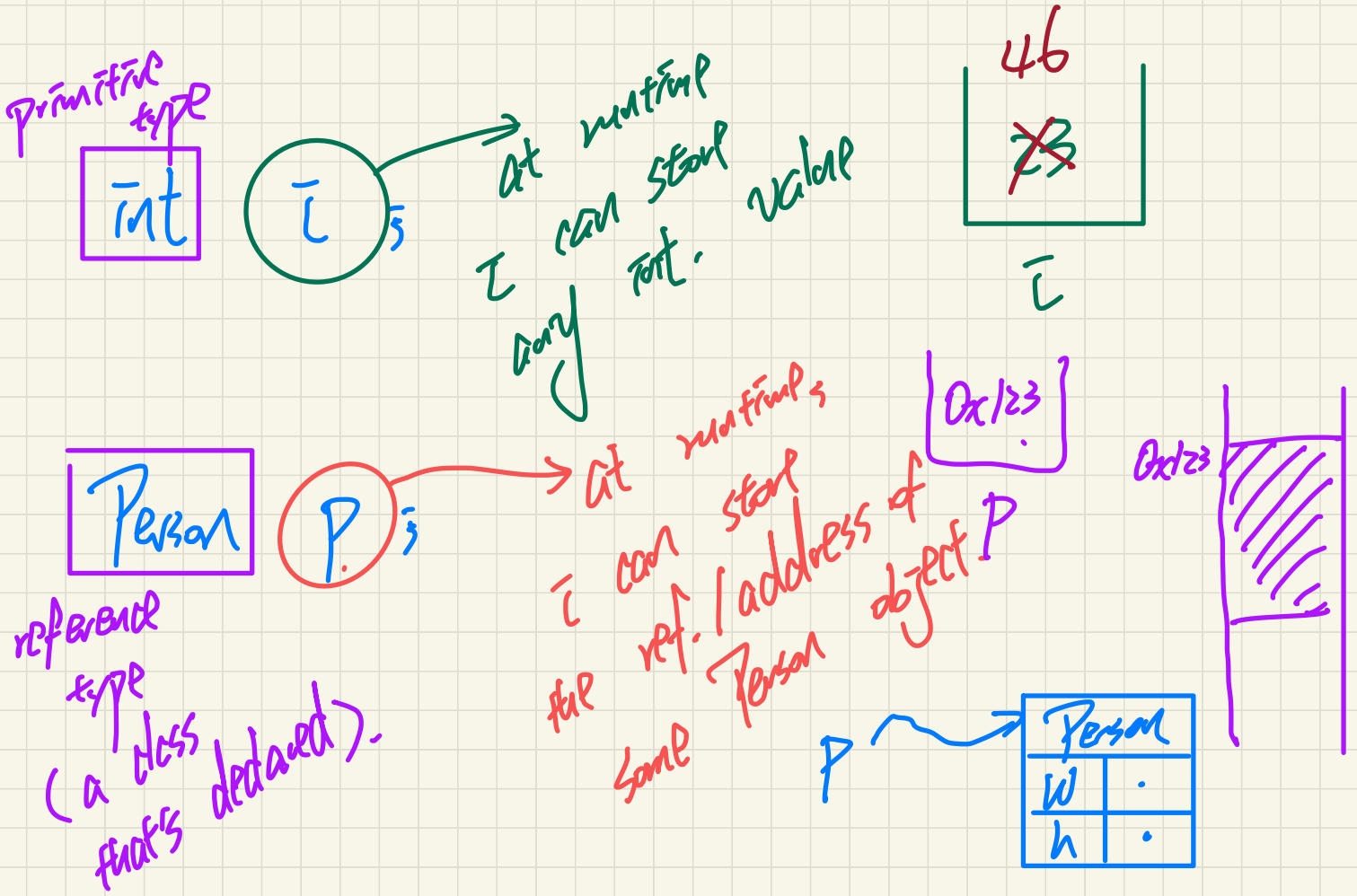
objects.
diff. content
obj1
obj2
gainW(2)
gainW(2)
same method with same input invoked

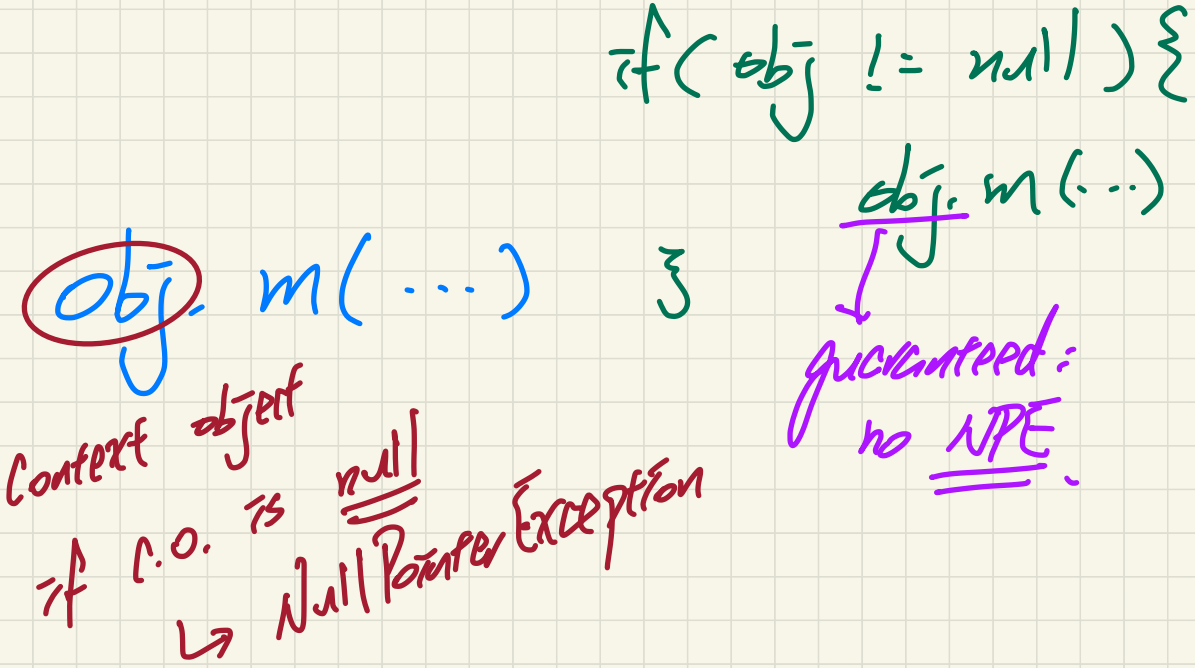
- In Java, each variable must be declared with a type.



i = 23 ;
i = 46 ;
i = 3b. 7 ; X
i = p ; X

- A variable's type denotes the set of values that is allowed to be stored in that variable.

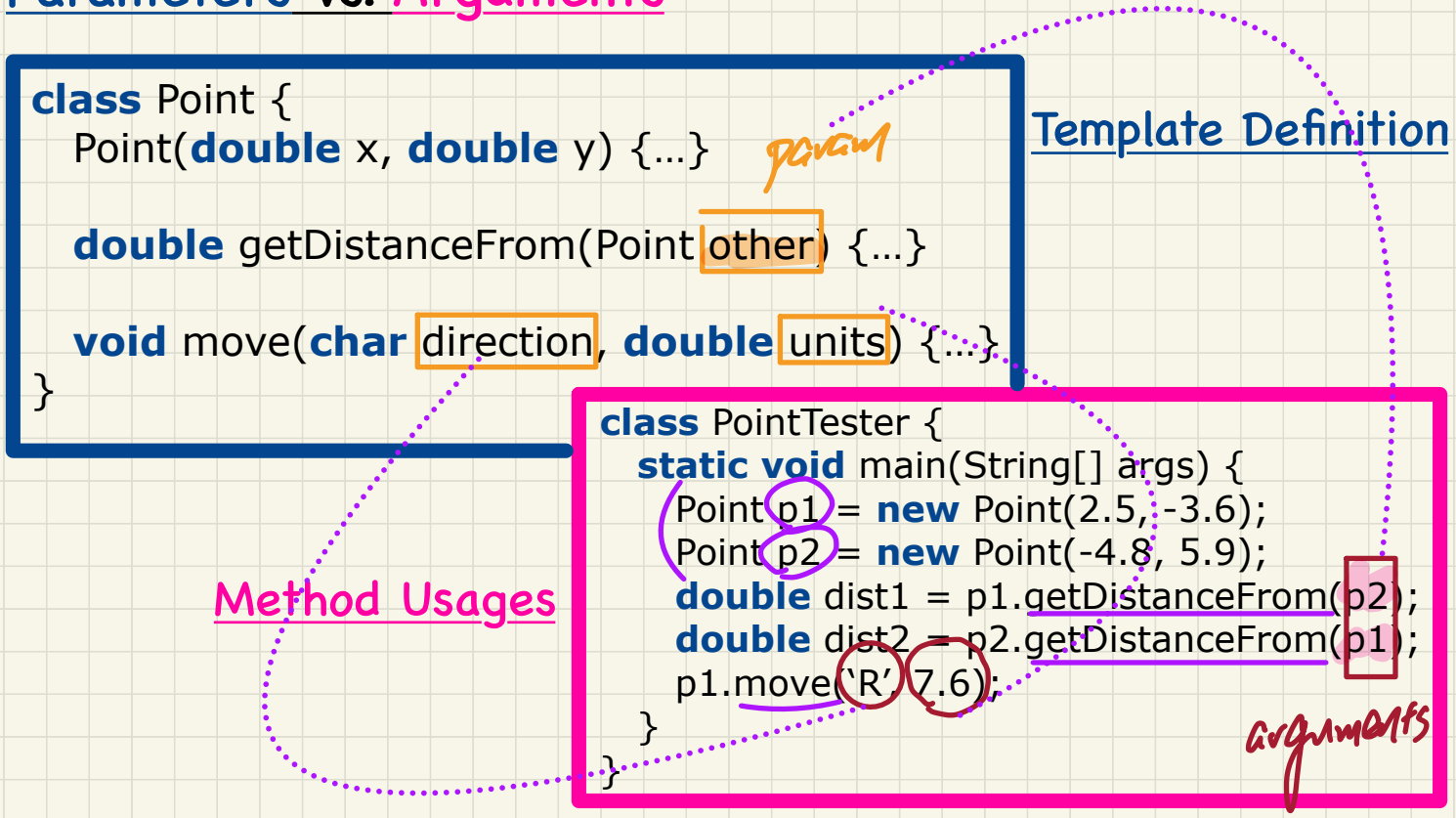

primitive type
int
i
5
At runtime i can store any int. value
46
23
i
Person
p
5
reference type (a class that's declared).
At runtime i can store the ref. / address of some Person object P
0x123
0x123
P
Person
W
h


if(obj != null){
obj.m(...)
}
obj.m(...)
Context object
if c.o. is null
↳ NullPointerException
guaranteed:
no NPE.

slides 16 – 47

↳ self study.

Parameters vs. Arguments



Scope of Variables in a Class

- Class-level variables may be accessed/modified between methods.

[Assume for now: non-static]

- Method-level parameters may be accessed within the declared method only.
- Method-level (local) variables may be accessed/modified within the declared method.

```
class MyClass {
  int i;
  int j;
  void m1 (int p1) {
    int m;
  }
  void m2 (int p2) {
    int n;
  }
}
```

class-level

i = 23; m = p1; m = p2; X

i = 46; n = i; n = m; X

In-Lab Demo - Sep. 11

Helper Methods

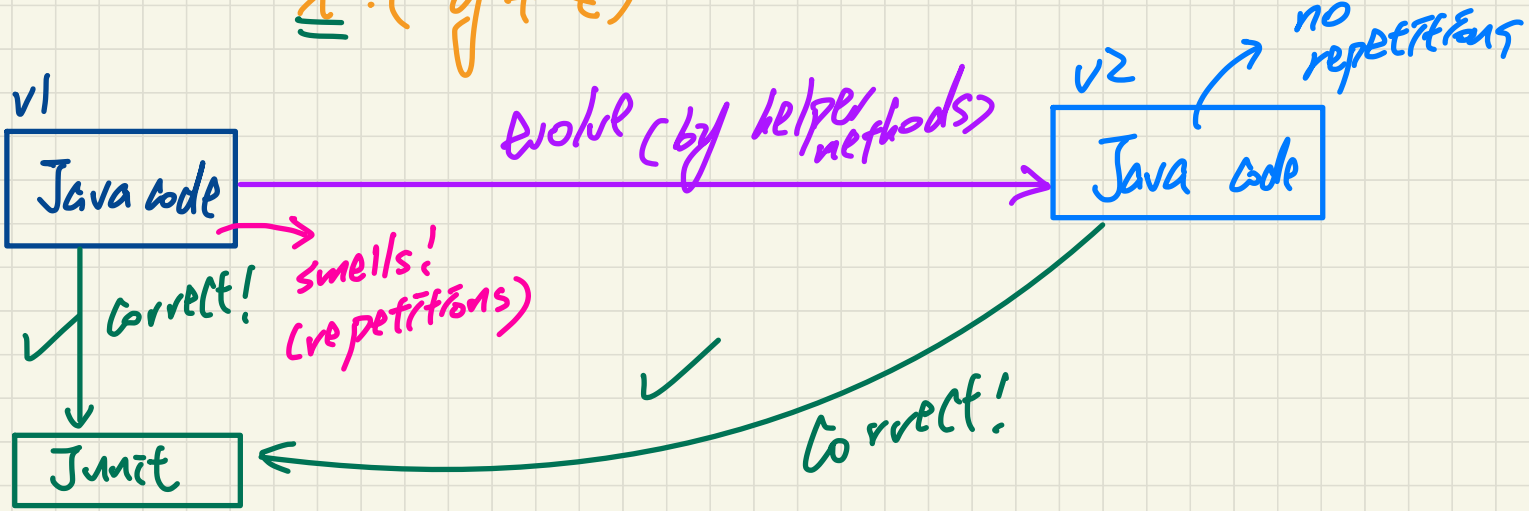
Code Smell
Refactoring
Breakpoints & Debugger

Refactoring

$$x \cdot y + x \cdot z$$

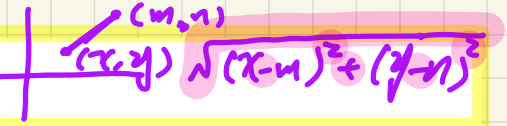
(refactor factor out x

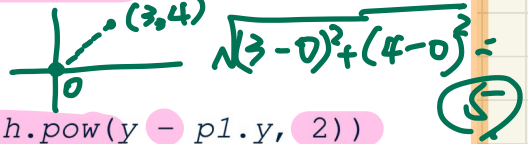
$$x \cdot (y + z)$$



Point: without a Helper Method

```
class Point { /* code smells:repetitions! */
  double x; double y;

  double getDistanceFromOrigin() {
    return Math.sqrt(Math.pow(this.x - 0, 2) + Math.pow(this.y - 0, 2));

  double getDistancesTo(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2)); }

  double getTriDistances(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2))
      +
      Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
    }
}
```





0.0001

assertEquals(actual, expected, ε)

↓

expected - ε ≤ actual ≤ expected + ε

Point: with a Helper Method

```
public class Point { /* Code Smell Eliminated */
  private double x; private double y;
  double getDistanceFrom(double otherX, double otherY) {
    return Math.sqrt(Math.pow(ohterX - this.x, 2) +
           Math.pow(otherY - this.y, 2));
  }
  double getDistanceFromOrigin() {
    return this.getDistanceFrom(0, 0);
  }
  double getDistancesTo(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y);
  }
  double getTriDistances(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y) +
           p1.getDistanceFrom(p2.x, p2.y)
  }
}
```

Lecture 3 - Sep. 12

Review of OOP

Use of this for Attribute Access
Tracing OO Code
Use of Mutators vs. Accessors
Design of Method Parameters
Reference Aliasing

Announcements/Reminders

- LabOP1 due tomorrow (Friday) at 12 noon!
- Priority: LabOP2 (tutorial videos + PDFs)
- Yesterday's in-lab demo materials (helper methods) released.

Constructors not using this Keyword

model

```java
public class Person {
    /*
     * Attributes.
     * Person instances have the same attribute names.
     * Person instances have specific attribute values.
     */
    double weight;
    double height;

    /*
     * Constructors
     */
    public Person() {

    }

    public Person(double newWeight, double newHeight) {
        weight = newWeight;
        height = newHeight;
    }
}
```

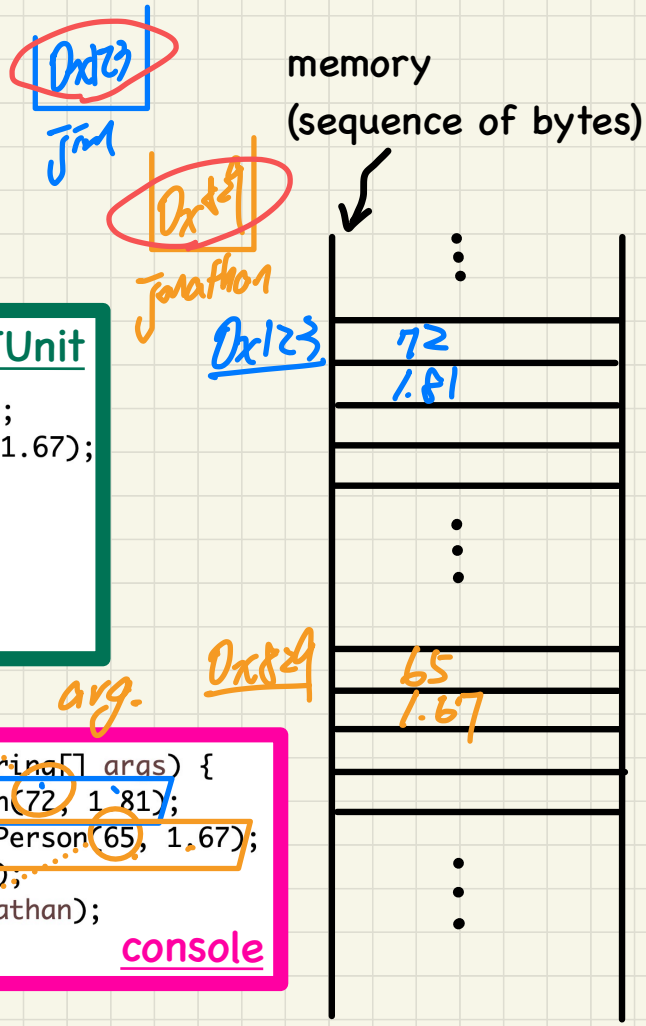
JUnit

```java
@Test
public void test_1() {
    Person jim = new Person(72, 1.81);
    Person jonathan = new Person(65, 1.67);
    assertTrue(jim != jonathan);
    assertFalse(jim == jonathan);
    assertNotSame(jim, jonathan);
    assertNotEquals(jim, jonathan);
}
```

console

```java
public static void main(String[] args) {
    Person jim = new Person(72, 1.81);
    Person jonathan = new Person(65, 1.67);
    System.out.println(jim);
    System.out.println(jonathan);
}
```

- Default Constructor?
- Parameters vs. Arguments
- Reference Variables



Constructors not using this Keyword

model

```
public class Person {
    /*
     * Attributes.
     * Person instances have the same attribute names.
     * Person instances have specific attribute values.
     */
    double weight;
    double height;

    /*
     * Constructors
     */
    public Person() {

    }

    public Person(double newWeight, double newHeight) {
        weight = newWeight;
        height = newHeight;
    }
}
```

Question

- What if names of parameter & attribute are the same?
- implicit "this"

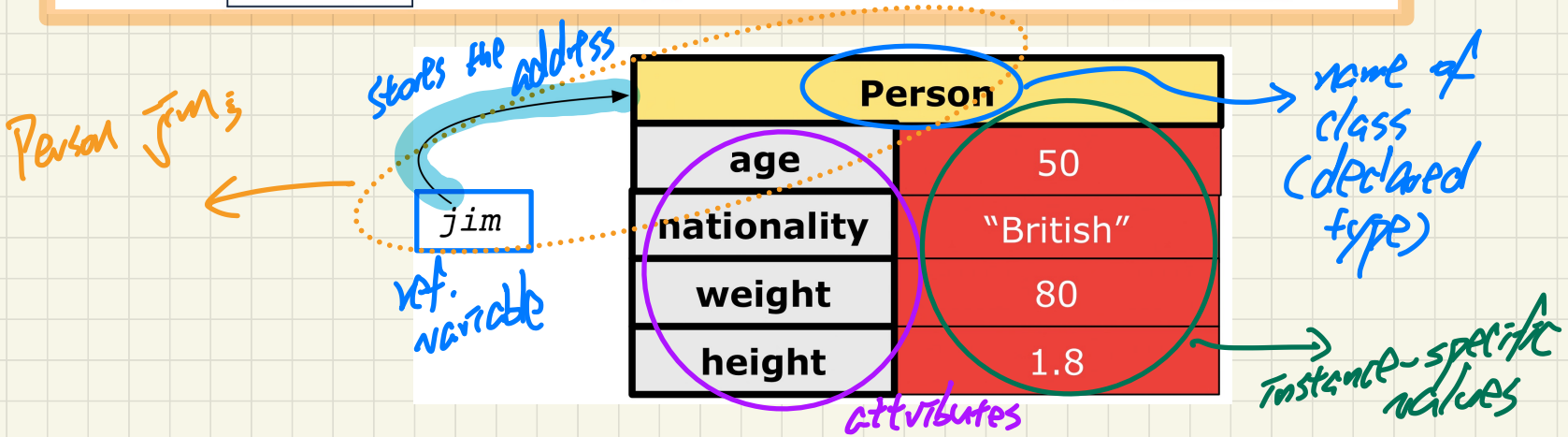
(variable) shadowing

1. does not initialize att. weight
2. instead, parm w. is assigned to itself.

Tracing OO Code: Visualizing Objects

To visualize an object:

- Draw a **rectangle box** to represent *contents* of that object:
  - **Title** indicates the *name of class* from which the object is instantiated.
  - **Left column** enumerates *names of attributes* of the instantiated class.
  - **Right column** fills in *values* of the corresponding attributes.
- Draw **arrow(s)** for *variable(s)* that store the object's *address*.



Effects of Creating New Objects

model

```
public class Person {
    /*
     * Attributes.
     * Person instances have the same attribute names.
     * Person instances have specific attribute values.
     */
    double weight;
    double height;

    /*
     * Constructors
     */
    public Person() {

    }

    public Person(double weight, double height) {
        this.weight = weight;
        this.height = height;
    }
}
```

- Variable Shadowing
- Visualizing Objects
- Context Object
- this
- dot notation

JUnit

```
@Test
public void test1() {
    Person jim = new Person(72, 1.81);
    Person jonathan = new Person(65, 1.67);
    assertTrue(jim != jonathan);
    assertFalse(jim == jonathan);
    assertNotSame(jim, jonathan);
    assertNotEquals(jim, jonathan);
}
```

Person jim = new Person(72, 1.81);

① ② ③

① declaring a ref var.

jim

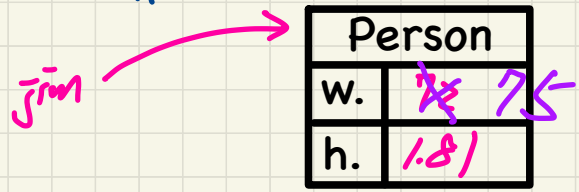
② creating an anonymous obj

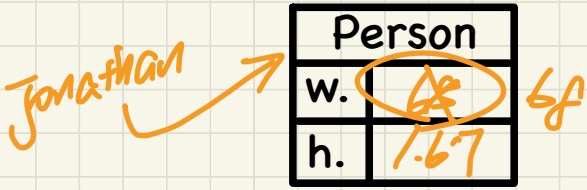
Person	
w.	72
h.	1.81

indicates that the class-level "weight" is refered to.

Accessors/Getters vs. Mutators/Setters

$bmi = \frac{w}{h^2}$






```java
public class Person {
    /*
     * Attributes.
     * Person instances have the same attribute names.
     * Person instances have specific attribute values.
     */
    double weight;
    double height;

    /* Accessors/Getters */
    public double getBMI() {
        double bmi = this.weight / (this.height * this.height);
        return bmi;
    }

    /* Mutators/Setters */
    public void gainWeightBy(double amount) {
        this.weight = this.weight + amount;
    }
}
```

```java
@Test
public void test_3() {
    Person jim = new Person(72, 1.81);
    Person jonathan = new Person(65, 1.67);

    assertEquals(21.977, jim.getBMI(), 0.01);
    assertEquals(23.307, jonathan.getBMI(), 0.01);

    jim.gainWeightBy(3);
    jonathan.gainWeightBy(3);

    assertEquals(22.893, jim.getBMI(), 0.01);
    assertEquals(24.382, jonathan.getBMI(), 0.01);
}
```

Use of Accessors vs. Mutators


```
class Person {
  void setWeight(double weight) { ... }
  double getBMI() { ... }
}
```

mutator

accessor

- Calls to *mutator methods* *cannot* be used as values.
  - e.g., `System.out.println(jim.setWeight(78.5));` ×
  - e.g., `double w = jim.setWeight(78.5);` ×
  - e.g., `jim.setWeight(78.5);` ✓
- Calls to *accessor methods* *should* be used as values.
  - e.g., `jim.getBMI();` compiles! but the return val not used! ■
  - e.g., `System.out.println(jim.getBMI());` ✓
  - e.g., `double w = jim.getBMI();` ✓

void

void

X

Method Parameters

- **Principle 1:** A *constructor* needs an *input parameter* for every attribute that you wish to initialize.

  e.g., `Person(double w, double h)` vs. `Person(String fName, String lName)`

- **Principle 2:** A *mutator* method needs an *input parameter* for every attribute that you wish to modify.

  e.g., In `Point`, `void moveToXAxis()` vs. `void moveUpBy(double unit)`

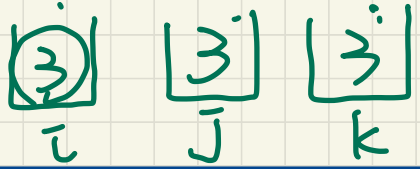
- **Principle 3:** An *accessor method* needs *input parameters* if the attributes alone are not sufficient for the intended computation to complete.

  e.g., In `Point`, `double getDistFromOrigin()` vs. `double getDistFrom(Point other)`

Copying Primitive vs. Reference Values

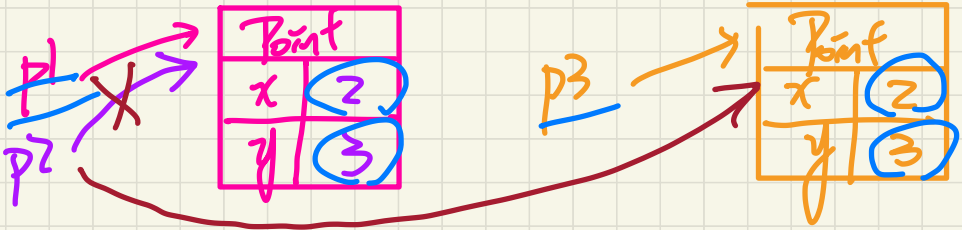
Primitive

```
int i = 3;
int j = i;    System.out.println(i == j);/*true*/
int k = 3;    System.out.println(k == i && k == j);/*true*/
```



Reference

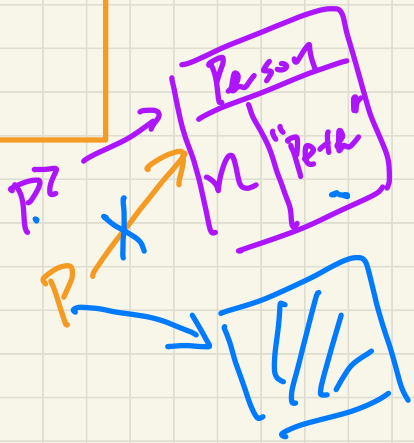
```
Point p1 = new Point(2, 3);
Point p2 = p1;  System.out.println(p1 == p2);/*true*/
Point p3 = new Point(2, 3);
Systme.out.println(p3 == p1 || p3 == p2);/*false*/
Systme.out.println(p3.x == p1.x && p3.y == p1.y);/*true*/
Systme.out.println(p3.x == p2.x && p3.y == p2.y);
```



p2 = p3 = 3

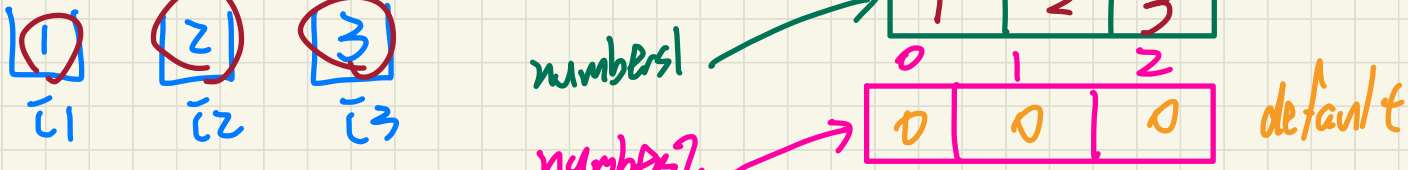
Fall 2022 2050

```
Person p2 = new Person("Pete");
m1(p2);
print(p2.name);

void m1(Person p) {
    p = new Person("John");
}

m2(p2); print(p2.name);

void m2(Person p) {
    p.changeName("John");
}
```



Copying Primitive Values

```
int i1 = 1;
int i2 = 2;
int i3 = 3;
int[] numbers1 = {i1, i2, i3};
int[] numbers2 = new int[numbers1.length];
for(int i = 0; i < numbers1.length; i ++) {
  numbers2[i] = numbers1[i];
}
numbers1[0] = 4;
System.out.println(numbers1[0]);
System.out.println(numbers2[0]);
```



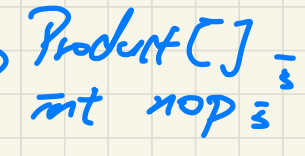
Lecture 4 - Sep. 17

Review of OOP

Reference Aliasing & Primitive Arrays
Ref-Typed Parameters vs. Return Values

Announcements/Reminders

- Lab0P2 due this Friday at 12 noon!
- Lab1 to be released after Lab0P2 is due.
- Priority: Lab0P2 (tutorial videos + PDFs)
- This Thursday's office hour will be re-scheduled.



Copying Primitive Values

```
int i1 = 1;
int i2 = 2;
int i3 = 3;
int[] numbers1 = {i1, i2, i3};
int[] numbers2 = new int[numbers1.length];
for(int i = 0; i < numbers1.length; i ++) {
numbers2[i] = numbers1[i];
}
numbers1[0] = 4;
System.out.println(numbers1[0]);
System.out.println(numbers2[0]);
```

0 1 2

3

1st it. i == 0

n2[0] = n1[0];

2nd it. i == 1

n2[1] = n1[1];

3rd it. i == 2

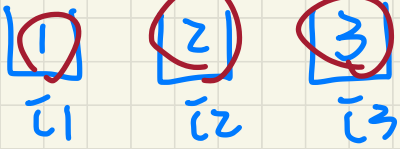
n2[2] = n1[2]

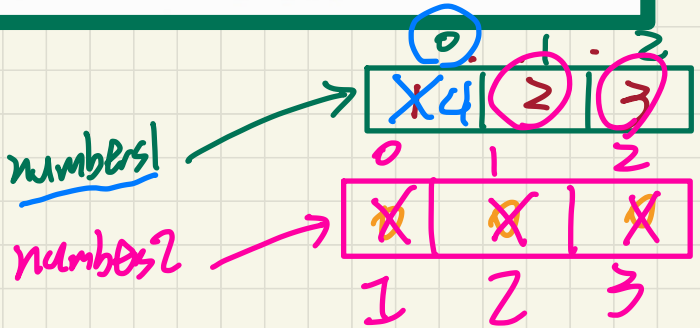
after i++

i == 3

i < n1.length

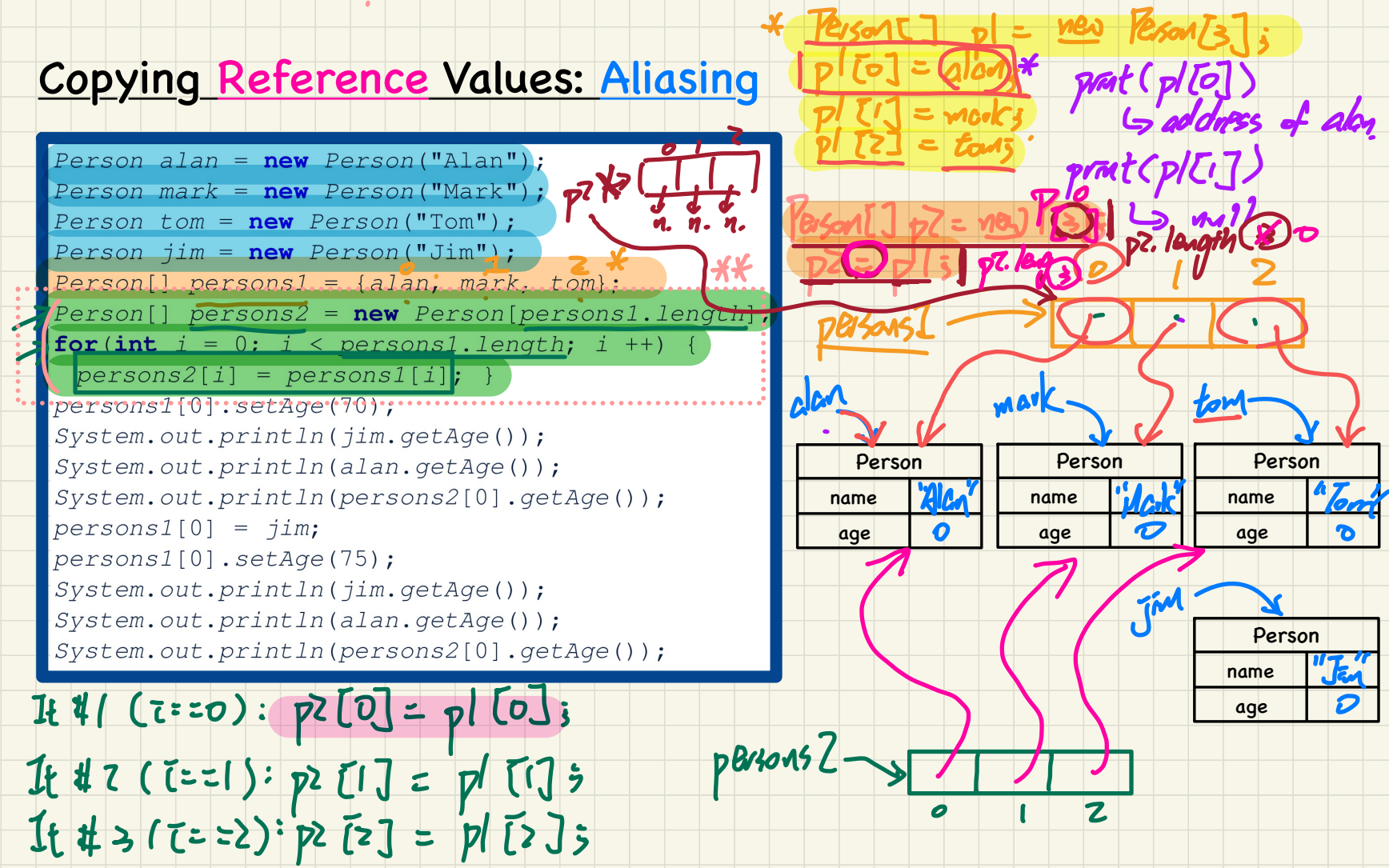
F





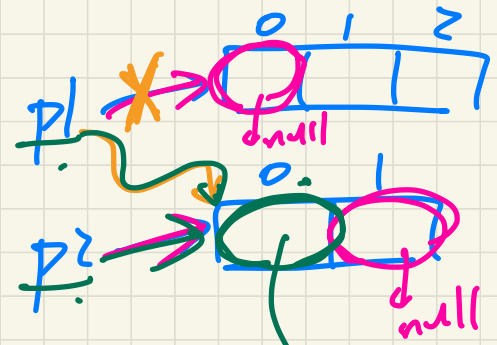
Copying Reference Values: Aliasing

```
Person alan = new Person("Alan");
Person mark = new Person("Mark");
Person tom = new Person("Tom");
Person jim = new Person("Jim");
Person[] persons1 = {alan, mark, tom};
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons1.length; i ++) {
  persons2[i] = persons1[i]; }
persons1[0].setAge(70);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
persons1[0] = jim;
persons1[0].setAge(75);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
```



```
Person p1 = new Person[3];
Person p2 = new Person[2];
```

jvm → Person



p1.length 3

p2.length 2

p1 == p2 false

p1[0] == p2[1] true

null null

p1[0] == jvm ✓ p1[0] == p2[0] ✓

p2[0] == jvm

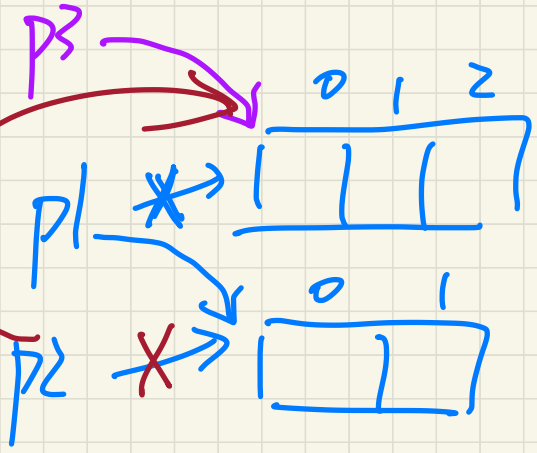
aliasing (multiple variables store same address)

p1 = p2;

p1[0] = jvm;

p1 == p2

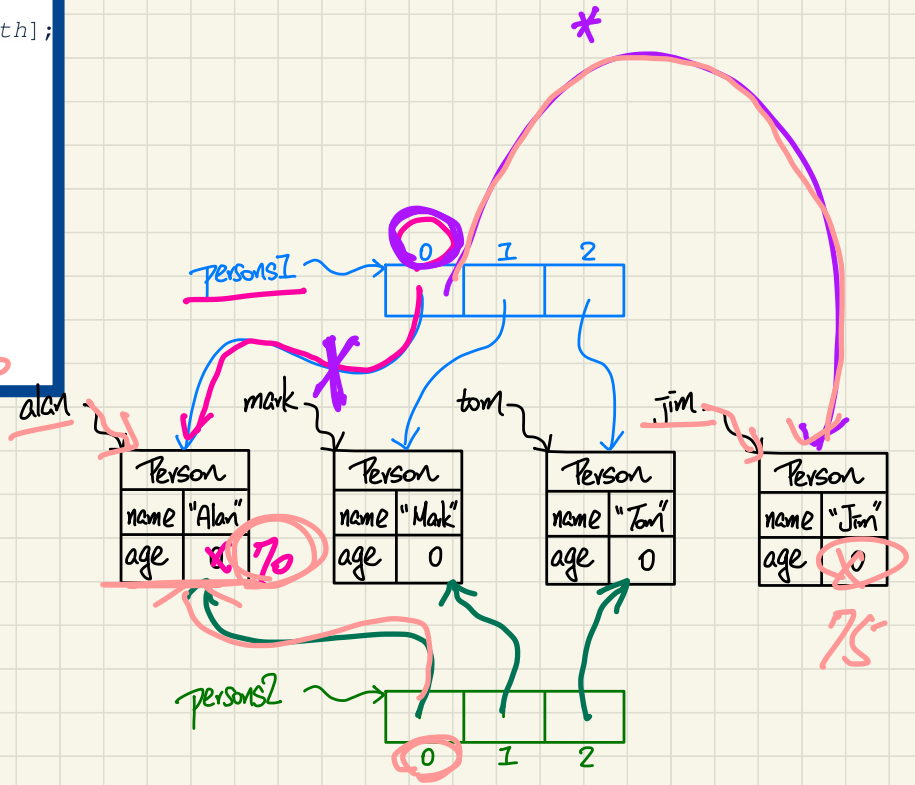
```
Person[] p1;
Person[] p2;
Person[] p3;
```




```
p1 = new Person[3];
p2 = new Person[2];
p3 = p1;
p1 = p2;
```

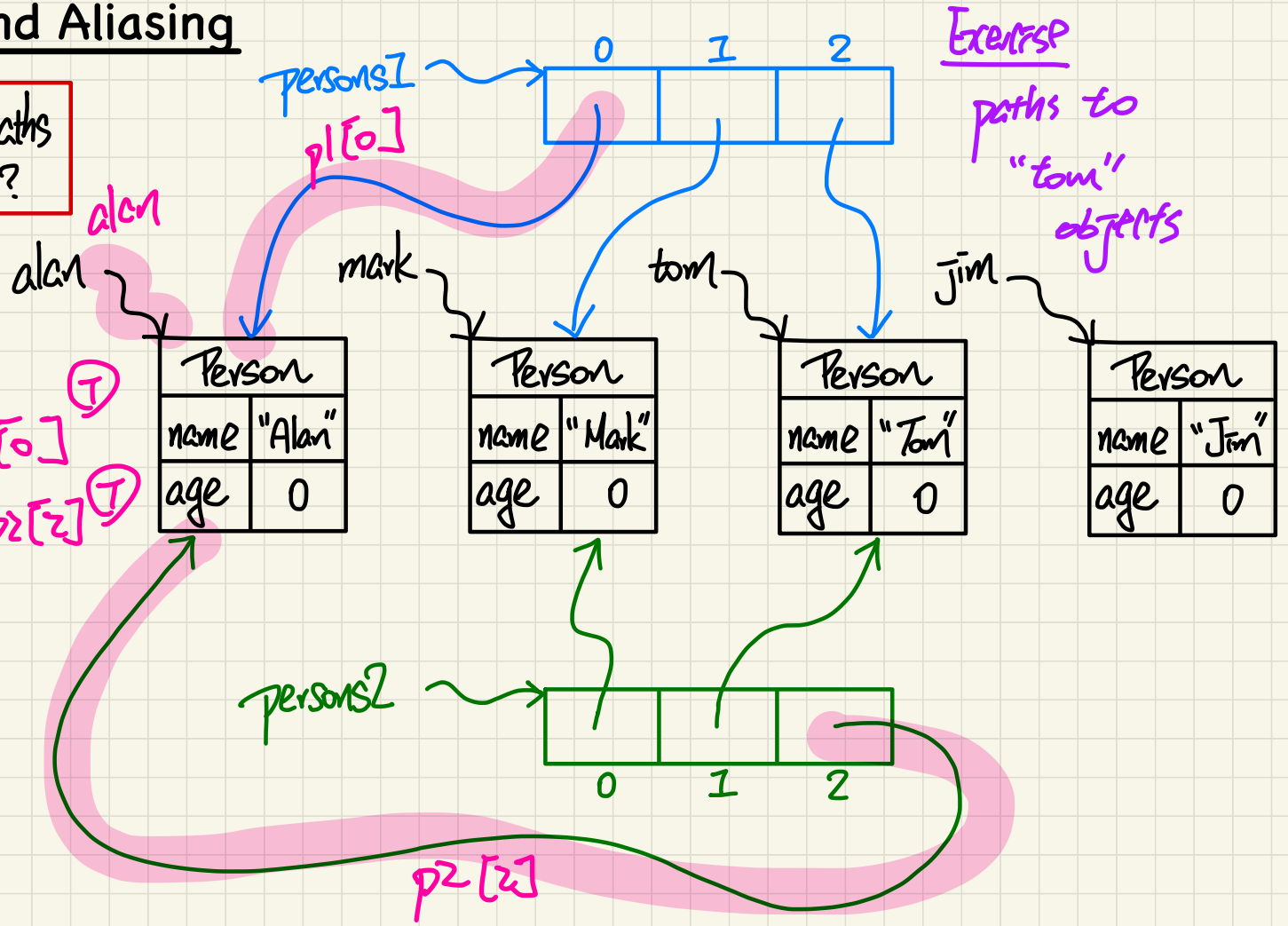
```
p2 = p3;
```

```java
Person alan = new Person("Alan");
Person mark = new Person("Mark");
Person tom = new Person("Tom");
Person jim = new Person("Jim");
Person[] persons1 = {alan, mark, tom};
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons1.length; i ++) {
  persons2[i] = persons1[i]; }
persons1[0].setAge(70);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
persons1[0] = jim;
persons1[0].setAge(75);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
```



Arrays and Aliasing

All alias paths to "Alan"?



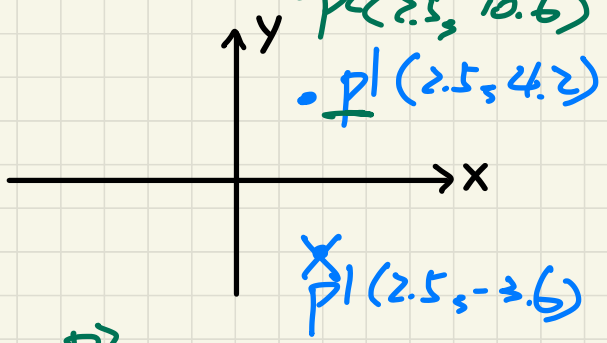
Exercise

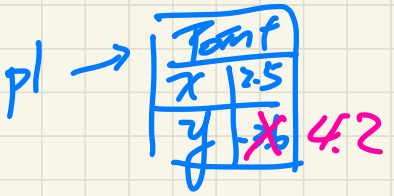
paths to "tom" objects

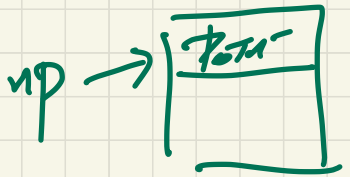
Reference-Typed Return Values


```
public class Point {
   /* A mutator modifying the context Point object */
   public void moveUp (int i) {
      this.y = this.y + i;
   }

   /* An accessor returning a new Point object */
   public Point movedUpBy(int i) {
      Point np = new Point(this.x, this.y);
      np.moveUp(i);
      return np;
   }
}
```

```
public class PointTester {
   public static void main(String[] args) {
      Point p1 = new Point(2.5, -3.6);
      p1.moveUp(7.8);
      Point p2 = p1.movedUpBy(6.4);
      System.out.println(p1 == p2);
   }
}
```



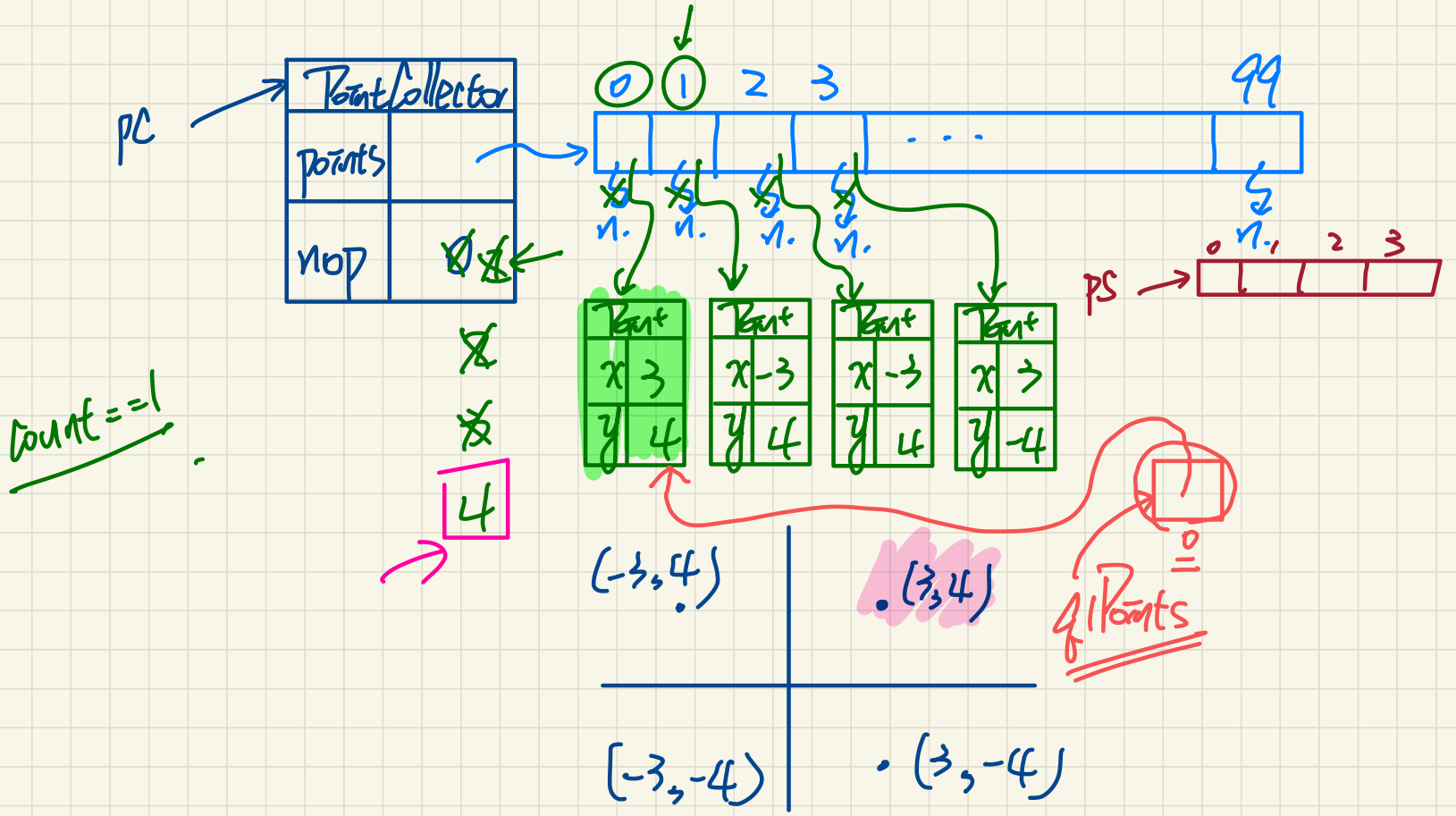




In-Lab Demo - Sep. 18

Programming Pattern

Reference-Typed Arrays
Integer Counter
Breakpoints & Debugger


PC
PointCollector
points
nop
0
1
2
3
99
n.
ps
Point
x 3
y 4
x -3
y 4
x -3
y 4
x 3
y -4
count == 1
4
(-3,4)
(3,4)
(-3,-4)
(3,-4)
qlPoints
0

Programming Pattern

```java
public class PointCollector {
    private Point[] points;
    private int nop; /* number of points */

    public PointCollector() {
        this.points = new Point[100];
    }

    public void addPoint(double x, double y) {
        this.points[this.nop] = new Point(x, y);
        this.nop++;
    }

    public int getNumberOfPoints() {
        return this.nop;
    }
    public Point[] getPointsInQuadrantI() {
        Point[] ps = new Point[this.nop];
        int count = 0;

        /* number of points in Quadrant I */
        for(int i = 0; i < this.nop; i++) {
            Point p = this.points[i];
            if(p.getX() > 0 && p.getY() > 0) {
                ps[count] = p; count++;
            }
        }
        Point[] q1Points = new Point[count];

        /* ps contains null if count < nop */
        for(int i = 0; i < count; i++) {
            q1Points[i] = ps[i];
        }
        return q1Points;
    }
}
```

# of q1 points

count == 1

```java
@Test
public void test() {
    PointCollector pc = new PointCollector();
    assertEquals(0, pc.getNumberOfPoints());

    pc.addPoint(3, 4);
    assertEquals(1, pc.getNumberOfPoints());

    pc.addPoint(-3, 4);
    assertEquals(2, pc.getNumberOfPoints());

    pc.addPoint(-3, -4);
    assertEquals(3, pc.getNumberOfPoints());

    pc.addPoint(3, -4);
    assertEquals(4, pc.getNumberOfPoints());

    Point[] ps = pc.getPointsInQuadrantI();
    assertEquals(1, ps.length);
    assertTrue(ps[0].getX() == 3 && ps[0].getY() == 4);
}
```

In-Lab Demo - Sep. 18

Programming Pattern

Reference-Typed Arrays
Integer Counter
Breakpoints & Debugger

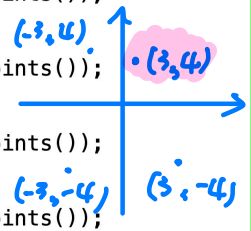
Programming Pattern

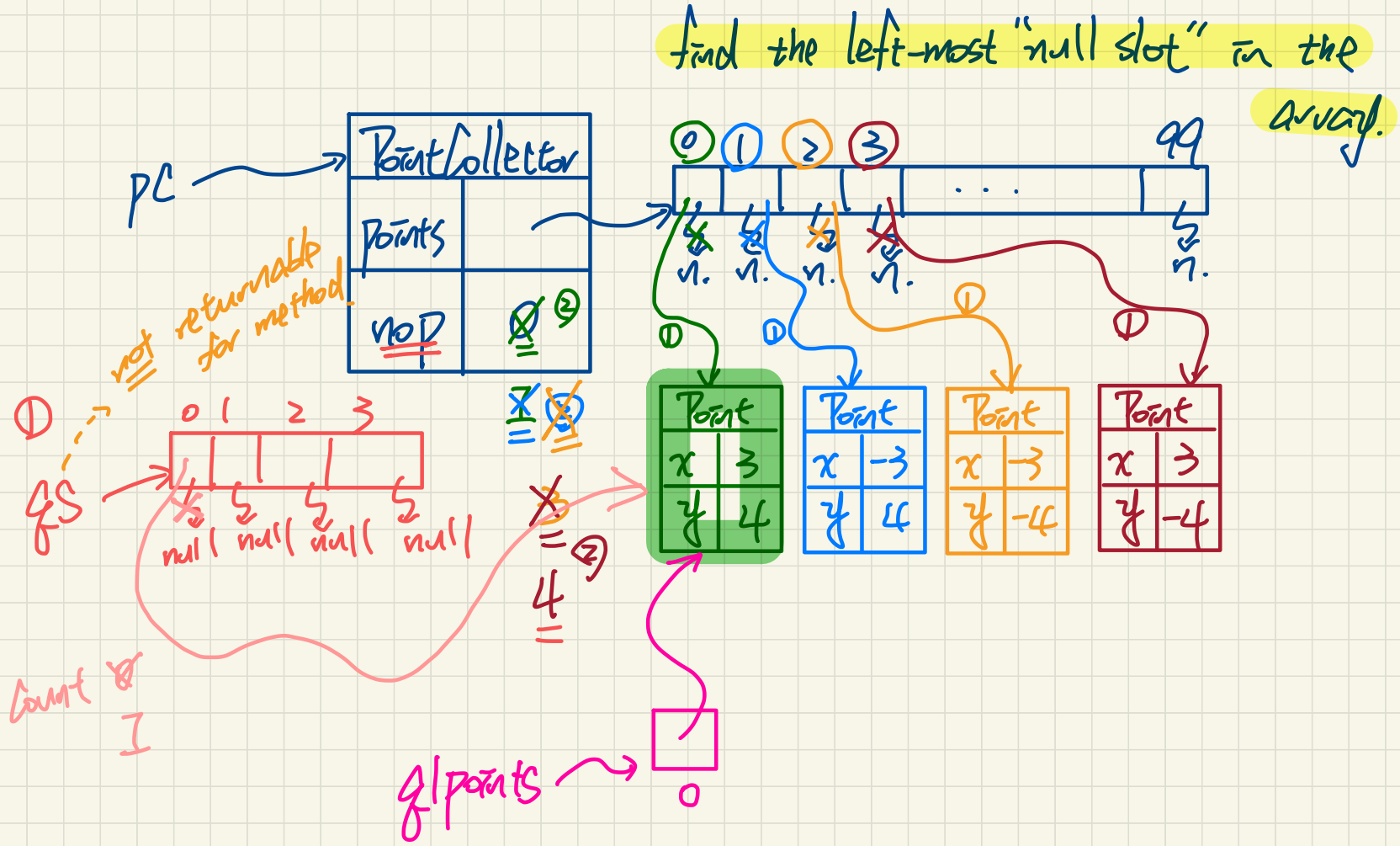
```java
public class PointCollector {
    private Point[] points;
    private int nop; /* number of points */

    public PointCollector() {
        this.points = new Point[100];
    }

    public void addPoint(double x, double y) {
        this.points[this.nop] = new Point(x, y);
        this.nop++;
    }

    public int getNumberOfPoints() {
        return this.nop;
    }

    public Point[] getPointsInQuadrantI() {
        Point[] ps = new Point[this.nop];
        int count = 0;

        /* number of points in Quadrant I */
        for(int i = 0; i < this.nop; i++) {
            Point p = this.points[i];
            if(p.getX() > 0 && p.getY() > 0) {
                ps[count] = p; count++;
            }
        }
        Point[] q1Points = new Point[count];

        /* ps contains null if count < nop */
        for(int i = 0; i < count; i++) {
            q1Points[i] = ps[i];
        }
        return q1Points;
    }
}
```

figure out # of Q1 points.

knowing the # of Q1 p

```java
@Test
public void test() {
    PointCollector pc = new PointCollector();
    assertEquals(0, pc.getNumberOfPoints());

    pc.addPoint(3, 4);
    assertEquals(1, pc.getNumberOfPoints());

    pc.addPoint(-3, 4);
    assertEquals(2, pc.getNumberOfPoints());

    pc.addPoint(-3, -4);
    assertEquals(3, pc.getNumberOfPoints());

    pc.addPoint(3, -4);
    assertEquals(4, pc.getNumberOfPoints());

    Point[] ps = pc.getPointsInQuadrantI();
    assertEquals(1, ps.length);
    assertTrue(ps[0].getX() == 3 && ps[0].getY() == 4);
}
```




find the left-most "null slot" in the array.
PC
PointCollector
points
nOP
not returnable for method
qs
null
null
null
null
Count & I
glpoints
Point
x 3
y 4
Point
x -3
y 4
Point
x 3
y -4
Point
x 3
y -4
0
1
2
3
99

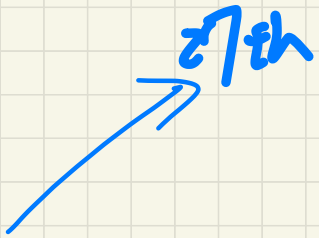
Lecture 5 - Sep. 19

Review of OOP

Anonymous Objects
Reference to this
Static Variables

Announcements/Reminders

- Lab0P2 due tomorrow (Friday) at 12 **noon**!
- Lab1 to be released after Lab0P2 is due.
- In-Lab demo on the **Programming Pattern**
- Today's office hour to be re-scheduled
- **Mockup Programming Test** next Fri (5pm or 6pm)



Anonymous Objects


```
1  double square(double x) {
2    double sqr = x * x;
3    return sqr; }
```

anonymous

```
1  double square(double x) {
2    return x * x; }
```

```
1  Person getP(String n) {
2    Person p = new Person(n);
3    return p; }
```

anonymous

```
1  Person getP(String n) {
2    return new Person(n); }
```

object expression (without ref. a variable)

Exercise

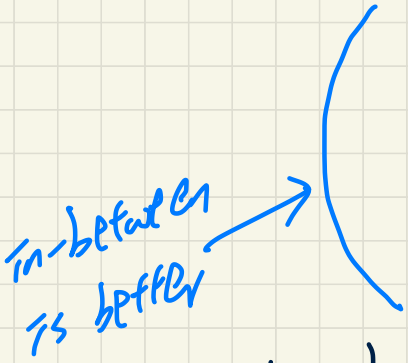
```
class Member {
  private Order[] orders;
  private int noo;
  /* constructor ommitted */
  public void addOrder(Order o) {
    this.orders[this.noo] = o;
    this.noo++;
  }
  public void addOrder(String n, double p, double q) {
    [redacted]
  }
}
```

overloading

```
// Order o = new Order(n, p, q);
this.orders[noo] = o;
noo ++;
this.addOrder(o);
// this.addOrder( );
```

(1) No anonymous objects.

↳ many local, intermediate variables



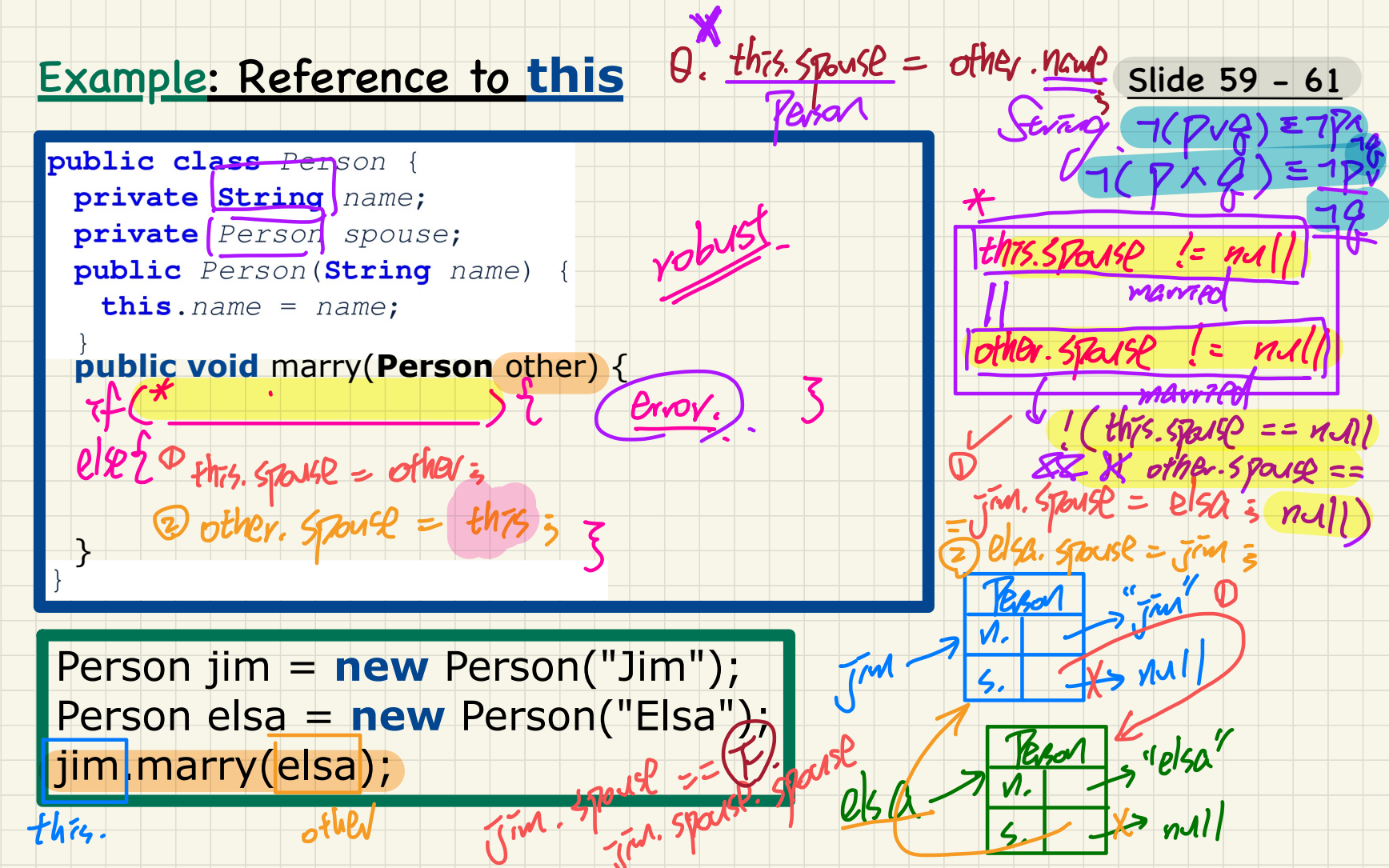
(2) No intermediate variable

↳ long expressions

Example: Reference to this


```
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
  public void marry(Person other) {
    if(*                                ) {
    else{ ① this.spouse = other;
          ② other.spouse = this;
    }
}
```

```
Person jim = new Person("Jim");
Person elsa = new Person("Elsa");
jim.marry(elsa);
```



Exercise

```
void marry(Person other) {
    if(                                    ) {
        this.spouse = other;
        other.spouse = this;
    }
    else { error }
}
```

Managing Account IDs: Manual


```
public class Account {
  private int id;
  private String owner;
  public int getID() { return this.id; }
  public Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

Principles

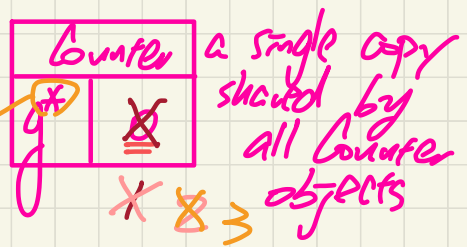
different Account objects should have diff. ids.

```
class AccountTester {
  Account acc1 = new Account(1, "Jim");
  Account acc2 = new Account(2, "Jeremy");
  System.out.println(acc1.getID() != acc2.getID());
}
```

too much burden on users of Account class

Declaring Global Variables among Objects

Ref. to static var.: ClassName.varId




```
public class Counter {
    private int l;
    static int g = 0;

    public Counter() {
        this.l = 0;
    }

    public int getLocal() {
        return this.l;
    }

    public void incrementLocal() {
        this.l ++;
    }

    public void incrementGlobal() {
        g ++;
    }
}
```

non-static var. → each Counter has its own copy of "l"

In const, only init. non-static att.

static var. → a single copy shared by All Counter objects

c1 c2

```
public class CounterTester {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        System.out.println("c1's local: " + c1.getLocal());
        System.out.println("c2's local: " + c2.getLocal());
        System.out.println("Global accessed via c1: " + c1.g);
        System.out.println("Global accessed via c2: " + c2.g);
        System.out.println("Global accessed via Counter: " + Counter.g);

        c1.incrementLocal();

        c2.incrementLocal();

        c1.incrementGlobal();

        c2.incrementGlobal();

        Counter.g = Counter.g + 1; // Counter.global ++;
    }
}
```

c1 c2 1 1

0 0

Class name

static var.

Counter.g 1
c1.g 1
c2.g 1

Counter.g 2
c1.g 2 c2.g 2

Lecture 6 - Sep. 24

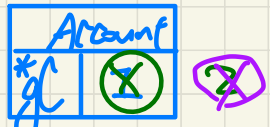
Review of OOP, Exceptions

Static Variables: Common Errors
Caller vs. Callee
Tracing Method Call Chain via Call Stack

Announcements/Reminders

- Lab1 released
- Mockup Programming Test this Fri (5pm or 6pm)
- Guides for WrittenTest1 and ProgTest1 to be released
- Reminder of rules for class attendance checks

Managing Account IDs: Automatic




```
class Account {
  private static int globalCounter = 1;
  private int id; String owner;
  public Account(String owner) {
    this.id = globalCounter;
    globalCounter++;
    this.owner = owner; } }
```

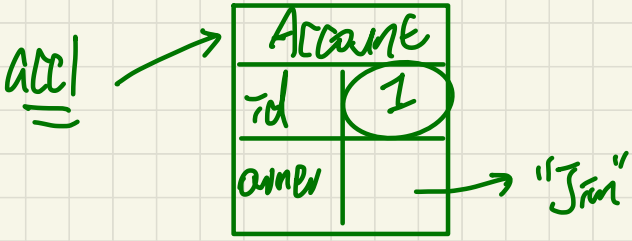
shared. init

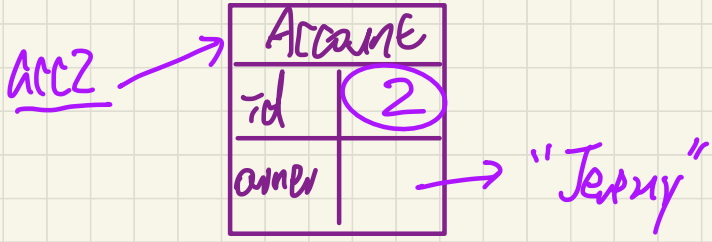
parameters not including id.

local — each obj of type Account has its own copy

Q. What if "id" is also static?

```
class AccountTester {
  Account acc1 = new Account("Jim");
  Account acc2 = new Account("Jeremy");
  System.out.println(acc1.getID() != acc2.getID()); }
```





Alt.

```
class Account {
    static int gC = 1;
    static int id;
    public Account() {
        id = gC;
        gC++;
    }
}
```

```
Account a1 = new Account();
Account a2 = new Account();
```

Account	
gC	~~1~~ ~~2~~ 3
id	~~1~~ ~~2~~

a2 → Account

a1 → Account

① a1.id 1

② a2.id 2

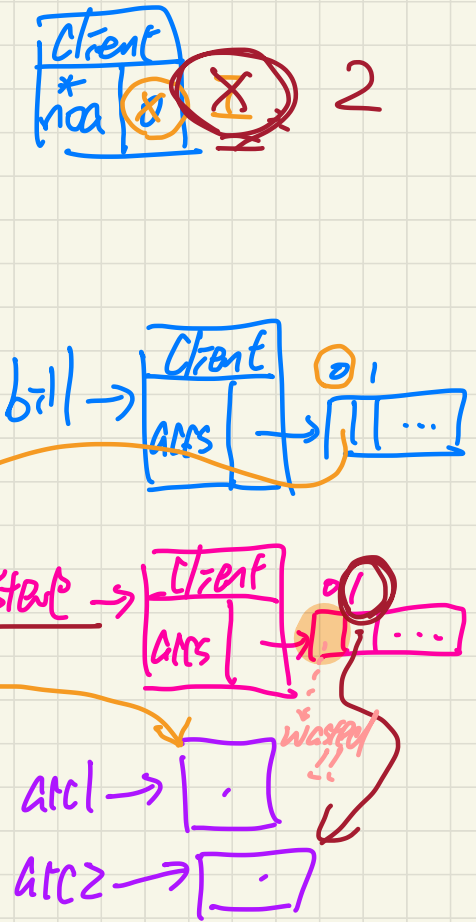
a1.id 2

↳ wrong!

Misuse of Static Variables


```
public class Client {
  private Account[] accounts;
  private static int numberOfAccounts = 0;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  } }
```

```
public class ClientTester {
  Client bill = new Client("Bill");
  Client steve = new Client("Steve");
  Account acc1 = new Account();
  Account acc2 = new Account();
  bill.addAccount(acc1);
    /* [redacted] */
  steve.addAccount(acc2);
    /* [redacted] */
}
```



Use of Static Variables: Common Error

Slides 80 - 82

```
Bank cibc = new Bank();
cibc.branchName();
```

```
1  public class Bank {
2      private string branchName;
3      public String getBrachName() { return this.branchName; }
4      private static int nextAccountNumber = 0;
5      public static String getInfo() {
6          nextAccountNumber++;
7          return this.branchName + nextAccountNumber;
8      }
9  }
```

non-static

static

Bank.getInfo()

class name

static method

Compilation error!

To call this static method:

Bank.getInfo()

should be a C.O.

Bank.branchName()

non-static

Fix 1:

```
1  public class Bank {
2     private string branchName;
3     public String getBrachName() { return this.branchName; }
4     private static int nextAccountNumber = 0;
5     public static String getInfo() {
6        nextAccountNumber++;
7        return this.branchName + nextAccountNumber;
8     }
9  }
```

Fix 2:

```
1  public class Bank {
2     private string branchName;
3     public String getBrachName() { return this.branchName; }
4     private static int nextAccountNumber = 0;
5     public static String getInfo() {
6        nextAccountNumber++;
7        return this.branchName + nextAccountNumber;
8     }
9  }
```

static

Is there a shared branch name by all Banks? No!!

```
public class MyClass {
    public static void main ( ... ) {

    }
}
```

main

↓

Entry point of execution

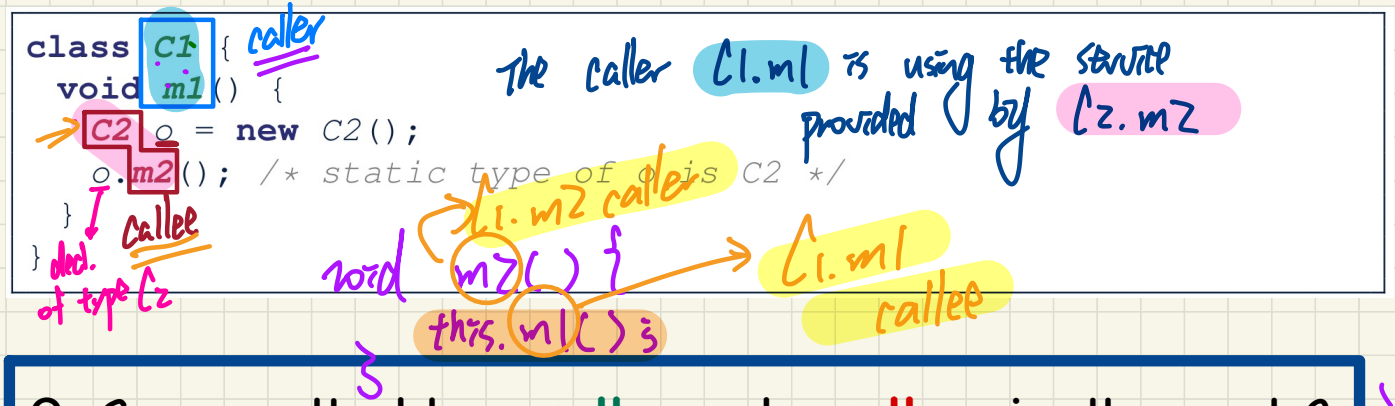
(MyClass.main())

↓

when exec. starts; no object creation is necessary

Caller vs. Callee

- **caller** is the **client** using the service provided by another method.
- **callee** is the **supplier** providing the service to another method.



Q: Can a method be a caller and a callee simultaneously? YES!

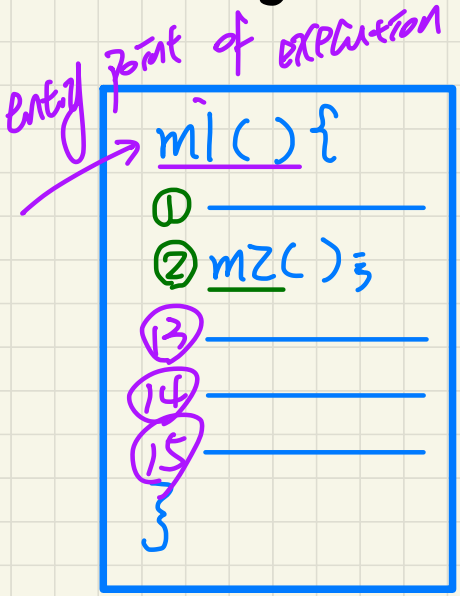
(1) Make C1.m1 a callee as well

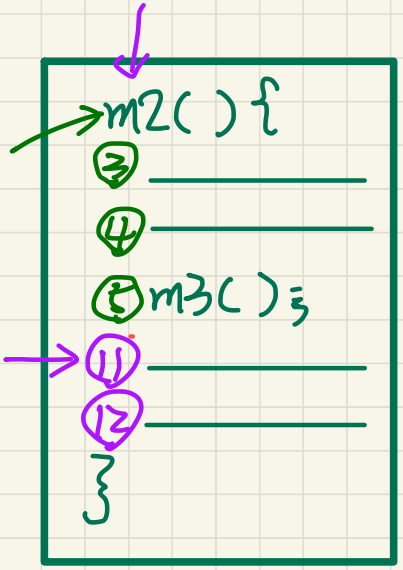
(2) Exercise. Make C2.m2 a caller.

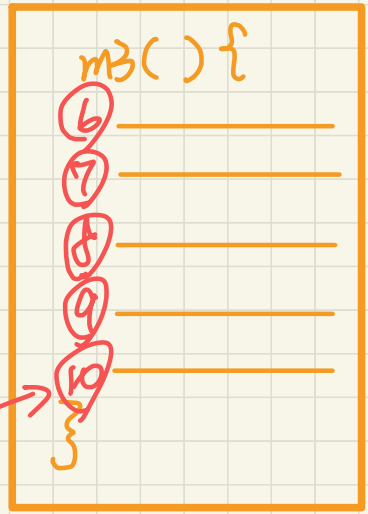
```
class C3 {
  void m3() {
    C1 o = new C1(); o.m1();
  }
}
```

C1.m1 is now a caller.

Visualizing a Call Chain using a Stack

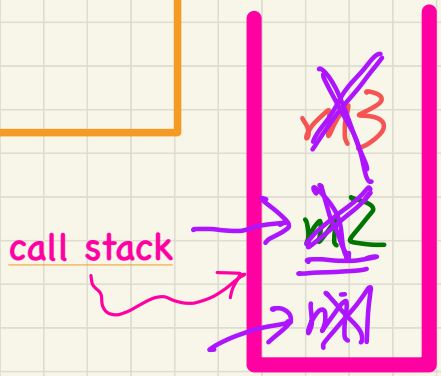






push / add

pop / remove



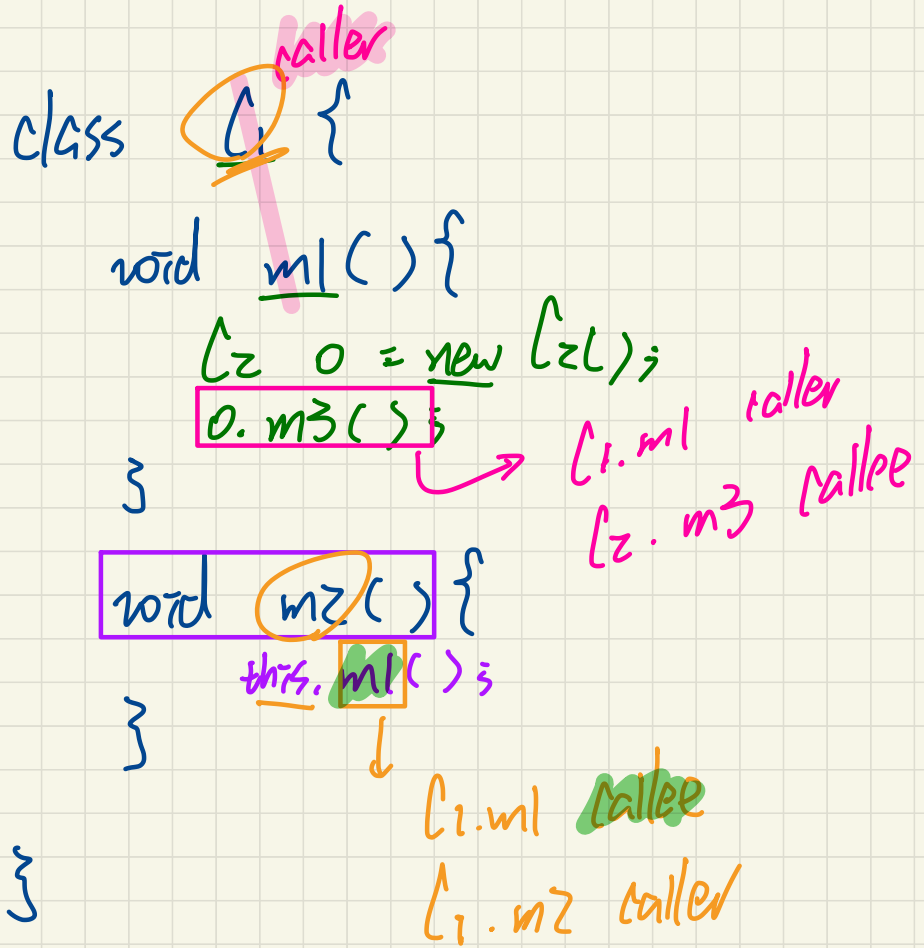
Lecture 7 - Sep. 26

Review of OOP, Exceptions

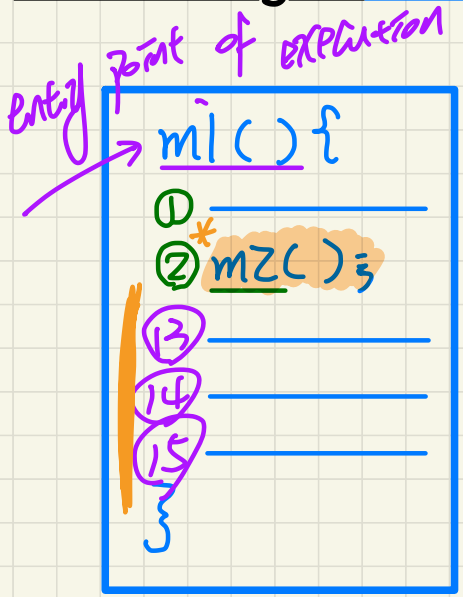
Chronological Order of Method Calls
How Exception Disrupts Execution Flow
Catch-or-Specify Requirement
Example: To Handle or Not to Handle (V1)

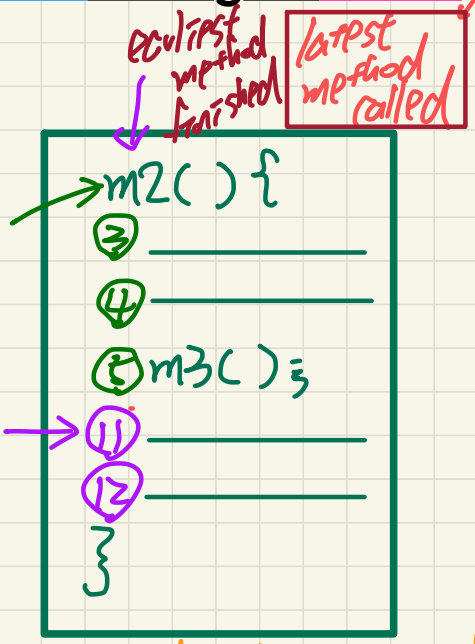
Announcements/Reminders

- Lab1 released
- In-Lab demo: Incremental Development for Lab1
- Mockup Programming Test tomorrow (5pm or 6pm)
- Guides for WrittenTest1 and ProgTest1 released
- WrittenTest1 review (Zoom) on Monday, time TBA


caller
class C1 {
void m1() {
C2 o = new C2();
o.m3();
}
C1.m1 caller
C2.m3 callee
void m2() {
this.m1();
}
C1.m1 callee
C1.m2 caller
}

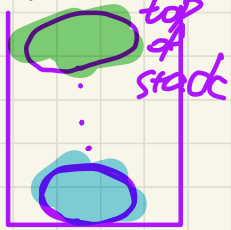
Visualizing a Call Chain using a Stack





chronological order of method calls: m1, m2, m3

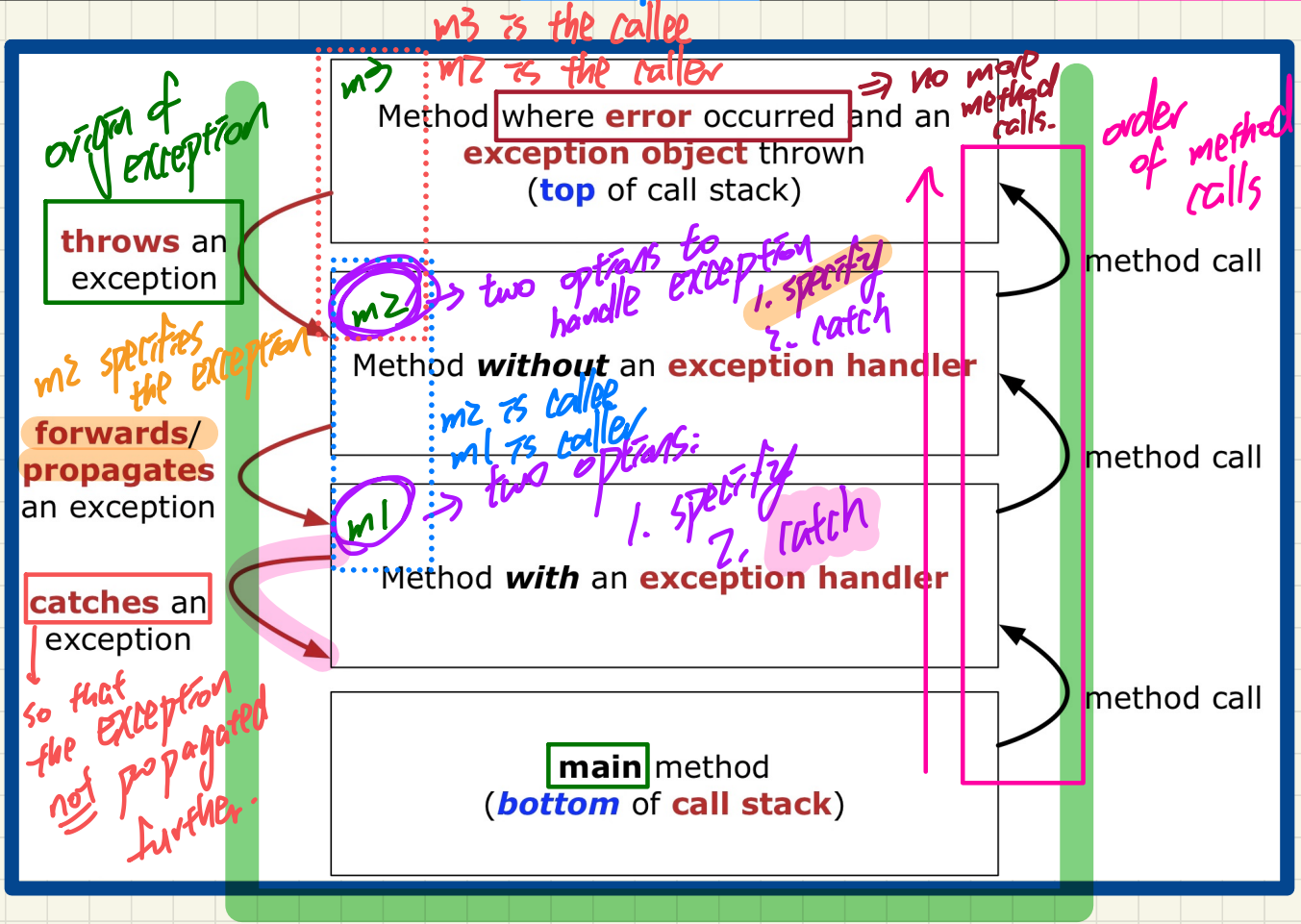
latest method finished



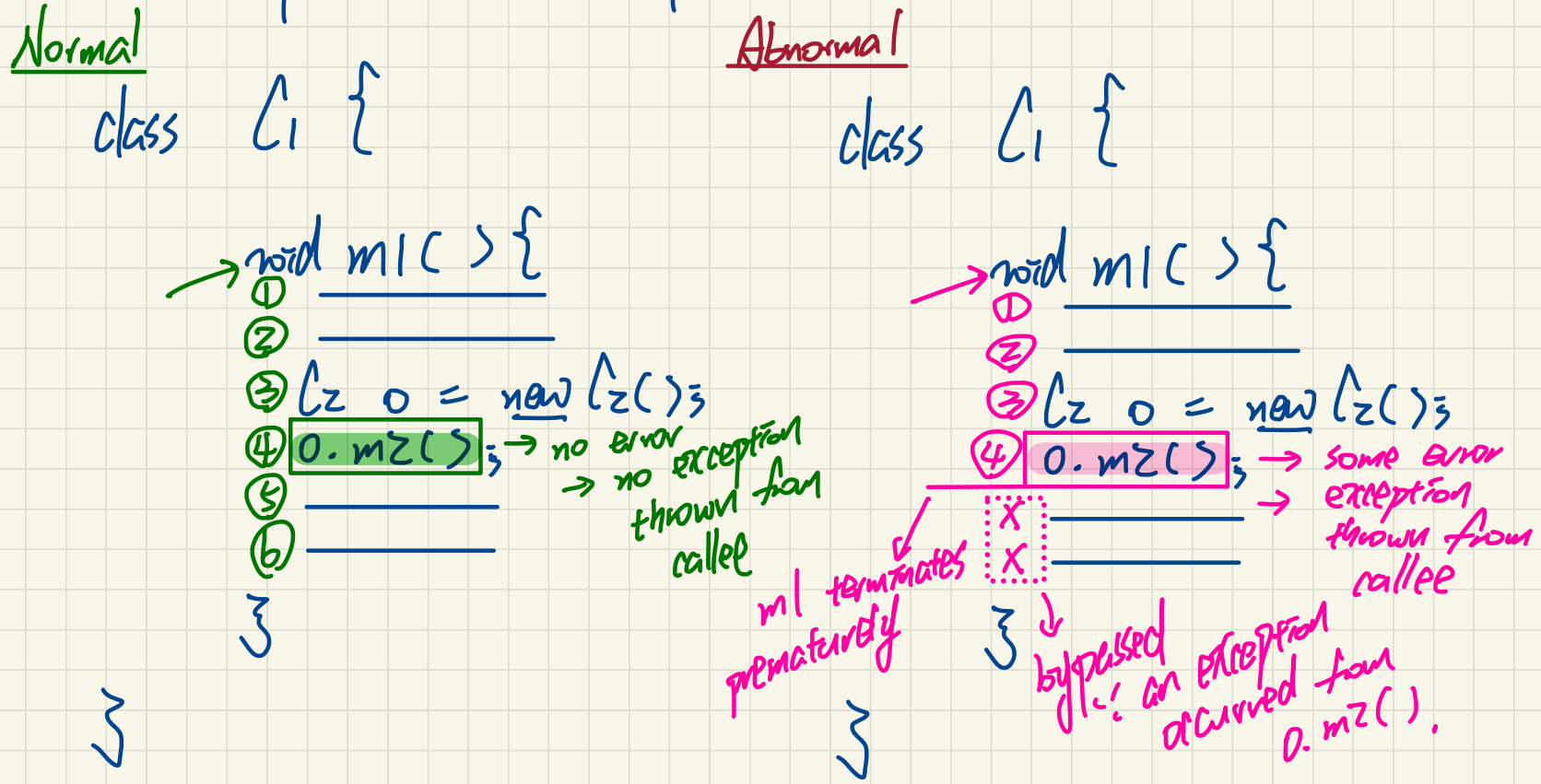
call stack

At *, m2 is called.
(1) execution of m2 is now prioritized.
(2) remainder of m1 will only be resumed after m2 is finished.

What to Do When an Exception is Thrown: Call Stack



Exceptions: Disrupt the Normal Flow of Exec.



Example: To Handle or **Not** To Handle?

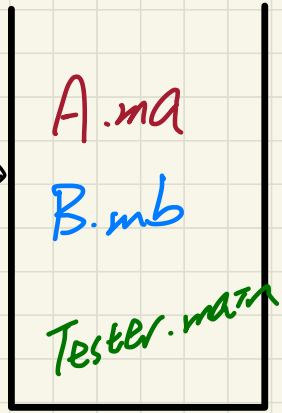
```
class A {
  ma(int i) {
    if(i < 0) { /* Error */ }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) {
    A oa = new A();
    oa.ma(i); /* Error occurs if i < 0 */
  } }
```

Version 1:
Handle it in `B.mb`
Version 2:
Pass it from `B.mb` and handle it in `Tester.main`
Version 3:
Pass it from `B.mb`, then from `Tester.main`, then throw it to the console.

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Where can the error be handled? */
  } }
```

call stack




```
class NegValException extends Exception {
  NegValException(String s) { super(s); }
}
```

Written Test 1

Review Q&A

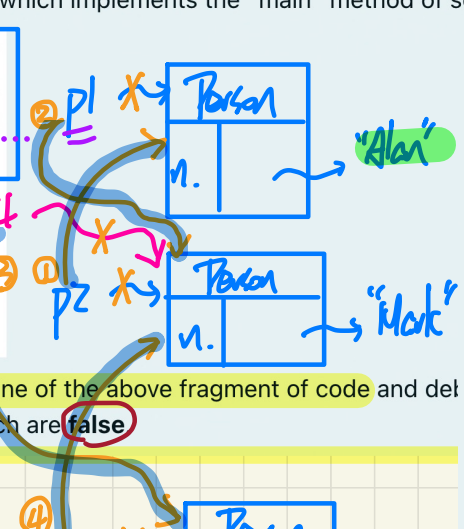
Aliasing
Call Stack vs. Catch-or-Specify Req.
NullPointerException
ArrayIndexOutOfBoundsException

Practice Written Test 1

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!");
```

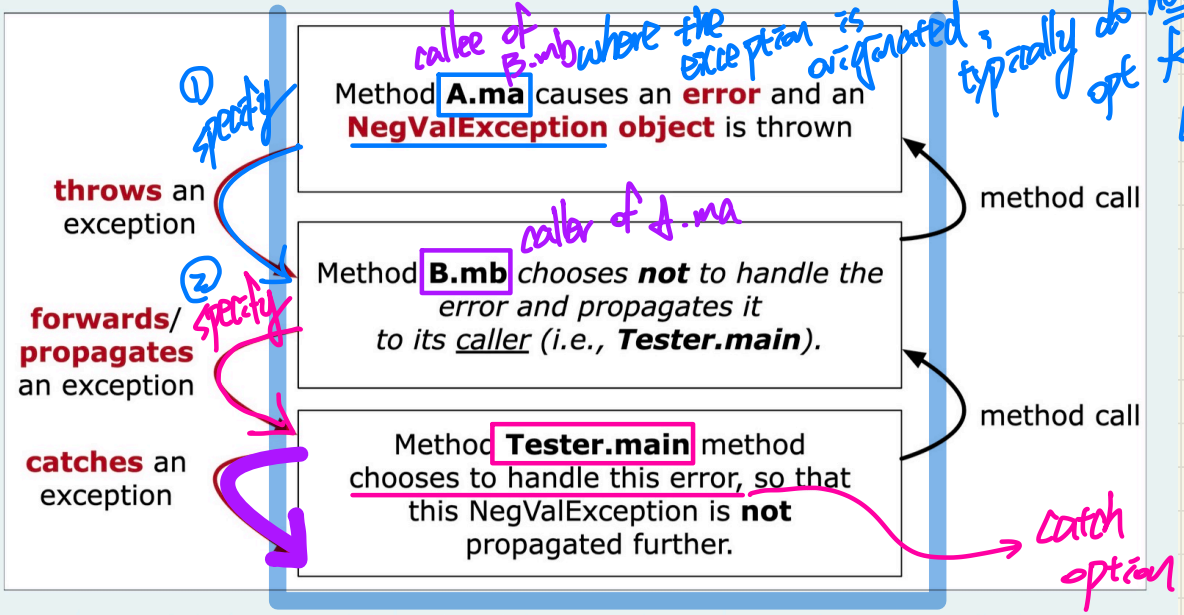


Now say we place a breakpoint at the last line of the above fragment of code and del
following list of statements, choose **all** which are **false**

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

Q6

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



Atypical

```
ma() {
  if(...){
  try{
    throw new...
  } catch(...){...}
  else{
    ...
  }
}
```

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.

2.

Q7

m: the first method from top of the stack that opts for the catch option

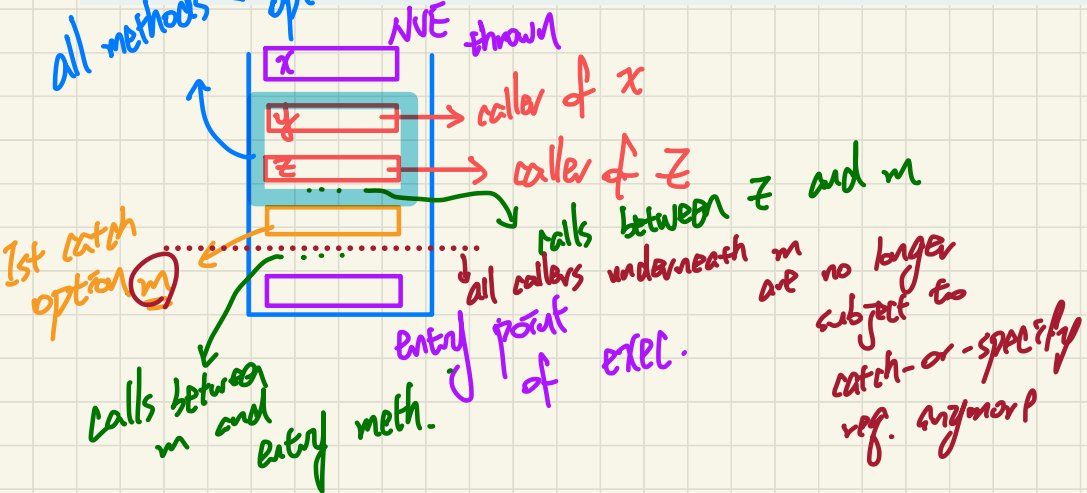
thrown by method x

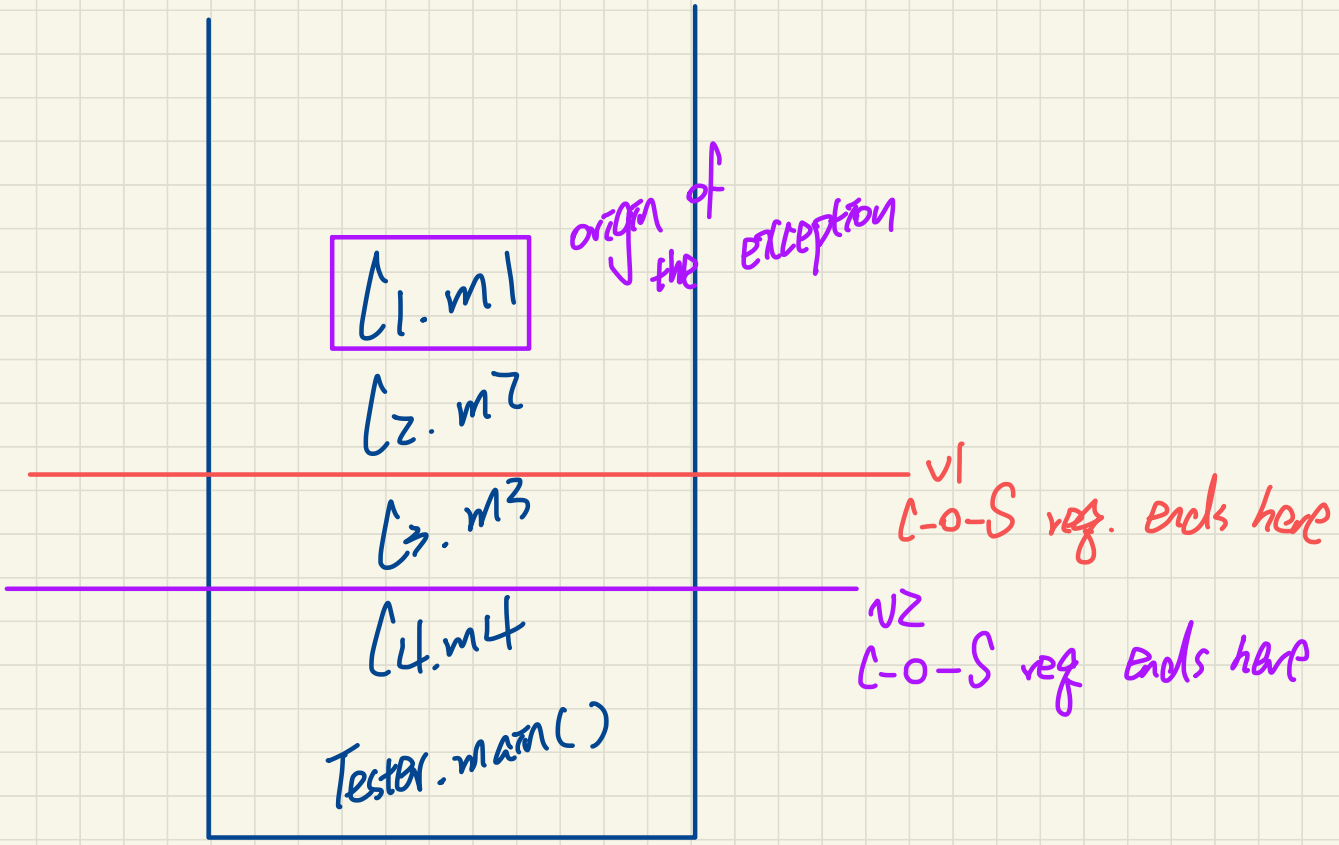
At a runtime call stack, if a method implements a try-catch block to handle a *NegValException* that may be thrown from its **callee**, then this method's **caller** is still obliged to either catch or specify that *NegValException*.

Select one:

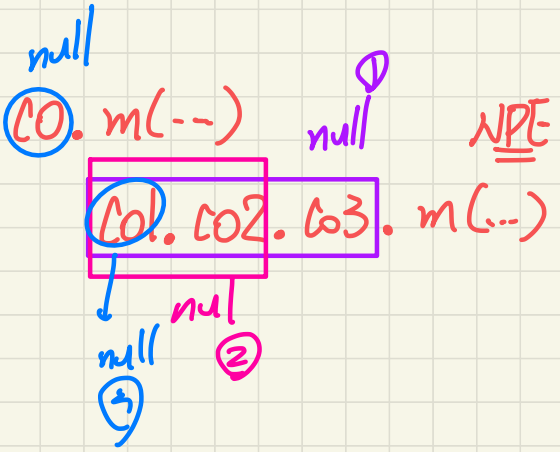
- True
- False

all methods (callers) opt for the specify option




C1.m1
origin of the exception
C2.m2
C3.m3
v1
C-o-S req. ends here
C4.m4
v2
C-o-S req ends here
Tester.main()

NullPointerException



ArrayIndexOutOfBoundsException

a[exp]

↓

AIOBE if: $0 \le exp \le a.length - 1$

Assertion Failure

assertTrue(...);

Lecture 8 - Oct. 1

Exceptions

CoS Req.: Exec. Flow vs. Call Stack
To Handle or Not to Handle (V2 - V4)
More Examples on Exceptions
Exceptions vs. Console Tester

Announcements/Reminders

- Written Test 1 tomorrow (Wednesday)
- WrittenTest1 review session recording released
- Lab1 due this Friday
- Mockup Programming Test grading tests released
- Mockup Programming Test feedback to be released

Catch-or-Specify Requirement: Execution Flows

Normal Flow of Execution

```
... /* before, ouside try-catch block */
try {
  o.m(...); /* may throw SomeException */
  ... /* rest of try-block */
}
catch (SomeException se) {
  ... /* rest of catch-block */
}
... /* after, ouside try-catch block */
```

excep not occurred

X bypassed

When the exception does not occur

Abnormal Flow of Execution

```
... /* before, ouside try-catch block */
try {
  o.m(...) /* may throw SomeException */
  ... /* rest of try-block */
}
catch (SomeException se) {
  ... /* rest of catch-block */
}
... /* after, ouside try-catch block */
```

excep. occurred

X bypassed

When the exception occurs

Catch-or-Specify Requirement: Call Stack

Q1. Origin of Exception: $C1.m1$

Q2. Methods subject to CoS Req:

$C1.m1 \sim C_{i-1}.m_{i-1}$

Q3. Methods free from CoS Req:

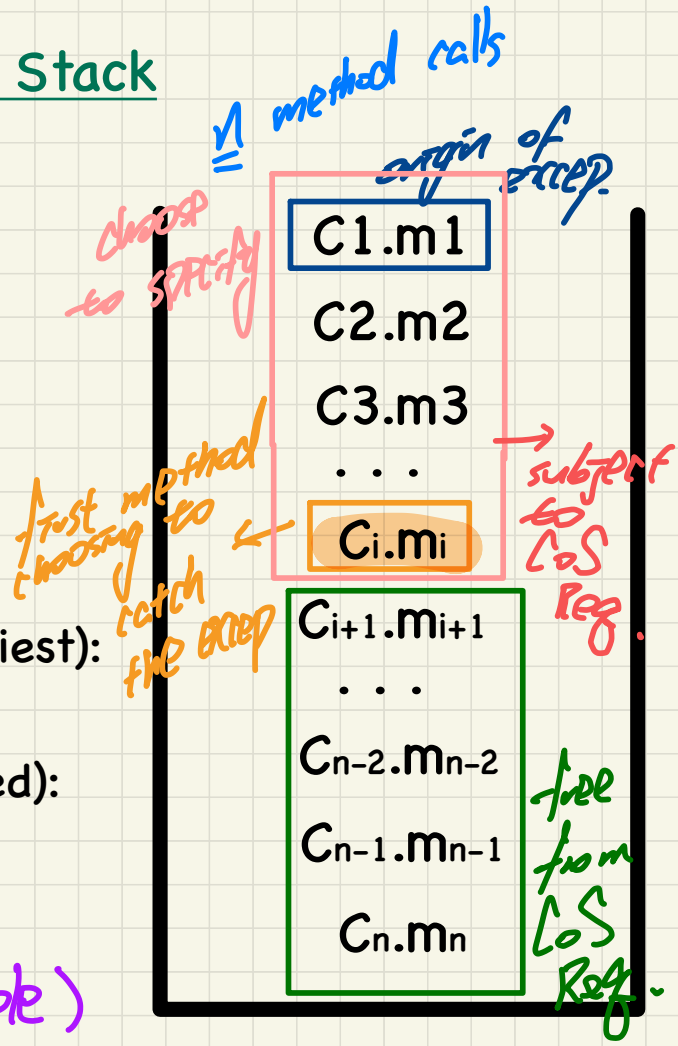
$C_{i+1}.m_{i+1} \sim Cn.mn$

Q4. Extreme Case 1 (exception handled earliest):

$C2.m2$

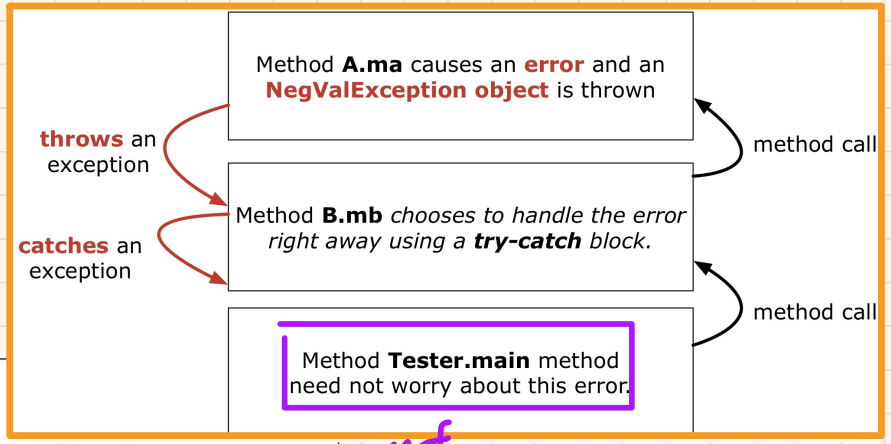
Q5. Extreme Case 2 (exception never handled):

All methods choose to specify (excep. thrown to console)



Version 1:

Handle the Exception in B.mb




```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

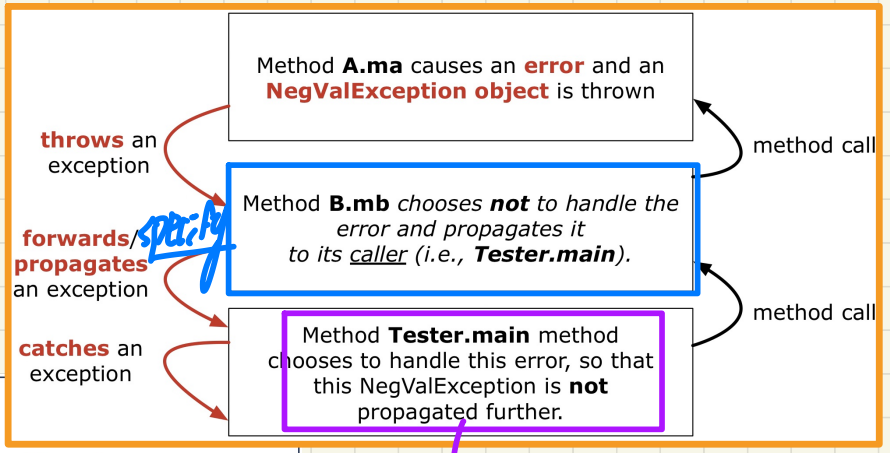
```
class B {
  mb(int i) {
    A oa = new A();
    try { oa.ma(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Error, if any, would have been handled in B.mb.
  } }
```

not subject to CoS Req.

Version 2:

Handle the Exception in Tester.main



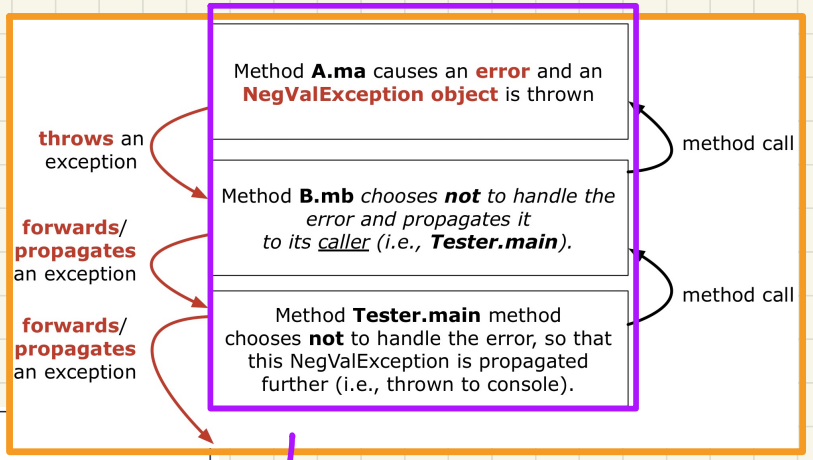

```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    try { ob.mb(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

Version 3:

Handle in Neither Classes on Call Stack



all subject to LoS req.

```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

```
class Tester {
  public static void main(String[] args) throws NegValException {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i);
  } }
```

Error Handling via Exceptions: Circles (Version 1)

```
public class InvalidRadiusException extends Exception {
  public InvalidRadiusException(String s) {
    super(s);
  }
}
```

Test Case 1:

User enters 10

Test Case 2:

User enters -5

```
class Circle {
  double radius;
  Circle() { /* radius defaults to 0 */ }
  void setRadius(double r) throws InvalidRadiusException {
    if (r < 0) {
      throw new InvalidRadiusException("Negative radius.");
    }
    else { radius = r; }
  }
  double getArea() { return radius * radius * 3.14; }
}
```

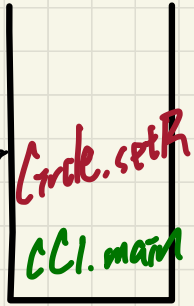
caller	callee

```
class CircleCalculator1 {
  public static void main(String[] args) {
    Circle c = new Circle();
    try {
      c.setRadius(10);
      double area = c.getArea();
      System.out.println("Area: " + area);
    }
    catch(InvalidRadiusException e) {
      System.out.println(e);
    }
  }
}
```

Caller?

Callee?

call stack



Error Handling via Exceptions: Circles (Version 2)

```
public class InvalidRadiusException extends Exception {
  public InvalidRadiusException(String s) {
    super(s);
  }
}
```

Test Case:

User enters -5

Then user enters 10

Exercise!

```
class Circle {
  double radius;
  Circle() { /* radius defaults to 0 */ }
  void setRadius(double r) throws InvalidRadiusException {
    if (r < 0) {
      throw new InvalidRadiusException("Negative radius.");
    }
    else { radius = r; }
  }
  double getArea() { return radius * radius * 3.14; }
}
```

```
public class CircleCalculator2 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean inputRadiusIsValid = false;
    while (!inputRadiusIsValid) {
      System.out.println("Enter a radius:");
      double r = input.nextDouble();
      Circle c = new Circle();
      try { c.setRadius(r);
            inputRadiusIsValid = true;
            System.out.print("Circle with radius " + r);
            System.out.println(" has area: "+ c.getArea()); }
      catch(InvalidRadiusException e) { print("Try again!"); }
    } } }
```

keep prompting for a non-neg. radius

may throw IRE

except. not occurred ⇒ ready to exit loop

except. occurred!

Error Handling via Exceptions: Banks

Test Case:
User enters -5000000

```
public class InvalidTransactionException extends Exception {
  public InvalidTransactionException(String s) {
    super(s);
  }
}
```

```
class Account {
  int id; double balance;
  Account() { /* balance defaults to 0 */ }
  void withdraw(double a) throws InvalidTransactionException {
    if (a < 0 || balance - a < 0) {
      throw new InvalidTransactionException("Invalid withdraw."); }
    else { balance -= a; }
  }
}
```

specify

-5M

```
class Bank {
  Account[] accounts; int numberOfAccounts;
  Account(int id) { ... }
  void withdraw(int id, double a)
      throws InvalidTransactionException {
    for(int i = 0; i < numberOfAccounts; i ++) {
      if(accounts[i].id == id) {
        accounts[i].withdraw(a);
      }
    } /* end for */ }
```

specify

-5M

Account

```
class BankApplication {
  pubic static void main(String[] args) {
    Bank b = new Bank();
    Account acc1 = new Account(23);
    b.addAccount(acc1);
    Scanner input = new Scanner(System.in);
    double a = input.nextDouble();
    try {
      b.withdraw(23, a);
      System.out.println(acc1.balance); }
    catch (InvalidTransactionException e) {
      System.out.println(e); } } }
```

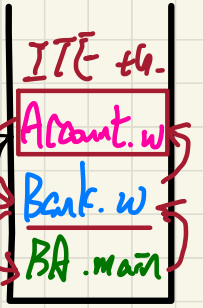
-5M -5M

catch option.

b → Bank { noa: 1, accounts } → 0 1 ... b.accounts.lenth - 1

acc1 → Account { id: 23, bal.: 0 }

call stack



More Example: Multiple Catch Blocks

```
1  double r = ...;
2  double a = ...;
3  try{
4    Bank b = new Bank();
5    b.addAccount(new Account(34));
6    b.deposit(34, 100);
7    b.withdraw(34, a);
8    Circle c = new Circle();
9    c.setRadius(r);
10   System.out.println(r.getArea());
   }
   catch(NegativeRadiusException e) {
     System.out.println(r + " is not a valid radius value.");
     e.printStackTrace();
   }
   catch(InvalidTransactionException e) {
     System.out.println(r + " is not a valid transaction value.");
     e.printStackTrace();
   }
```

may throw ITE

may throw NRE

Test Case 1:

a: -5000000

r: 23

Test Case 2:

a: 100

r: -5

execute!

More Example: Parsing Strings as Integers

```
Scanner input = new Scanner(System.in);
boolean validInteger = false;
while (!validInteger) {
  System.out.println("Enter an integer:");
  String userInput = input.nextLine();
  try {
    int userInteger = Integer.parseInt(userInput);
    validInteger = true;
  }
  catch(NumberFormatException e) {
    System.out.println(userInput + " is not a valid integer.");
    /* validInteger remains false */
  }
}
```

Test Case:

User Enters: "twenty-three"

User Then Enters: 23

"twenty-three" → trigger NFE

"23" → does not trigger NFE

may throw NFE

Error Handling via Console Messages: Circles

```
1  class Circle {
2    double radius;
3    Circle() { /* radius defaults to 0 */ }
4    void setRadius(double r) {
5      if ( r < 0 ) { System.out.println( "Invalid radius." ); }
6      else { radius = r; }
7    }
8    double getArea() { return radius * radius * 3.14; }
9  }
```

a bad alternative to throwing an excep.

no CoS req. enforced

any more

Caller?

Callee?

call stack

C.sR

CC.main

```
1  class CircleCalculator {
2    public static void main(String[] args) {
3      Circle c = new Circle();
4      c.setRadius(-10);
5      double area = c.getArea();
6      System.out.println("Area: " + area);
7
8  }
```

no excep. thrown (only a msg printed to console)

still executed despite Error.

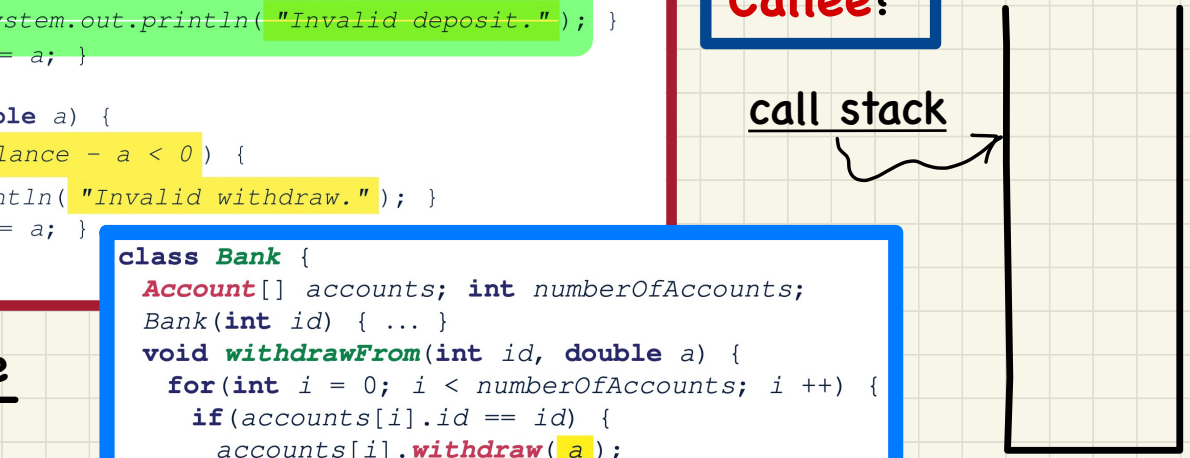
Error Handling via Console Messages: Banks

```
class Account {
  int id; double balance;
  Account(int id) { this.id = id; /* balance defaults to 0 */ }
  void deposit(double a) {
    if ( a < 0 ) { System.out.println( "Invalid deposit." ); }
    else { balance += a; }
  }
  void withdraw(double a) {
    if ( a < 0 || balance - a < 0 ) {
      System.out.println( "Invalid withdraw." ); }
    else { balance -= a; }
  }
}
```

Caller?

Callee?

call stack



```
class Bank {
  Account[] accounts; int numberOfAccounts;
  Bank(int id) { ... }
  void withdrawFrom(int id, double a) {
    for(int i = 0; i < numberOfAccounts; i ++) {
      if(accounts[i].id == id) {
        accounts[i].withdraw( a );
      }
    } /
  } /*
}
```

```
class BankApplication {
  pubic static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    Bank b = new Bank(); Account acc1 = new Account(23);
    b.addAccount(acc1);
    double a = input.nextDouble();
    b.withdrawFrom(23, a );
    System.out.println("Transaction Completed.");
}
```

context	caller	callee

Lecture 9 - Oct. 3

TDD with JUnit

Deriving Test Cases
JUnit Test Method vs. Method Under Test
Regression Testing
JUnit Test: An Exception Not Expected

Announcements/Reminders

- **ProgTest1** this Wednesday
- Released:
  - ✓ **Written Test 1** result and answers
  - ✓ **Lab2** instructions
  - ✓ **Lab1** solutions (PDF & video)
  - ✓ ProgTest1 Review (PracticeTest1 solution) recording

Review: Specify-or-Catch Principle

Approach 1 – Specify: Indicate in the method signature that a specific exception might be thrown.

Example 1: Method that throws the exception

```
class C1 {
  void m1(int x) throws ValueTooSmallException {
    if(x < 0) {
      throw new ValueTooSmallException("val " + x);
    }
  }
}
```

origin of excep.

Example 2: Method that calls another which throws the exception

```
class C2 {
  C1 c1;
  void m2(int x) throws ValueTooSmallException {
    c1.m1(x);
  }
}
```

caller of some method that might throw exception.

Review: Specify-or-Catch Principle

Approach 2 – Catch: Handle the thrown exception(s) in a try-catch block.

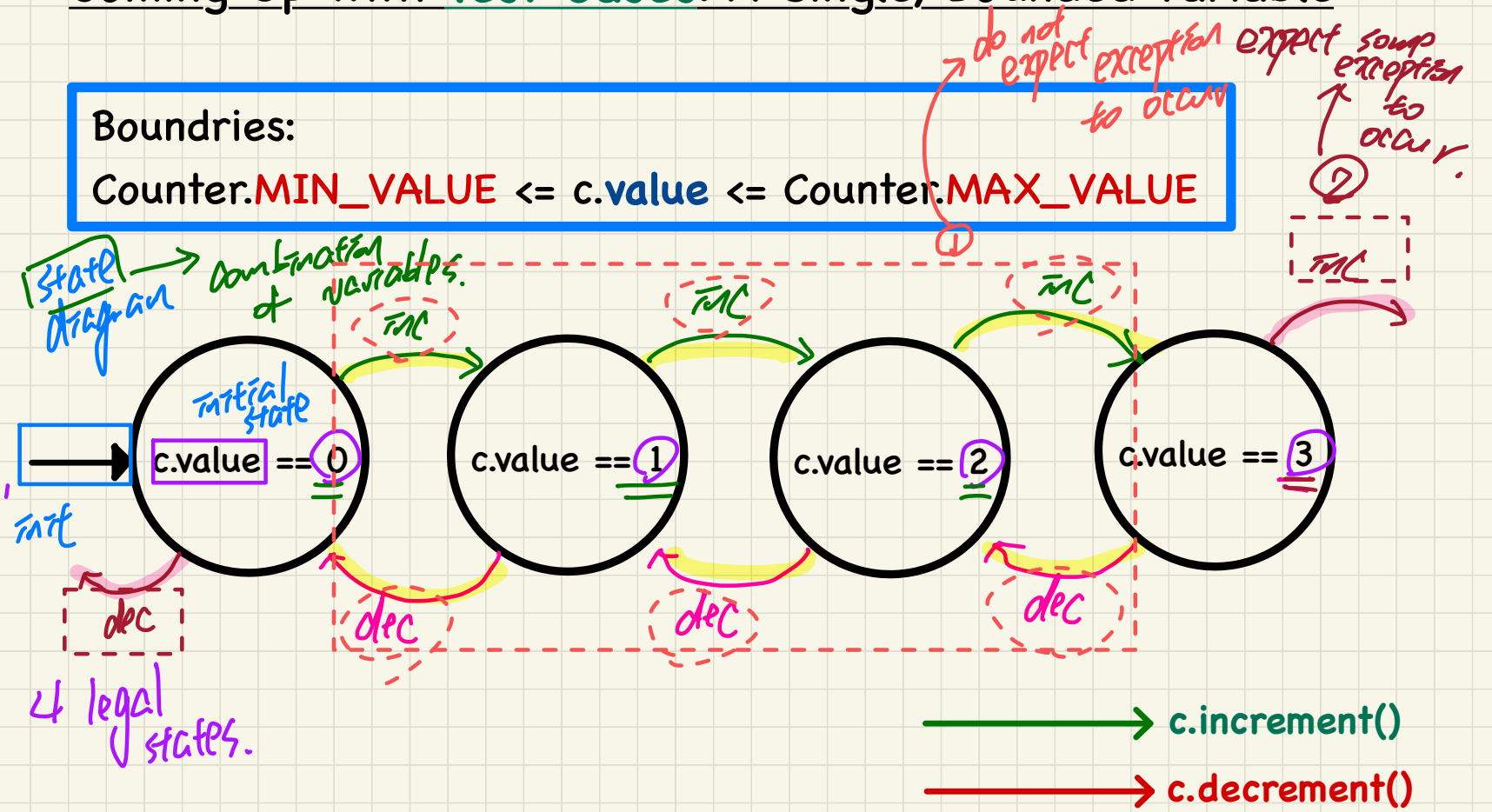
```
class C3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int x = input.nextInt();
    C2 c2 = new c2();
    try {
      c2.m2(x);
    }
    catch(ValueTooSmallException e) { ... }
  }
}
```

caller

Coming Up with Test Cases: A Single, Bounded Variable

Boundries:

Counter.MIN_VALUE <= c.value <= Counter.MAX_VALUE



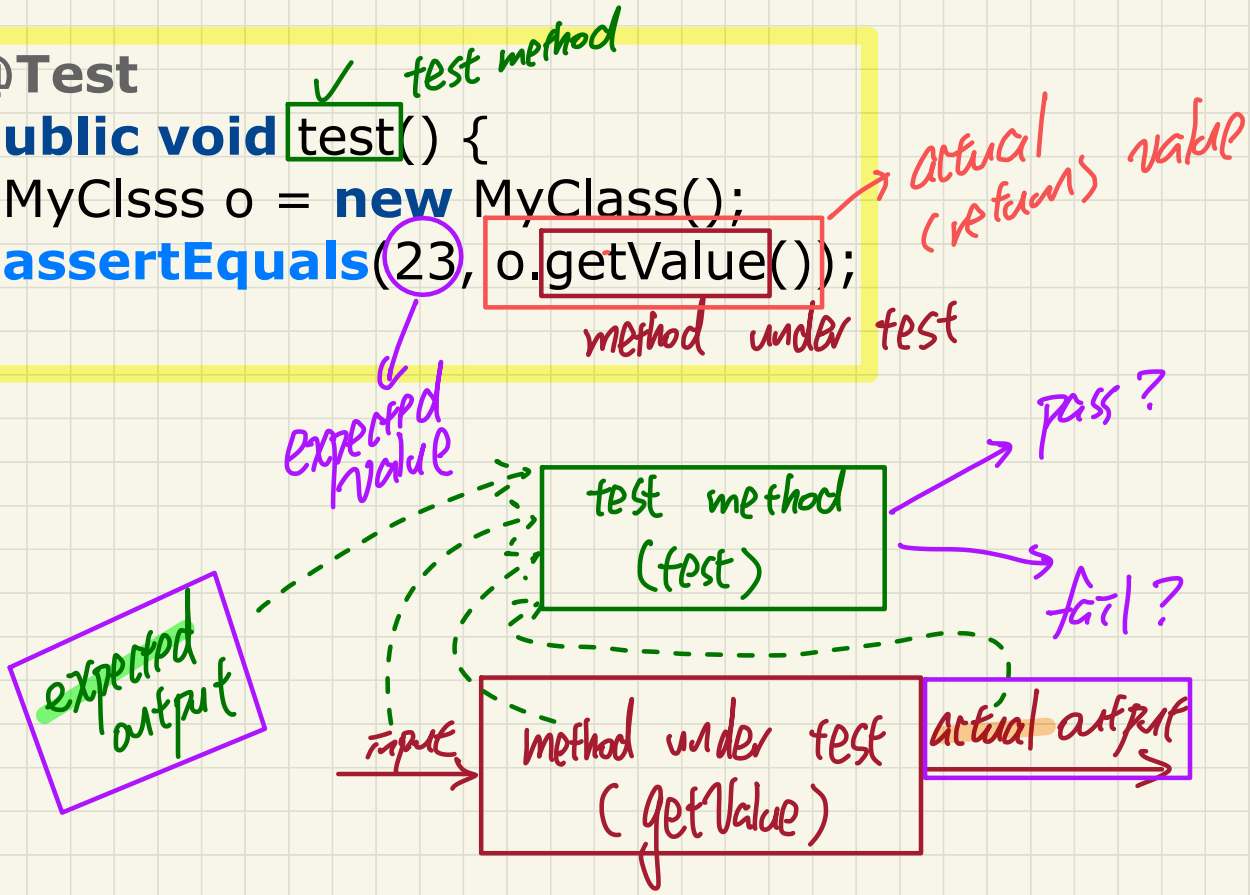
A Class for Bounded Counters

```
public class Counter {
  public final static int MAX_VALUE = 3;
  public final static int MIN_VALUE = 0;
  private int value;
  public Counter() {
    this.value = Counter.MIN_VALUE;
  }
  public int getValue() {
    return value;
  }
  ... /* more later!
```

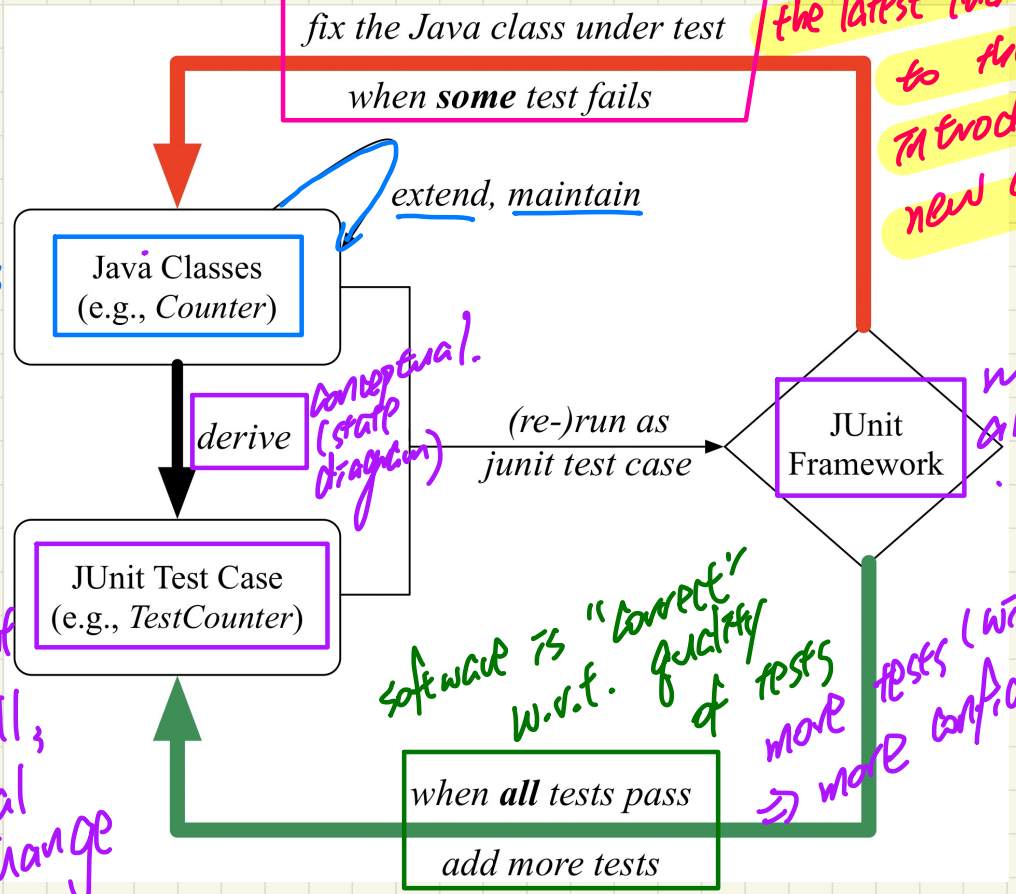
```
/* class Counter */
  public void increment() throws ValueTooLargeException {
    if(value == Counter.MAX_VALUE) {
      throw new ValueTooLargeException("counter value is " + value);
    }
    else { value ++; }
  }

  public void decrement() throws ValueTooSmallException {
    if(value == Counter.MIN_VALUE) {
      throw new ValueTooSmallException("counter value is " + value);
    }
    else { value --; }
  }
}
```

JUnit Test Method vs. Method Under Test

```
@Test
public void test() {
  MyClsss o = new MyClass();
  assertEquals(23, o.getValue());
}
```



Test-Driven Development (TDD): Regression Testing



the latest change made to the devel. code introduces a new error

classes under test

conceptual (state diagram)

run all tests

frequency of re-running all tests is important

Ideal: every small, logical change

software is "correct" w.r.t. quality of tests

more tests (with quality) ⇒ more confidence

A Default Test Case that Fails

fail();

The result of running a test is considered:

- ***Failure*** if either
  - an ***assertion failure*** (e.g., caused by `fail`, `assertTrue`, `assertEquals`) occurs
  - an unexpected (e.g., *`NullPointerException`*, *`ArrayIndexOutOfBoundException`*) ~~or unhandled~~ ***exception*** thrown
- ***Success*** if neither ***assertion failures*** nor (unexpected ~~or unhandled~~) ***exceptions*** occur.

TestCounter.java

```
1  package tests;
2  import static org.junit.Assert.*;
3  import org.junit.Test;
4  public class TestCounter {
5      @Test
6      public void test() {
7          //fail("Not yet implemented");
8      }
9  }
10
```

assertTrue(false);
assertFalse(true);

Q: What is the easiest way to making this test pass?

Examples: JUnit Assertions (1)

Consider the following class:

```
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; }
}
```

Then consider these assertions. Do they ***pass*** or ***fail***?

```
Point p;
assertNull(p);
assertTrue(p == null);
assertFalse(p != null);
assertEquals(3, p.getX());
p = new Point(3, 4);
assertNull(p);
assertTrue(p == null);
assertFalse(p != null);
assertEquals(3, p.getX());
assertTrue(p.getX() == 3 && p.getY() == 4);
```

Examples: JUnit Assertions (2)

Consider the following class:

```
class Circle {
  double radius;
  Circle(double radius) { this.radius = radius; }
  int getArea() { return 3.14 * radius * radius; }
}
```

Then consider these assertions. Do they ***pass*** or ***fail***?

```
Circle c = new Circle(3.4);
assertEquals(36.2984, c.getArea(), 0.01);
```

expected actual tolerance (ε)

=

expected − ε ≤ c.getArea() ≤ expected + ε

JUnit: An Exception Not Expected

```
1  @Test
2  public void testIncAfterCreation() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.increment();
7      assertEquals(1, c.getValue());
8    }
9    catch(ValueTooLargeException e) {
10     /* Exception is not expected to be thrown. */
11     fail("ValueTooLargeException is not expected.");
12   }
13 }
```

may throw VTLE

reaching this line means no exception occurred.

reaching the catch block means VTLE occurred

What if increment is implemented correctly?

v1 of TMC. (no VTLE thrown)

Expected Behaviour:

Calling c.increment()
when c.value is 0 should **not**
trigger a ValueTooLargeException

```
1  @Test
2  public void testIncAfterCreation() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.increment();
7      assertEquals(1, c.getValue());
8    }
9    catch(ValueTooLargeException e) {
10     /* Exception is not expected to be thrown. */
11     fail("ValueTooLargeException is not expected.");
12   }
13 }
```

What if increment is implemented incorrectly?

e.g., It throws VTLE when
c.value < Counter.MAX_VALUE

v2 of TMC (VTLE thrown unexpectedly)

Running JUnit Test 1 on Correct Implementation

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

Correct imp.

prediction of test result: pass

```
1  @Test
2  public void testIncAfterCreation() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.increment();
7      assertEquals(1, c.getValue());
8    }
9    catch(ValueTooLargeException e) {
10     /* Exception is not expected to be thrown. */
11     fail("ValueTooLargeException is not expected.");
12   }
13 }
```

Running JUnit Test 1 on Incorrect Implementation

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

wrong impl. ⇒ test should fail

prediction & test result: fail ✓

```
1  @Test
2  public void testIncAfterCreation() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.increment();
7      assertEquals(1, c.getValue());
8    }
9    catch(ValueTooLargeException e) {
10     /* Exception is not expected to be thrown. */
11     fail("ValueTooLargeException is not expected.");
12   }
13 }
```

Programming Test 1

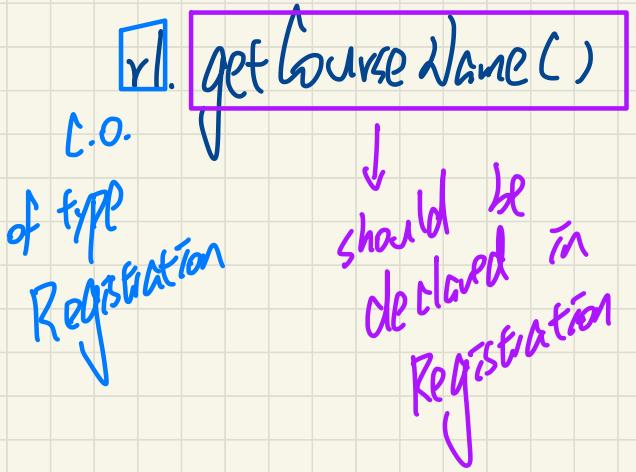
Review Q&A

PracticeTest1 Solution

Announcements/Reminders

Released:

- Lab1 solutions (a walkthrough tutorial video)
- Grading Tests of Lab1 and PracticeTest1


r1.
getCourseName()
C.O.
of type
Registration
should be
declared in
Registration

Tip:

to avoid compilation errors penalty,

around 15 minutes before the test ends,

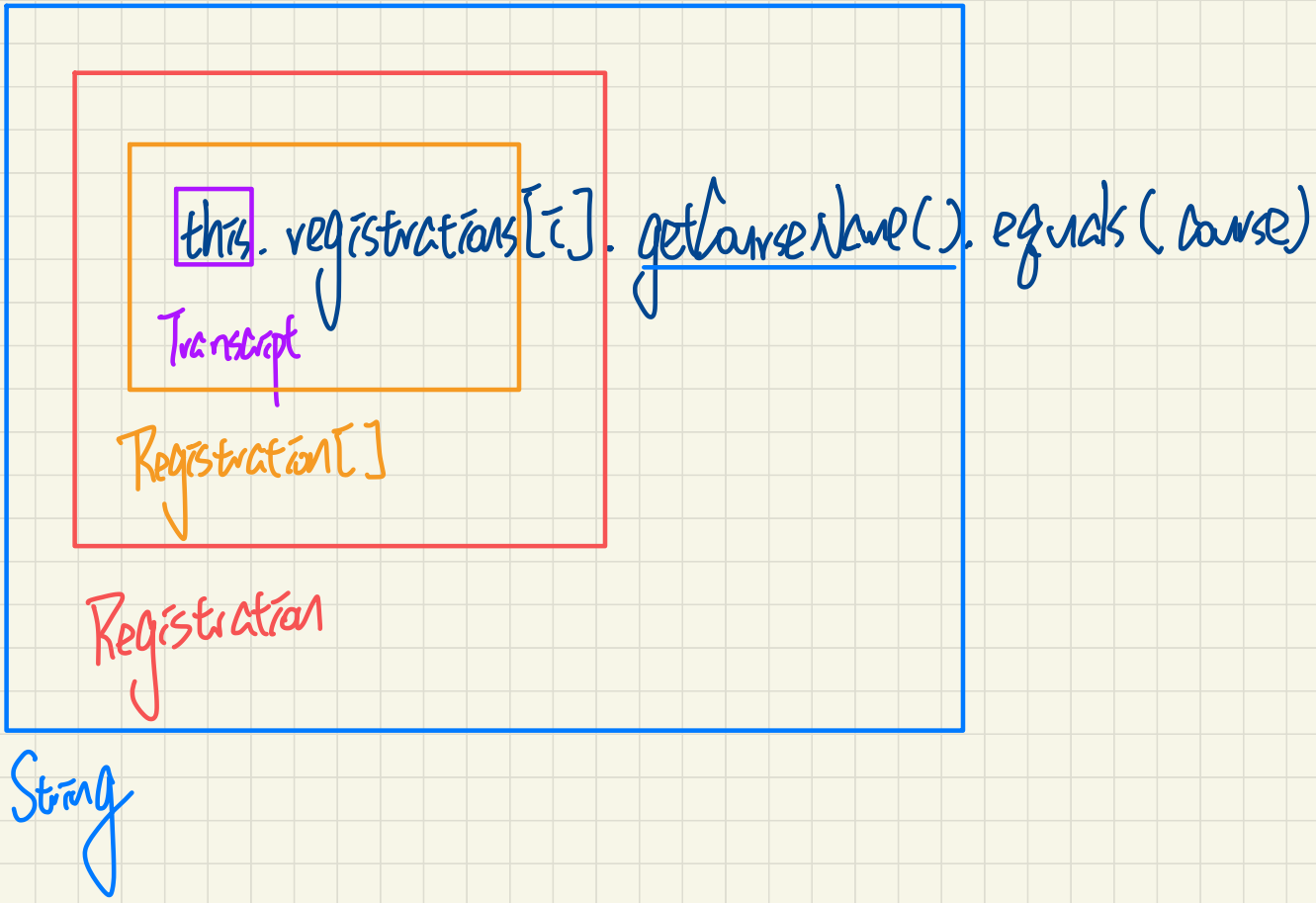
(1) uncomment all remaining tests

(2) declare all the required classes/methods

↓

of accessors,

supply default imp.


this.registrations[i].getCourseName().equals(course)
Transcript
Registration[]
Registration
String

Lecture 10 - Oct. 8

TDD with JUnit, Object Equality

JUnit Test: Exception Expected vs. Not
Using Loops in JUnit Test Methods
Default equals Method in Object Classes

Announcements/Reminders

- ProgTest1 tomorrow
- ProgTest1 review session materials released
- Written Test 1 results released
- Lab1 solution released
- Lab2 released

JUnit: An Exception Expected

```
1  @Test
2  public void testDecFromMinValue() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.decrement();
7      fail("ValueTooSmallException is expected.");
8    }
9    catch(ValueTooSmallException e) {
10     /* Exception is expected to be thrown. */
11   }
12 }
```

exp: VTSE should occur

c.v == 0

expected VTSE did not occur → fail

VTSE occurred as expected → pass

What if increment is implemented correctly?

Expected Behaviour:

Calling c.decrement() when c.value is 0 should trigger a ValueTooSmallException.

```
1  @Test
2  public void testDecFromMinValue() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.decrement();
7      fail("ValueTooSmallException is expected.");
8    }
9    catch(ValueTooSmallException e) {
10     /* Exception is expected to be thrown. */
11   }
12 }
```

What if increment is implemented incorrectly?

e.g., It only throws VTSE when c.value < Counter.MIN_VALUE

Running JUnit Test 2 on Correct Implementation

```
public void decrement() throws ValueTooSmallException {
  if(value == Counter.MIN_VALUE) {
    throw new ValueTooSmallException("counter value is " + value);
  }
  else { value --; }
  }
}
```

```
1  @Test
2  public void testDecFromMinValue() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.decrement();
7      fail("ValueTooSmallException is expected.");
8    }
9    catch(ValueTooSmallException e) {
10     /* Exception is expected to be thrown. */
11   }
12 }
```

abnormal flow → pass

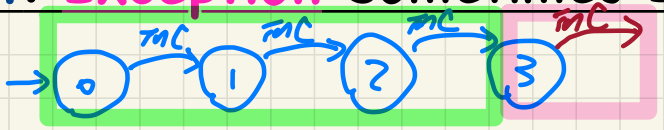
Running JUnit Test 2 on Incorrect Implementation

```
public void decrement() throws ValueTooSmallException {
  if(value <= Counter.MIN_VALUE) {
    throw new ValueTooSmallException("counter value is " + value);
  }
  else { value --; }
 }
}
```

```
1  @Test
2  public void testDecFromMinValue() {
3    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.decrement();
7      fail("ValueTooSmallException is expected.");
8    }
9    catch(ValueTooSmallException e) {
10     /* Exception is expected to be thrown. */
11   }
12 }
```

normal flow → test fails!

JUnit: Exception Sometimes Expected, Somtimes Not




```
1  @Test
2  public void testIncFromMaxValue() {
3    Counter c = new Counter();
4    try {
5      c.increment(); c.increment(); c.increment();
6    }
7    catch (ValueTooLargeException e) {
8      fail("ValueTooLargeException was thrown unexpectedly.");
9    }
10   assertEquals(Counter.MAX_VALUE, c.getValue());
11   try {
12     c.increment();
13     fail("ValueTooLargeException was NOT thrown as expected.");
14   }
15   catch (ValueTooLargeException e) {
16     /* Do nothing: ValueTooLargeException thrown as expected. */
17   }
18 }
```

VTLE not expected

VTLE expected

Expected Behaviour:

Calling c.increment()
3 times to reach c's max should not
trigger any ValueTooLargeException.

Calling c.increment()
when c is already at its max should
trigger a ValueTooLargeException

Running JUnit Test 3 on Correct Implementation

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

```
1  @Test
2  public void testIncFromMaxValue() {
3    Counter c = new Counter();
4    try {
5      c.increment(); c.increment(); c.increment();
6    }
7    catch (ValueTooLargeException e) {
8      fail("ValueTooLargeException was thrown unexpectedly.");
9    }
10   assertEquals(Counter.MAX_VALUE, c.getValue());
11   try {
12     c.increment();
13     fail("ValueTooLargeException was NOT thrown as expected.");
14   }
15   catch (ValueTooLargeException e) {
16     /* Do nothing: ValueTooLargeException thrown as expected. */
17   }
18 }
```

Running JUnit Test 3 on Incorrect Implementation

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

unexpected/wrong

```
1  @Test
2  public void testIncFromMaxValue() {
3    Counter c = new Counter();
4    try {
5      c.increment(); c.increment(); c.increment();
6    }
7    catch (ValueTooLargeException e) {
8      fail("ValueTooLargeException was thrown unexpectedly.");
9    }
10   assertEquals(Counter.MAX_VALUE, c.getValue());
11   try {
12     c.increment();
13     fail("ValueTooLargeException was NOT thrown as expected.");
14
15   catch (ValueTooLargeException e) {
16     /* Do nothing: ValueTooLargeException thrown as expected. */
17   }
18 }
```

fail :(

good test reports a wrong impl.

Running JUnit Test 3 on Incorrect Implementation

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

```
1  @Test
2  public void testIncFromMaxValue() {
3    Counter c = new Counter();
4    try {
5      c.increment(); c.increment(); c.increment();
6    }
7    catch (ValueTooLargeException e) {
8      fail("ValueTooLargeException was thrown unexpectedly.");
9    }
10   assertEquals(Counter.MAX_VALUE, c.getValue());
11   try {
12     c.increment();
13     fail("ValueTooLargeException was NOT thrown as expected.");
14   }
15   catch (ValueTooLargeException e) {
16     /* Do nothing: ValueTooLargeException thrown as expected. */
17   }
18 }
```

test fails
∴ the VTLE did not occur, as expected in 2nd phase

Exercise: Console Tester vs. JUnit Test

Q. Can this ***console tester*** work like the ***JUnit test*** `testIncFromMaxValue` does?

```
1  public class CounterTester {
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      }
9      catch (ValueTooLargeException e) {
10       println("Error: ValueTooLargeException thrown unexpectedly.");
11     }
12     try {
13       c.increment();
14       println("Error: ValueTooLargeException NOT thrown.");
15     } /* end of inner try */
16     catch (ValueTooLargeException e) {
17       println("Success: ValueTooLargeException thrown.");
18     }
19   } /* end of main method */
20 } /* end of CounterTester class */
```

NTLE thrown unexpected

printing out an error about test failing will not stop the exception flow.

Exercise: keep tracing until see which msg printed

Hint: What if one of the first 3 c.increment() mistakenly throws a ValueTooLargeException?

Exercise: Combining `catch` Blocks?

Q: Can we rewrite `testIncFromMaxValue` to:

```
1  @Test
2  public void testIncFromMaxValue() {
3    Counter c = new Counter();
4    try {
5      c.increment();
6      c.increment();
7      c.increment();
8      assertEquals(Counter.MAX_VALUE, c.getValue());
9      c.increment();
10     fail("ValueTooLargeException was NOT thrown as expected.");
11   }
12   catch (ValueTooLargeException e) { }
13 }
```

VTLE can be from here (an expected)

VTLE can be from here (expected)

Is a VTLE here expected or not expected?

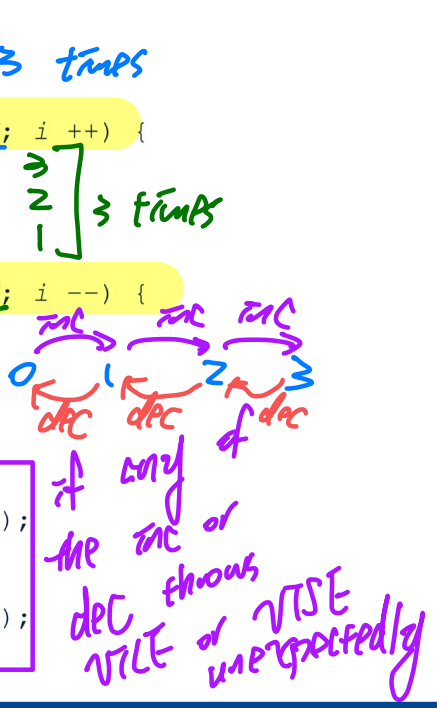
Hint: Say Line 12 is executed,

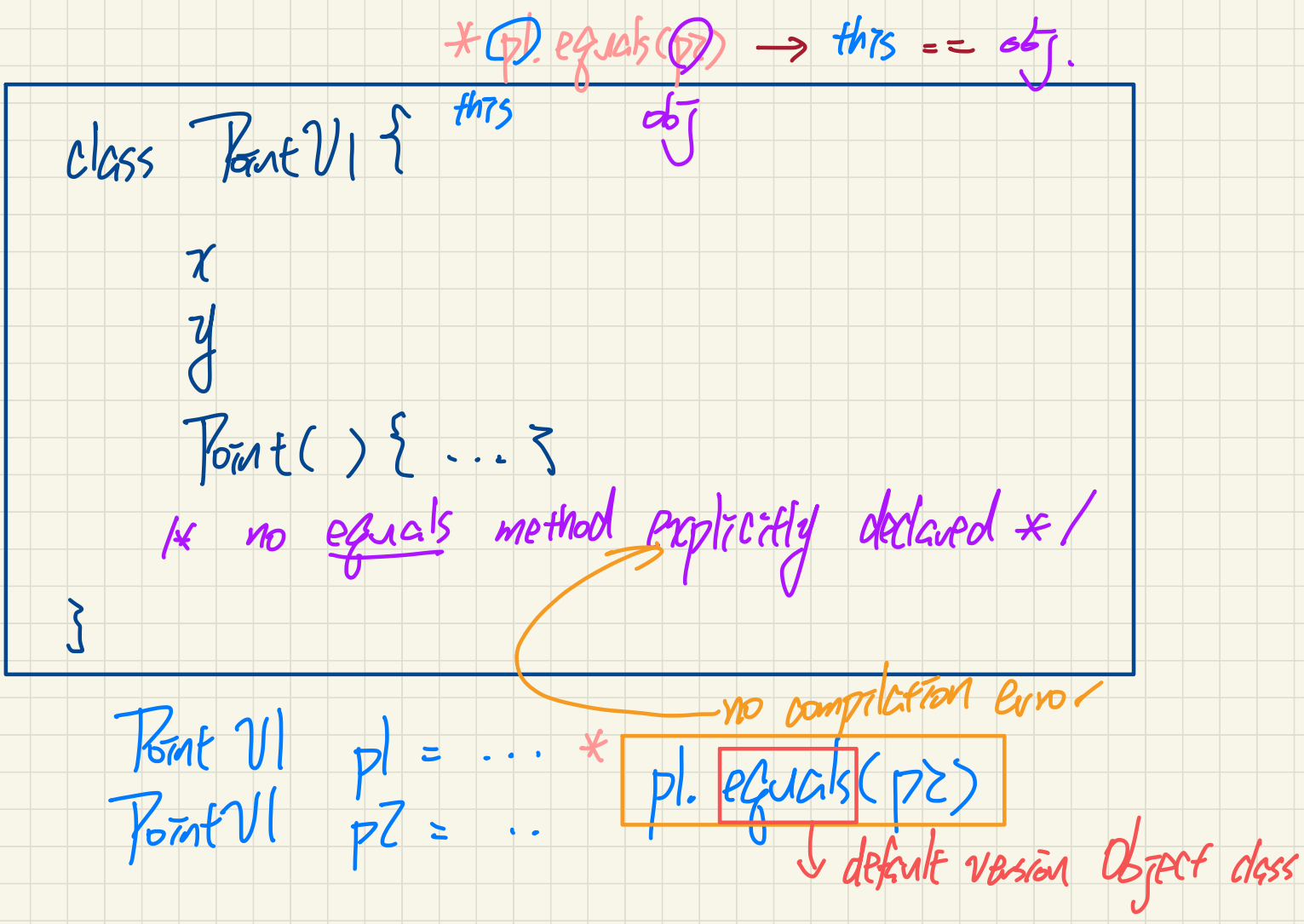
is it clear if that ValueTooLargeException was thrown as expected?

Testing Many Values in a Single Test

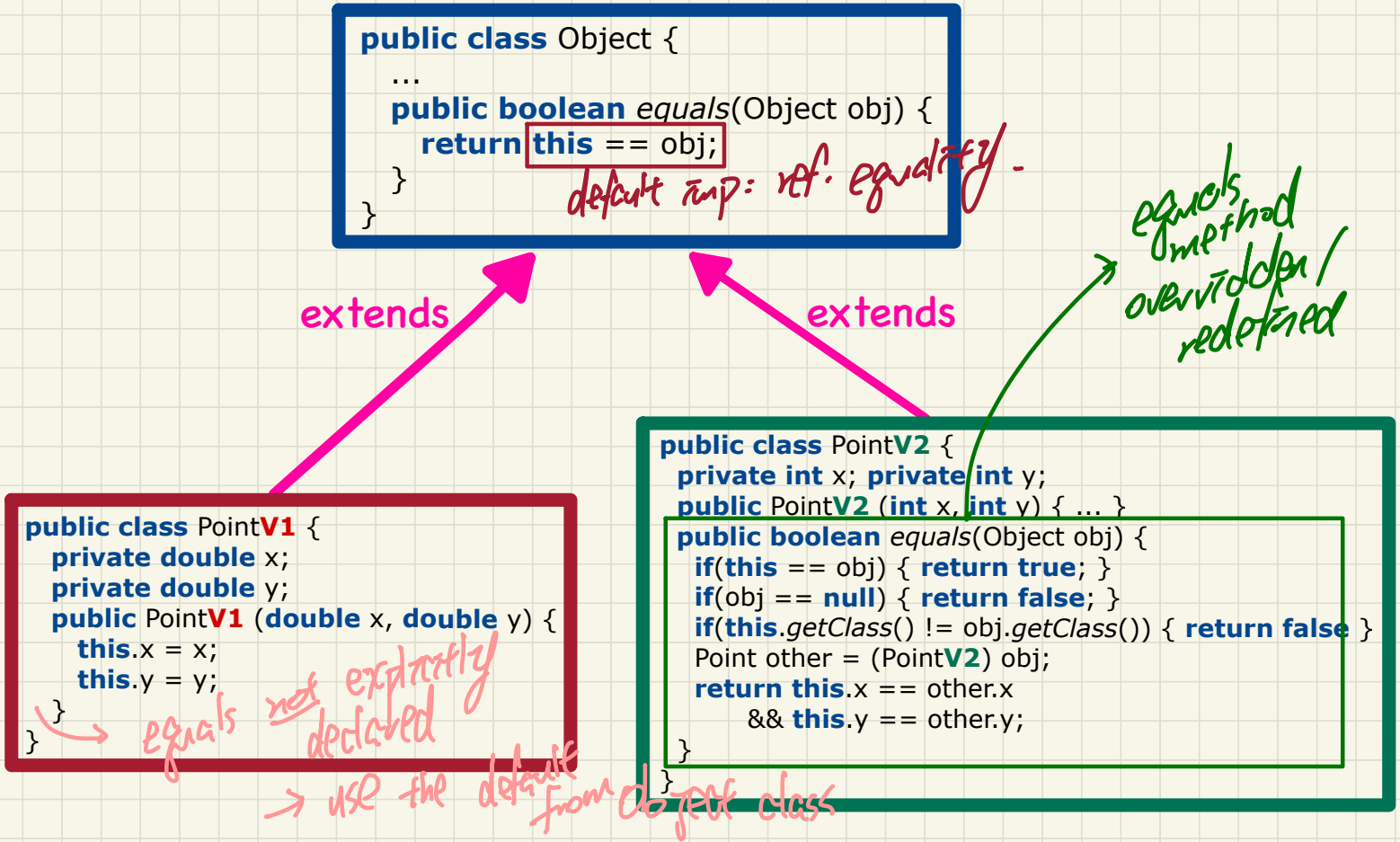
Loops can make it effective on generating test cases:

```
1  @Test
2  public void testIncDecFromMiddleValues() {
3    Counter c = new Counter();
4    try {
5      for(int i = Counter.MIN_VALUE; i < Counter.MAX_VALUE; i ++) {
6        int currentValue = c.getValue();
7        c.increment();
8        assertEquals(currentValue + 1, c.getValue());
9      }
10     for(int i = Counter.MAX_VALUE; i > Counter.MIN_VALUE; i --) {
11       int currentValue = c.getValue();
12       c.decrement();
13       assertEquals(currentValue - 1, c.getValue());
14     }
15   }
16   catch(ValueTooLargeException e) {
17     fail("ValueTooLargeException is thrown unexpectedly");
18   }
19   catch(ValueTooSmallException e) {
20     fail("ValueTooSmallException is thrown unexpectedly");
21   }
22 }
```




* p1.equals(p2) → this == obj.
this
obj
class PointV1 {
x
y
Point( ) { ... }
/* no equals method explicitly declared */
}
no compilation error
PointV1 p1 = ... *
PointV1 p2 = ..
p1.equals(p2)
default version Object class

The equals Method: To Override or Not?



Lecture 11 - Oct. 10

Object Equality

equals Method: Default vs. Overridden
Overriding equals Method: Phases 1 - 3
Static Type, Dynamic Type, Type Casting

Announcements/Reminders

Office Hours during Reading Week TBA:

- Help on Lab2
- Go over WrittenTest1 answers
- Questions on course materials, e.g., OOP reviews

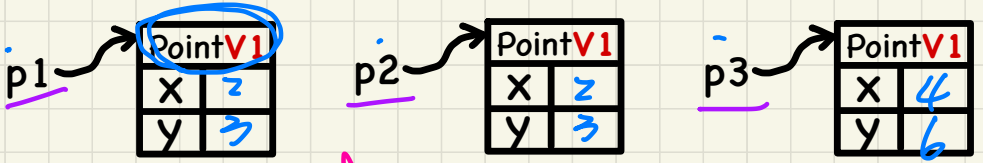
Bonus Opportunity: Midterm Course Survey (eClass)

The equals Method: Default Version

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

```
1  String s = "(2, 3)";
2  PointV1 p1 = new PointV1(2, 3);
3  PointV1 p2 = new PointV1(2, 3);
4  PointV1 p3 = new PointV1(4, 6);
5  System.out.println(p1 == p2); /* [redacted] */
6  System.out.println(p2 == p3); /* [redacted] */
7  System.out.println(p1.equals(p1)); /* [redacted] */
8  System.out.println(p1.equals(null)); /* [redacted] */
9  System.out.println(p1.equals(s)); /* [redacted] */
10 System.out.println(p1.equals(p2)); /* [redacted] */
11 System.out.println(p2.equals(p3)); /* [redacted] */
```

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



The equals Method: Overridden Version

Phase 1

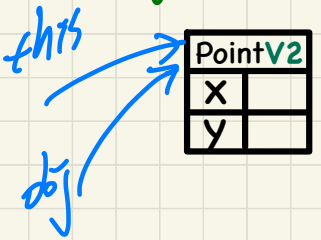
```
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

extends

```
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 }
    Point other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

overridden/redefined equals.

if C.O. "this" is the same object as the input param "obj" → they're equal.



The equals Method: Overridden Version

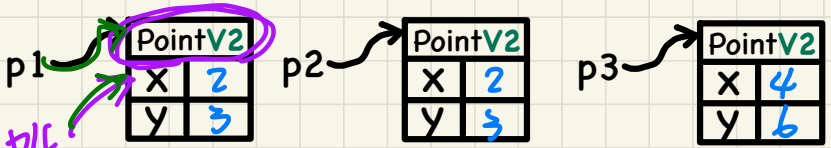
Example 1: Trace L7

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

```
1  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2); /* [redacted] */
6  System.out.println(p2 == p3); /* [redacted] */
7  System.out.println(p1.equals(p1)); /* [redacted] */
8  System.out.println(p1.equals(null)); /* [redacted] */
9  System.out.println(p1.equals(s)); /* [redacted] */
10 System.out.println(p1.equals(p2)); /* [redacted] */
11 System.out.println(p2.equals(p3)); /* [redacted] */
```

extends

```
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 }
      Point other = (PointV2) obj;
      return this.x == other.x
          && this.y == other.y;
   }
}
```



Context object pointing to a PointV2 object → overridden version of equals in PointV2 indeed

method overloading

vs.

method overridding

(re-definition)

The equals Method: Overridden Version

Phase 2

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

extends

```
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 }
    Point other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

reaching this phase means !(this == obj)

* logically unnecessary
if this == null (T)
→ NPE would have occurred.

p1.equals(x); → null → false

** Q. What if this == null is also true?

① if(obj == null){
② if(this == null) return T;
③ else return F;
}

The equals Method: Overridden Version

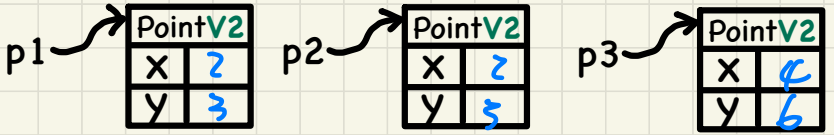
Example 1: Trace L8

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

extends

```
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 }
    Point other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

```
1  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2);   /* [redacted] */
6  System.out.println(p2 == p3);   /* [redacted] */
7  System.out.println(p1.equals(p1));   /* [redacted] */
8  System.out.println(p1.equals(null));   /* [redacted] */
9  System.out.println(p1.equals(s));   /* [redacted] */
10 System.out.println(p1.equals(p2));   /* [redacted] */
11 System.out.println(p2.equals(p3));   /* [redacted] */
```



Static Type, Dynamic Type, Type Casting

Static Type: a ref variable's **declared** type

Dynamic Type: type of address **currently** stored in a ref variable

Type **Casting**: creating an expression of certain **static** type

determines the method that's applicable to the C.O.

```
class C1 {
  ...
  public void m1() {...}
}

class C2 {
  ...
  public void m2() {...}
}
```

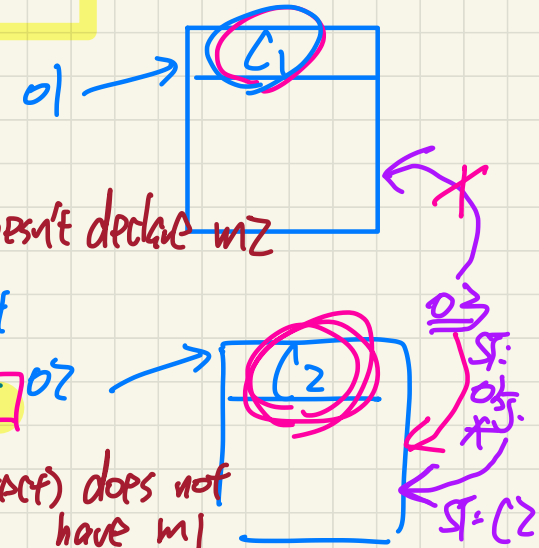
```
① C1 o1 = new C1();      S.T. D.T.
② C2 o2 = new C2();
o1.m1(); ✓
o1.m2(); X ∵ ST of o1 doesn't declare m2
o2.m1(); X
o2.m2(); ✓
Object o3;                ST: Object
o3 = o1;                  ST of o3: Object, DT of o3: C1
o3.m1; X ∵ ST of o3 (Object) does not have m1
o3.m2; X
((C1) o3).m1(); ✓
((C1) o3).m2(); X
o3 = o2;                  ST of o3: Object, DT of o3: C2
* ((C2) o3).m1(); X
((C2) o3).m2(); ✓
```

Object class does not declare m1 and m2.

exp of ST: C1

Q. ((C2) o3).m2(); don't worry for now!

To cast an obj, always cast it to its D.T. D.T.



The equals Method: Overridden Version

Phase 3

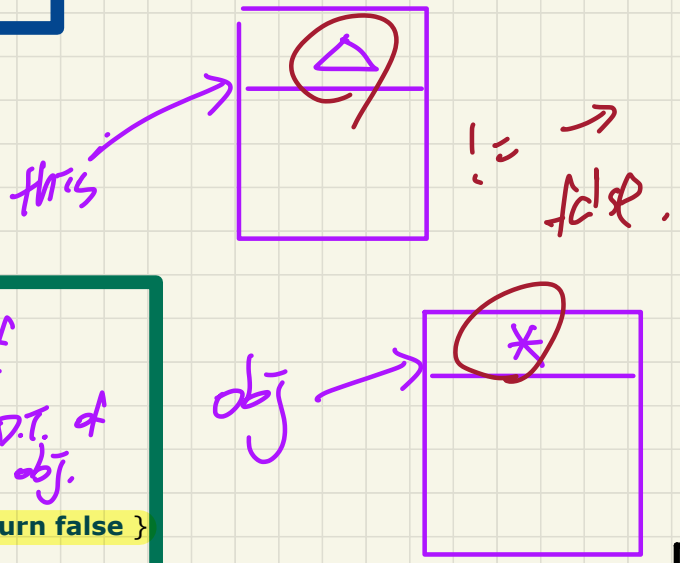
```
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

extends

```
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 }
    Point other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

D.T. of this

D.T. of obj.



PointV2	
x	
y	

The equals Method: Overridden Version

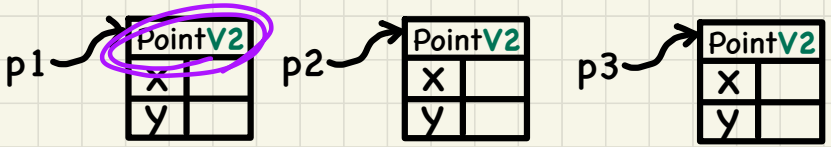
Example 1: Trace L9

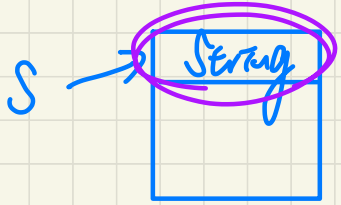
```
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

extends

```
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; }
    Point other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

```
1  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2);   /* [redacted] */
6  System.out.println(p2 == p3);   /* [redacted] */
7  System.out.println(p1.equals(p1));   /* [redacted] */
8  System.out.println(p1.equals(null));   /* [redacted] */
9  System.out.println(p1.equals(s));   /* [redacted] */
10 System.out.println(p1.equals(p2));   /* [redacted] */
11 System.out.println(p2.equals(p3));   /* [redacted] */
```





Lecture 12 - Oct. 22

Object Equality

Reference Equality vs. Object Equality
JUnit: assertSame vs. assertEquals
Logical Op.: Short-Circuit Evaluation

Announcements/Reminders

- ProgTest1 results to be released by next Monday
- Lab2 due this Friday
- ProgTest2 on Wednesday, October 30

  + PDF Guide released

The equals Method: Overridden Version

Phase 4

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

extends

```
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. this != obj
2. obj != null
3. this and obj have same RT.

PointV2.

static type

can invoke only "obj" methods from Object class

compiles ! ST of other is PointV2

this, other → PointV2 (ST: PV2)

obj → PointV2 (ST: Object)

* without this line of type casting,

return this.x == obj.x
&& this.y == obj.y

ST: Object

⇒ x and y are not applicable

PointV2	
x	
y	

The equals Method: Overridden Version

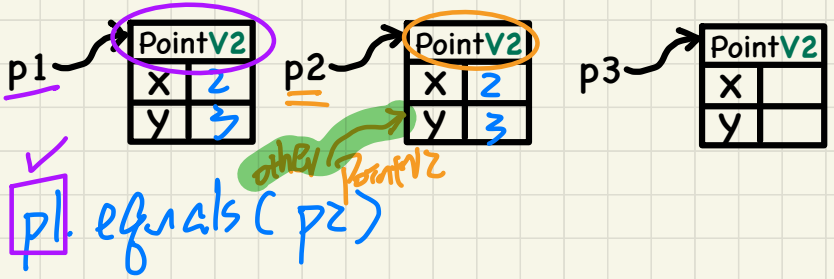
Example 1: Trace L10

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

extends

```
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  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2);   /*        */
6  System.out.println(p2 == p3);   /*        */
7  System.out.println(p1.equals(p1));   /*        */
8  System.out.println(p1.equals(null));   /*        */
9  System.out.println(p1.equals(s));   /*        */
10 System.out.println(p1.equals(p2));   /*        */
11 System.out.println(p2.equals(p3));   /*        */
```



The equals Method: Overridden Version

Example 1: Trace L11

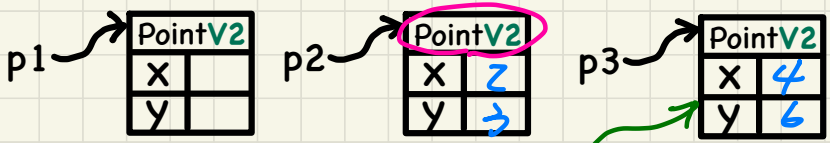
```
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

extends

```
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  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2);   /*       */
6  System.out.println(p2 == p3);   /*       */
7  System.out.println(p1.equals(p1));   /*       */
8  System.out.println(p1.equals(null));   /*       */
9  System.out.println(p1.equals(s));   /*       */
10 System.out.println(p1.equals(p2));   /*       */
11 System.out.println(p2.equals(p3));   /*       */
```

F.



p2.equals(p3)

C.O.
D.T.: PointV2

other
ST: PointV2

The equals Method:

To Override or Not?

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

extends

extends

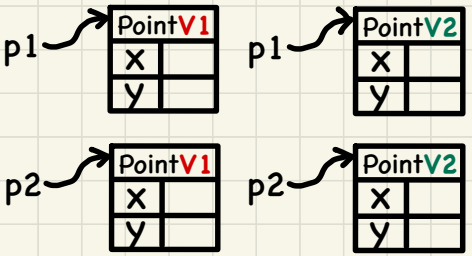
```
public class PointV1 {
   private int x;
   private int y;
   public PointV1 (int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

```
public class PointV2 {
   private int x; double y;
   public PointV2 (double x, double y) { ... }
   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;
   }
}
```

D.T.

```
1  String s = "(2, 3)";
2  PointV1 p1 = new PointV1(2, 3);
3  PointV1 p2 = new PointV1(2, 3);
4  PointV1 p3 = new PointV1(4, 6);
5  System.out.println(p1 == p2);   /* false */
6  System.out.println(p2 == p3);   /* false */
7  System.out.println(p1.equals(p1));   /* true */
8  System.out.println(p1.equals(null));   /* false */
9  System.out.println(p1.equals(s));   /* false */
10 System.out.println(p1.equals(p2));   /* false */
11 System.out.println(p2.equals(p3));   /* false */
```

```
1  String s = "(2, 3)";
2  PointV2 p1 = new PointV2(2, 3);
3  PointV2 p2 = new PointV2(2, 3);
4  PointV2 p3 = new PointV2(4, 6);
5  System.out.println(p1 == p2);   /* false */
6  System.out.println(p2 == p3);   /* false */
7  System.out.println(p1.equals(p1));   /* true */
8  System.out.println(p1.equals(null));   /* false */
9  System.out.println(p1.equals(s));   /* false */
10 System.out.println(p1.equals(p2));   /* true */
11 System.out.println(p2.equals(p3));   /* false */
```



The equals Method: Overridden Version

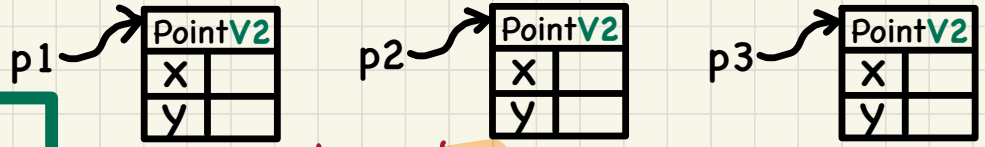
Example 2

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

extends

```
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  PointV2 p1 = new PointV2(3, 4);
2  PointV2 p2 = new PointV2(3, 4);
3  PointV2 p3 = new PointV2(4, 5);
4  System.out.println(p1 == p1);   /*      */
5  System.out.println(p1.equals(p1));   /*      */
6  System.out.println(p1 == p2);   /*      */
7  System.out.println(p1.equals(p2));   /*      */
8  System.out.println(p2 == p3);   /*      */
9  System.out.println(p2.equals(p3));   /*      */
```



(A) Two objects are reference-equal.

(B) Two objects are contents-equal.

1- If (A) is true, then (B) is true.

2- If (B) is true, then (A) is true.

¬ x.equals(null)

↳ x.equals(null)

should always return false

return true means x == null

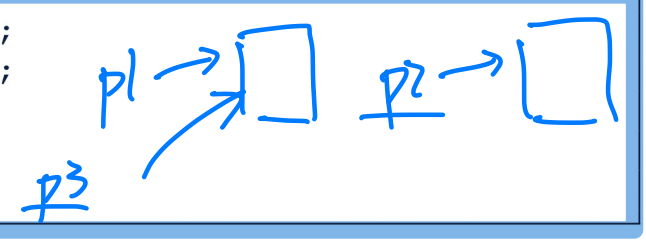
↳ NPE would have occurred.

assertSame vs. assertEquals

assertSame(exp1, exp2)

- Passes if exp1 and exp2 are references to the same object
  - ≈ **assertTrue**(exp1 == exp2)
  - ≈ **assertFalse**(exp1 != exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV1 p3 = p1;
assertSame(p1, p3);
assertSame(p2, p3);
```



assertEquals(exp1, exp2)

- ≈ exp1 == exp2 if exp1 and exp2 are **primitive** type

```
int i = 10;
int j = 20;
assertEquals(i, j);
```

assertEquals: Reference Comparison or Not

assertEquals(exp1, exp2) → ref. types

- ≈ exp1.**equals**(exp2) if exp1 and exp2 are ***reference*** type

Case 1: If `equals` is ***not*** explicitly overridden in `exp1`'s declared type
≈ ***assertSame***(exp1, exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2);
assertEquals(p2, p3);
```

p1.eq.(p2) → default version of equals
p2.eq.(p3) → default version of equals

Case 2: If `equals` is explicitly ***overridden*** in `exp1`'s declared type
≈ exp1.***equals***(exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2);
assertEquals(p2, p3);
assertEquals(p3, p2);
```

D.T.
(p3).eq.(p2) → invoke the overridden version in PointV2

* given that both p2 and p3 store (3,4), is p3.equals(p2) returning true? (F)

(p1 == p2) (F)
"p2 == p3" (F)

ref type ref type

① assertEquals(exp1 , exp2)

context object argument

↳ assertTrue(exp1.equals(exp2))

D.T. determines version of equals to invoke.

② assertEquals(exp2 , exp1)

↳ assertTrue(exp2.equals(exp1))

Commutativity

$$p \wedge q \equiv q \wedge p$$

In Java:

$$p \;\&\&\; q \;\overset{?}{\equiv}\; q \;\&\&\; p$$

No!!!

Short-Circuit Evaluation: &&

Left Operand `op1`	Right Operand `op2`	`op1 && op2`
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

Test Inputs:

x = 0, y = 10

x = 5, y = 10

```
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x != 0 && y / x > 2) {
    System.out.println("y / x is greater than 2");
}
else { /* !(x != 0 && y / x > 2) == (x == 0 || y / x <= 2) */
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is not greater than 2");
    }
}
```

p && q

① Evaluate p.

② 1 If p eval. to true then eval. q.

② 2 If p eval. to false
↳ skip eval. of q.

Short-Circuit Evaluation: ||

Left Operand op1	Right Operand op2	op1 || op2
false	false	false
true	false	true
false	true	true
true	true	true

Test Inputs:

x = 0, y = 10

x = 5, y = 10

```
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x == 0 || y / x > 2) {
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is greater than 2");
    }
}
else { /* !(x == 0 || y / x > 2) == (x != 0 && y / x <= 2) */
    System.out.println("y / x is not greater than 2");
}
```

p || q

① Evaluate p

② If p eval. to false, then eval. q.

② If p eval. to true, then skip eval. of q.

Lecture 13 - Oct. 24

Object Equality, Call by Value

Short-Circuit Evaluation: && vs. ||
equals: Person vs. PersonCollector

Announcements/Reminders

- Lab2 due tomorrow at noon
- Lab3 to be released tomorrow
- ProgTest2 next Wednesday, October 30
  + PDF Guide released

Short-Circuit Evaluation: &&

Conjunction

Left Operand op1	Right Operand op2	op1 && op2
true ✓	true ✓	✓ true
true ✓	false ✓	✓ ~~false~~
false ✓	- true	false
false ✓	- false	false

Test Inputs:

- x = 0, y = 10
- x = 5, y = 10

eval →

expl && exp2

```
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x != 0 && y / x > 2) {
    System.out.println("y / x is greater than 2");
}
else { /* !(x != 0 && y / x > 2) == (x == 0 || y / x <= 2) */
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is not greater than 2");
    }
}
```

x = 0, y = 10

0 != 0 && 10/0 > 2

F → false. skip

① expl : T

eval exp2

② expl : F

skip eval. exp2

→ && → F

Short-Circuit Evaluation: ||

disjunction

Left Operand op1	Right Operand op2	op1 || op2
false	false	false
true	false	true
false	true	true
true	true	true

Test Inputs:

x = 0, y = 10

x = 5, y = 10

```
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x == 0 || y / x > 2) {
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is greater than 2");
    }
}
else { /* !(x == 0 || y / x > 2) == (x != 0 && y / x <= 2) */
    System.out.println("y / x is not greater than 2");
}
```

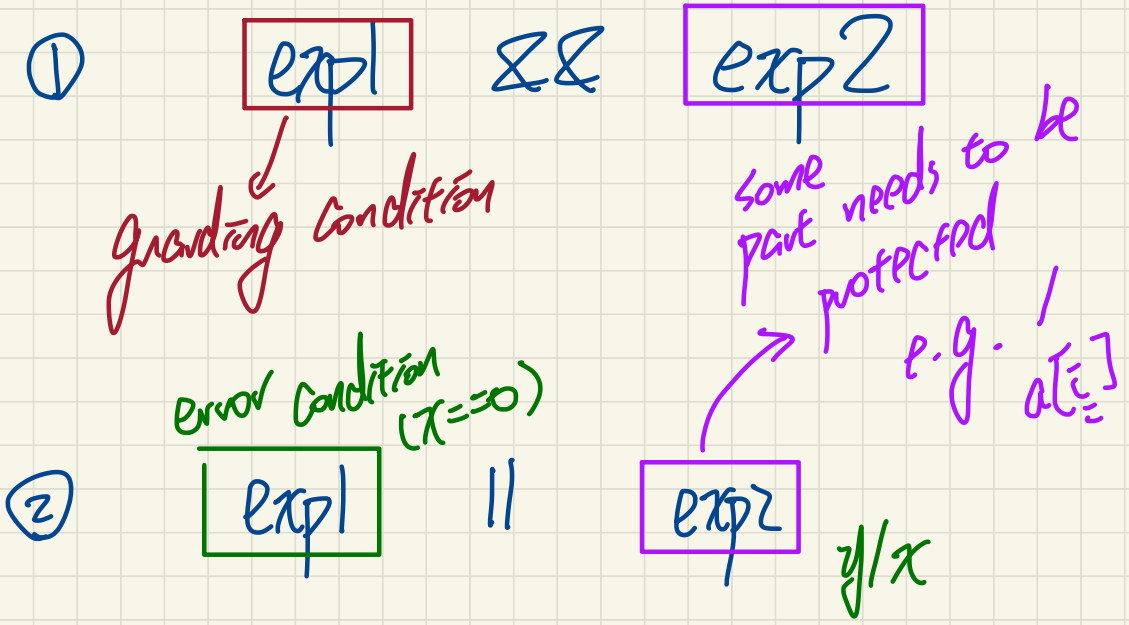
0 == 0 (T) || 10/0 > 2 skipped (T)

exp1 || exp2

① exp1 : F
eval. exp2 to decide

② exp1 : T ✓
skip eval. exp2
↳ _||_ T.

Short Circuit Evaluation



Exercise

① ____A.____ && ____B.____ && a[i] > 0

② ____C.____ || ____D.____ || a[i] > 0

Short-Circuit Evaluation: Common Errors

Test Inputs:

x = 0, y = 10

Short-Circuit Evaluation is not exploited: crash when `x == 0`

```
if (y / x > 2 && x != 0) {
  /* do something */
}
else {
  /* print error */ }
```

|10/0| > 2 → crash.

↓ guarding cond. will not be reached!

Short-Circuit Evaluation is not exploited: crash when `x == 0`

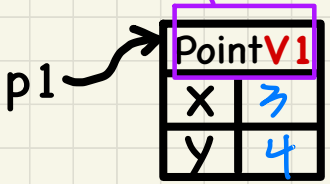
```
if (y / x <= 2 || x == 0) {
  /* print error */
}
else {
  /* do something */ }
```

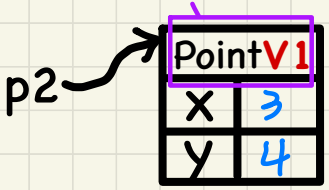
Testing Default Equality of Points in JUnit

```
@Test
public void testEqualityOfPointV1() {
  PointV1 p1 = new PointV1(3, 4); PointV1 p2 = new PointV1(3, 4);
  assertFalse(p1 == p2); assertFalse(p2 == p1);
  /* assertSame(p1, p2); assertSame(p2, p1); */ /* both fail */
  assertFalse(p1.equals(p2)); assertFalse(p2.equals(p1));
  assertTrue(p1.getX() == p2.getX() && p2.getY() == p2.getY());
}
```

p1 == p2

p2 == p1






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

extends

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

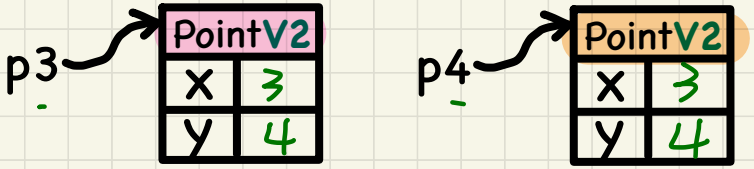
Testing Overridden Equality of Points in JUnit

```
@Test
public void testEqualityOfPointV2() {
  PointV2 p3 = new PointV2(3, 4); PointV2 p4 = new PointV2(3, 4);
  assertFalse(p3 == p4); assertFalse(p4 == p3);
  /* assertSame(p3, p4); assertSame(p4, p3); */ /* both fail */
①assertTrue(p3.equals(p4)); assertTrue(p4.equals(p3));
②assertEquals(p3, p4); assertEquals(p4, p3);
}
```

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

extends

```
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 assertTrue(p3.equals(p4))

② assertEquals(p3, p4)

Testing Equality of Points in JUnit: Default vs. Overridden

```
@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));
}
```

"p1==p2"

Q. Do these two assertions pass for the same reason?

* p1 == p2 (F)

** equals from PointV2

Q.

(F)

(F)

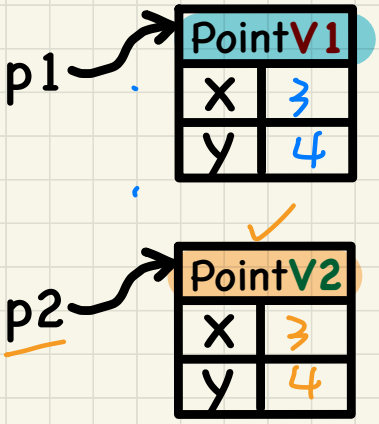
```
public class Object {
  ...
  public boolean equals(Object obj) {
    return this == obj;
  }
}
```

* D.T. of C.O. PointV1

** D.T. of C.O. PointV2

extends

extends




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

```
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;
  }
}
```

*Exercise

Combine ① and ② using conjunction (&&).

```
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; }
①   if(obj == null) { return false; }
②   if(this.getClass() != obj.getClass()) { return false }
    PointV2 other = (PointV2) obj;
    return this.x == other.x
        && this.y == other.y;
  }
}
```

(A) if (① || ②) { return false; }

① → obj == null

avoids NPE when obj is null

(B) if (② || ①) { return false; }

② → ... obj.getClass()

① → obj == null

may still have NPE when obj is null

Exercise: Two Persons are equal if their names and measures are equal

```
1  public class Person {
2    private String firstName; private String lastName;
3    private double weight; private double height;
4    public boolean equals(Object obj) {
5      if(this == obj) { return true; }
6      if(obj == null || this.getClass() != obj.getClass()) { return false; }
7      Person other = (Person) obj;
8      return
9          this.weight == other.weight
10       && this.height == other.height
11       && this.firstName.equals(other.firstName)
12       && this.lastName.equals(other.lastName);
13   }
14 }
```

this.firstName.equals(...)

Person

String.

① Object ② Person ③ String

Q1: At Line 6, will there be a **NullPointerException** if obj == **null**?

Q2: At Line 6, what if we change it to:

if(this.*getClass*() != obj.*getClass*() || obj == **null**)

NPE if obj is null.

Q3: At Lines 11 & 12 which version of the **equals** method is called?

Exercise: PersonCollectors are equal if their arrays of persons are equal

```
class PersonCollector {
  private Person[] persons;
  private int nop; /* number of persons */
  public PersonCollector() { ... }
  public void addPerson(Person p) { ... }
  public int getNop() { return this.nop; }
  public Person[] getPersons() { ... }
}
```

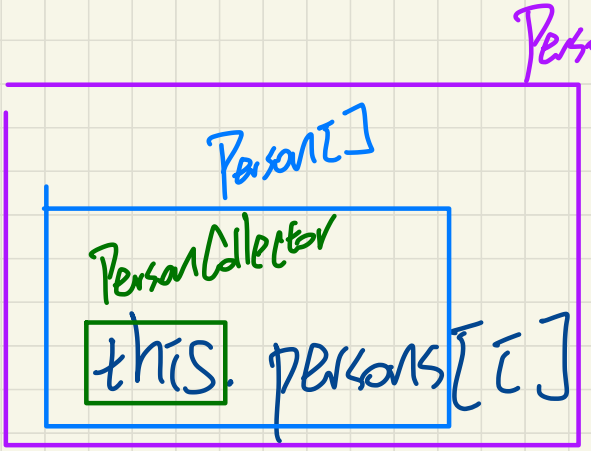
```
1  public boolean equals(Object obj) {
2    if(this == obj) { return true; }
3    if(obj == null || this.getClass() != obj.getClass()) { return false; }
4    PersonCollector other = (PersonCollector) obj;
5    boolean equal = false;
6    if(this.nop == other.nop) {
7      equal = true;
8      for(int i = 0; equal && i < this.nop; i ++) {
9        equal = this.persons[i].equals(other.persons[i]);
10     }
11   }
12   return equal;
13 }
```

as soon as equal becomes false exit from the loop

Q: At Line 9 of PersonCollector's equals method which version of the equals method is called?

```
1  public class Person {
2    private String firstName; private String lastName;
3    private double weight; private double height;
4    public boolean equals(Object obj) {
5      if(this == obj) { return true; }
6      if(obj == null || this.getClass() != obj.getClass()) { return false; }
7      Person other = (Person) obj;
8      return
9          this.weight == other.weight
10       && this.height == other.height
11       && this.firstName.equals(other.firstName)
12       && this.lastName.equals(other.lastName);
13   }
14 }
```

* the two Person[] store the same # of persons.



(A) Object

(B) Person

(C) PersonCollector

(D) String

Alt.

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

String class

Lecture 14 - Oct. 29

Call by Value

equals: PersonsCollector
Call by Value

Announcements/Reminders

- ProgTest1 marks, submissions, grading tests released
- ProgTest2 tomorrow during your enrolled lab session
- Lab3 due this Friday at noon

Exercise: PersonCollectors are equal if their arrays of persons are equal

```
class PersonCollector {
  private Person[] persons;
  private int nop; /* number of persons */
  public PersonCollector() { ... }
  public void addPerson(Person p) { ... }
  public int getNop() { return this.nop; }
  public Person[] getPersons() { ... }
}
```

```
1  public boolean equals(Object obj) {
2    if(this == obj) { return true; }
3    if(obj == null || this.getClass() != obj.getClass()) { return false; }
4    PersonCollector other = (PersonCollector) obj;
5    boolean equal = false;
6    if(this.nop == other.nop) {
7      equal = true;
8      for(int i = 0; equal && i < this.nop; i ++) {
9        equal = this.persons[i].equals(other.persons[i]);
10     }
11   }
12   return equal;
13 }
```

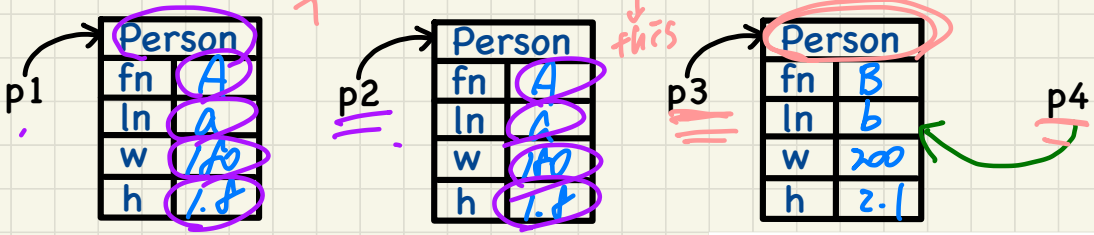
* as soon as equal becomes false exit from the loop

Q: At Line 9 of PersonCollector's equals method which version of the equals method is called?

```
1  public class Person {
2    private String firstName; private String lastName;
3    private double weight; private double height;
4    public boolean equals(Object obj) {
5      if(this == obj) { return true; }
6      if(obj == null || this.getClass() != obj.getClass()) { return false; }
7      Person other = (Person) obj;
8      return
9          this.weight == other.weight
10       && this.height == other.height
11       && this.firstName.equals(other.firstName)
12       && this.lastName.equals(other.lastName);
13   }
14 }
```

Testing Equality of Person/PersonCollector in JUnit (1)

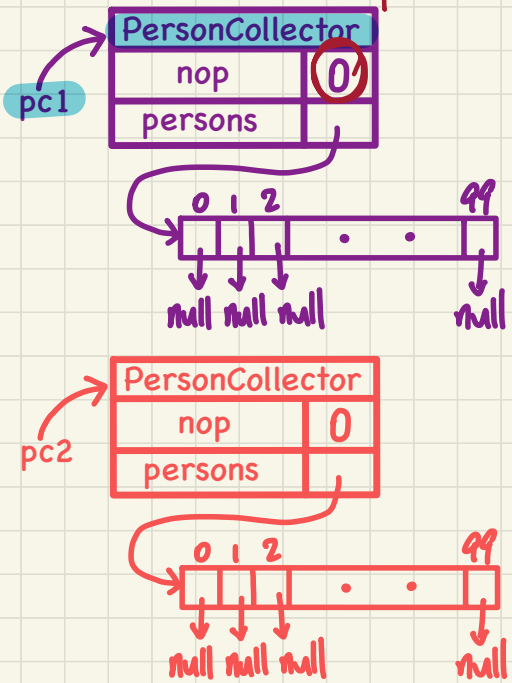
```
@Test
public void testPersonCollector() {
  Person p1 = new Person("A", "a", 180, 1.8);
  Person p2 = new Person("A", "a", 180, 1.8);
  Person p3 = new Person("B", "b", 200, 2.1);
  Person p4 = p3;
  assertFalse(p1 == p2); assertTrue(p1.equals(p2));
  assertTrue(p3 == p4); assertTrue(p3.equals(p4));
```




```
public class Person {
  private String firstName; private String lastName;
  private double weight; private double height;
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null || this.getClass() != obj.getClass()) { return false; }
    Person other = (Person) obj;
    return
        this.weight == other.weight
      && this.height == other.height
      && this.firstName.equals(other.firstName)
      && this.lastName.equals(other.lastName);
  }
}
```

Testing Equality of Person/PersonCollector in JUnit (2)
(continued from testPersonCollector)

```
PersonCollector pc1 = new PersonCollector();
PersonCollector pc2 = new PersonCollector();
assertFalse(pc1 == pc2); assertTrue(pc1.equals(pc2));
```



Q: How about **assertTrue**(pc2.**equals**(pc1))?

```
class PersonCollector {
  private Person[] persons;
  private int nop; /* number of persons */
  public PersonCollector() { ... }
  public void addPerson(Person p) { ... }
  public int getNop() { return this.nop; }
  public Person[] getPersons() { ... }
}
```

```
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;
}
```

Testing Equality of Person/PersonCollector in JUnit (3)

(continued from testPersonCollector)

```
pc1.addPerson(p1);
assertFalse(pc1.equals(pc2));
pc2.addPerson(p2);
assertFalse(pc1.getPersons()[0] == pc2.getPersons()[0]);
assertTrue(pc1.getPersons()[0].equals(pc2.getPersons()[0]));
assertTrue(pc1.equals(pc2));
pc1.addPerson(p3);
pc2.addPerson(p4);
assertTrue(pc1.getPersons()[1] == pc2.getPersons()[1]);
assertTrue(pc1.getPersons()[1].equals(pc2.getPersons()[1]));
assertTrue(pc1.equals(pc2));
```

Person

```
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  Person other = (Person) obj;
  return
      this.weight == other.weight
   && this.height == other.height
   && this.firstName.equals(other.firstName)
   && this.lastName.equals(other.lastName);
}
```

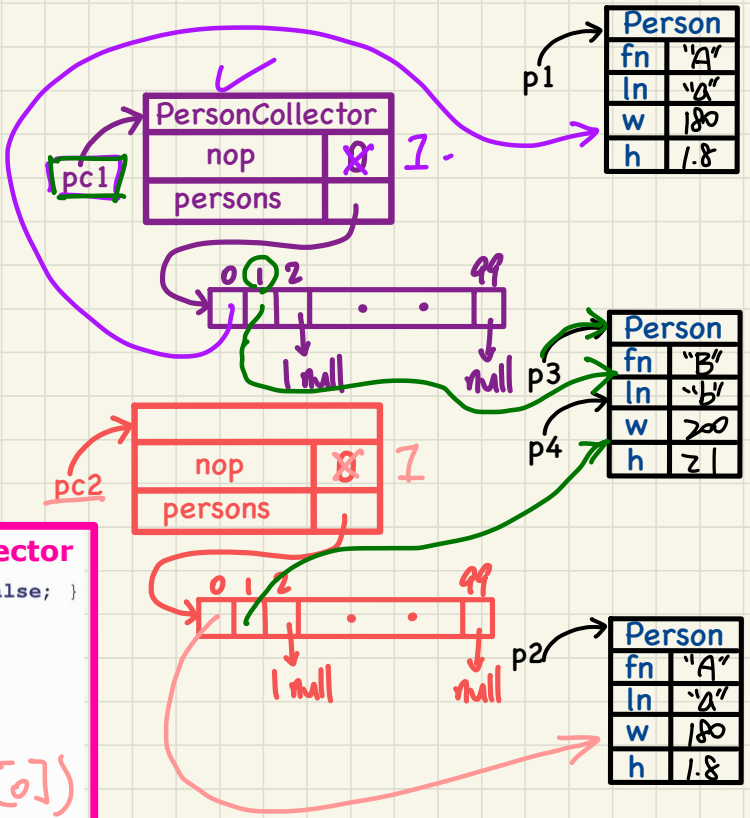
PersonCollector

```
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;
}
```

pc1.nop 1
pc2.nop 0

this.persons[0].equals(other.persons[0])

pc2 Person pc1



Testing Equality of Person/PersonCollector in JUnit (4)

(continued from testPersonCollector)

```
pc1.addPerson(new Person("A", "a", 175, 1.75));
pc2.addPerson(new Person("A", "a", 165, 1.55));
assertFalse(pc1.getPersons()[2] == pc2.getPersons()[2]);
assertFalse(pc1.getPersons()[2].equals(pc2.getPersons()[2]));
assertFalse(pc1.equals(pc2));
```

F

Person	
fn	A
ln	a
w	175
h	1.75

Person

```
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  Person other = (Person) obj;
  return
       this.weight == other.weight
    && this.height == other.height
    && this.firstName.equals(other.firstName)
    && this.lastName.equals(other.lastName);
}
```

PersonCollector

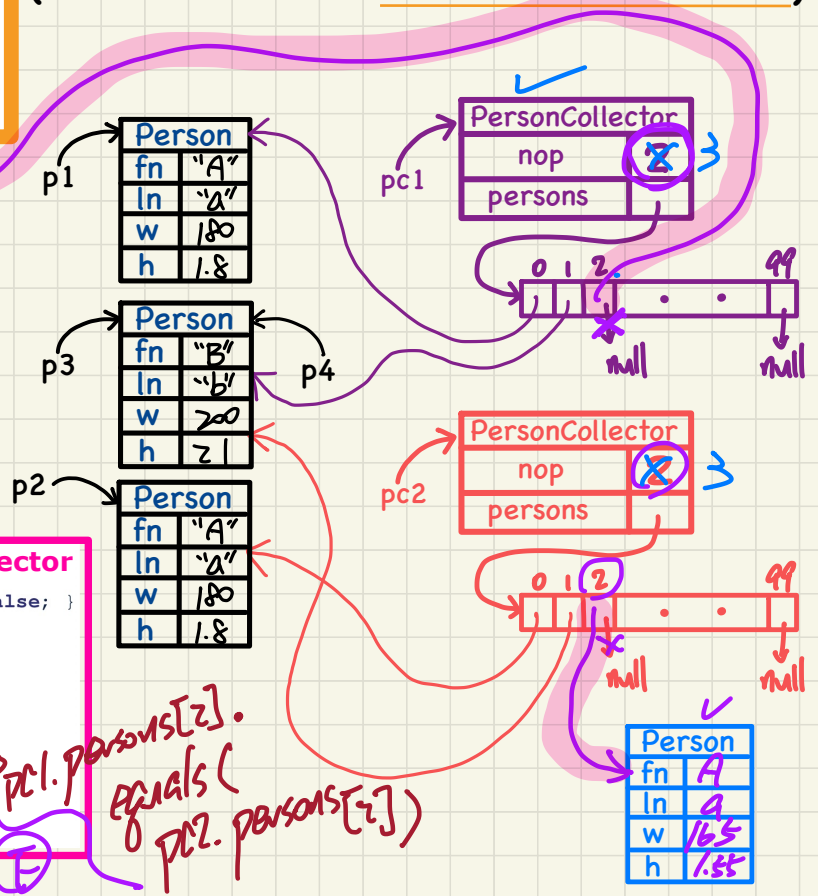
```
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  PersonCollector other = (PersonCollector) obj;
  boolean equal = false;
  if(this.nop == other.nop) {
    equal = true;
    for(int i = 0; equal && i 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

3, F, F

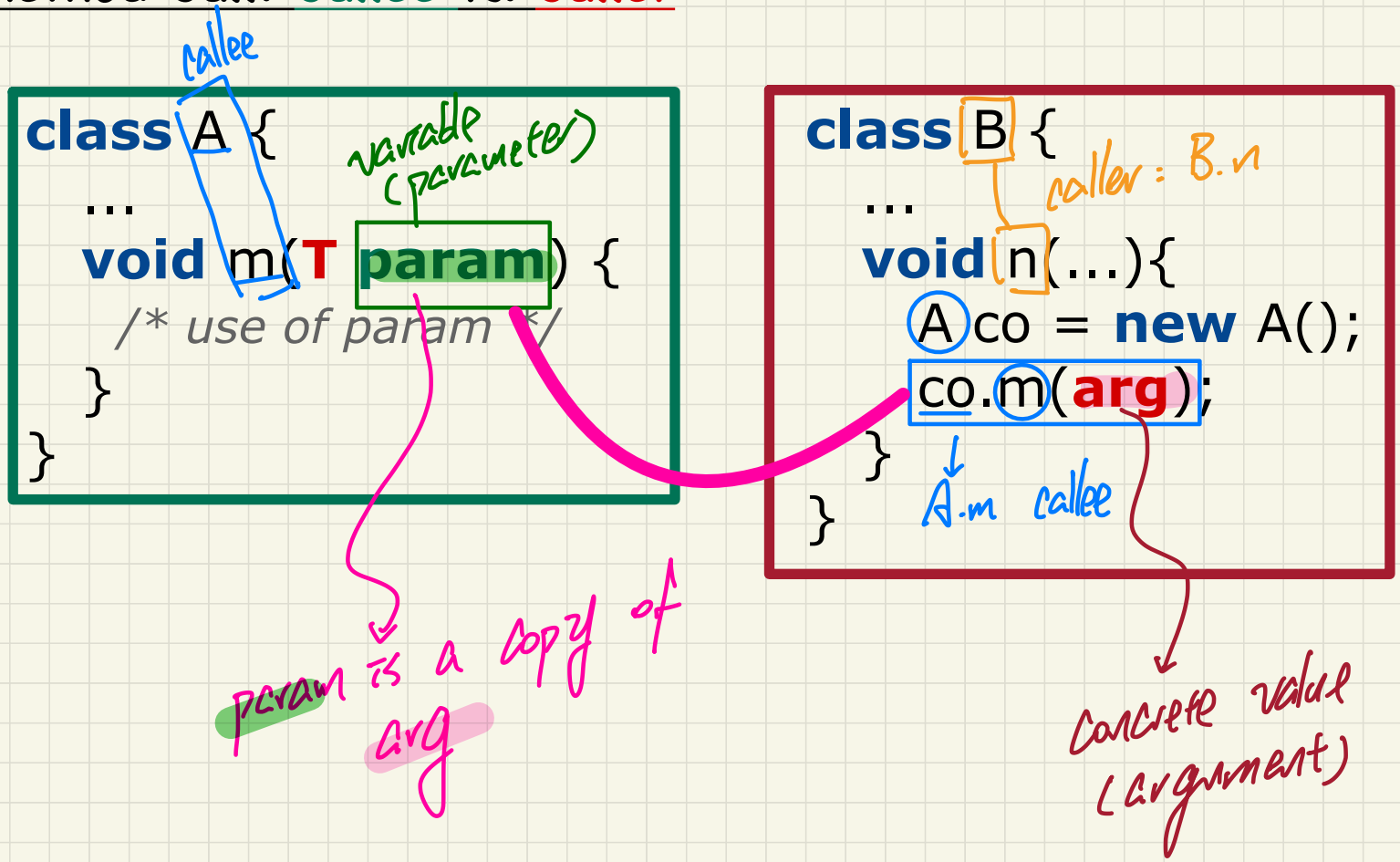
I: 0 → 1 → 2

pc1.persons[2].equals(pc2.persons[2])

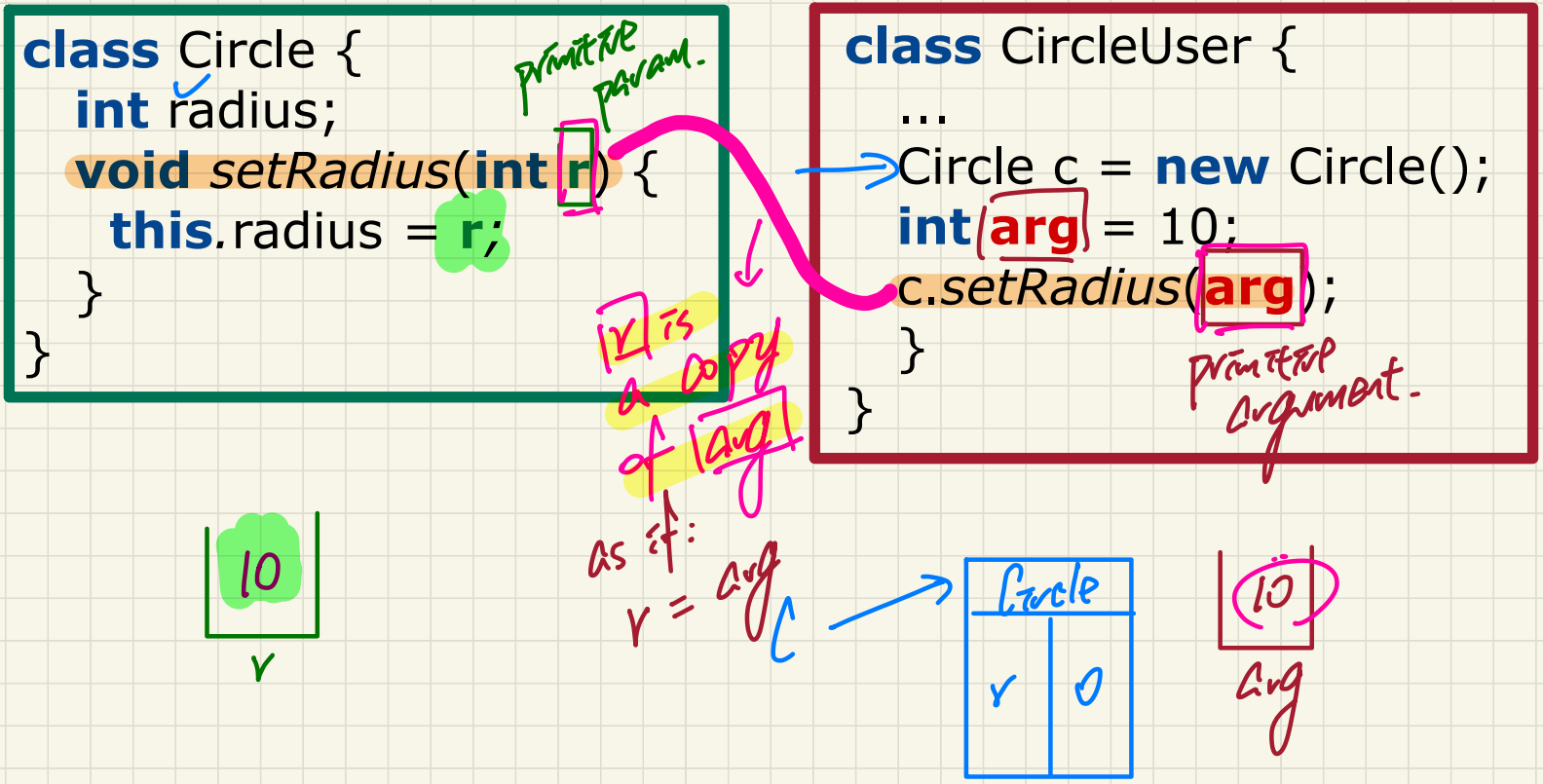
T



Method Call: Callee vs. Caller



Call by Value: Primitive Argument



Call by Value: Reference Argument

```
class Circle {
   int radius;
   Circle() {}
   Circle(int r) {
      this.radius = r;
   }
   void setRadius(Circle c) {
      this.radius = c.radius;
   }
}
```

```
class CircleUser {
   ...
   Circle c = new Circle();
   Circle arg = new Circle(10);
   c.setRadius(arg);
   }
}
```

ref. type parameter

ref. type argument

c is a copy of arg

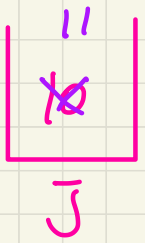
as if: c = arg

Call by Value: Re-Assigning Primitive 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 testCallByVal() {
3    Util u = new Util();
4    int i = 10;
5    assertTrue(i == 10);
6    u.reassignInt(i);
7    assertTrue(i == 10);
8  }
```

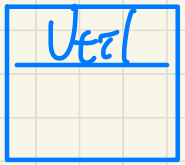
↳ arg stays untouched.

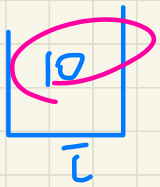


call by value:

j = i

u





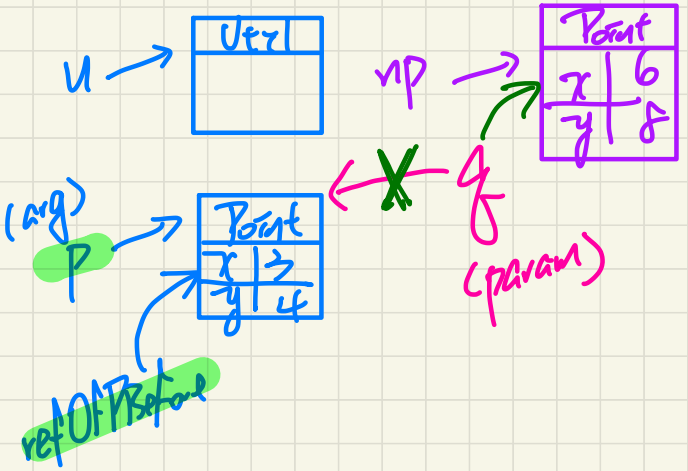
Call by Value: Re-Assigning Reference Parameter

```
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); } }
```

call by value: q = p

```
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 }
```

never possible to re-assign eg. p using a method call.

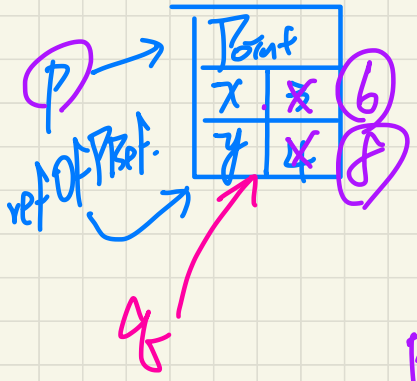



```
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x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

Call by Value: Calling Mutator on Reference Parameter

```
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_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 }
```



Contents of the object pointed by P were modified

```
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: Invoking the Overridden equals

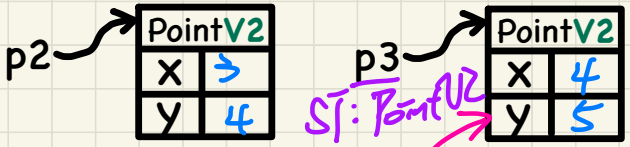
```
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

extends

```
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;
  }
}
```

obj = p3 ✓

```
1  PointV2 p1 = new PointV2(3, 4);
2  PointV2 p2 = new PointV2(3, 4);
3  PointV2 p3 = new PointV2(4, 5);
4  System.out.println(p1 == p1);   /* true */
5  System.out.println(p1.equals(p1));   /* true */
6  System.out.println(p1 == p2);   /* false */
7  System.out.println(p1.equals(p2));   /* true */
8  System.out.println(p2 == p3);   /* false */
9  System.out.println(p2.equals(p3));   /* false */
```



obj ⇓ ST: Object

Lecture 15 - Oct. 31

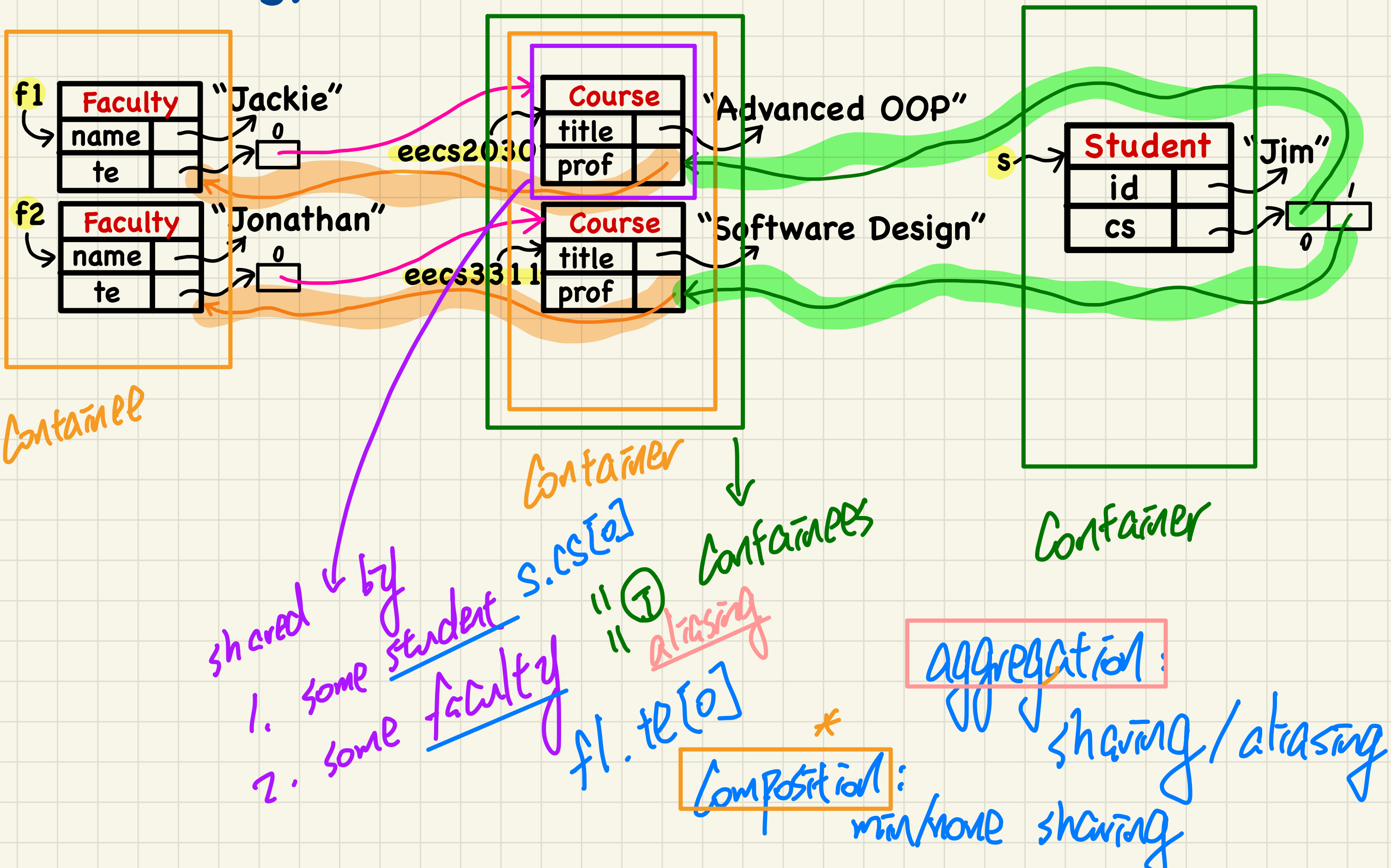
Aggregation and Composition

Modelling: Aggregation vs. Composition
Dot Notation Exercise
Copy Constructor, Shallow Copy

Announcements/Reminders

- Lab3 due tomororw at noon
- Lab4 to be released tomorrow
- ProgTest2 results & feedback tentatively Monday Nov 11

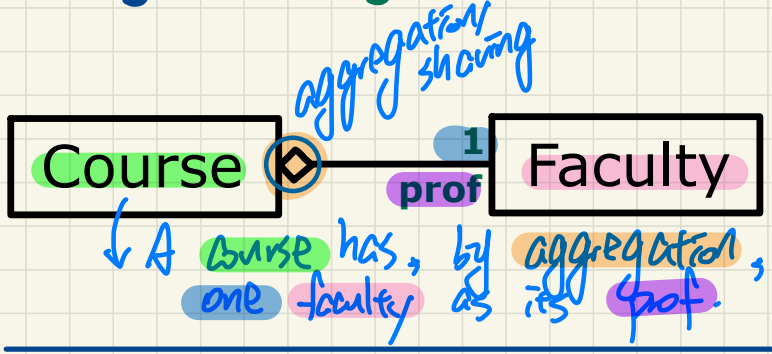
Terminology: Container vs. Containee



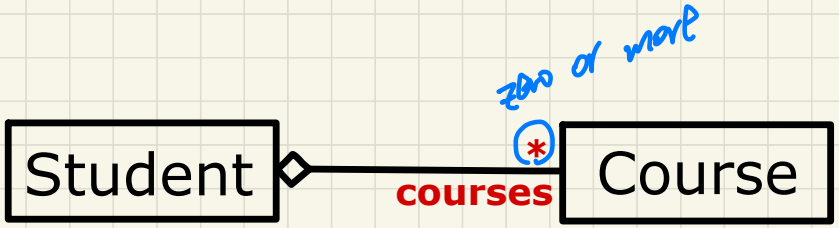
Aggregation: Design

composition: no sharing.

Design 1: Single Containee



Design 2: Multiple Containees



Java Implementation

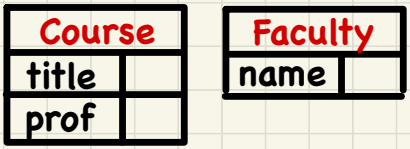
```
class Course {
   Faculty prof;
   ...
}
```

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

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

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

Aggregation (1)




```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"));
}
```

Aggregation (2)

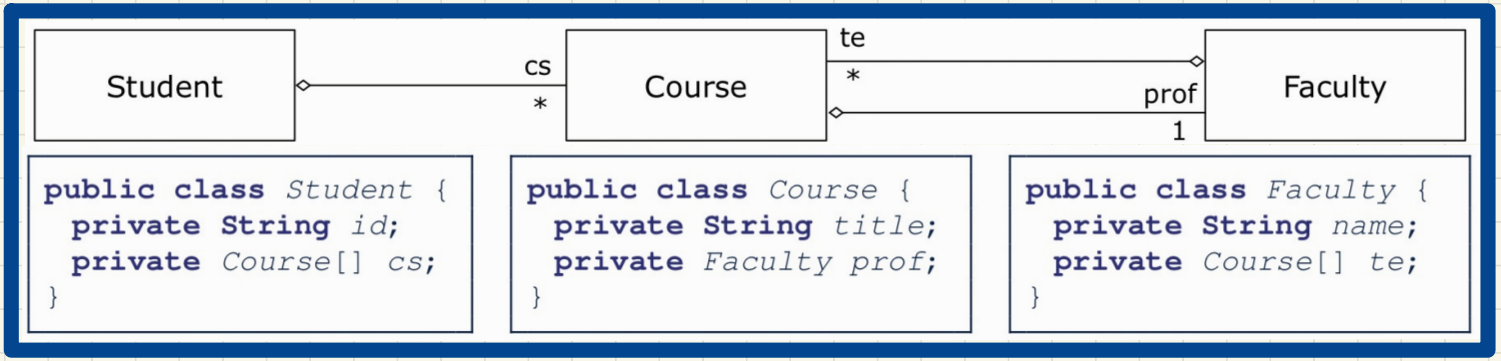

```java
public class Student {
  private String id; Course[] cs; int noc; /* # of courses */
  public Student(String id) { ... }
  public void addCourse(Course c) { ... }
  public Course[] getCS() { ... }
}
```

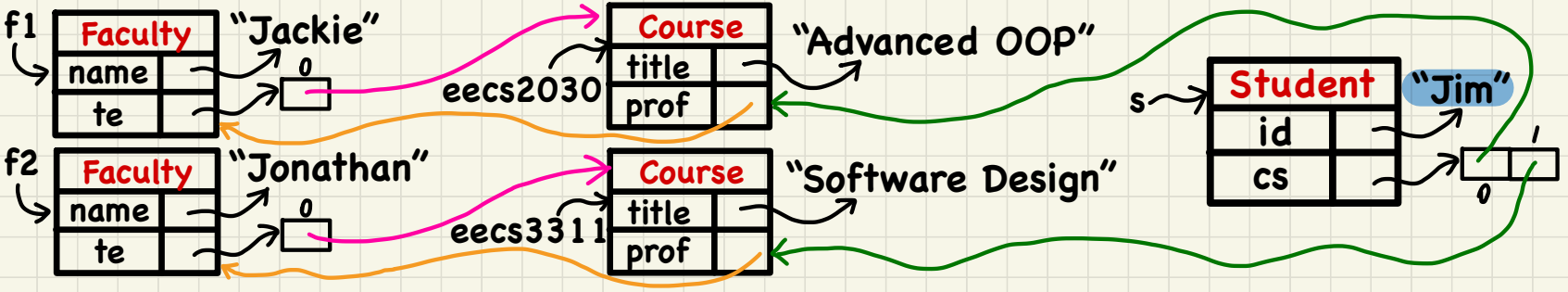
```java
public class Course { private String title; private Faculty prof; }
```

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

```java
@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]);
}
```

Runtime Object Structure: Student, Course, Faculty



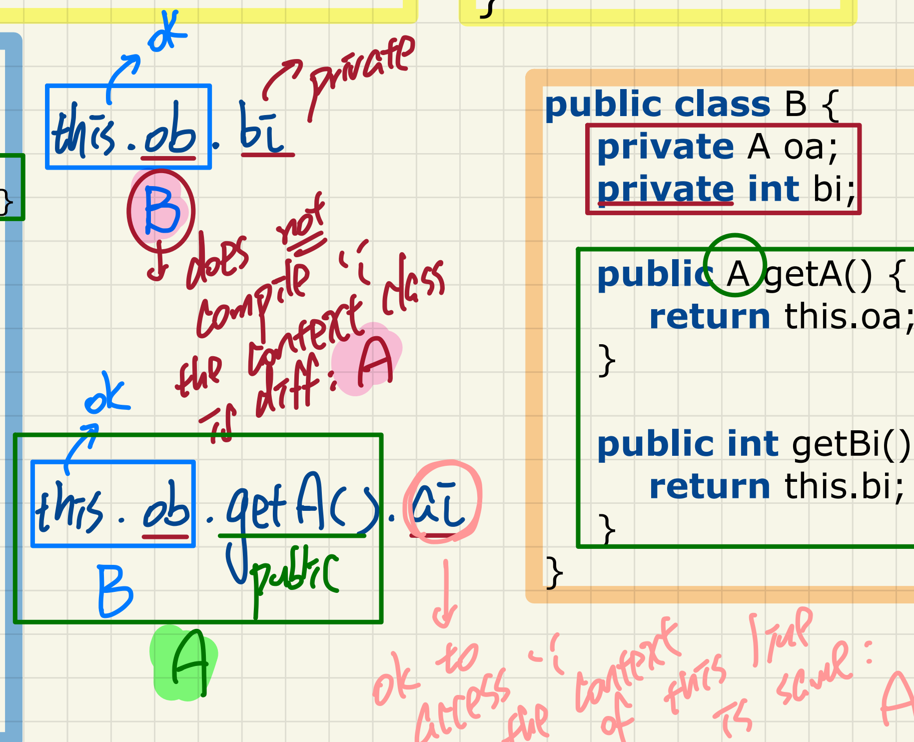


Dot Notation: Private Attributes/Fields

Principle: **Private** attribute is accessible if the **context object**'s type matches the **context class** (where the method is defined).

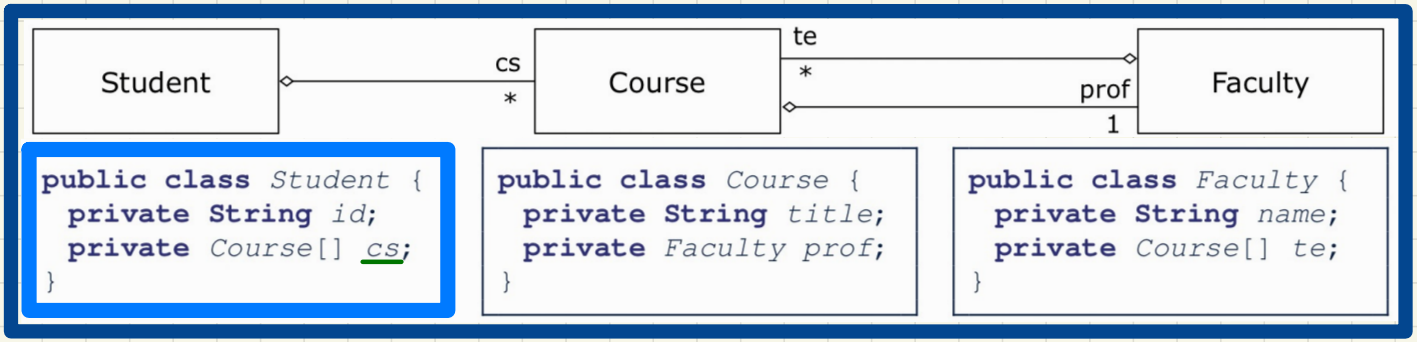
```
class X {
  ...
  exp.privateAtt
  ...
}
```

```
public class A {
  private B ob;
  private int ai;
  public B getB() { return this.ob; }
  public int getAi() { return this.ai; }
  public int am() {
    int result;
    result = this.ai;
    result = this.getAi();
    result = this.ob.bi;
    result = this.getB().bi;
    result = this.ob.getBi();
    result = this.getB().getBi();
    result = this.ob.getA().ai;
    result = this.ob.getA().getAi();
    result = this.ob.oa.ai;
    result = this.ob.oa.getAi();
    return result;
  }
}
```




```
public class B {
  private A oa;
  private int bi;

  public A getA() {
    return this.oa;
  }

  public int getBi() {
    return this.bi;
  }
}
```

Dot Notation for Navigating Classes (1)



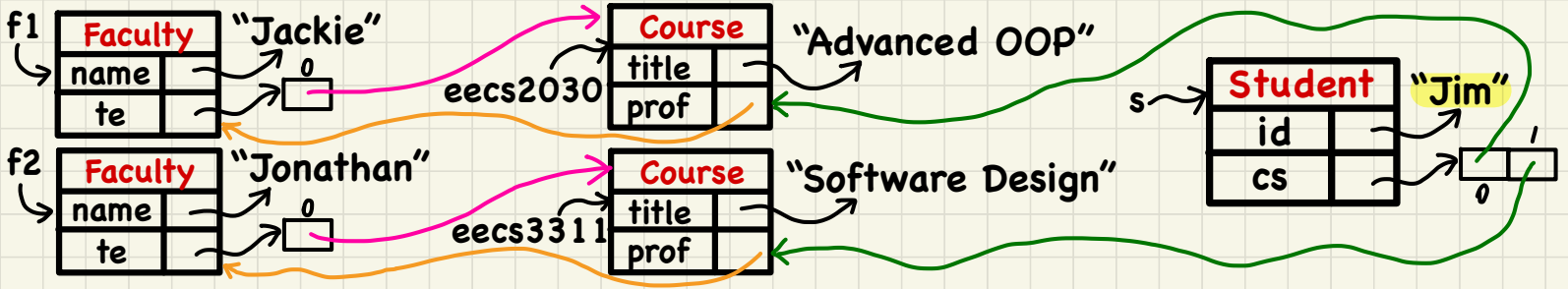

```
/* Get the student's id.
 */
String getID() {
    this.id
}
```

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

title ?

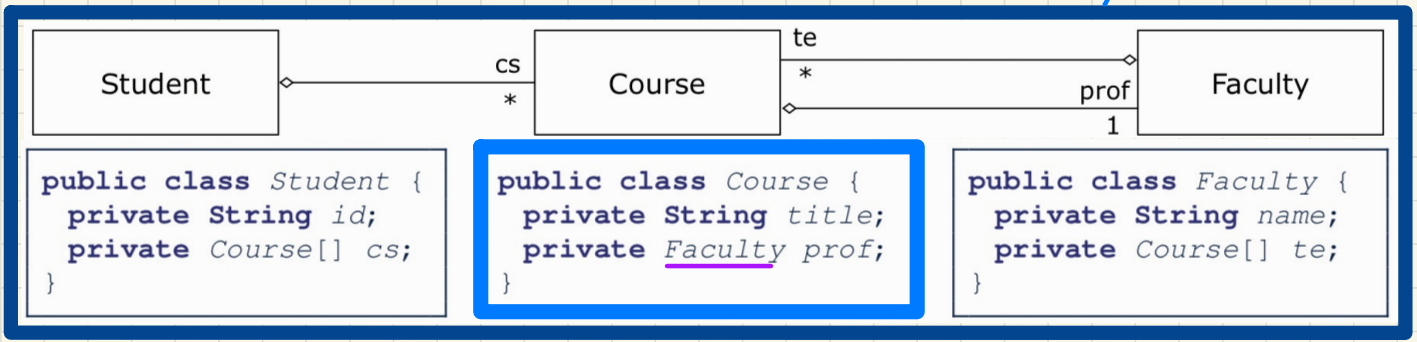
```
/* Name of
 * ith course's instructor
 */
String getName(int i) {
    this.cs[i].getProf().getName()
}
```

prof ? name ?



Dot Notation for Navigating Classes (2)

this.prof.te[i].title

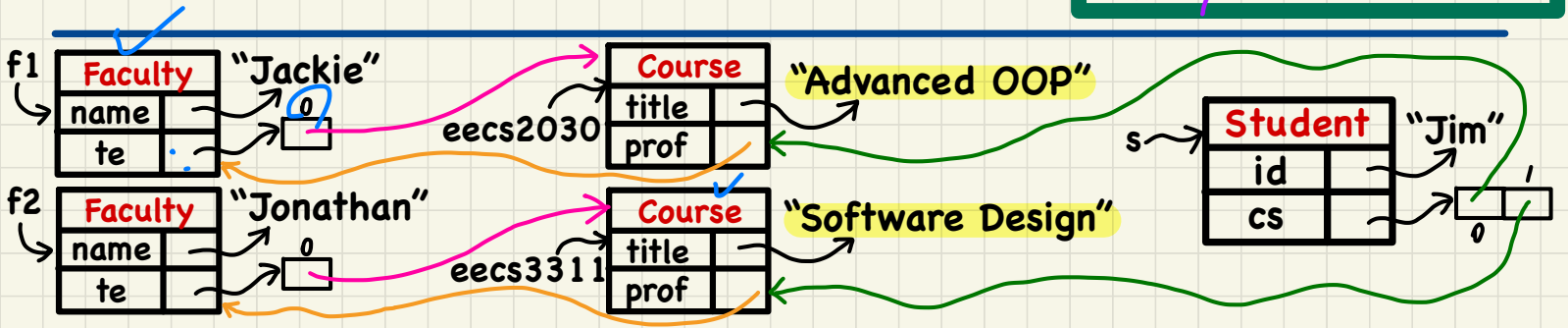



```
/* Get course's title.
 */
String getTitle() {
    this.title
}
```

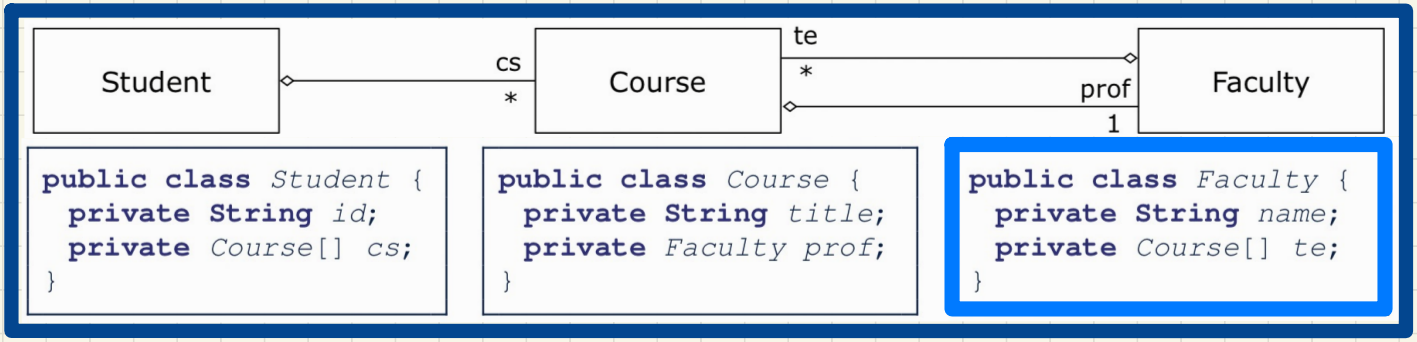
```
/* Name of instructor
 */
String getName() {
    this.prof.getName()
}
```

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

Faculty · te? · title?



Dot Notation for Navigating Classes (3) Exercise



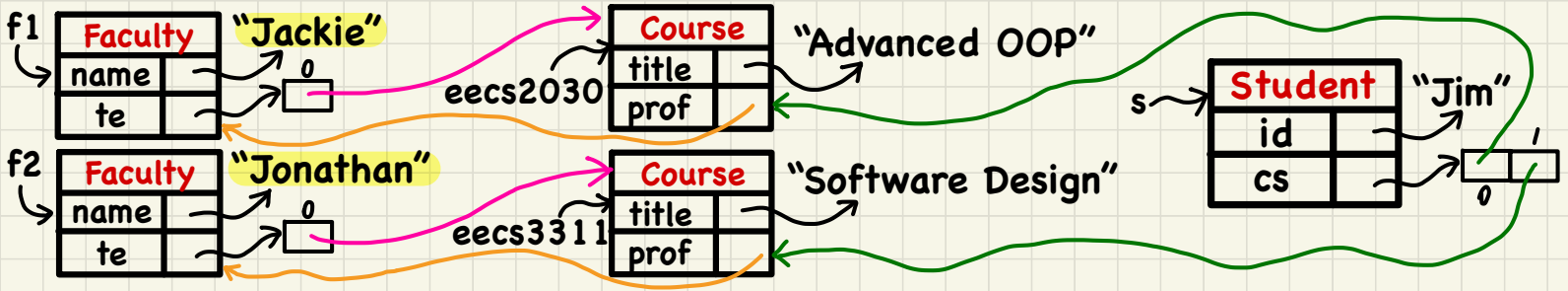

```
public class Student {
  private String id;
  private Course[] cs;
}
```

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

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

```
/* Name of instructor
 */
String getName() {


}
```

```
/* Title of instructor's
 * ith teaching course
 */
String getTitle(int i) {


}
```



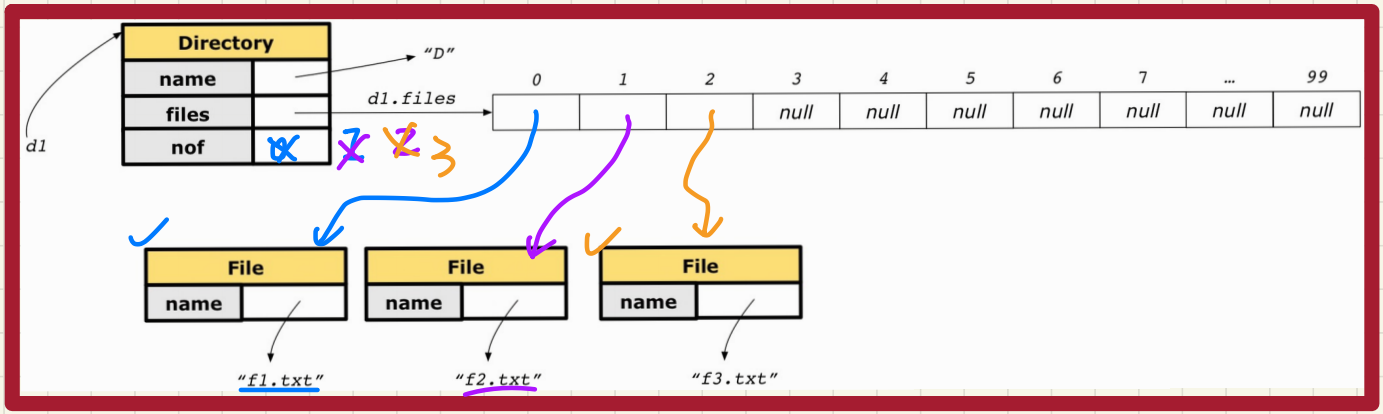
Composition: No Sharing

```
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(String name) {
    this.name = name;
    files = new File[100];
  }
  void addFile(String fileName) {
    files[nof] = new File(fileName);
    nof ++;
  }
}
```

```
class File {
  String name;
  File(String name) {
    this.name = name;
  }
}
```

```
public File[] getFiles() {
  return this.files;
}
```

```
1  @Test
2  public void testComposition() {
3    Directory d1 = new Directory("D");
4    d1.addFile("f1.txt");
5    d1.addFile("f2.txt");
6    d1.addFile("f3.txt");
7    assertTrue(
8      d1.files[0].name.equals("f1.txt"))
9  }
```



Composition: Copy Constructor (Shallow Copy)

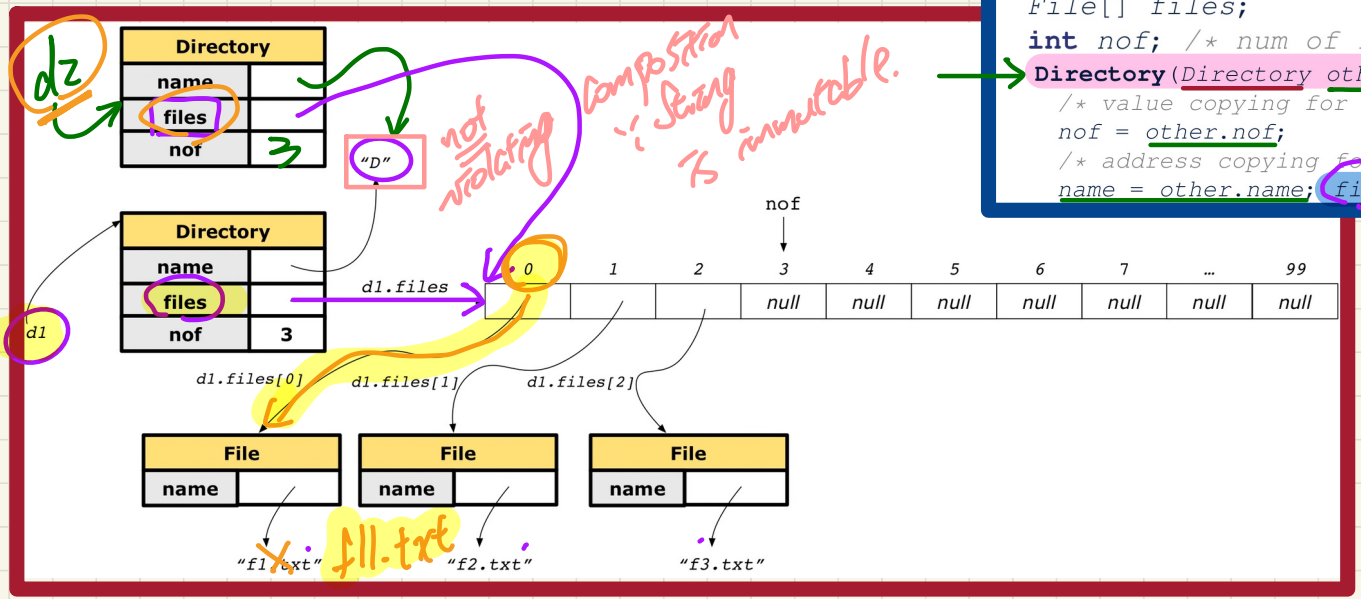
```
@Test
public void testShallowCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt"
  Directory d2 = new Directory(d1);
  assertTrue(d1.files == d2.files);
  d2.files[0].changeName("f11.txt");
  assertFalse(d1.files[0].name.equals("f1.txt")); }
```

also modifies a file shared by d1.

```
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(Directory other) {
    /* value copying for primitive type */
    nof = other.nof;
    /* address copying for reference type */
    name = other.name; files = other.files; }
```

d1

shallow copy: just copy the ref. of the array.



public class X {

copy constructor.

public X (X other) {

}

Lecture 16 - Nov. 5

Composition, Inheritance

Composition: Deep Copy
Design Attempts without Inheritance
Inheritance: Use of extends, super

Announcements/Reminders

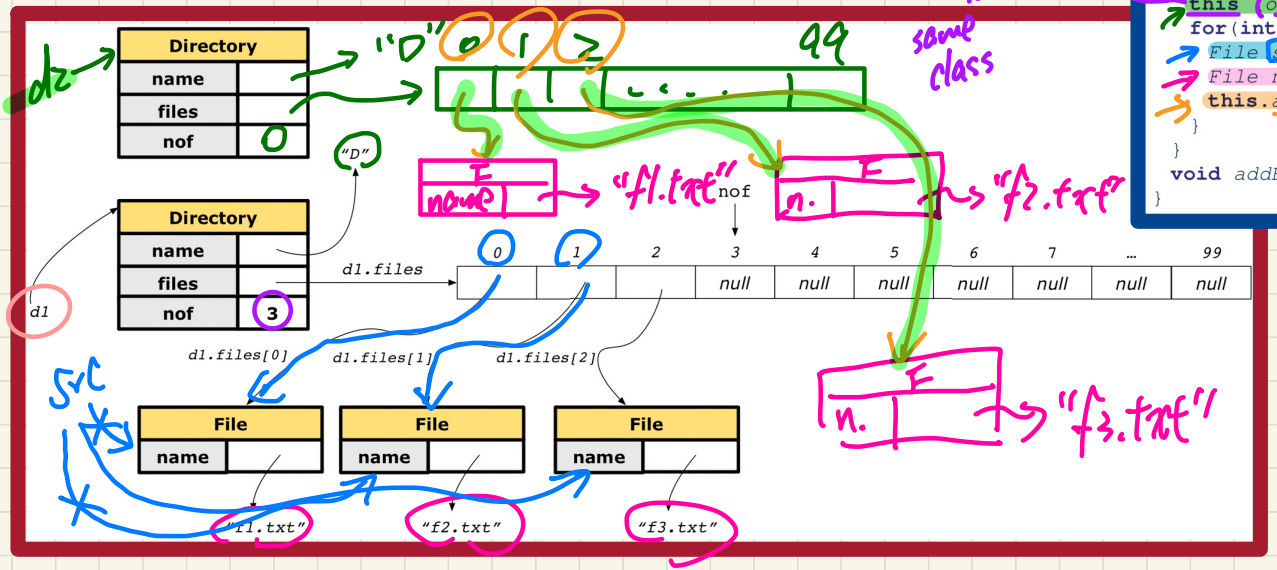
- Lab4 released (ProgTest3 on November 20)
- Guide & Questions for WrittenTest2 to be released tmw
- In-Lab Demo on Inheritance tomorrow
- ProgTest2 results & feedback tentatively Monday Nov 11

Composition: Copy Constructor (Deep Copy)

```
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.files != d2.files);
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0].name.equals("f1.txt")); }
```

```
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this(other.name);
    for(int i = 0; i < other.nof; i ++) {
      File src = other.files[i];
      File nf = new File(src);
      this.addFile(nf);
    }
  }
  void addFile(File f) { ... }
}
```



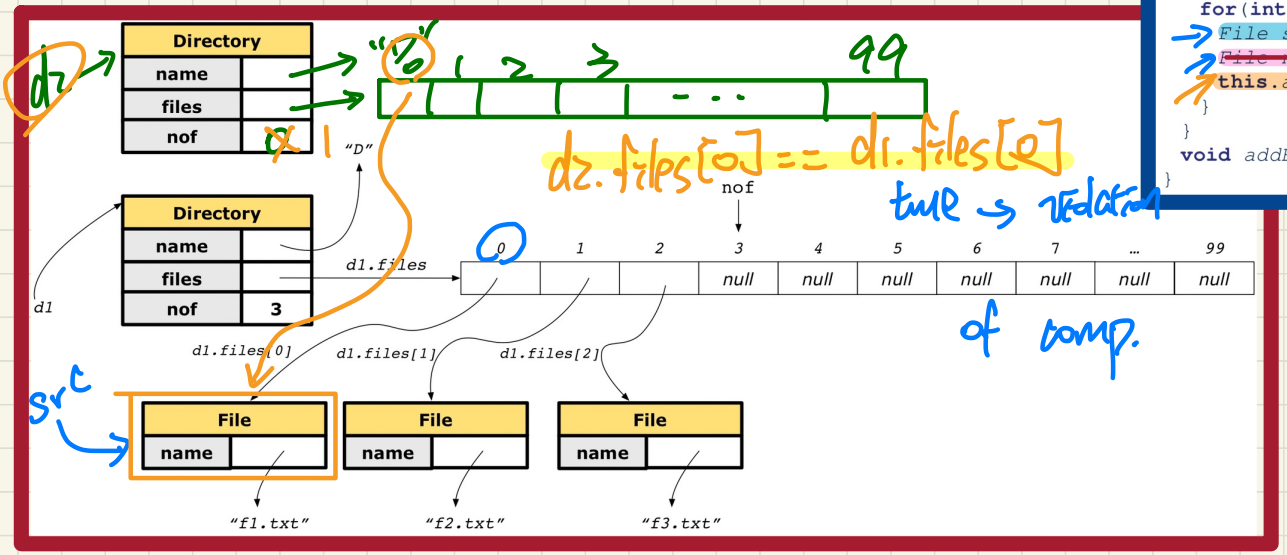
Exercise: Copy Constructor (Composition?)

```
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt"
  Directory d2 = new Directory(d1);
  assertTrue(d1.files != d2.files);
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0] == d2.files[0]);
```

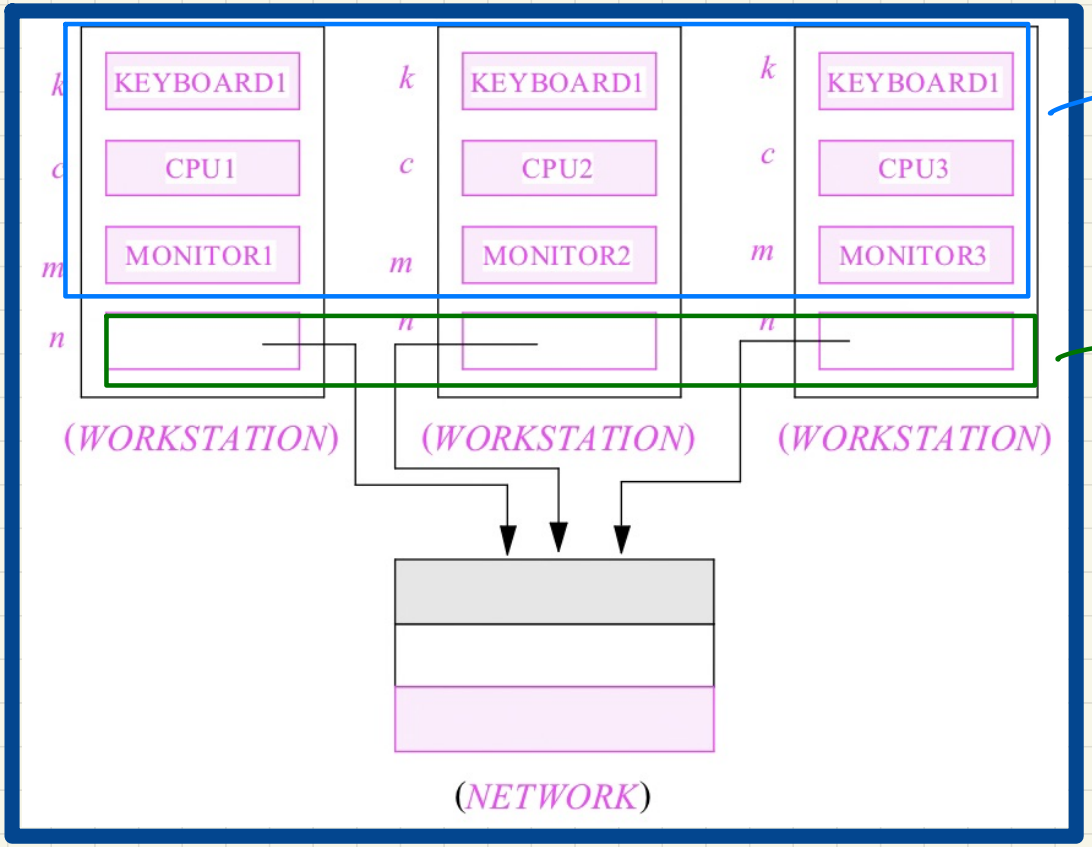
```
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this (other.name);
    for(int i = 0; i < other.nof; i ++) {
      File src = other.files[i];
      File nf = new File(src);
      this.addFile(nf);
    }
  }
  void addFile(File f) { ... }
}
```

src



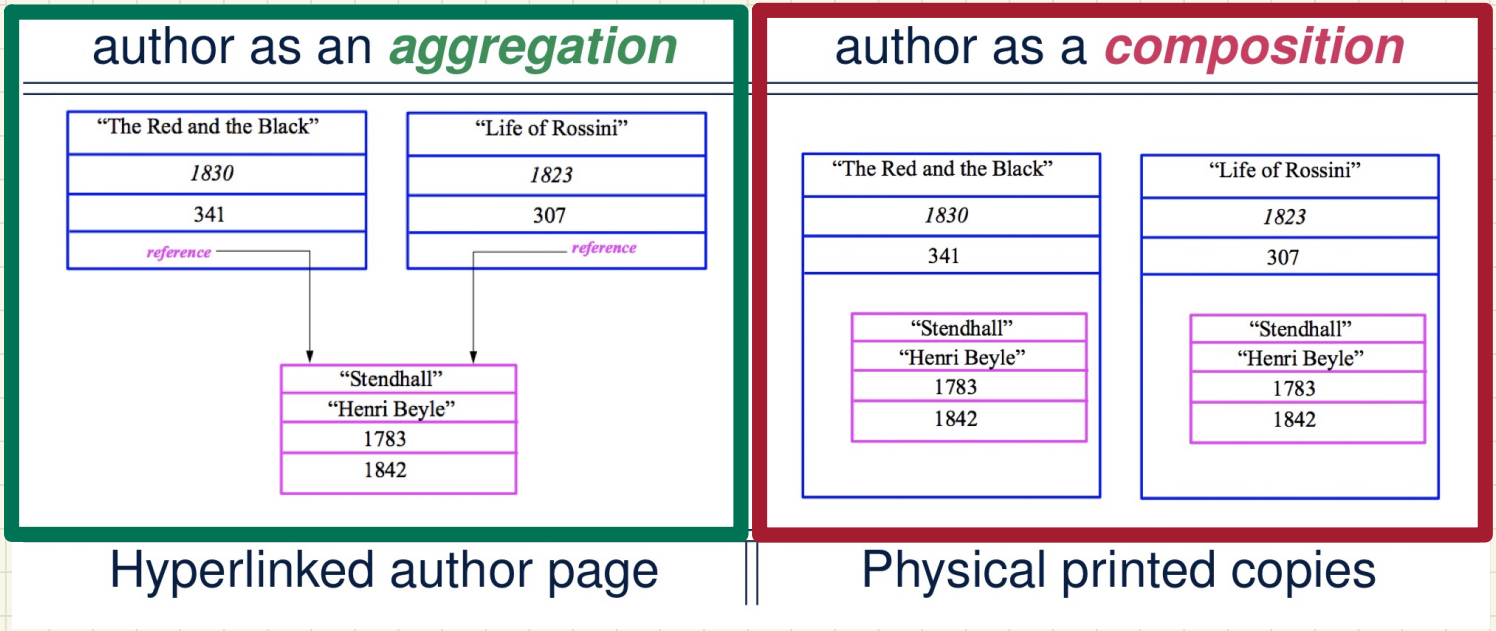
Modelling: Aggregation vs. Composition



composition

aggregation

Implementation: Aggregation or Composition



Inheritance: Motivating Problem

Nouns -> classes, attributes, accessors
Verbs -> mutators

Problem: A student management system stores data about students. There are two kinds of university students: resident students and non-resident students. Both kinds of students have a name and a list of registered courses. Both kinds of students are restricted to register for no more than 10 courses. When calculating the tuition for a student, a base amount is first determined from the list of courses they are currently registered (each course has an associated fee). For a non-resident student, there is a discount rate applied to the base amount to waive the fee for on-campus accommodation. For a resident student, there is a premium rate applied to the base amount to account for the fee for on-campus accommodation and meals.

First Design Attempt

```
public class Student {
   private Course[] courses;
   private int noc;

   private int kind;
   private double premiumRate;
   private double discountRate;

   public Student (int kind){
      this.kind = kind;
   }
   ...
}
```

1 → RS
0 → NRS

rs1.register(2020);
nrs1.register(2020);

```
Student rs1 = new Student (1);
Student nrs1 = new Student (0);
```

```
public double getTuition(){
   double tuition = 0;
   for(int i = 0; i < this.noc; i++){
      tuition += this.courses[i].fee;
   }
   if (this.kind == 1) {
      return tuition * this. premiumRate;
   }
   else if (this.kind == 0) {
      return tuition * this.discountRate;
   }
}
```

```
public void register(Course c){
   int MAX = -1;
   if (this.kind == 1) { MAX = 6; }
   else if (this.kind == 0) { MAX = 4; }
   if (this.noc == MAX) { /* Error */ }
   else {
      this.courses[this.noc] = c;
      this.noc ++;
   }
}
```

First Design Attempt

```
public class Student {
   private Course[] courses;
   private int noc;

   private int kind;
   private double premiumRate;
   private double discountRate;

   public Student (int kind){
      this.kind = kind;
   }
   ...
}
```

violation of cohesion

```
public double getTuition(){
   double tuition = 0;
   for(int i = 0; i < this.noc; i++){
      tuition += this.courses[i].fee;
   }
   if (this.kind == 1) {
      return tuition * this. premiumRate;
   }
   else if (this.kind == 2) {
      return tuition * this.discountRate;
   }
}
```

```
public void register(Course c){
   int MAX = -1;
   if (this.kind == 1) { MAX = 6; }
   else if (this.kind == 2) { MAX = 4; }
   if (this.noc == MAX) { /* Error */ }
   else {
      this.courses[this.noc] = c;
      this.noc ++;
   }
}
```

Good design?

Judge by **Cohesion**

attributes and methods in the same class should be under a unifying theme.

First Design Attempt

```
public class Student {
   private Course[] courses;
   private int noc;

   private int kind;
   private double premiumRate;
   private double discountRate;

   public Student (int kind){
      this.kind = kind;
   }
   ...
}
```

```
public double getTuition(){
   double tuition = 0;
   for(int i = 0; i < this.noc; i++){
      tuition += this.courses[i].fee;
   }
   if (this.kind == 1) {
      return tuition * this. premiumRate;
   }
   else if (this.kind == 0) {
      return tuition * this.discountRate;
   }
}
```

{else if (this.kind == 2) { ... }

```
public void register(Course c){
   int MAX = -1;
   if (this.kind == 1) { MAX = 6; }
   else if (this.kind == 0) { MAX = 4; }
   if (this.noc == MAX) { /* Error */ }
   else {
      this.courses[this.noc] = c;
      this.noc ++;
   }
}
```

else if (this.kind == 2) { ... }

Good design?

Judge by Single Choice Principle

- Repeated if-conditions
- A new kind is introduced? → e.g. international student
- An existing kind is obselete?

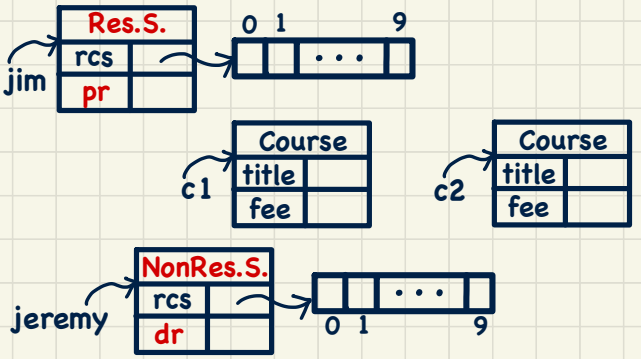
(kind == 2)

Testing Student Classes (without inheritance)

```java
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a m
  public ResidentStudent(String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc] = c;
    this.noc ++;
  }
  public double getTuition() {
    double tuition = 0;
    for(int i = 0; i < this.noc; i ++) {
      tuition += this.courses[i].fee;
    }
    return tuition * this.premiumRate;
  }
}
```

```java
public class NonResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double discountRate; /* assume a
  public NonResidentStudent(String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc] = c;
    this.noc ++;
  }
  public double getTuition() {
    double tuition = 0;
    for(int i = 0; i < this.noc; i ++) {
      tuition += this.courses[i].fee;
    }
    return tuition * this.discountRate;
  }
}
```

```java
public class StudentTester {
  public static void main(String[] args) {
    Course c1 = new Course("EECS2030", 500.00); /* title and fee */
    Course c2 = new Course("EECS3311", 500.00); /* title and fee */
    ResidentStudent jim = new ResidentStudent("J. Davis");
    jim.setPremiumRate(1.25);
    jim.register(c1); jim.register(c2);
    NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons")
    jeremy.setDiscountRate(0.75);
    jeremy.register(c1); jeremy.register(c2);
    System.out.println("Jim pays " + jim.getTuition());
    System.out.println("Jeremy pays " + jeremy.getTuition());
  }
}
```



Student Classes (with inheritance)

```
class Student {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  Student (String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[i].fee;
    }
    return tuition; /* base amount only */
  }
}
```

parent class / super class

* new attributes declared in individual child classes.

** super(...) invokes the constructor from the super class

*** super.someMethod(...) invokes the version of method in super class.

getTuition is overridden in RS

overridden in NRS

code reuse: everything declared in Student is inherited to RS

child class / sub class

```
class ResidentStudent extends Student {
  double premiumRate; /* there's a mutator meth
  ResidentStudent (String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * premiumRate;
  }
}
```

```
class NonResidentStudent extends Student {
  double discountRate; /* there's a mutator method
  NonResidentStudent (String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * discountRate;
  }
}
```

In-Lab Demo - Nov. 6

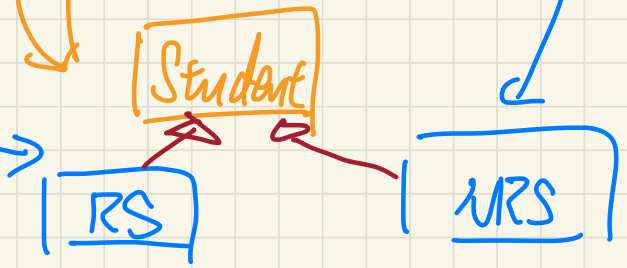
Programming with Inheritance

extends, equals
Visualizing Child Objects
Tracing Method Calls in Eclipse

Student Classes (without inheritance)

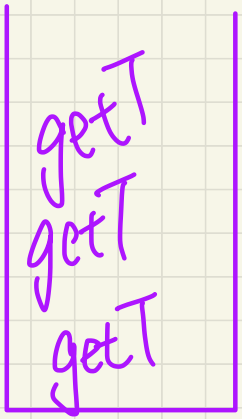
```java
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a m
  public ResidentStudent(String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc] = c;
    this.noc ++;
  }
  public double getTuition() {
    double tuition = 0;
    for(int i = 0; i < this.noc; i ++) {
      tuition += this.courses[i].fee;
    }
    return tuition * this.premiumRate;
  }
}
```

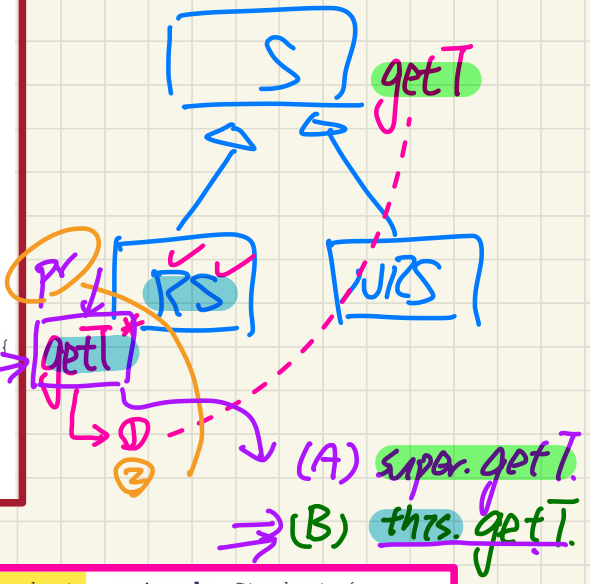
```java
public class NonResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double discountRate; /* assume a
  public NonResidentStudent(String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc] = c;
    this.noc ++;
  }
  public double getTuition() {
    double tuition = 0;
    for(int i = 0; i < this.noc; i ++) {
      tuition += this.courses[i].fee;
    }
    return tuition * this.discountRate;
  }
}
```



Recall: Student Classes (with inheritance)

```java
class Student {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  Student(String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[i].fee;
    }
    return tuition; /* base amount only */
  }
}
```






```java
class ResidentStudent extends Student {
  double premiumRate; /* there's a mutator meth
  ResidentStudent(String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * premiumRate;
  }
}
```

```java
class NonResidentStudent extends Student {
  double discountRate; /* there's a mutator method
  NonResidentStudent(String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * discountRate;
  }
}
```

Lecture 17 - Nov. 7

Inheritance

Implementing a Child Class: Principles
Visibility: Class, Attribute/Method
Static Types: Expectations
Polymorphism: Intuition

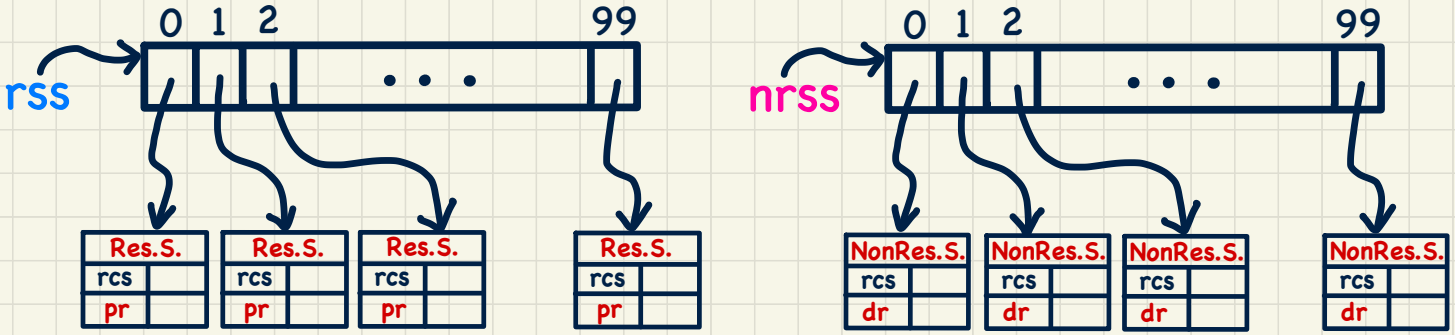
Announcements/Reminders

- Lab4 released (ProgTest3 on November 20)
- Guide & Questions for WrittenTest2 released
- Materials for In-Lab Demo on Inheritance released
- ProgTest2 results & feedback Monday Nov 11

A Collection of Students (without inheritance)

```
public class StudentManagementSystem {
  private ResidentStudent[] rss;
  private NonResidentStudent[] nrss;
  private int nors; /* number of resident students */
  private int nonrs; /* number of non-resident students */
  public void addRS(ResidentStudent rs){ rss[nors]=rs; nors++; }
  public void addNRS(NonResidentStudent nrs){ nrss[nonrs]=nrs;nonrs++; }
  public void registerAll(Course c) {
    for(int i = 0; i < nors; i ++) { rss[i].register(c); }
    for(int i = 0; i < nonrs; i ++) { nrss[i].register(c); }
  }
}
```

loops correspond to # of kinds of students.



Recall: Student Classes (with inheritance)

(4) inherited methods, if re-declared, are overridden to a new version

X String names

When writing a new subclass:

(1) cannot re-declare inherited attributes

(2) add methods/attributes unique to subclasses

(3) inherited methods, if not re-declared, will just remain available

```
class Student {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  Student(String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[i].fee;
    }
    return tuition; /* base amount only */
  }
}
```

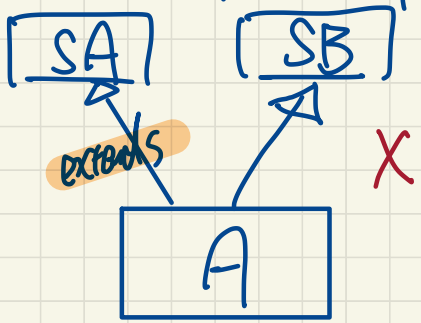
```
class ResidentStudent extends Student {
  double premiumRate;
  ResidentStudent(String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * premiumRate;
  }
}
```

setR

```
class NonResidentStudent extends Student {
  double discountRate;
  NonResidentStudent(String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * discountRate;
  }
}
```

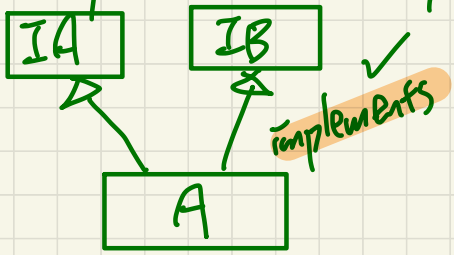
setR

Multiple inheritance is forbidden in most of OOP languages

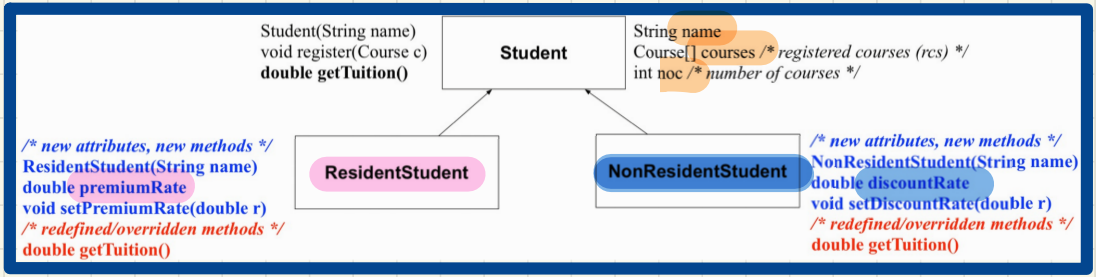


(diamond problem)

A class may implement multiple interfaces



Visualizing Parent and Child Objects

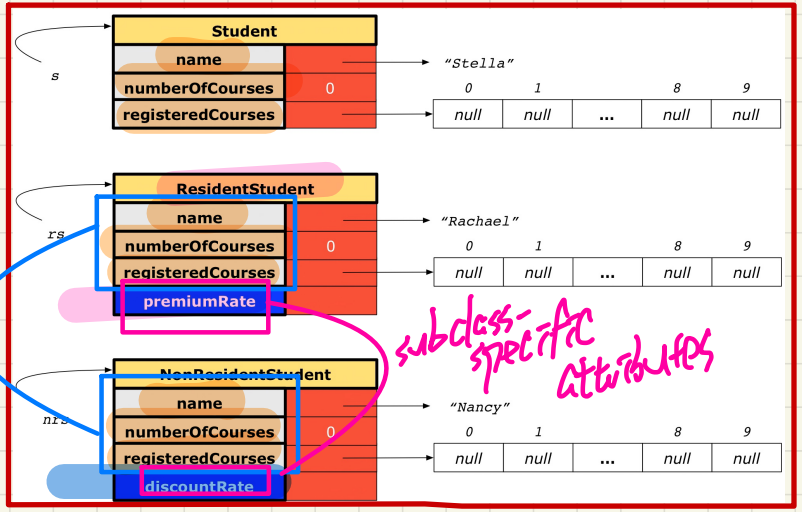


Inheritance Hirarchy

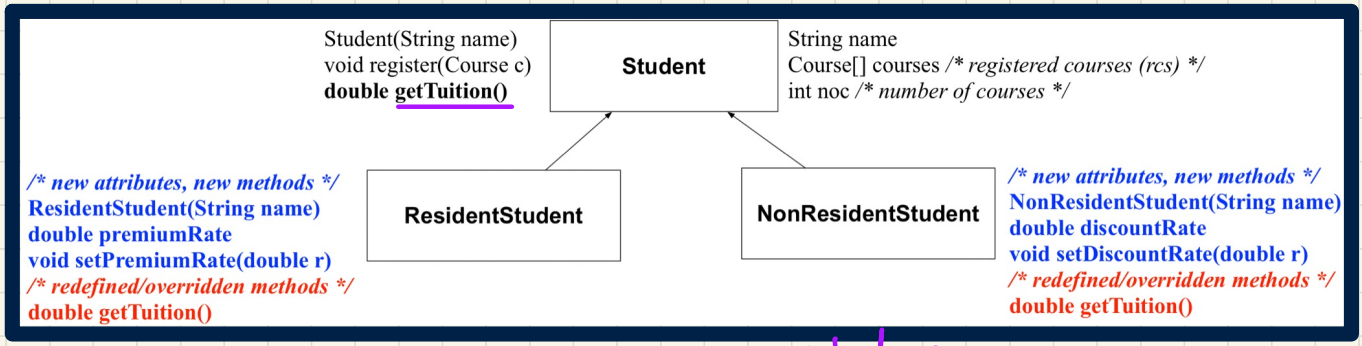
```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

Declaring Static Types

Runtime Object Structure



Testing Student Classes (with inheritance)



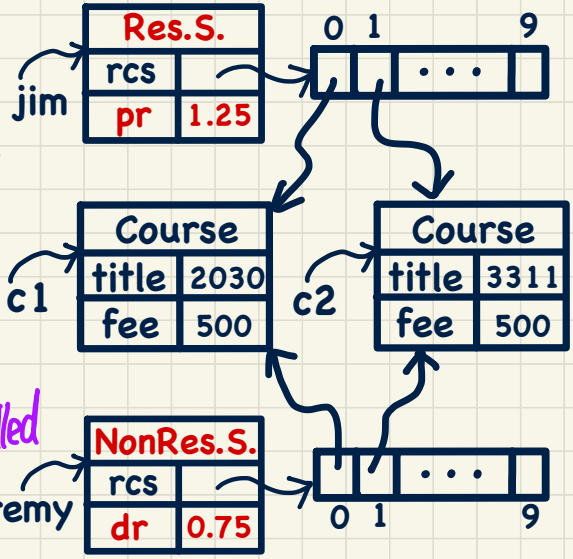
in-lab demo

```
public class StudentTester {
  public static void main(String[] args) {
    Course c1 = new Course("EECS2030", 500.00); /* title and fee */
    Course c2 = new Course("EECS3311", 500.00); /* title and fee */
    ResidentStudent jim = new ResidentStudent("J. Davis");
    jim.setPremiumRate(1.25);
    jim.register(c1); jim.register(c2);
    NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons");
    jeremy.setDiscountRate(0.75);
    jeremy.register(c1); jeremy.register(c2);
    System.out.println("Jim pays " + jim.getTuition());
    System.out.println("Jeremy pays " + jeremy.getTuition());
  }
}
```

invoke const. from Student

2. both reuse getT from Student class

1. diff versions of getT called

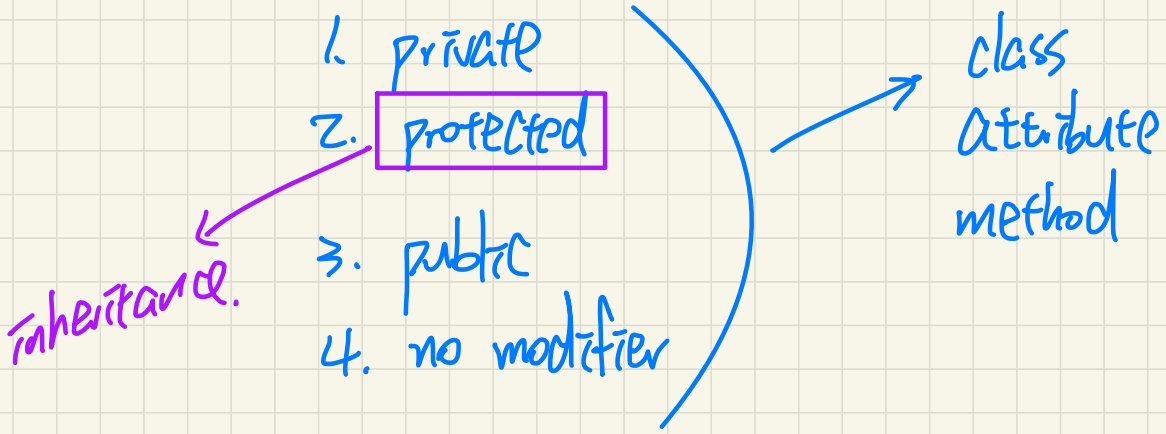


Modifiers

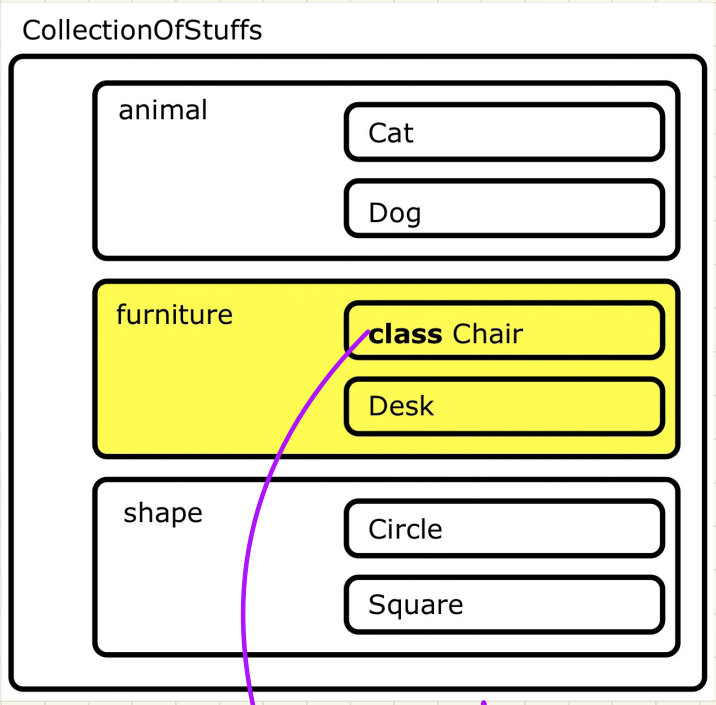
1. private
2. protected
3. public
4. no modifier

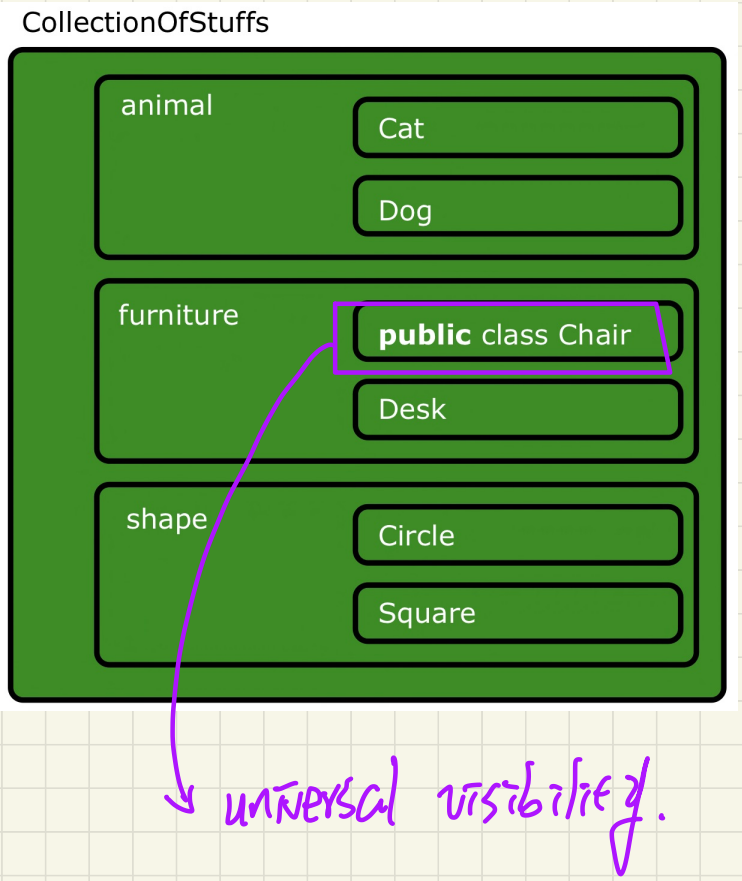
inheritance.

class
attribute
method

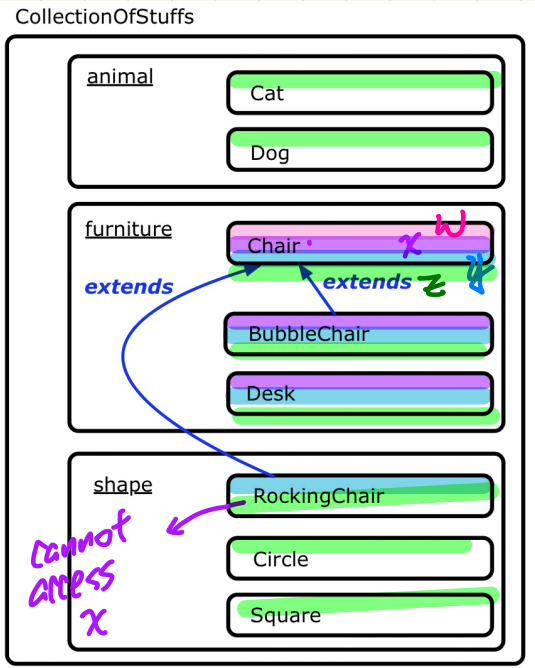


Visibility: Classes





Visibility: Attributes and Methods




```
public class Chair {
   private int w;
   int x;
   protected int y;
   public int z;
}
```

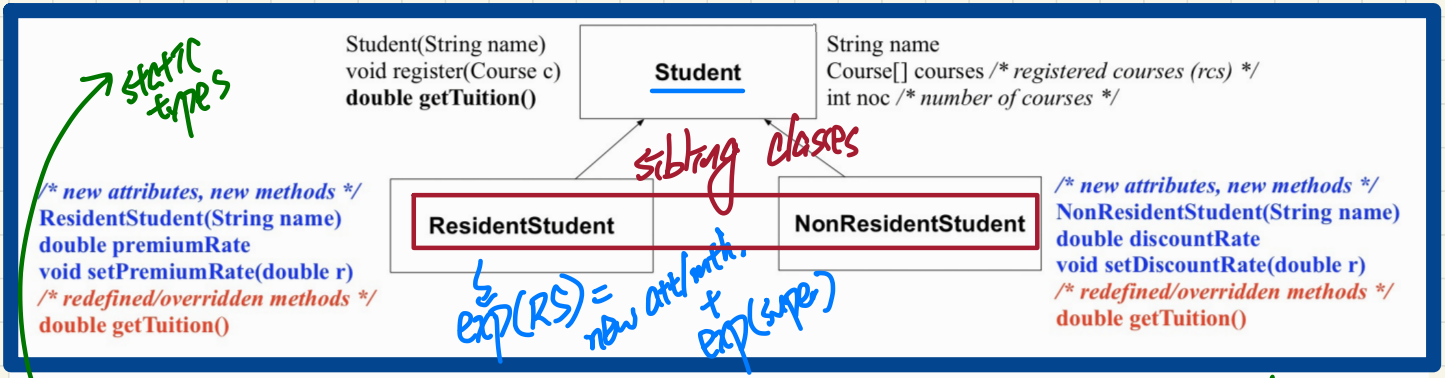
not inherited to sub-classes.

as if:
① no modifier
+
② subclasses in diff. package.

	Class	Package	Subclass (same pkg)	Subclass (different pkg)	Non-Subclass (across Project)
public					
protected					
no modifier	✓	✓	✓		
private					

Student Classes (with inheritance): Expectations

what attrs/methods can be called on a ref. variable?



if exp. not met
↳ compilation errors

```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

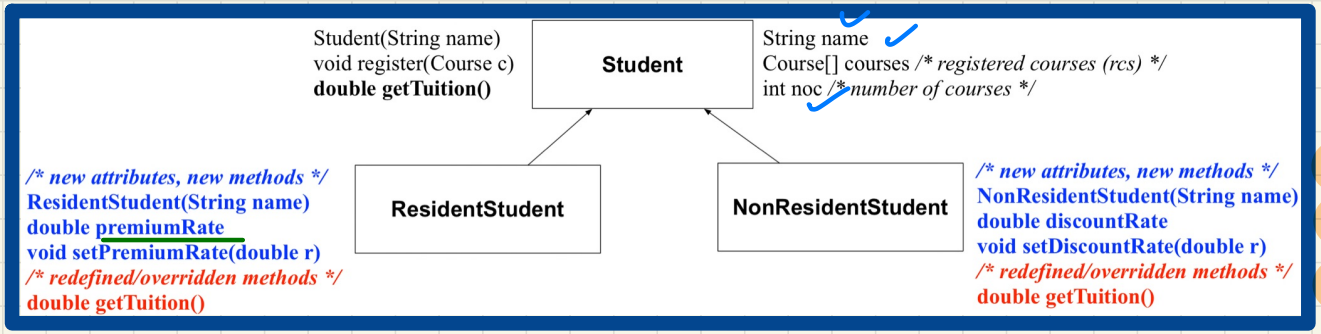
inherited exp. from parent.

siblings exp. not relevant.

	name	rcs	noc	reg	getT	pr	setPR	dr	setDR
s.									
rs.									
nrs.									

new exp specific to sub-classes

Intuition: Polymorphism



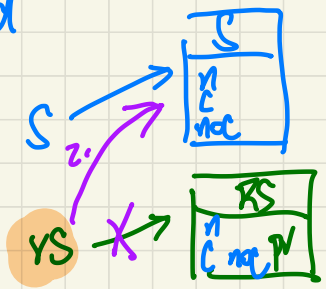

```
1  Student s = new Student("Stella");
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.25);
4  s = rs; /* Is this valid? */
5  rs = s; /* Is this valid? */
```

ST: parent

ST: child

1. Assume rs = s compiled

2. Execute the re-assignment



3. exp. of rs

rs.n
rs.c
rs.noc
rs.pr

but dynamically rs points to a Student,

which does not have pr

↳ crash!

Back to step 1: rs = s should not compile

Written Test 2

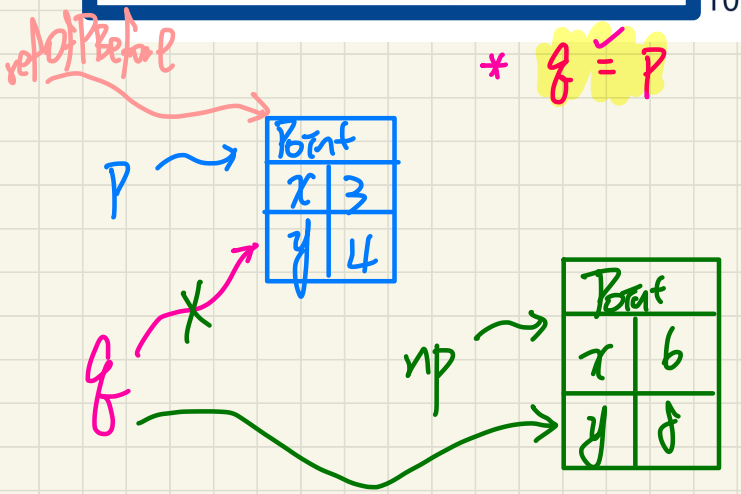
Review Q&A

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

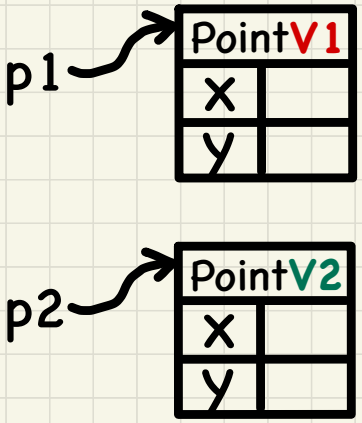
```java
@Test
public void testEqualityOfPointV1andPointv2() {
  PointV1 p1 = new PointV1(3, 4); PointV2 p2 = new PointV2(3, 4);
  /* These two assertions do not compile because p1 and p2 are of different types. */
  /* assertFalse(p1 == p2); assertFalse(p2 == p1); */
  /* assertSame can take objects 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)

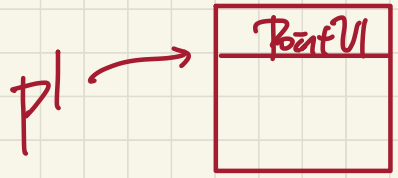
==

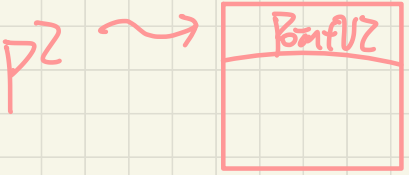
(obj2 == obj1)

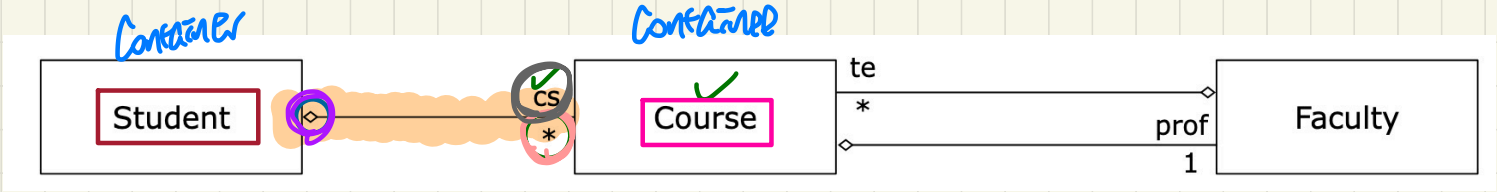
true.

(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

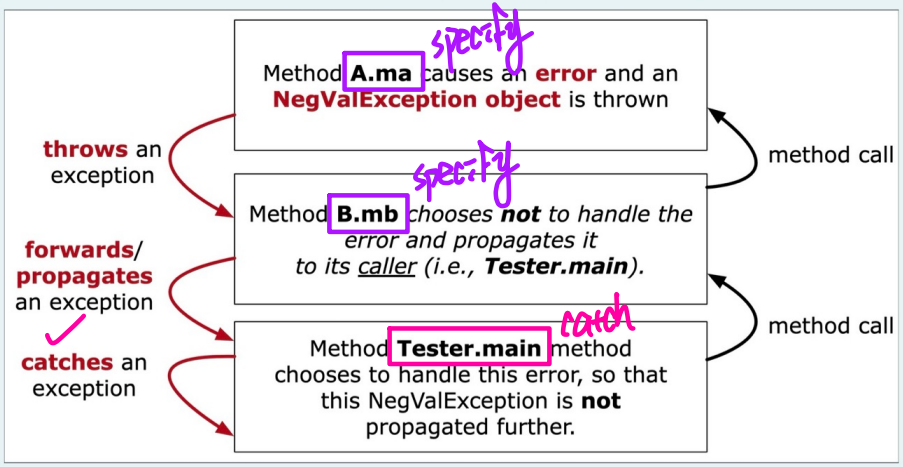
Person

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

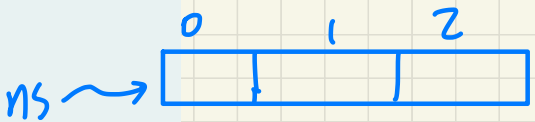
Assume a non-empty integer array **ns** of length 3 and an integer variable **i**.

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

Lecture 18 - Nov. 12

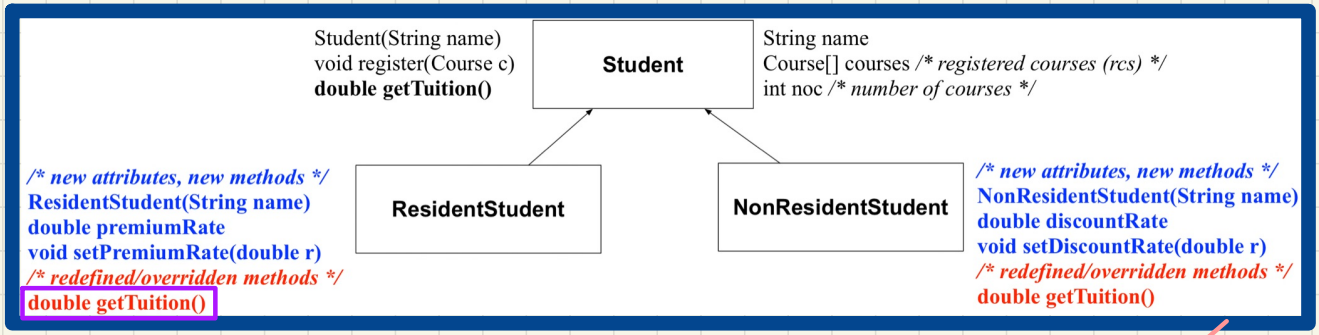
Inheritance

Dynamic Binding: Intuition
Ancestors vs Descendants, Expectation
Variable Substitutions

Announcements/Reminders

- **WrittenTest2** tomorrow
- **Lab4** due this Friday at noon
- **ProgTest3** next Wednesday, November 20
- **ProgTest2** results & feedback released

Intuition: Dynamic Binding




```
1  Course eecs2030 = new Course("EECS2030", 100.0);
2  Student s;
3  ResidentStudent rs = new ResidentStudent("Rachel");
4  NonResidentStudent nrs = new NonResidentStudent("Nancy");
5  rs.setPremiumRate(1.25); rs.register(eecs2030);
6  nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7  s = rs; System.out.println(s.getTuition());
8  s = nrs; System.out.println(s.getTuition());
```

ST of S

DT

DT

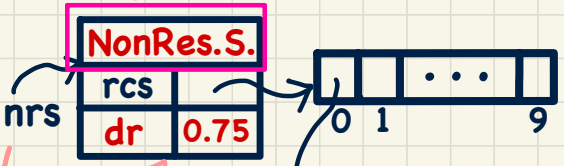
DT: RS

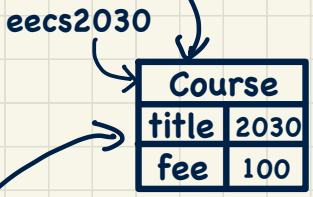
RS version called

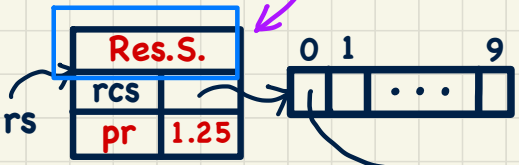
NRS version called

DT: NRS

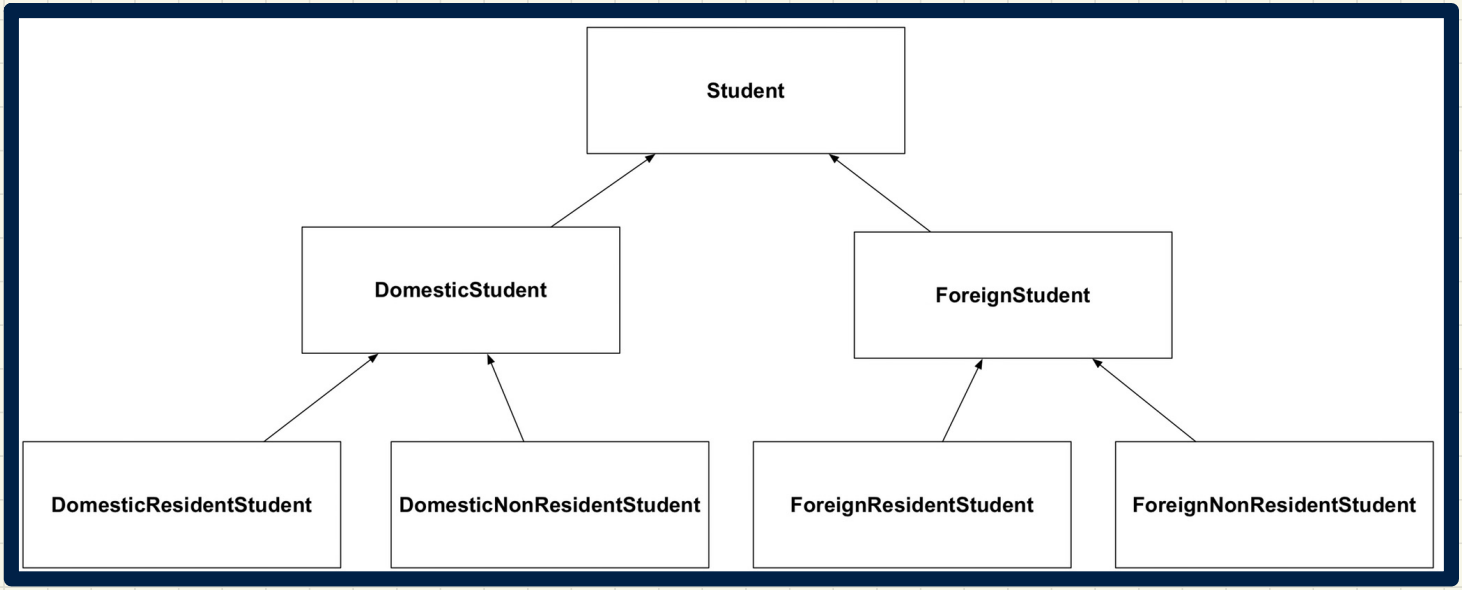
	ST of S	DT of S
Before ①	Student	null
After ①		RS
Before ②		RS
After ②		NRS







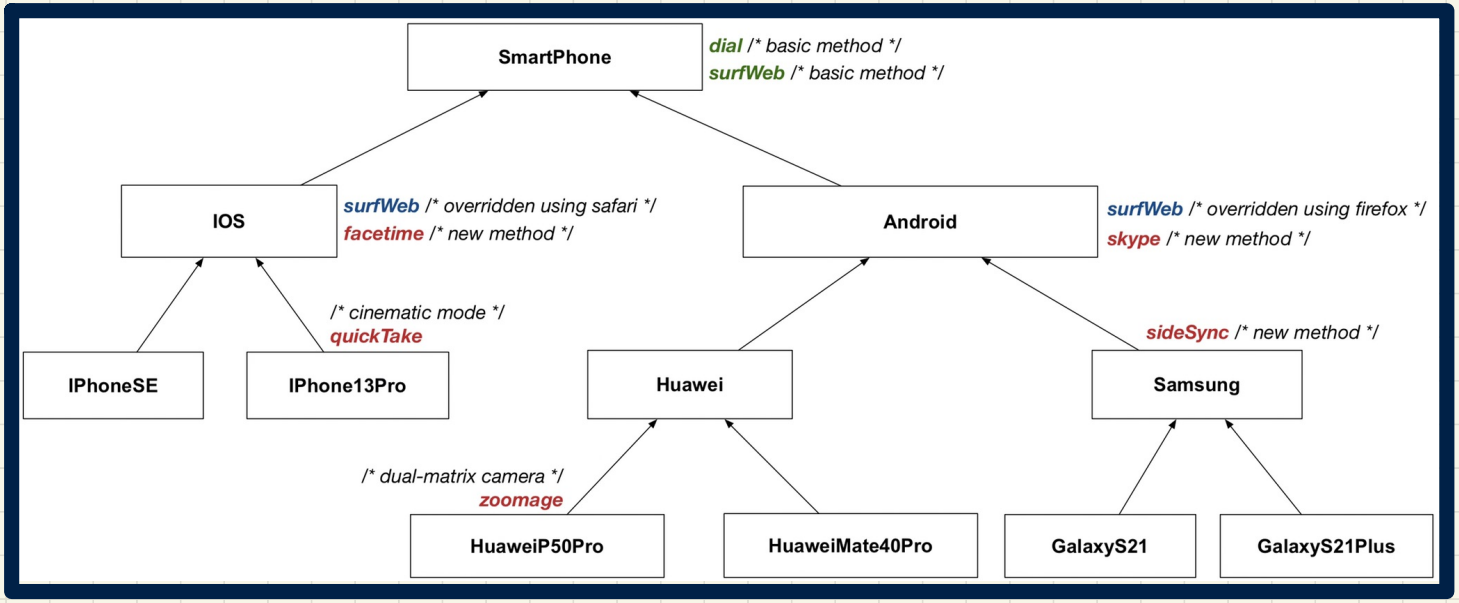
Multi-Level Inheritance Hierarchy: Students



Reflections:

- For **Design 1**, how many encodings to check for each method?
- For **Design 2**, how many arrays to store for SMS?
- For **Design 3**, where are common attributes/methods stored?

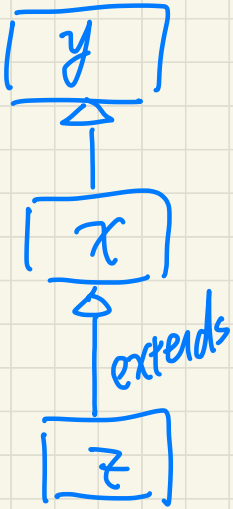
Multi-Level Inheritance Hierarchy: Smartphones



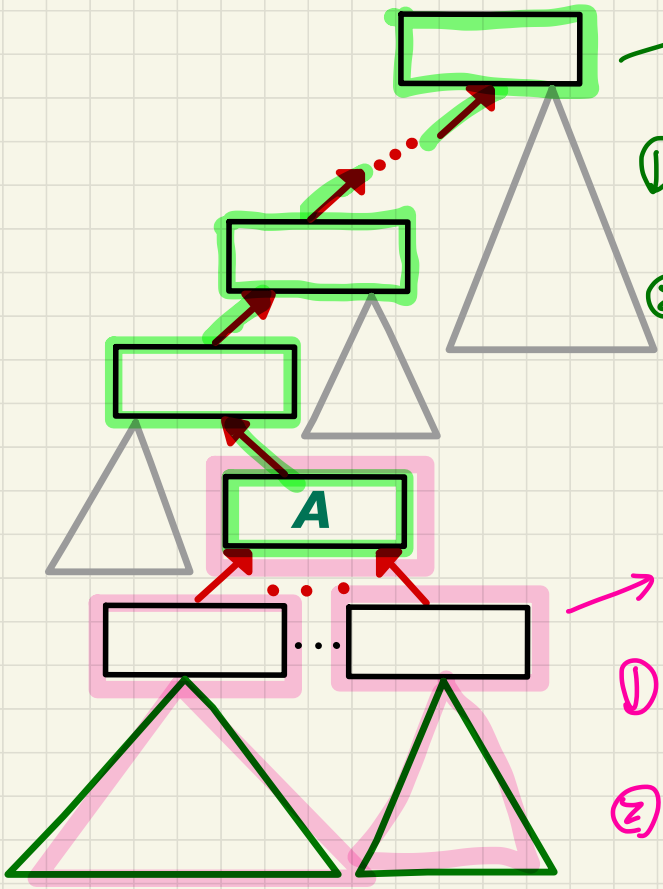
Reflections:

- For Design 1, how many encodings to check for each method?
- For Design 2, how many arrays to store for SMS?
- For Design 3, where are common attributes/methods stored?

Inheritance Forms a Type Hierarchy



z indirectly extends y



→ ancestors of A

① union of all attributes inherited to A.

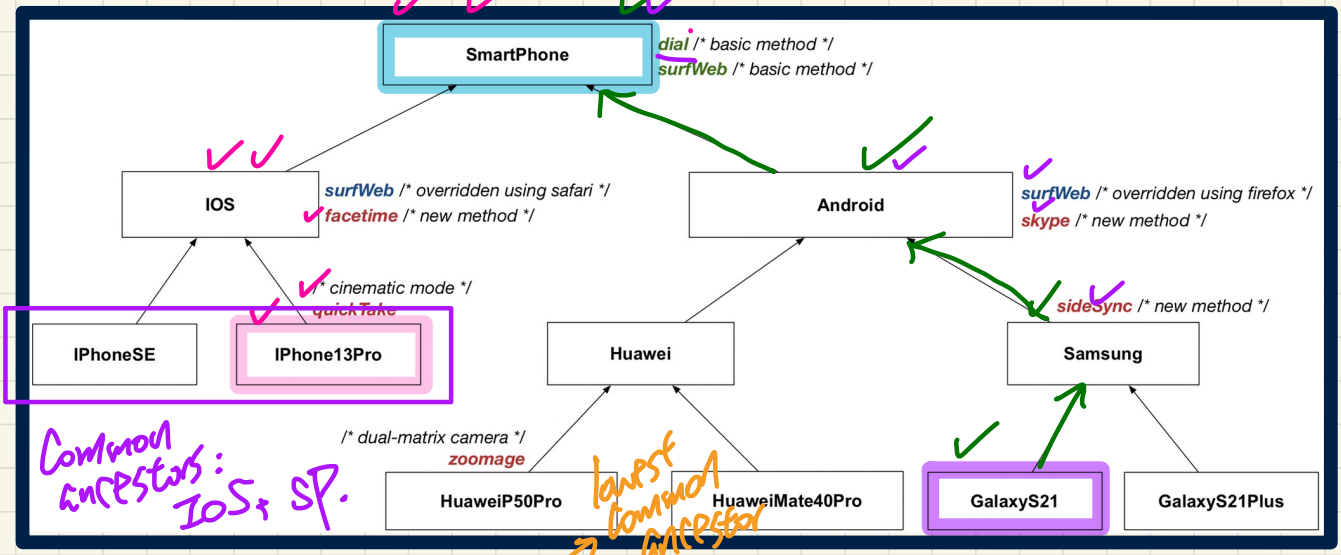
② exp(A) ≥ exp of any ancestor class

→ descendents of A.

① A's exp inherited to each of them

② exp(A) ≤ exp of any descendent class

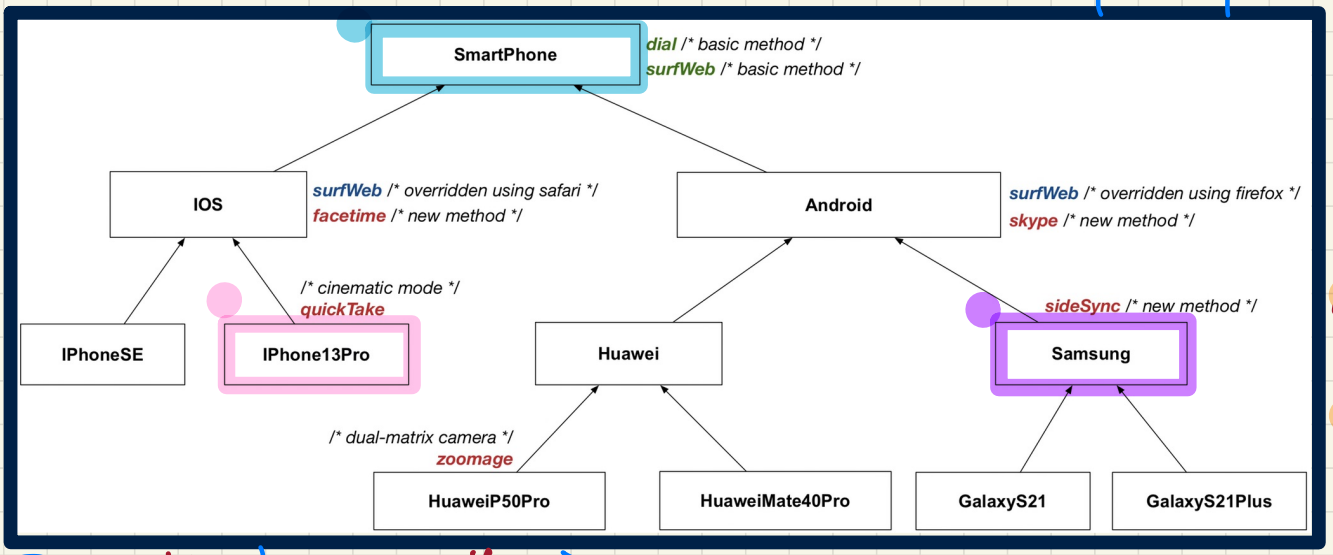
Inheritance Accumulates Code for Reuse



	ancestors	expectations	descendants
	GS21, SS, An., SP	sideSync, skype, surfWeb, dial	GS21
	IP13Pro, IOS, SP	quickTake, facetime, dial, surfWeb	IP13Pro
	SP	dial, surfWeb	all classes

inherited exp from common ancestor.

Inheritance Accumulates Code for Reuse



① sp1 = sp2 ; → to be valid the exp of sp1's ST must be satisfied ⇒ sp2 and sp3 are both accep. ∵ their STs are descendants

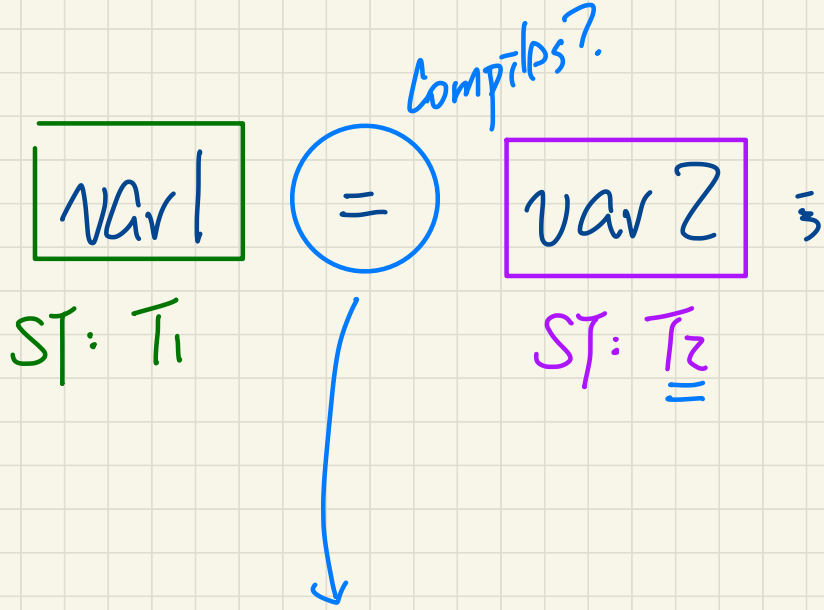
sp1 = sp3 ;

② sp2 = sp1 ; ✗ sp2 = sp3 ; ✓

```
SmartPhone sp1;
IPhone13Pro sp2;
Samsung sp3;
```

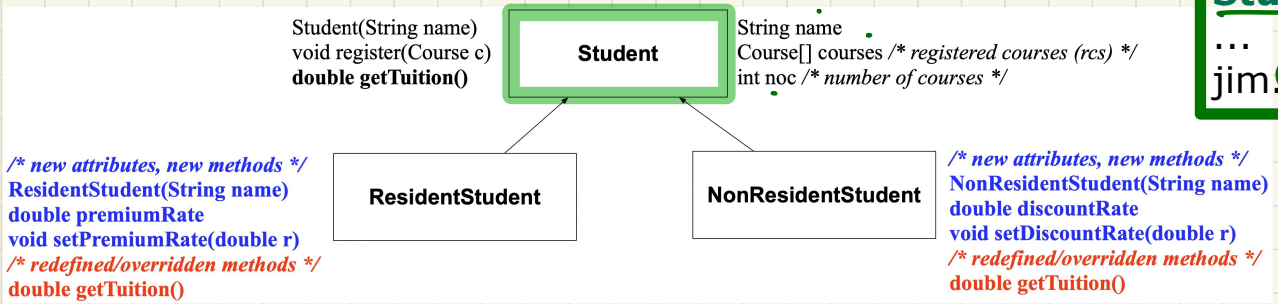
```
sp1 = ?;
sp2 = ?;
sp3 = ?;
```

in order for the var. assignments to be Compilable, what types of variables can appear at the RST of ?


compiles?
var1
=
var2
;
ST: T1
ST: T2
compiles if: ① T2 can satisfy the expectation of T1
② T2 is a descendent class of T1

Static Types determine Expectations

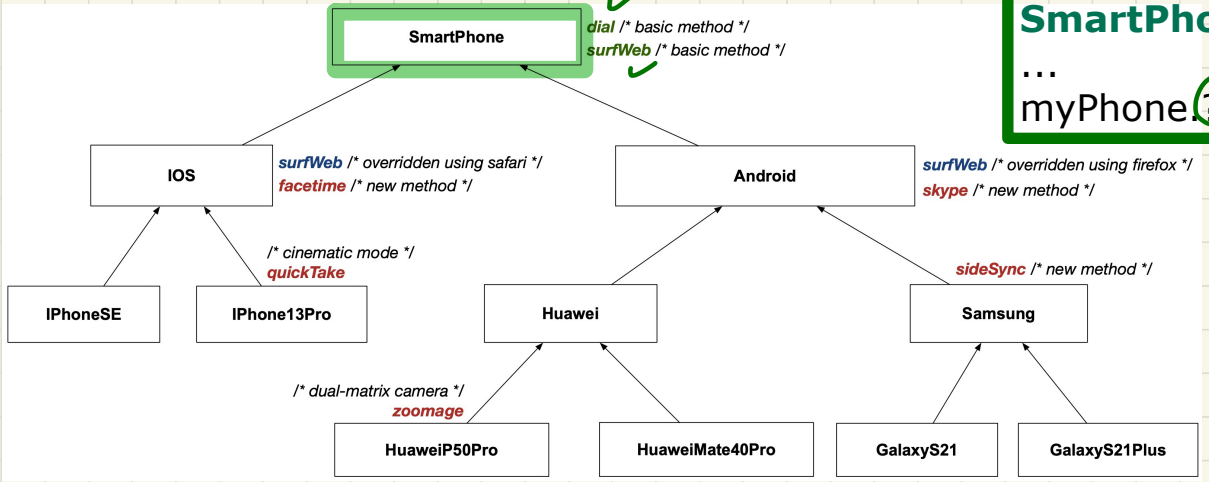
Inheritance Herarchy: Students



Declare:
Student jim;
...
jim.??

all att/meths in Student

Inheritance Herarchy: Smart Phones



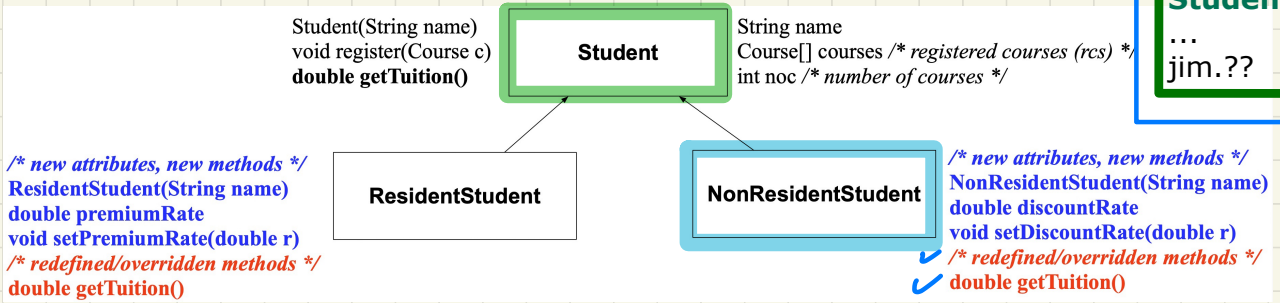
Declare:
SmartPhone myPhone;
...
myPhone.??

all att/meths in SP.

Static Types determine Expectations

jim = alan;
jim.setDr(..); // ST of jim remains Student

Inheritance Herarchy: Students

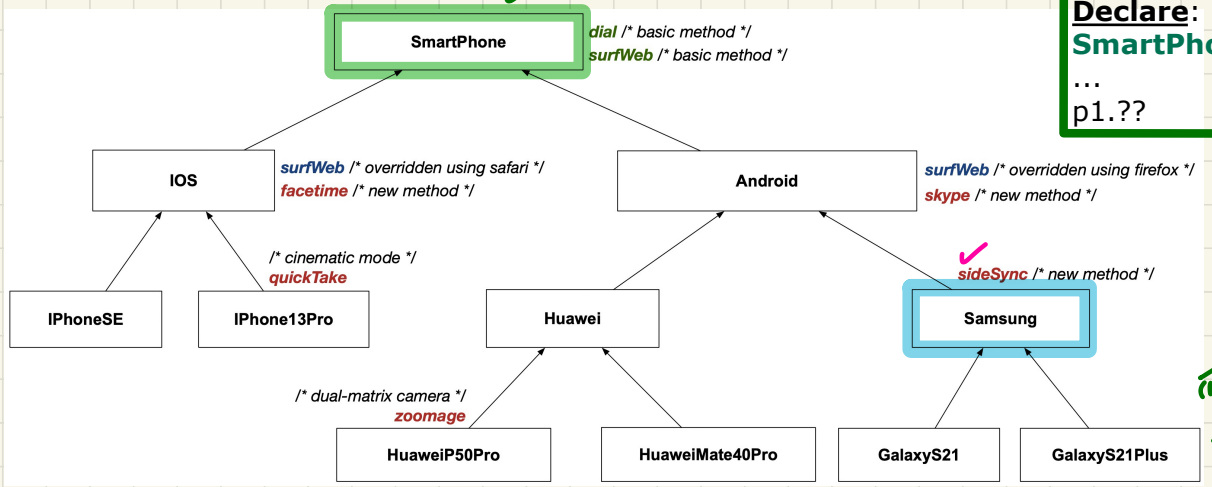


Declare: **Student** jim;
...
jim.??

Declare: **NRS** alan;
...
alan.??

① all attributes of Student
② dr, setDr.

Inheritance Herarchy: Smart Phones



Declare: **SmartPhone** p1;
...
p1.??

Declare: **Samsung** p2;
...
p2.??

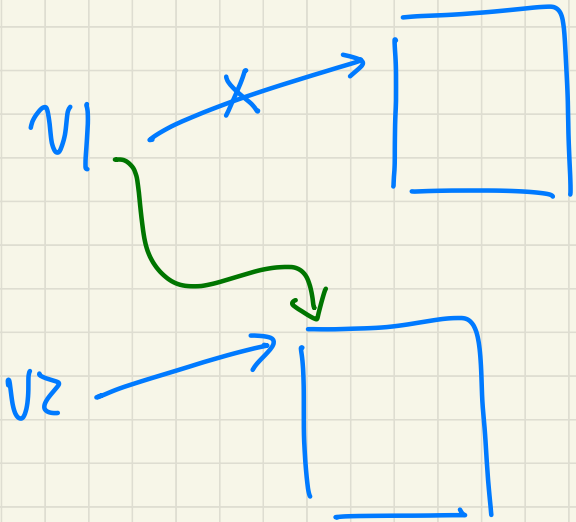
sideSync
skype
surfWeb
dial

inherited from SP class.

v1 is substituted by v2

v1 = v2

v1.x



Lecture 19 - Nov. 14

Inheritance

Polymorphism vs. Dynamic Binding
Type Casts: Named vs. Anonymous
Casts: Compilable vs. ClassCastException

Announcements/Reminders

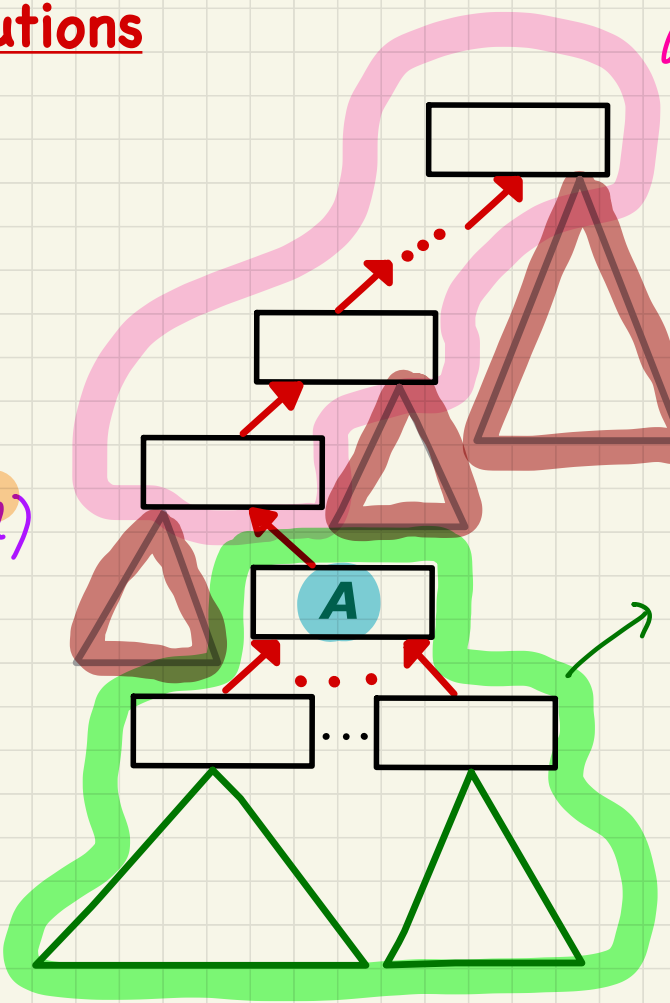
- **WrittenTest2** results to be released by Monday
- **Lab4** due tomorrow at noon
- **Lab5** to be released tomorrow
- **ProgTest3** next Wednesday, November 20
  + **Lab4** grading tests
  + **Lab4** solution video
- **Bonus** Opportunity coming: Formal Course Evaluation

Rules of Substitutions

```
A oa = ...;
? ob = ...;
oa = ob;
```

ST: A

what can the ST of ob be to make this substitution compile? (?)

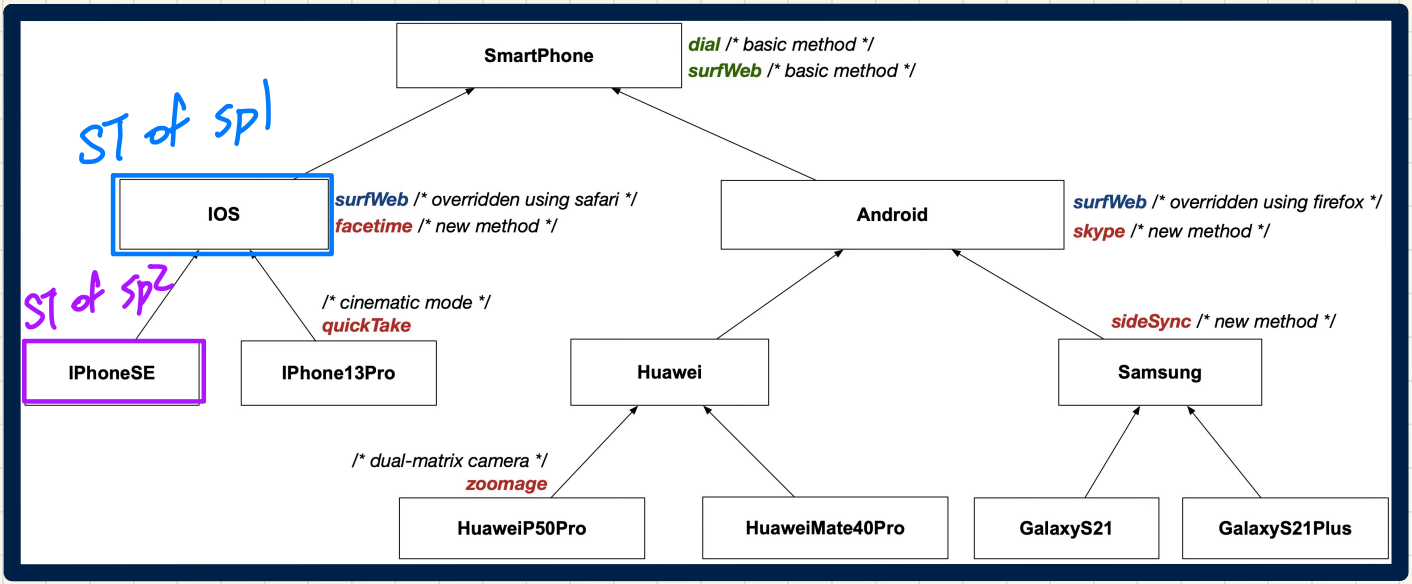


ancestors of A's parent (with less expectation)
↳ unsafe

neither ancestors nor descendants (with incompatible exp.)
↳ unsafe

descendant classes of A that can serve as the ST of ob (safe)

Rules of Substitutions (1)



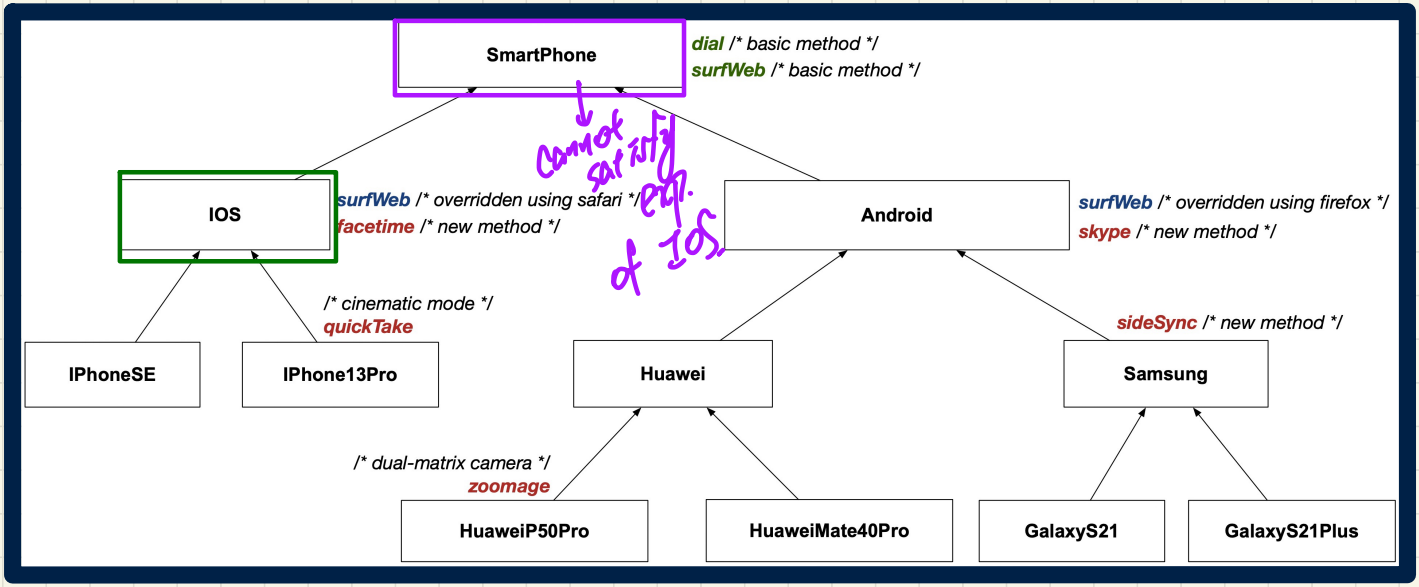
Declarations:

```
IOS sp1;
IPhoneSE sp2;
IPhone13Pro sp3;
```

Substitutions:

```
sp1 = sp2;
sp1 = sp3;
```

Rules of Substitutions (2)



Declarations:

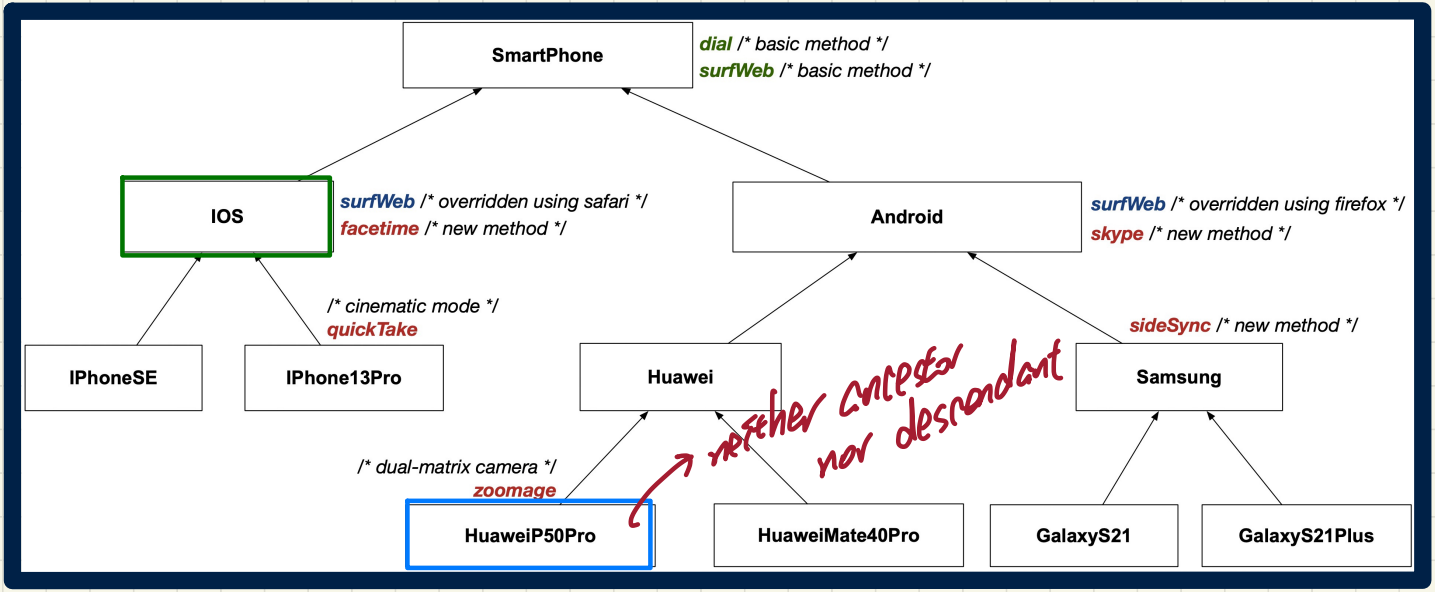
```
IOS sp1;
SmartPhone sp2;
```

Substitutions:

```
sp1 = sp2;
```

X

Rules of Substitutions (3)



Declarations:

```
IOS sp1;
HuaweiP50Pro sp2;
```

Substitutions:

```
sp1 = sp2;
```

X

Visualization: Static Type vs. Dynamic Type

Declaration:

Student s;

ST

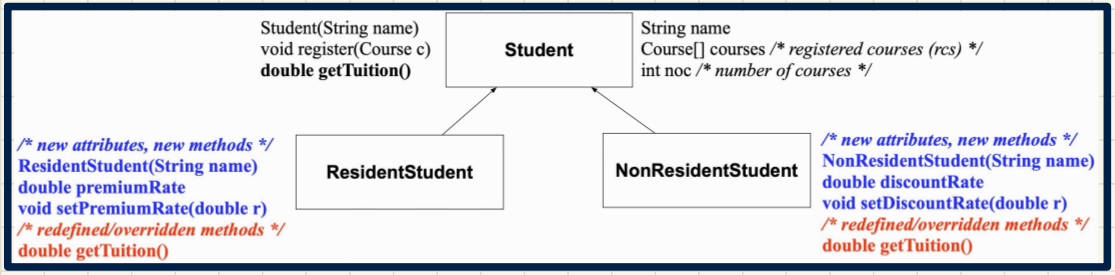
Substitution:

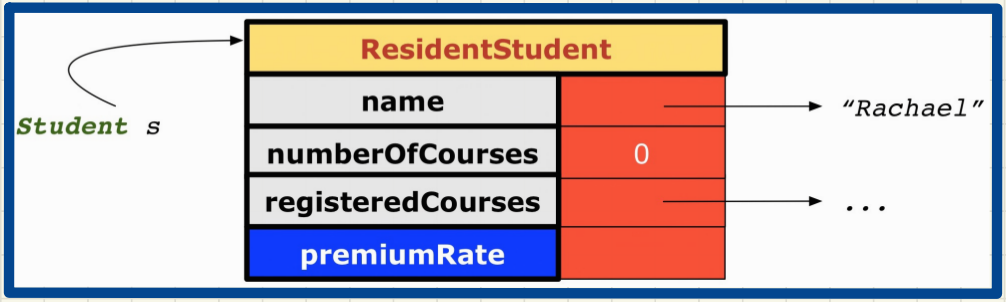
s = **new ResidentStudent**("Rachael");

DT

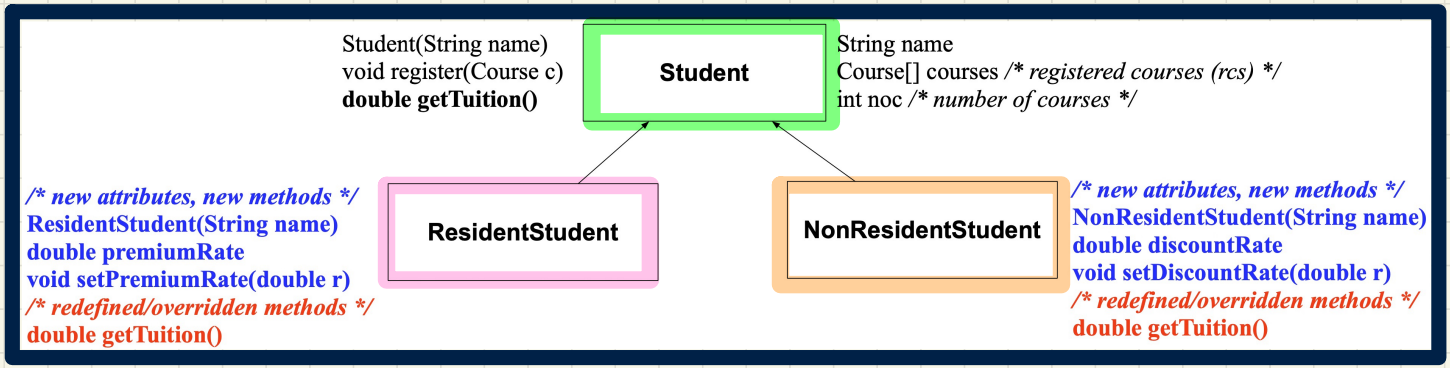
Static Type: Expectation

Dynamic Type: Accumulation of Code





Change of Dynamic Type (1.1)



Example 1:

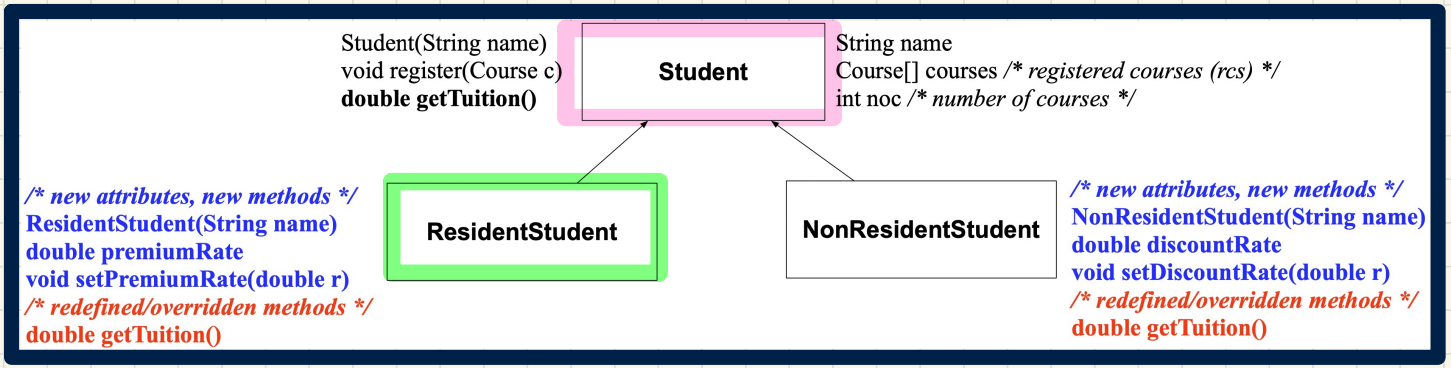
```
Student jim = new ResidentStudent(...);
jim = new NonResidentStudent(...);
```

DT of jim: RS.

DT of jim: NRS

both classes can fulfill the exp. of jim's ST.

Change of Dynamic Type (1.2)



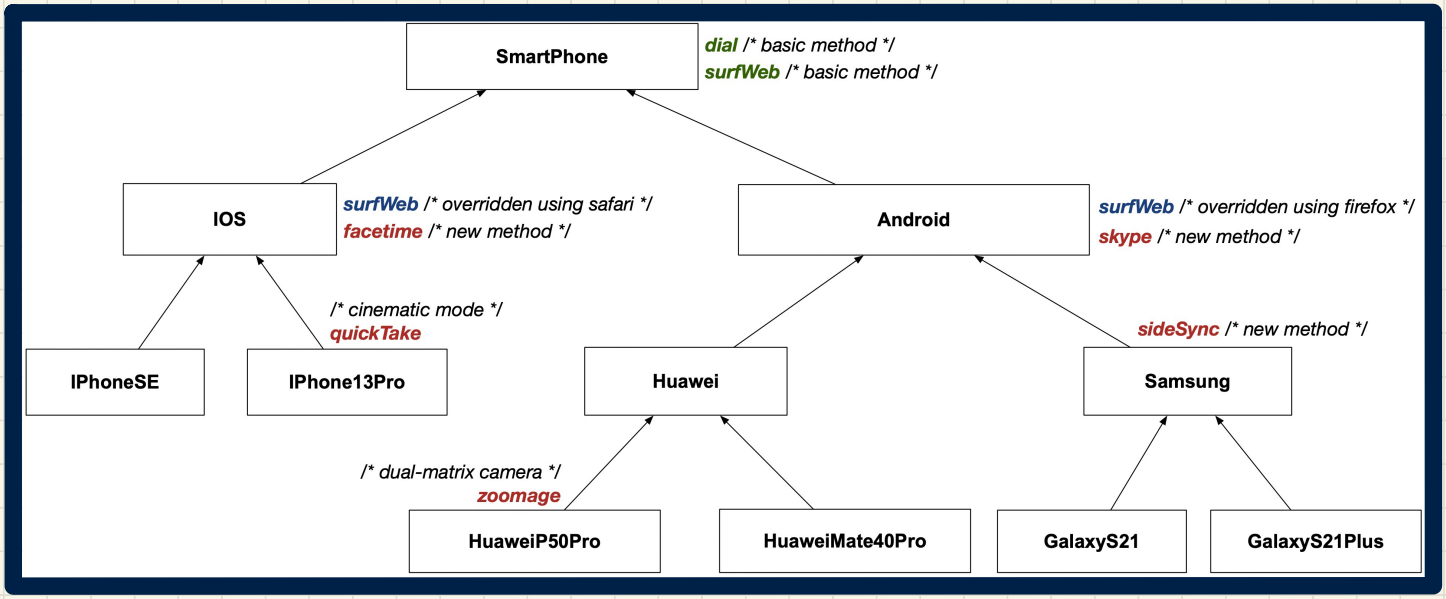
Example 2:

```
ResidentStudent jeremy = new Student(...);
```

① Student is not a descendant of jeremy's ST Student

② Student cannot fulfill the exp. of jeremy's ST.

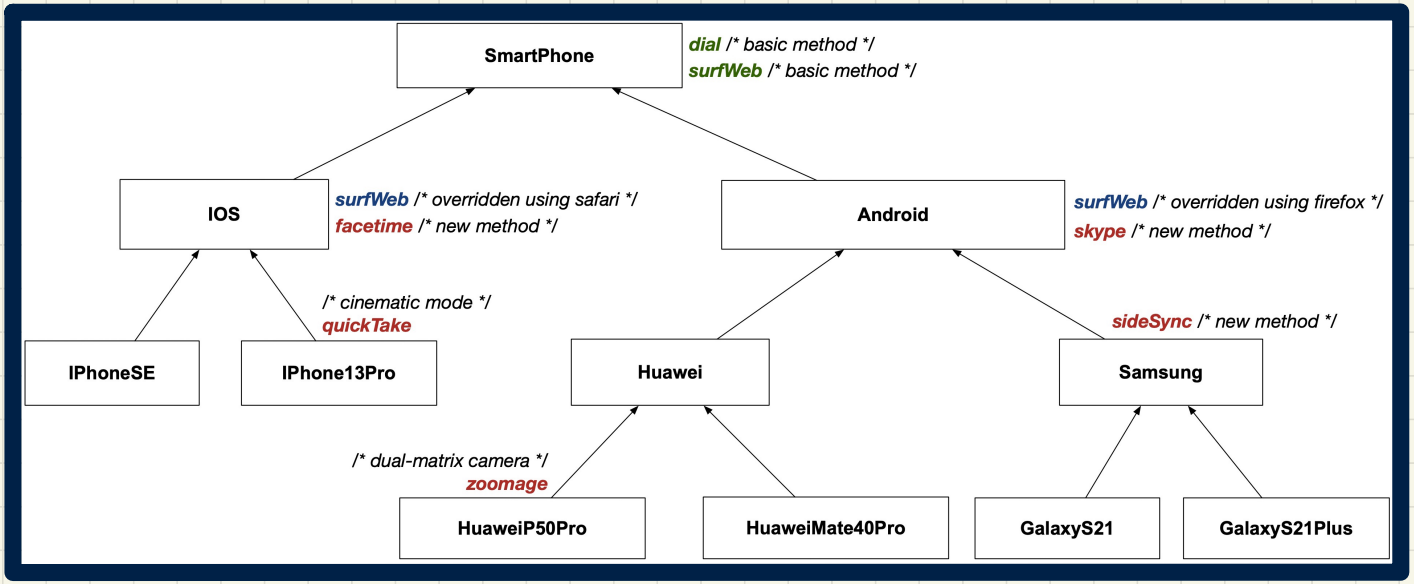
Change of Dynamic Type: Exercise (1)



Exercise 1:

```
Android myPhone = new HuaweiP50Pro(...);
myPhone = new GalaxyS21(...);
```

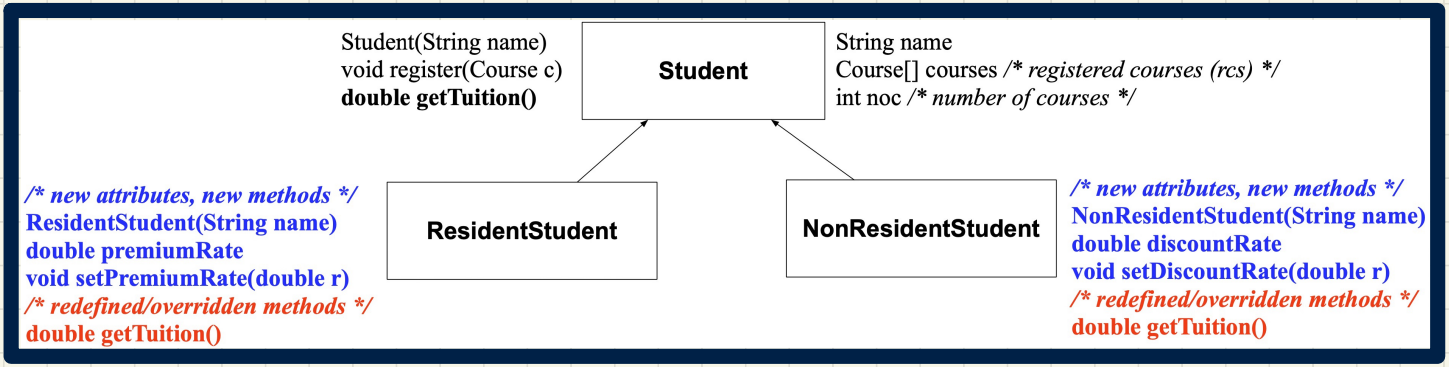
Change of Dynamic Type: Exercise (2)



Exercise 2:

```
IOS myPhone = new HuaweiP50Pro(...);
myPhone = new GalaxyS21(...);
```

Change of Dynamic Type (2.1)



Given:

```
Student jim = new Student(...);
ResidentStudent rs = new ResidentStudent(...);
NonResidentStudent nrs = new NonResidentStudent(...);
```

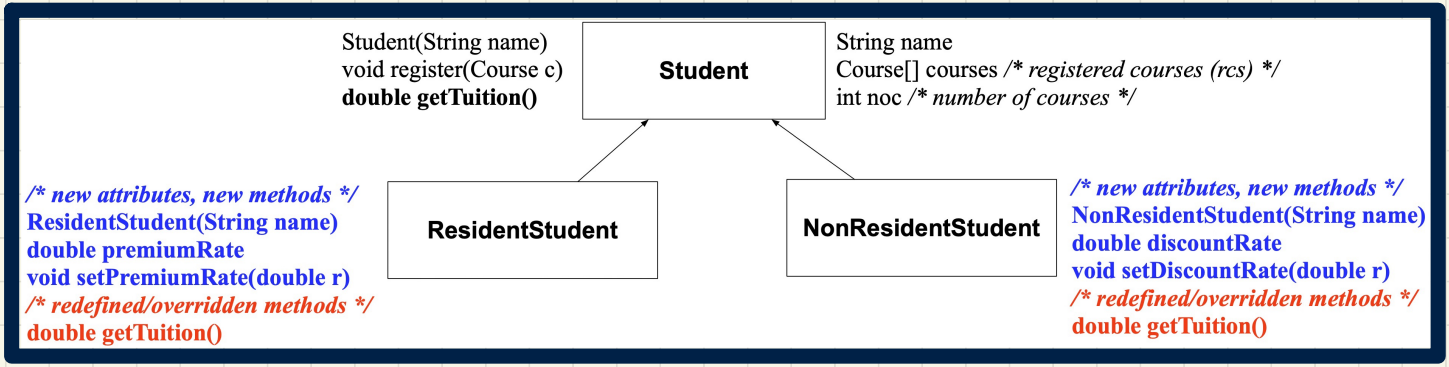
Example 1:

```
jim = rs;
println(jim.getTuition());
jim = nrs;
println(jim.getTuition());
```

what's DT?

what's DT?

Change of Dynamic Type (2.2)



Given:

```
Student jim = new Student(...);
ResidentStudent rs = new ResidentStudent(...);
NonResidentStudent nrs = new NonResidentStudent(...);
```

Example 2:

```
rs = jim;
println(rs.getTuition());
nrs = jim;
println(nrs.getTuition());
```

Polymorphism and Dynamic Binding

Polymorphism:

An object's **static type** may allow **multiple** possible **dynamic types**.

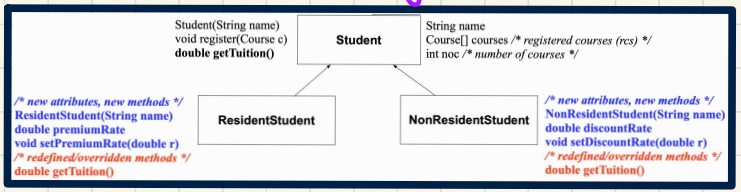
⇒ Each **dynamic type** has its **version** of method.

Dynamic Binding:

An object's **dynamic type** determines the **version** of method being invoked.

→ at RT, DT of jim can be any of the descendants of jim's ST.

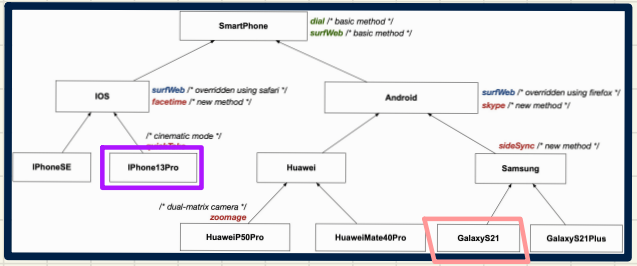
```
Student jim = new ResidentStudent(...);
jim.getTuition();
jim = new NonResidentStudent(...);
jim.getTuition();
```



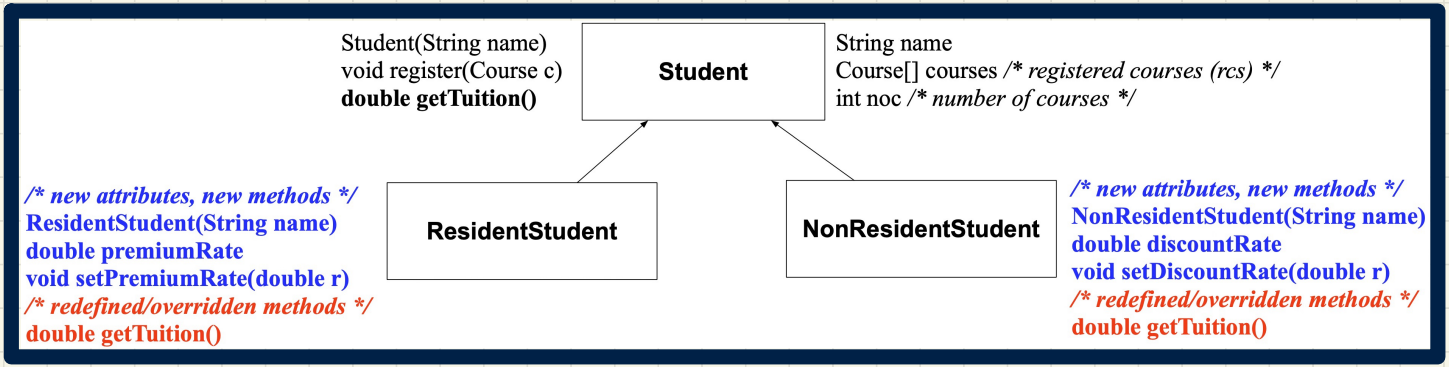

```
SmartPhone sp1 = new IPhone13Pro(...);
SmartPhone sp2 = new GalaxyS21(...);
sp1.surfWeb();
sp1 = sp2;
sp1.surfWeb();
```

DT: IP13Pro (safari)

DT: GS21 (firefox).

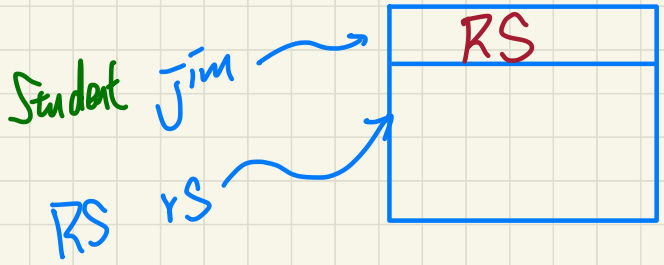


Type Cast: Motivation

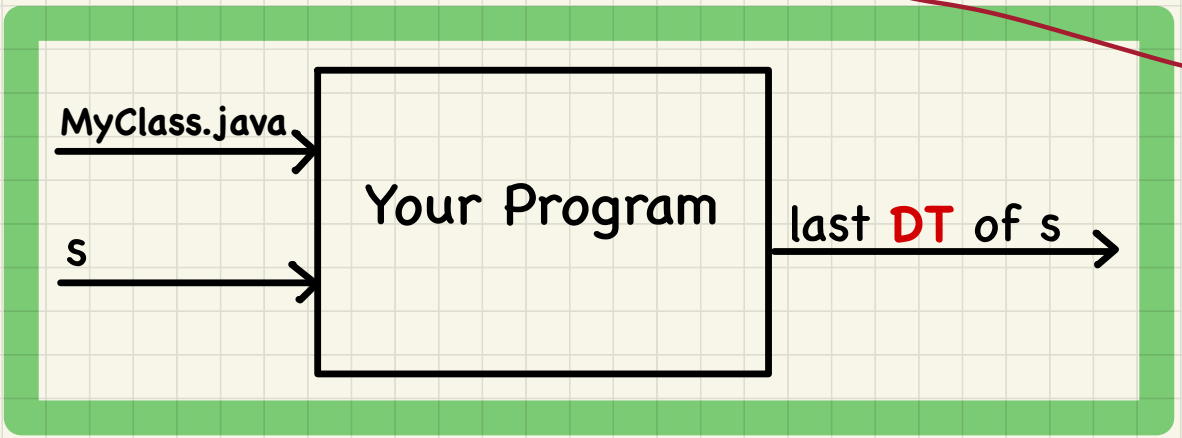



```
1  Student jim = new ResidentStudent("J. Davis");
2  ResidentStudent rs = jim;
3  rs.setPremiumRate(1.5);
```

ST: Student



An A+ Challenge: Inferring the DT of a Variable



known to be impossible (undecidable)

```
class MyClass {
   main (...)
      Student s = ...;
      ...
      s = new ResidentStudent(...);
   }
}
```

Anatomy of a Type Cast

Student jim = **new ResidentStudent**("Jim");

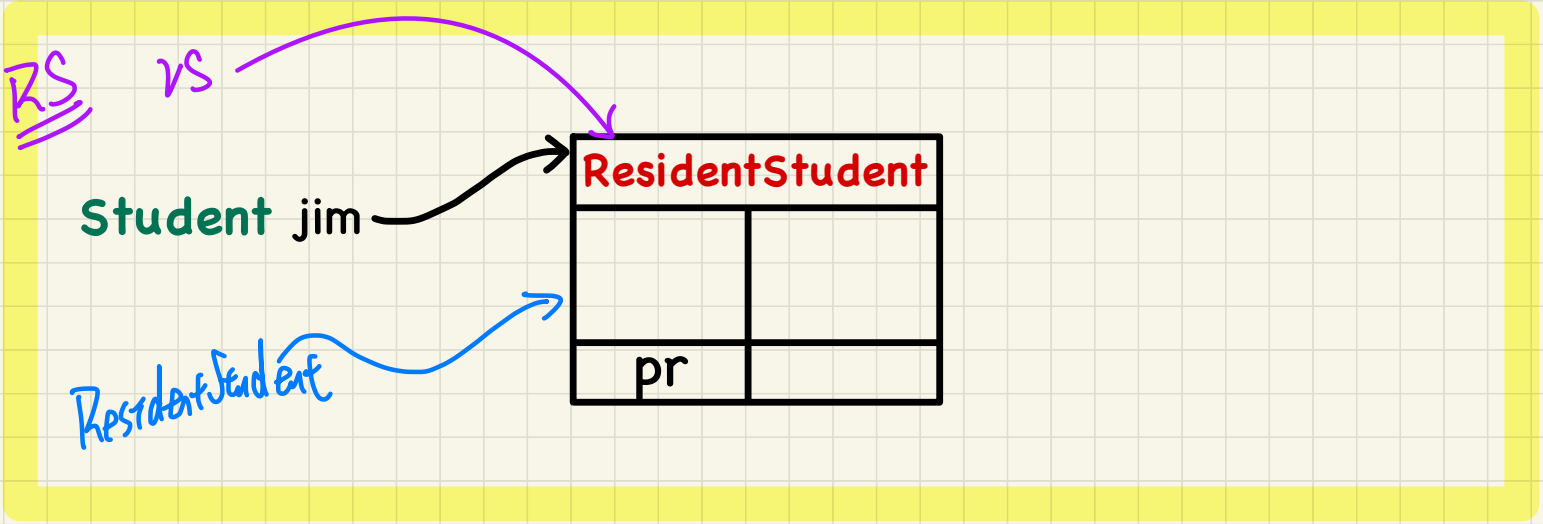
ST: *ResidentStudent*

ResidentStudent rs = (*ResidentStudent*) *jim* ;

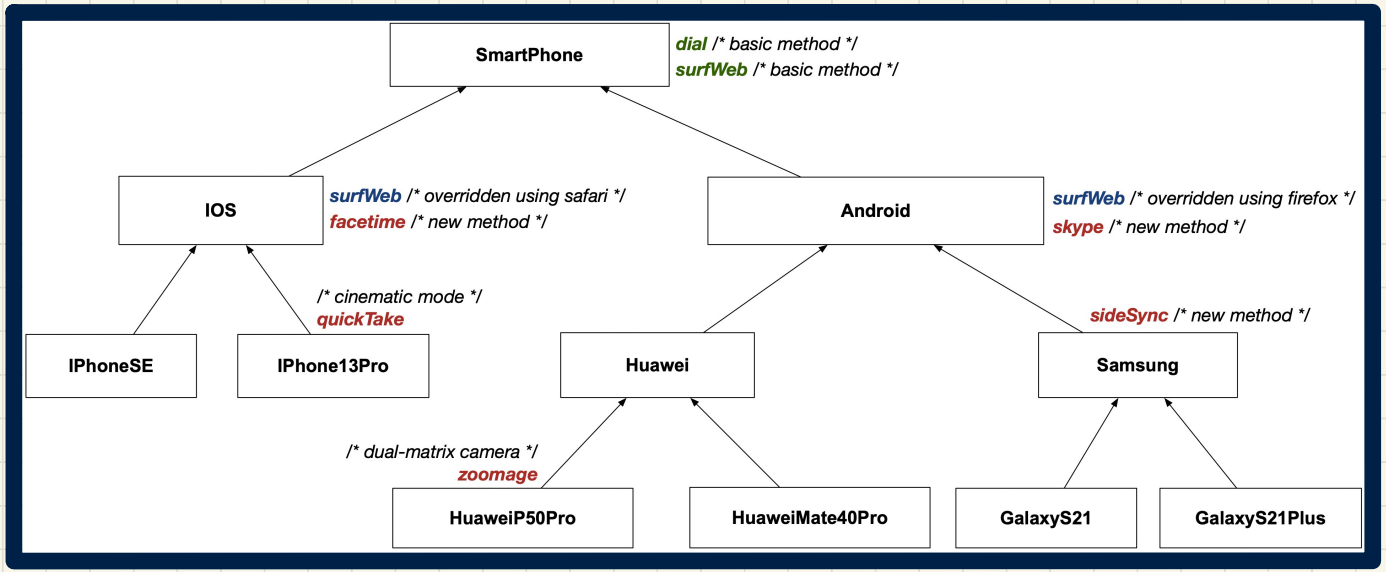
valid substitution

ST: Student

an alias whose ST is *ResidentStudent*



Type Cast: Named vs. Anonymous



Named Cast: Use intermediate variable to store the cast result.

```
SmartPhone aPhone = new IPhone13Pro();
IOS forHeeyeon = (IPhone13Pro) aPhone;
forHeeyeon.facetime();
```

Anonymous Cast: Use the cast result directly.

```
SmartPhone aPhone = new IPhone13Pro();
((IPhone13Pro) aPhone).facetime();
```

↳ expression of ST: IP13Pro

Exercise

```
SmartPhone aPhone = new IPhone13Pro();
(IPhone13Pro) aPhone.facetime();
```

error first
↳ not valid :(
aPhone's ST SP does not exp. facetime

Compilable Casts: Upwards vs. Downwards

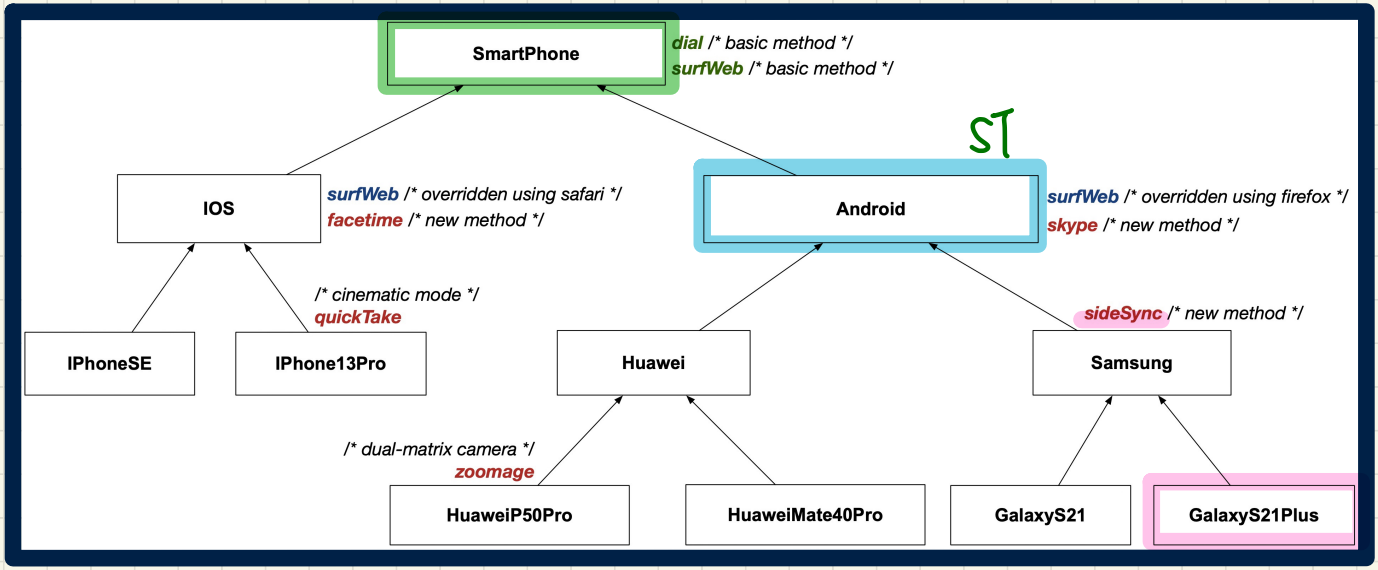
```
Android myPhone = new GalaxyS21Plus();
SmartPhone sp = (SmartPhone) myPhone;
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
```

upward casting

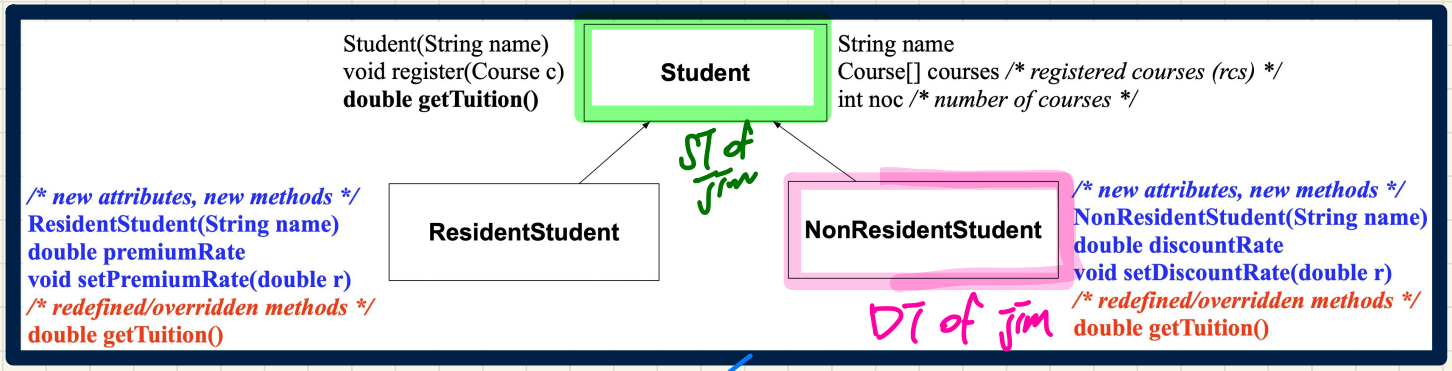
downward casting.

Expectations

	sp	myPhone	ga
dial			
surfWeb			
skype			
sideSync			
facetime			
quickTake			
zoomage			



Compilable Type Cast May Fail at Runtime (1)




```
1  Student jim = new NonResidentStudent("J. Davis");
2  ResidentStudent rs = (ResidentStudent) jim;
3  rs.setPremiumRate(1.5);
```

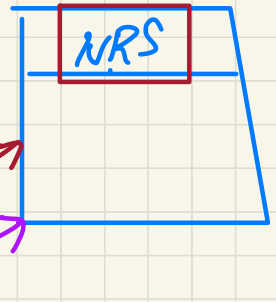
ClassCastException.

Compilable cast (downward) Stud. jim

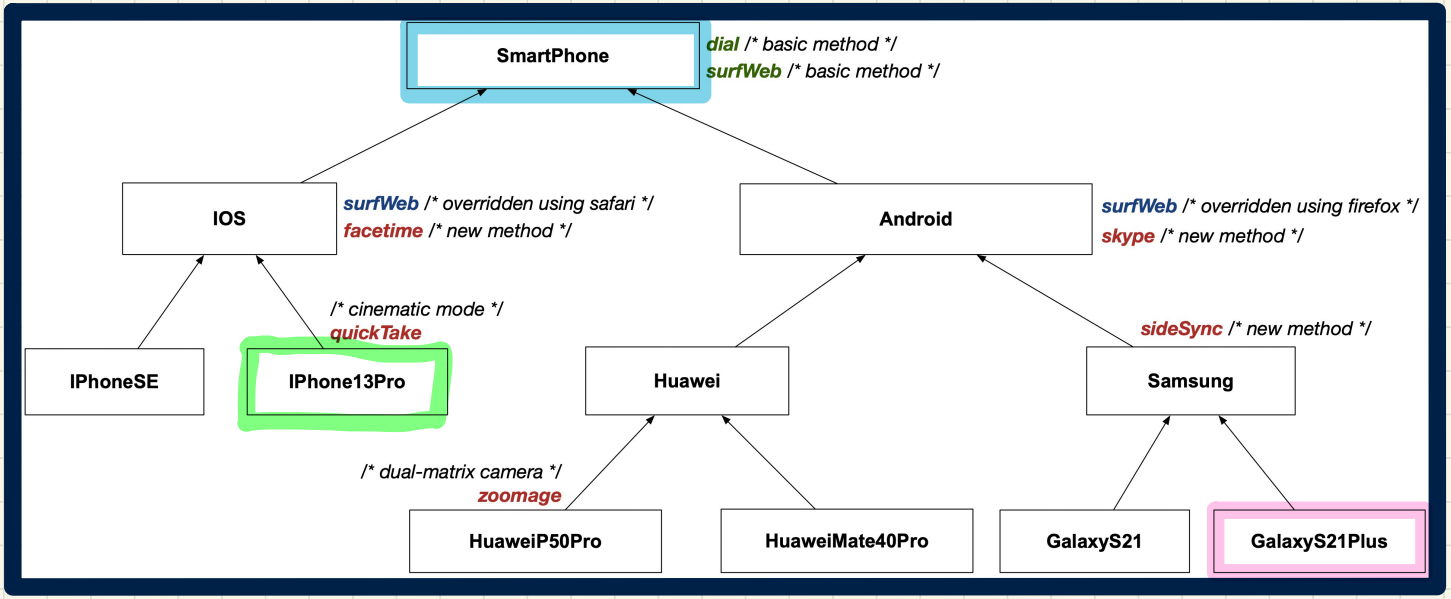
① rs expected to be used as RS

② rs currently pointing to a NRS, DT which cannot fulfill exp. of RS

Res. Stud. rs



Compilable Type Cast May Fail at Runtime (2)



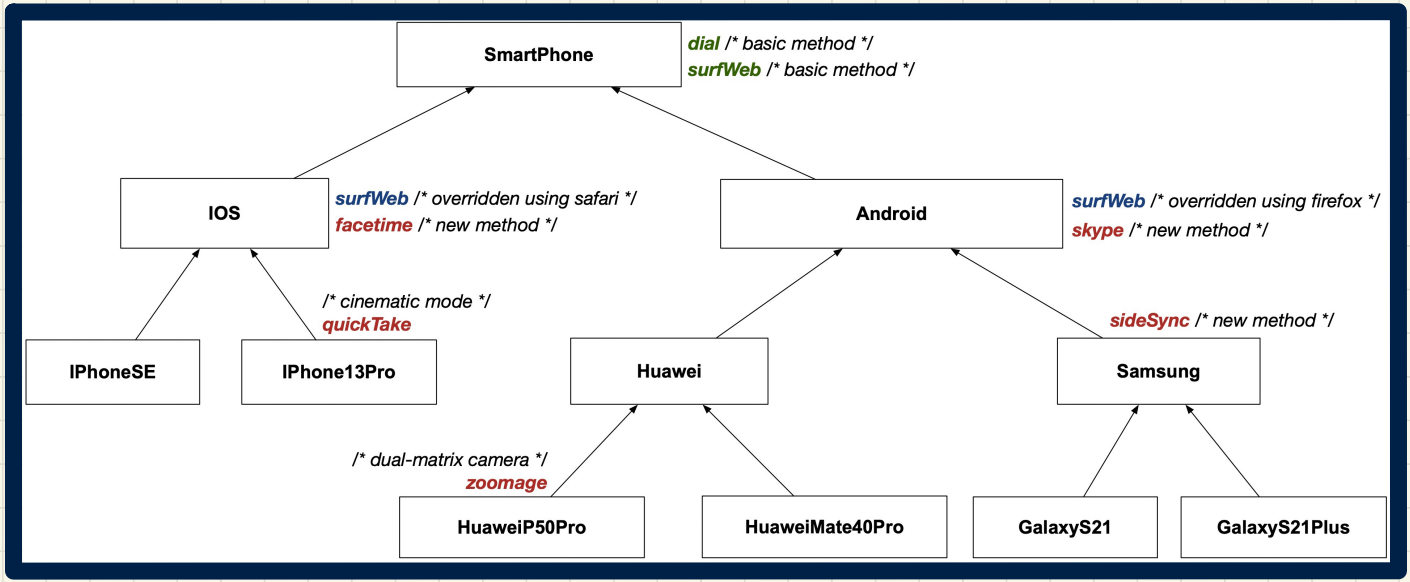

```
1  SmartPhone aPhone = new GalaxyS21Plus();
2  IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
3  forHeeyeon.quickTake();
```

compiles ∵ downward cast

exp of cast type IP13Pro

ClassCastException ∵ DT GS21P cannot fulfill

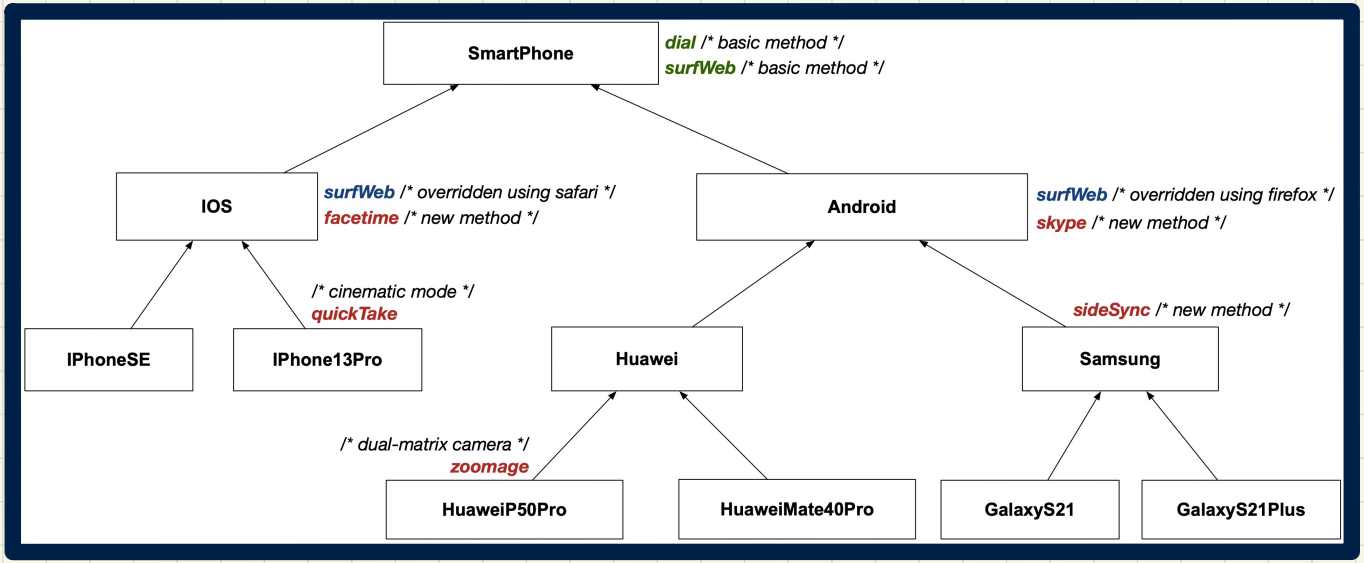
Exercise: Compilable Type Cast? Fail at Runtime? (1)




```
SmartPhone myPhone = new Samsung();
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
```

Compilable? ClassCastException at runtime?

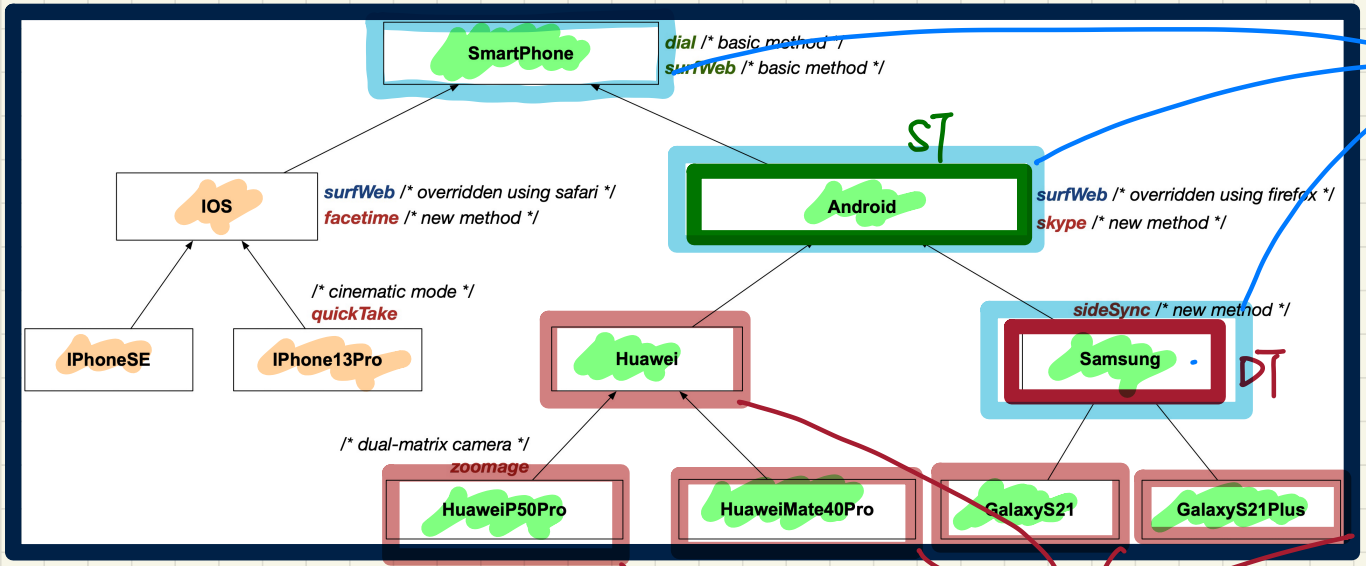
Exercise: Compilable Type Cast? Fail at Runtime? (2)




```
SmartPhone myPhone = new Samsung();
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
IPhone13Pro ip = (IPhone13Pro) myPhone;
```

Compilable? ClassCastException at runtime?

Compilable Cast vs. Exception-Free Cast



can be fulfilled by DT.

```
Android myPhone = new Samsung();
```

Cannot be fulfilled by DT. runtime.

Compile time

- Compilable Casts
- Non-Compilable Casts
- Exception-Free Casts
- ClassCastException

Lecture 20 - Nov. 19

Inheritance

Type Casts: Exercise
Checking Dynamic Types: instanceof
Polymorphic Method Parameters

Announcements/Reminders

- **WrittenTest2** results released
- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- **ProgTest3** tomorrow
  + **Not** covered: equals method, copy constructors
- **Bonus** Opportunity coming: Formal Course Evaluation

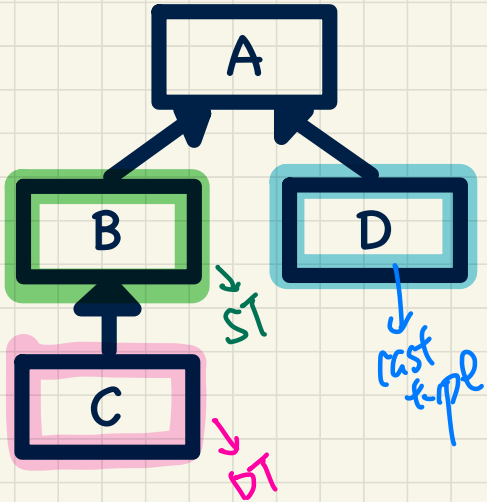
Exercise: Compilable Cast vs. Exception-Free Cast

```
class A { }
class B extends A { }
class C extends B { }
class D extends A { }
```

PointV2 ((Object) obj)
String

```
1  B b = new C();
2  D d = (D) b;
```

→ cast not compilable ∵ neither upward nor downward



D d = (D) ((A) b)

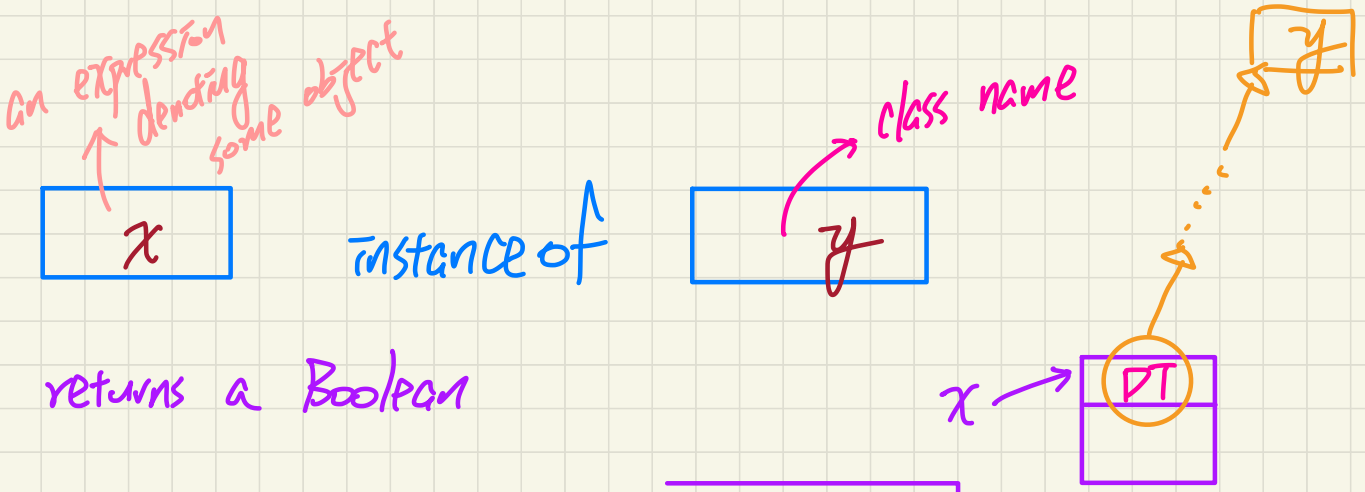
upward

downward

↳ ClassCastException

∵ DT C cannot fulfill exp. of cast type D

not a descendant of



↳ returns a Boolean

↳ true if x's DT can fulfil the exp. of y.

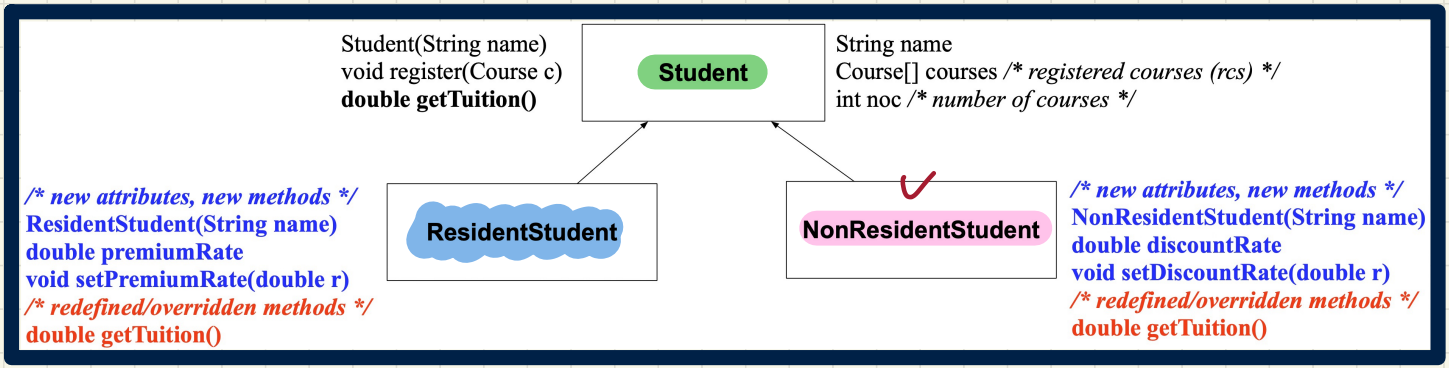
① x's DT is descendant of y

② y is ancestor of x's DT

↳ true means we can cast x to y (y) x

ClassCastException free

Checking Dynamic Types at Runtime (1)

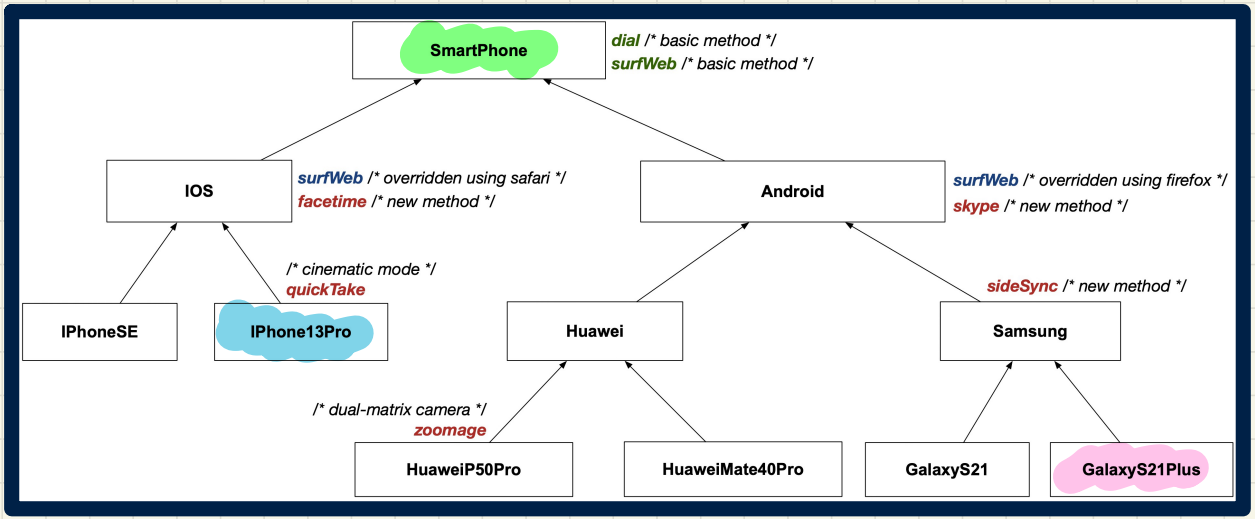



```
1  Student jim = new NonResidentStudent("J. Davis");
2  if (jim instanceof ResidentStudent) {
3    ResidentStudent rs = (ResidentStudent) jim;
4    rs.setPremiumRate(1.5);
5  }
```

↓ downward cast

* Can the DT of jim (NRS) fulfill exp of cast type RS?
  ↳ No → instanceof returns false

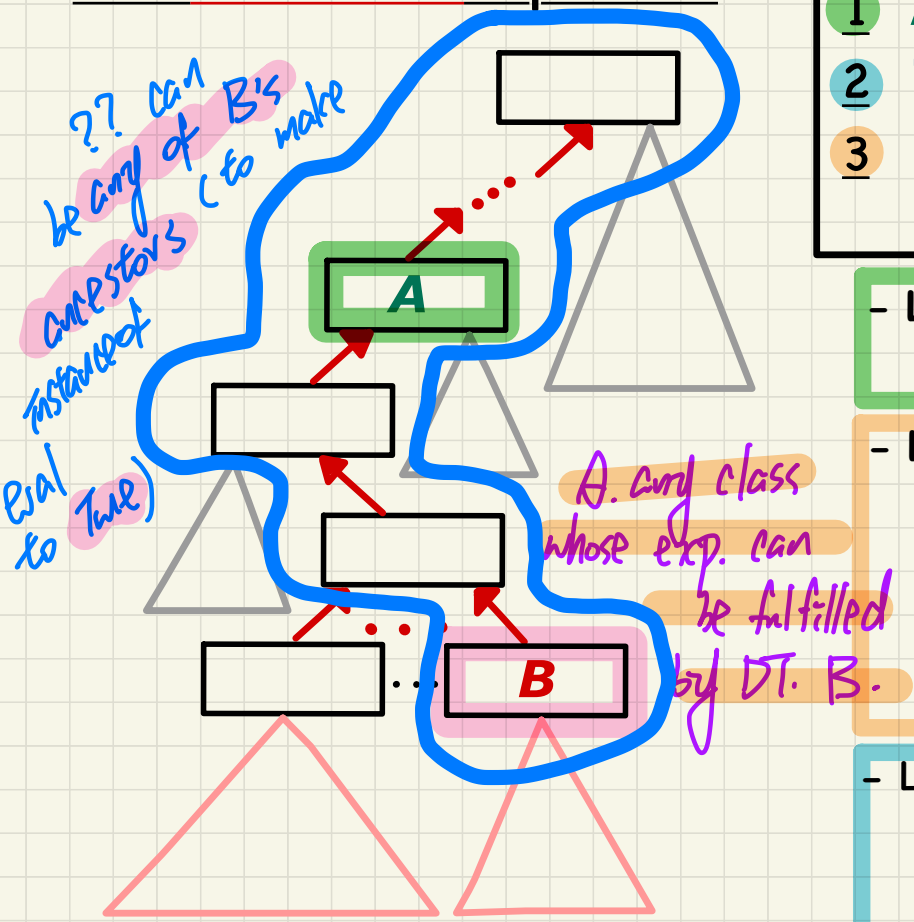
Checking Dynamic Types at Runtime (2)




```
1  SmartPhone aPhone = new GalaxyS21Plus();
2  if (aPhone instanceof IPhone13Pro) {
3    IOS forHeeyeon = (IPhone13Pro) aPhone;
4    forHeeyeon.facetime();
5  }
```

true ⇒ safe to write (IP13Pro) aPhone

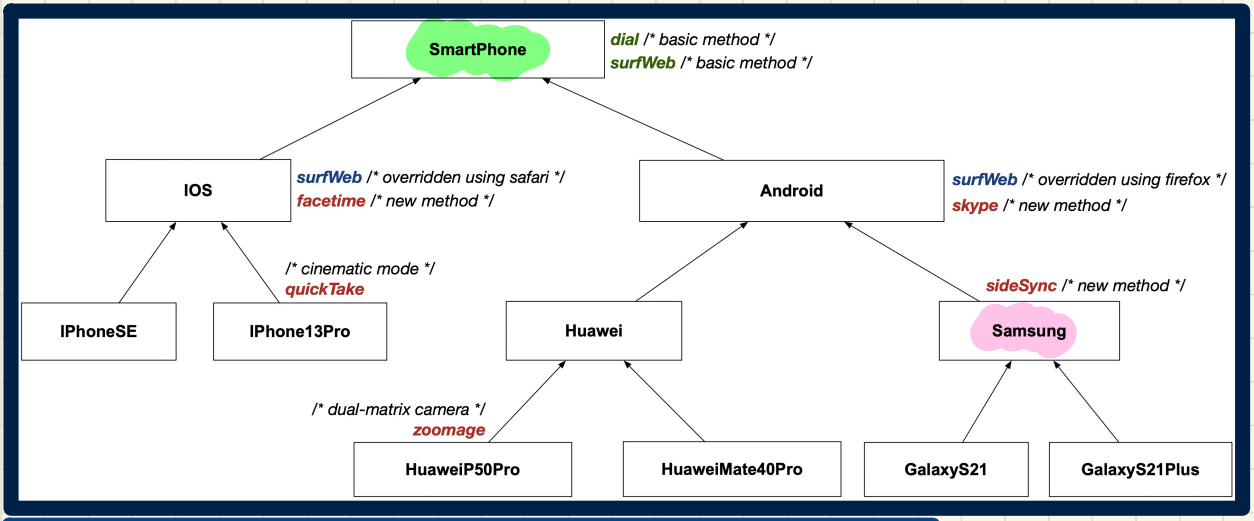
The instanceof Operator




```
1  A obj = new B();
2  if (obj instanceof ??) {
3    ?? obj2 = (??) obj;
   }
```

- L1 compiles if B can fulfill expectations of A.
- L3:
  - Compiles if Up or Down cast w.r.t. A.
  - ClassCastException if B cannot fulfill expectations on ??.
- L2:
  - Evaluates to true if B can fulfill expectations on ??.

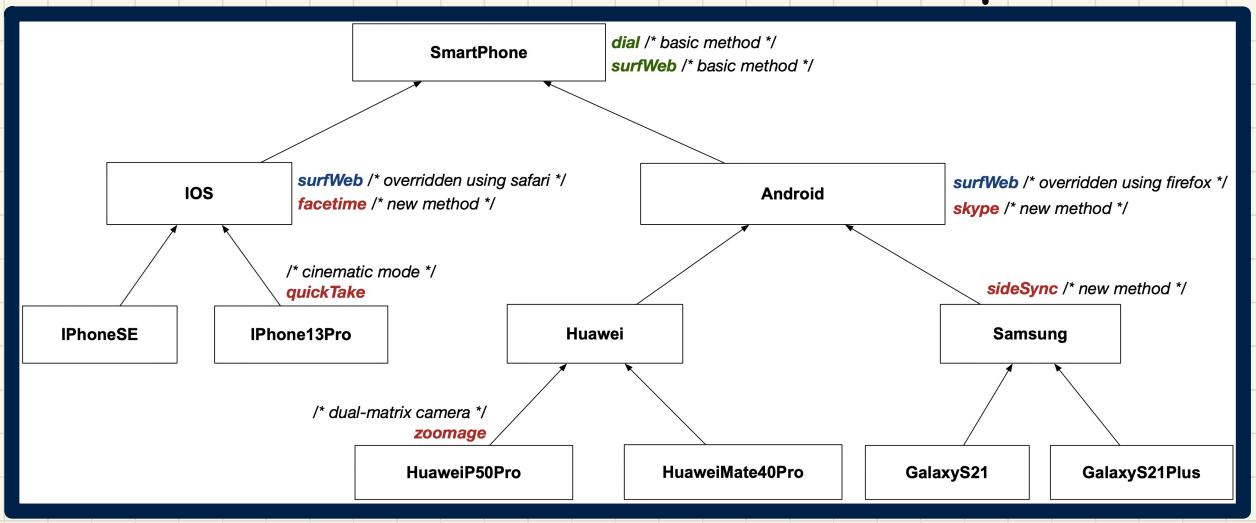
Use of the instanceof Operator




```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android); ✓
println(myPhone instanceof Samsung); ✓
println(myPhone instanceof GalaxyS21); ✗
println(myPhone instanceof IOS); ✗
println(myPhone instanceof IPhone13Pro); ✗
```

myPhone instanceof ??
evaluates to true if
Samsung can
fulfill expectations on ??.

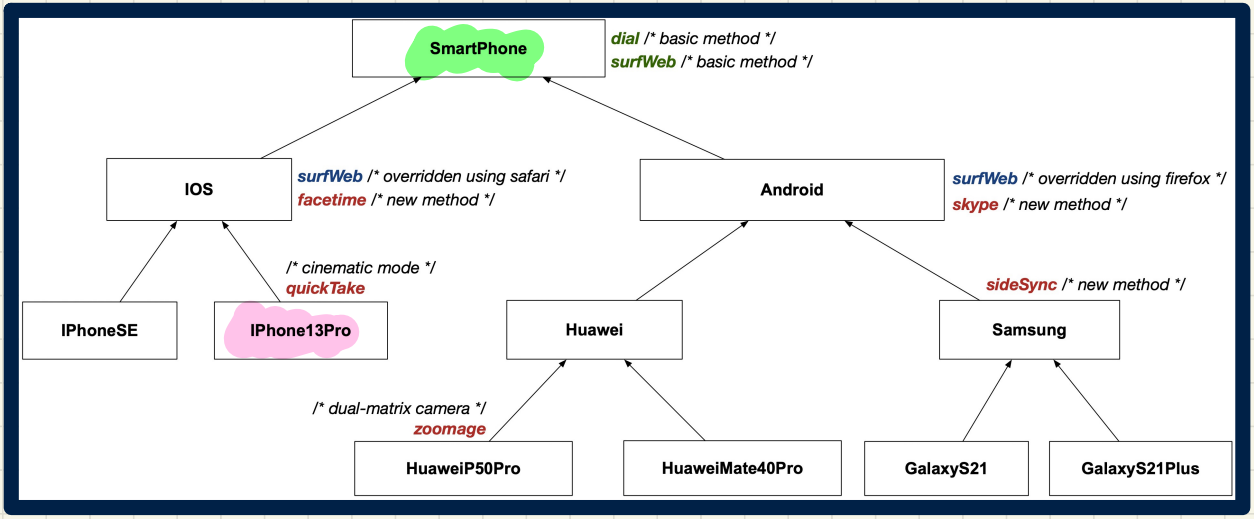
Safe Cast via Use of the instanceof Operator




```
1  SmartPhone myPhone = new Samsung();
2  /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3  if(myPhone instanceof Samsung) {
4    Samsung samsung = (Samsung) myPhone;
5  }
6  if(myPhone instanceof GalaxyS21Plus) {
7    GalaxyS21Plus galaxy = (GalaxyS21Plus) myPhone;
8  }
9  if(myphone instanceof HuaweiMate40Pro) {
10   Huawei hw = (HuaweiMate40Pro) myPhone;
11 }
```

myPhone instanceof ??
evaluates to true if
Samsung can
fulfill expectations on ??.

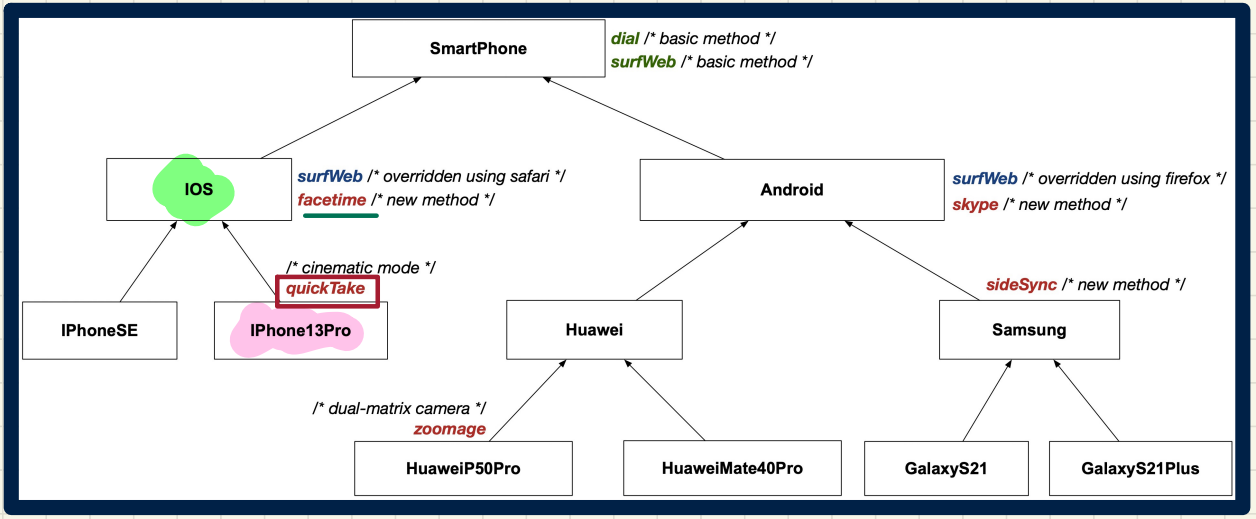
Static Types, Casts, Polymorphism (1)




```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  SmartPhone sp = new IPhone13Pro();
2  sp.dial();
3  sp.facetime();
4  sp.quickTake();
```

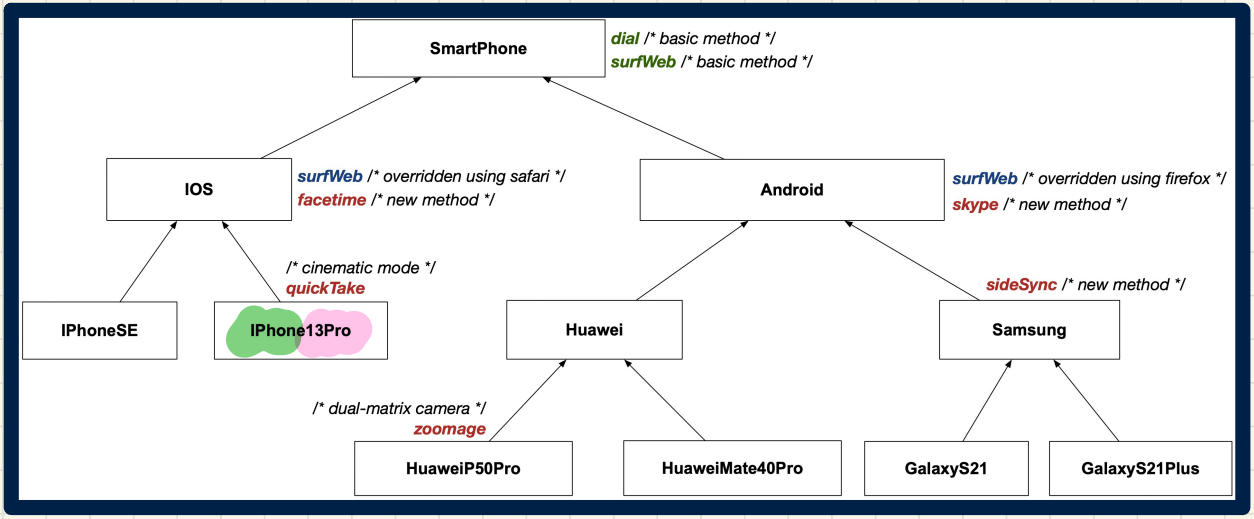
Static Types, Casts, Polymorphism (2)




```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  IOS ip = new IPhone13Pro();
2  ip.dial();
3  ip.facetime();
4  ip.quickTake();
```

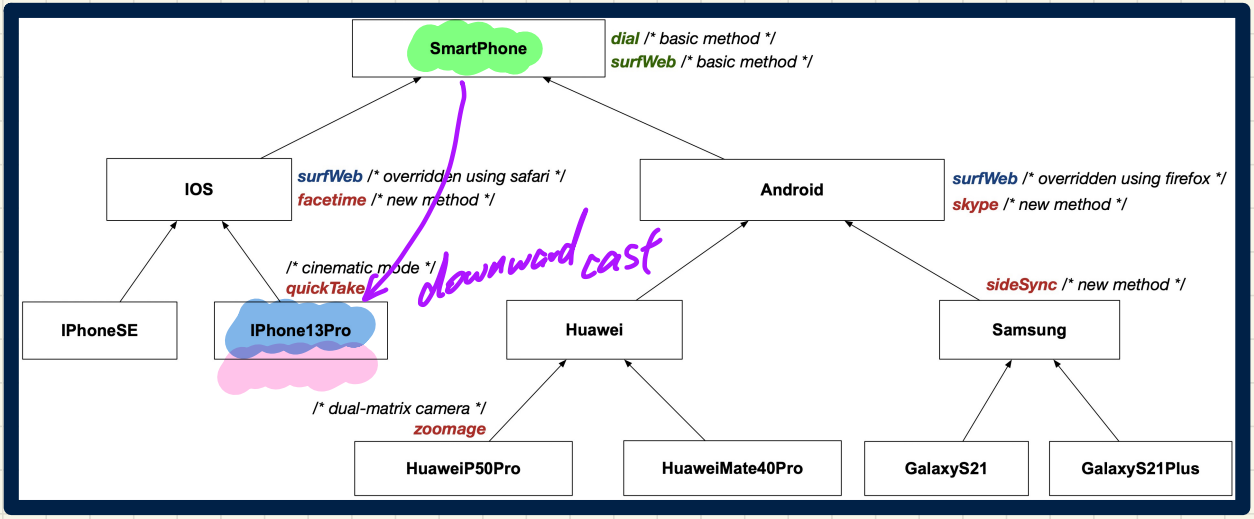
Static Types, Casts, Polymorphism (3)




```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  IPhone13Pro ip6sp = new IPhone13Pro();
2  ip6sp.dial();
3  ip6sp.facetime();
4  ip6sp.quickTake();
```

Static Types, Casts, Polymorphism (4)

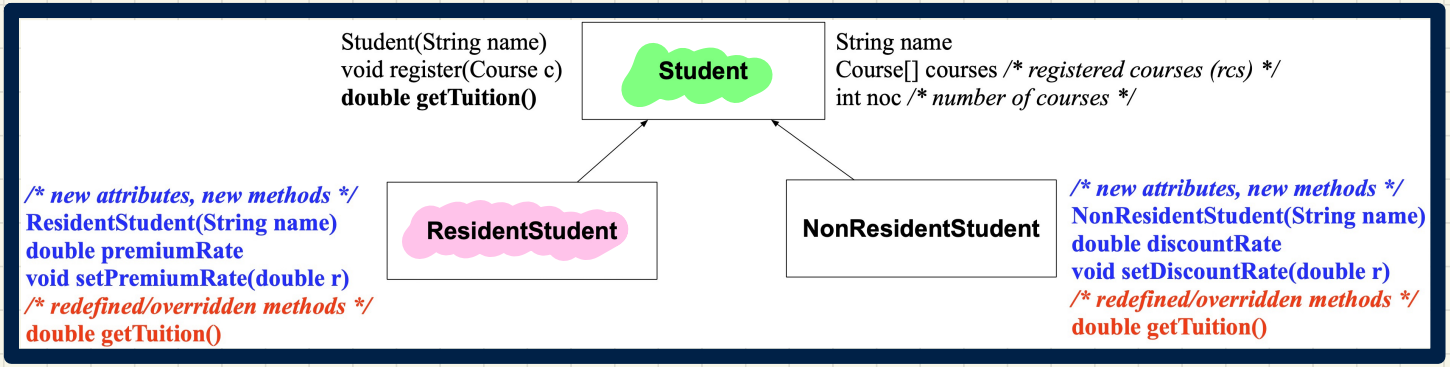



```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  SmartPhone sp = new IPhone13Pro();
2  ((IPhone13Pro) sp).dial();
3  ((IPhone13Pro) sp).facetime();
4  ((IPhone13Pro) sp).quickTake();
```

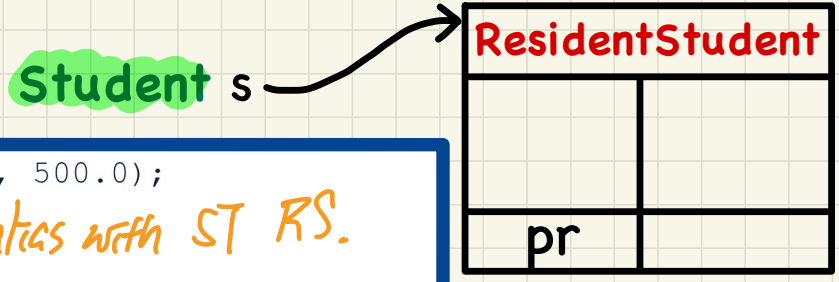
also with ST IP13Pro. (no CCE).

Static Types, Casts, Polymorphism (5)

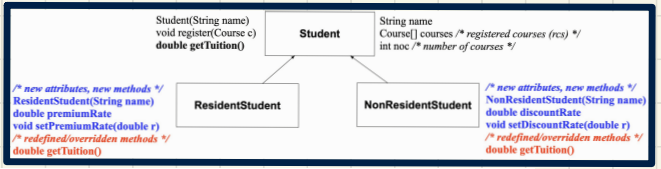



```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if(s instanceof ResidentStudent) {
  ((ResidentStudent) s).setPremiumRate(1.75);
  System.out.println(((ResidentStudent) s).getTuition());
}
```

aligns with ST RS.



Polymorphic Parameters (1)



ST of param. rs

```
1  class StudentManagementSystem {
2    Student [] ss; /* ss[i] has static type [redacted] */ int c;
3    void addRS(ResidentStudent rs) { ss[c] = rs; c ++; }
4    void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5    void addStudent(Student s) { ss[c] = s; c++; } }
```

Q. Static type of ss[0], ss[1], ..., ss[ss.length - 1]?

Student.

Q. In method addRS, does ss[c] = rs compile?

ST: Student ST: RS

call by value: rs = 0 = param arg.

Q. Under what circumstances can the following method call be valid/compilable?

sms.addRS(o) → ST of arg. o should be a descendant of ST of param rs (RS).

Polymorphic Parameters (2)

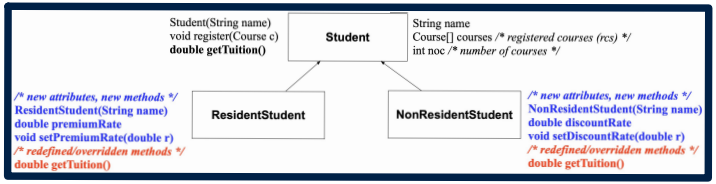
```
1  class StudentManagementSystem {
2    Student [] ss; /* ss[i] has static type Student */ int c;
3    void addRS(ResidentStudent rs) { ss[c] = rs; c ++; }
4    void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5    void addStudent(Student s) { ss[c] = s; c++; } }
```

call by value: rs = ?

```
Student s1 = new Student();
Student s2 = new ResidentStudent();
Student s3 = new NonResidentStudent();
ResidentStudent rs = new ResidentStudent();
NonResidentStudent nrs = new NonResidentStudent();
StudentManagementSystem sms = new StudentManagementSystem();
sms.addRS(s1);   ✗
sms.addRS(s2);   ✗
sms.addRS(s3);   ✗
sms.addRS(rs);   ✓
sms.addRS(nrs);  ✗
sms.addStudent(s1);  ✓
sms.addStudent(s2);  ✓
sms.addStudent(s3);  ✓
sms.addStudent(rs);  ✓
sms.addStudent(nrs); ✓
```

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

call by value: s = ?



Lecture 21 - Nov. 21

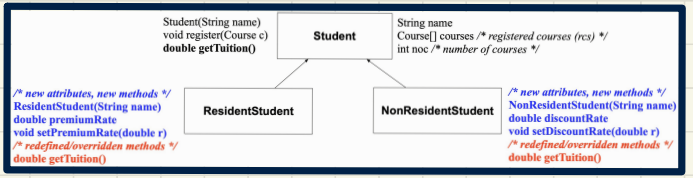
Inheritance

Polymorphic Arrays
Polymorphic Return Values
Type-Checking Rules
Solving Problems Recursively

Announcements/Reminders

- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- **ProgTest3** grading process to start on Monday
- **Exam** Review Sessions eClass Polling
- **Bonus** Opportunity coming: Formal Course Evaluation

Casting Arguments



void addRS(ResidentStudent rs)

sms.addRS((ResidentStudent) s) compiles? (downcast)

```
1 Student s = new Student("Stella");
2 /* s' ST: Student; s' DT: Student */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
```

CCE? ↳ if DT of S is not a descent of cast type RS

ClassCastException? YES.

```
1 Student s = new NonResidentStudent("Nancy");
2 /* s' ST: Student; s' DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
```

ClassCastException? YES.

```
1 Student s = new ResidentStudent("Rachael");
2 /* s' ST: Student; s' DT: ResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
```

ClassCastException? NO.

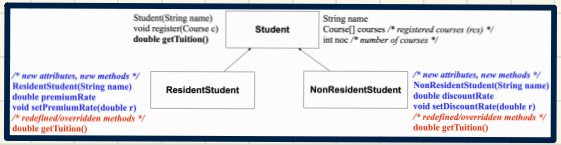
---

sms.addRS((ResidentStudent) nrs) compiles?

```
1 NonResidentStudent nrs = new NonResidentStudent();
2 /* ST: NonResidentStudent; DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(nrs); ×
```

No. Neither upward nor downward cast.

A Polymorphic Collection of Students




```
1  ResidentStudent rs = new ResidentStudent("Rachael");
2  rs.setPremiumRate(1.5);
3  NonResidentStudent nrs = new NonResidentStudent("Nancy");
4  nrs.setDiscountRate(0.5);
5  StudentManagementSystem sms = new StudentManagementSystem();
6  sms.addStudent( rs ); /* polymorphism */
7  sms.addStudent( nrs ); /* polymorphism */
8  Course eecs2030 = new Course("EECS2030", 500.0);
9  sms.registerAll(eecs2030);
10 for(int i = 0; i < sms.numberOfStudents; i ++) {
11   /* Dynamic Binding:
12    * Right version of getTuition will be called */
13   System.out.println(sms.students[i].getTuition());
14 }
```

S = rs

```
class StudentManagementSystem {
  Student[] students;
  int numOfStudents;
  void addStudent(Student s) {
    students[numOfStudents] = s;
    numOfStudents ++;
  }

  void registerAll (Course c) {
    for(int i = 0; i < numberOfStudents; i ++) {
      students[i].register(c)
    }
  }
}
```

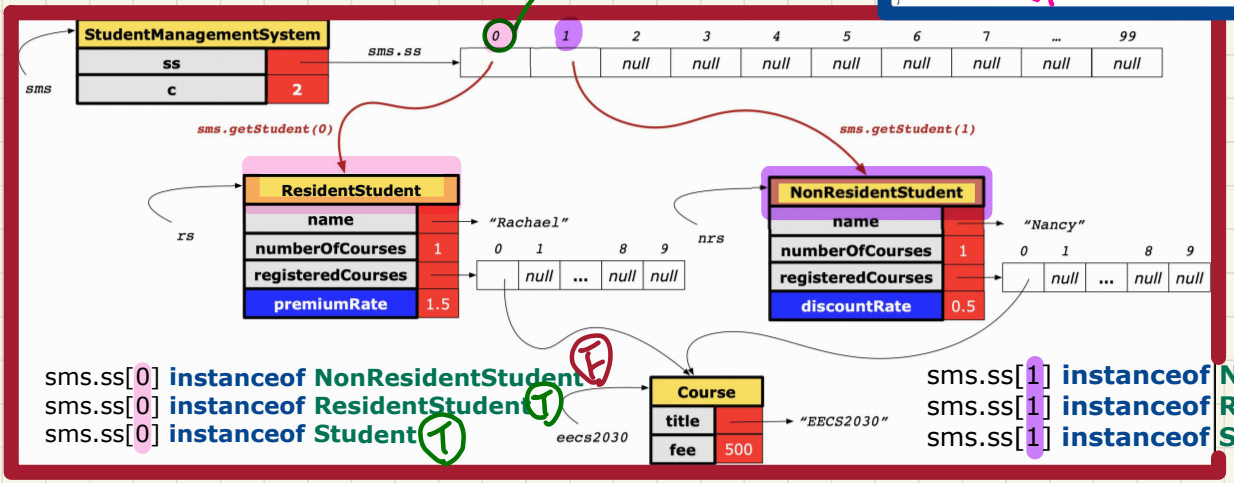
ST of each elem is Student

polymorphism: dynamically s can point to descendants of Student

?

ST: Student
DT: RS, NRS

ss[0]: ST Student.

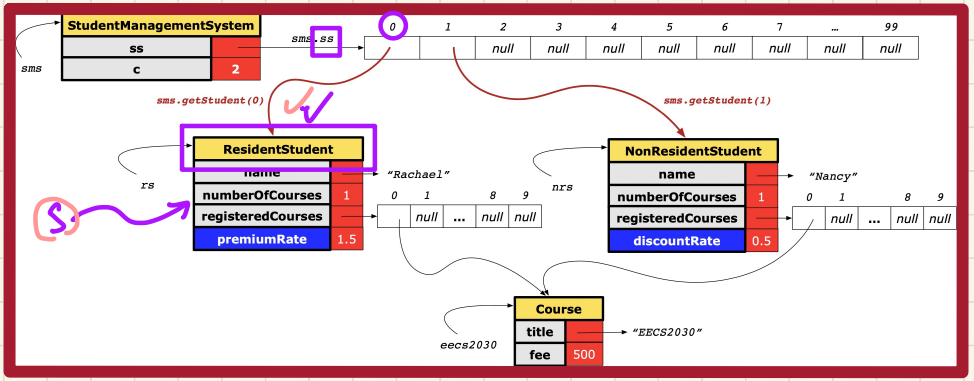


Polymorphic Return Types

```
Course eecs2030 = new Course("EECS2030", 500);
ResidentStudent rs = new ResidentStudent("Rachael");
rs.setPremiumRate(1.5); rs.register(eecs2030);
NonResidentStudent nrs = new NonResidentStudent("Nancy");
nrs.setDiscountRate(0.5); nrs.register(eecs2030);
StudentManagementSystem sms = new StudentManagementSystem();
sms.addStudent(rs); sms.addStudent(nrs);
Student s = sms.getStudent(0) ; /* dynamic type of s? */
                static return type: Student
print(s instanceof Student && s instanceof ResidentStudent);/*tru
print(s instanceof NonResidentStudent); /* false */
print( s.getTuition() );/*Version in ResidentStudent called:750*/
ResidentStudent rs2 = sms.getStudent(0); ×
s = sms.getStudent(1) ; /* dynamic type of s? */
        static return type: Student
print(s instanceof Student && s instanceof NonResidentStudent);/*
print(s instanceof ResidentStudent); /* false */
print( s.getTuition() );/*Version in NonResidentStudent called:250*/
NonResidentStudent nrs2 = sms.getStudent(1); ×
```

dynamic binding: RS version called

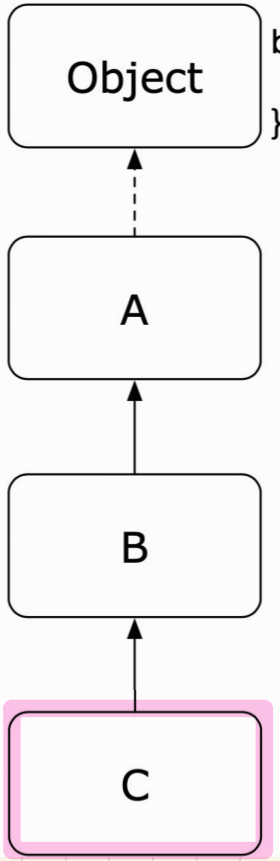
```
class StudentManagementSystem {
 Student[] ss; int c;
 void addStudent(Student s) { ss[c] = s; c++; }
 Student getStudent(int i) {
  Student s = null;
  if(i < 0 || i >= c) {
   throw new IllegalArgumentException("Invalid
  }
  else {
   s = ss[i];
  }
  return s;
 } }
```



Summary: Type Checking Rules

Code	Condition to be Type Correct
`x = y`	Is `y`'s ***ST*** a **descendant** of `x`'s ***ST***?
`x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***?
`z = x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***? Is ***ST*** of `m`'s return value a **descendant** of `z`'s ***ST***?
`(C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***?
`x = (C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is `C` a **descendant** of `x`'s ***ST***?
`x.m((C) y)`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is method `m` defined in `x`'s ***ST***? Is `C` a **descendant** of `m`'s parameter's ***ST***?

Overridden Methods and Dynamic Binding (1)



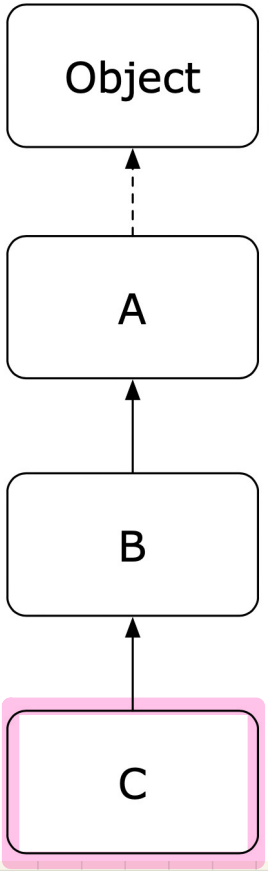

```
boolean equals (Object obj) {
  return this == obj;
}
```

```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  /*equals not overridden*/
}
```

```
1  Object c1 = new C();
2  Object c2 = new C();
3  println(c1.equals(c2));
```

L3 calls which version of `equals`? [`Object`]

Overridden Methods and Dynamic Binding (2)




```
boolean equals (Object obj) {
  return this == obj;
}
```

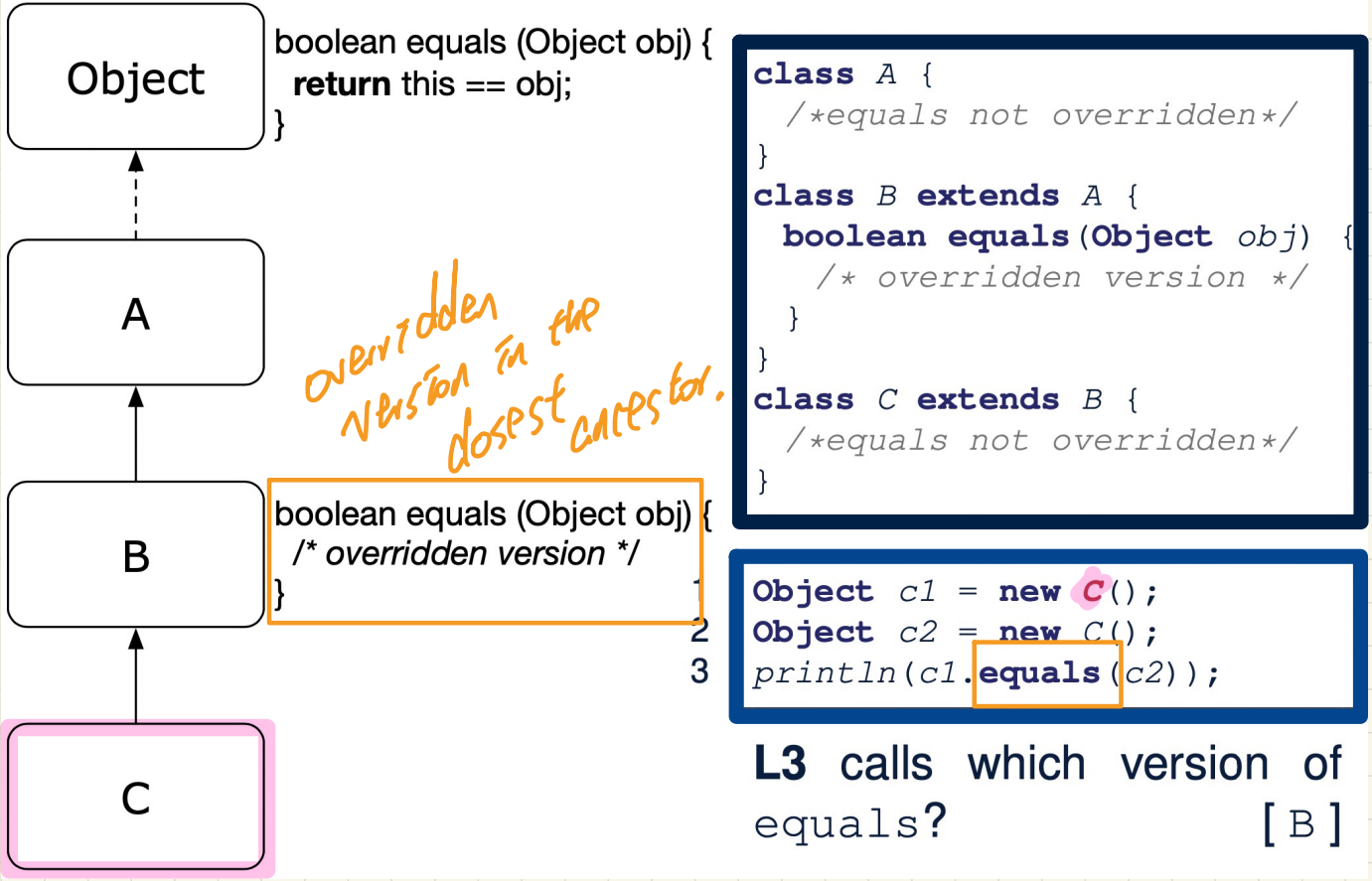
```
boolean equals (Object obj) {
  /* overridden version */
}
```

```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
```

```
1 Object c1 = new C();
2 Object c2 = new C();
3 println(c1.equals(c2));
```

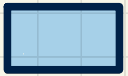
L3 calls which version of `equals`? [C]

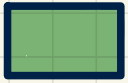
Overridden Methods and Dynamic Binding (3)



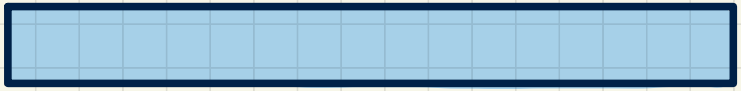
Solving a Problem Recursively

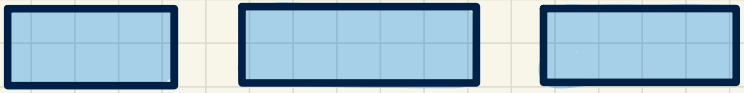
Base Case

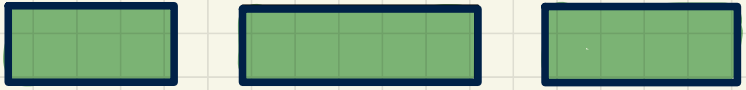
Given a small problem: 

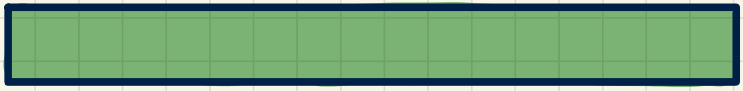
Solve it directly: 

Recursive Case

Given a big problem: 

Divide it into smaller problems: 

Assume solutions to smaller problems: 

Combine solutions to smaller problems: 

given problem

```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

must be a strictly smaller problem ($j < i$)

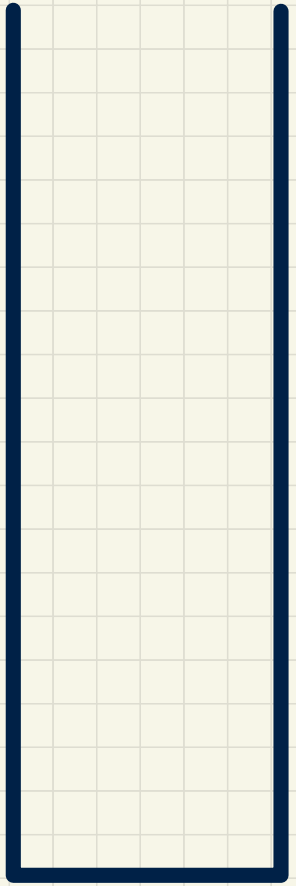
calling m itself ⇒ recursion

Tracing Recursion via a Stack

- When a method is called, it is ***activated*** (and becomes *active*) and *pushed* onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and *pushed* onto the stack.

  ⇒ The stack contains activation records of all *active* methods.
  - *Top* of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

  ⇒ The current point of execution is returned to the new *top* of stack (which was ***suspended*** and just became ***active***).
- Execution terminates when the stack becomes *empty*.



Runtime Stack

Recursive Solution: factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

not a recursive definition.

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{(base case)} \\ (n-1)! \cdot n & \text{if } n \geq 1 \quad \text{(recursive case)} \end{cases}$$

$n!$: problem of size n

$(n-1)!$: a strictly smaller problem

$(n-1)!$: factorial problem

Lecture 22 - Nov. 26

Recursion

Tracing: Factorial, Fibonacci Sequence
Recursion on Str.: Palindrome, Reversal
Exam Info

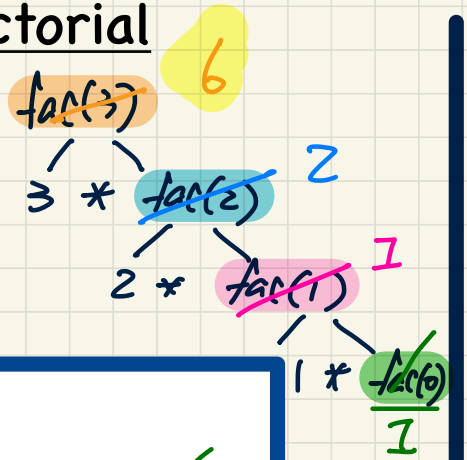
Announcements/Reminders

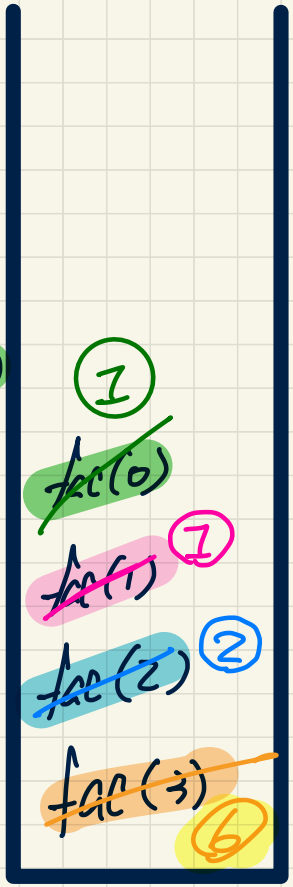
- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- Learning resources on **Recursion**
- **Exam** Review Sessions eClass Polling

Recursive Solution in Java: factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

```
int factorial (int n) {
  int result;
  if (n == 0) { /* base case */ result = 1; }
  else { /* recursive case */
    result = n * factorial (n - 1);
  }
  return result;
}
```



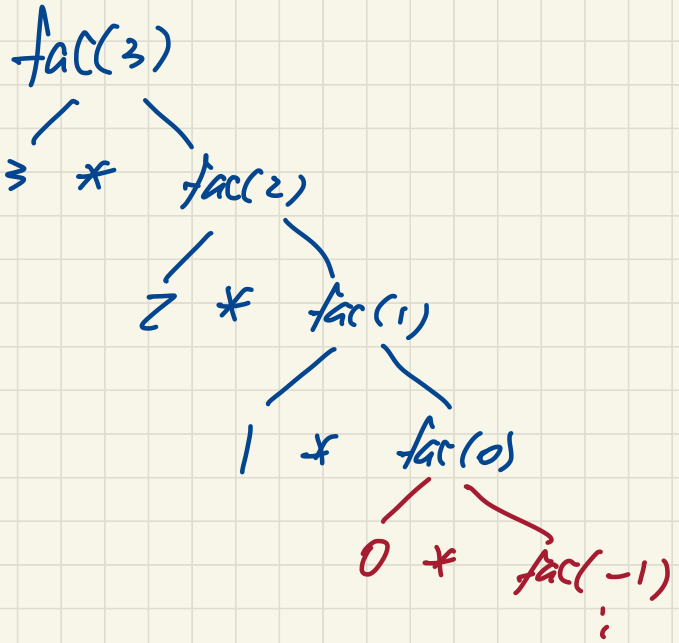


Runtime Stack

Example: factorial(3)

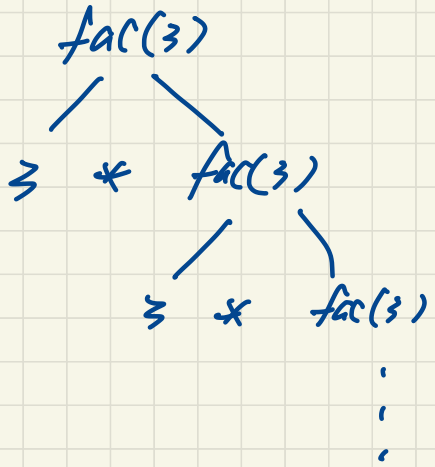
Common Errors of Recursion (1)

```
int factorial (int n) {
  return n * factorial (n - 1);
}
```

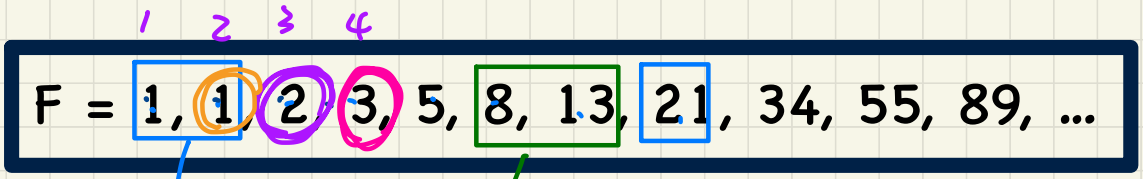


Common Errors of Recursion (2)

```
int factorial(int n) {
  if(n == 0) { /* base case */ return 1; }
  else { /* recursive case */ return n * factorial(n); }
}
```

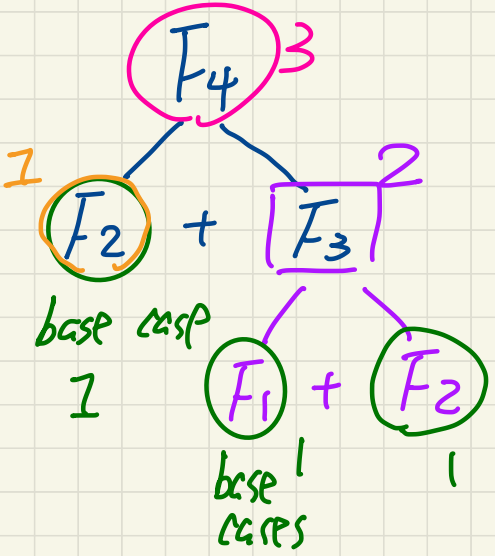


Recursive Solution: Fibonacci Numbers



F_1

F_2



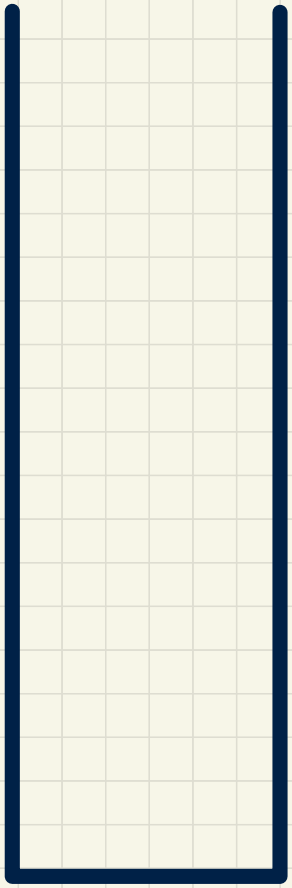
Recursive Solution in Java: Fibonacci Numbers

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
int fib(int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib(n - 1) + fib(n - 2);
  }
  return result;
}
```

two recursive calls

Example: fib(4)

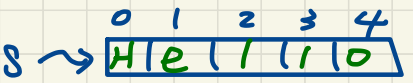


Runtime Stack

Use of String

s.length() = 4

s ~> "Hello"




```
public class StringTester {
  public static void main(String[] args) {
    String s = "abcd";
    System.out.println(s.isEmpty()); /* false */
    /* Characters in index range [0, 0) */
    String t0 = s.substring(0, 0);
    System.out.println(t0); /* "" */
    /* Characters in index range [0, 4) */
    String t1 = s.substring(0, 4);
    System.out.println(t1); /* "abcd" */
    /* Characters in index range [1, 3) */
    String t2 = s.substring(1, 3);
    System.out.println(t2); /* "bc" */
    String t3 = s.substring(0, 2) + s.substring(2, 4);
    System.out.println(s.equals(t3)); /* true */
    for(int i = 0; i < s.length(); i ++) {
      System.out.print(s.charAt(i));
    }
    System.out.println();
  }
}
```

0 1 2 3

[0, -1] i > j → Empty

s.substring(i, j)

≤

[i, j)

"

[i, j-1]

For any given string s,

given int. i s.t. $0 \leq i \leq s.length() - 1$

✓ $s.equals(s.substring(0, i) + s.substring(i, \leq s.length()))$

Recursions on Strings

ex. "abcd"

Palindrome

"racecar" → strictly smaller problem

"aracecars" != "raceacar"

no need to recur further.

Reversal

rev("abcd")

rev("bcd") + a

d c b

Number of Occurrences

"abca"

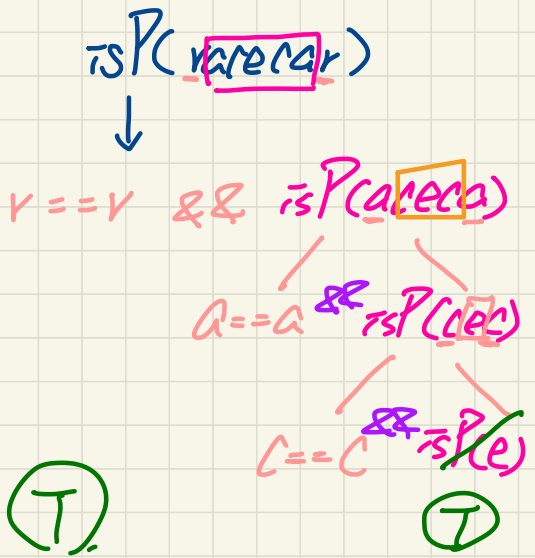
'a'

'b'

Problem: Palindrome

```
boolean isPalindrome (String word) {
  if(word.length() == 0 || word.length() == 1) {
    /* base case */
    return true;
  }
  else {
    /* recursive case */
    char firstChar = word.charAt(0);
    char lastChar = word.charAt(word.length() - 1);
    String middle = word.substring(1, word.length() - 1);
    return
        firstChar == lastChar
        /* See the API of java.lang.String.substring. */
        && isPalindrome (middle);
  }
}
```

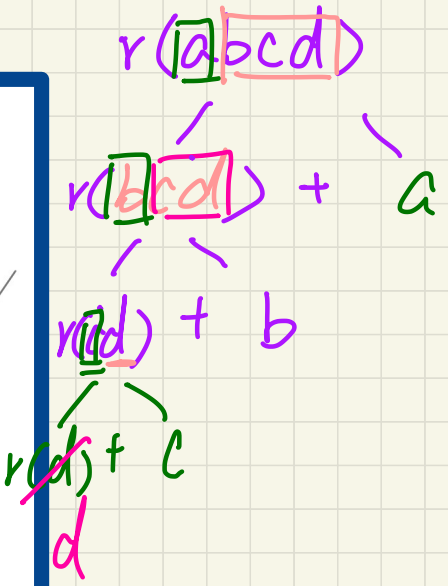
string with 1st & last char removed



ex. isP(racebar)

Problem: Reverse of a String

```
String reverseOf (String s) {
  if(s.isEmpty()) { /* base case 1 */
    return "";
  }
  else if(s.length() == 1) { /* base case 2 */
    return s;
  }
  else { /* recursive case */
    String tail = s.substring(1, s.length());
    String reverseOfTail = reverseOf (tail);
    char head = s.charAt(0);
    return reverseOfTail + head;
  }
}
```



Exam Info

- When: 10am to 1pm, Sunday, December 8
- Where: TC Sobeys
- Format: Some Multiple Choice & Mostly Written
- Coverage: **Everything** (lecture materials & labs)
  + slides, iPad notes, code examples
  + https://codingbat.com/java/Recursion-1
- Restrictions:
  + No data sheet
  + No sketch paper (Exam booklet includes it)
- What you should bring:
  + Valid, Physical Photo ID (strict)
  + Water/Snack

1. explanation
2. output
3. justification
4. code

Lecture 23 - Nov. 28

Recursion

Recursion on Strings: occurrencesOf
Recursion on Arrays: Call by Value
Recursion on Arrays: allPositive, isSorted

Announcements/Reminders

- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- Learning resources on **Recursion**
- **Exam** Review Sessions eClass Polling
- A tutorial session on **recursion**: 4pm on Sunday
- **ProgTest3** results to be released on Monday

Recursions on Strings

ex. "abcd"

Palindrome

"racecar" → strictly smaller problem

"aracecars" != "raceacar"

no need to recur further.

Reversal

rev("abcd")

rev("bcd") + a

d c b

Number of Occurrences

"abca"

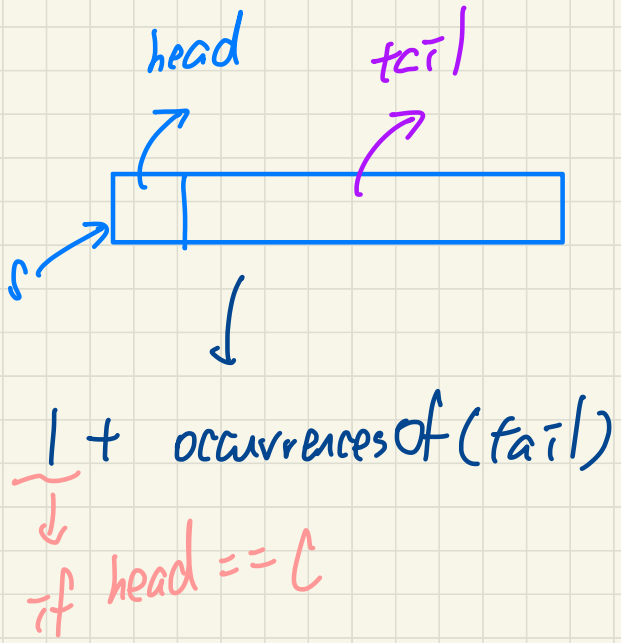
of 'a'

of 'a' → 1 + occurrences of ("bca", 'a')

of 'b' → 0 + occurrences of ("bca", 'b')

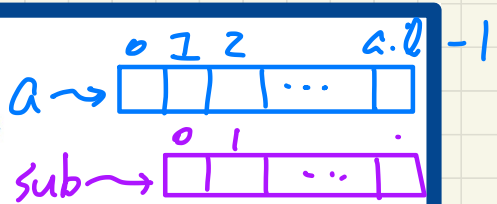
Problem: Number of Occurrences

```
int occurrencesOf(String s, char c) {
  if(s.isEmpty()) {
    /* Base Case */
    return 0;
  }
  else {
    /* Recursive Case */
    char head = s.charAt(0);
    String tail = s.substring(1, s.length());
    if(head == c) {
      return 1 + occurrencesOf(tail, c);
    }
    else {
      return 0 + occurrencesOf(tail, c);
    }
  }
}
```



Recursion on an Array: Passing new Sub-Arrays

```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```

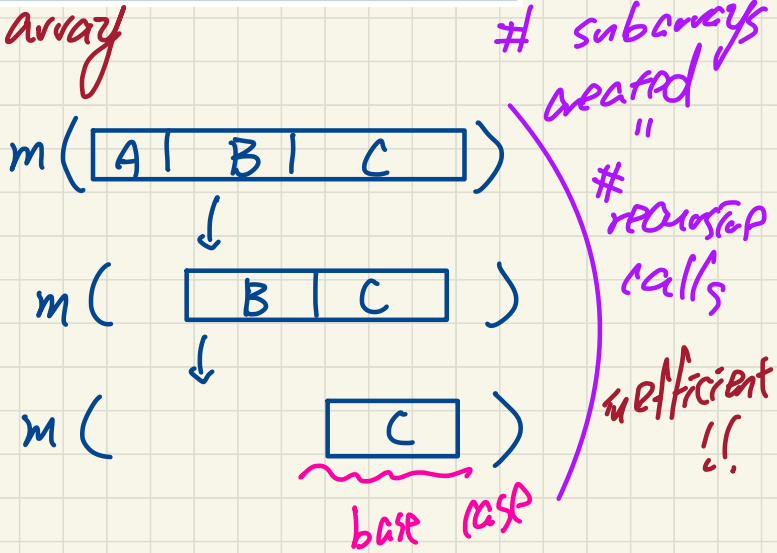


recursive call on a strictly smaller array

Say a1 = {}, consider m(a1)

↳ reached base case

Say a2 = {A, B, C}, consider m(a2)



Recursion on an Array: Passing Same Array Reference

non-base case: from < to

```
void m(int[] a, int from, int to) {
  if(from > to) { /* base case */ }
  else if(from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

empty range

single ele. range

Say a1 = {}, consider m(a1, 0, a1.length - 1)

length 0

-1

0 > -1 → base case #1

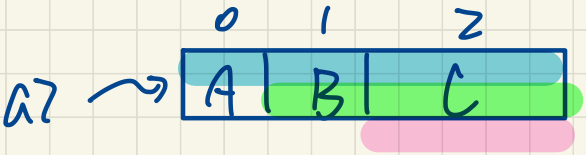
(from) (to)

Say a2 = {A, B, C}, consider m(a2, 0, a2.length - 1)

m(a2, 0, 2)

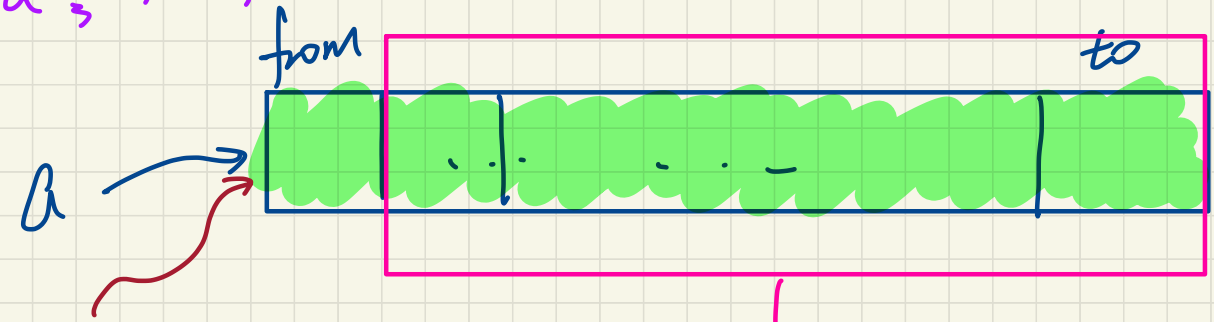
m(a2, 1, 2)

m(a2, 2, 2)



base case #2

m(a, from, to)



m(a, from+1, to)

strictly smaller subrange.

Problem: Are All Numbers Positive?

Is there some positive number?

$\exists x \cdot P(x)$ → False ∵ no witness to show satisfy.

$x \in \phi$

Say a = {}

$\forall x \cdot P(x) \equiv$ True → ∵ no witness in empty collection to show violation of property.

$x \in \phi$

Say a = {4} True

Say a = {4,7,3,9} True

0 1 2 3

allPos({4,7,3,9}, 0, 3)

a[0] > 0 && allPos(a, 1, 3)

Say a = {5,3,-2,9} False

Problem: Are All Numbers Positive?

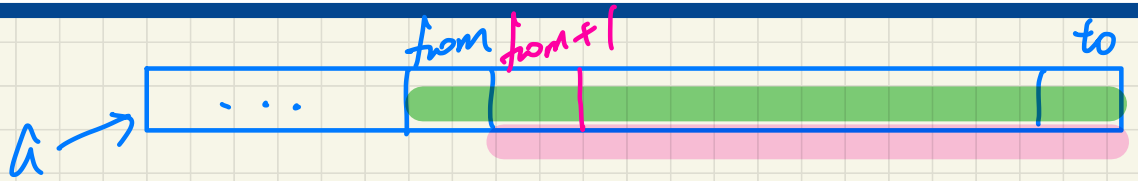
what a user needs to supply when using the solution

public

private

recursive helper method

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```



Tracing Recursion: allPositive

non-descending : $\neg(a[i] > a[i+1])$

non-ascending ? → $a[i] \leq a[i+1]$

Say a = {}

allPositive(a)

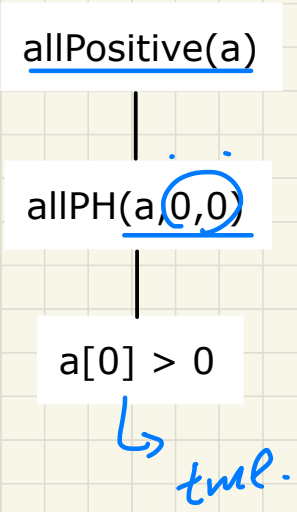
allPH(a,0,-1)

tail.

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

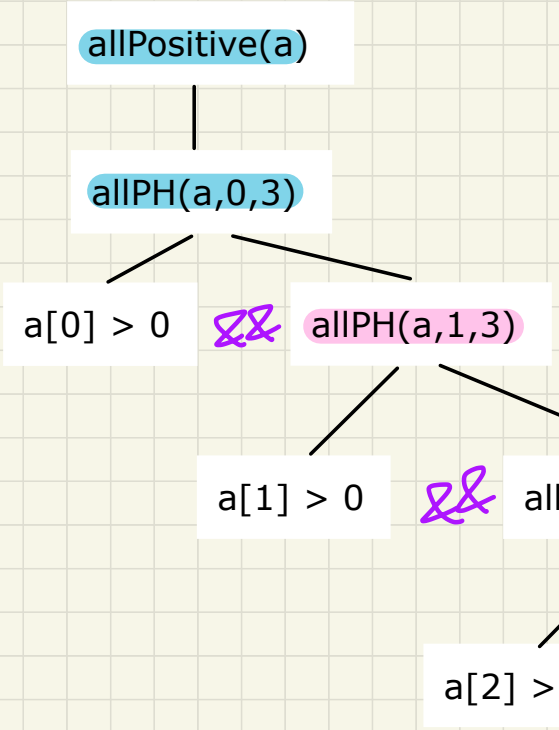
Say a = {4}



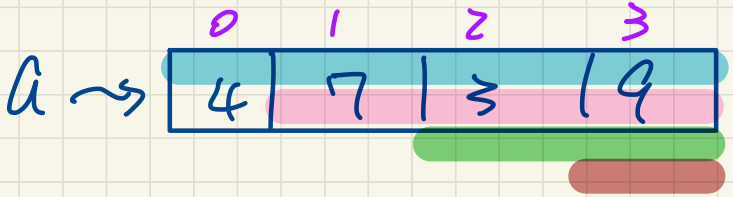

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

Say a = {4,7,3,9}



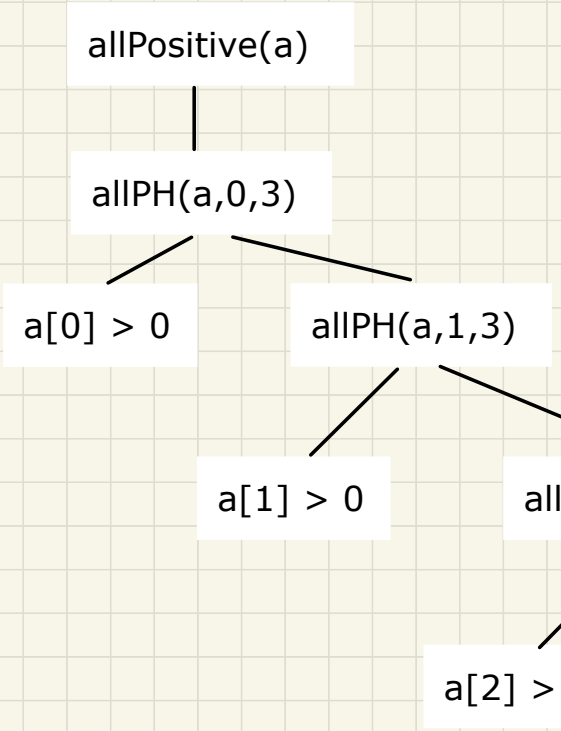

```java
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```



a[3]>0

Tracing Recursion: allPositive

Say a = {5,3,-2,9}




```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Problem: Are Numbers Sorted?

Say a = {} True

Say a = {4} True

Say a = {3,6,6,7}

isSorted(a, 0, 3)

a[0] ≤ a[1] && isSorted(a, 1, 3)

3 ≤ 6

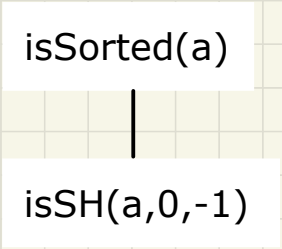
Say a = {3,6,5,7}

Problem: Are Numbers Sorted?

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

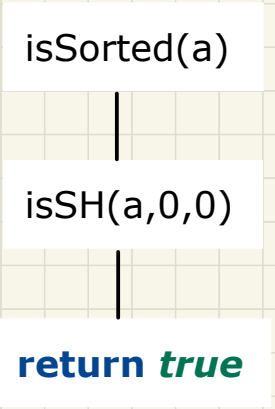
Say a = {}




```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

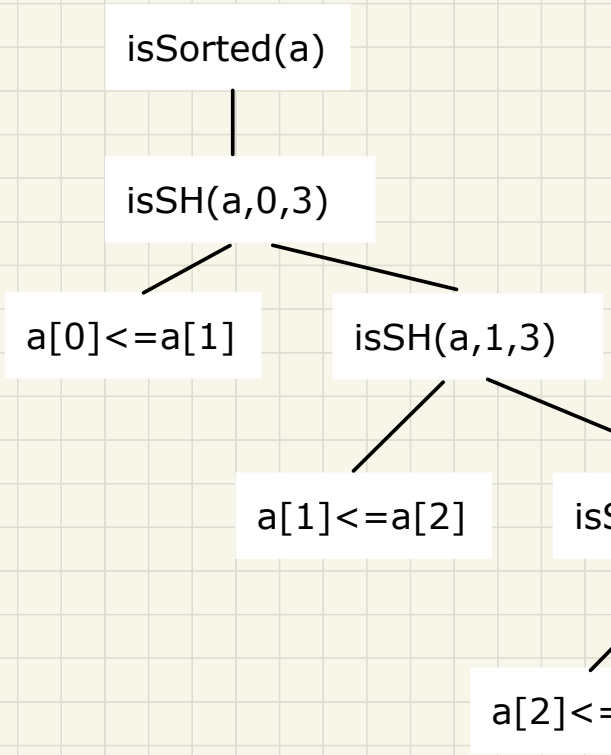
Say a = {4}




```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

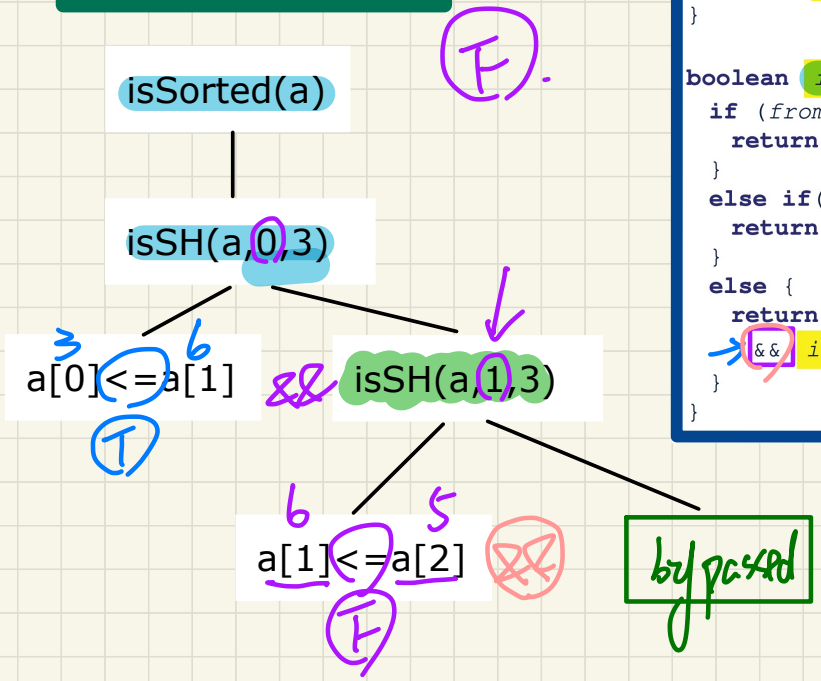
Say a = {3,6,6,7}



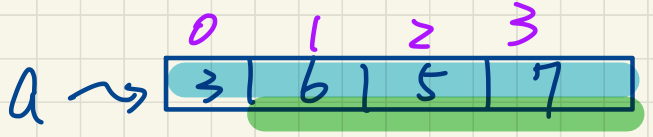

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

Say a = {3,6,5,7}




```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
    && isSortedHelper(a, from + 1, to);
  }
}
```



Recursion

Tutorial

triangle
array220
arithmeticArray

Problem on Recursion

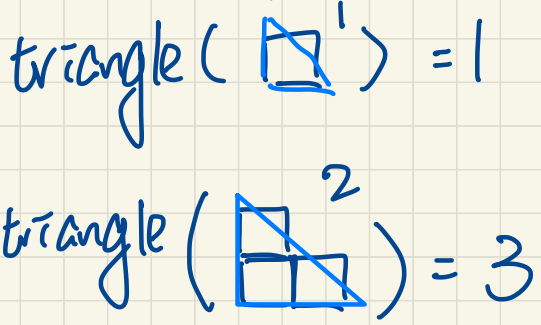
https://codingbat.com/prob/p194781

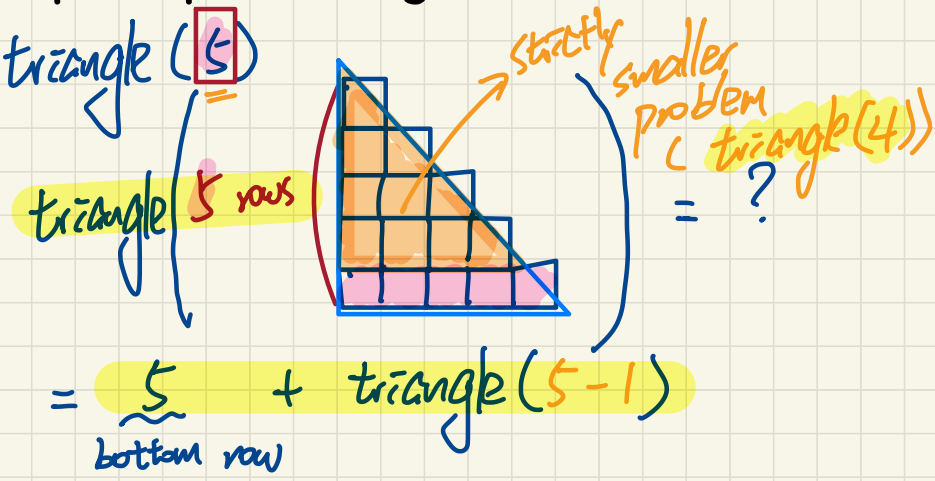
We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

# of rows

triangle(0) → 0
triangle(1) → 1
triangle(2) → 3

Hint: Visually, how do the example input triangles look like?



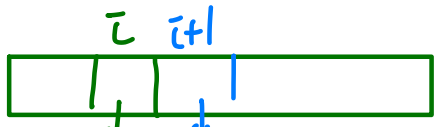


Problem on Recursion

https://codingbat.com/prob/p173469

Given an array of ints, compute recursively if the array contains somewhere a value followed in the array by that value times 10. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass index+1 to move down the array. The initial call will pass in index as 0.

array220([1, 2, 20], 0) → true
array220([3, 30], 0) → true
array220([3], 0) → false



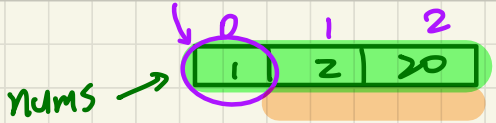
Try two versions:

boolean array220(**int**[] nums, **int** from)
boolean array220(**int**[] nums, **int** from, **int** to)

Ver. 1 assumes that the 'to' index denotes the end of the array.

Ver. 2 does not make such an assumption, though 'to' typically is the last index.

Hint: Max value of 'from' before an ArrayIndexOutOfBoundsException occurs?



array220(nums, 0)
nums[1] == nums[0]*10
from
|| array220(nums, 1)

Problem on Recursion

Return an array storing the first n numbers in an arithmetic sequence.

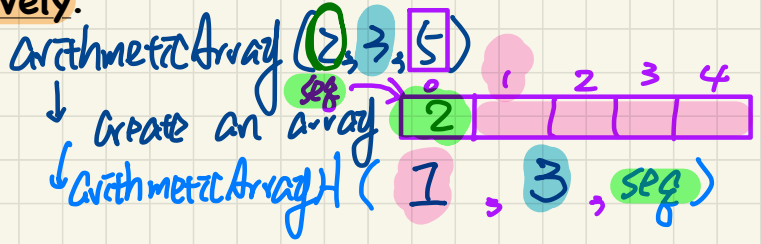
int[] arithmeticArray(int start, int diff, int n)

'start': the first term in an arithmetic sequence
'diff': the common difference between terms in an arithmetic sequence
'n': the first n numbers in an arithmetic sequence

e.g., arithmeticArray(2, 3, 5) returns an array {2, 5, 8, 11, 14}.

Hint: Let this method first create an array of the right size, then pass its **reference** to a recursive helper method, which modifies the array contents **recursively**.

arithmeticArray(2, 3, 0) → {}
arithmeticArray(2, 3, 1) → {2}



Lecture 24 - Dec. 3

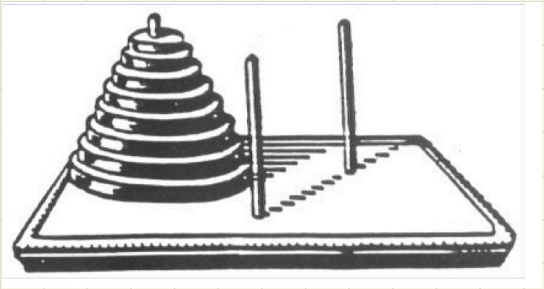
Recursion

Tower of Hanoi: Specification, Legend
Tower of Hanoi: Java, Tracing
Tower of Hanoi: Running Time

Announcements/Reminders

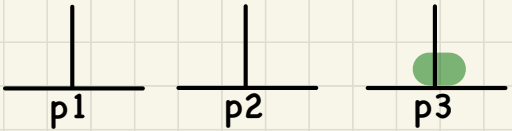
- **Lab5** due **midnight** today
  + Required study: **Abstract Classes** & **Interfaces**
- **ProgTest3** results released
- Extra office hours: 3pm to 5pm on Thursday
- **Exam** Review Session (Zoom): 3pm on Friday
- Materials for tutorial session on **recursion**

Tower of Hanoi: Strategy



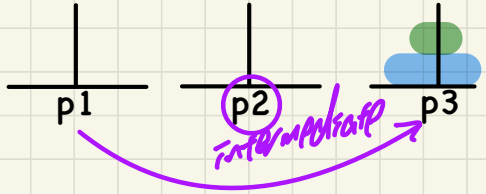
Consider 1 disk: A

move  from p1 to p3



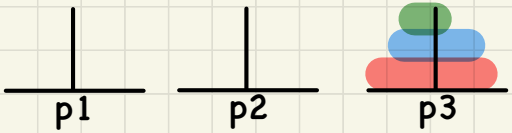
Consider 2 disks: A < B

move  from p1 to p3

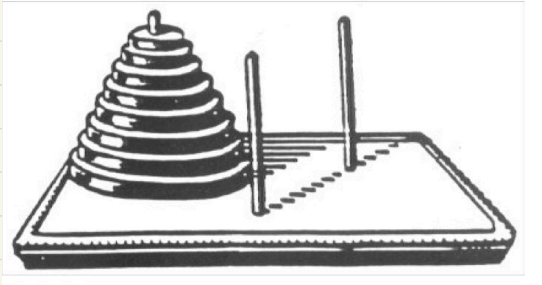


Consider 3 disks: A < B < C

move  from p1 to p3



Tower of Hanoi: Strategy

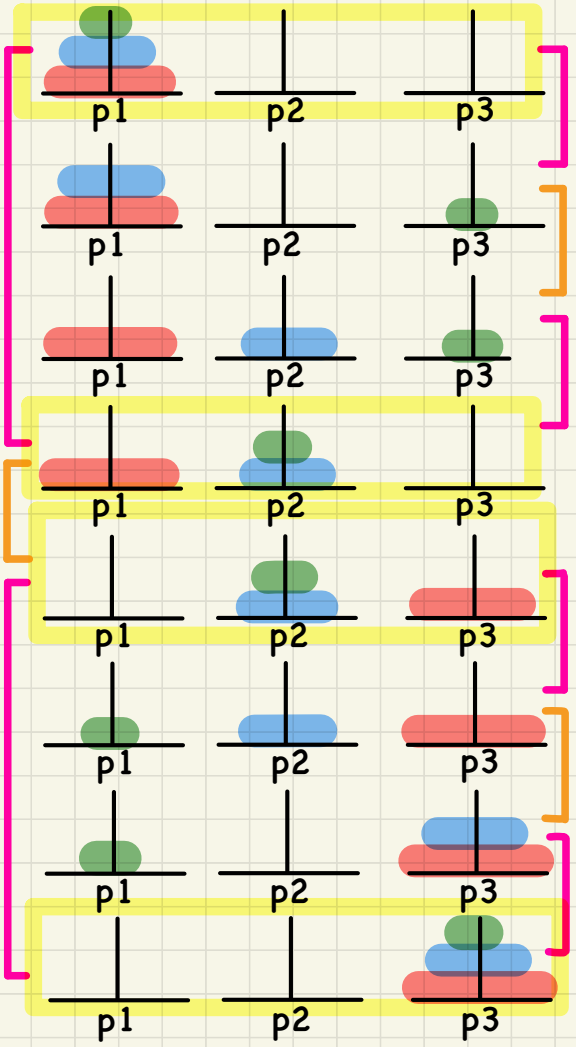


Consider 3 disks A < B < C

move  from p1 to p3

move  from p1 to p2

move  from p2 to p3



move  from p1 to p3

move  from p1 to p2

move  from p3 to p2

move  from p1 to p3

move  from p2 to p1

move  from p2 to p3

move  from p1 to p3

Tower of Honoi in Java

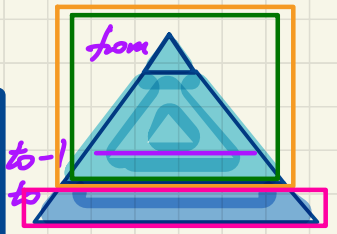

```
void towerOfHanoi(String[] disks) {
  tohHelper(disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int ori, int des){
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + ori + " to " + des);
  }
  else {
    int intermediate = 6 - ori - des;
  ① tohHelper(disks, from, to - 1, ori, intermediate);
  ② print("move " + disks[to] + " from " + ori + " to " + des);
  ③ tohHelper(disks, from, to - 1, intermediate, des);
  }
}
```

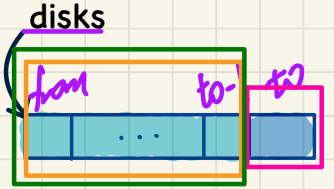
peg 1 peg 3

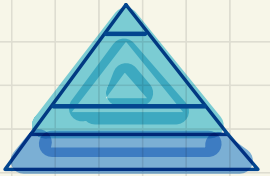
range of array/tower

only one disk in the tower

p2 to p3 → p(6-2-3) → p1

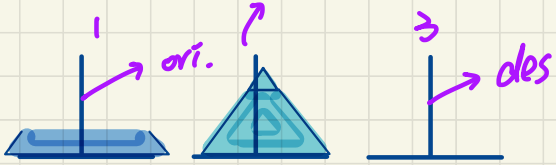






Stage 1

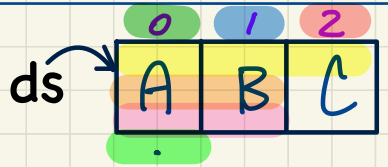
int: 6-1-3 = 2

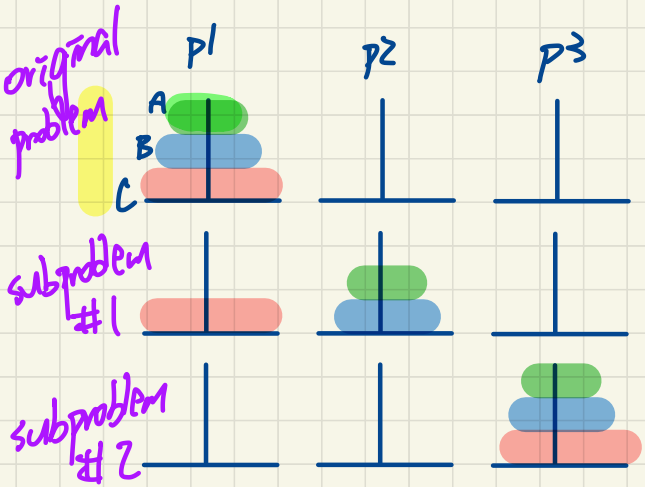


Stage 2

Stage 3

Tower of Hanoi: Tracing





tohH(ds, 0, 2, p1, p3)

subproblem #1

tohH(ds, 0, 1, p1, p2) ④ move C: p1 to p3 tohH(ds, 0, 1, p2, p3)

subproblem #2

intermediate: p3

① tohH(ds, 0, 0, p1, p3) → move A: p1 to p3

base case

② move B: p1 to p2

③ tohH(ds, 0, 0, p3, p2) → move A: p3 to p2

base case

⑤

⑥

⑦.

EXERCISE

Tower of Hanoi: Tracing

Say *ds* (disks) is $\{A, B, C\}$, where $A < B < C$.

$$
tohH(ds, \underbrace{0,2}_{\{A,B,C\}}, p1, p3) = \begin{cases}
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p1, p2) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3} \\
\texttt{Move B: p1 to p2} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p3, p2) = \{ \texttt{Move A: p3 to p2}
\end{cases} \\
\texttt{Move C: p1 to p3} \\
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p2, p3) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p2, p1) = \{ \texttt{Move A: p2 to p1} \\
\texttt{Move B: p2 to p3} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3}
\end{cases}
\end{cases}
$$

Tower of Hanoi: Running Time

$T(n) = ?$ $T(n-[?])$ $n-1$ $= T(1)$

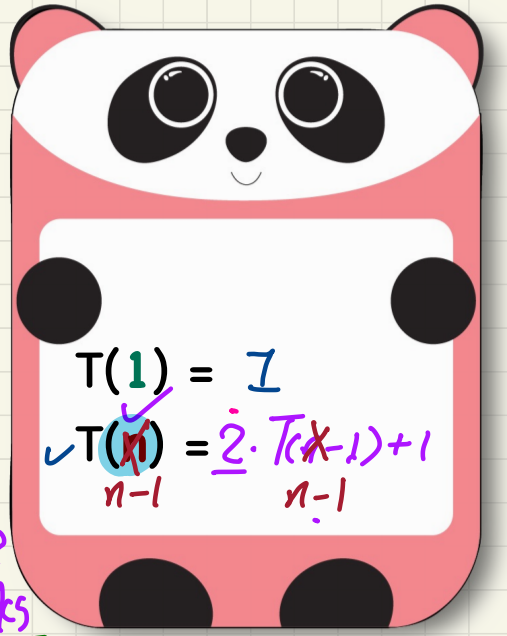
Running Time as a Recurrence Relation

```
void towerOfHanoi(String[] disks) {
  tohHelper(disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int ori, int des) {
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + ori + " to " + des);
  }
  else {
    int intermediate = 6 - ori - des;
    tohHelper(disks, from, to - 1, ori, intermediate);
    print("move " + disks[to] + " from " + ori + " to " + des);
    tohHelper(disks, from, to - 1, intermediate, des);
  }
}
```

T(n) — n disks

T(n-1) — 1.

T(n-1)



$T(n) = 2 \cdot T(n-1) + 1$

$= 2 \cdot (2 \cdot T(n-2) + 1) + 1$

$= 2 \cdot (2 \cdot (2 \cdot T(n-3) + 1) + 1) + 1$

2^3 — $1 + 3$

$2^{n-1} + (n-1)$ seconds to complete n disks

$= 2 \cdot (2 \cdots \cdot T(1)) + 1 + 1 + 1$

2^{n-1} — 1 — $n-1$

Final Exam

Review Q&A

Consider a mutator method `m` defined in class `SomeClass`.

Assume that the **correct** implementation of method `m`is such that:

- When `m` is invoked for the first time, no exceptions occur.
- When `m` is invoked for the second time, a `SomeExceptoin` occurs.

```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      obj.m();
      fail();
    }
    catch(SomeException e) {
    }
  }
}
```

```
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
        System.out.println("Fail");
      }
      catch(SomeException e) {

      }
    }
    catch(SomeException e) {
    ✓System.out.println("Fail");
    }
  }
}
```

appropriate

1st call → expect no exception
↳ if exc. occurred → caught and "fail" printed
if exc. not occurred → proceed with normal flow

2nd call → expect SomeExc.

reaching this line means the expected SomeEx did not occur.

```
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {

    }
    try {
      obj.m();
      fail();
    }
    catch(SomeException e) {

    }
  }
}
```

SomeExc occurred (unexpectedly)

no fail!

inappropriate!

```
public class Tester { /* This is a JUnit tester. *
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
        fail();
      }
      catch(SomeException e) {

      }
    }
    catch(SomeException e) {
      fail();
    }
  }
}
```

appropriate but unnecessarily complex

```java
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {
      System.out.println("Fail");
    }
    try {
      obj.m();
      System.out.println("Fail");
    }
    catch(SomeException e) {

    }
  }
}
```

SomeExc. happ.

As soon as a fail-ure occurres, normal flow of exec should be interrupted

should not be continued if a "fail" occurrs already.

↳ (need to have nested try-catch)

```java
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
    }
    catch(SomeException e) {
      fail();
    }
    try {
      obj.m();
      fail();
    }
    catch(SomeException e) {

    }
  }
}
```

As soon as the first "fail" occurs, the flow is interrupted. (no need to have nested try-catch)

```java
public class Tester { /* This is a JUnit tester. */
  /* import of JUnit assertions omitted */
  @Test
  public void test() {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      try {
        obj.m();
      }
      catch(SomeException e) {
        /* occurred as expected */
      }
    }
    catch(SomeException e) {
      fail();
    }
  }
}
```

1st call

2nd call → SomeEx. → pass

otherwise → fail()

```java
public class Tester { /* This is a console tester. */
  public static void main(String[] args) {
    SomeClass obj = new SomeClass();
    try {
      obj.m();
      obj.m();
      System.out.println("Fail");
    }
    catch(SomeException e) {
      System.out.println("Fail");
    }
  }
}
```

① ② 1st & 2nd calls have opposite ext?

can be from ① or ② (inappropriate)

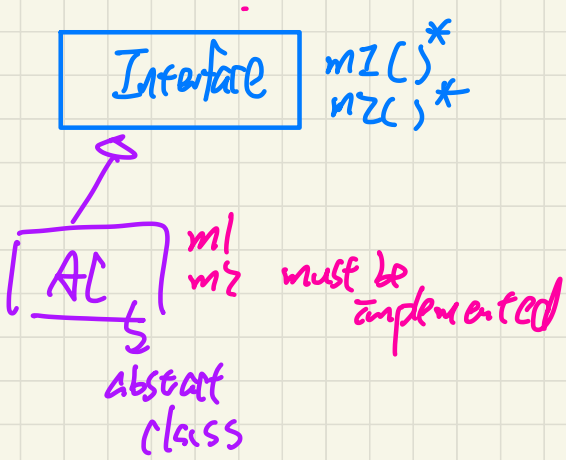
if the expected SomeEx. did not occur, the test would not fail!

inappropriate

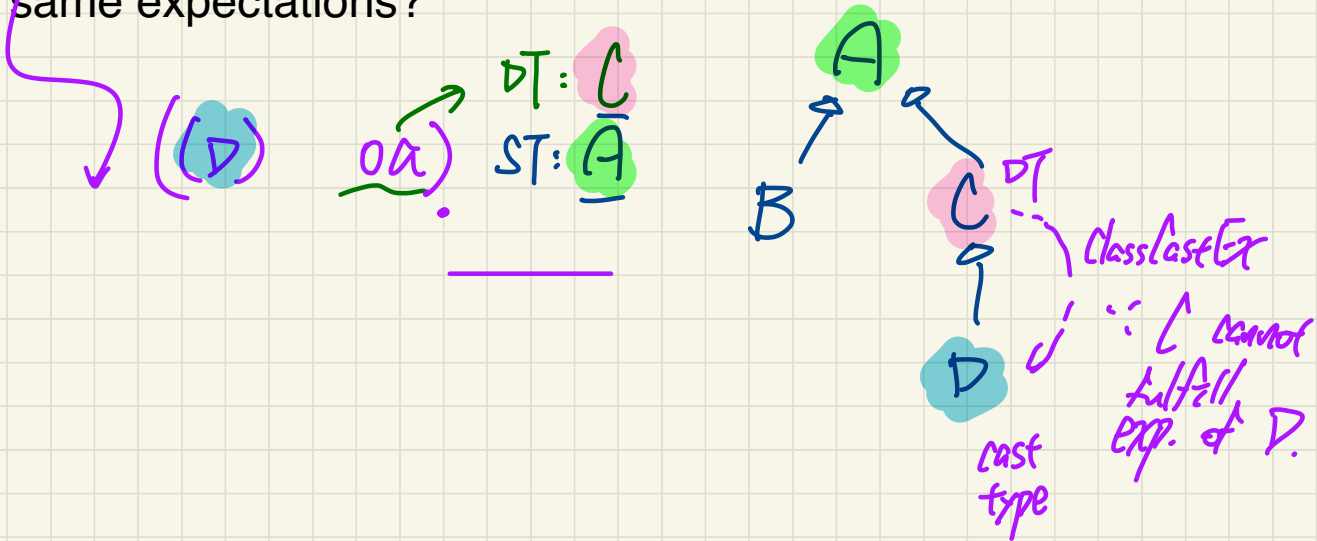
Past lab on recursion

↳ only focus on arrays

↳ ArrayList not covered.



Are descendant classes only useful if they have more expectations than their ancestor classes? If not, on page 5 of your written notes on "Static Types, Expectations, Dynamic Types, and Type Casts", when you cast (D) oa, it does not work because the "d" attribute is not declared in class C. But if we suppose that class D simply didn't have the "od.d" expectation (so only "a" and "c" attributes), would it still cause a ClassCastException even though D and C would have the same expectations?



Given a string and a non-empty substring **sub**, compute recursively the largest substring which starts and ends with sub and return its length.

strDist("catcowcat", "cat") → 9
strDist("catcowcat", "cow") → 3
strDist("cccatcowcatxx", "cat") → 9

hints on recursive thinking to come!

```
1  class Collector {
2    A[] as; int numberOfAs;
3    B[] bs; int numberOfBs;
4    Collector() {
5      as = new A[10]; bs = new B[10]; }
6    void addA(A a) {
7      as[numberOfAs] = a; numberOfAs++; }
8    void addB(B b) {
9      bs[numberOfBs] = b; numberOfBs++; }
10   void callAll() {
11     for(int i = 0; i < numberOfAs; i ++)
12     { as[i].mi(); }
13     for(int i = 0; i < numberOfBs; i ++)
14     { bs[i].mi(); }
15   }
16 }
```

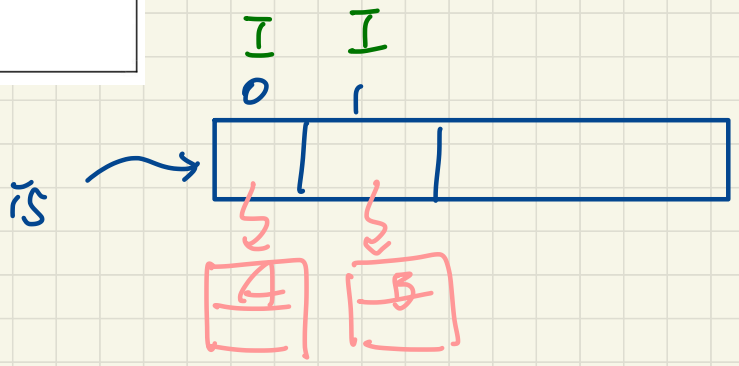
I[] is;

is

```
void addI(I i) {
  is[no1] = i; no1++
}
```

① C.addI(new A());

② C.addI(new B());



I hope you enjoyed learning with me 🤗

All the best to you !