

EECS2030 (Sec. E) 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

Progtest

[].

- WSC
- Eclipse

- format $\approx$ Lab1 $\sim$ Lab5

grading

1. Compile!

2. starters

3. extra grading tests

↳ 1. Informal Problem descriptions

2. JUnit test cases

(starters)

basic.

develop
Java project

inferred

extra grading tests

Course Learning Outcomes (CLOs)

void m()
(int i)

CLO1 Implement an Application Programming Interface (API).

CLO2 Test the implementation.

Unit test vs. S.O.P.

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.

OOP Review Question

```
class A {  
    int i;  
    B b1;  
    B[] b2;  
}  
  
class B { ... }
```

primitive (circled around 'int')

reference (circled around 'B' and 'B[]')

[23]
└

Obj23

b1

memory

Obj23

object
instantiated
from class B

b2 [Obj23]

Obj23

address
of some
object B

b2

┌ ──┴──┐
│ │ │ │ │
└ ──┴──┘

└ ──┴──┘

Lecture 2 - Sep. 10

Review of OOP

Observe-Model-Execute Process

Java Data Types

NullPointerException

Parameters vs. Arguments

Scope of Variables

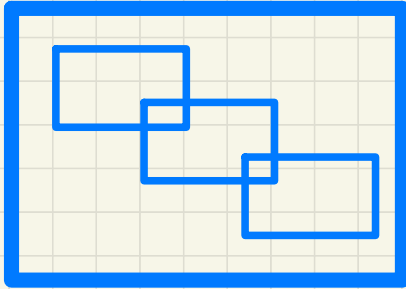
Announcements/Reminders

- Priority: LabOP1 (tutorial videos + PDFs)
- LabOP2 released soon
- Wednesday's lab: in-lab code demo!

helper
methods

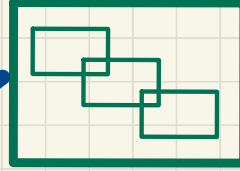
Separation of Concerns

model



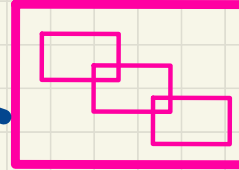
- Classes & Methods
- Methods
 - * constructors
 - * accessors: **return** statements
 - * mutators: **no return** statements
 - * containing **no** print statements

junit_tests



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

console_apps



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

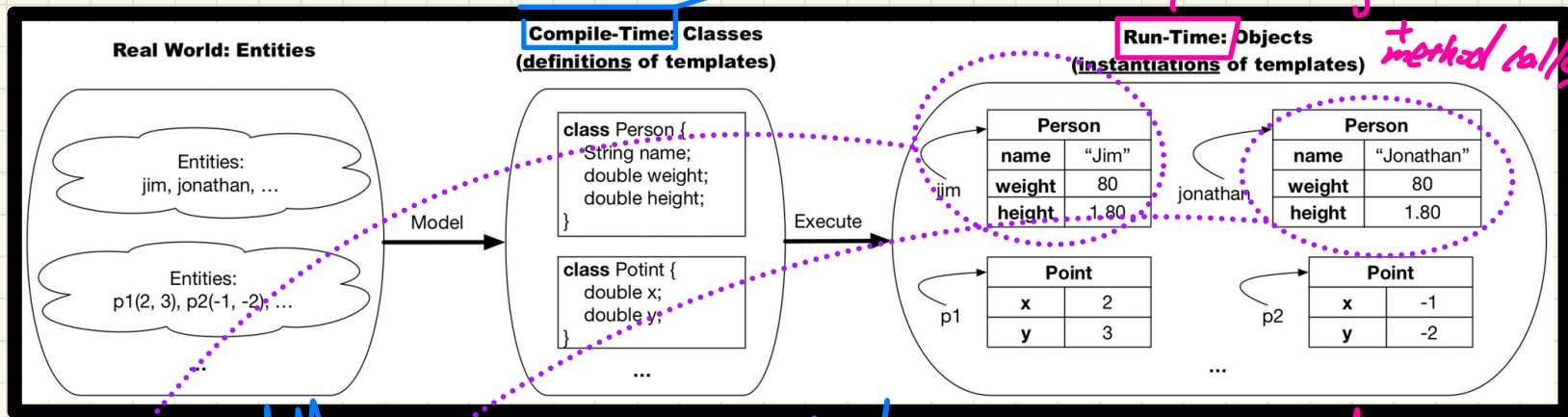
use

use

Observe-Model-Execute Process

developing
code in
editor

1. everything
compiles
2. object creations
+ method calls



1. different entities of the same kind



jim



jonathan

Entities: jim, jonathan

Attributes: weight, height

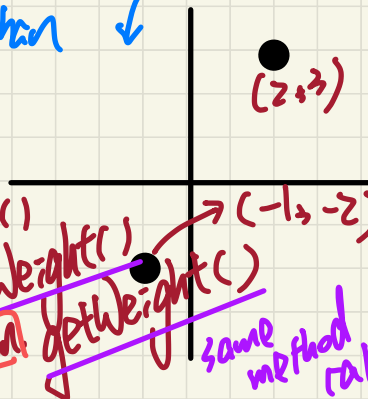
Changes:

Inquiries:

Template:

different context obj. jim jonathan
getWeight() getWeight() getWeight()
same method call

Exercise



3. break points + debugger

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**
- 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

change ↪ rename an attr
↪ rename a class
↪ helper methods -

obj1.m1()
obj1.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-axes. 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.

Thinking: Templates vs. Instances

Jim gainW(z);
c.o. expected to be modified for its att values

Templates

Person

Common

Attribute Definitions
(Types)

double weight
double height

Common

Behaviour Definitions
(Headers/API)

double getWeight() ^{accessor}
~~void~~ gainWeight(double d) ^{mutator}

Instance-Specific

Attribute Values

Jim. getBMI();
Jon. getBMI();

Instance-Specific

Behaviour Occurrence

Jim. gainW(z);
Jon. gainW(z);

Exercise -

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

e.g. int i; ^{set of int.} i = 23;
 i = 46.7; ~~i~~

e.g. Person p;
 Person p;

p = ?;

① p = new Person(...);

② p = p2;

③ p = p3;

↓
Instantiated
from a subclass
of
Person.

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

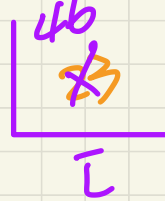
primitive
type

int

i

at runtime,

i can store any integer



reference
type

Person

p

at runtime,

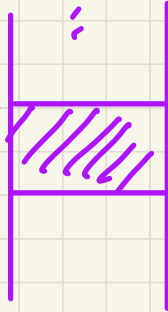
p can store the ref./address
of some Person object.

(1)

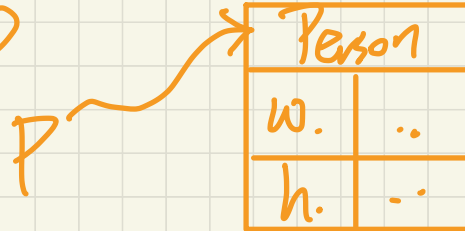
0x123

p

0x123



(2)



Null Pointer Exception

NPE if $\bar{jim} == null$

$\bar{jim}$.getBMI()

- method call

obj.m(...);

- context object is null

(2) Co.m1() null?

Co.m1().m2().m3() → NPE

(1) null?
?

(3) Co.m1().m2()
null?

Slides 16 ~ 47

assigned readings!

Parameters vs. Arguments

```
class Point {  
    Point(double x, double y) {...}  
  
    double getDistanceFrom(Point other) {...}  
  
    void move(char direction, double units) {...}  
}
```

Method Usages

```
class PointTester {  
    static void main(String[] args) {  
        Point p1 = new Point(2.5, -3.6);  
        Point p2 = new Point(-4.8, 5.9);  
        double dist1 = p1.getDistanceFrom(p2);  
        double dist2 = p2.getDistanceFrom(p1);  
        p1.move('R', 7.6);  
    }  
}
```

Template Definition

arguments

param.

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; class-level  
    int j;  
    void m1 (int p1) {  
        int m; local var. m = i; m = p1; int m2 = m;  
    }  
    void m2 (int p2) {  
        int n; i = n; n = p1; n = m;  
    }  
}
```

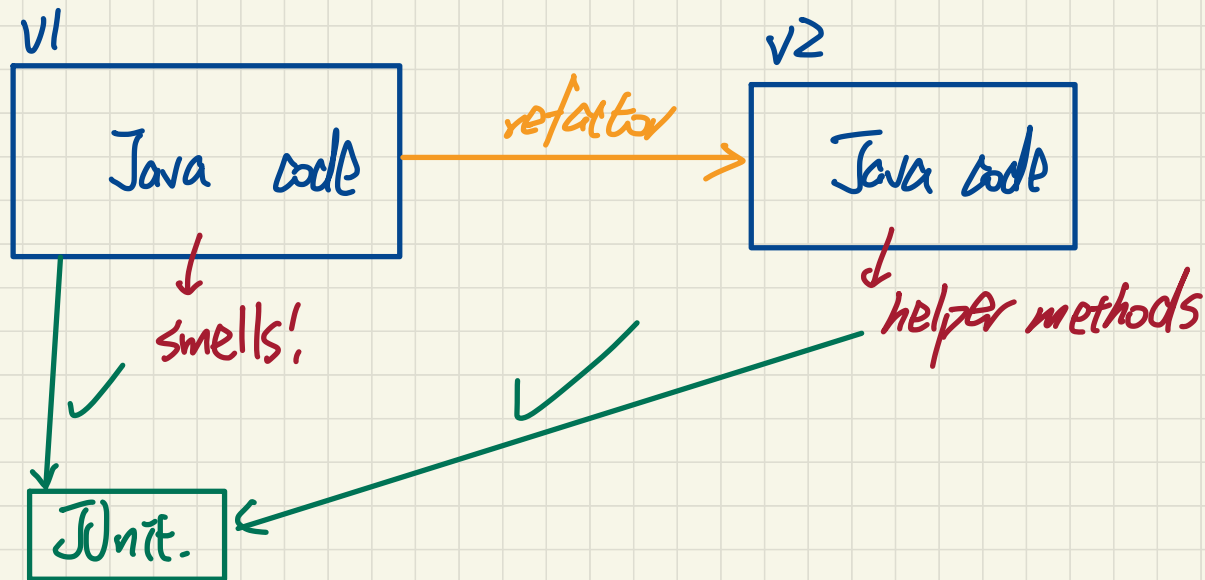
method-level

In-Lab Demo - Sep. 11

Helper Methods

Code Smell
Refactoring
Breakpoints & Debugger

$$\underline{x} \cdot y + \underline{x} \cdot z \\ \downarrow \\ x \cdot (y + z)$$



PersonCollector: without a Helper Method

```
public class PersonCollector {  
    private Person[] ps;  
    private final int MAX = 100; /* max # of persons to store */  
    private int nop; /* number of persons */  
    public PersonCollector() {  
        this.ps = new Person[MAX];  
    }  
    public void addPerson(Person p) {  
        this.ps[this.nop] = p;  
        this.nop++;  
    }  
}
```

```
    public boolean personExists(String n) {  
        boolean found = false;  
        for(int i = 0; i < nop; i++) {  
            if(ps[i].getName().equals(n)) { found = true; } }  
        return found;  
    }  
    public void changeWeightOf(String n, double w) {  
        for(int i = 0; i < nop; i++) {  
            if(ps[i].getName().equals(n)) { ps[i].setWeight(w); } }  
    }  
    public void changeHeightOf(String n, double h) {  
        for(int i = 0; i < nop; i++) {  
            if(ps[i].getName().equals(n)) { ps[i].setHeight(h); } }  
    }  
}
```

PersonCollector: with a Helper Method

```
public class PersonCollector {  
    private Person[] ps;  
    private final int MAX = 100; /* max # of persons to store */  
    private int nop; /* number of persons */  
    public PersonCollector() {  
        this.ps = new Person[MAX];  
    }  
    public void addPerson(Person p) {  
        this.ps[this.nop] = p;  
        this.nop++;  
    }  
}
```

```
    private int indexOf(String n) { /* Helper Methods */  
        int i = -1;  
        for(int j = 0; j < nop; j++) {  
            if(ps[j].getName().equals(n)) { i = j; }  
        }  
        return i; /* -1 if not found; >= 0 if found. */  
    }  
    public boolean personExists(String n) {  
        return this.indexOf(n) >= 0; }  
    public void changeWeightOf(String n, double w) {  
        int i = indexOf(n); if(i >= 0) { ps[i].setWeight(w); }  
    }  
    public void changeHeightOf(String n, double h) {  
        int i = indexOf(n); if(i >= 0) { ps[i].setHeight(h); }  
    }  
}
```

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

```
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;  
    }  
}
```

model

@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);  
}
```

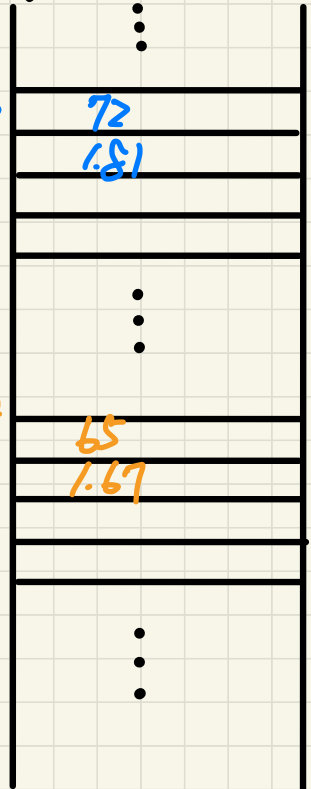
JUnit

!jim.equals(jonathan)

```
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);  
}
```

console

memory
(sequence of bytes)



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

Constructors not using this Keyword

```
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) {  
        this.weight = newWeight; weight  
        this.height = newHeight; height  
    }  
}
```

model

Question

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

(variable) shadowing

weight

height

this.weight

this.height

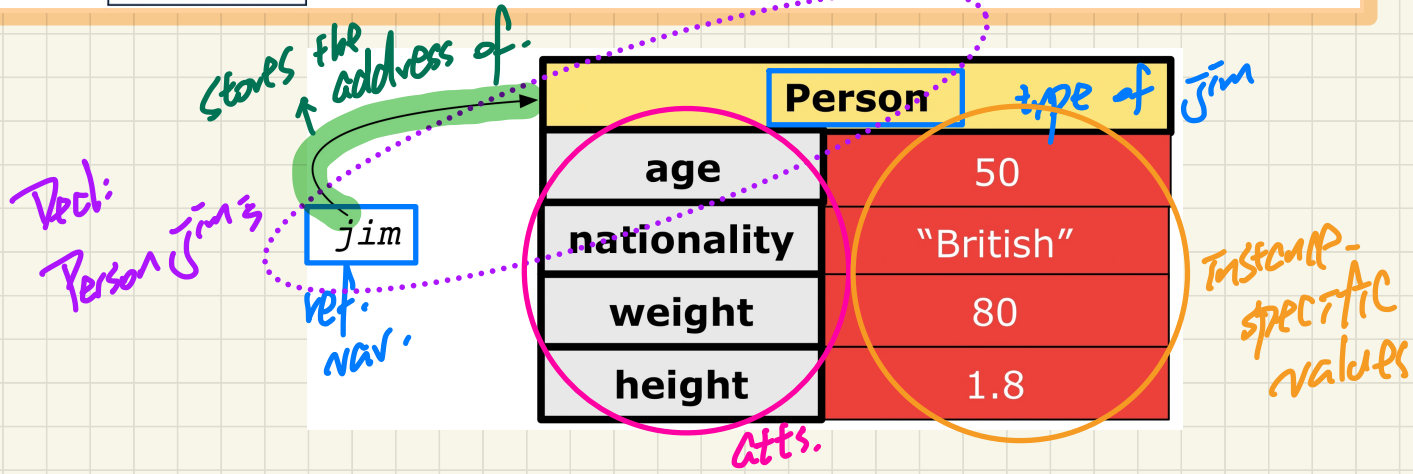
model

```
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) {  
        this.weight = newWeight;  
        height = newHeight;  
    }  
}
```

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

```
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;  
    }  
}
```

model

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

object

being
created

@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);  
}
```

JUnit

Person jim = new Person(72, 1.81);

①

③

②

① declare a ref. var.

jim

②

Person	
w.	72
h.	1.81

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

Accessors/Getters vs. Mutators/Setters

```
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;
    }
}
```

Person	
w.	72
h.	1.81

Person	
w.	65
h.	1.67

```
@Test
public void test3() {
    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);
}
```

assertEquals(23.46, m(), decimal (ε) 0.01);

Lecture 4 - Sep. 17

Review of OOP

Reference Aliasing & Primitive Arrays

Announcements/Reminders

- LabOP2 due this Friday at 12 **noon**!
- Lab1 to be released after LabOP2 is due.
- Priority: LabOP2 (tutorial videos + PDFs)
- This Thursday's office hour will be re-scheduled.

*Product[] ps;
int nops;*

Use of Accessors vs. Mutators

Slide 48

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

return

Automatic
garbage
collection (automated
mem.
mgmt.)
no return
value

- 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 leaving out the return value* [blue box] ✓
 - e.g., `System.out.println(jim.getBMI());` ✓
 - e.g., `double w = jim.getBMI();` ✓
- is not useful*

1. An accessor that's also a mutator ^{modifies some attr.}

^{non-void $\Rightarrow$ accessor}
double get BMI() {
 ^{modification to attr}
 this.weight += 23;
 ^{return bmi;}
}

[2.] A mutator that's an accessor is not possible.
? why.

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

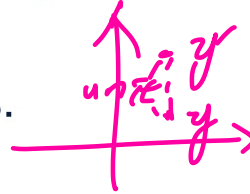
e.g., `Person(double w, double h)` vs.

`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)`

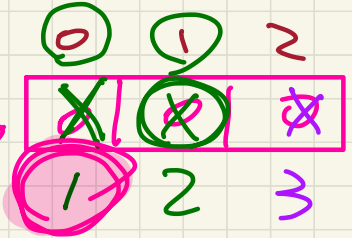
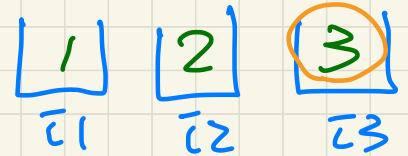
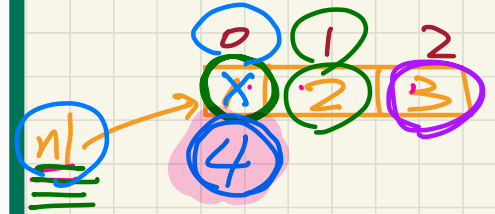
fun. getDist()

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]);
    
```

alt. $n2 = \{n1[0], n1[1], n1[2]\};$



* How many iterations?

It # 1 ($i=0$) $n2[0] = n1[0];$

It # 2 ($i=1$) $n2[1] = n1[1];$

It # 3 ($i=2$) $n2[2] = n1[2];$

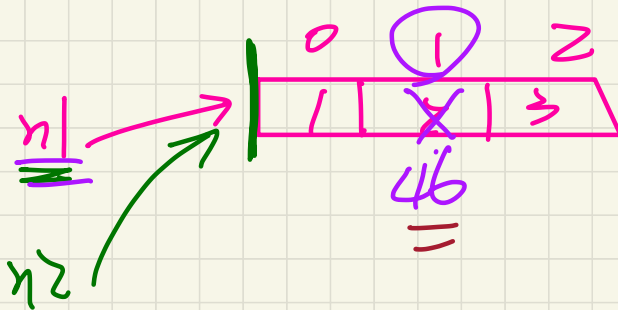
```

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]);

```

Copying
for starting

address of
numbers1
to numbers2.



$n1 == n2$ (T)

$n1[1] = 46$;
 $n2[1] == 2$ (F)

Copying Primitive vs. Reference Values

```
int i = 3;
int j = i; System.out.println(i == j); [T]
int k = 3; System.out.println(k == i && k == j); [T]
```

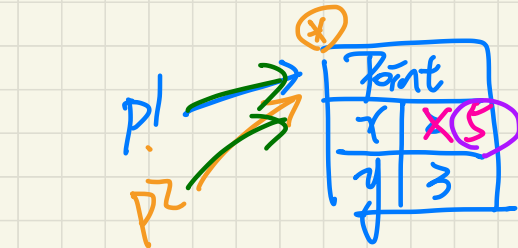


Primitive

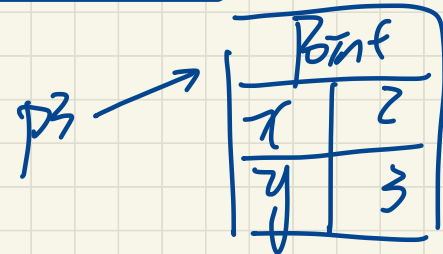
② let p2 point to where p1 points to
* ① copy the address stored in p1 into p2

```
Point p1 = new Point(2, 3);
Point p2 = p1; System.out.println(p1 == p2); [T]
Point p3 = new Point(2, 3);
System.out.println(p3 == p1 || p3 == p2); [F]
System.out.println(p3.x == p1.x && p3.y == p1.y); [F]
System.out.println(p3.x == p2.x && p3.y == p2.y); [F]
```

Reference



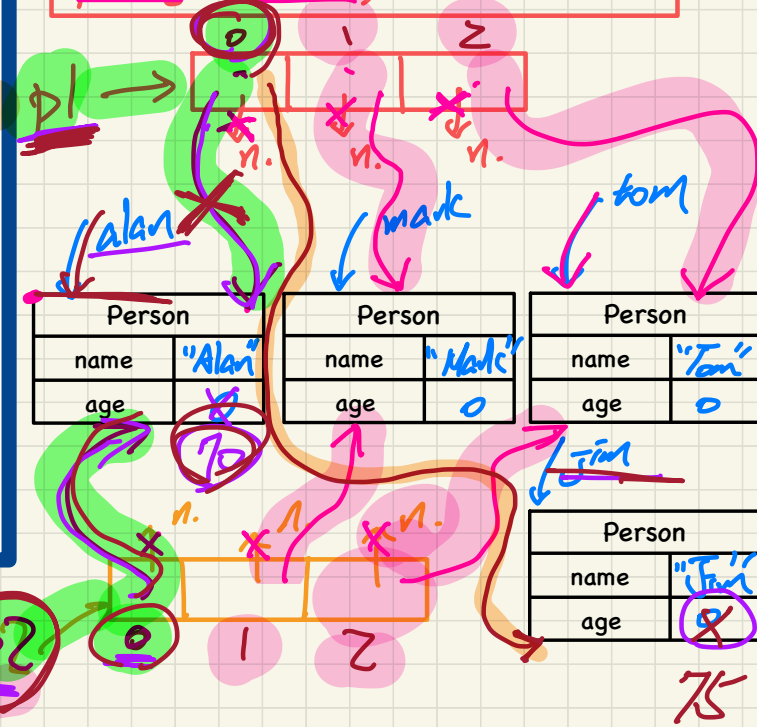
** $p1.setX(5);$
 $p2.getX() = 5$



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()); 70
    System.out.println(persons2[0].getAge()); 70
    * persons1[0] = jim;
    * persons1[0].setAge(75);
    System.out.println(jim.getAge()); 75 70 70
    System.out.println(alan.getAge());
    System.out.println(persons2[0].getAge());
```

* `Person[] p1 = new Person[3];`
`p1[0] = alan;` ① copy add. stored in alan to index 0 of p1.
`p1[1] = mark;`
`p1[2] = tom;`

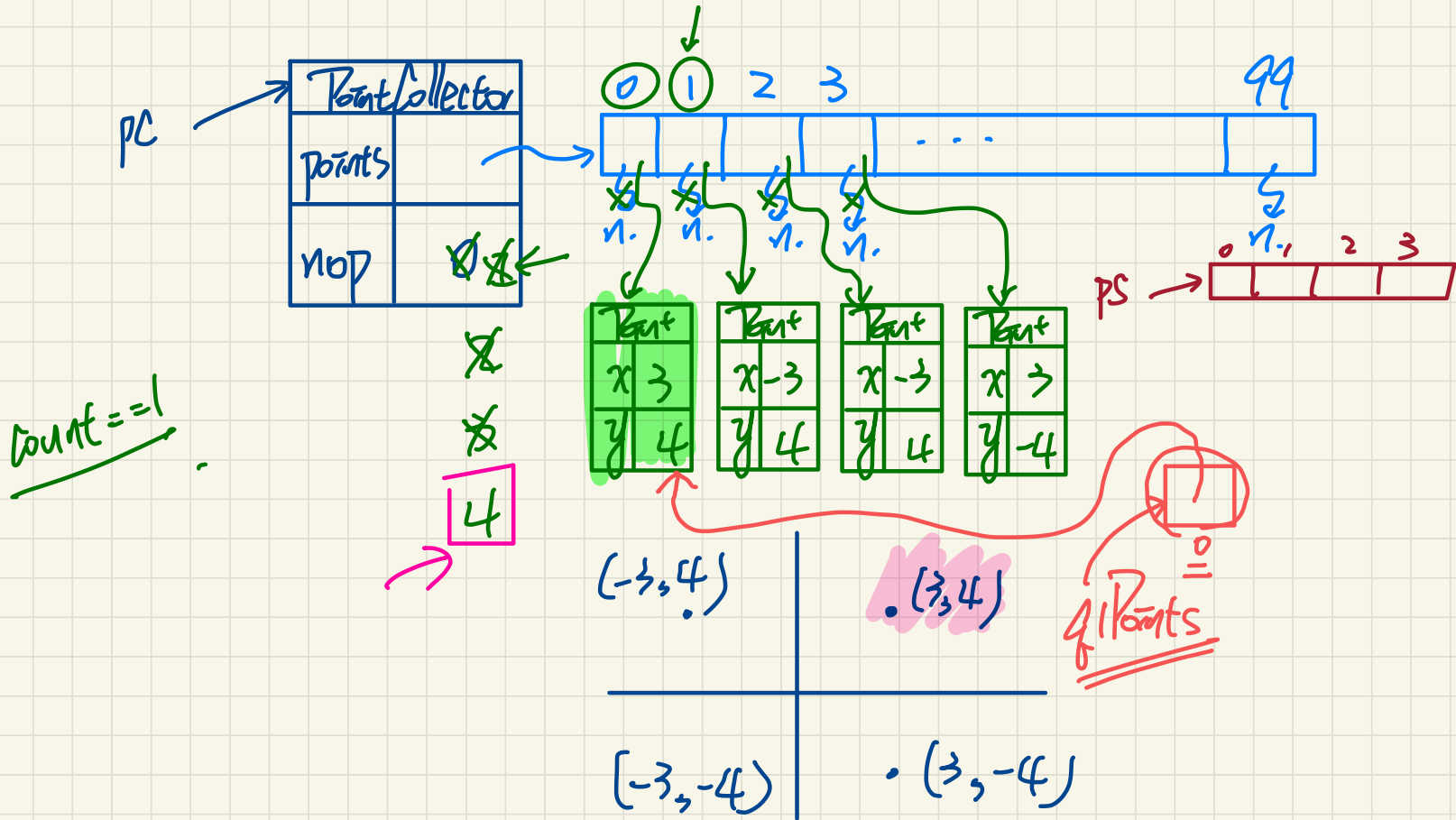


** It #1 (i=0): `p2[0] = p1[0];`
It #2 (i=1) `p2[1] = p1[1];`
It #3 (i=2) `p2[2] = p1[2];`

In-Lab Demo - Sep. 18

Programming Pattern

Reference-Typed Arrays
Integer Counter
Breakpoints & Debugger



Programming Pattern

```
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

```
@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);
}
```


Lecture 5 - Sep. 19

Review of OOP

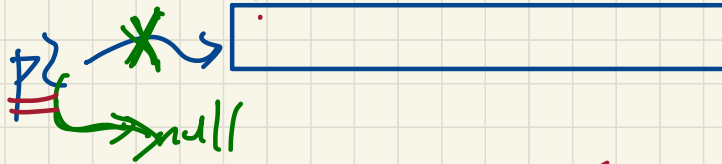
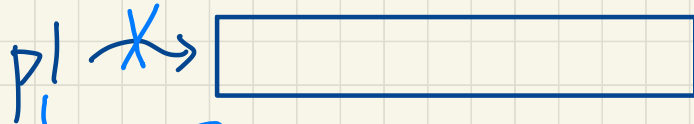
Ref-Typed Parameters vs. Return Values
Anonymous Objects
Reference to this
Static Variables

Announcements/Reminders

- LabOP2 due tomorrow (Friday) at 12 noon!
- Lab1 to be released after LabOP2 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)

Person[] p1 = new _____

→ Person[] p2 = new _____



p1 = p2 ;

p2 = null ;

- ① `p1 == null` (F)
- ② `p2 == null` (T)
- ③ `p1 == p2` (F)

Arrays and Aliasing

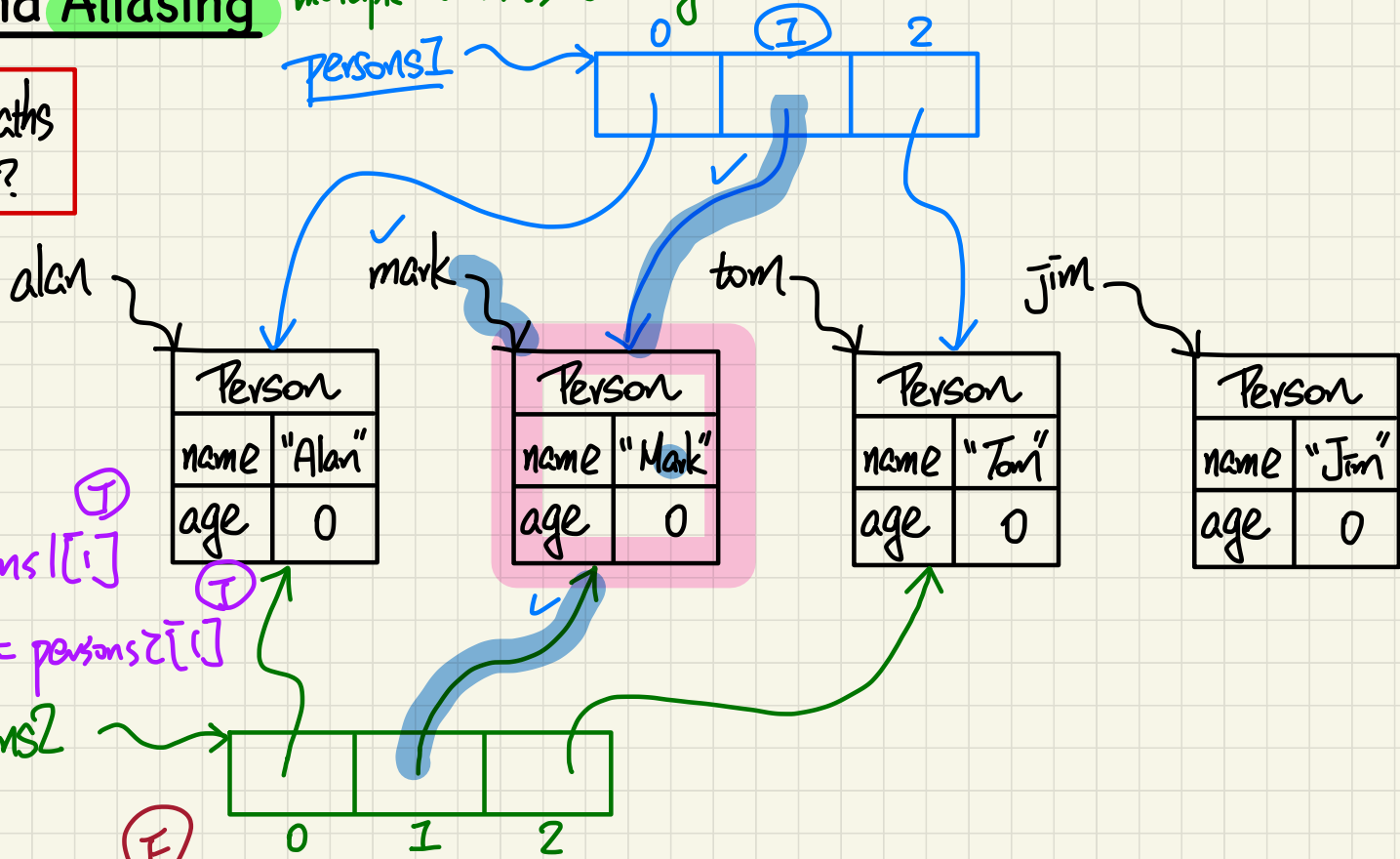
multiple variables storing the same address

All alias paths
to "Mark" ?

mark
persons[1]
persons2[1]

mark == persons[1]
persons[1] == persons2[1]

persons2
persons[1] == persons2[2]



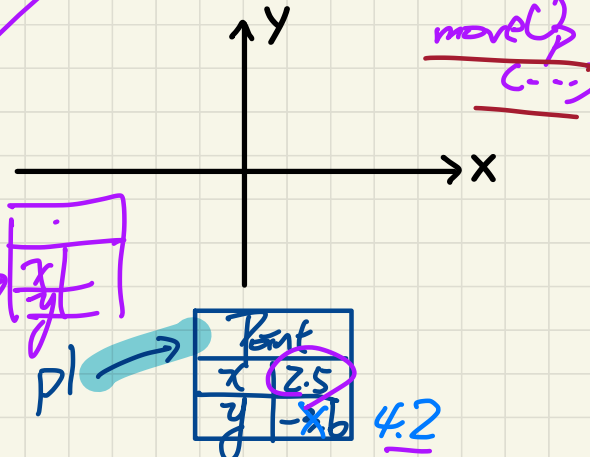
Reference-Typed Return Values

Slide 53

```
public class Point {  
    /* A mutator modifying the context Point object */  
    public void moveUp(int x) {  
        this.y = this.y + x;  
    }  
  
    /* An accessor returning a new Point object */  
    public Point movedUpBy(int x) {  
        Point np = new Point(this.x, this.y);  
        np.moveUp(x);  
        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);  
    }  
}
```

X return (new Point(this.x, this.y))



helper method call.

returns the address of some Point object

Anonymous Objects

Slide 56 - 58

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

Anonymous
exp.

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

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

Anonymous
obj

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

```
class Member {  
    private Order[] orders;  
    private int noo;  
    /* constructor omitted */  
    public void addOrder(Order o) {  
        this.orders[this.noo] = o;  
        this.noo++;  
    }
```

overloaded
method

```
public void addOrder(String n, double p, double q) {
```

11 Order o = new Order(n, p, q);

```
this.orders[this.noo] = o;  
this.noo++;
```

12. this.addOrder(o);

13. this.addOrder(____);

Exercise

Current
class to
look for
method
called

$$\neg p \vee \neg q \equiv \neg(p \wedge q)$$

Example: Reference to **this**

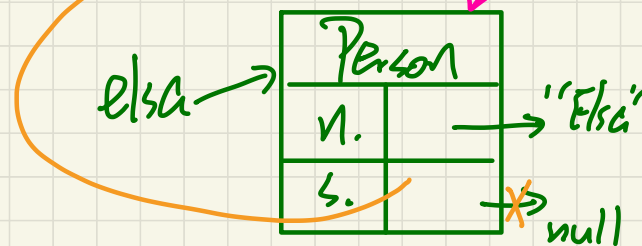
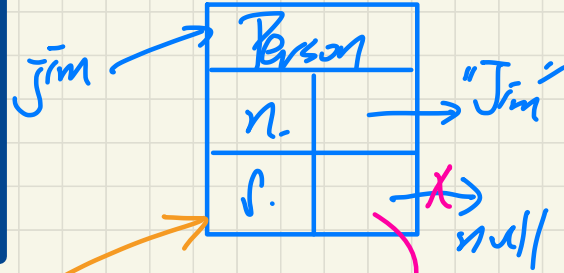
Slide 59 - 61

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

* this.spouse != null "this" is married
 || ~~&&~~ → not enough
 other.spouse != null "other" is married

alt.
 !(this.spouse == null
 &&
 other.spouse
 == null)
 robustness

- ① jim.spouse = elsa;
- ② elsa.spouse = jim;



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

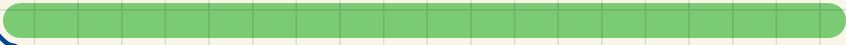
this. (points to jim)
 other (points to elsa)

Exercise

void

marry (Person other) {

?

if () {

this.spouse = other;

other.spouse = this;

}

else { error }

}

Lecture 6 - Sep. 24

Review of OOP, Exceptions

***Static Variables, Common Errors
Caller vs. Callee***

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: Manual

Slide 75

```
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;  
    }  
}
```

explicit parameter, presumably not duplicated

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

1. burden on the user of Account to specify unique id.
2. not a compilation error if ids are duplicated.

Declaring Global Variables among Objects

if not int. in the same tmp, it will be int. to default value (not Const.).

* each instance/object of Counter has its own copy (instance-specific value)

** all instances share the same copy (global).

```
public class Counter {  
    private int l; // non-static var.  
    static int g = 0; // int. (attributes)  
    // static var (global).  
    public Counter() {  
        this.l = 0;  
    } // only int. non-static var. in Const.
```

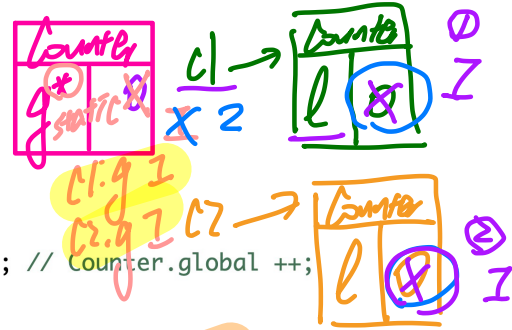
```
    public int getLocal() {  
        return this.l;  
    }
```

```
    public void incrementLocal() {  
        this.l ++;  
    } // c1 c2
```

```
    public void incrementGlobal() {  
        g ++; // warning  
    } // no need of t.o.  
}
```

```
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);  
    }
```

- ① c1.incrementLocal();
 - ② c2.incrementLocal();
 - ③ c1.incrementGlobal();
 - ④ c2.incrementGlobal();
- ```
Counter.g = Counter.g + 1; // Counter.global ++;
```



Counter.g ++;  
Counter.ClassName.varName g

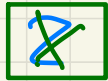
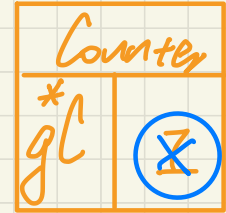
c1.g 2  
c2.g 2

# Managing Account IDs: Automatic

Slides 76 - 77

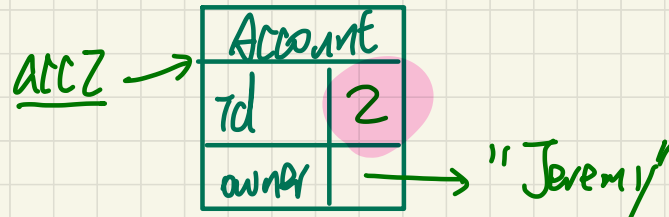
```
class Account {
 private static int globalCounter = 1;
 private int id; String owner;
 public Account(String owner) {
 *this.id = globalCounter;
 globalCounter ++;
 this.owner = owner; } }
//
```

parameter list  
not including  
id. burden of  
user of Account  
any more!



3

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

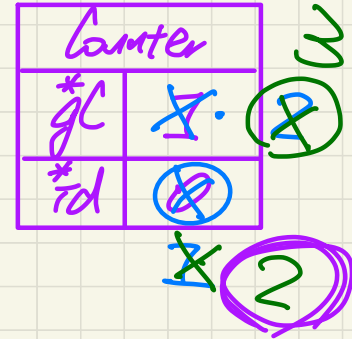


```

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

```

Handwritten annotations: A purple checkmark is above the `globalCounter` line. A purple arrow points to `static`. A blue arrow points to `id`. A green arrow points to `this.id`. A green arrow points to `globalCounter ++`. A green arrow points to `this.owner`.

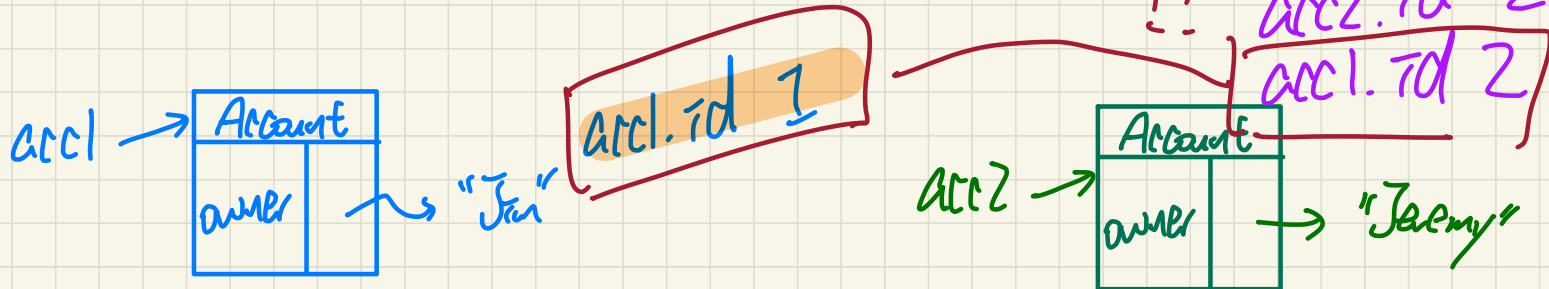


```

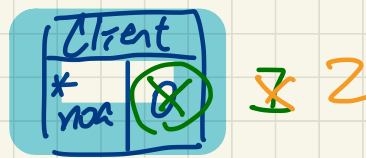
class AccountTester {
 Account acc1 = new Account("Jim");
 Account acc2 = new Account("Jeremy");
 System.out.println(acc1.getID() != acc2.getID()); }

```

Handwritten annotations: A blue arrow points to `Account` in the first line. A green arrow points to `new` in the first line. A blue arrow points to `Account` in the second line. A green arrow points to `new` in the second line. A blue arrow points to `System.out.println`.



# Misuse of Static Variables



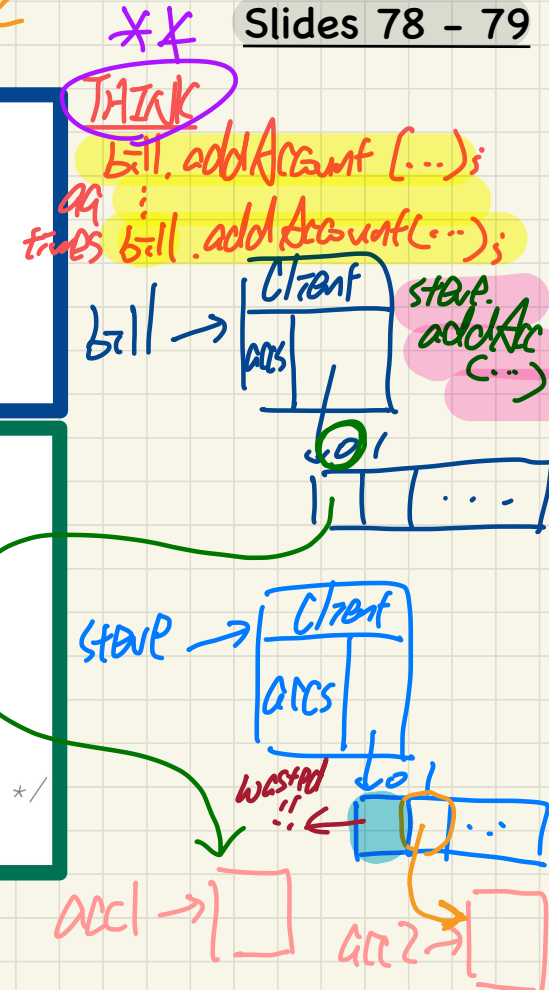
Slides 78 - 79

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

*Handwritten notes:*  
- `accounts` is boxed in orange.  
- `static` is circled in green.  
- `addAccount` is circled in green.  
- `acc` is circled in green.  
- `this.numberOfAccounts` is underlined in green.  
- `bill` is written in orange below the closing brace.  
- `acc1` and `acc2` are written in orange above the assignment line.  
- `bill` is written in orange above the increment line.

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

*Handwritten notes:*  
- `bill` and `steve` are underlined in blue.  
- `acc1` and `acc2` are underlined in red.  
- `bill.addAccount(acc1);` is highlighted in green.  
- `steve.addAccount(acc2);` is highlighted in orange.



# Use of Static Variables: Common Error

Bank c-ibc = new Bank();  
c-ibc.branchName

Slides 80 - 82

```
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 }
```

↓ should be replaced

by some obj.

class name

Expected Usage: Bank.getInfo() [no context obj.]

↳ Bank.branchName X ∵ 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 }
```

S =  
String "23" +  
"46"  
[S] "2346"

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 }
```

not ideal  
∴ each Bank obj. should have diff. branchName.

## Caller vs. Callee

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

```
class C1 {
 void m1() {
 C2 o = new C2();
 o.m2(); /* static type of o is C2 */
 }
}
```

*calling context (caller)*

*C2.m2 being called (callee)*

Q: Can a method be a **caller** and a **callee** simultaneously?

## Lecture 7 - Sep. 26

### Review of OOP, Exceptions

***Tracing Method Call Chain via Call Stack  
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 vs. Callee

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

```
class C1 {
 void m1() {
 C2 o = new C2();
 o.m2(); /* static type of o is C2 */
 }
}
```

calling context (caller) →

C2

C2.m2 - being called (callee)

(2) Example: Make C2.m2 a callee.

Q: Can a method be a **caller** and a **callee** simultaneously?

(1) Make C1.m1 a callee

(1.a) class C1 {  
 void m2() {  
 C1 o = new C1();  
 o.m1();  
 }  
}

caller

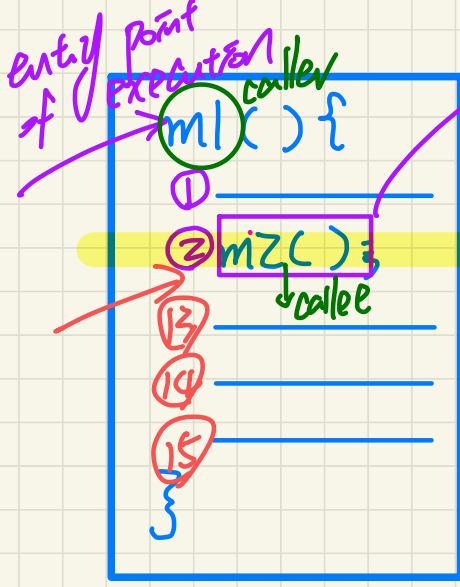
callee

(1.b) class C2 {  
 void m4() {  
 C1 o = new C1();  
 o.m1();  
 }  
}

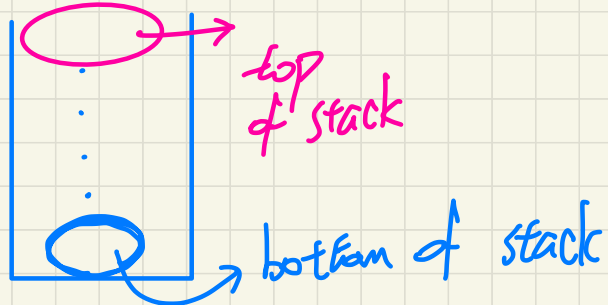
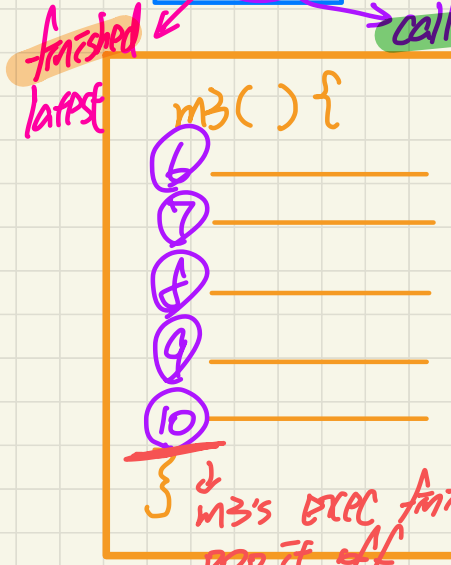
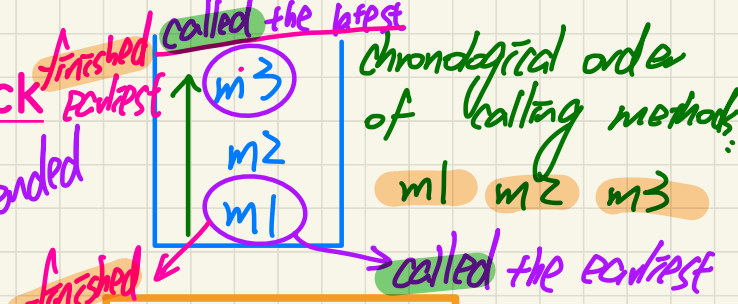
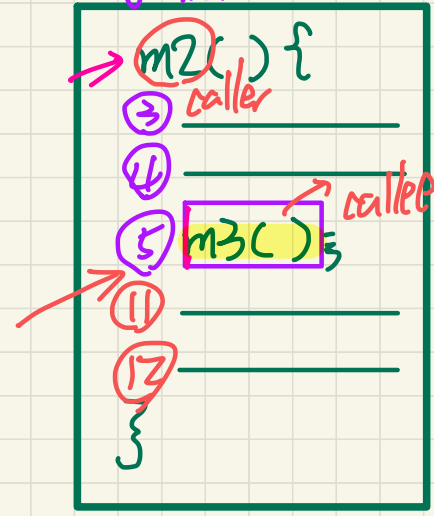
caller

callee

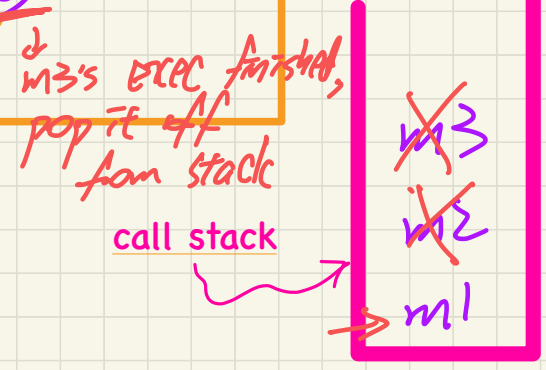
# Visualizing a Call Chain using a Stack



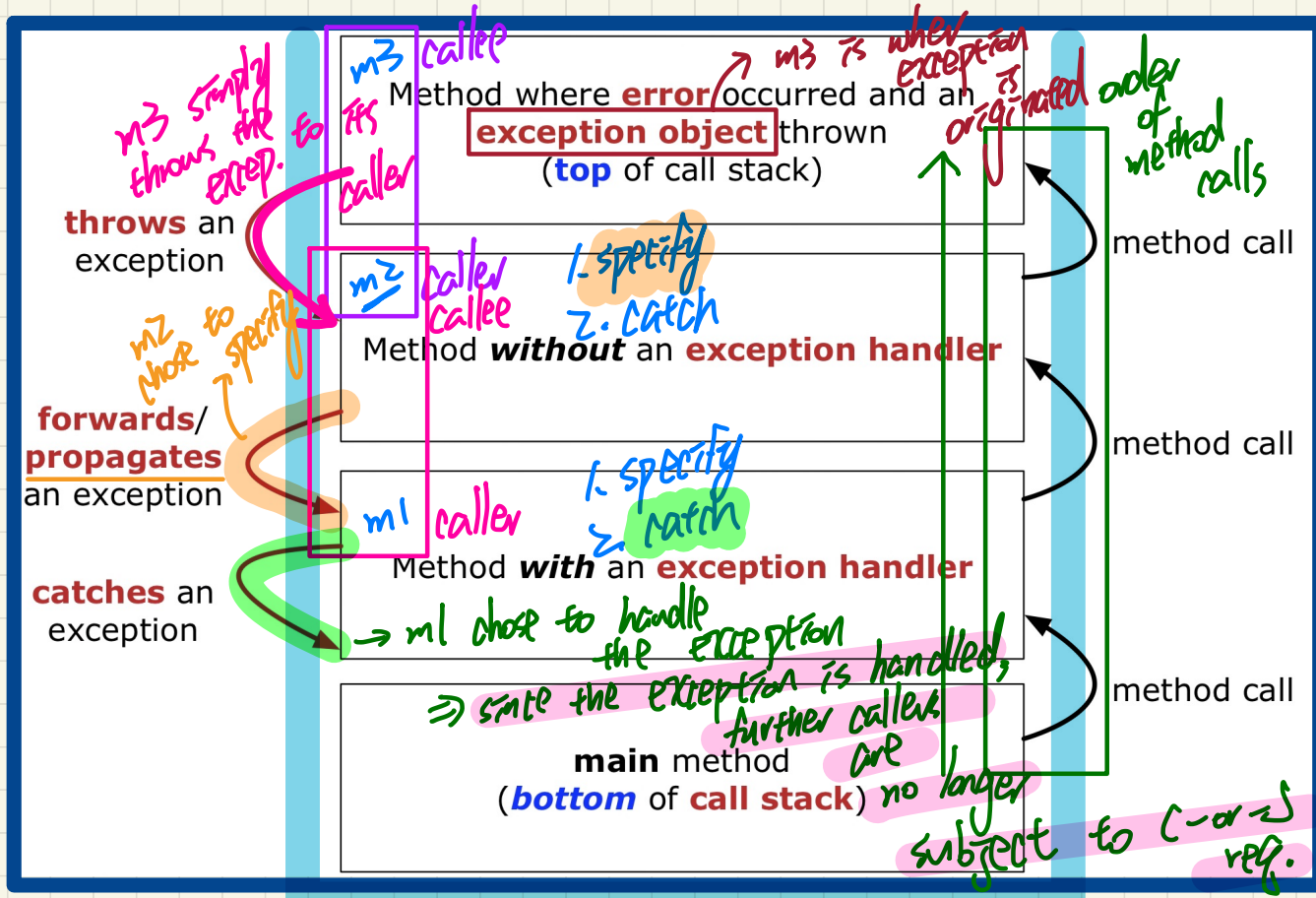
while m2 is being executed, m1's exec. is suspended



add/push  
remove/pop



# What to Do When an Exception is Thrown: Call Stack



Exceptions: When occurred, Normal Flow of Exec. Disrupted

Normal

```
class C1 {
 void m1() {
 ① _____
 ② _____
 ③ C2 o = new C2();
 ④ o.m2();
 ⑤ _____
 ⑥ _____
 }
}
```

→ no error  
→ no exception  
thrown back  
to the caller

Abnormal/Exception

```
class C1 {
 void m1() {
 ① _____
 ② _____
 ③ C2 o = new C2();
 ④ o.m2();
 X _____
 X _____
 }
}
```

→ some error  
→ exception  
thrown here

↓  
bypassed  
∴ exec flow  
disrupted!



# 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 */
 }
}
```

```
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? */
 }
}
```

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

## 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.

call  
stack

A.ma

B.mb

Tester.main

## Lecture 8 - Oct. 1

### Exceptions

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

## 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, outside try-catch block */
try {
 ① o.m(...); /* may throw SomeException */
 ② ... /* rest of try-block */
}
catch (SomeException se) {
 X ... /* rest of catch-block */
}
③ ... /* after, outside try-catch block */
```

Handwritten annotations for Normal Flow:

- Green arrow from ① to the text "excep. does not occur".
- Green arrow from the catch block to the text "by passed".

When the exception does not occur

## Abnormal Flow of Execution

```
... /* before, outside try-catch block */
try {
 ① o.m(...); /* may throw SomeException */
 X ... /* rest of try-block */
}
catch (SomeException se) {
 ② ... /* rest of catch-block */
}
④ ... /* after, outside try-catch block */
```

Handwritten annotations for Abnormal Flow:

- Pink arrow from ① to the text "occurs".
- Pink arrow from the try block to the text "bypassed".

When the exception occurs

# Catch-or-Specify Requirement: Call Stack

Q1. Origin of Exception:

$C1.m1$

Q2. Methods subject to CoS Req:

$C1.m1 \sim C2.m2$

Q3. Methods free from CoS Req:

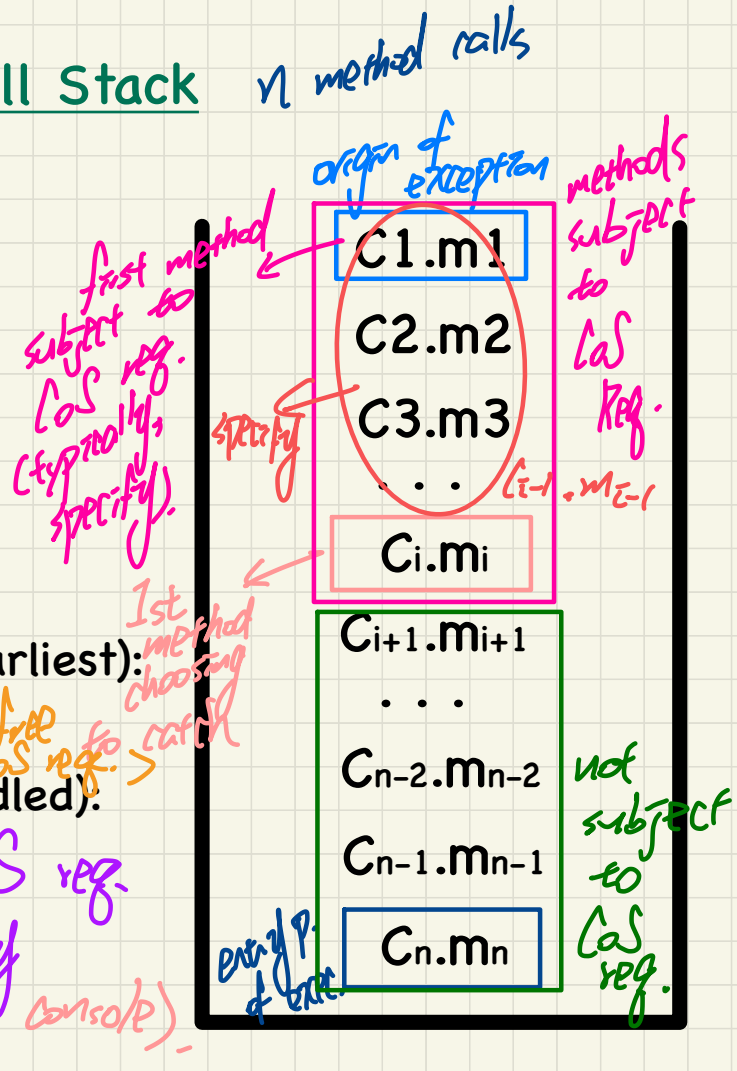
$C_{i+1}.m_{i+1}$

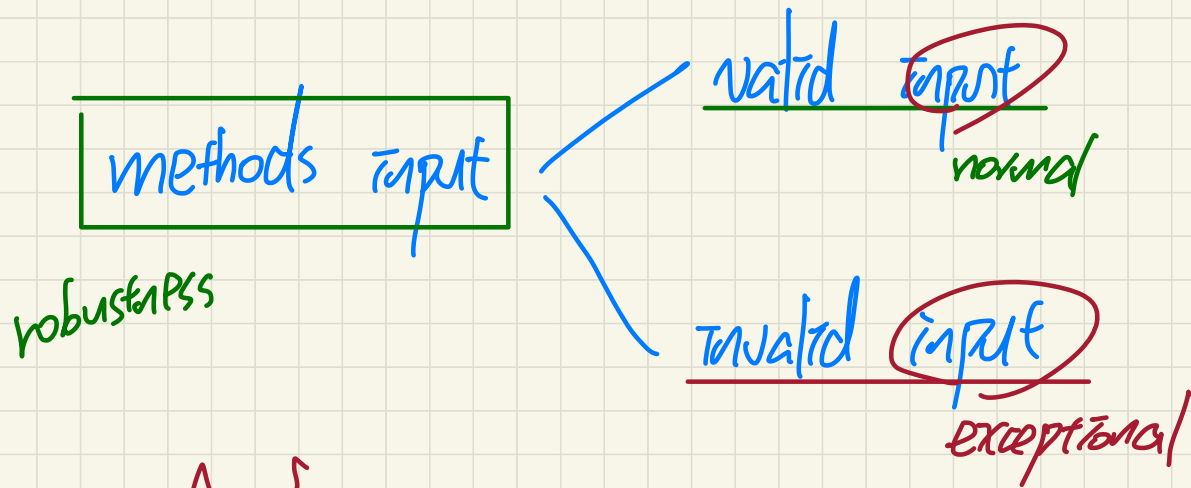
Q4. Extreme Case 1 (exception handled earliest):

handled in  $C2.m2$  ( $C3.m3 \sim Cn.mn$  free from CoS req.)

Q5. Extreme Case 2 (exception never handled):

$C1.m1 \sim Cn.mn$  subject to CoS req.  
but all chose to specify  
(excep. thrown to console)





$\{$   
 $m(\dots)$   
 $0.m(\dots)$   
 $\}$

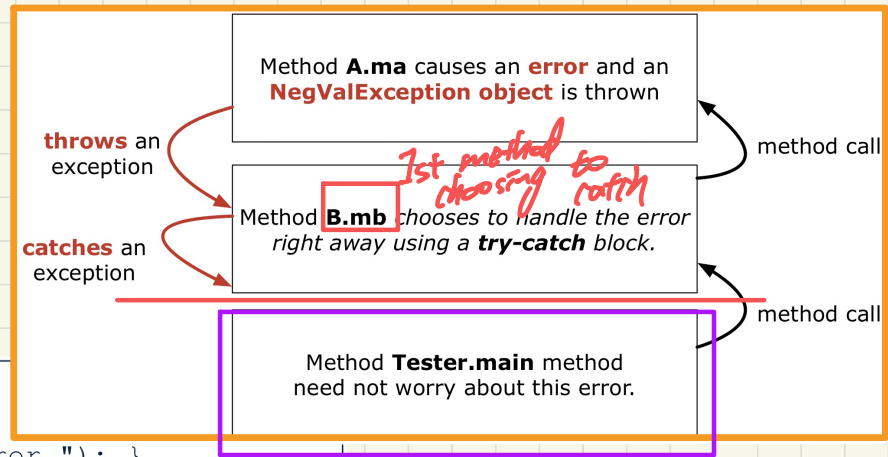
## Version 1:

### 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. */
 } }
}
```



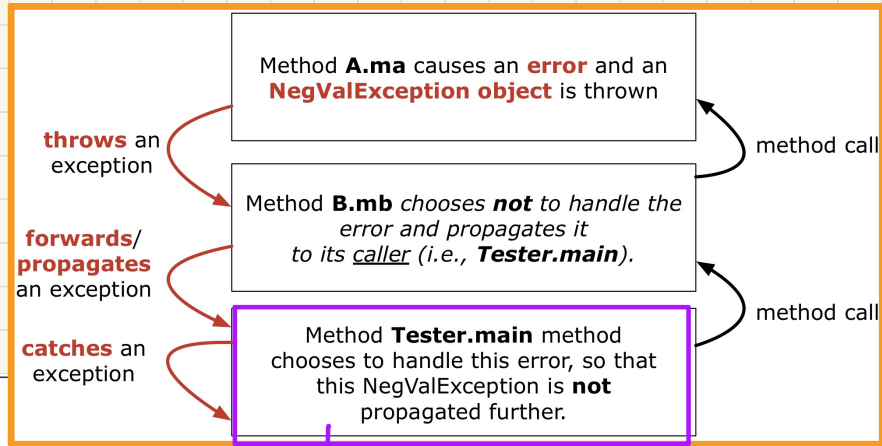
## Version 2:

## 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. */ }
 } }
}
```



↓  
since its caller  
has not handled the excep,  
Tester.main still  
subject to COS Reg.



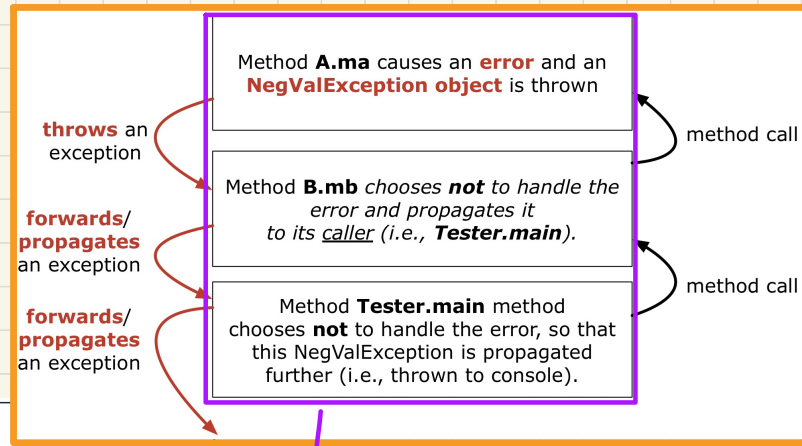
## Version 3:

## Handle in Neither Classes on Call Stack

```
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);
 } }
}
```



all methods  
subject to last Reg.,  
and all opt to specify.

# 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 of this method subject to C/S Req

④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ ⑯ ⑰ ⑱ ⑲ ⑳ ㉑ ㉒ ㉓ ㉔ ㉕ ㉖ ㉗ ㉘ ㉙ ㉚ ㉛ ㉜ ㉝ ㉞ ㉟ ㊱ ㊲ ㊳ ㊴ ㊵ ㊶ ㊷ ㊸ ㊹ ㊺ ㊻ ㊼ ㊽ ㊾ ㊿

Caller?  
Callee?

call stack

C.setR.  
C.C. main

```
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);
 }
 }
}
```

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ ⑯ ⑰ ⑱ ⑲ ⑳ ㉑ ㉒ ㉓ ㉔ ㉕ ㉖ ㉗ ㉘ ㉙ ㉚ ㉛ ㉜ ㉝ ㉞ ㉟ ㊱ ㊲ ㊳ ㊴ ㊵ ㊶ ㊷ ㊸ ㊹ ㊺ ㊻ ㊼ ㊽ ㊾ ㊿

# 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

```
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) {
 1 Scanner input = new Scanner(System.in);
 2 boolean inputRadiusIsValid = false;
 3 while (!inputRadiusIsValid) {
 4 System.out.println("Enter a radius:");
 5 double r = input.nextDouble();
 6 Circle c = new Circle();
 7 try { c.setRadius(r); }
 8 inputRadiusIsValid = true;
 9 System.out.print("Circle with radius " + r);
 10 System.out.println(" has area: " + c.getArea()); }
 11 catch (InvalidRadiusException e) { print("Try again!"); }
 } } }
```

# Error Handling via Exceptions: Banks

Test Case:

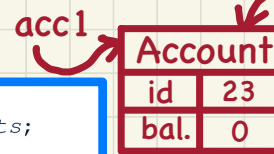
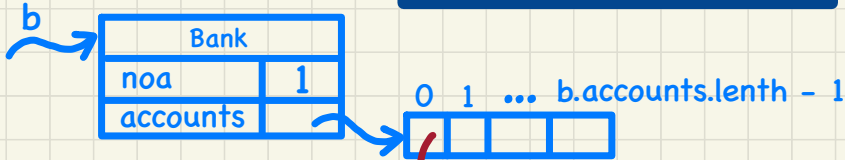
User enters **-5000000**

```
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 (1.5M || balance - a < 0) {
 2 throw new InvalidTransactionException("Invalid withdraw.");
 }
 else { balance -= a; }
 }
}
```

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

```
class BankApplication {
 public static void main(String[] args) {
 1 Bank b = new Bank();
 2 Account acc1 = new Account(23);
 3 b.addAccount(acc1);
 4 Scanner input = new Scanner(System.in);
 5 double a = input.nextDouble();
 6 try {
 7 b.withdraw(23, a);
 X System.out.println(acc1.balance);
 }
 11 catch (InvalidTransactionException e) {
 12 System.out.println(e);
 }
 }
}
```



Account

`accounts[i].withdraw(a)`

`Account[]`

call stack

A-W  
B-W  
BA.main

## More Example: Multiple Catch Blocks

Assume: customized exceptions only

→ order of their catch blocks does not matter.

```
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, x); // may throw ITE
 Circle c = new Circle();
 x c.setRadius(r); // NRE may be thrown
 System.out.println(r.getArea());
}
X catch (NegativeRadiusException e) {
 System.out.println(r + " is not a valid radius value.");
 e.printStackTrace();
}
⑧ catch (InvalidTransactionException e) {
⑨ System.out.println(x + " is not a valid transaction value.");
⑩ e.printStackTrace(); a
}
```

### Test Case 1:

a: -5000000

r: 23

### Test Case 2:

a: 100

r: -5

## **Lecture 9 - Oct. 3**

### **TDD with JUnit**

***Parsing Integers***

***Error Handling: Console vs. Exceptions***

***Deriving Test Cases***

***JUnit Test Method vs. Method Under Test***

***Regression Testing***

***JUnit Test: An Exception Not Expected***

## Announcements/Reminders

- **Written Test 1** result to be released Fri or Mon
- **Lab1** due tomorrow (Friday) at noon
- **Lab2** to be released tomorrow

## More Example: Parsing Strings as Integers

```
1 Scanner input = new Scanner(System.in);
2 boolean validInteger = false;
3 while (!validInteger) {
4 System.out.println("Enter an integer:");
5 String userInput = input.nextLine();
6 try {
7 int userInteger = Integer.parseInt(userInput);
8 validInteger = true;
9 } catch (NumberFormatException e) {
10 System.out.println(userInput + " is not a valid integer.");
11 /* validInteger remains false */
12 }
13 }
```

Handwritten annotations on the code:

- Line 1: Scanner
- Line 2: boolean validInteger = false;
- Line 3: while (!validInteger) {
- Line 4: System.out.println("Enter an integer:");
- Line 5: String userInput = input.nextLine();
- Line 6: try {
- Line 7: int userInteger = Integer.parseInt(userInput);
- Line 8: validInteger = true;
- Line 9: }
- Line 10: } catch (NumberFormatException e) {
- Line 11: System.out.println(userInput + " is not a valid integer.");
- Line 12: /\* validInteger remains false \*/
- Line 13: }

Additional annotations:

- "twenty-three" written above line 5.
- "23" written below line 5.
- "twenty-three" written above line 7.
- "NFE" written next to line 7.
- "23" written below line 7.

Test Case: ✓

User Enters: twenty-three

User Then Enters: 23 ✓



# 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 }
```

Caller?  
Callee?

① No more CoS Reg. enforced on the callers  
② caller of this method (error originated) is no longer obliged to handle the error.

call stack

```
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 }
```

C. SR  
CC.m

--- given that the callee already signaled an error, caller still proceeds to calculate

# 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

| context | caller | callee |
|---------|--------|--------|
|         |        |        |
|         |        |        |
|         |        |        |

```
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 {
 public 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.");
 }
}
```

## 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 exception*

**Example 2:** Method that calls another which throws the exception

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

## 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) { ... }
 }
}
```

**\*\* min a test , and a VICE/VISE happened → pass**  
no VICE/VISE → fail

## Coming Up with Test Cases: A Single, Bounded Variable

**\* min a test , and an exception happened → fail**  
no exception → pass

**\* no exception expected**

Boundries:

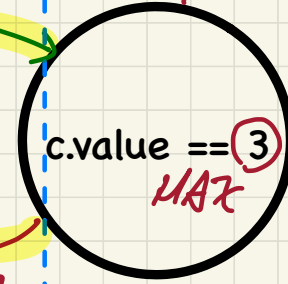
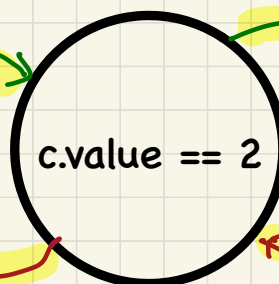
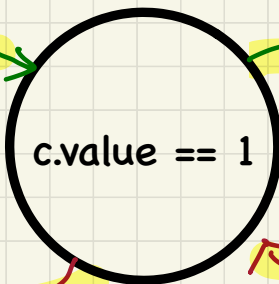
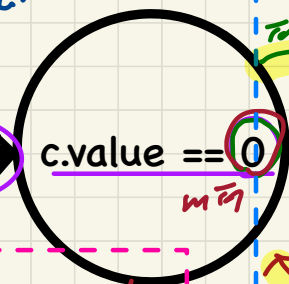
Counter.MIN\_VALUE <= c.value <= Counter.MAX\_VALUE

**\*\* exception expected**  
②

TMC

**State diagram** → a combination of the variables of interest

init counter obj created.



→ c.increment()

→ c.decrement()

# 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() {
```

```
 MyClass o = new MyClass();
```

```
 assertEquals(23, o.getValue());
```

```
}
```

✓ test method

return value:  
actual value

method under test

expected  
value

expected  
output

JUnit test method  
(test)

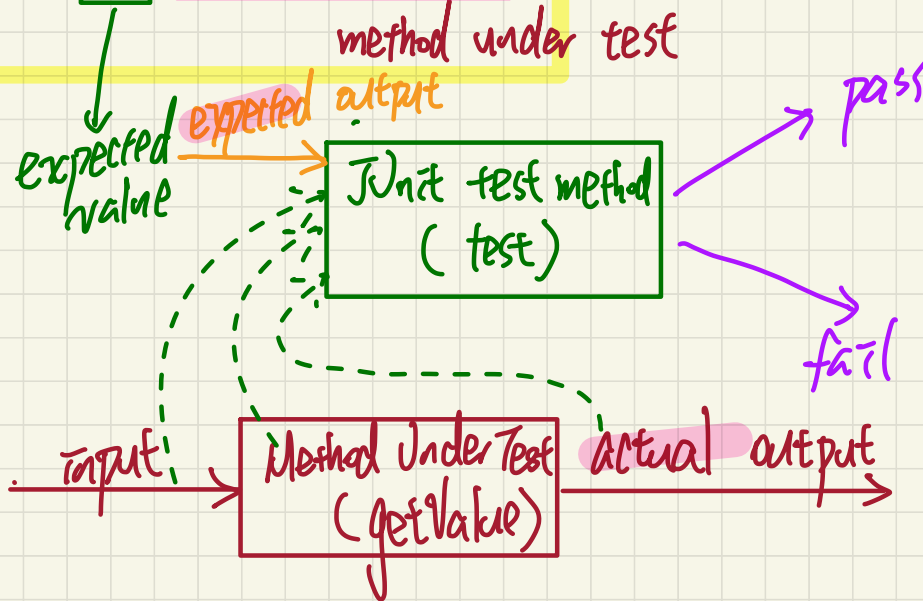
pass

fail

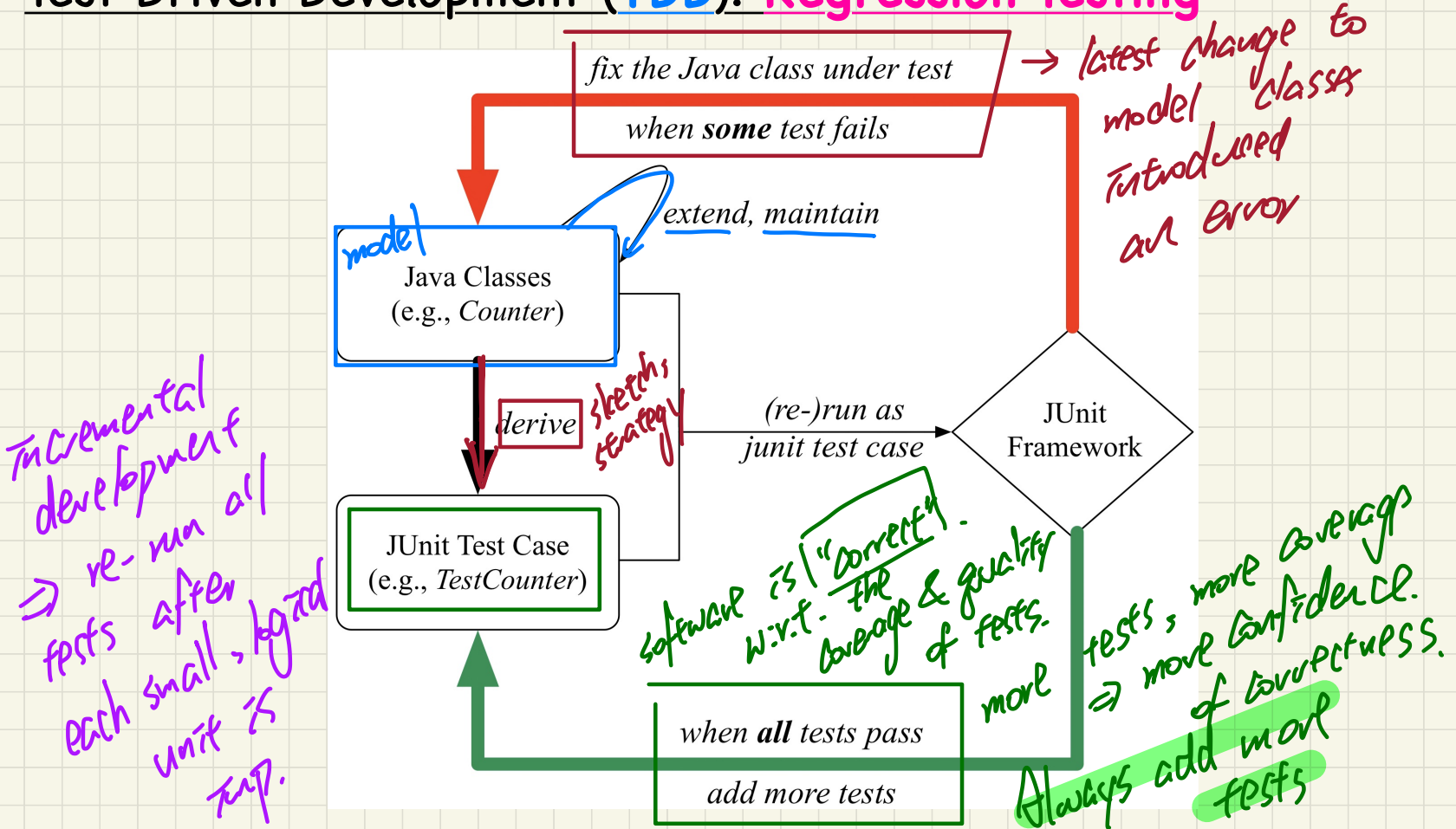
input

Method Under Test  
(getValue)

actual  
output



# Test-Driven Development (TDD): Regression Testing





# 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 } catch (ValueTooLargeException e) {
9 /* Exception is not expected to be thrown. */
10 fail("ValueTooLargeException is not expected.");
11 }
12 }
13 }
```

*Handwritten notes:*  
① not throw VTLE  
② VTLE X  
③ may throw VTLE  
④ fail

What if increment is implemented correctly?

- ① not throw VTLE  
② increment

## 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 }
```

*Handwritten notes:*  
① did not inc.  
② throw VTLE unexpectedly

What if increment is implemented incorrectly?

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

# Running JUnit Test 1 on Correct Implementation

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

Handwritten annotations: 0, 0 → 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000

```
1 @Test
2 public void testIncAfterCreation() {
3 1 Counter c = new Counter();
4 2 assertEquals(Counter.MIN_VALUE, c.getValue());
5 3 try { c.v == 0 ✓
6 4 c.increment(); I
7 5 assertEquals(1, c.getValue());
8 }
9 X catch (ValueTooLargeException e) {
10 /* Exception is not expected to be thrown. */
11 X fail ("ValueTooLargeException is not expected.");
12 }
13 6
```

# Running JUnit Test 1 on Incorrect Implementation

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

```
1 @Test
2 public void testIncAfterCreation() {
3 1 Counter c = new Counter();
4 2 assertEquals(Counter.MIN_VALUE, c.getValue());
5 3 try { c.v == 0
6 4 c.increment();
7 X assertEquals(1, c.getValue());
8 }
9 7 catch (ValueTooLargeException e) {
10 /* Exception is not expected to be thrown. */
11 8 fail("ValueTooLargeException is not expected.");
12 }
13 }
```

↳ reject the unexpected VTLCE by failing the test

## Lecture 10 - Oct. 8

### TDD with JUnit, Object Equality.

***JUnit Test: Exception Expected vs. Not  
Using Loops in JUnit Test Methods***

## Announcements/Reminders

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

## A Default Test Case that **Fails**

The result of running a test is considered:

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

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);
```

$$\text{expected} - \epsilon \leq \text{actual} \leq \text{expected} + \epsilon$$



public void increment() throws VTL --.

```
1 @Test
2 public void testIncAfterCreation() {
3 Counter c = new Counter();
4 assertEquals(Counter.MIN_VALUE, c.getValue());
5 try { c.v == 0;
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 }
```

Annotations in the code:  
- Line 5: *C.v == 0* (handwritten)  
- Line 6: *c.increment()* is circled in blue. A red arrow points to it with the text "wrong: throw VTL unexpectedly".  
- Line 7: *assertEquals(1, c.getValue());* is underlined in purple.  
- Line 9: *catch (ValueTooLargeException e)* is boxed in red.  
- Line 11: *fail("ValueTooLargeException is not expected.");* is boxed in yellow.  
- A red arrow points from the red box on line 9 to the text "replace by VTLSE" on the right.  
- A purple arrow points from the yellow box on line 11 to the text "no assertion fail. no unexpected." at the bottom.  
- A red arrow points from the text "replace by VTLSE" to the text "Compilation error" below it.

replace by  
VTLSE  
↳ Compilation  
error

① No VTLSE occurred → expected  
→ test value is incremented

# JUnit: An Exception Expected

```
1 @Test
2 public void testDecFromMinValue() {
3 Counter c = new Counter();
4 assertEquals(Counter.MIN_VALUE, c.getValue());
5 try { c.v -= 0
6 ① c.decrement();
7 ② fail("ValueTooSmallException is expected.");
8 }
9 catch (ValueTooSmallException e) {
10 /* Exception is expected to be thrown. */
11 }
12 }
```

*Expect. VTSE should occur*

*pass*

*\* being able to move to next line means the expected VTSE did not occ.*

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 {
 ⑤ if (value == Counter.MIN_VALUE) {
 ⑥ throw new ValueTooSmallException("counter value is " + value);
 }
 X else { value --; }
}
```

```
1 @Test
2 public void testDecFromMinValue() {
3 ① Counter c = new Counter(); | c.v == 0
4 ② assertEquals(Counter.MIN_VALUE, c.getValue());
5 ③ try {
6 ④ c.decrement();
7 X fail("ValueTooSmallException is expected.");
8 }
9 ⑦ catch (ValueTooSmallException e) {
10 → /* Exception is expected to be thrown. */
11 }
12 }
```

NTSE occurred  
↳ expc flow disrupted  
↳ pass!

pass.

←  
—  
—  
—

# Running JUnit Test 2 on Incorrect Implementation



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

*Handwritten notes:*  
- Red arrow points to `if` condition.  
- Red arrow points to `else` block.  
- `0 → -1` written below the `value --;` line.

*\* exec flow normal: no exception occurred.*

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

*Handwritten notes:*  
- Red arrow points to `fail` method call.  
- Red arrow points to `catch` block.  
- Red arrow points to `c.decrement();` line.  
- Red arrow points to `try` block.  
- Red arrow points to `assertEquals` line.  
- Red arrow points to `Counter c = new Counter();` line.  
- Red arrow points to `@Test` line.  
- Red arrow points to `testDecFromMinValue()` line.  
- Red arrow points to `public void` line.  
- Red arrow points to `fail` method call.  
- Red arrow points to `catch` block.  
- Red arrow points to `c.decrement();` line.  
- Red arrow points to `try` block.  
- Red arrow points to `assertEquals` line.  
- Red arrow points to `Counter c = new Counter();` line.  
- Red arrow points to `@Test` line.  
- Red arrow points to `testDecFromMinValue()` line.  
- Red arrow points to `public void` line.

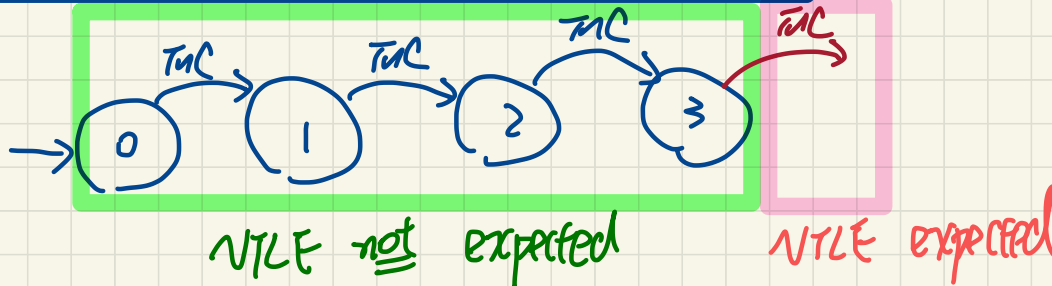
# JUnit: Exception Sometimes Expected, Sometimes **Not**

```
1 @Test
2 public void testIncFromMaxValue() { NtLE not expected
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(); NtLE expected
13 fail("ValueTooLargeException was NOT thrown as expected.");
14 }
15 catch (ValueTooLargeException e) {
16 /* Do nothing: ValueTooLargeException thrown as expected. */
17 }
18 }
```

## 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 }
14 catch (ValueTooLargeException e) {
15 // Do nothing: ValueTooLargeException thrown as expected. */
16 }
17 }
18

```

# Running JUnit Test 3 on Incorrect Implementation

```
public void increment() throws ValueTooLargeException {
 ④ if (value == Counter.MAX_VALUE) {
 ⑤ throw new ValueTooLargeException("counter value is " + value);
 }
 else { value++; }
}
```

```
1 @Test
2 public void testIncFromMaxValue() {
3 Counter c = new Counter();
4 ① try { V=0 VLT thrown unexpectedly
5 ② c.increment(); X c.increment(); c.increment();
6 }
7 ③ catch (ValueTooLargeException e) {
8 ④ 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 is 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 }
```

Handwritten annotations on the test code:

- Line 3:  $c.v = 0$  ✓
- Line 5:  $c.v == 2$  ✓
- Line 6:  $c.v == 3$  ✓
- Line 11:  $c.v == 3$  ✓
- Line 12: "no VTLException thrown as expected" (highlighted)
- Line 13: "fail" is crossed out with an X.
- Line 15: "catch" is crossed out with an X.



# 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 */
```

*scry VtLE occurred unexpectedly rather than fail();*

*not appropriate to continue the test after we knew it failed*

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 }
```

*Handwritten notes:*

- Lines 5-7: `c.increment();` are boxed in green. Next to them is the note "VTLF not expected".
- Line 9: `c.increment();` is boxed in red. Next to it is the note "VTLF expected".
- Line 12: The `catch` block is underlined in purple. A purple circle is drawn around the block, with an arrow pointing to it and the text "pass? fail?".
- Below the code, the note "VTLF occurred" is written in purple.

Hint: Say Line 12 is executed,

is it clear if that `ValueTooLargeException` was thrown as expected?

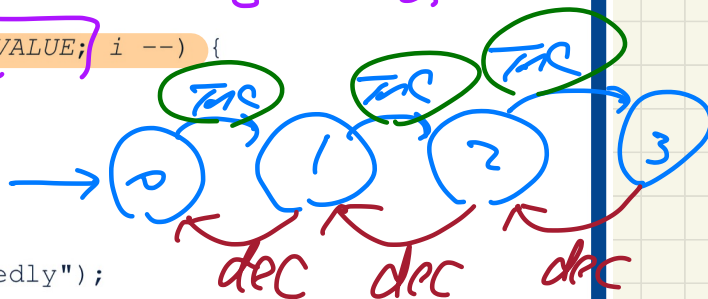
# Testing Many Values in a Single Test

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 }
```

0 1 2 ] 3 times

3 2 1 ] 3 times



## Lecture 11 - Oct. 10

### Object Equality.

***equals Method: Default vs. Overridden***  
***Overriding equals Method: Phases 1 - 2***  
***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)

```
class PointV1 {
```

```
 int x;
```

```
 int y;
```

```
 PointV1(int x, int y) { ... }
```

```
}
```

```
PointV1 p1 = new ...
PointV1 p2 = new ...
println(p1.equals(p2))
```

Q1. Does this compile,  
given that equals  
is not declared in PointV1?  
YES.

Q2. Where does equals  
come from?  
Object.

# The equals Method: To **Override** or **Not**?

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

parent class of PointV1/V2  
compare addresses

1. does not declare equals == method
  2. use the inherited version of equals from object class
- 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 }
 Point other = (PointV2) obj;
 return this.x == other.x
 && this.y == other.y;
 }
}
```

extends  
inheritance  
child class of Object

overriding/re-defining the inherited equals method

# The equals Method: Default Version $s \rightsquigarrow "(2, 3)"$

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

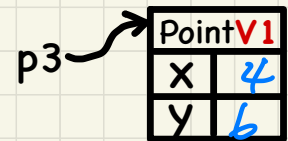
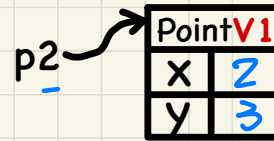
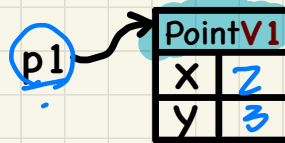
*Handwritten notes:*  $p1 \neq p1$ ,  $p1 \neq null$

```
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); // F
6 System.out.println(p2 == p3); // F
7 System.out.println(p1.equals(p1)); // T
8 System.out.println(p1.equals(null)); // F
9 System.out.println(p1.equals(s)); // F
10 System.out.println(p1.equals(p2)); // F
11 System.out.println(p2.equals(p3)); // F
```

*Handwritten notes:*  $p1 \neq p1$ ,  $p1 \neq null$

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

*Handwritten notes:*  $p1 \neq p1$ ,  $p1 \neq null$



$p1.equals(s)$

↳ equals from Object:  $p1 == s$  (F)

Note: writing  $p1 \neq s \rightarrow$  compilation errors

\*  $p1.equals(p1)$   
C.O.  
no explicit  
equals in PointV1  
extends  
→ use the  
default  
version  
in  
Object



# The equals Method: **Overridden** Version

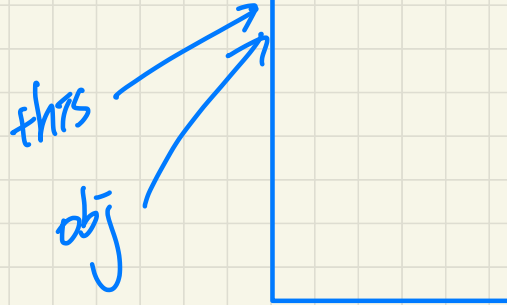
Phase 1

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

pl. equals(..);

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

| PointV2 |  |
|---------|--|
| x       |  |
| y       |  |

# The equals Method: Overridden Version

## Example 1: Trace L7

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

*PointV2 p4 = p1;*

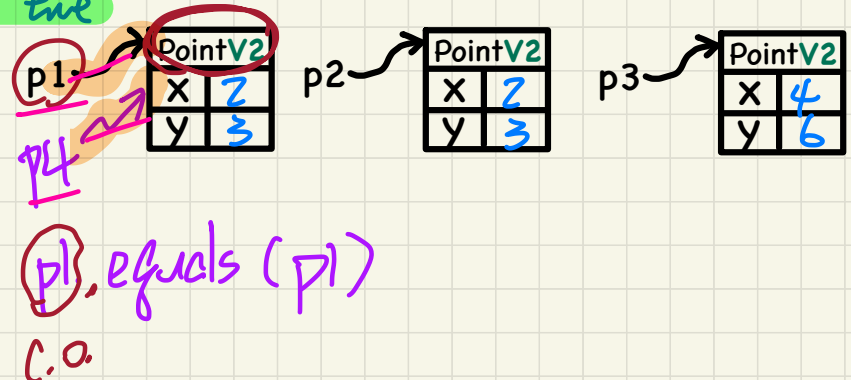
extends

*p1 equals p4*

*if (p1 == p4) { m. true }*  
*if (p1 == p1) return true*

```
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);
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

PointV2 p5 = null;  
p5.getX();  
NPE

Phase 2

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

alt

```
if (obj == null) {
 if (this == null) {
 return true; }
 else return false;
}
```

extends

sensible?

```
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;
 }
}
```

this → [ ]  
obj → null

| PointV2 |  |
|---------|--|
| x       |  |
| y       |  |

pl. equals(..)  
if  
if  
null

NPE  
reaching this  
line means  
this != obj  
this != null

# The equals Method: Overridden Version

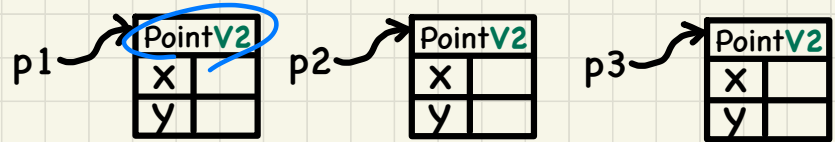
## Example 1: Trace L8

```
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); /* ████████ */
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)); /* ████████ */
```



obj → null

# Static Type, Dynamic Type, Type Casting

For now,  
always cast to  
the same D.T.

**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

if a  
C.O. is  
of certain  
ST, it determines  
the range  
of methods  
applicable  
on that  
C.O.

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

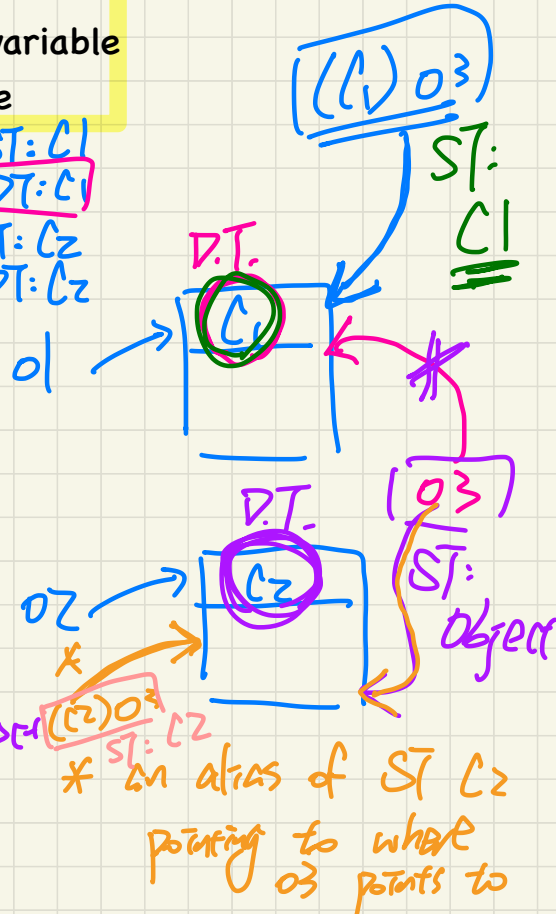
class C2 {
 ...
 public void m2() {...}
}
```

```
C1 o1 = new C1(); ST: C1 DT: C1
C2 o2 = new C2(); ST: C2 DT: C2
o1.m1(); ✓
o1.m2(); X 'ST C1 does not contain m2.
o2.m1(); X
o2.m2(); ✓

Object o3; ST: Object
o3 = o1; ST o3: Object DT o3: C1
o3.m1(); X
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 has  
no declaration  
of m1 and m2.

exp of  
static type  
C1



## Lecture 12 - Oct. 22

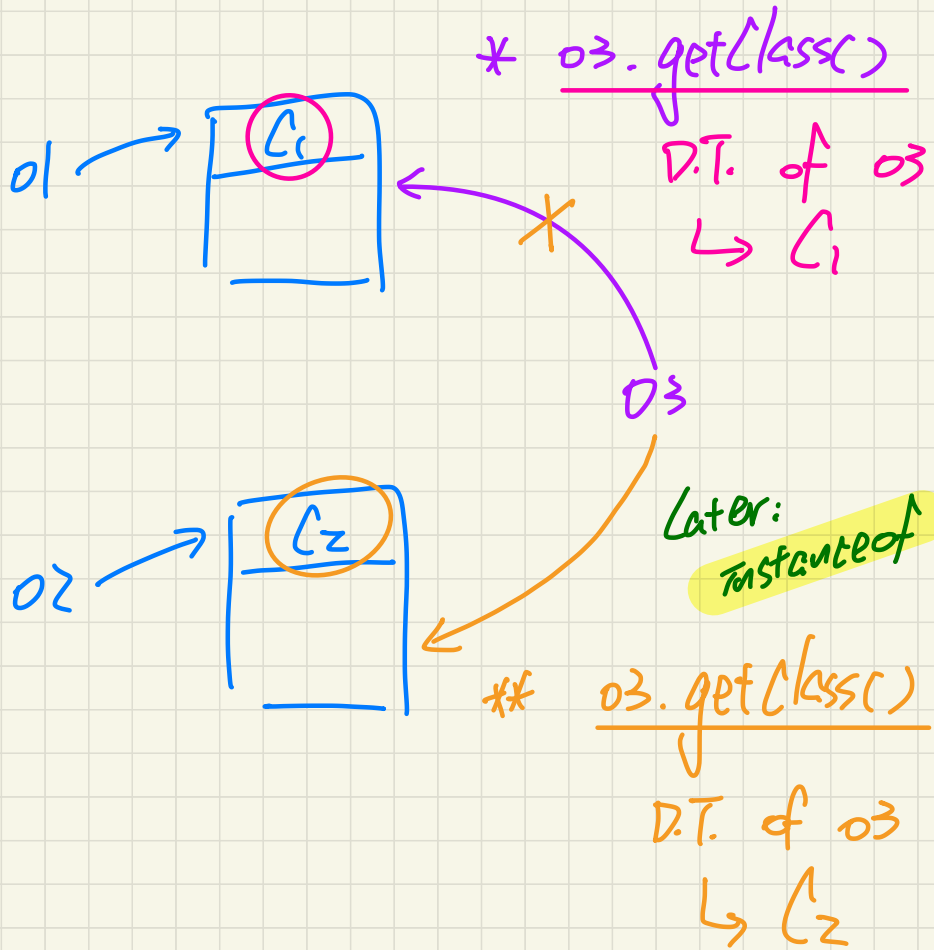
### Object Equality.

***Reference Equality vs. Object Equality***  
***JUnit: assertEquals vs. assertEquals***

## Announcements/Reminders

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

```
C1 o1 = new C1();
C2 o2 = new C2();
o1.m1();
o1.m2();
o2.m1();
o2.m2();
Object o3,
o3 = o1, *
o3.m1;
o3.m2;
((C1) o3).m1();
((C1) o3).m2();
o3 = o2; **
((C2) o3).m1();
((C2) o3).m2();
```





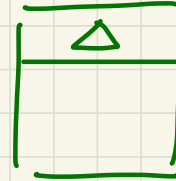
# The equals Method: Overridden Version

Phase 3

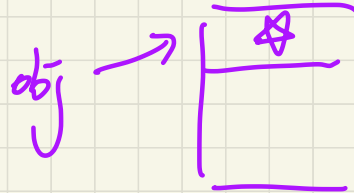
```
public class Object {
 ...
 public boolean equals(Object obj) {
 return this == obj;
 }
}
```

extends

this



```
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;
 }
}
```



if the two objects have distinct D.T.s, no comparison → false.

| PointV2 |  |
|---------|--|
| x       |  |
| y       |  |

# The equals Method: Overridden Version

## Example 1: Trace L9

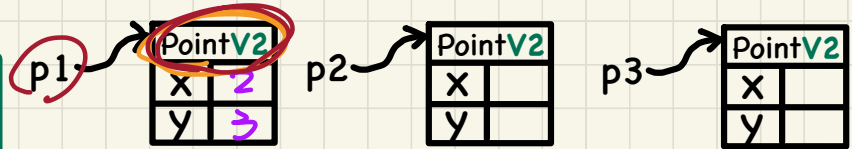
```
public class Object {
 ...
 public boolean equals(Object obj) {
 return this == obj;
 }
}
```

```
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); /* */
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)); /* */
```

PointV2 extends Object  
Pl.get class() s.get class() String

```
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;
 }
}
```

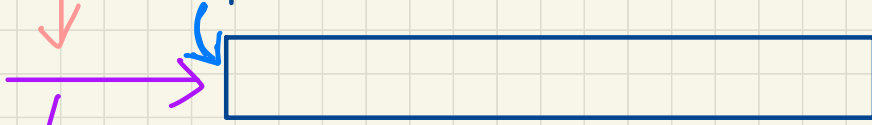
Exercise: if instances eval. to false  
PointV2 != String



p1 equals (s)  
D.T. of C.O.  
IS. PointV2  
→ invoke version of equals from PointV2

$\neg (b_1) \{ \text{return} \dots \}$

$\neg (b_2) \{ \text{return} \dots \}$



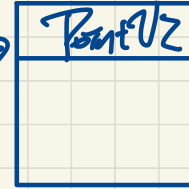
being able  
to reach this line:  
 $!b_1 \ \&\& \ !b_2$   
 $!(b_1 || b_2)$

# The equals Method: Overridden Version

Phase 4

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

\* return this.x == obj.x  
&& this.y == obj.y;



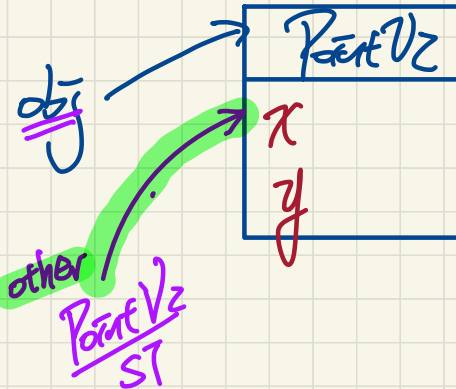
this

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;
 }
}
```

ST of obj: Object  
↳ can only call  
Object methods  
on obj.

\* alias of ST: PointV2



other  
PointV2  
ST

| PointV2 |  |
|---------|--|
| x       |  |
| y       |  |

\*\*  
(PointV2 obj).x  
||  
this.x  
&&

- ① this != obj
- ② obj != null
- ③ this and obj have same D.T.

\*\*

# The equals Method: Overridden Version

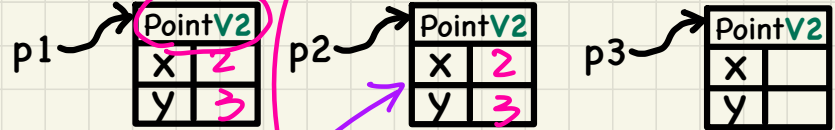
## Example 1: Trace L10

```
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)); /* */
```



D.T. of p1 : PointP2

other  
is: PointV2

# The equals Method: Overridden Version

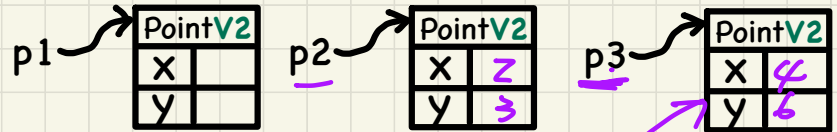
## Example 1: Trace L11

```
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)); /* */
```



other

# The equals Method:

## To Override or Not?

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

extends

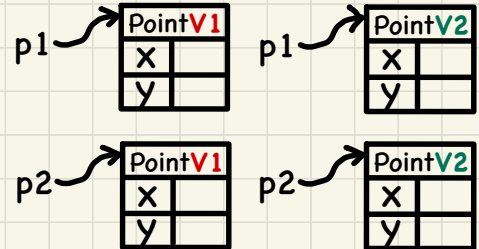
extends

```
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;
 }
}
```

```
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;
 }
}
```

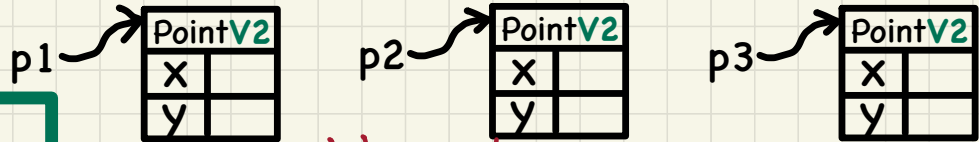
extends

$T \Rightarrow T == T$

```
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;
 }
}
```

$P \Rightarrow Q$   $P \Rightarrow Q$

```
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); /* F */
7 System.out.println(p1.equals(p2)); /* T */
8 System.out.println(p2 == p3); /* */
9 System.out.println(p2.equals(p3)); /* */
```



$obj1 == obj2$

(A) Two objects are **reference**-equal.

(B) Two objects are **contents**-equal.

$\hookrightarrow obj1.equals(obj2)$

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



ref. eq.

✓ (1)

$$\text{obj1} == \text{obj2}$$

$\Rightarrow$

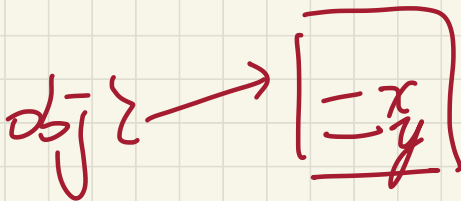
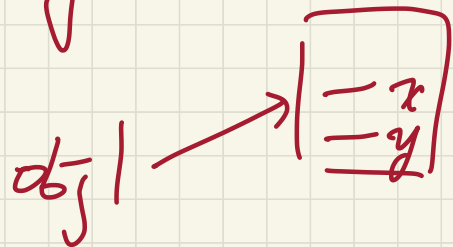
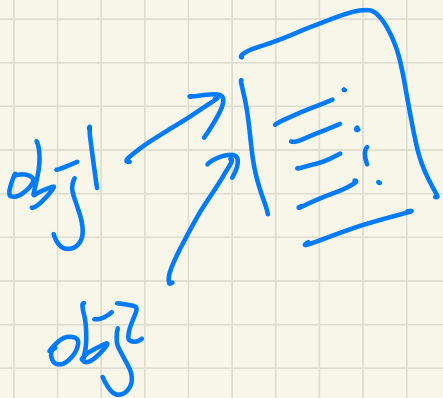
$$\text{obj1.equals(obj2)}$$

obj. eq.

□ (2)

$$\text{obj1.equals(obj2)} \overset{\times}{\Rightarrow} \text{obj1} == \text{obj2}$$

not in general



$\neg x.equals(null)$

$\hookrightarrow x.equals(null)$  should  
always evaluate to false  
if  $x$  cannot  
be null  
(otherwise NPE).

## assertSame vs. assertEquals

→ address.

**assertSame**(exp1, exp2)

→ exp1 == exp2

- Passes if exp1 and exp2 are references to the same object

≈ **assertTrue**(exp1 == exp2)

≈ **assertFalse**(exp1 != exp2) → pass



```
Point v1 p1 = new Point v1(3, 4);
```

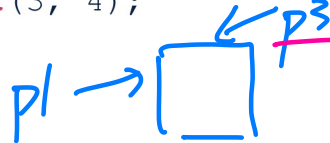
```
Point v1 p2 = new Point v1(3, 4);
```

```
Point v1 p3 = p1;
```

```
assertSame(p1, p3);
```

```
assertSame(p2, p3);
```

pass  
fail



exp1 != exp2 → false



**assertEquals**(exp1, exp2)

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

```
int i = 10;
```

```
int j = 20;
```

```
assertEquals(i, j);
```

assertFalse(i != j) → false.

→ assertTrue(i == j) false.

## assertEquals: Reference Comparison or Not

\* Writing

`assertEquals(exp1, exp2)`

- `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); → p1.eg.(p2) → p1==p2 → False fail
assertEquals(p2, p3); → p2.eg.(p3) → *p2==p3 → False fail
```

**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);
```

*fail. → invoke equals in?*

*p2 == p3  
directly causes  
compilation  
error*

*Thus, fail*

ref types

`assertEquals ( exp1 , exp2 )`

↳ `exp1.equals(exp2)`

① `assertEquals ( exp1 , exp2 )`

↳ `exp1.equals(exp2)`

invoke the equals  
method defined in  
exp1's D.T.

② `assertEquals ( exp2 , exp1 )`

↳ `exp2.equals(exp1)`

invoke the equals  
method defined in  
exp2's D.T.

## 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

# assertEquals: Reference Comparison or Not

\* Writing  
 $p2 == p3$   
directly causes  
compilation  
error.

**assertEquals**(exp1, exp2)

- $\approx$  exp1.equals(exp2) if exp1 and exp2 are **reference** type

**Case 1:** If equals is **not** explicitly overridden in exp1's declared type  
 $\approx$  **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);
```

**Case 2:** If equals is explicitly **overridden** in exp1's declared type  
 $\approx$  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);
```

which version of  
equals invoked?

①  
②

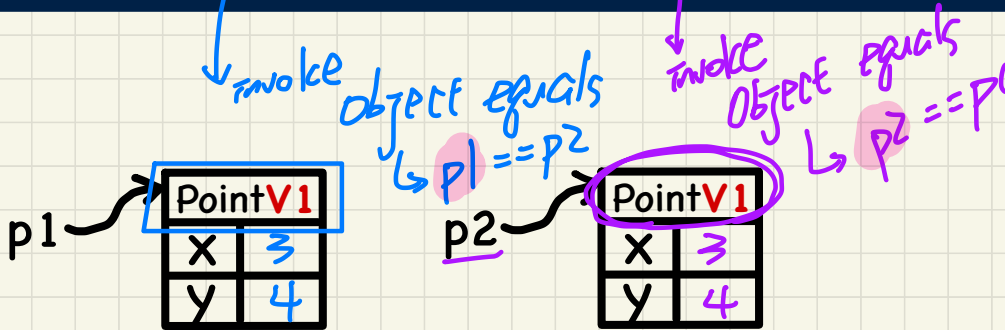
D.T. PointV2  
↳ PointV2 var.

D.T. PointV1  
↳ Object var.



# 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());
}
```



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

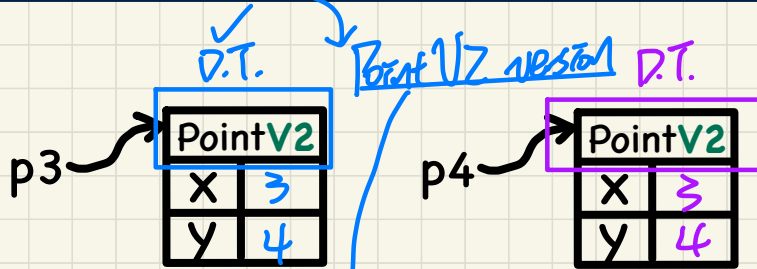
extends

```
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 */
 assertTrue(p3.equals(p4)); assertTrue(p4.equals(p3));
 assertEquals(p3, p4); assertEquals(p4, p3);
}
```

pass I  
assertFalse (p3 == p4)  
assertSame (p3, p4)  
p3 == p4  
False



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

extends

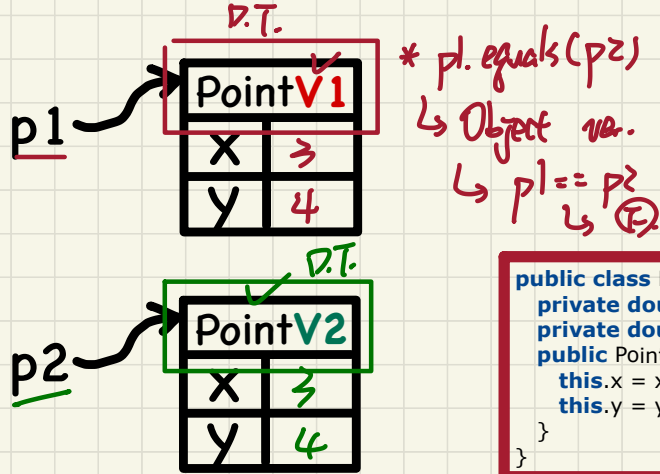
```
public class PointV2 {
 private int x;
 private int y;
 public PointV2(int x, int y) { ... }
 public boolean equals(Object obj) {
 if (this == obj) { return true; }
 if (obj == null) { return false; }
 if (this.getClass() != obj.getClass()) { return false; }
 PointV2 other = (PointV2) obj;
 return this.x == other.x
 && this.y == other.y;
 }
}
```

true.

# 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. */
 /* assertEquals(p1, p2); assertEquals(p2, p1); */
 /* assertEquals can take objects of different types and fail. */
 /* assertEquals(p1, p2); → "p1 == p2" → fail */
 /* assertEquals(p2, p1); → "p2 == p1" → fail */
 /* version of equals from Object is called */
 assertEquals(p1, p2); → pass
 /* version of equals from PointV2 is called */
 assertEquals(p2, p1); → pass
}
```

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



extends

\*\* p2.eq.(p1)  
↳ PointV2 var.

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;
 }
}
```

## Math

$$P \wedge Q$$

$$P \vee Q$$

Commutativity

$$P \wedge Q \equiv Q \wedge P$$

## Java

$$P \&\& Q$$

$$P \parallel Q$$

$$P \&\& Q$$

$$Q \&\& P$$

(short-circuit  
evaluation)

# 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");
 }
}
```

Handwritten annotations on the code:

- 0 != 0 (boxed, F.)
- && (boxed, F.)
- y / x > 2 (circled, bypass.)
- q.c. (under the if statement)

exp1 && exp2

① Eval. L → R

②.① Eval exp1: false  
Skip eval. of exp2  
&& eval to false.

②.② Eval exp1.: true  
Eval exp2.

# 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:"); e.c.
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");
}
```

*Handwritten notes:*

- $0 == 0$  is boxed in green, with a green 'T' below it.
- $10 / 0 > 2$  is written in orange, with a green 'F' below it.
- $0 == 0$  is underlined with a green line, with a green 'T' below it.
- $10 / 0 > 2$  is underlined with an orange line, with a green 'F' below it.
- bypass* is written in green.
- return (T)* is written in green.

expl || exp2

① Eval L → R

②.1 Eval expl. → T  
Skip eval. exp. 2  
↳ —||— return (T).

②.2 Eval expl → F  
Eval exp2.

# Short Circuit Evaluation

①

$x \neq 0$   
 $\boxed{\text{exp1}}$

$\&\&$

exp2  
↳

$y/x$   
 $a[i]$

guarding  
constraint

things might  
go wrong

②

$x == 0$   
 $\boxed{\text{exp1}}$

$||$

exp2

error  
condition

## Exercise

① (A) ~~==~~ (B) ~~==~~  $a[i] > 0$

AIDFBĒ

② (C) ~~||~~ (D) ~~||~~  $a[i] > 0$



# Short-Circuit Evaluation: Common Errors

EXERCISE

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 */ }
```

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

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

```

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;
 }
}

```

✓ (A) if ( ① || ② ) { return false; }

✗ (B) if ( ③ || ④ ) { return false; }

... obj.getClass()... obj == null What if obj is null.  
 NPE not even reached

**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 }
```

- (1) Object
- (2) Point VZ
- (3) Person
- (4) String

String  
Person  
this.firstName.equals(...)

**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)`

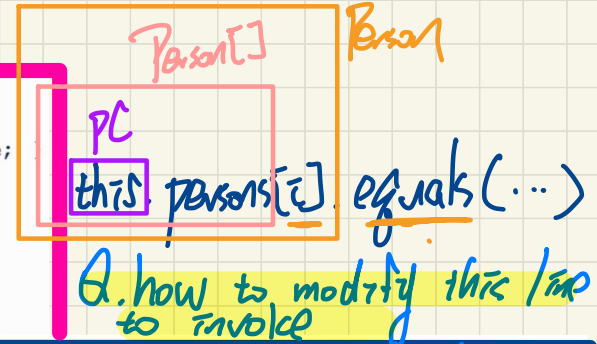
**Q3:** At Lines 11 & 12 which version of the **equals** method is called?

## 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 }
```

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 }
```

equals from the String class?

## Lecture 14 - Oct. 29

### Call by Value

***equals: PersonsCollector  
Call by Value***

## Announcements/Reminders

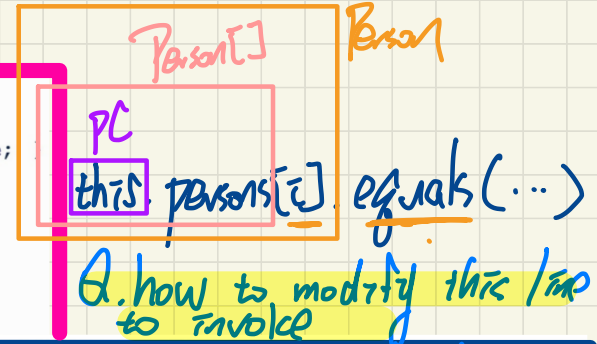
- **ProgTest1** marks, submissions, grading tests released
- **ProgTest2** tomorrow during your enrolled lab session
- **Lab3** due this Friday at noon

## 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 }
```

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 }
```

equals from the String class?

# 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));
}
```

\* p1.equals(p2) (T) — which parts?  
\*\* p3.equals(p4) (T)

p1

| Person |     |
|--------|-----|
| fn     | A   |
| ln     | a   |
| w      | 180 |
| h      | 1.8 |

p2

| Person |     |
|--------|-----|
| fn     | A   |
| ln     | a   |
| w      | 180 |
| h      | 1.8 |

p3

| Person |     |
|--------|-----|
| fn     | B   |
| ln     | b   |
| w      | 200 |
| h      | 2.1 |

p4

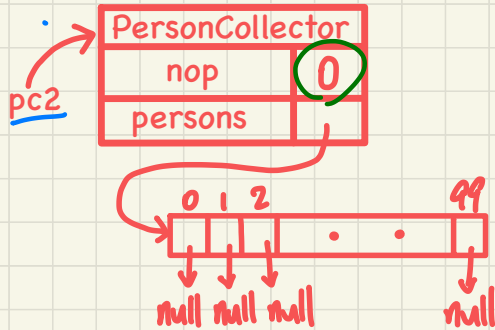
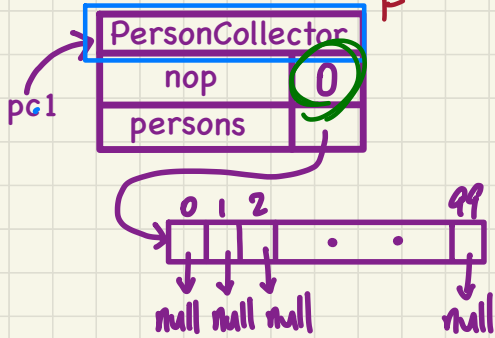
```
public class Person {
 private String firstName; private String lastName;
 private double weight; private double height;
 public boolean equals(Object obj) {
 if (this == obj) { return true; }
 if (obj == null || this.getClass() != obj.getClass()) { return false; }
 Person other = (Person) obj;
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

```
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);
}
```

Person

p3 == p4

pc1.persons[1].equals  
(pc2.persons[1]);

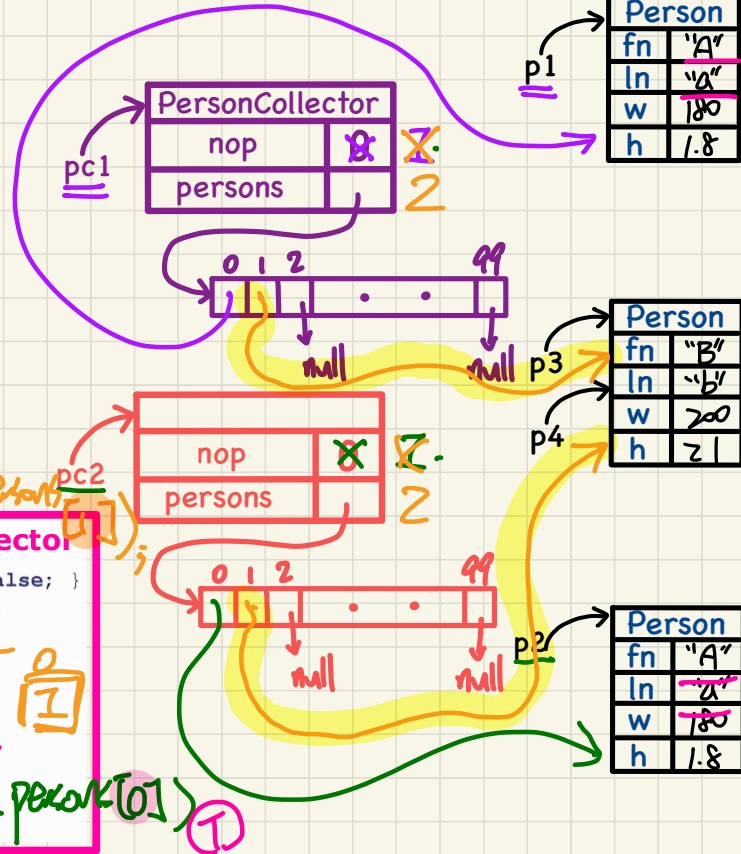
```
public boolean equals(Object obj) {
 if(this == obj) { return true; }
 if(obj == null || this.getClass() != obj.getClass()) { return false; }
 PersonCollector other = (PersonCollector) obj;
 boolean equal = false;
 if(this.nop == other.nop) {
 equal = true;
 for(int i = 0; equal && i < this.nop;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

PersonCollector

i = 0

Person

equal = pc1.persons[0].equals(pc2.persons[0]);



# 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));
```

\* 3rd iteration:

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

```
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);
}
```

Person

```
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; i < this.nop; i++) {
 equal = this.persons[i].equals(other.persons[i]);
 }
 }
 return equal;
```

PersonCollector

| Person |      |
|--------|------|
| fn     | A    |
| ln     | a    |
| w      | 175  |
| h      | 1.75 |

| Person |     |
|--------|-----|
| fn     | "A" |
| ln     | "a" |
| w      | 180 |
| h      | 1.8 |

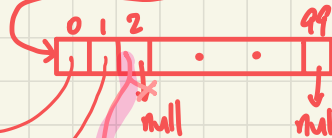
| Person |     |
|--------|-----|
| fn     | "B" |
| ln     | "b" |
| w      | 200 |
| h      | 2.1 |

| Person |     |
|--------|-----|
| fn     | "A" |
| ln     | "a" |
| w      | 180 |
| h      | 1.8 |

| PersonCollector |   |
|-----------------|---|
| nop             | 2 |
| persons         |   |



| PersonCollector |   |
|-----------------|---|
| nop             | 2 |
| persons         |   |



| Person |      |
|--------|------|
| fn     | A    |
| ln     | a    |
| w      | 165  |
| h      | 1.55 |

## Method Call: Callee vs. Caller

```
class A {
 ...
 void m(T param) {
 /* use of param */
 }
}
```

*declaration of callee method*

*variable.*

```
class B {
 ...
 void n(...){
 A co = new A();
 co.m(arg);
 }
}
```

*caller: B.n*

*call by value*

*callee A.m*

Call by value :

*param = arg*

when making a method call, there's an implicit variable assignment:

*param = arg;*

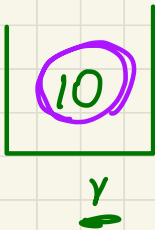
*concrete value used to call the method*

## Call by Value: Primitive Argument

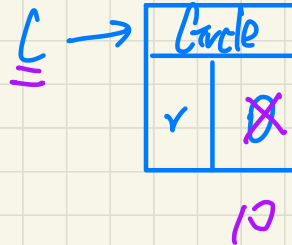
```
class Circle {
 int radius;
 void setRadius(int r) {
 this.radius = r;
 }
}
```

Call by value!

r = arg



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

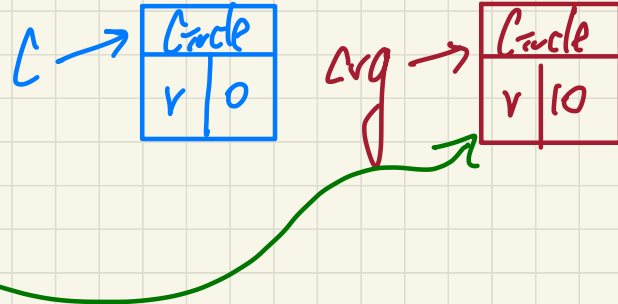


## Call by Value: Reference Argument

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

Call by  
value:  
c = arg

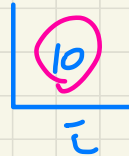
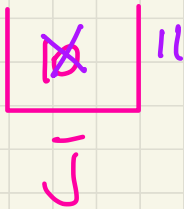
```
class CircleUser {
 ...
 → Circle c = new Circle();
 → Circle arg = new Circle(10);
 c.setRadius(arg);
}
```



# 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 }
```



i is untouched by the method.

## 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); } }

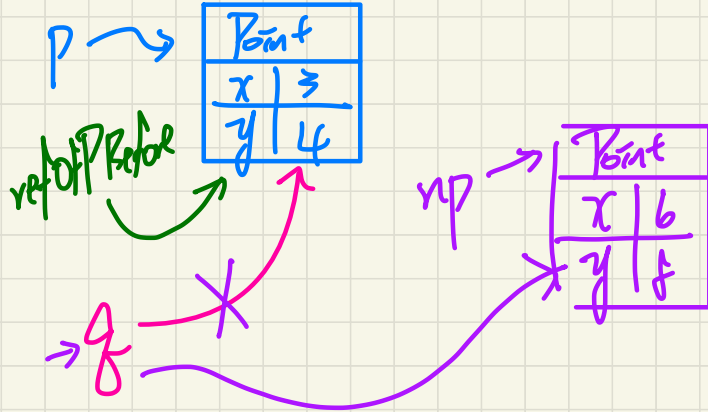
```

```

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 }

```

→ address  
the value  
of  $P$   
stays the  
same.



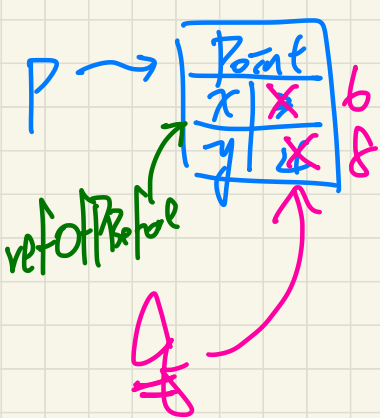
```
public class Point {
 private int x;
 private int y;
 public Point(int x, int y) {
 this.x = x;
 this.y = y;
 }
 public int getX() { return this.x; }
 public int getY() { return this.y; }
 public void moveVertically(int y){ this.y += y; }
 public void moveHorizontally(int x){ this.x += x; }
}
```



# 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 }
```



Call by value:  
 $q = p$

```
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; }
}
```

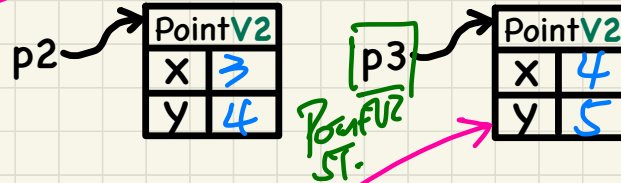
# Call by Value: Invoking the Overridden equals

```
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;
 }
}
```

```
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 */
```



Handwritten notes:

- obj** (with a pink arrow pointing to the `obj` parameter in the `equals` method)
- ST: object** (with a pink arrow pointing to the `obj` parameter)
- obj.x** and **obj.y** (with pink arrows pointing to the `x` and `y` attributes of the object)

Call by value:  
obj = p3

## Lecture 15 - Oct. 31

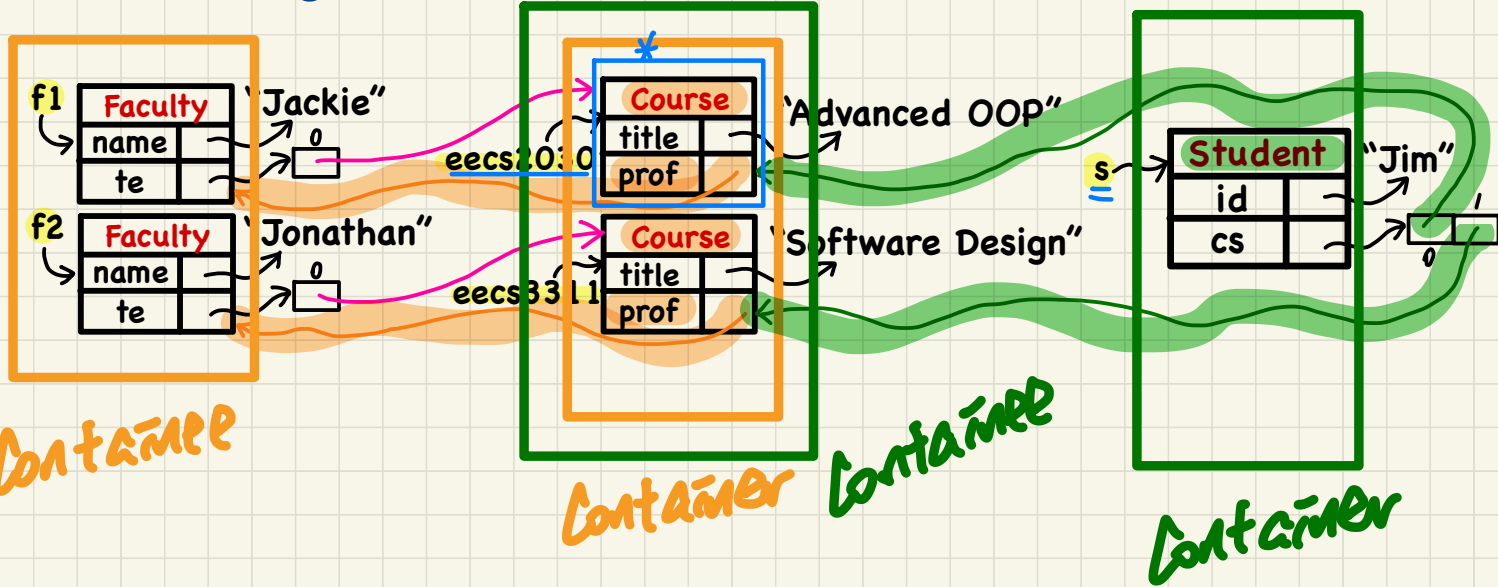
### Aggregation and Composition

***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



sharing: aliasing

\* f1.te[0] sharing.  
s.cs[0] (=) (T)

aggregation: sharing/aliasing

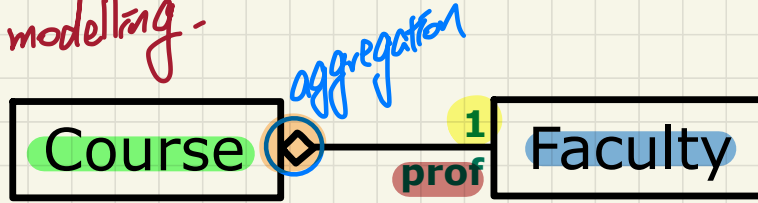
composition: little/no sharing

# Aggregation: Design



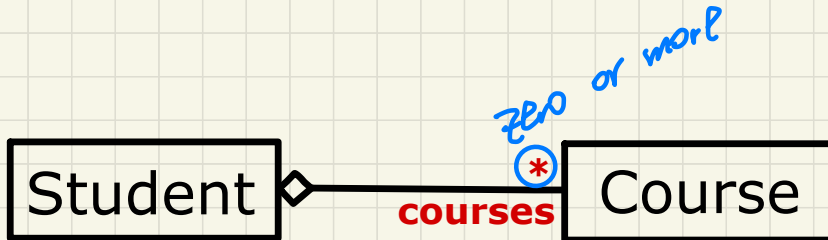
## Design 1: Single Containee

modelling -



A **Course** has, by aggregation, a single **faculty** as its **prof.**

## Design 2: Multiple Containees



# Java Implementation

programming

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

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

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

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

# Aggregation (1)

| Course |  |
|--------|--|
| title  |  |
| prof   |  |

| Faculty |  |
|---------|--|
| name    |  |
|         |  |

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

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

```
@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)

| Student |  |
|---------|--|
| id      |  |
| cs      |  |

| Faculty |  |
|---------|--|
| name    |  |
| te      |  |

| Course |  |
|--------|--|
| title  |  |
| prof   |  |

```
@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]);
}
```

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

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

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



# 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
 ...
}
```

Handwritten notes:  $\chi$  ✓,  $\gamma$  ✗

```
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();
 ① result = this.ob.bi;
 result = this.getB().bi;
 result = this.ob.getBi();
 result = this.getB().getBi();
 ② result = this.ob.getA().ai;
 result = this.ob.getA().getAi();
 result = this.ob.oi.ai;
 result = this.ob.oi.getAi();
 return result;
 }
}
```

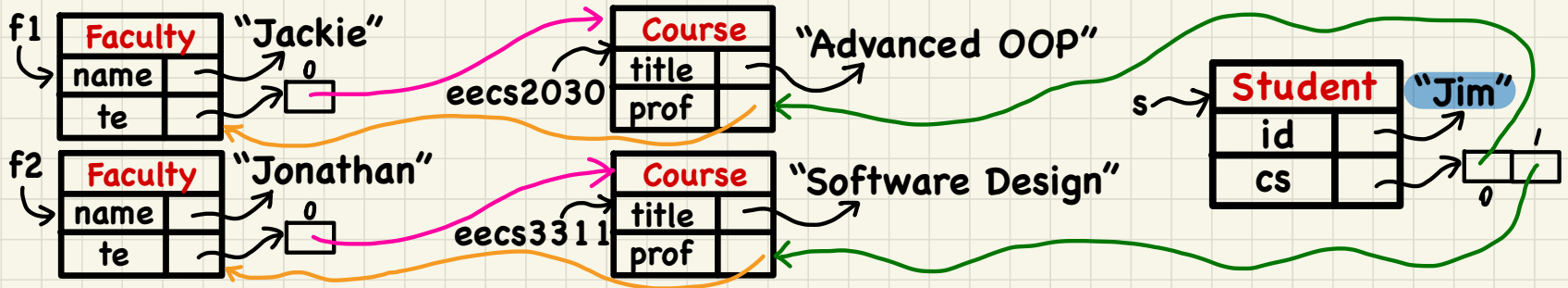
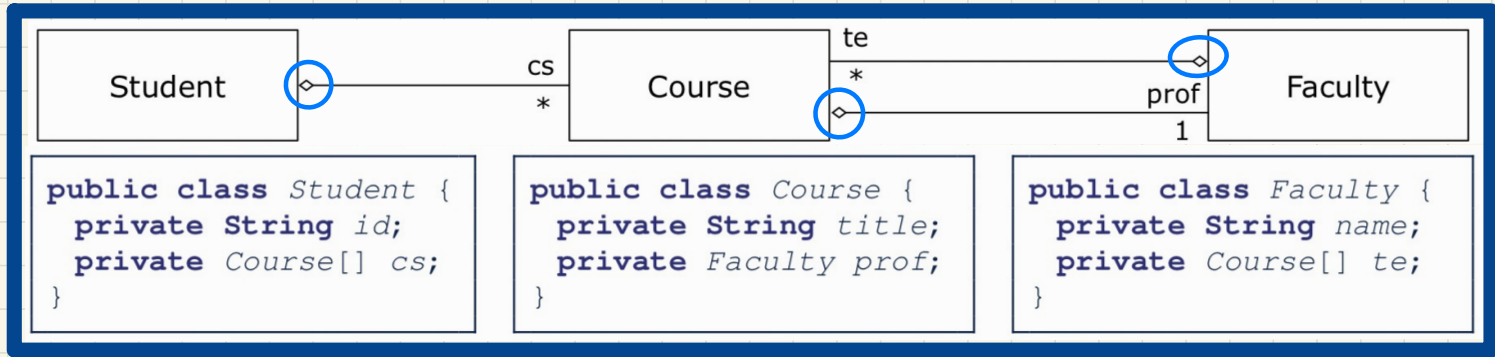
Handwritten notes: **Context: A** (with a red 'X' pointing to `this.ob.bi`), **Context path** (pointing to `this.ob`), **ob declared in A** (in a pink box), **B** (underlined, with a red 'X' pointing to `this.ob.bi`), **A** (in a green circle, pointing to `this.ob.oi.getAi()`), **private** (in red, pointing to `ai`).

① ✓  
this: ob - bi  
ob declared in A

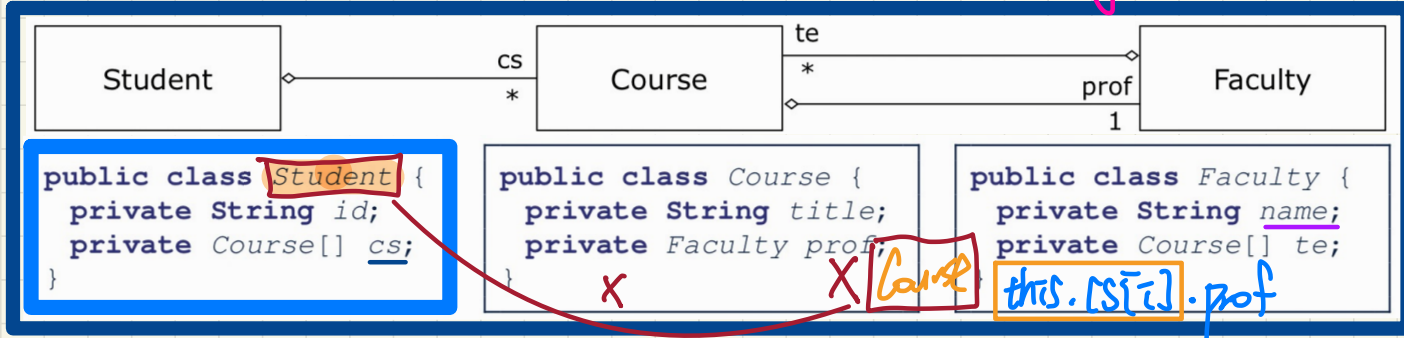
② this.ob.getA().ai  
B A

```
public class B {
 private A oa;
 private int bi;
 public A getA() {
 return this.oi;
 }
 public int getBi() {
 return this.bi;
 }
}
```

# Runtime Object Structure: Student, Course, Faculty



# Dot Notation for Navigating Classes (1) *\*s.getName() → Jonathan.*



```

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

```

Handwritten: *this.getId()*

```

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

```

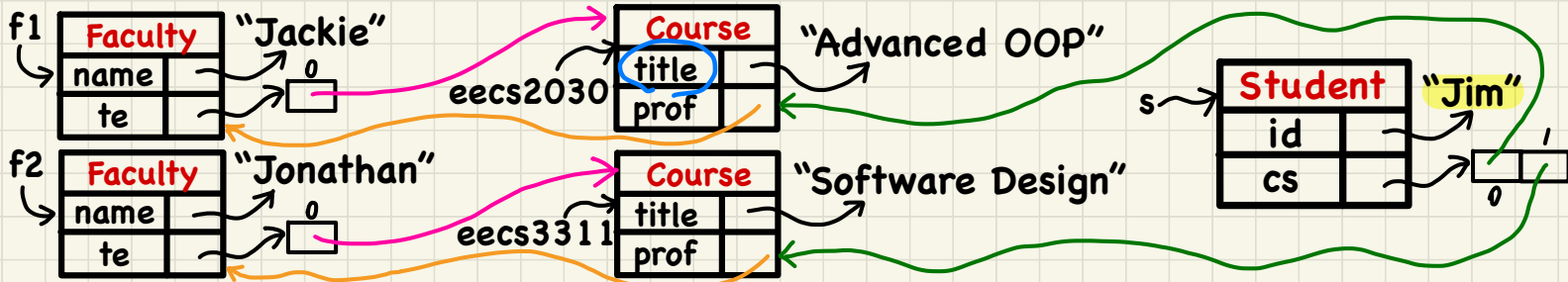
Handwritten: *s.getTitle()*, *Course*

```

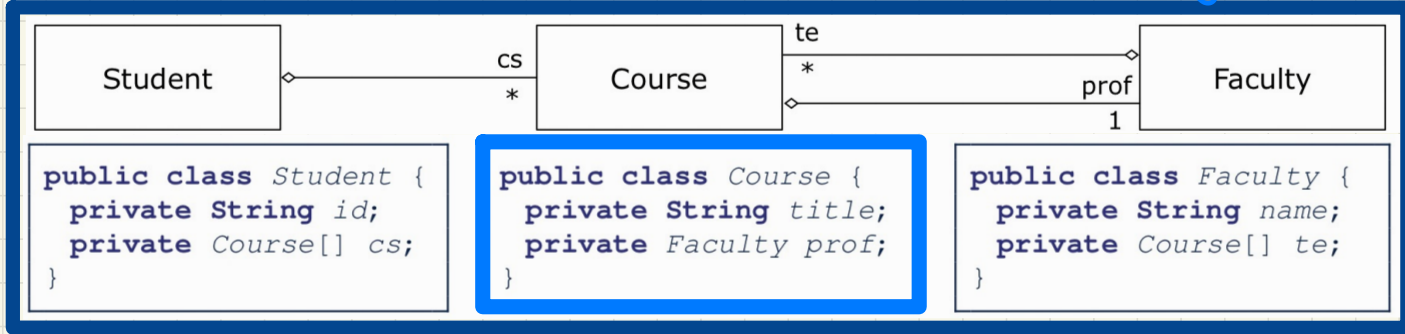
/* Name of *
 * ith course's instructor
*/
String getName(int i) {
 this.cs[i].getProf().getName()
}

```

Handwritten: *prof? name?*



# Dot Notation for Navigating Classes (2) `eecs3311.getTitle(0)`



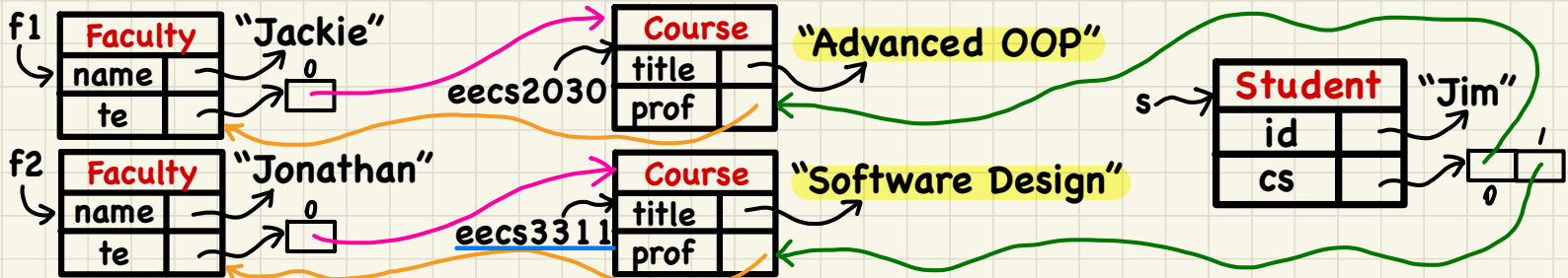
```
/* Get course's title.
*/
String getTitle() {

}
```

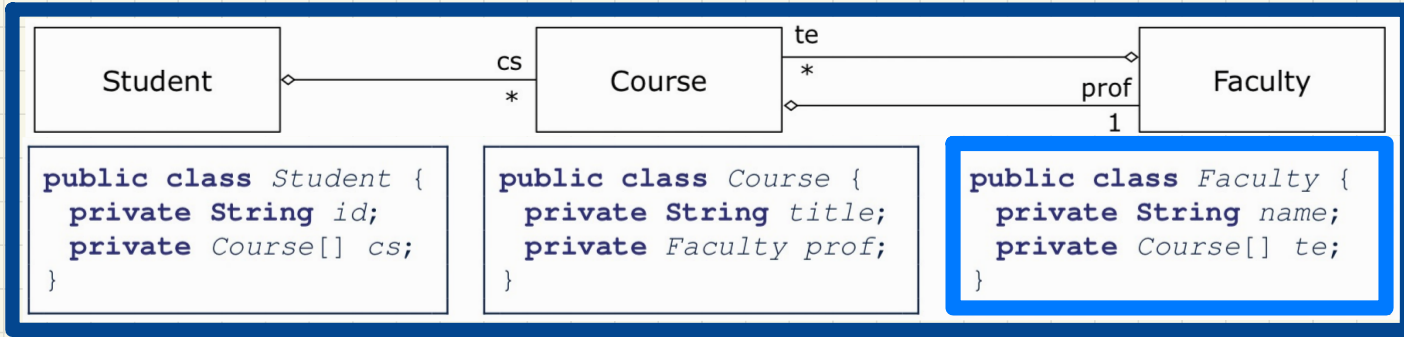
```
/* Name of instructor
*/
String getName() {

}
```

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



# Dot Notation for Navigating Classes (3)

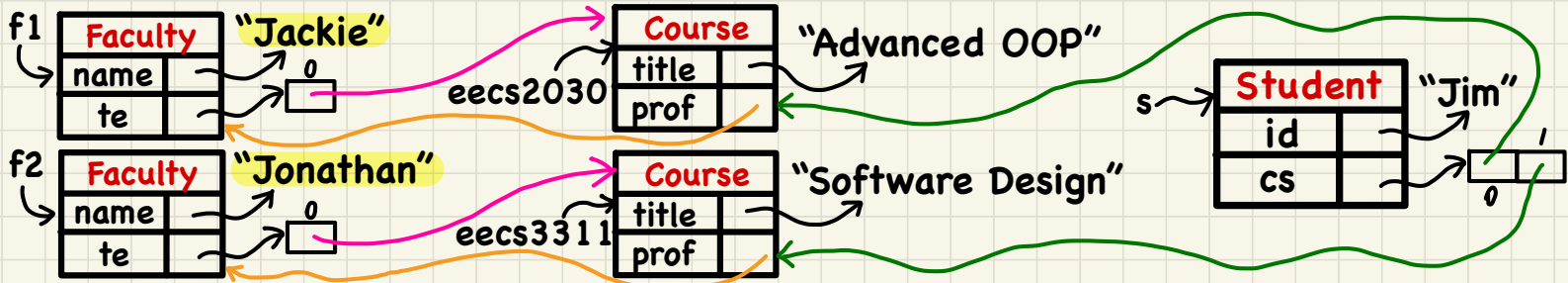


```
/* Name of instructor
*/
String getName() {

}
```

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

}
```



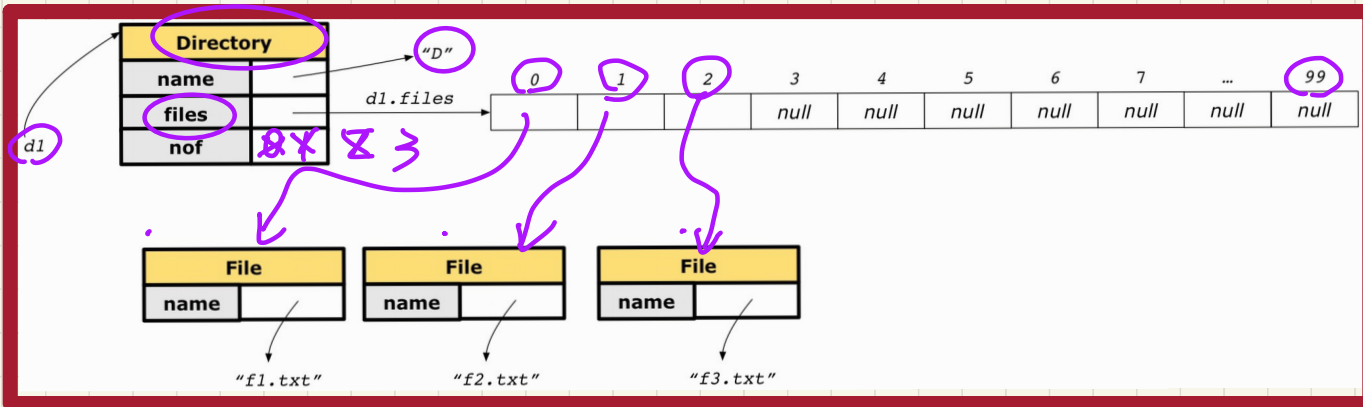
# Composition: No Sharing

```
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)
}
```



```
public class X {
```

```
 public X(X other) {
```

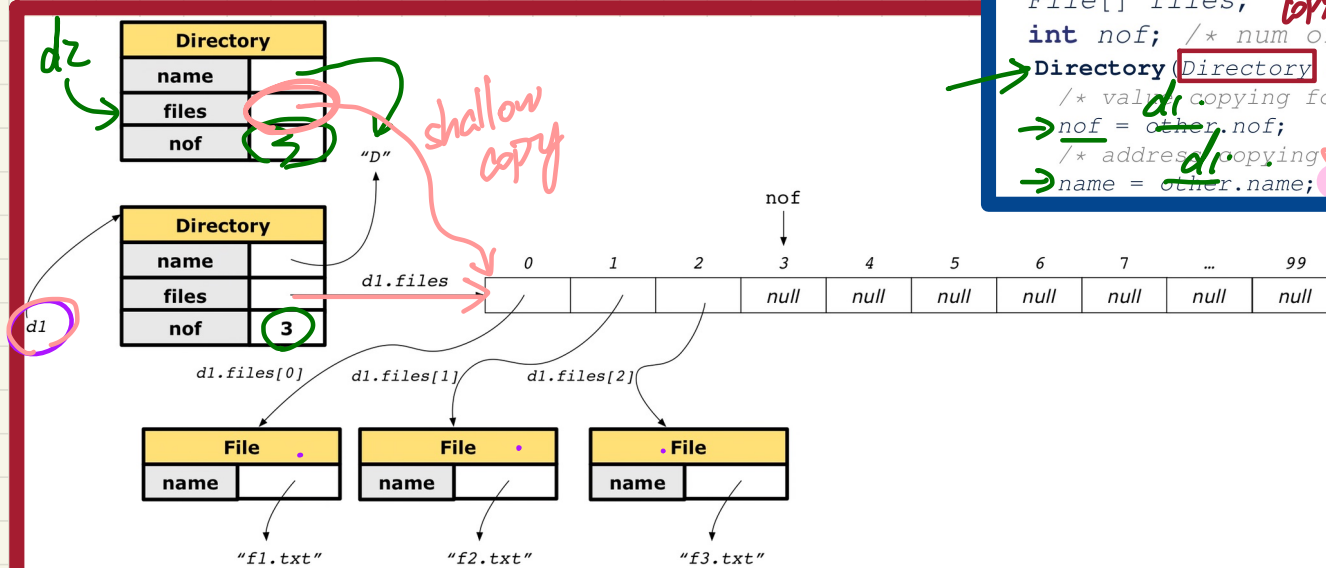
Copy constructor.

...  
Create based on an X object  
on another X object

# 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")); }
}
```

```
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; }
}
```





## Lecture 16 - Nov. 5

### Composition, Inheritance

***Composition: Deep Copy***

***Design Attempts without Inheritance***

***Inheritance: Use of extends, super***

## Announcements/Reminders

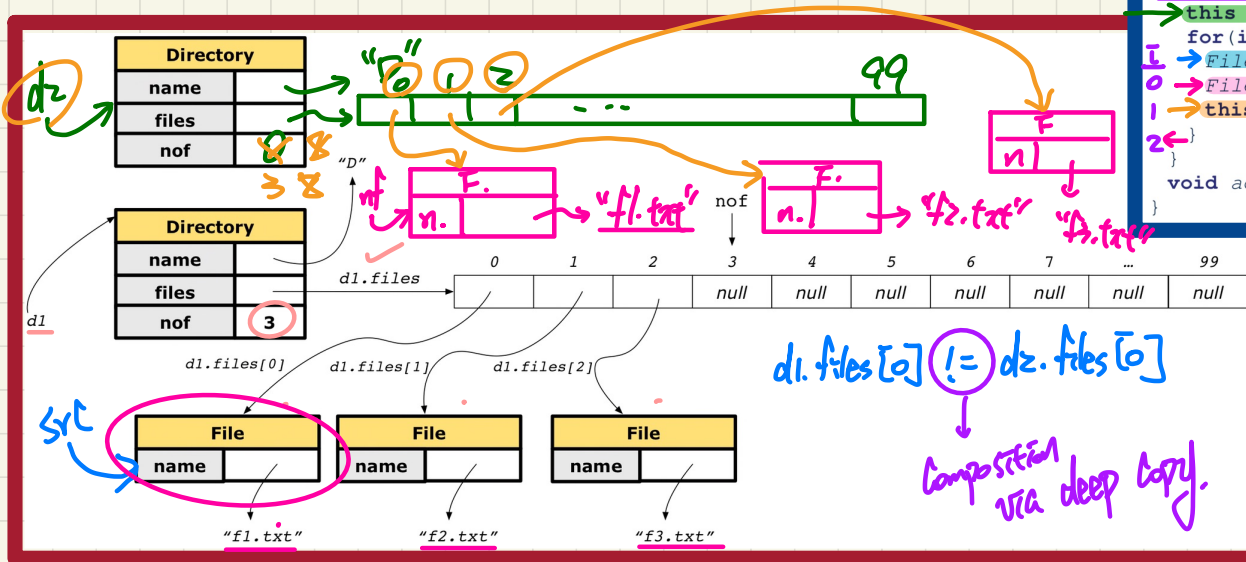
- **Lab4** released (**ProgTest3** on November 20)
- Guide & Questions for **WrittenTest2** to be released tmw
- In-Lab Demo on **Inheritance** tomorrow
- **ProgTest2** results & feedback tentatively Monday Nov 11

# 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) { ... }
 }
}
```



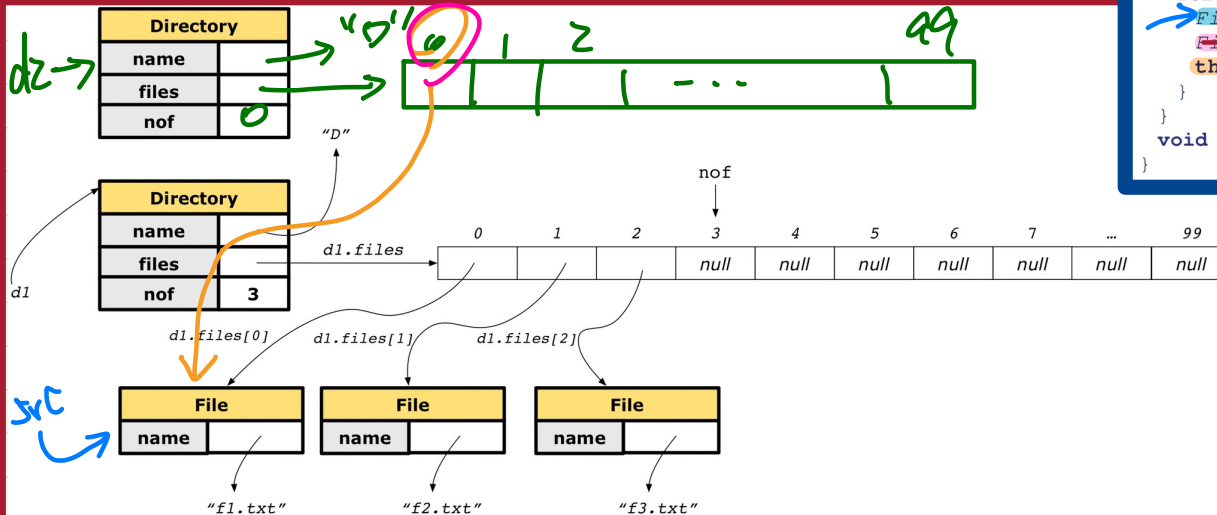
# Exercise: Copy Constructor (Composition?)

```
@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]);
}
```

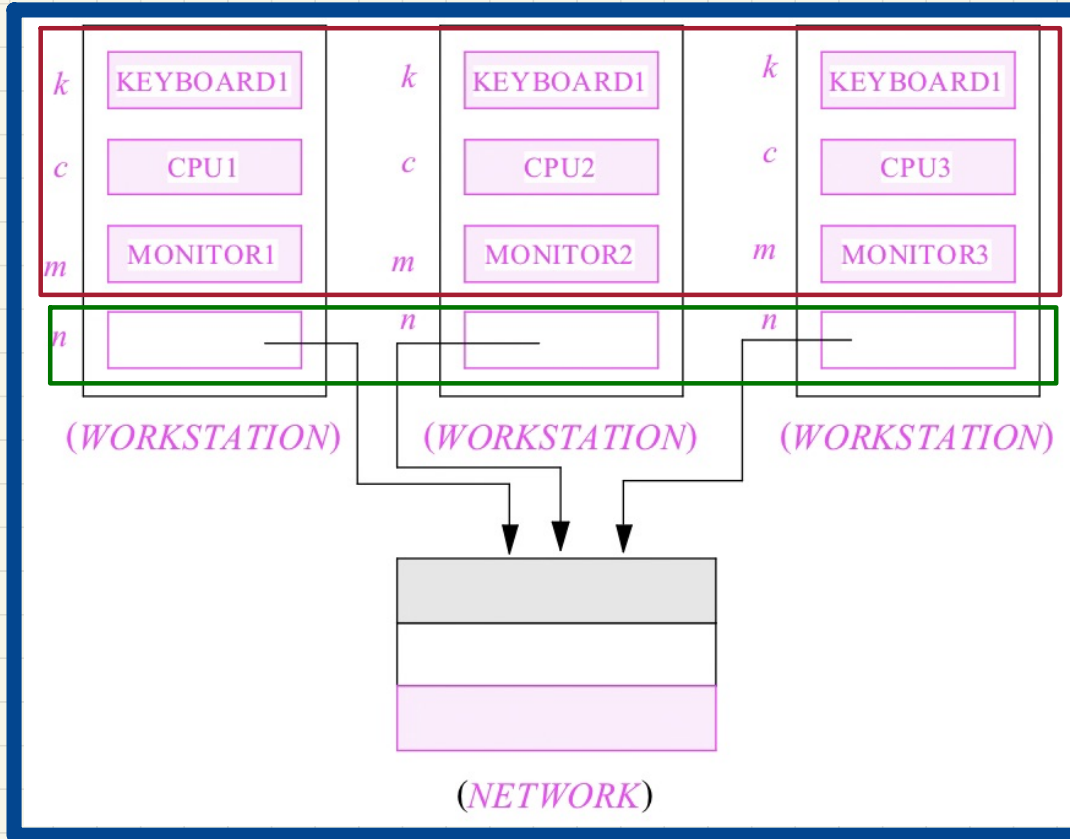
```
class File {
 File(File other) {
 this.name =
 new String(other.name);
 }
}
```

```
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) src);
 }
 }
 void addFile(File f) { ... }
}
```

$d1.files \neq d2.files$      $d2.files[0] == \textcircled{D}$



# Modelling: Aggregation vs. Composition

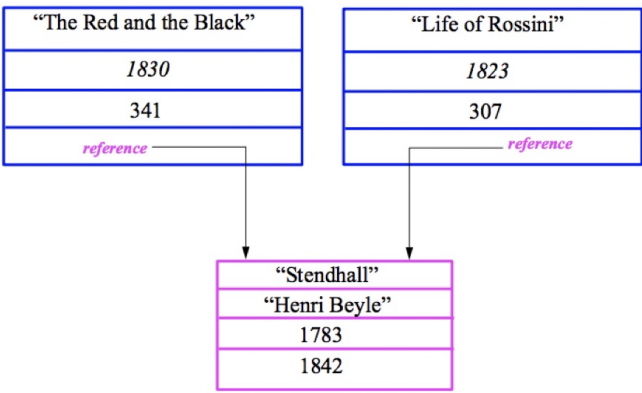


*Composition*

*aggregation*

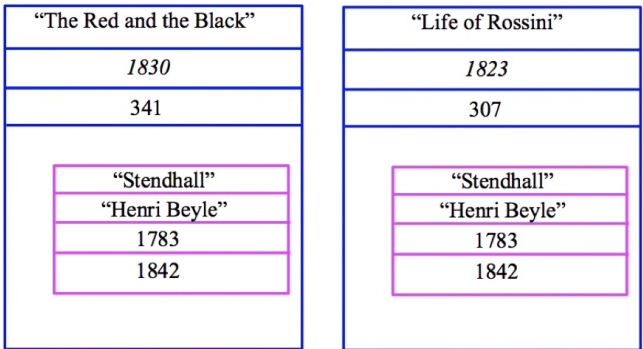
# Implementation: Aggregation or Composition

## author as an *aggregation*



Hyperlinked author page

## author as a *composition*



Physical printed copies

## 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

how to simulate OO in a non-OO language?

```
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;
 }
 ...
}
```

*Handwritten notes:*  
- **RS** (Red Square) next to `kind`  
- **NRS** (Pink Square) next to `premiumRate` and `discountRate`  
- `kind` is underlined in pink  
- `premiumRate` and `discountRate` are underlined in pink  
- `kind` is underlined in pink in the constructor  
- `kind` is underlined in pink in the constructor  
- `kind` is underlined in pink in the constructor

*Handwritten code:*  
`Student rs = new Student(1);`  
`Student nrs = new Student(2);`

```
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;
 }
}
```

*Handwritten notes:*  
- `tuition` is underlined in blue  
- `base amt.` is written in blue below `tuition`

```
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 ++;
 }
}
```



# 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;
 }
 ...
}
```

*only apply to RS*

*only apply to NRS*

```
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 a class follow 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 == 2) {
 return tuition * this.discountRate;
 }
 else if (this.kind == 3) { ... }
```

```
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 Single Choice Principle

- **Repeated** if-conditions
- A new kind is **introduced?**
- An existing kind is **obsolete?**

informational  
"3"

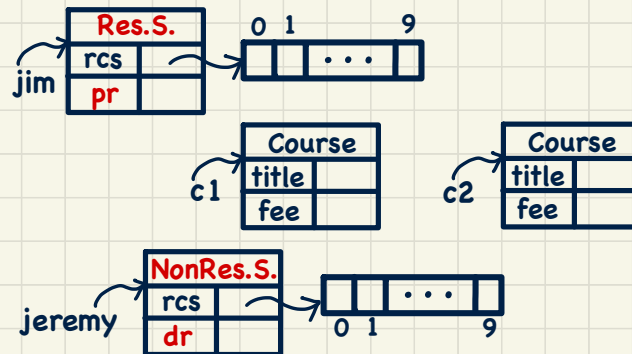
else if  
(this.kind == 3)  
{ ... }

# Testing Student Classes (without inheritance)

```
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;
 }
}
```

```
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;
 }
}
```

```
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)

\* invoke method defined at the super class

```
class Student { parent/super class
 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 */
 }
}
```

child/sub class  
RS inherits all attributes and methods from S

RS and NRS have the same parent/super class.

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

*Invoking constructor at the super class*

```
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***

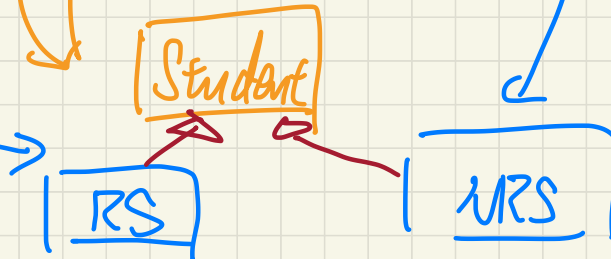
***Visualizing Child Objects***

***Tracing Method Calls in Eclipse***

# Student Classes (without inheritance)

```
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;
 }
}
```

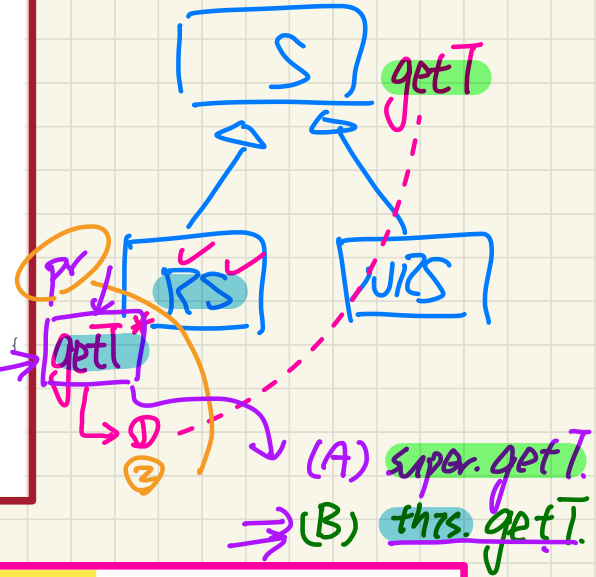
```
public class NonResidentStudent {
 private String name;
 private Course[] courses; private int noc;
 private double discountRate; /* assume a
 public NonResidentStudent (String name) {
 this.name = name;
 this.courses = new Course[10];
 }
 public void register(Course c) {
 this.courses[this.noc] = 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)

```
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 */
 }
}
```

getT  
getT  
getT



```
class ResidentStudent extends Student {
 double premiumRate; /* there's a mutator method */
 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;
 }
}
```

## Lecture 17 - Nov. 7

### Inheritance

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



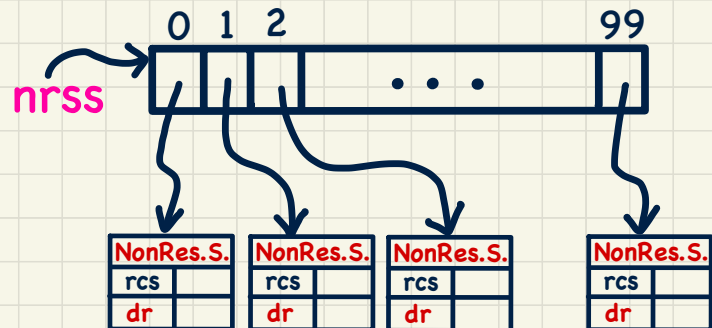
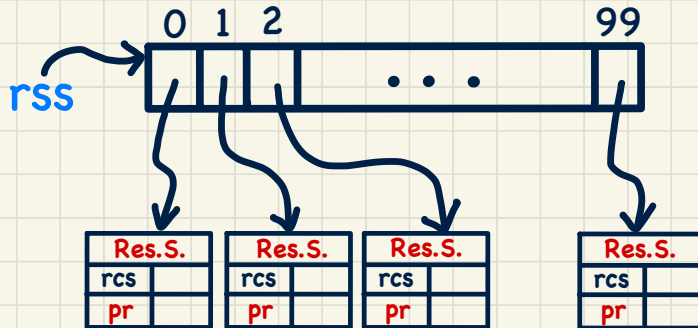
## Announcements/Reminders

- **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); }
 }
}
```

↓ # for loops correspond to # kinds of students



# Recall: 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 */
 }
}
```

In writing a subclass :

(1) cannot re-declare inherited attributes

(2) add new attributes/methods.

(3) happy with inherited methods, leave as is

(4) not satisfied with inherited methods → override them

inherited  
(and cannot be re-declared)

not overridden

parent version

overridden

new attributes in sub-classes

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

set P

child version 1

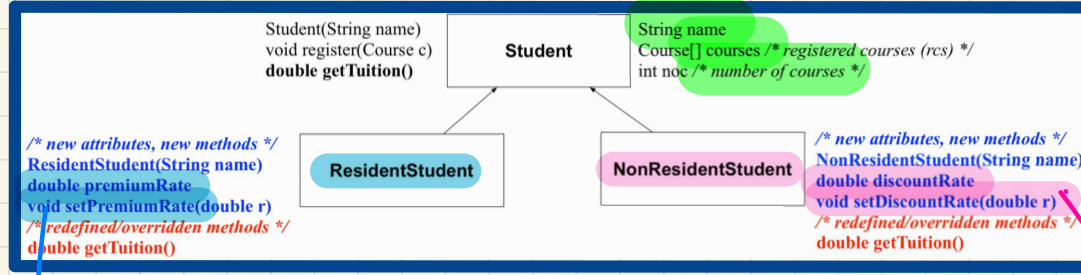
this.name = "...";

```
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;
 }
}
```

set P

child version 2

# Visualizing Parent and Child Objects



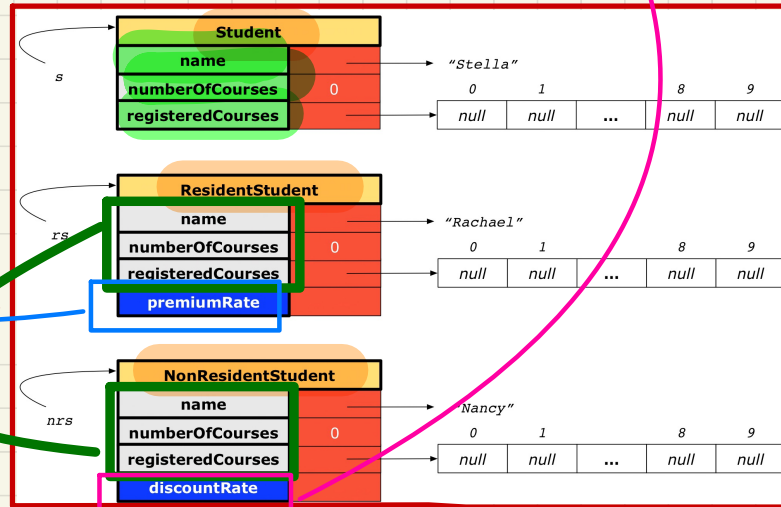
## Inheritance Hierarchy

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

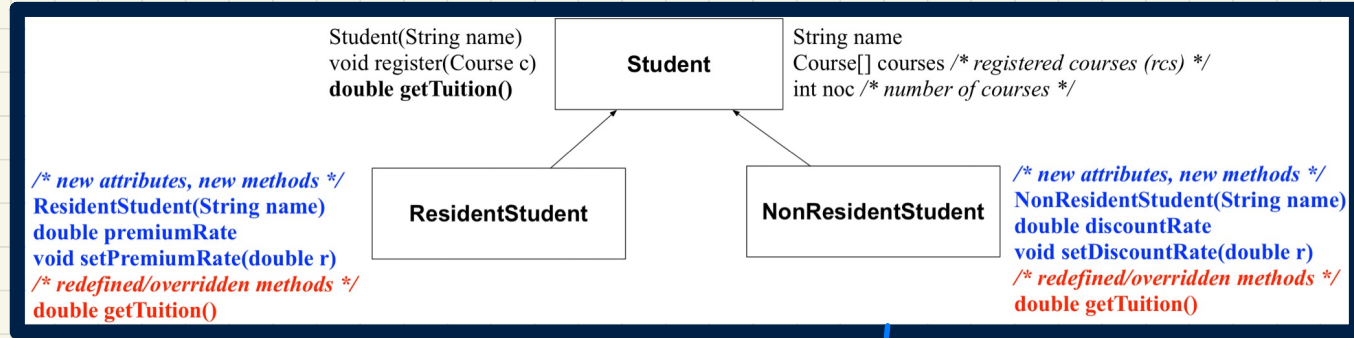
## Declaring Static Types

## Runtime Object Structure

*Inherited from Student*



# Testing Student Classes (with inheritance)



```

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());
 }
}

```

→ in-lab demo

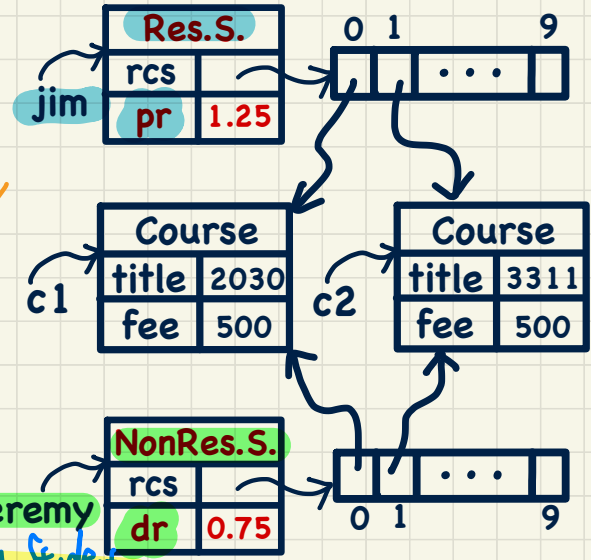
→ invoke Student constructor

version in Student invoked

→ invoke ① the specific

child version

② both first invoke getT from Student



# Modifiers in Java

① private

② protected

③ public

④ no modifier

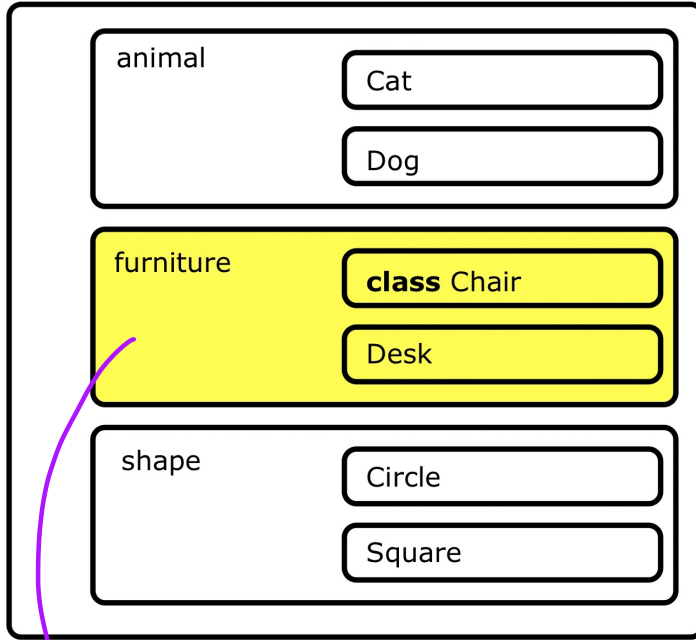
inheritance

classes

attributes/methods

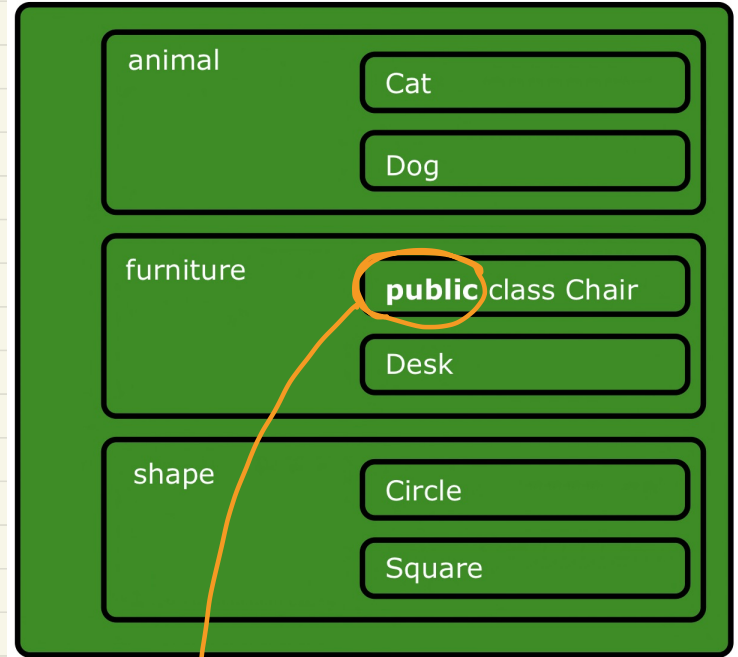
# Visibility: Classes

CollectionOfStuffs



package-level  
visibility

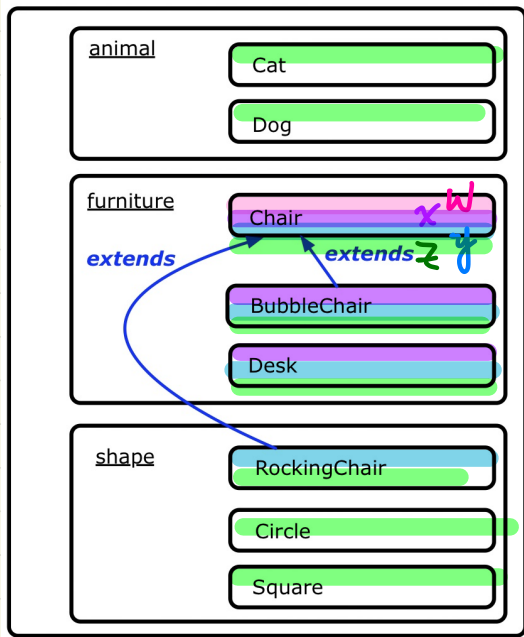
CollectionOfStuffs



universal  
visibility.

# Visibility: Attributes and Methods

CollectionOfStuffs



```

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

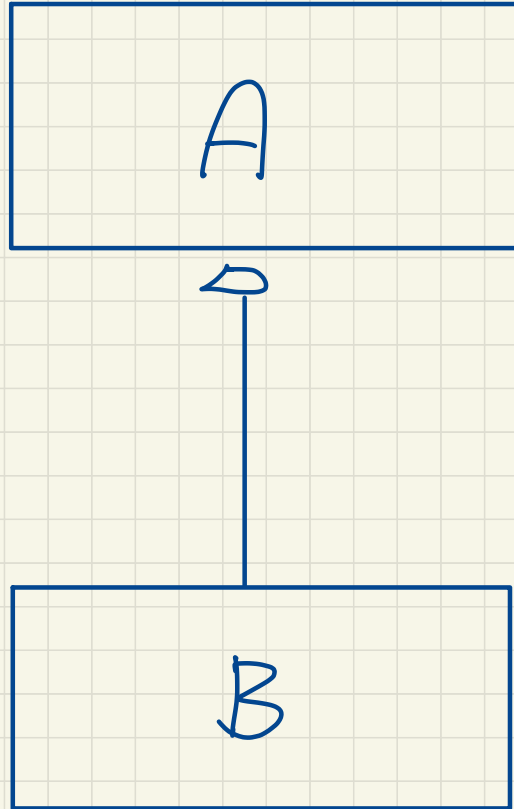
```

not inherited to subclasses

as if:  
(1) pkg visib.  
(2) subclass (in diff. pkg.)

|             | CLASS | PACKAGE | SUBCLASS<br>(same pkg) | SUBCLASS<br>(different pkg) | NON-SUBCLASS<br>(across Project) |
|-------------|-------|---------|------------------------|-----------------------------|----------------------------------|
| public      | Green | Green   | Green                  | Green                       | Green                            |
| protected   | Green | Green   | Green                  | Green                       | Red                              |
| no modifier | Green | Green   | Green                  | Red                         | Red                              |
| private     | Green | Red     | Red                    | Red                         | Red                              |





(1) all attributes private.

(2) as you discover  
necessarily, make attrs.  
protected

# Student Classes (with inheritance): Expectations

exp. of a variable determines the range of atts/methods that can be called on that variable

Student(String name)  
void register(Course c)  
double getTuition()

Student

String name  
Course[] courses /\* registered courses (rcs) \*/  
int noc /\* number of courses \*/

ResidentStudent

NonResidentStudent

/\* new attributes, new methods \*/  
ResidentStudent(String name)  
double premiumRate  
void setPremiumRate(double r)  
/\* redefined/overridden methods \*/  
double getTuition()

/\* new attributes, new methods \*/  
NonResidentStudent(String name)  
double discountRate  
void setDiscountRate(double r)  
/\* redefined/overridden methods \*/  
double getTuition()

exp(RS) = exp(Student) + new atts/methods

sibling classes

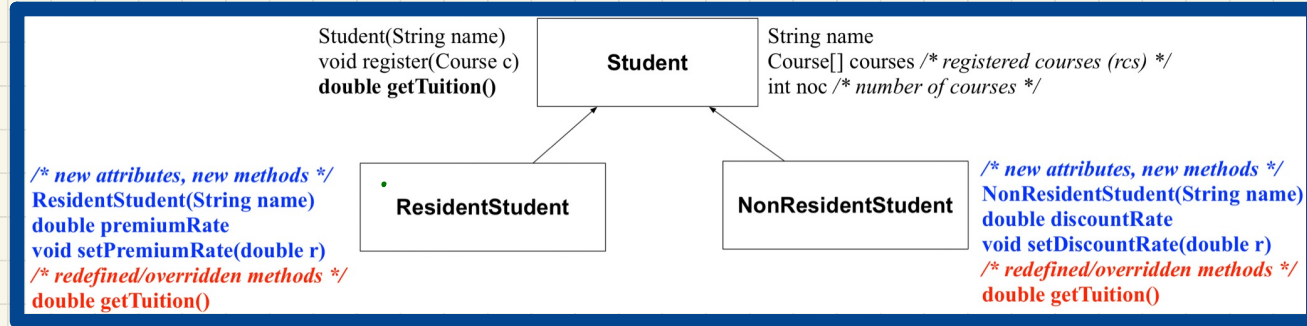
```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

inherited exp from parent

|      | name | rcs | noc | reg | getT | pr | setPR | dr | setDR |
|------|------|-----|-----|-----|------|----|-------|----|-------|
| S.   |      |     |     |     |      |    |       |    |       |
| rs.  |      |     |     |     |      |    |       |    |       |
| nrs. |      |     |     |     |      |    |       |    |       |

sibling exp.  
new exp introduced in subclasses

# Intuition: Polymorphism



```

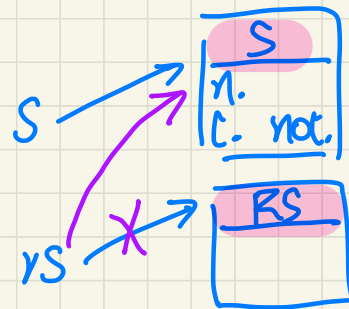
1 Student s = new Student("Stella");
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.25);
4 s = rs; (A) Is this valid? */
5 rs = s; (B) Is this valid? */

```

ST: parent class

ST of subclass

1. Assume  $rs \odot S$  compiles.
2. Execute the re-assignment.
3. Exp. of  $rs$ : \*



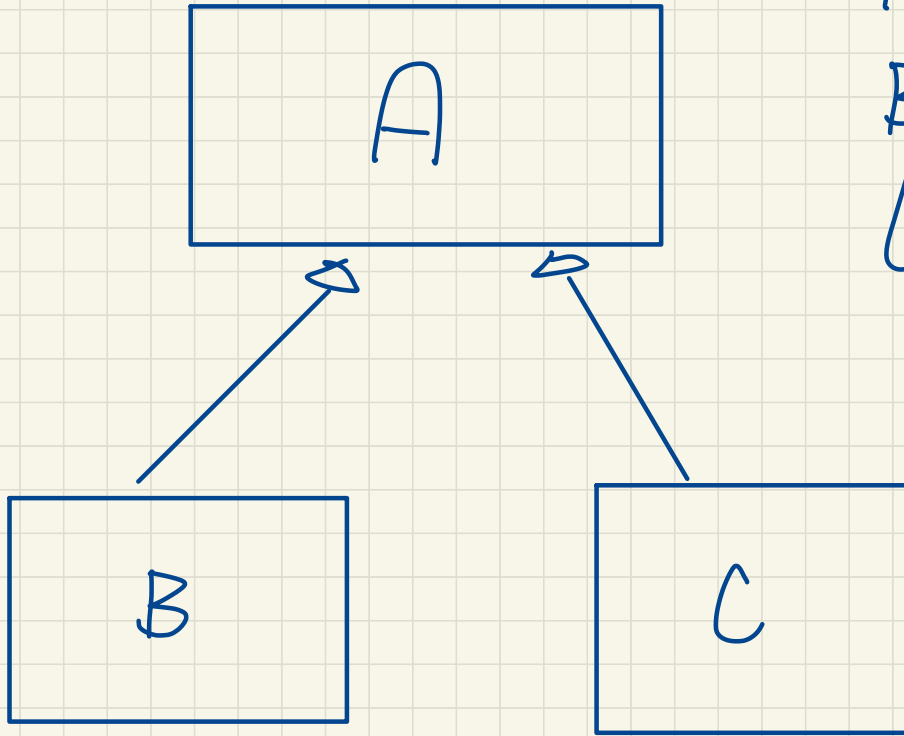
\*

rs. name  
rs. noc  
rs. courses  
rs. pr

Student  
does not  
crash  
have  
pr

4. Assumption was wrong:

$rs = S$   
not compatible



A    o1 = new A();

B    o2 = new B();

C    o3 = new C();

o1 = o2;

o1 = o3;

o2 = o3;

o2 = o1;

o3 = o1;

o3 = o2;

## Written Test 2

### Review Q&A

***Call by Value, Caller vs. Callee  
assertSame vs. ==***

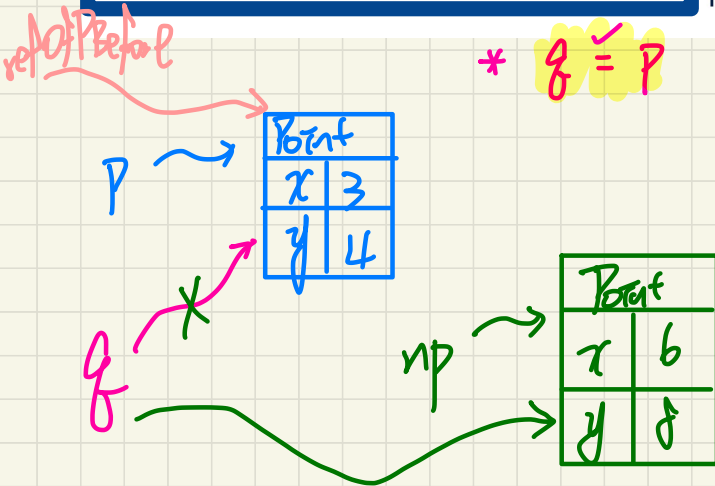
***Modelling Diagram of Aggregation  
Catch-or-Specify Requirement  
Short-Circuit Evaluation***

# 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 nos;
 void register(Course c) { --- } }
```

```
class SMS {
 Student[] ss;
 void registerAll(Course c) {
 for(int i=0; i<nos; i++) {
 ss[i].register(c);
 }
 }
}
```

callee: Student.register  
caller: SMS.registerAll

Student  
ss[i].register(c)

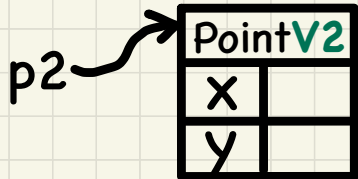
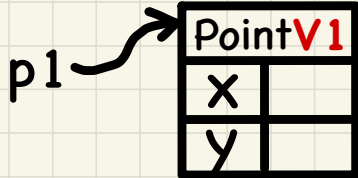
# Testing Equality of Points in JUnit: Default vs. Overridden

```
@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. */
 /* assertEquals(p1, p2); assertEquals(p2, p1); */
 /* assertEquals can take objects of different types and fail. */
 /* assertEquals(p1, p2); */ /* compiles, but fails */
 /* assertEquals(p2, p1); */ /* compiles, but fails */
 /* version of equals from Object is called */
 assertEquals(p1, p2);
 /* version of equals from PointV2 is called */
 assertEquals(p2, p1);
}
```

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

*assertEquals(p1, p2)*

—  $\overset{\checkmark}{=}$  —  
↓  
LHS and RHS  
compatible types.  
→ "p1 == p2"



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;
 }
}
```



(1)

$$\begin{aligned} & (obj1 == obj2) \\ == & \\ & (obj2 == obj1) \end{aligned}$$

true.

(2)

p1.equals(p2)

version depends  
on DT  
of p1

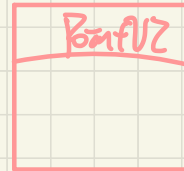
p2.equals(p1)

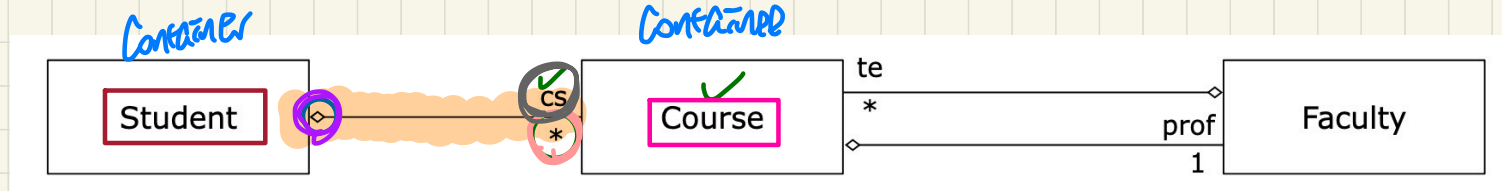
version depends  
on DT  
of p2

p1 →



p2 ~





**Student**, by aggregation, has  
zero or more **Course** objects  
as their **CS**

≥ aggregation relations.

```

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;
}

```

residing/context  
class  
(PersonCollector)

Person[]

Person

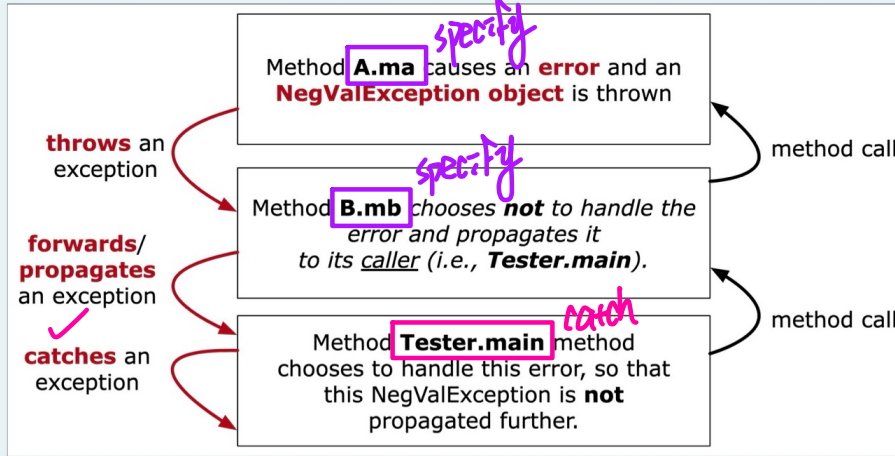
Person[]

Person

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`:



catch-or-specify  
↓  
throws - - -

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:  $0 \leq i$  &&  $i < ns.length$  &&  $ns[i] \% 2 == 1$

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

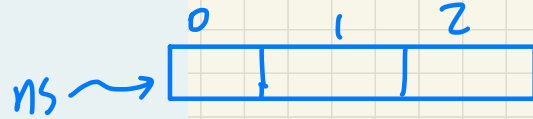
Consider the following fragment of code:

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

$ns.length == 3$



Correct order:  $0 \leq i$  &&  $i < ns.length$  &&  $ns[i] \% 2 == 1$

$0 \leq i$        $i < ns.length$

$0 \leq i$  &&  $ns[i] \% 2 == 1$  &&  $i < ns.length$

any value of  $i$  that's negative will

guard cond. meant for "too-large"  $i$  value but this gc.

cause this exp. to eval to false, and the short-circuit evaluation will skip the rest

## 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

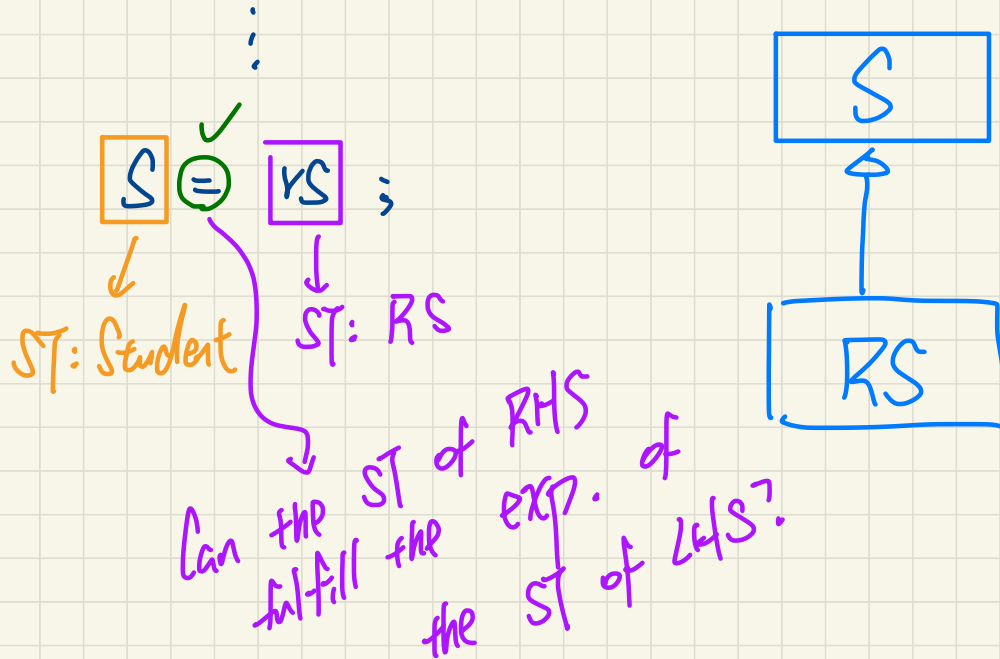
Student

$S = \dots$

Resident Student

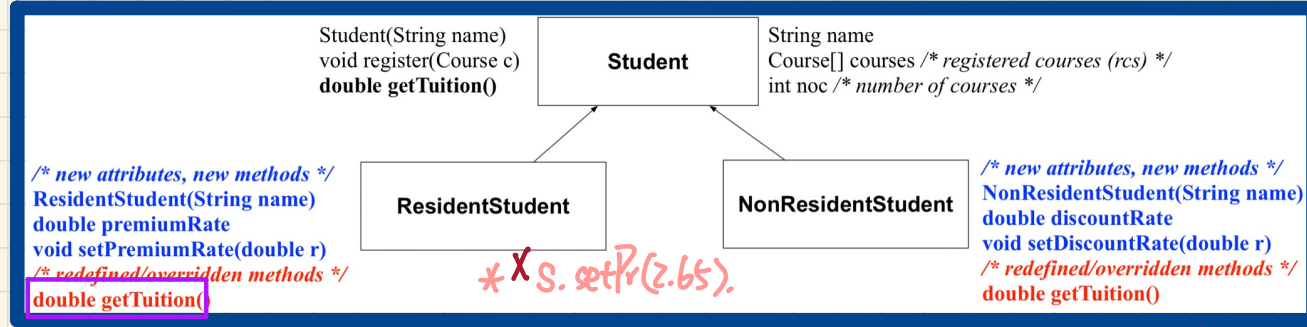
$rs = \dots$

$rs^x = S;$





# Intuition: Dynamic Binding

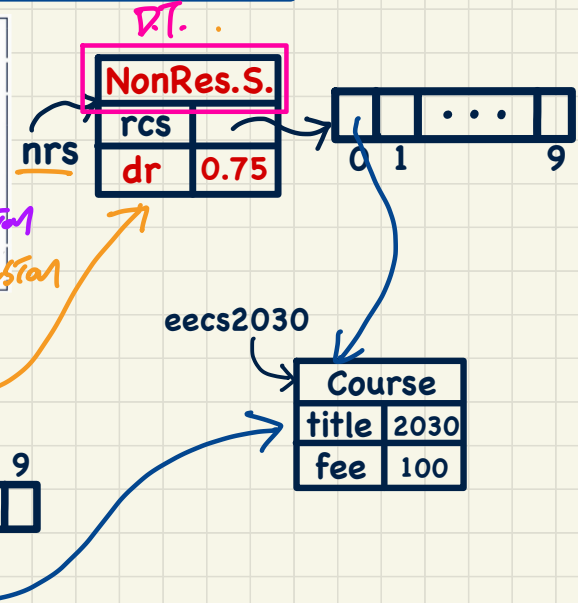


```

1 Course eeecs2030 = 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(eeecs2030);
6 nrs.setDiscountRate(0.75); nrs.register(eeecs2030);
7 s = rs; System.out.println(s.getTuition());
8 s = nrs; System.out.println(s.getTuition());

```

Handwritten annotations on the code:  
 - Line 2: **Student s;** is circled in green, with "DT." written next to it.  
 - Line 3: **ResidentStudent** is circled in blue, with "DT." written next to it.  
 - Line 4: **NonResidentStudent** is circled in pink, with "DT." written next to it.  
 - Line 7: **s = rs;** is circled in blue, with "DT: RS" written next to it.  
 - Line 8: **s = nrs;** is circled in orange, with "DT: NRS" written next to it.  
 - Line 7: **s.getTuition();** is circled in blue, with "→ invoke RS version" written next to it.  
 - Line 8: **s.getTuition();** is circled in orange, with "→ invoke NRS version" written next to it.



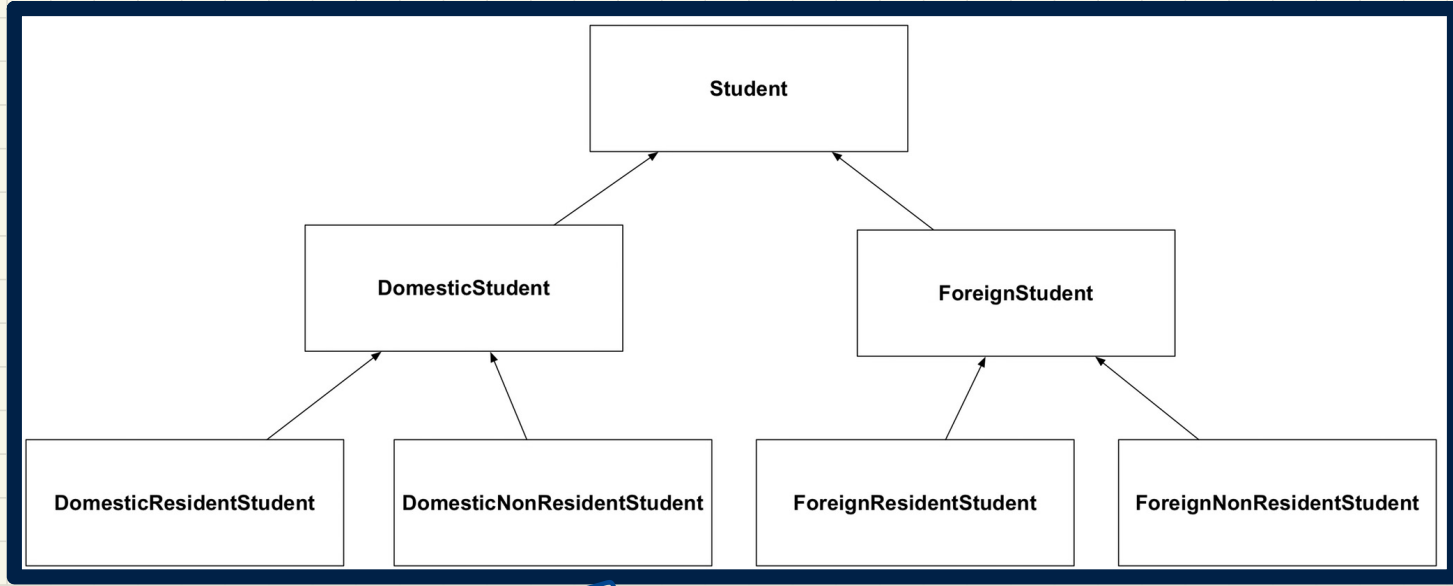
| ST of s | DT of s |
|---------|---------|
| ①       | null    |
| ②       | RS      |
| ③       | RS      |
| ④       | NRS     |

Handwritten word "Student" is written vertically next to the table.

Compilation : Static types

runtime : dynamic types

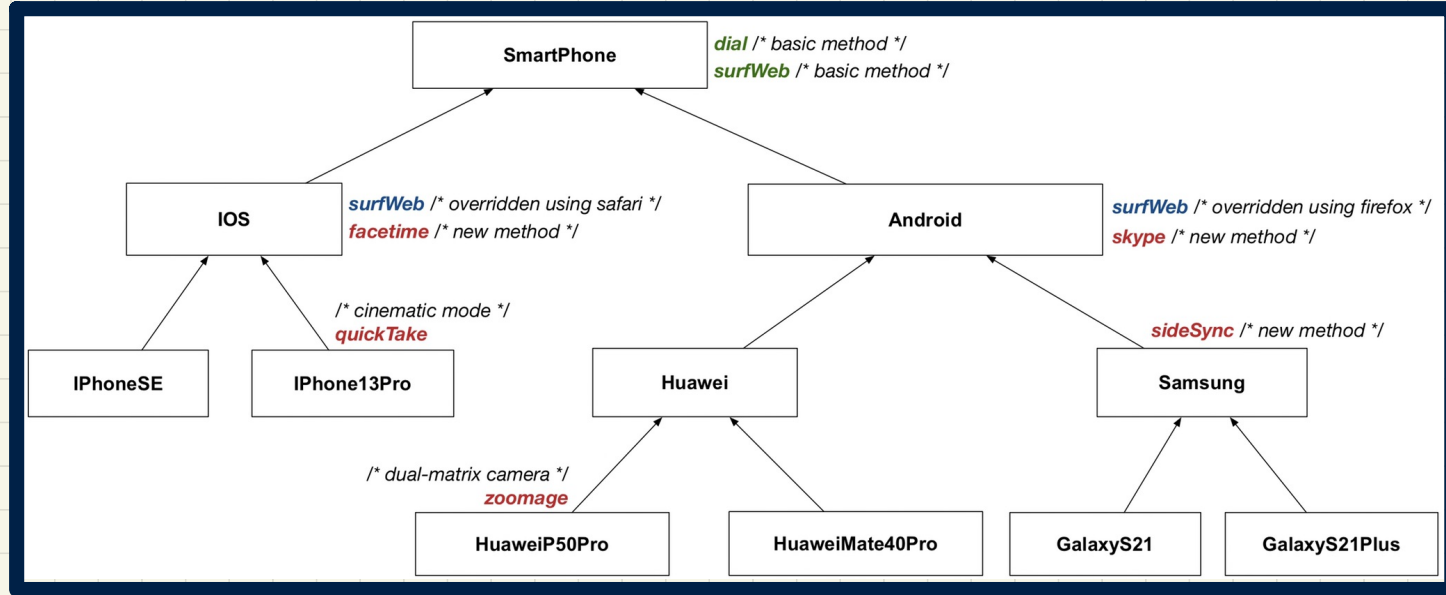
# Multi-Level Inheritance Hierarchy: Students



## Reflections:

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

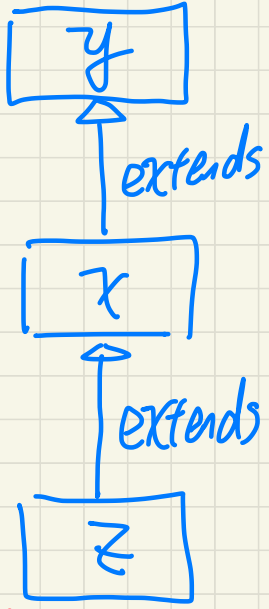
# Multi-Level Inheritance Hierarchy: Smartphones



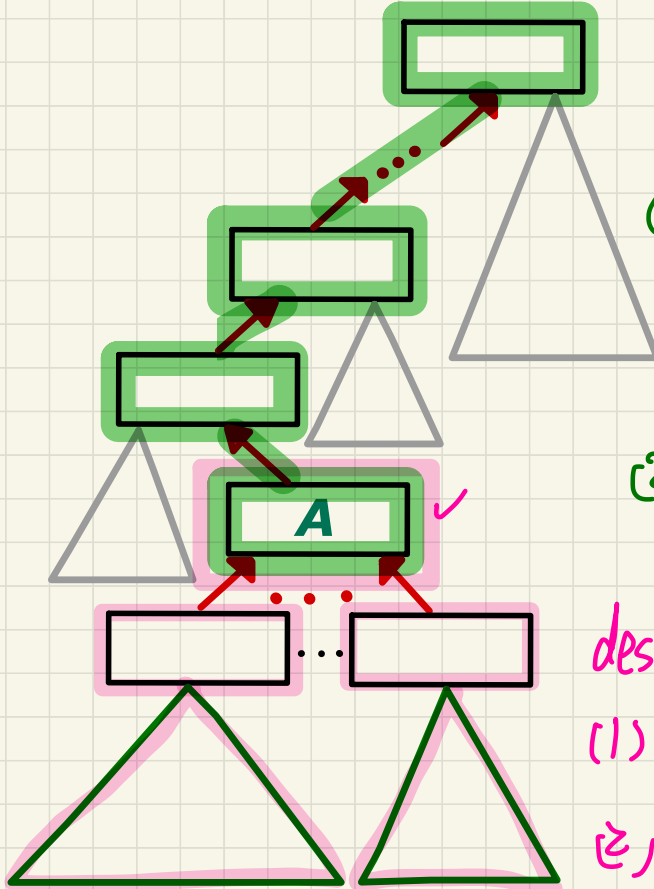
## Reflections:

- 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



$\downarrow$   
 $\mathbb{Z}$  indirectly extends  $\mathbb{X}$



ancestors of A:

(1) A **accumulates** the **attrs/methods** from its ancestors. *exp*

2,  $\exp(A) \geq$   
 $\exp$  of any ancestor

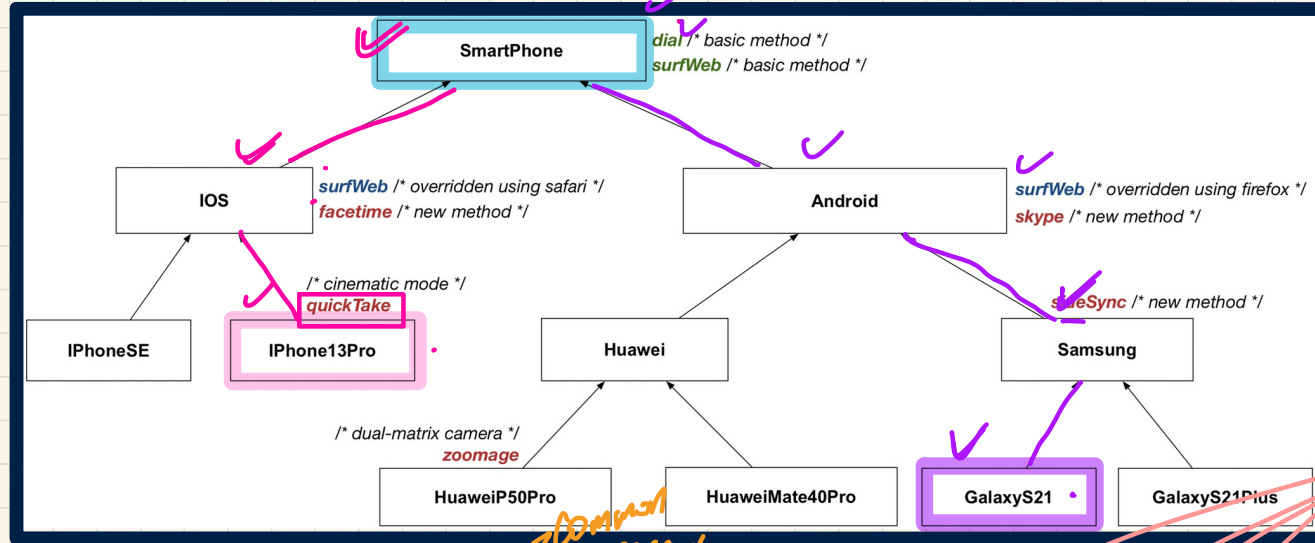
descendants of A:

(1)  $\exp(A)$  is inherited to all descendants

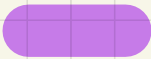
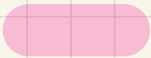
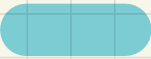
ii)  $\exp(A) \leq \exp$  of any descendant

# Inheritance Accumulates Code for Reuse

Common Ancestor



exp of  
Common  
Ancestor.

|                                                                                    | ancestors        | expectations                       | descendants |
|------------------------------------------------------------------------------------|------------------|------------------------------------|-------------|
|   | GSZl, SS, An, SP | side Sync, surfWeb, skype, dial    | GSZl        |
|   | IPBPro, IOS, SP  | quickTake, surfWeb, facetime, dial | IP13Pro     |
|  | SP               | dial, surfWeb                      | all classes |

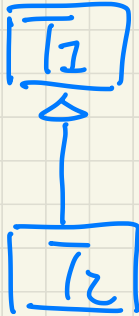
$T_1$  col = ...  
 $T_2$  col = ...

col.   

exp of  
col's ST

col.   

exp of  
col's ST

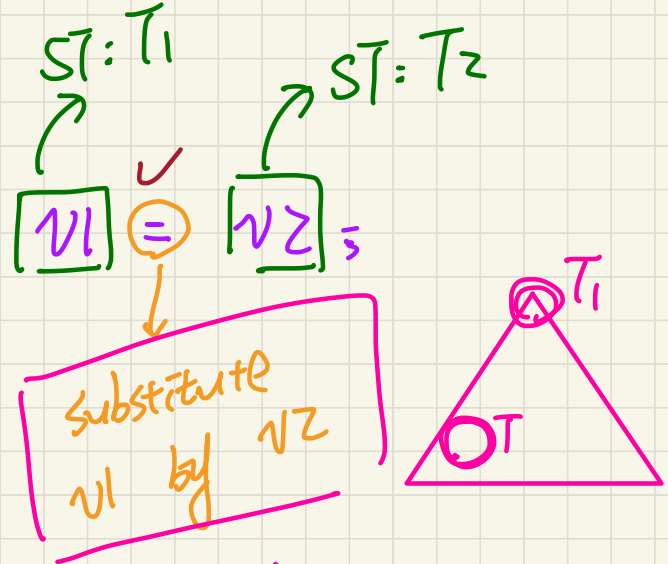
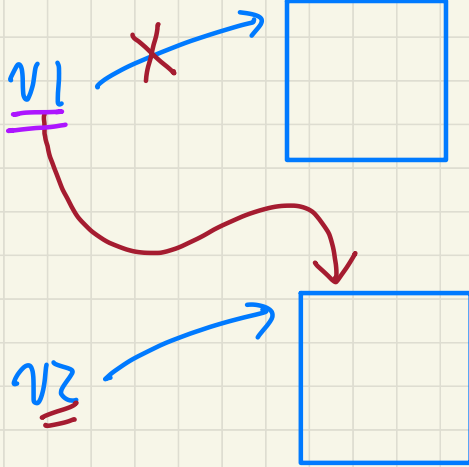


wider range of  
exp. (attrs/methods callable)  
-  $T_2$  is a descendant class  
of  $T_1$

$T_1 \quad v_1 = \dots$

$T_2 \quad v_2 = \dots$

$v_1.x$

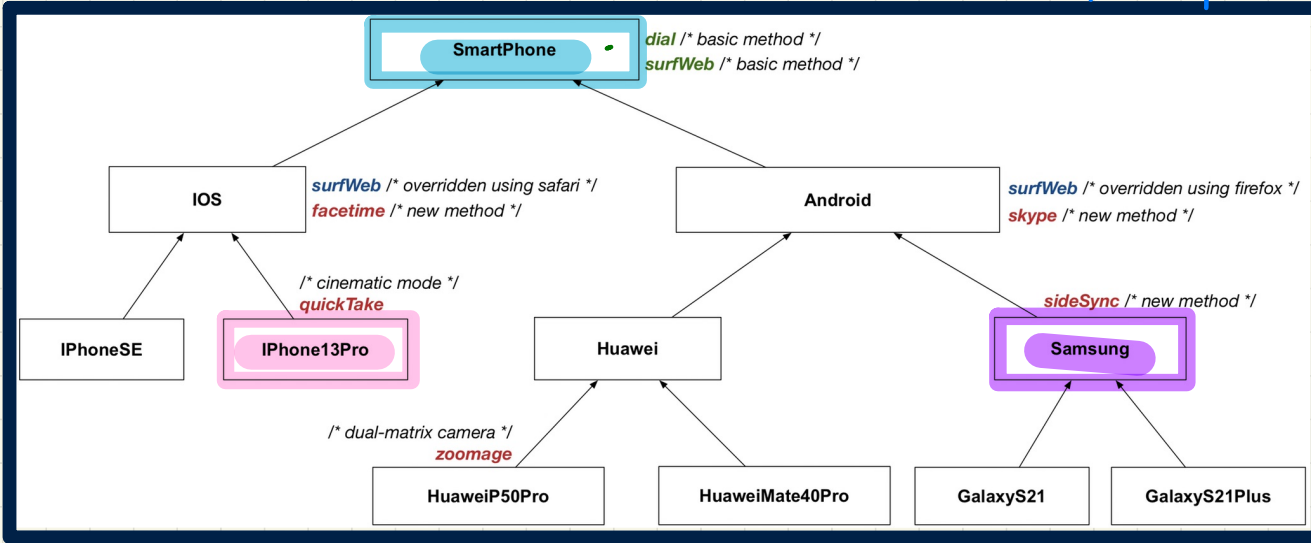


- ↓ valid if  $v_2$ 's ST  $T_2$
- (1) can fulfill exp of  $v_1$ 's ST  $T_1$
  - (2)  $v_2$ 's ST should be a dependent of  $v_1$ 's ST



# Inheritance Accumulates Code for Reuse

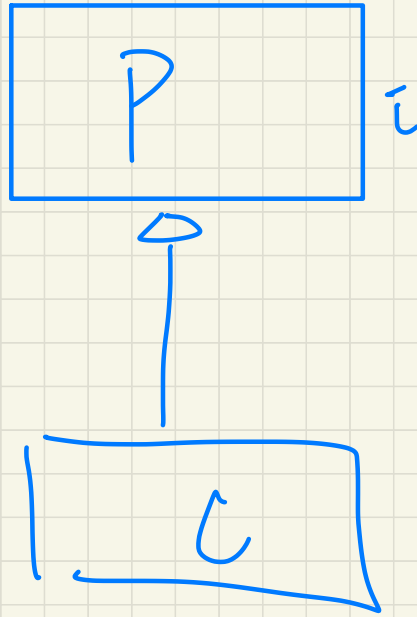
①  $sp1 = sp2$ ;  $sp2 = sp3$ ;   
 compiler's ST   
 IP13Pro can fulfill the exp of  $sp1$ 's ST   
 SmartPhone



**SmartPhone**  $sp1$ ;  
**iPhone13Pro**  $sp2$ ;  
**Samsung**  $sp3$ ;

✓  
 $sp1 = ?$ ;  
 $sp2 = ?$ ;  
 $sp3 = ?$ ;

②  $sp2 = sp1$ ;  
 $sp2 = sp3$ ; fail to compile  
 'ST of  $sp1$  cannot fulfill the exp of ST of  $sp2$



P objP = ...

C objC = ...

objP = ✓ objC

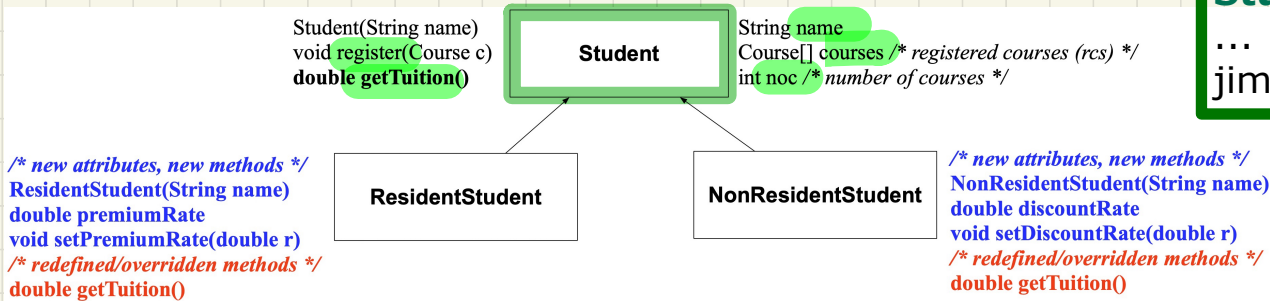
objC X objP



P is not a subclass of C.

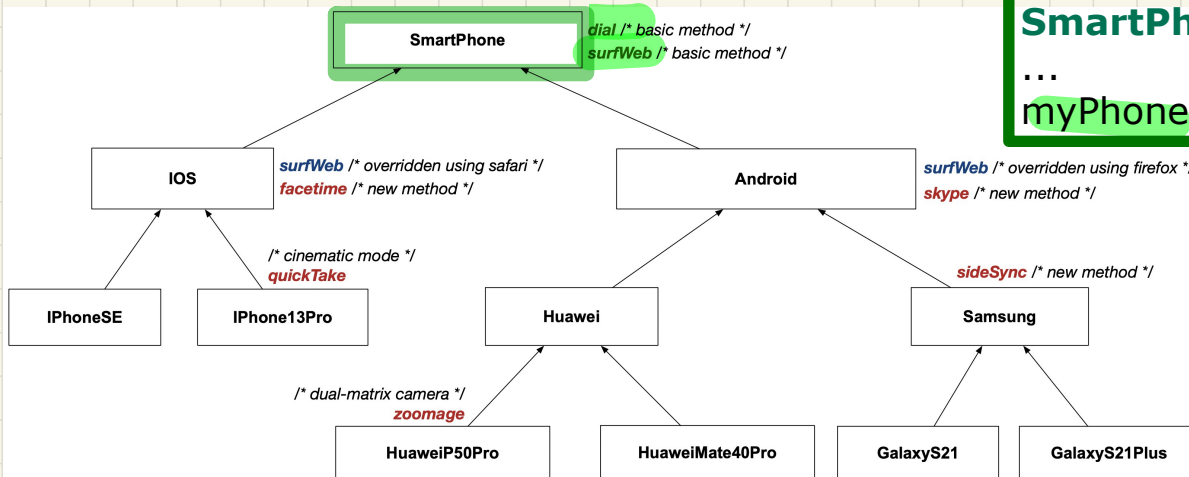
# Static Types determine Expectations

## Inheritance Hierarchy: Students



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

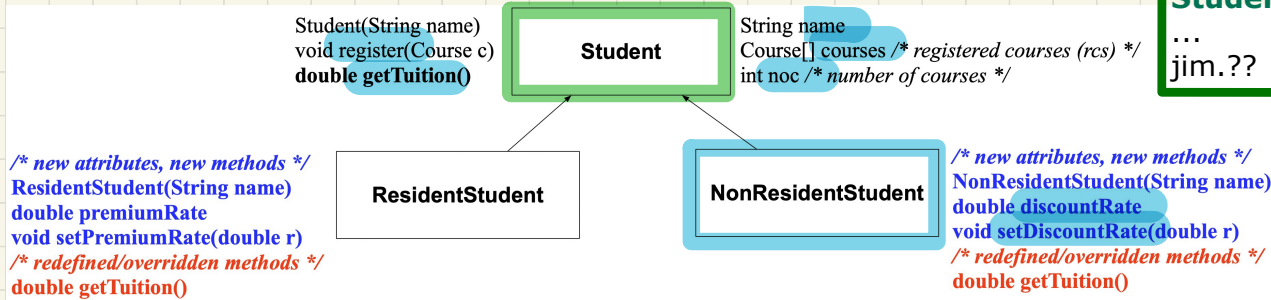
## Inheritance Hierarchy: Smart Phones



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

# Static Types determine Expectations

## Inheritance Hierarchy: Students

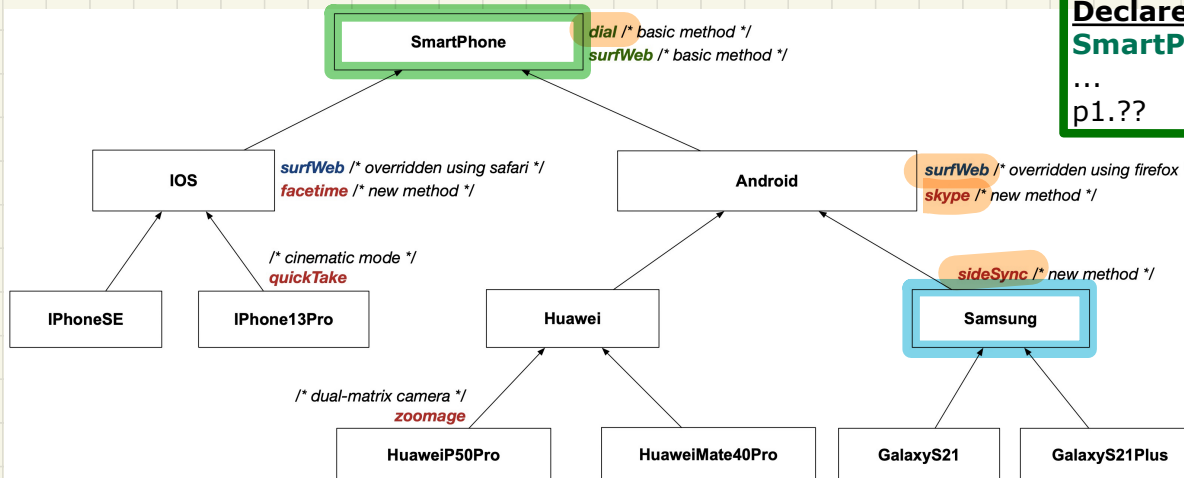


**Declare:**  
**Student** jim;  
...  
jim.??

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

exp(Student)  
+ dR +  
setDr.

## Inheritance Hierarchy: Smart Phones



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

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

## 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

ST of  $oa$

```

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

```

neither ancestors  
nor descendants  
of  $A \Rightarrow$  ambiguous  
expectations

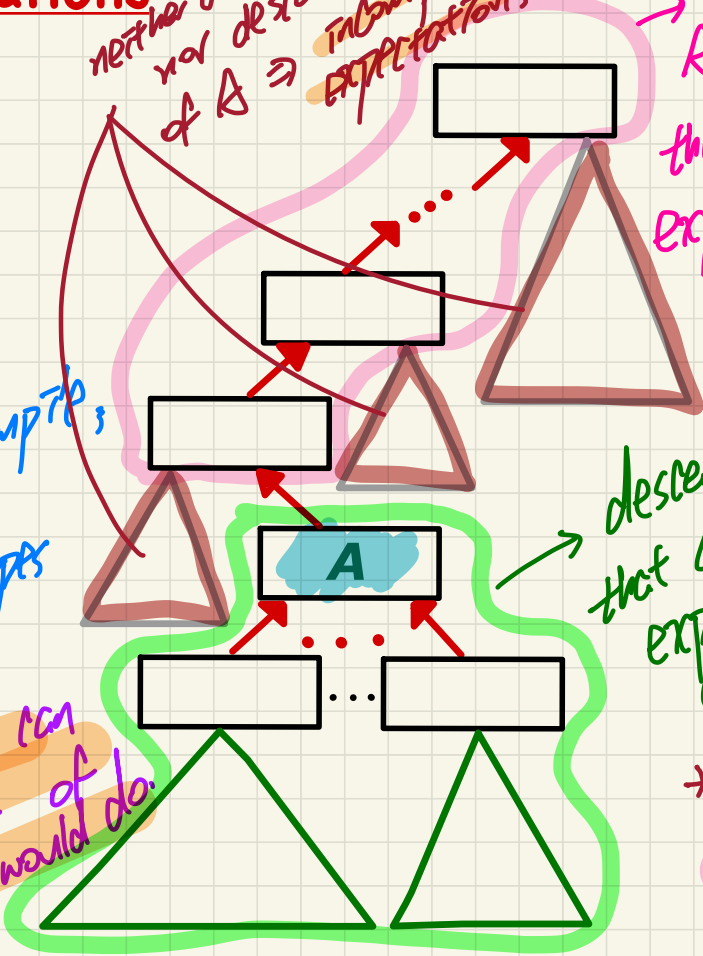
$\rightarrow$  ancestors of  
 $A$ 's parent  
they cannot fulfill  
exp. of  $A$  serve as the  
ST of  $ob$

in order for this  
substitution to compute;  
what can be the  
range of static types

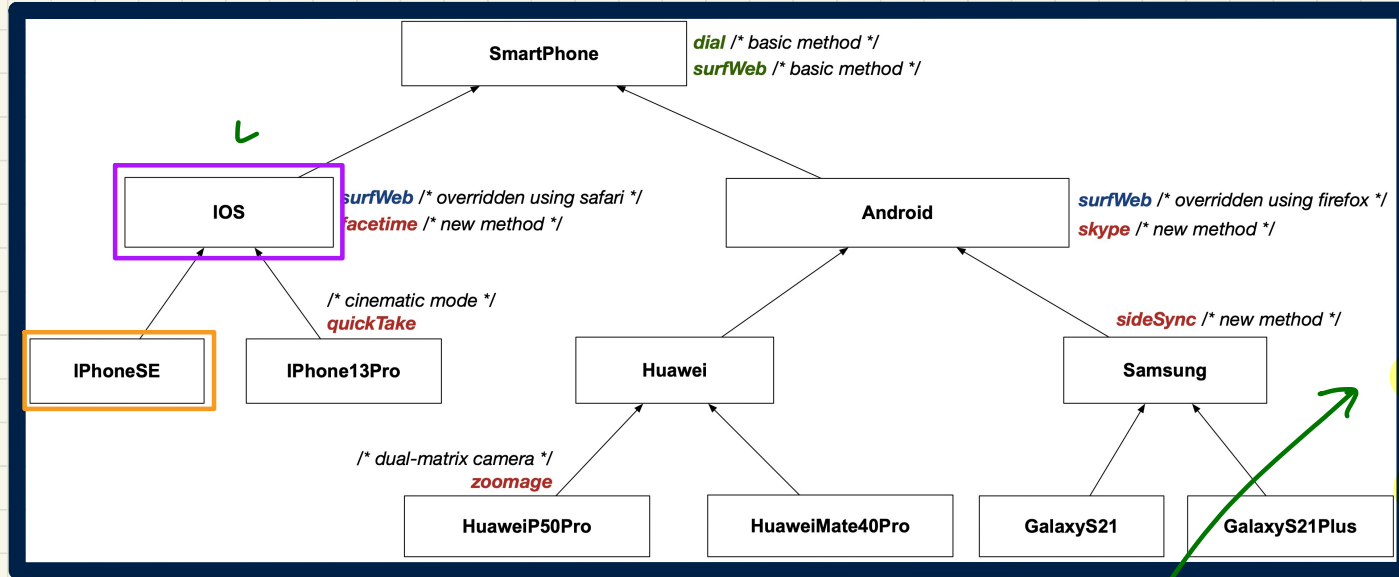
$\hookrightarrow$  \* any class that can  
fulfill the exp. of  
 $OA$ 's ST would do.

descendants of  $A$   
that can fulfill the  
exp. of  $A$  can serve  
as the ST of  $ob$ )

\* that is, any  
descendant class  
of  $A$



# Rules of Substitutions (1)



## Declarations:

**IOS** sp1;  
**iPhoneSE** sp2;  
**iPhone13Pro** sp3;

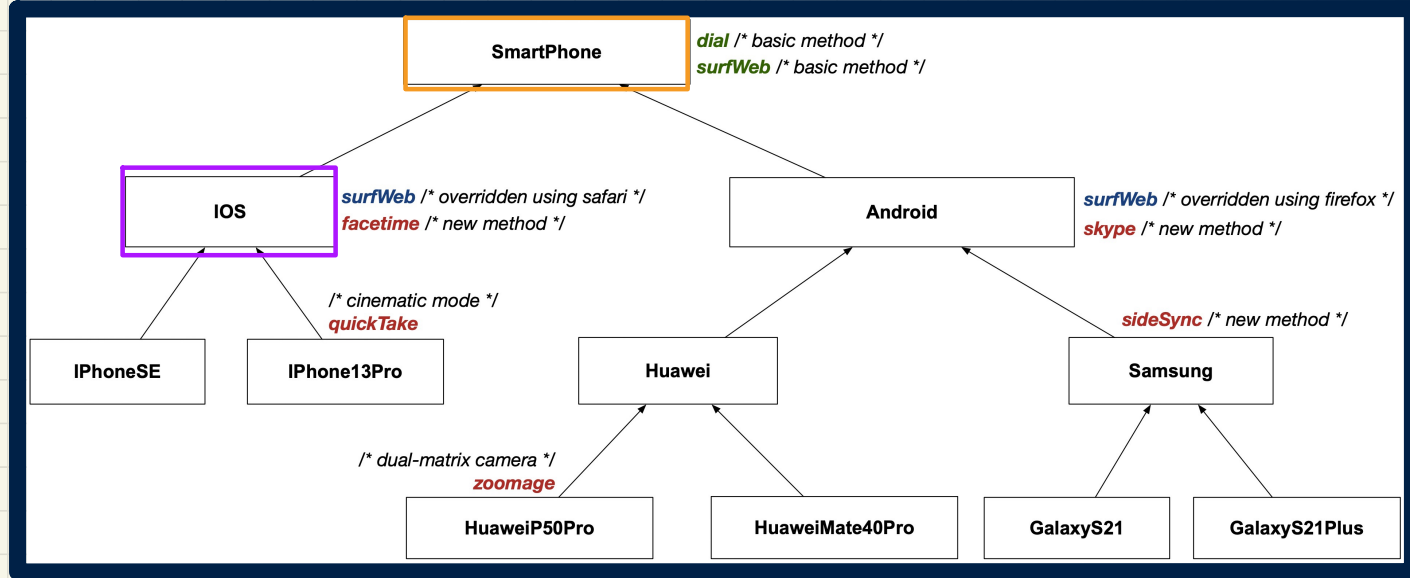
## Substitutions:

sp1 = sp2;  
sp1 = sp3;

Compiler  
: sp2's  
ST IPSE  
is a  
descendant  
of sp1's  
ST IOS



## Rules of Substitutions (2)



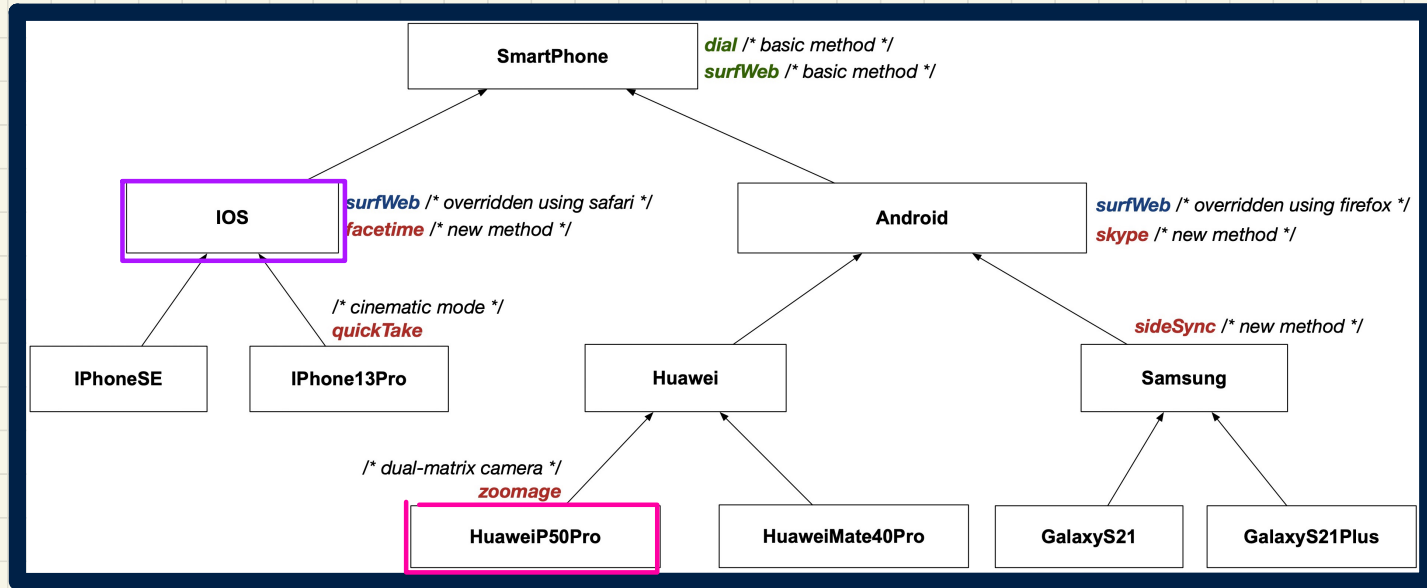
### Declarations:

**IOS** sp1;  
**SmartPhone** sp2;

### Substitutions:

sp1  $\neq$  sp2;

# Rules of Substitutions (3)



## Declarations:

**IOS** sp1;

**HuaweiP50Pro** sp2;

## Substitutions:

**sp1** <sup>x</sup> = **sp2**;

ST of sp1 (IOS)

ST of sp2 (HuaweiP50Pro) is neither an ancestor nor descendant of

# Visualization: Static Type vs. Dynamic Type

Declaration:

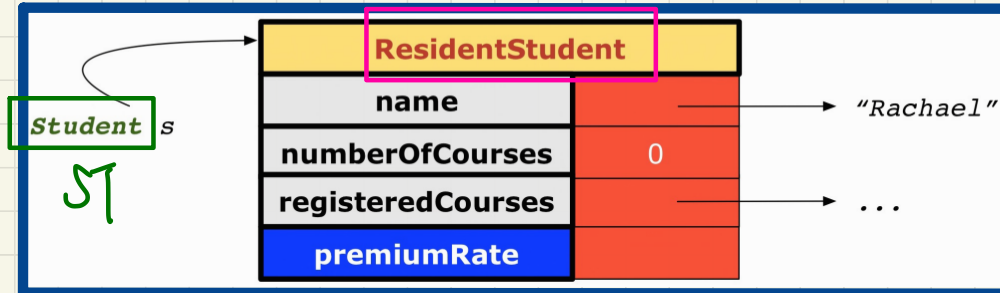
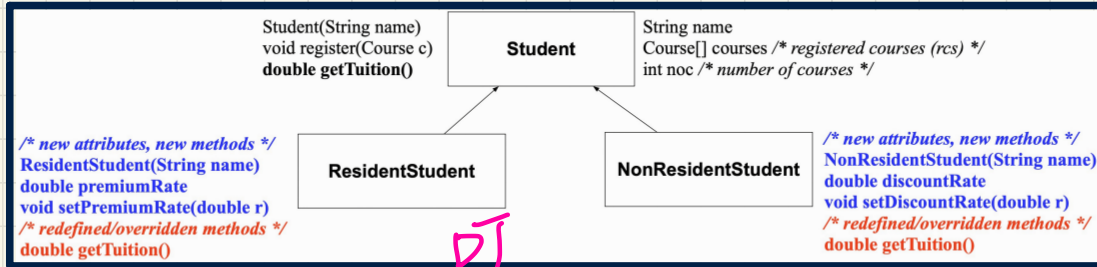
**Student** s;

Substitution:

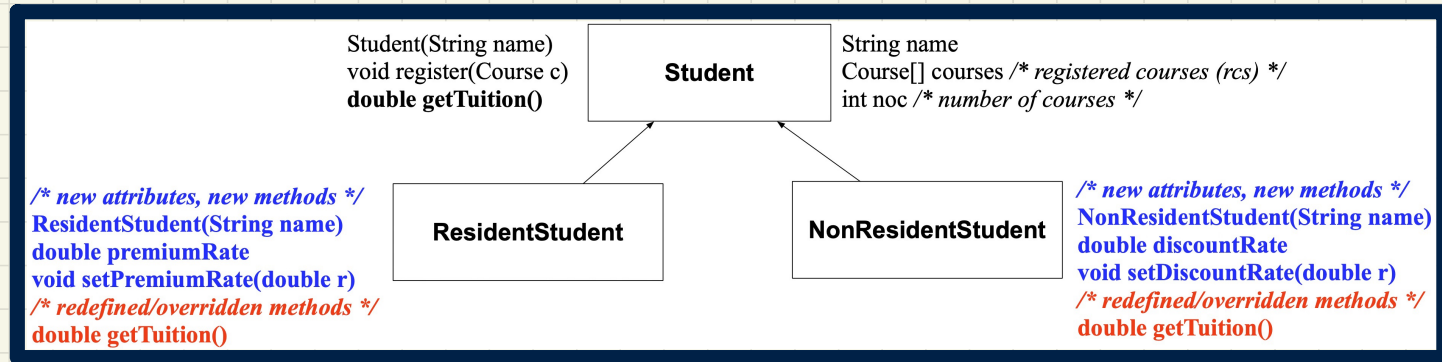
s = **new ResidentStudent**("Rachael");

Static Type: Expectation

Dynamic Type: Accumulation of Code



# Change of Dynamic Type (1.1)



## Example 1:

Student jim = new ResidentStudent(...);

jim = new NonResidentStudent(...);

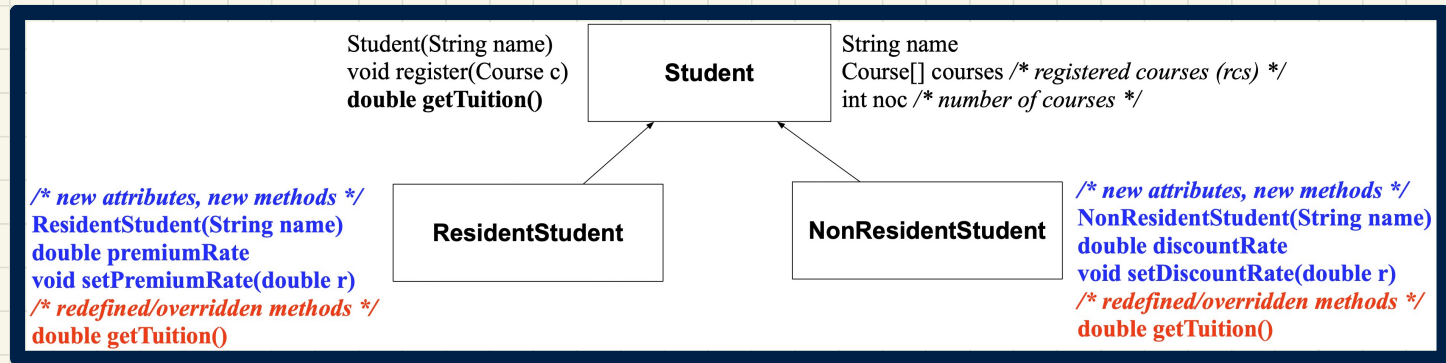
compiles 'c RS can fulfill the exp

of jim's ST (Student)  
(RS is a descendant of Student)

DT of jim: RS

DT of jim: NRS

## Change of Dynamic Type (1.2)

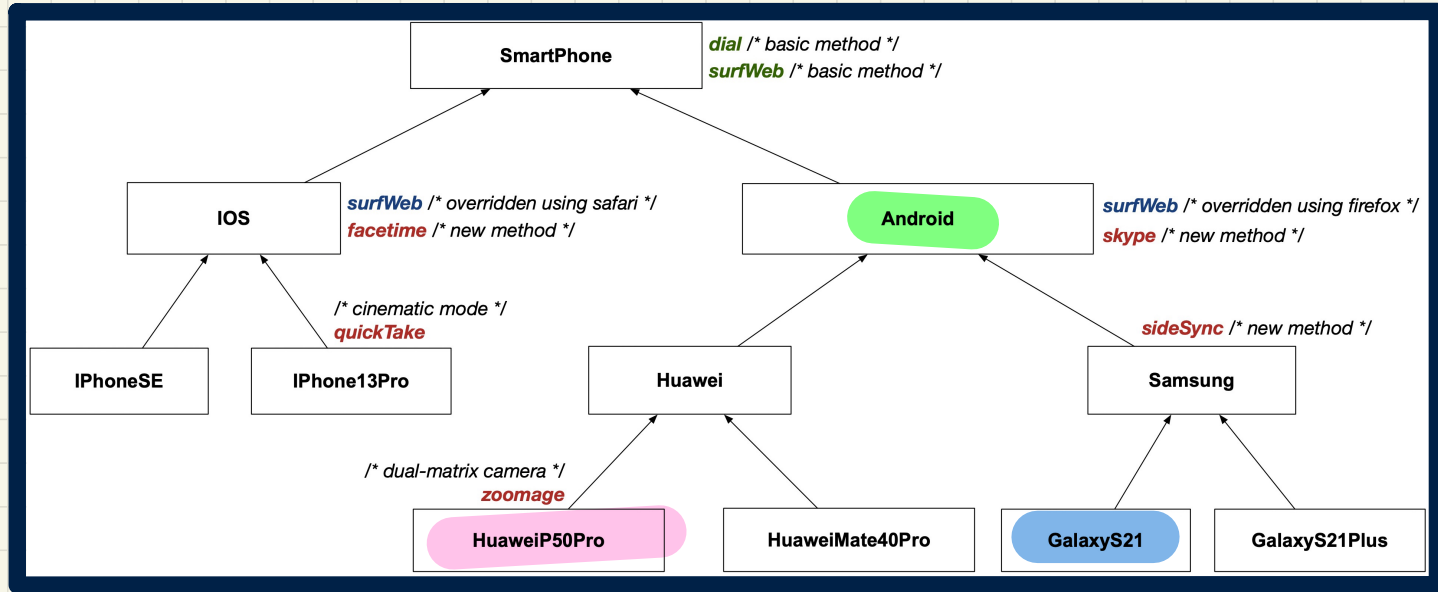


### Example 2:

ResidentStudent jeremy = new Student(...);

∴ Student is not a descendant of ST of jeremy (RS).

# Change of Dynamic Type: Exercise (1)



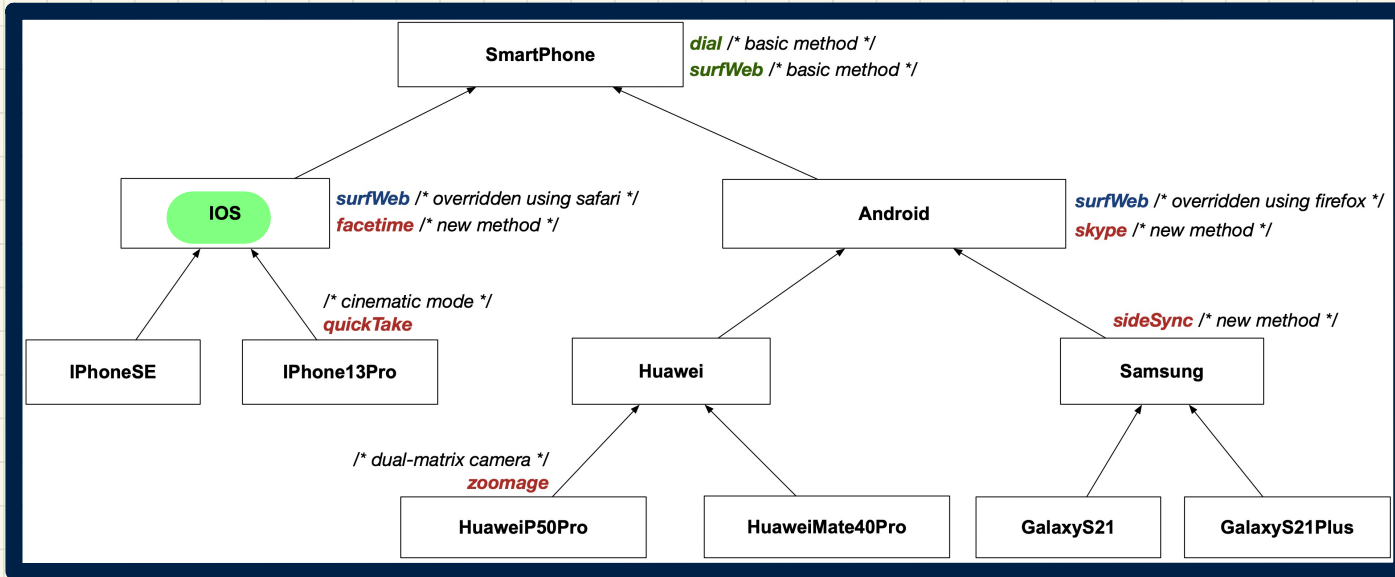
## Exercise 1:

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

DT: HP50Pro

DT: GS21

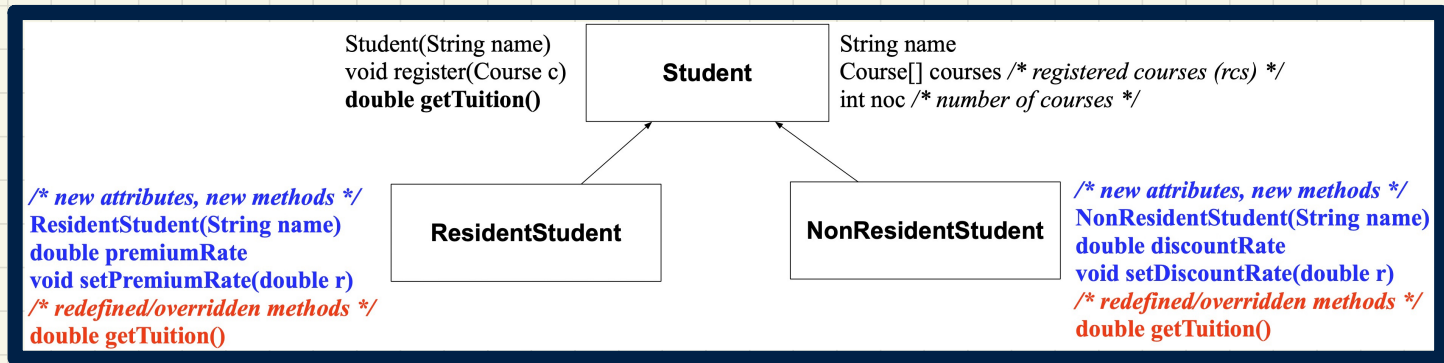
# Change of Dynamic Type: Exercise (2)



## Exercise 2:

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

# Change of **Dynamic** Type (2.1)



## Given:

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

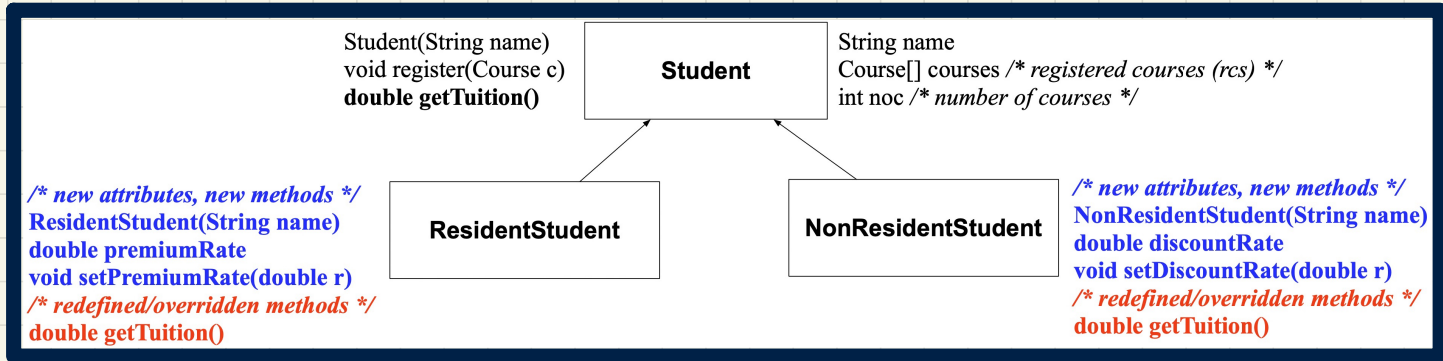
## Example 1:

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

DT: RS → version from RS (PR)  
DT: NRS → version from NRS (dr).



## 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**.

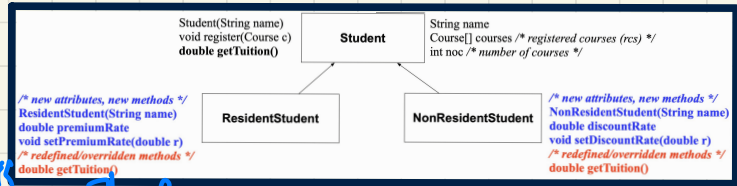
⇒ Each **dynamic type** has its version of method.

## Dynamic Binding:

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

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

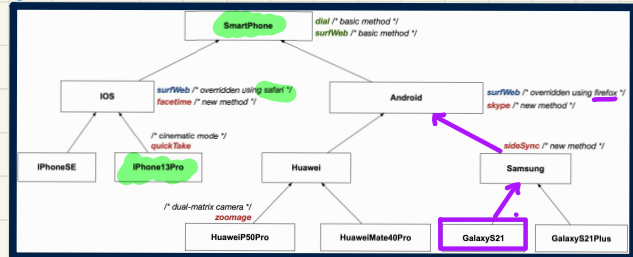
polymorphism:  
spl can have multiple possible DTs: descendants of SmartPhone



```
SmartPhone sp1 = new iPhone13Pro(...);
SmartPhone sp2 = new GalaxyS21(...);
```

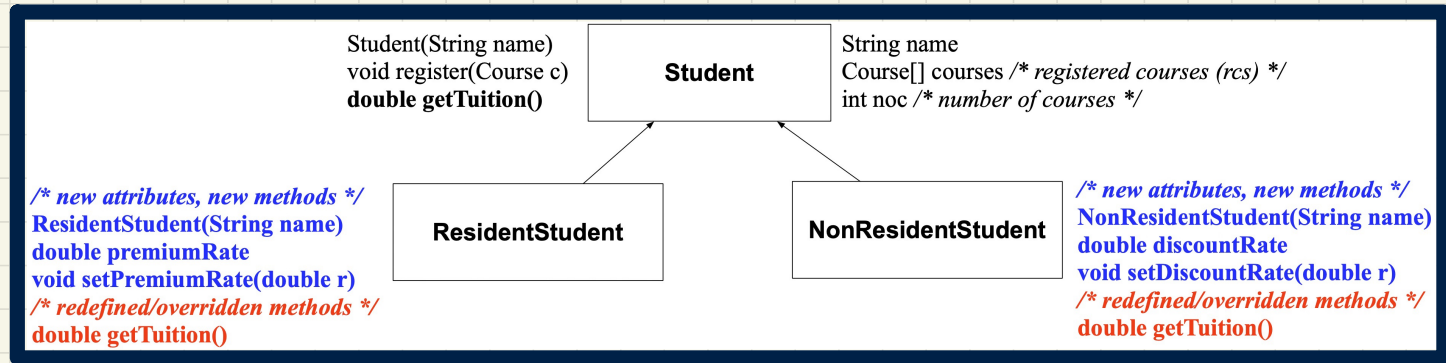
```
sp1.surfWeb();
sp1 = sp2;
sp1.surfWeb();
```

changes spl's DT from IP13Pro to GS21

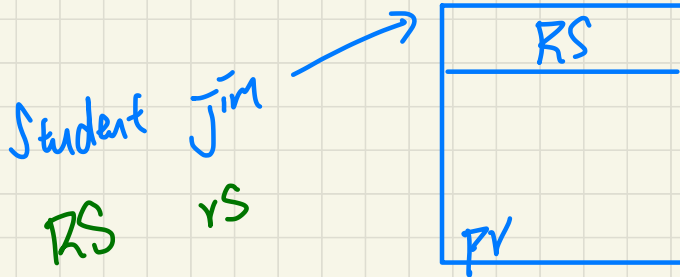


DT: GS21

# Type Cast: Motivation

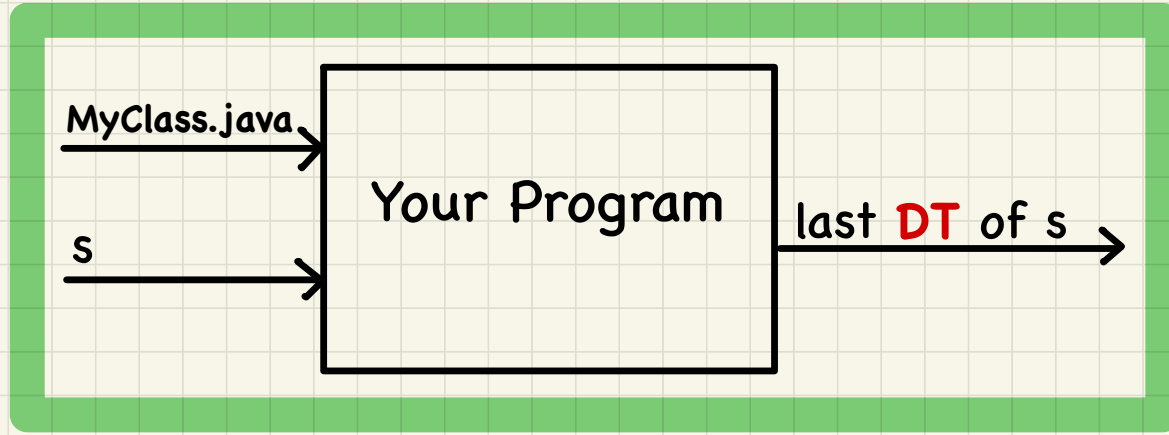


```
1 Student jim = new ResidentStudent("J. Davis");
2 ResidentStudent rs = jim;
3 rs.setPremiumRate(1.5);
```



# An **A+** Challenge: Inferring the **DT** of a Variable

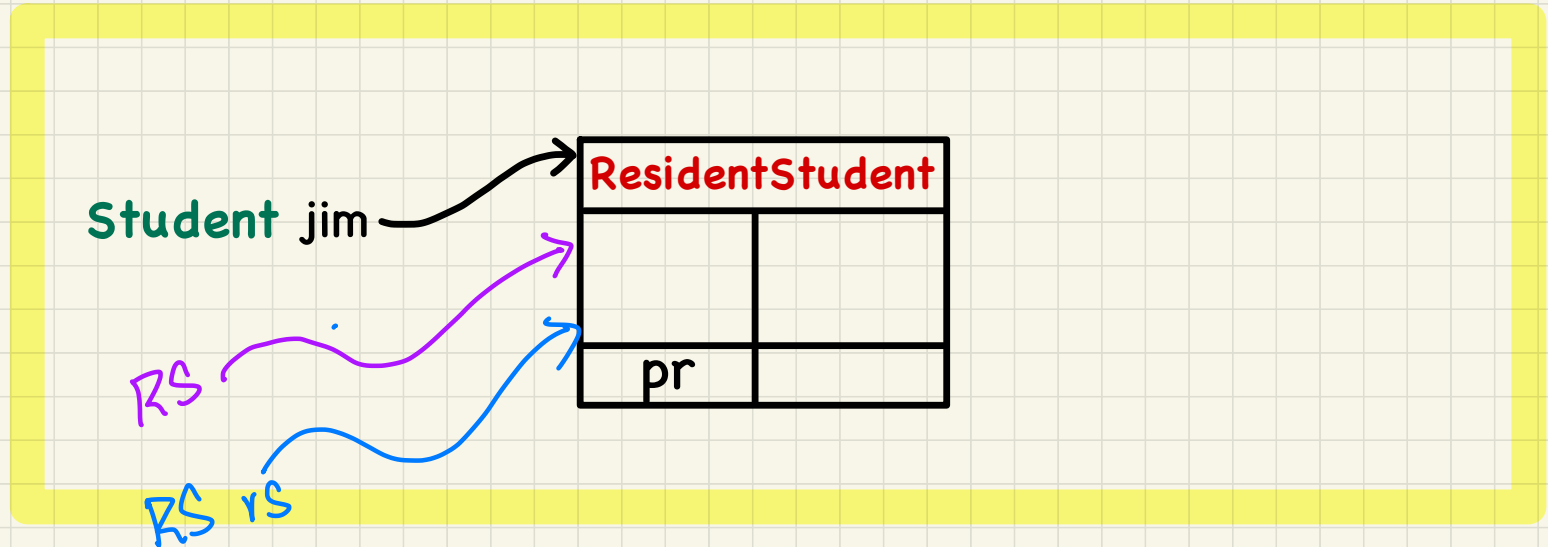
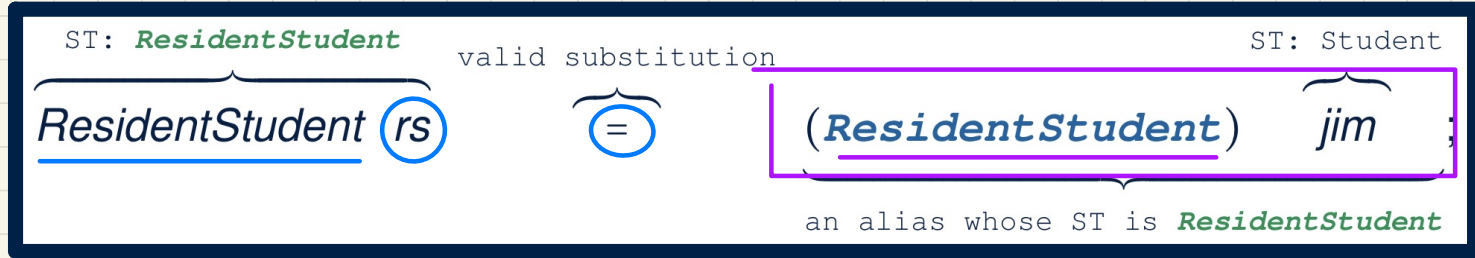
*undetectable*



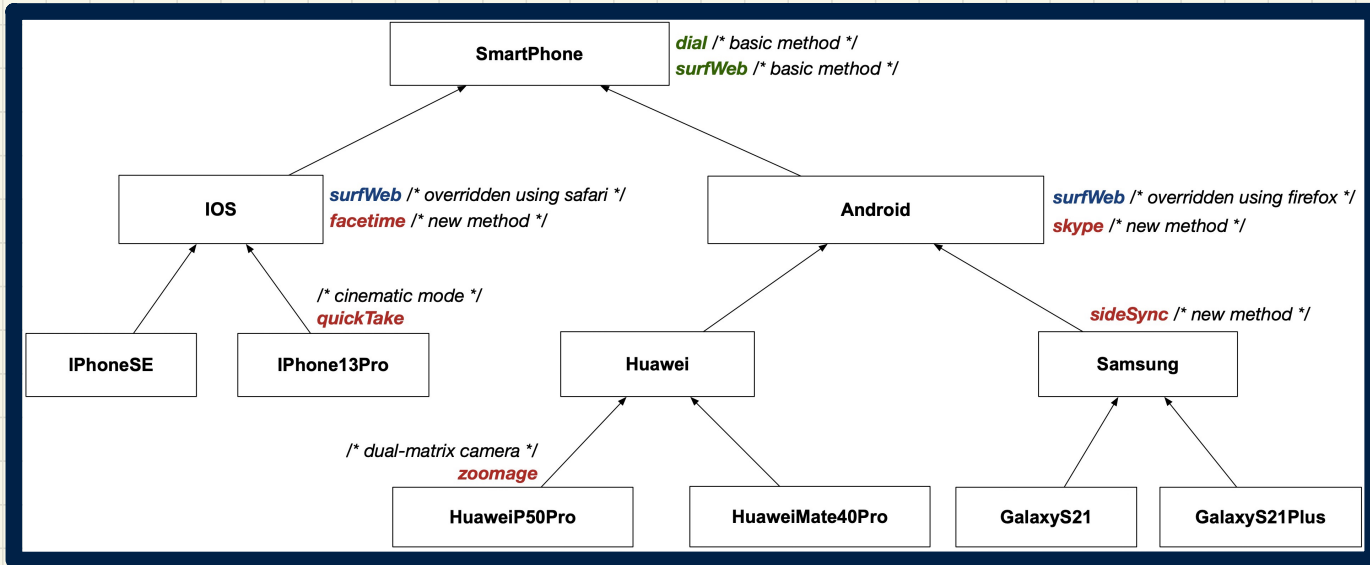
```
class MyClass {
 main (...)
 Student s = ...;
 ...
 s = new ResidentStudent(...);
 }
}
```

# Anatomy of a Type Cast

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



# 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();
```

↓  
alias of ST: IP3Pro

## Exercise

```
SmartPhone aPhone = new iPhone13Pro();
(iPhone13Pro) aPhone.facetime();
```

not compilable  
! facetime not  
part of the exp  
of aPhone's  
ST (SP)

# Compilable Casts: Upwards vs. Downwards

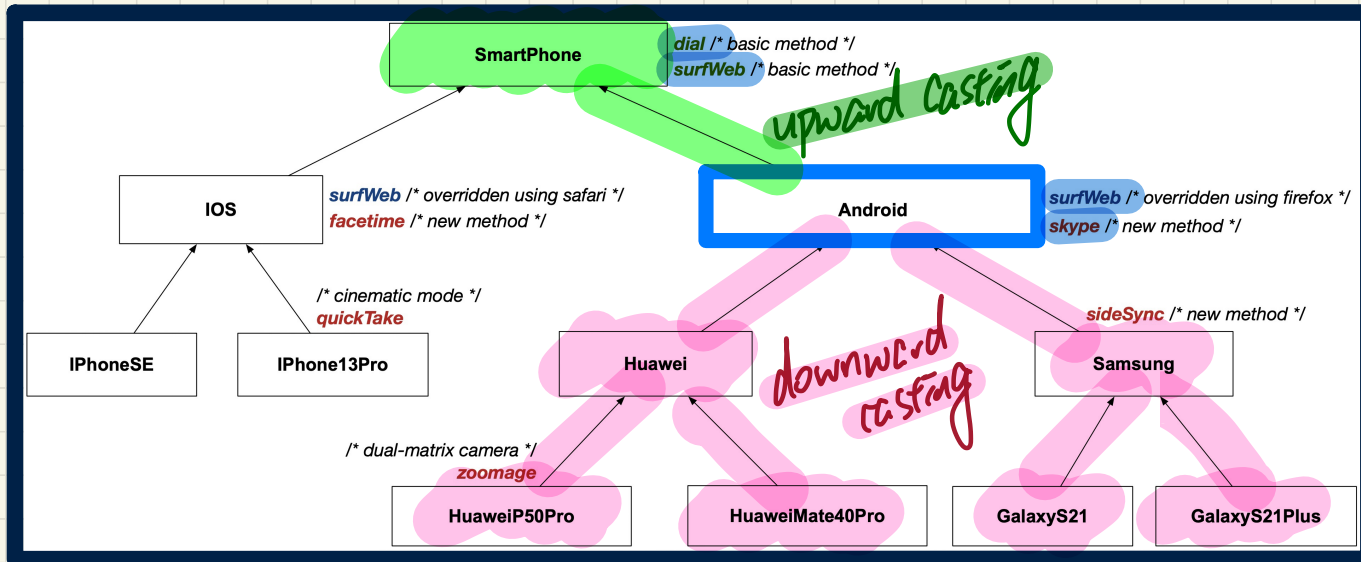
## Expectations

|           | sp | myPhone | ga |
|-----------|----|---------|----|
| dial      |    |         |    |
| surfWeb   |    |         |    |
| skype     |    |         |    |
| sideSync  |    |         |    |
| facetime  |    |         |    |
| quickTake |    |         |    |
| zoomage   |    |         |    |

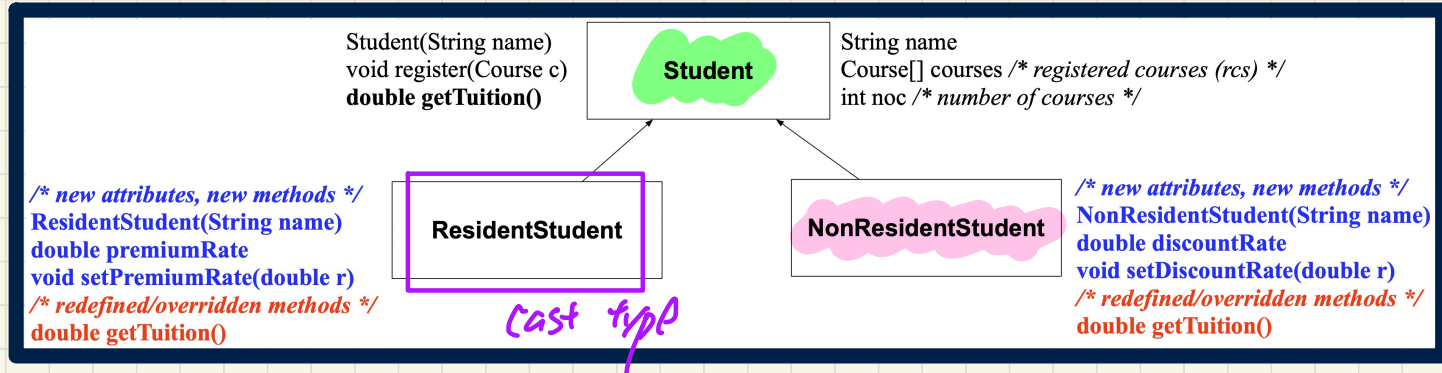
**Android** myPhone = **new GalaxyS21Plus**();

**SmartPhone** sp = (**SmartPhone**) myPhone;

**GalaxyS21Plus** ga = (**GalaxyS21Plus**) myPhone;



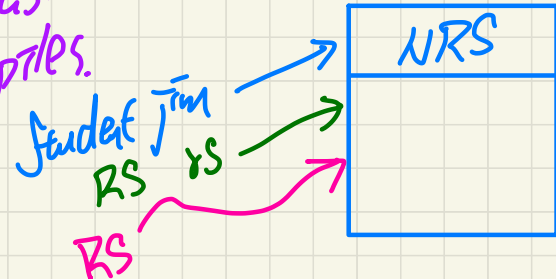
# Compilable Type Cast May Fail at Runtime (1)



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

→ at RT: *ClassNotFoundException*

2. DT NRS cannot fulfill the exp. of rs (RS) downward cast  
↳ *ClassNotFoundException* ↳ compiles.  
Runtime  
↳ rs is expected to be used as RS





## Lecture 20 - Nov. 19

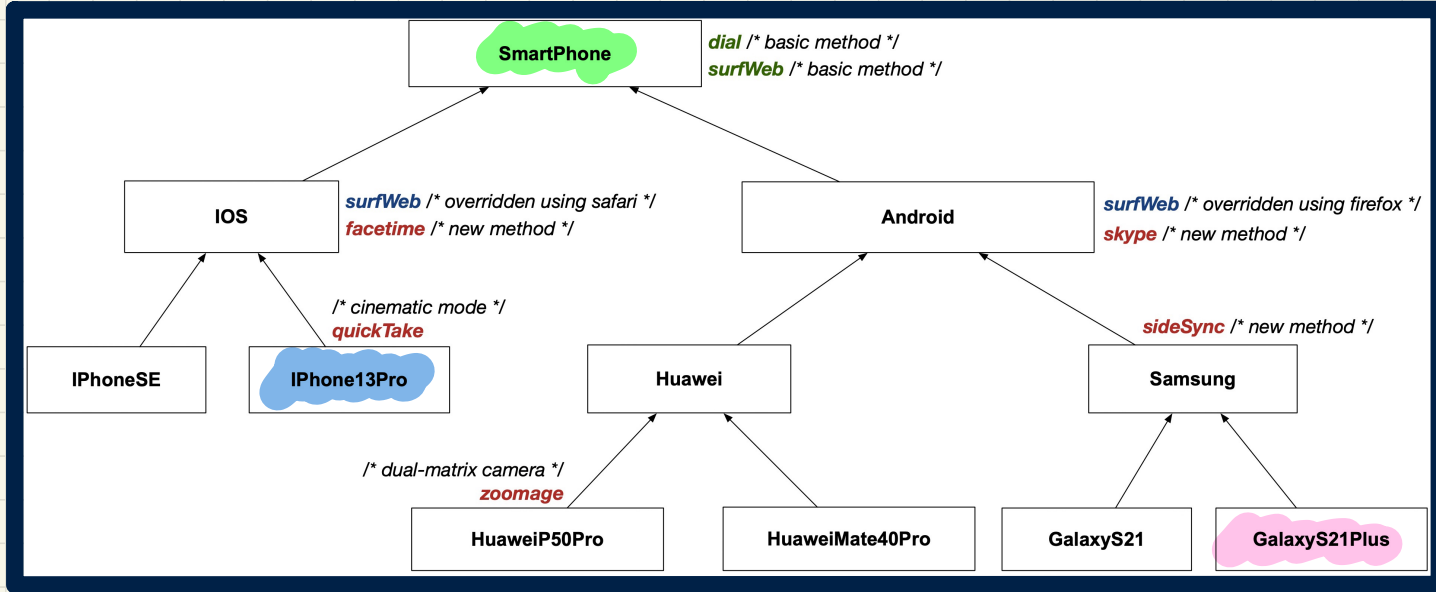
### Inheritance

***Type Casts: Exercise***  
***Checking Dynamic Types: instance of***  
***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

# Compilable Type Cast May **Fail** at Runtime (2)

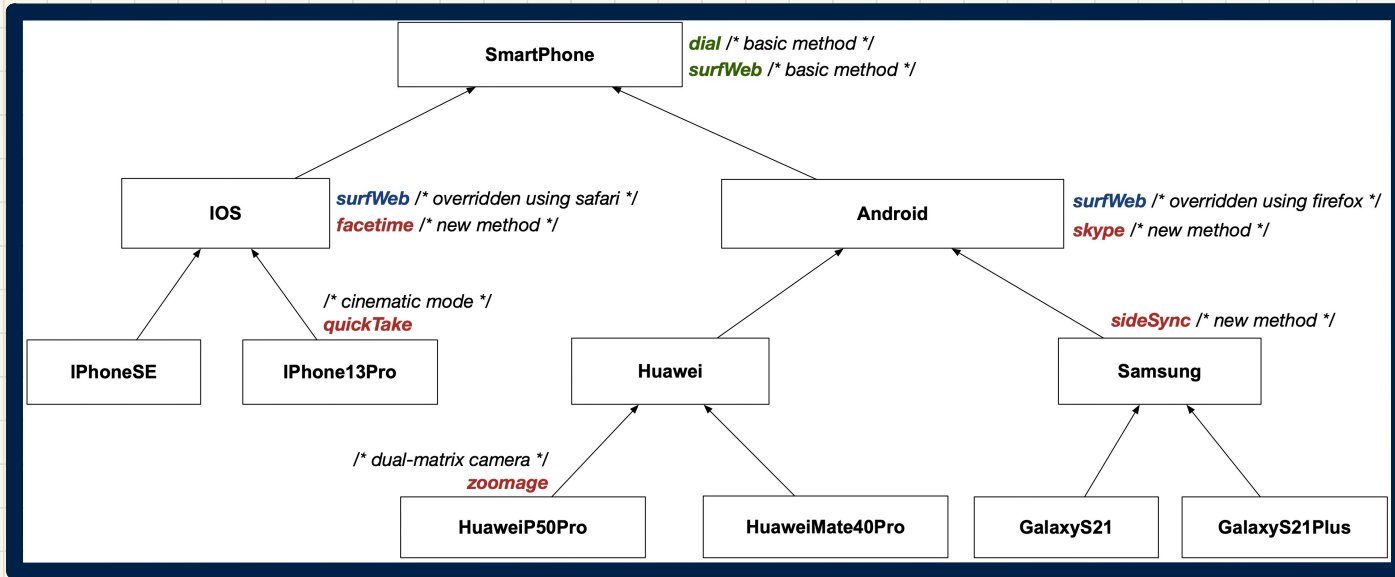


```
1 SmartPhone aPhone = new GalaxyS21Plus();
2 iPhone13Pro forHeeyeon = (iPhone13Pro) aPhone;
3 forHeeyeon.quickTake();
```

Compiles  
downward  
cast  
descendant of  
cast type iPhone13Pro

CCE: aPhone's DT (GalaxyS21Plus) is not a

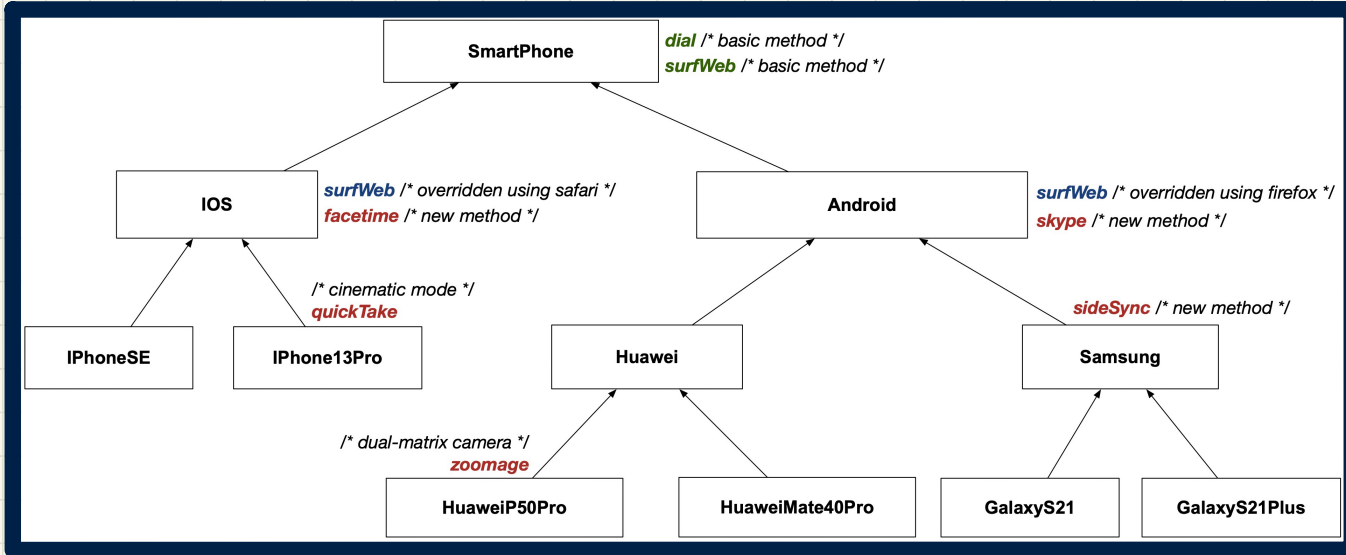
## Exercise: Compilable Type Cast? **Fail** at Runtime? (1)



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

Compilable? **ClassCastException** at runtime?

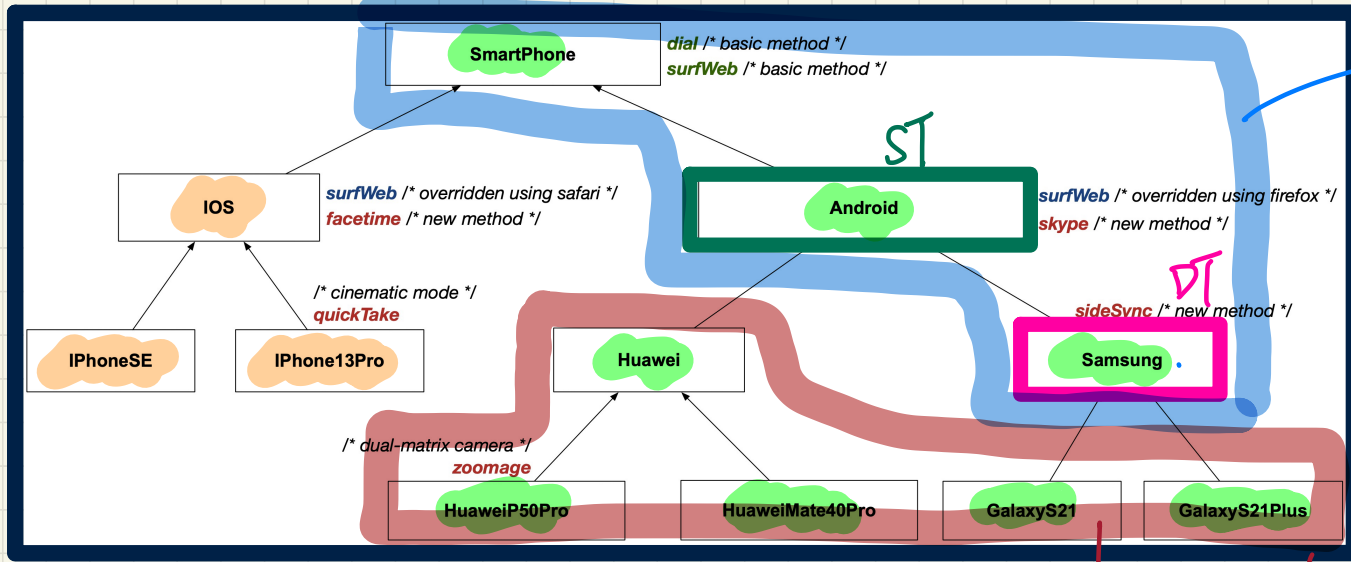
## Exercise: Compilable Type Cast? Fail at Runtime? (2)



```
SmartPhone myPhone = new Samsung();
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
iPhone13Pro ip = (iPhone13Pro) myPhone;
```

Compilable? ClassCastException at runtime?

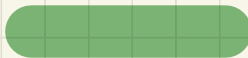
# Compilable Cast vs. Exception-Free Cast



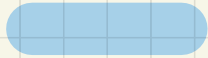
Android myPhone = new Samsung();

↓  
classes that are not ancestors of DT (Samsung).

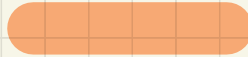
Compilable Casts



Exception-Free Casts



Non-Compilable Casts



ClassCastException

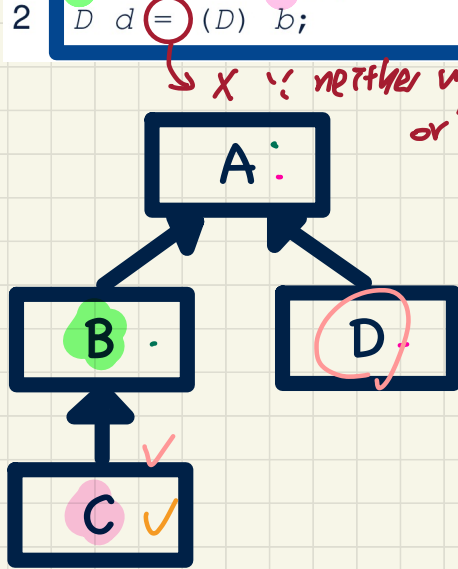


# Exercise: Compilable Cast vs. Exception-Free Cast

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

Compilable: ✓ ~~(Object)~~ (Object) obj  
ST: ~~Var~~ ~~cast~~

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



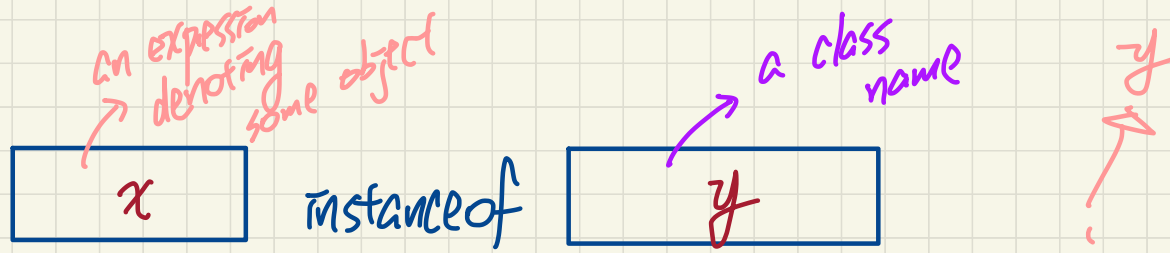
∴ D is not a descendant of D  
No CTE ∴ D is a descendant of A

$D d = (D) ((A) b)$   
upward cast

downward cast

↳ Runtime:

?  
 $(D) (A) b$

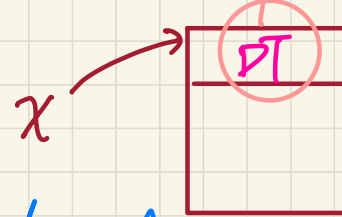


↳ evaluates to Boolean (T or F)

↳ True iff DT of  $x$  can fulfill the exp. of  $y$

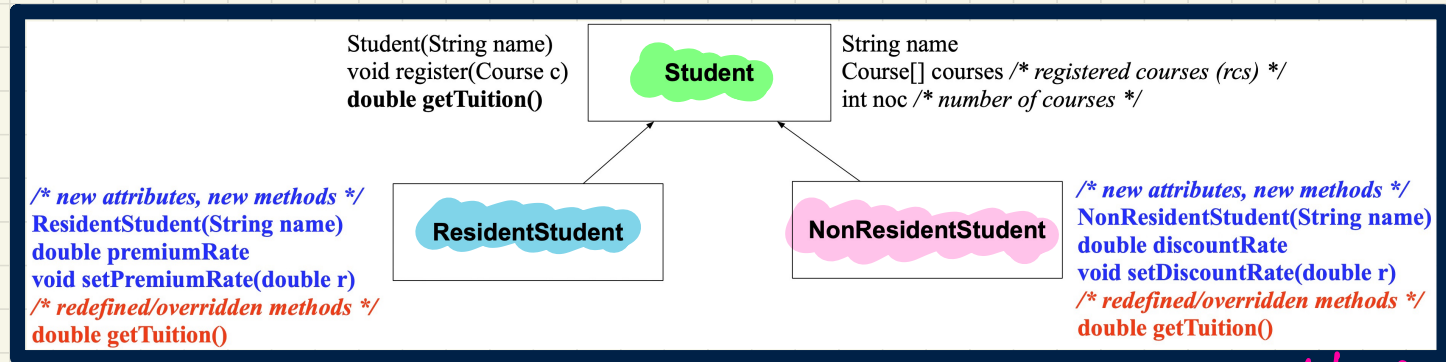
(2) DT of  $x$  is a descendant of  $y$   
(3)  $y$  is an ancestor of the DT of  $x$

↳ iff true, then it's safe to write  $(y) x$   
No CCE





# Checking Dynamic Types at Runtime (1)

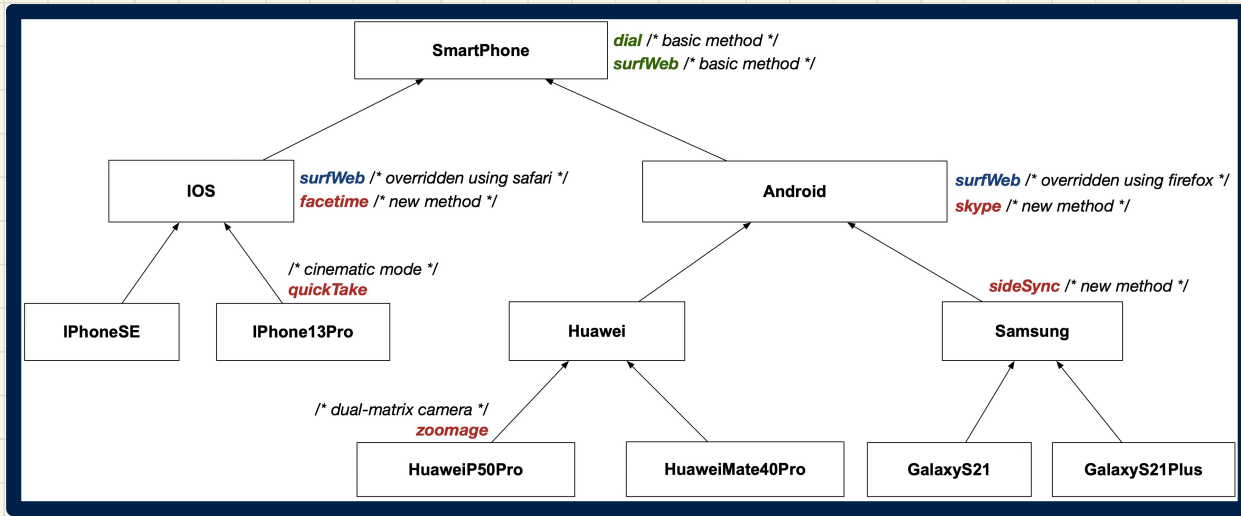


```
1 Student jim = new NonResidentStudent("J. Davis");
2 if (jim instanceof ResidentStudent) {
3 ResidentStudent rs = (ResidentStudent) jim;
4 rs.setPremiumRate(1.5);
5 }
```

evaluating to false avoids the CCE  
would trigger CCE

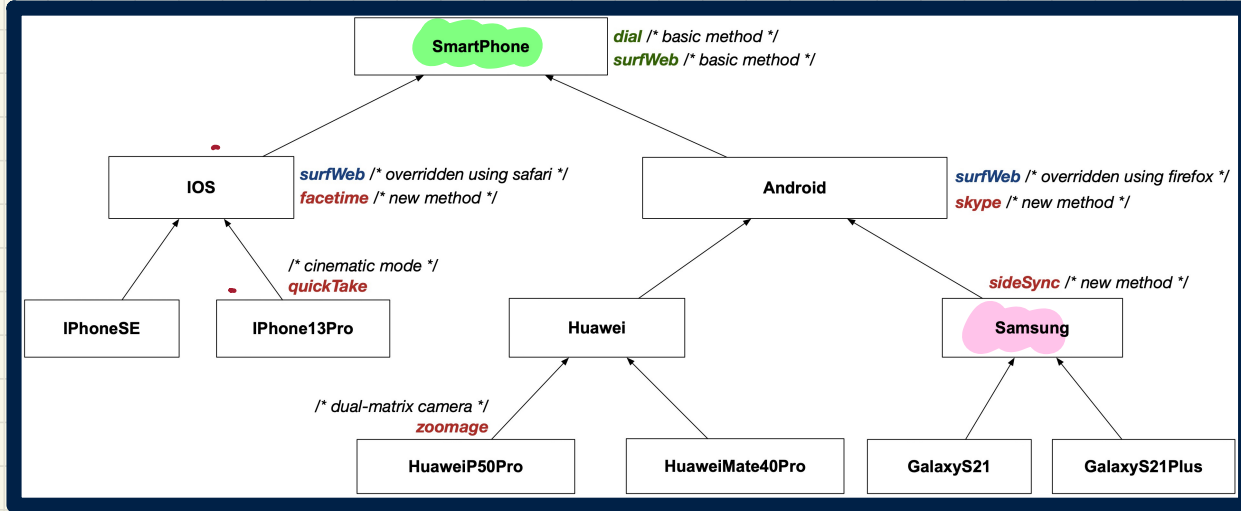
Can the DT of jim (NRS) fulfill exp. of RS?  
False: ∵ NRS is not a descent of RS.

# 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 }
```

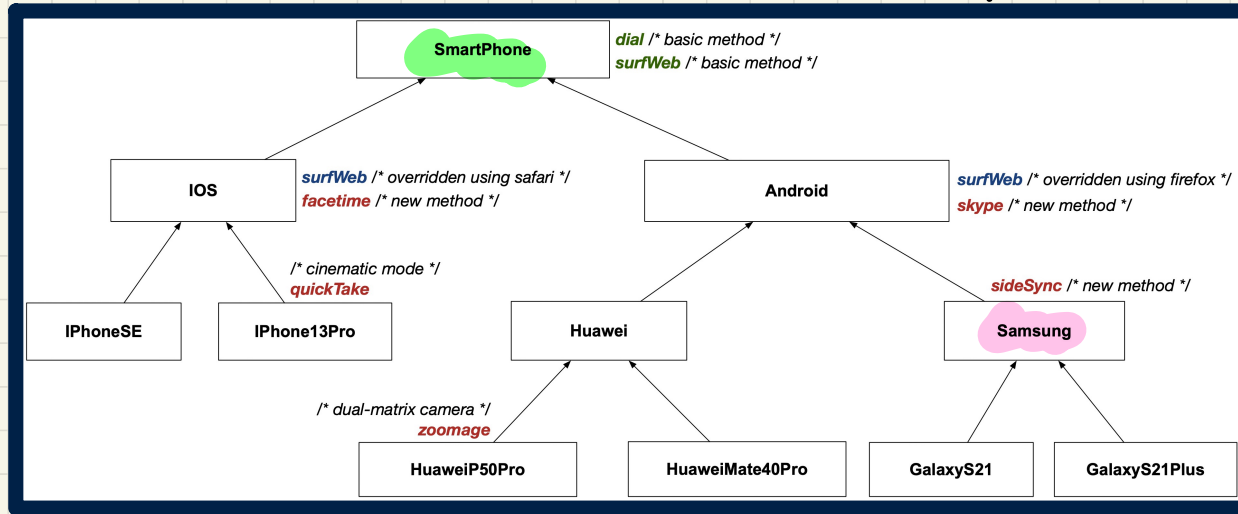
# Use of the instanceof Operator



```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android); (T)
println(myPhone instanceof Samsung); (T)
println(myPhone instanceof GalaxyS21); (F)
println(myPhone instanceof IOS); (F)
println(myPhone instanceof iPhone13Pro); (F).
```

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

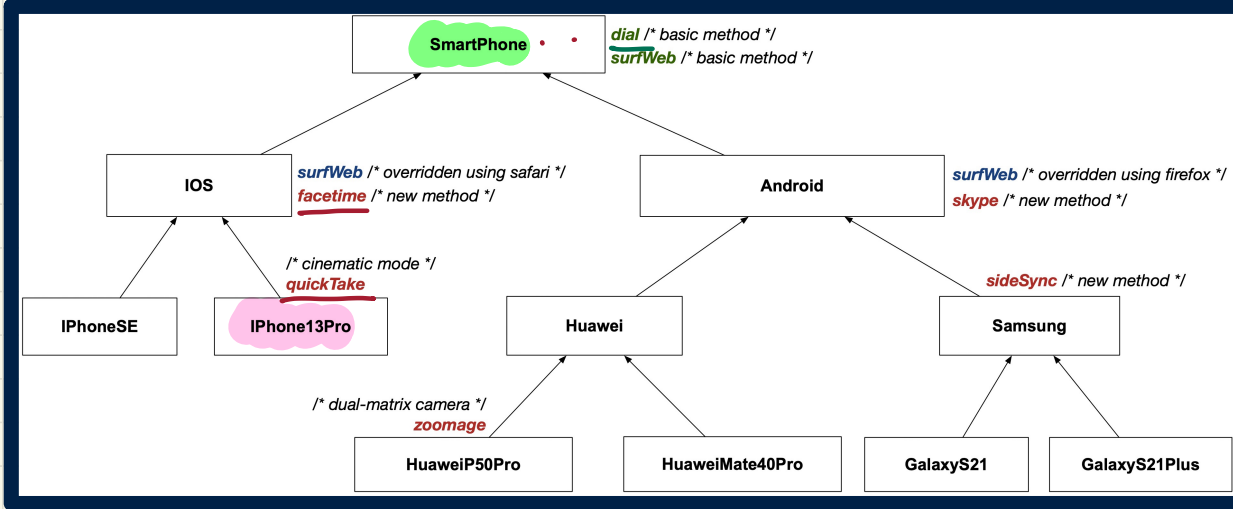
# Safe Cast via Use of the instanceof Operator



```
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 ??.

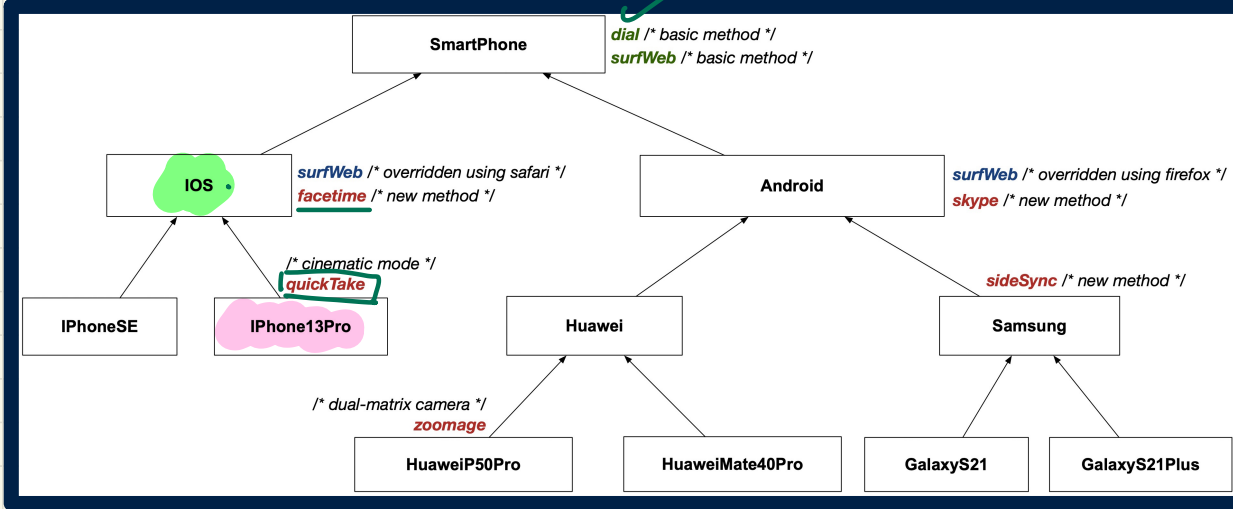
# Static Types, Casts, Polymorphism (1)



```
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(); ✗
```

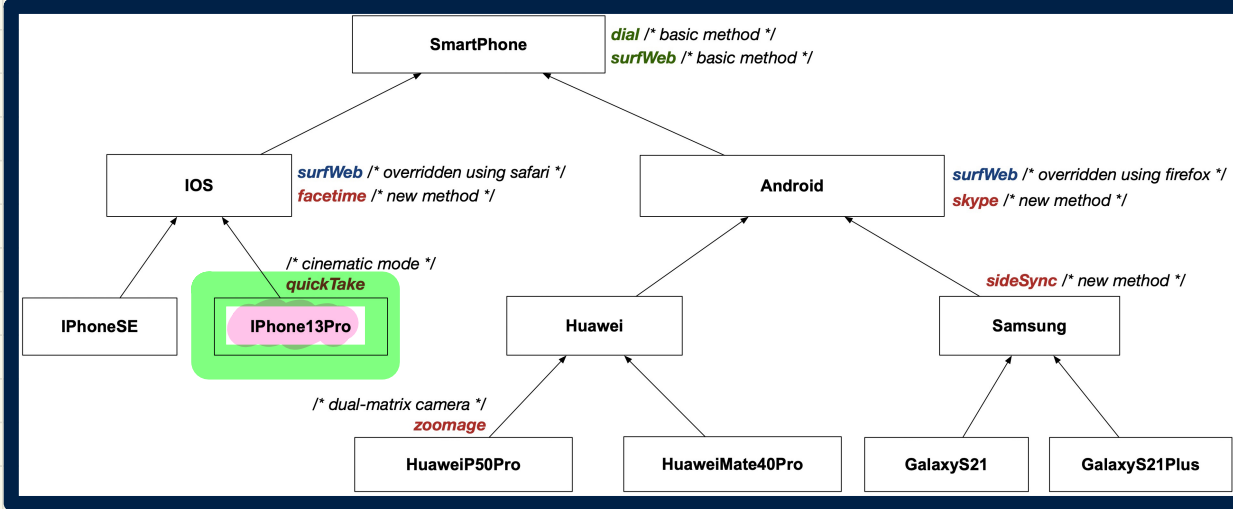
# Static Types, Casts, Polymorphism (2)



```
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(); ✗
```

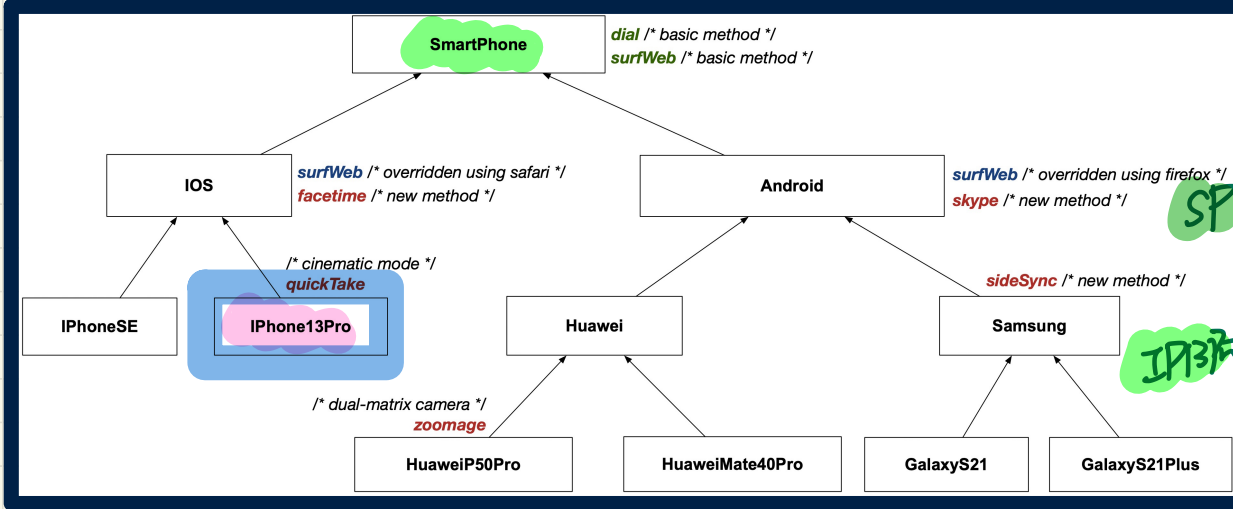
# Static Types, Casts, Polymorphism (3)



```
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)

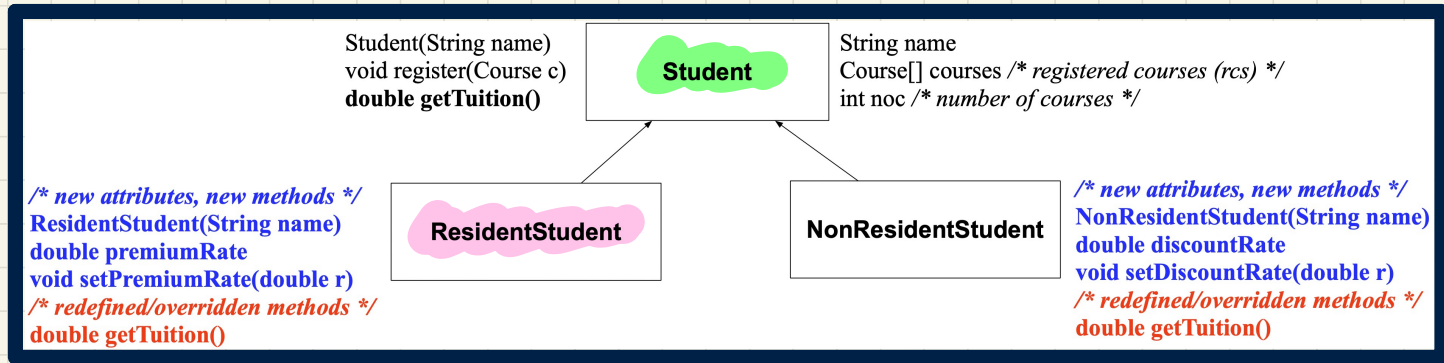


```
class SmartPhone {
 void dial() { ... }
}
class IOS extends SmartPhone {
 void facetime() { ... }
}
class iPhone13Pro extends IOS {
 void quickTake() { ... }
}
```

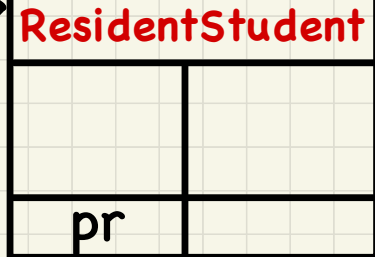
```
1 SmartPhone sp = new iPhone13Pro();
2 ((iPhone13Pro) sp).dial();
3 ((iPhone13Pro) sp).facetime();
4 ((iPhone13Pro) sp).quickTake();
```



# Static Types, Casts, Polymorphism (5)



Student s

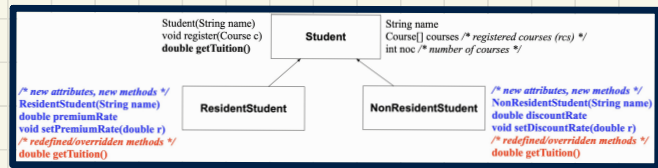


```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if (s instanceof ResidentStudent) {
 ((ResidentStudent) s).setPremiumRate(1.75);
 System.out.println(((ResidentStudent) s).getTuition());
}
```

*twl.*

*No CCE!*

# Polymorphic Parameters (1)



```
1 class StudentManagementSystem {
2 Student [] ss; /* ss[i] has static type ██████████ */ 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]`?

call by value:  $rs = 0$ ;  $\rightarrow$  ST of arg `Student`  
 $\rightarrow$  0 should be a descendant of ST of param. `rs`.

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

$\because$  ST of `rs` (RS) is a descendant of ST of `ss[c]` (Stud.)

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

`sms.addRS(o)`

## Lecture 21 - Nov. 21

### Inheritance

***Polymorphic Arrays***

***Polymorphic Return Values***

***Type-Checking Rules***

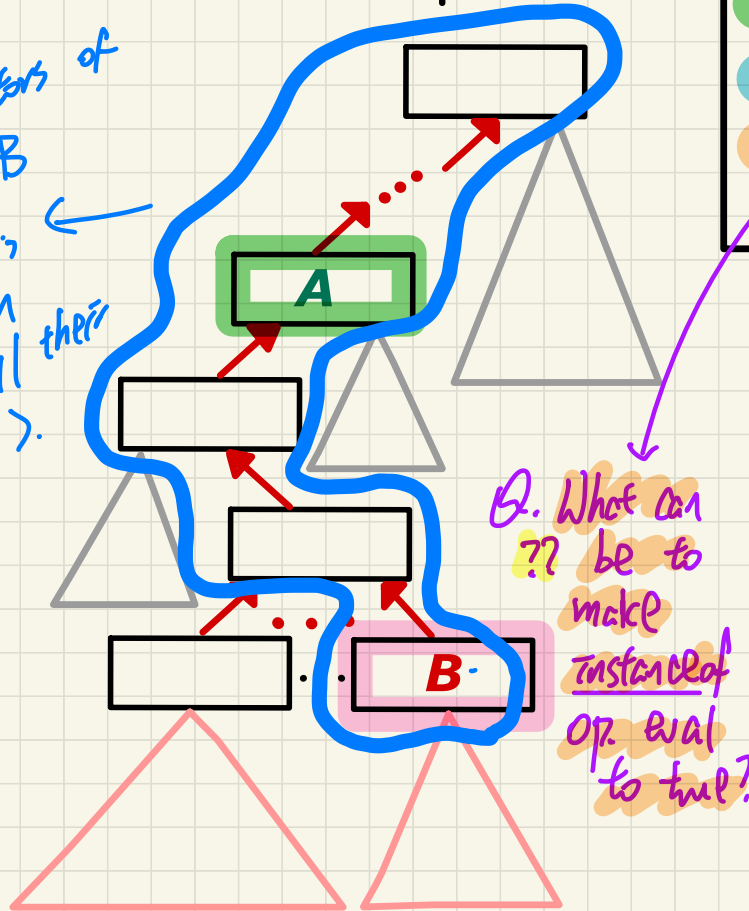
***Solving Problems Recursively***

## 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

# The **instanceof** Operator

Answers of  
PT B  
(i.e. B can  
fulfill their  
exp.).



```
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 ??.

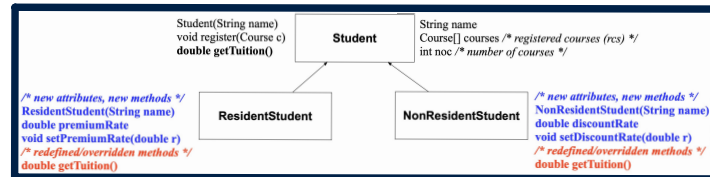
- L2:

- Evaluates to true if B can fulfill expectations on ??.

# Polymorphic Parameters (2)

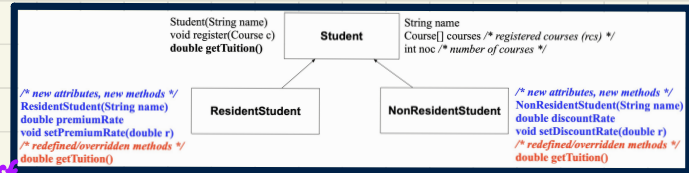
```
1 class StudentManagementSystem {
2 Student [] ss; /* ss[i] has static type Student */ int c;
3 void addRS(ResidentStudent rs) { ss[c] = rs; c++; }
4 void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5 void addStudent(Student s) { ss[c] = s; c++; } }
```

```
Student s1 = new Student();
Student s2 = new ResidentStudent();
Student s3 = new NonResidentStudent();
ResidentStudent rs = new ResidentStudent();
NonResidentStudent nrs = new NonResidentStudent();
StudentManagementSystem sms = new StudentManagementSystem();
1 sms.addRS(s1); X
2 sms.addRS(s2); X
3 sms.addRS(s3); X
4 sms.addRS(rs); ✓
5 sms.addRS(nrs); X
6 sms.addStudent(s1); ✓
7 sms.addStudent(s2); ✓
8 sms.addStudent(s3); ✓
9 sms.addStudent(rs); ✓
10 sms.addStudent(nrs); ✓
```



# Casting Arguments

```
void addRS(ResidentStudent rs)
```



```
sms.addRS((ResidentStudent) s) downward cast compiles?
```

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

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?

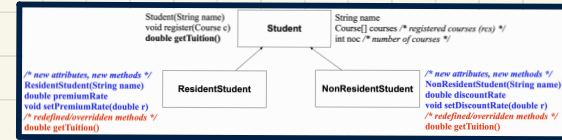
No CCE.

```
sms.addRS((ResidentStudent) nrs) no! it's neither upward nor downward. compiles?
```

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

ST: nrs

# 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 eeecs2030 = new Course("EECS2030", 500.0);
9 sms.registerAll(eeecs2030);
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 }

```

```

class StudentManagementSystem {
 Student[] students;
 int numOFStudents;

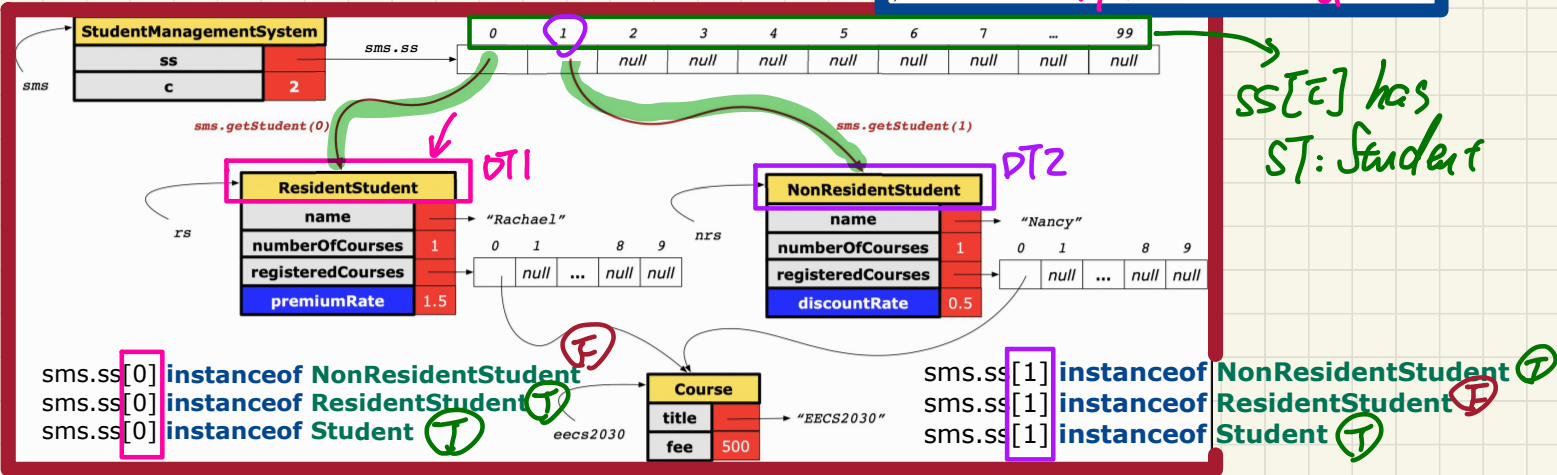
 void addStudent(Student s) {
 students[numOFStudents] = s;
 numOFStudents++;
 }

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

```

any arg. of ST that's descendant of Student can be passed (polymorphism).

ST: Student  
DT: RS or NRS (descendants of ST).





# 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); /* true */
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); /* true */
print(s instanceof ResidentStudent); /* false */
print(s.getTuition()); /* Version in NonResidentStudent called: 250 */
NonResidentStudent nrs2 = sms.getStudent(1); *

```

```

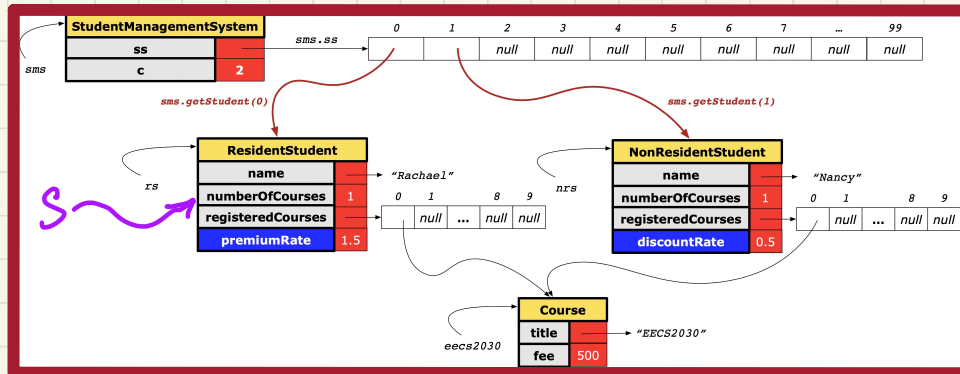
class StudentManagementSystem {
 Student[] ss; int c;
 void addStudent(Student s) { ss[c] = s; c++; }
 Student getStudent(int i) {
 Student s = null;
 if(i < 0 || i >= c) {
 throw new IllegalArgumentException("Invalid index");
 }
 else {
 s = ss[i];
 }
 return s;
 }
}

```

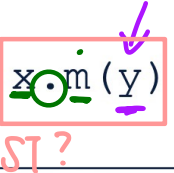
Polymorphic array  
 ⇒ s may be assigned an object whose DT is a descendant of Student

static type of return value

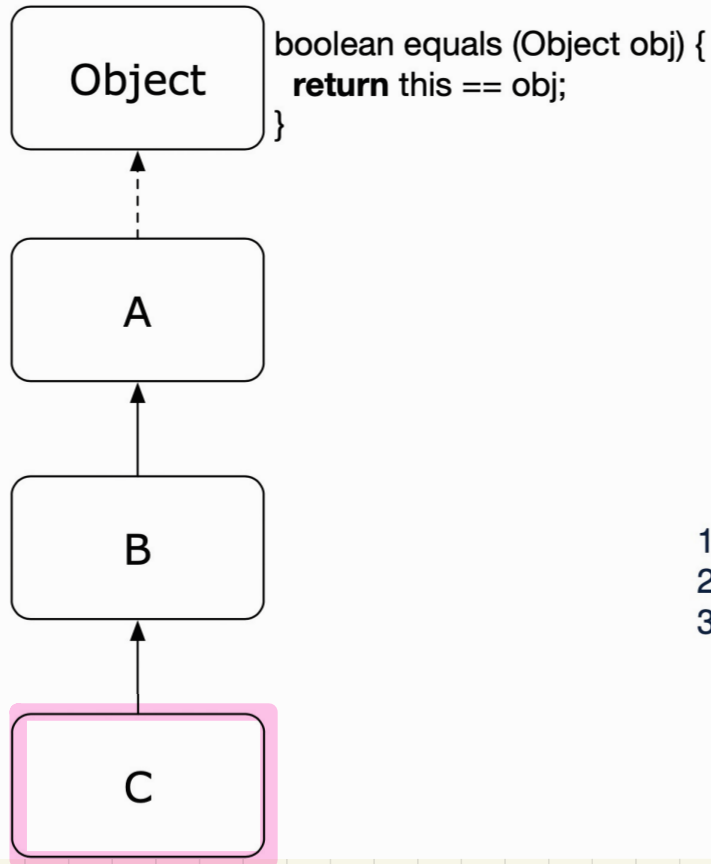
↳ polymorphism: DT of RV can be any of the descendants of Student



# Summary: Type Checking Rules

| CODE                                                                                              | CONDITION TO BE TYPE CORRECT                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $x = y$                                                                                           | Is $y$ 's <b>ST</b> a <b>descendant</b> of $x$ 's <b>ST</b> ?                                                                                                                                                                                                                                                                                                                                                                                                   |
| $x.m(y)$                                                                                          | Is method $m$ defined in $x$ 's <b>ST</b> ?<br>Is $y$ 's <b>ST</b> a <b>descendant</b> of $m$ 's parameter's <b>ST</b> ?                                                                                                                                                                                                                                                                                                                                        |
| $z = x.m(y)$<br> |  Is method $m$ defined in $x$ 's <b>ST</b> ?<br> Is $y$ 's <b>ST</b> a <b>descendant</b> of $m$ 's parameter's <b>ST</b> ?<br> Is <b>ST</b> of $m$ 's return value a <b>descendant</b> of $z$ 's <b>ST</b> ? |
| $(C) \ y$                                                                                         | Is $C$ an <b>ancestor</b> or a <b>descendant</b> of $y$ 's <b>ST</b> ?                                                                                                                                                                                                                                                                                                                                                                                          |
| $x = (C) \ y$                                                                                     | Is $C$ an <b>ancestor</b> or a <b>descendant</b> of $y$ 's <b>ST</b> ?<br>Is $C$ a <b>descendant</b> of $x$ 's <b>ST</b> ?                                                                                                                                                                                                                                                                                                                                      |
| $x.m((C) \ y)$                                                                                    | Is $C$ an <b>ancestor</b> or a <b>descendant</b> of $y$ 's <b>ST</b> ?<br>Is method $m$ defined in $x$ 's <b>ST</b> ?<br>Is $C$ a <b>descendant</b> of $m$ 's parameter's <b>ST</b> ?                                                                                                                                                                                                                                                                           |

# Overridden Methods and Dynamic Binding (1)

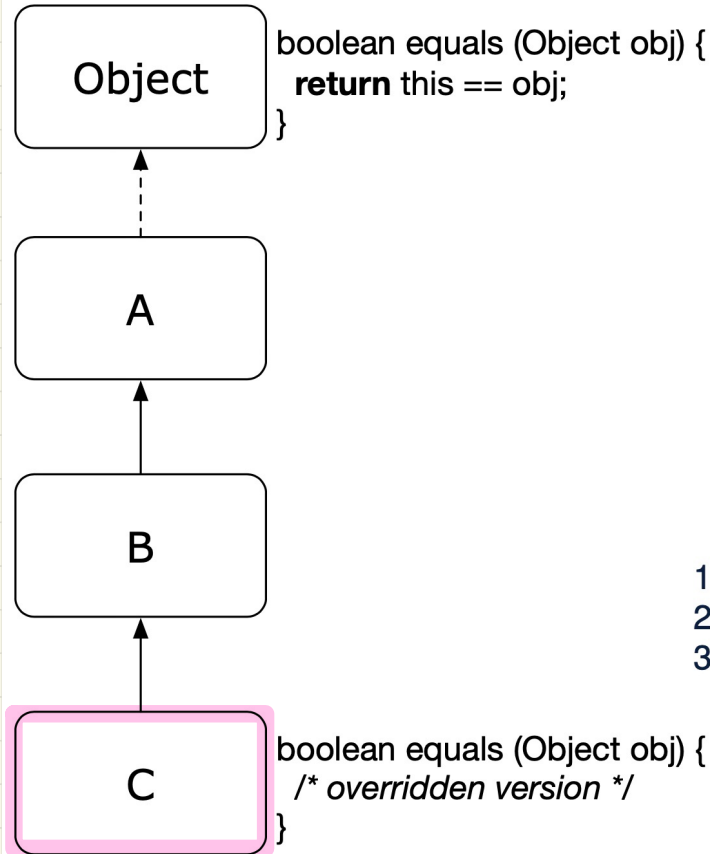


```
class A {
 /*equals not overridden*/
}
class B extends A {
 /*equals not overridden*/
}
class C extends B {
 /*equals not overridden*/
}
```

```
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)

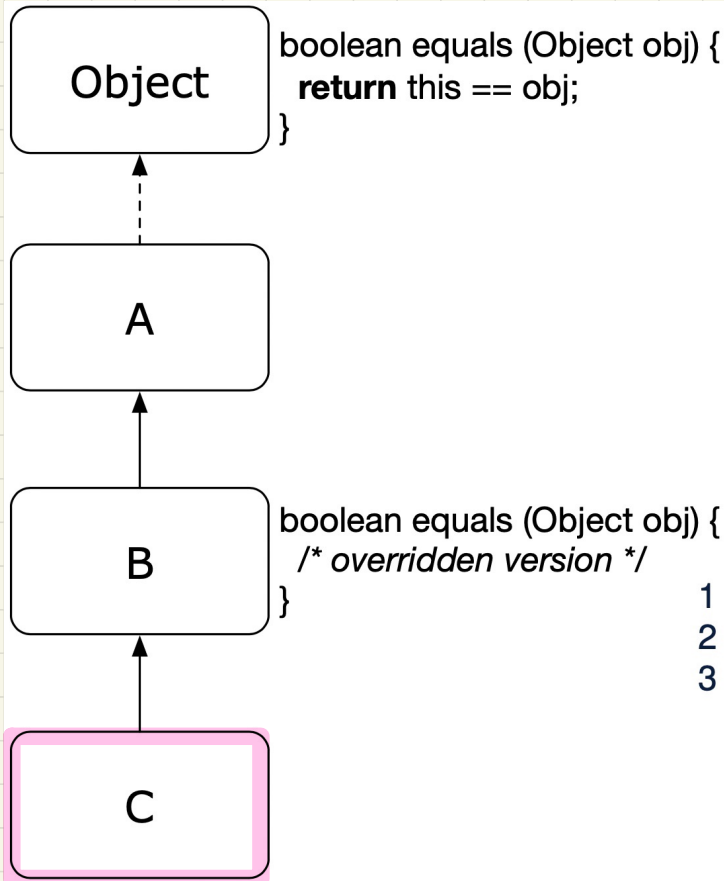


```
class A {
 /*equals not overridden*/
}
class B extends A {
 /*equals not overridden*/
}
class C extends B {
 boolean equals(Object obj) {
 /* overridden version */
 }
}
```

```
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)



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

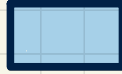
```
1 Object c1 = new C();
2 Object c2 = new C();
3 println(c1.equals(c2));
```

L3 calls which version of  
equals? [ B ]

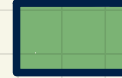
# Solving a Problem Recursively

Best  
Case

Given a **small** problem:

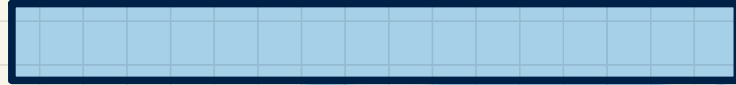


Solve it **directly**:

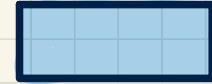
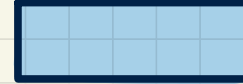
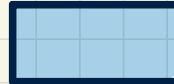


Recursive  
Cases

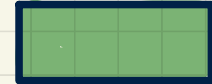
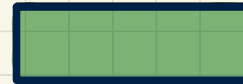
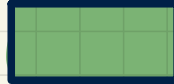
Given a **big** problem:



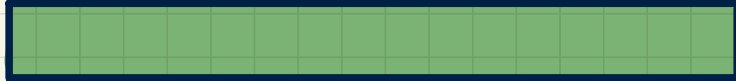
Divide it into **smaller** problems:



Assume solutions to **smaller** problems:



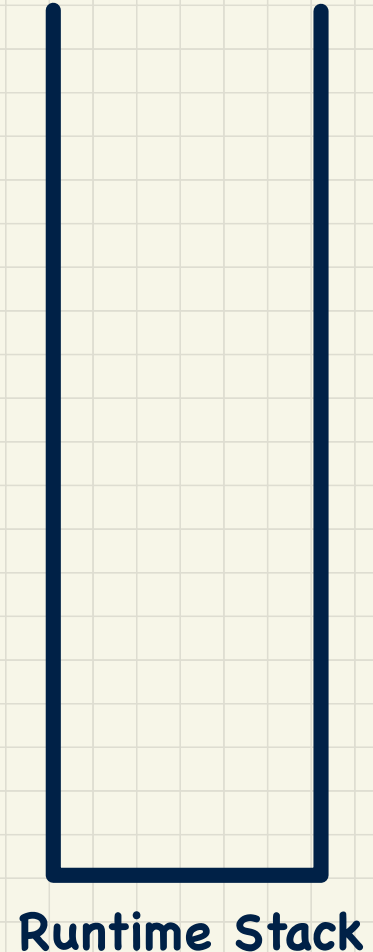
Combine solutions to **smaller** problems:



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

# 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**.



## 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: factorial

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

not recursive

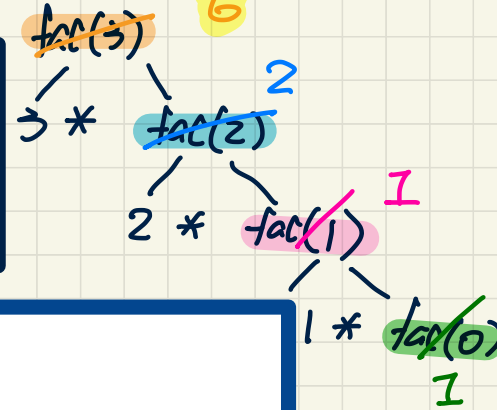
no recursive call to "!"

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

recursive call  
↓  
solution to a strictly smaller problem

## Recursive Solution in Java: factorial

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

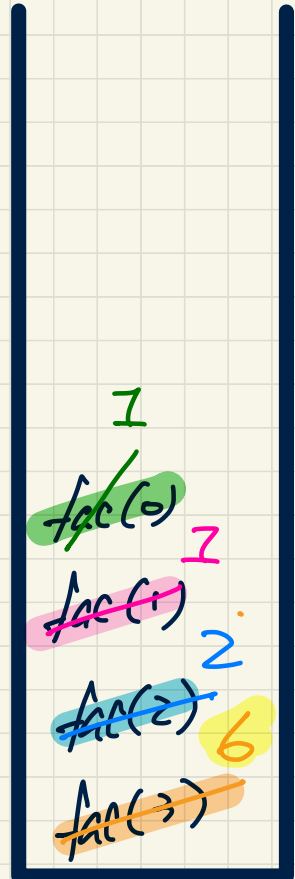


```
int factorial(int n) {
 int result;
 if (n == 0) { /* base case */ result = 1; }
 else { /* recursive case */
 result = n * factorial(n - 1);
 }
 return result;
}
```

Handwritten annotations on code:

- 3 \* fac(2) (orange box)
- 2 \* fac(1) (blue box)
- 1 \* fac(0) (pink box)
- 1 (green) above fac(0)
- 1 (green) below fac(0)
- 1 (green) below fac(1)
- 2 (blue) below fac(1)
- 2 (blue) below fac(2)
- 6 (orange) below fac(2)
- 1 (pink) below fac(1)
- 1 (green) below fac(0)

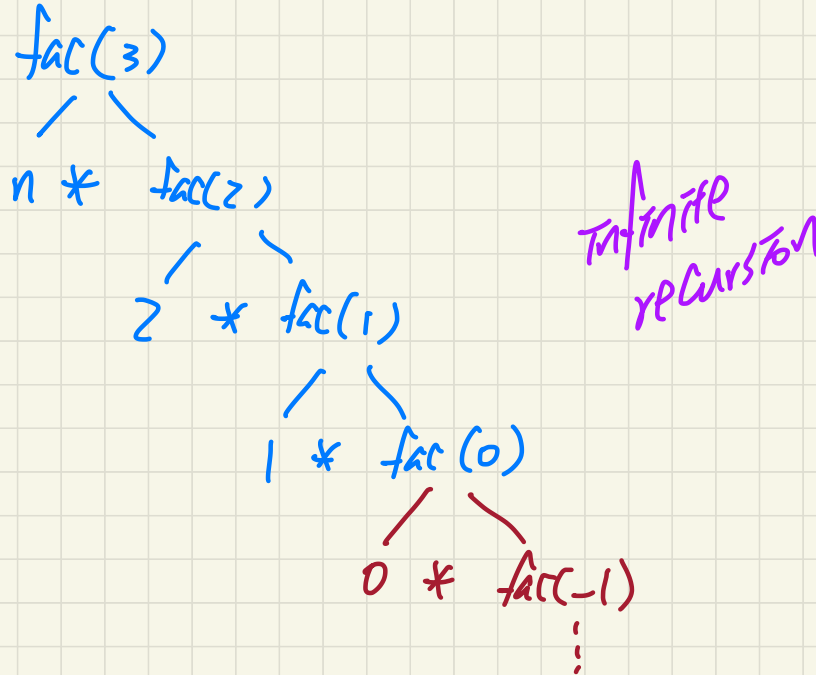
Example: factorial(3)



Runtime Stack

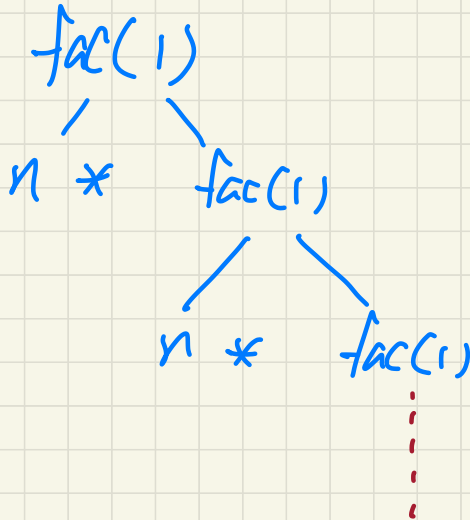
## Common Errors of Recursion (1)

```
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); }
}
```



*infinite  
recursion!*

## Recursive Solution: Fibonacci Numbers

$F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$

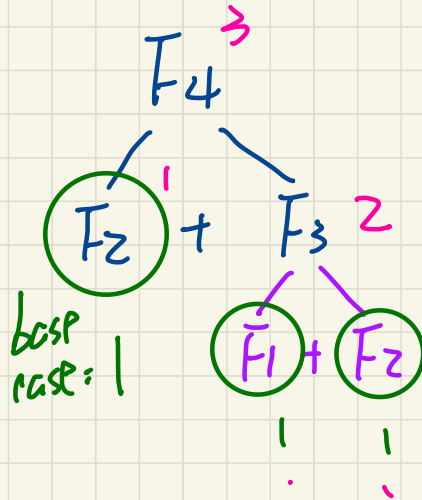
$F_1$   $F_2$

$F_4$

$F_6$

$F_7$

$F_8$



$$F_n = \begin{cases} 1 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 3 \end{cases}$$

two recursive calls

## 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;
}
```

Example: fib(4)

Runtime Stack

# Use of String

S = "Hello"

S  $\rightarrow$ 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| H | e | l | l | o |

S.length() - 1

S.substring(i, j)

form a substring  
by collecting characters

m: [i, j]

typically:  
[i ≤ j] [i, j-1]

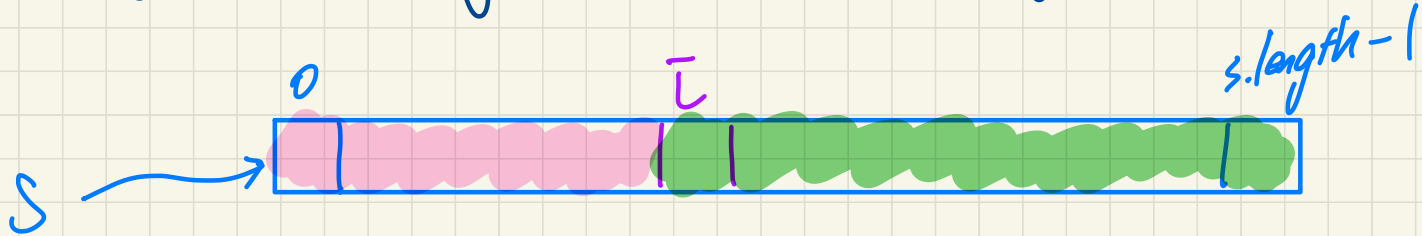
```
public class StringTester {
 public static void main(String[] args) {
 String s = "abcd";
 System.out.println(s.isEmpty()); /* false */
 /* Characters in index range [0, 0) */
 String t0 = s.substring(0, 0); [0, -1] → i > j → ""
 System.out.println(t0); /* "" */
 /* Characters in index range [0, 4) */
 String t1 = s.substring(0, 4); [0, 3]
 System.out.println(t1); /* "abcd" */
 /* Characters in index range [1, 3) */
 String t2 = s.substring(1, 3); [1, 2]
 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();
 }
}
```



For any non-null string  $s$ ,

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

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



# Recursions on Strings

ex "abcd"

## Reversal

"abcd"

## Palindrome

$isp(racecar)$

$\hookrightarrow r == r \ \&\& \ isp(aceca)$

"racecar"

"aracecars"

"raceacar"

## Number of Occurrences

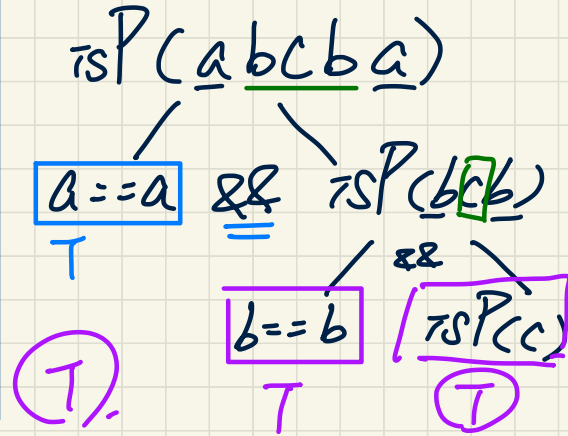
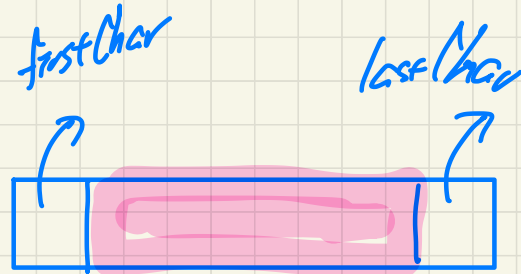
"abca"

'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);
 }
}
```



ex. `isP('racecar')`

ex. `isP('racebar')`

## 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. justification
3. definitions
4. tracing output
5. coding.
  - ↳ fragments
  - ↓ recursive method.

# 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?

triangle(1) = 1

triangle(2) = 3

triangle(5) = ?

5 rows

5 + triangle(5-1)

bottom row

statically smaller problem (triangle(4))

# 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.

reduce the size of the range of elements in the array to look at

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

10 ==

i i+1

A[i] \* A[i+1]

Try two versions:

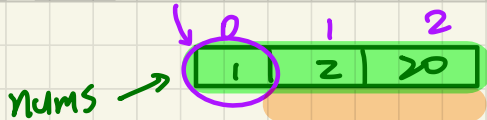
**boolean** array220(**int**[] nums, **int** from)

**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 || 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

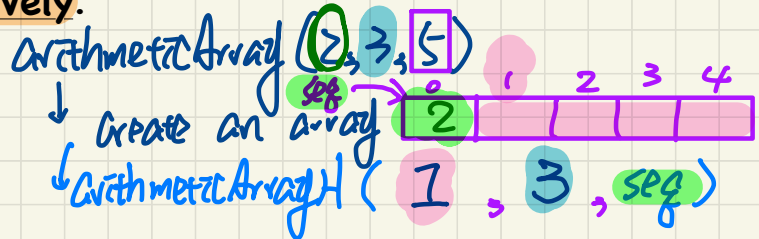
'n': the first n numbers in an arithmetic sequence

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(?, 3, 0) → {}

arithmeticArray(2, 3, 1) → {2}





## 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.

occ("abca", 'a')

isp(racecar)

Palindrome

↪ r == r && isp(aceca)

"racecar"

"aracecars"

"raceacar"

## Reversal

reverseOf("abcd") ↪ strictly smaller subproblem

reverseOf(bcd) + "a"  
dcba

## Number of Occurrences

occ("abca", 'a')

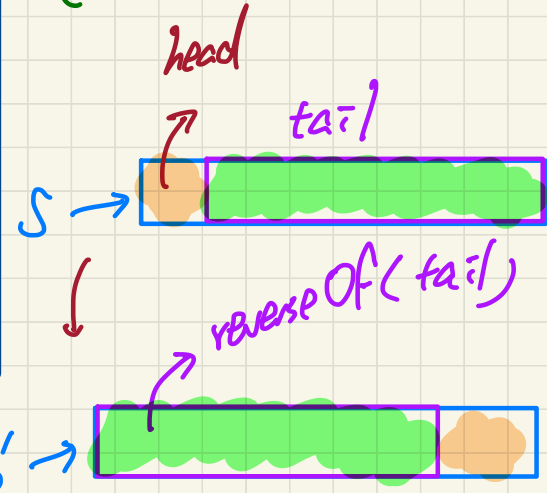
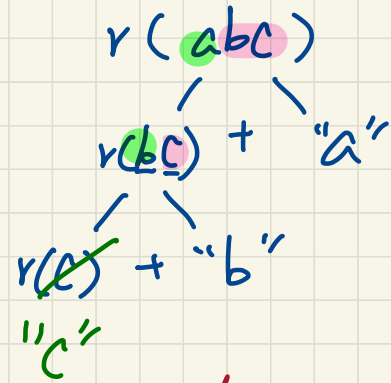
↪ 1 + occ(bca)

occ("abca", 'b')

↪ 0 + occ(bca)

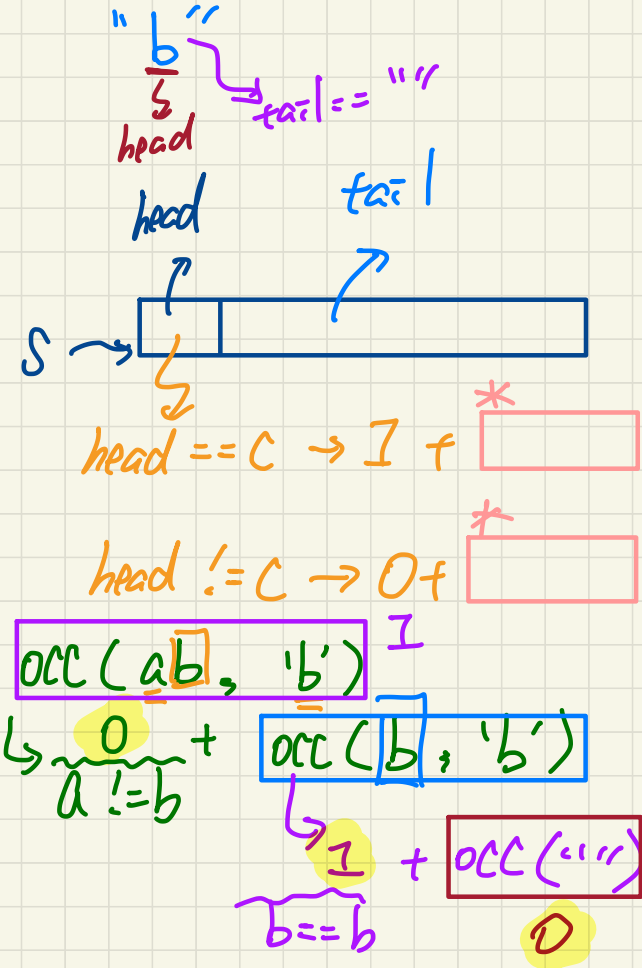
# Problem: Reverse of a String

```
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;
 }
}
```



# 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 new, strictly smaller array

Say  $a_1 = \{\}$ , consider  $m(a_1)$

↪ base case # 1

Say  $a_2 = \{A, B, C\}$ , consider  $m(a_2)$

time & space wasted to create these sub-arrays

$m(\begin{matrix} 0 & 1 & 2 \\ A & B & C \end{matrix})$

$m(\begin{matrix} 0 & 1 \\ B & C \end{matrix})$

$m(\begin{matrix} 0 \\ C \end{matrix})$

↪ base case # 2

# Recursion on an Array: Passing Same Array Reference

```
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) } }
```

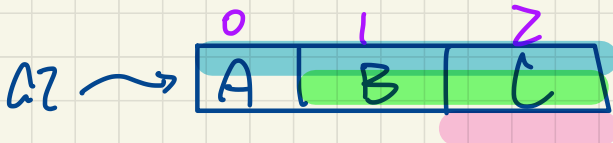
call by value

$0 - 1 = -1$

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

↳ base case # 1

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



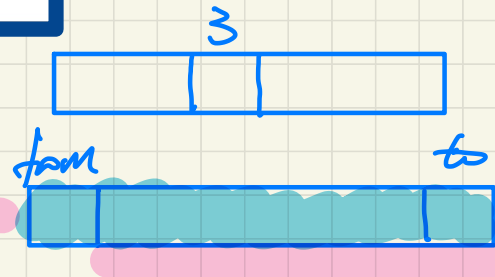
base case  
# 2

$m(a2, 0, 2)$

↓  
 $m(a2, 1, 2)$

↓  
 $m(a2, 2, 2)$

for non-empty range: from  $\leq$  to  
form a range of elements to look at in the current method call



## Problem: Are All Numbers Positive?

Say  $a = \{\}$

$\forall x \in \emptyset. P(x) \equiv \text{True}$  ] there's no witness to show some violation

Say  $a = \{4\}$

$\exists x \in \emptyset. P(x) \equiv \text{False}$  ] there's no witness to show some satisfaction

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

$\text{allP}(a, 0, 3)$   $\rightarrow a.length - 1$

$\hookrightarrow a[0] > 0$   $\boxed{\text{xx}}$   $\text{allP}(a, 1, 3)$

$\nexists a[0] \leq 0$   
 $\hookrightarrow$  by prev RHS

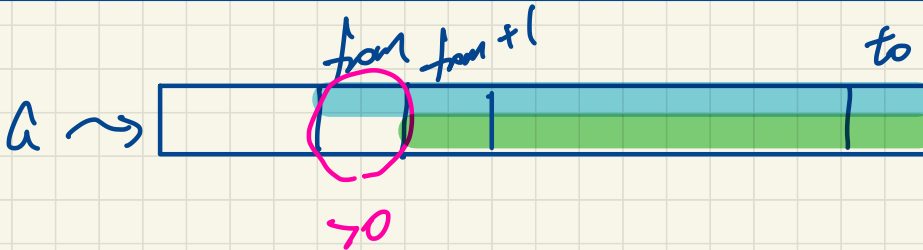
Say  $a = \{5, 3, -2, 9\}$



# Problem: Are All Numbers Positive?

```
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);
 }
}
```



recursive  
helper  
method

public

private

# Tracing Recursion: `allPositive`

Say `a = {}`

`allPositive(a)`

`allPH(a,0,-1)`

```
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  $a = \{4\}$

allPositive(a)

allPH(a,0,0)

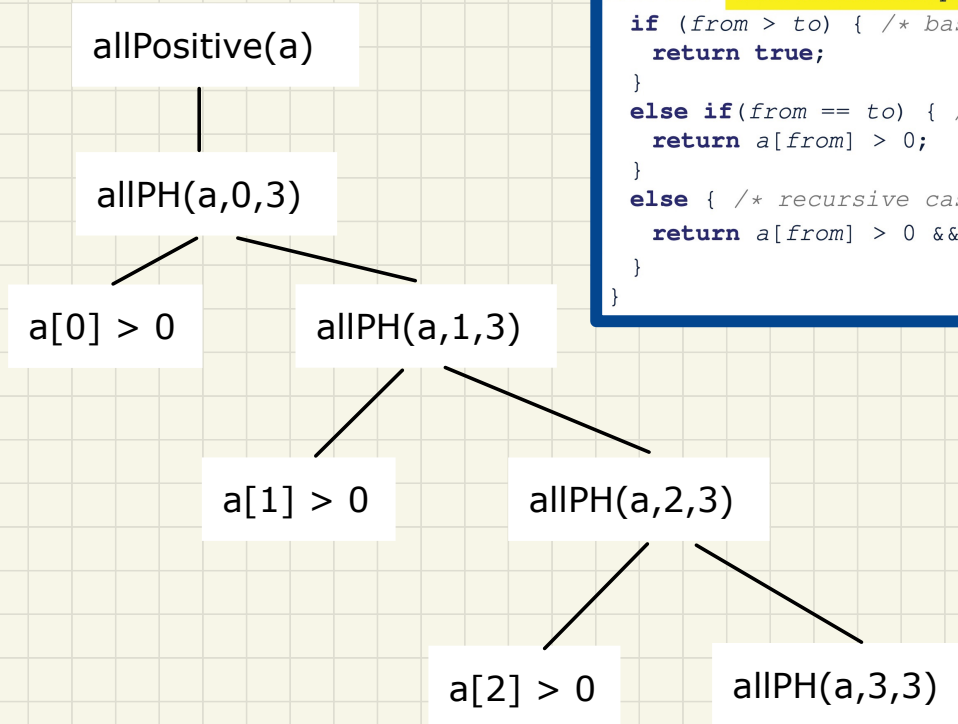
$a[0] > 0$

```
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  $a = \{4, 7, 3, 9\}$

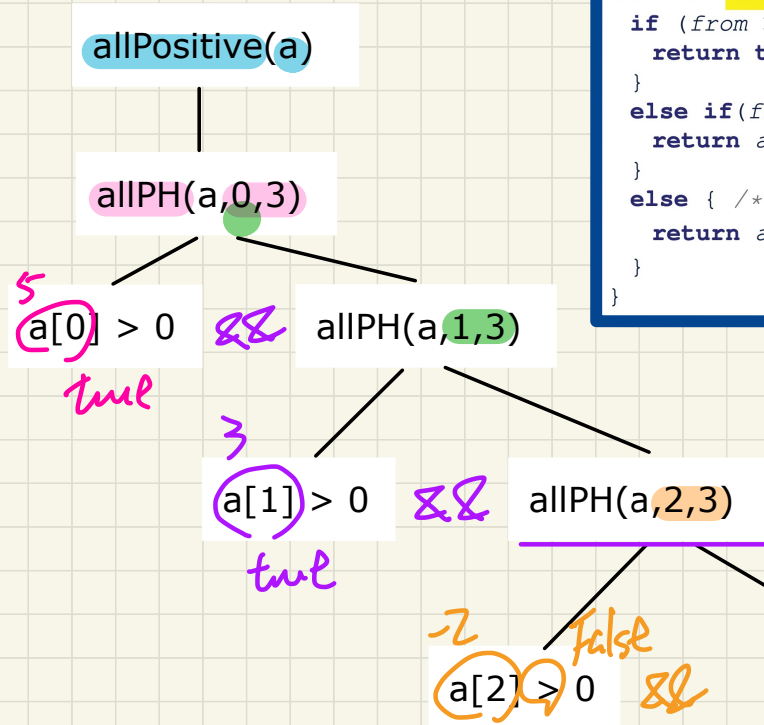


```
boolean allPositive(int[] a) {
 return allPositiveHelper(a, 0, a.length - 1);
}

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  $a = \{5, 3, -2, 9\}$



```
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);
 }
}
```

∴ short-circuit  
Evaluation

bypassed

non-ascending ?

non-descending

| $i$ | $j$ |
|-----|-----|
|     |     |

$a[i]$

$a[j]$

$$\neg (a[i] > a[j])$$

$$\equiv a[i] \leq a[j]$$

## Problem: Are Numbers Sorted?

Say  $a = \{\}$

Say  $a = \{4\}$

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

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

$isSorted(a, \overset{from}{0}, 3)$

$a[0] \leq a[1] \&\& isSorted(a, 1, 3)$

## 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  $a = \{\}$

isSorted(a)

isSH(a,0,-1)

```
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  $a = \{4\}$

isSorted(a)

isSH(a,0,0)

return **true**

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

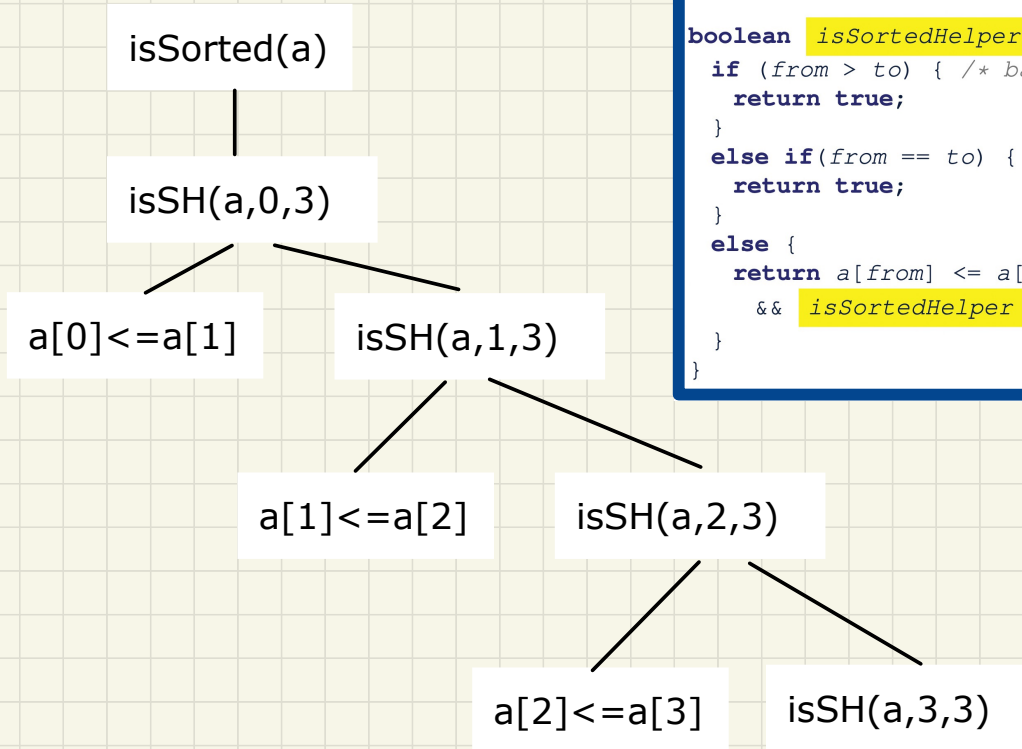
boolean isSortedHelper(int[] a, int from, int to) {
 if (from > to) { /* base case 1: empty range */
 return true;
 }
 else if (from == to) { /* base case 2: range of one element */
 return true;
 }
 else {
 return a[from] <= a[from + 1]
 && isSortedHelper(a, from + 1, to);
 }
}
```

# 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}

isSorted(a)

isSH(a, 0, 3)

a[0] <= a[1]

isSH(a, 1, 3)

a[1] <= a[2]

6 <= 5  
False

```
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);
 }
}
```

bypassed

## Lecture 24 - Dec. 3

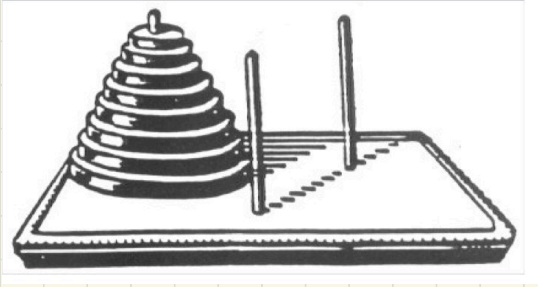
### Recursion

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

## Announcements/Reminders

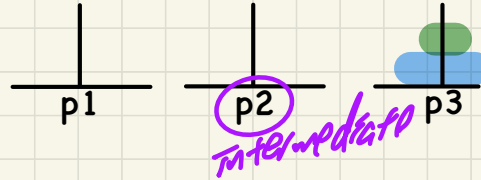
- **Lab5** due midnight today  
+ Required study: **Abstract Classes & Interfaces**
- **ProgTest3** results released
- Extra office hours: 3pm to 5pm on Thursday
- **Exam** Review Session (Zoom): 3pm on Friday
- Materials for tutorial session on **recursion**

# Tower of Hanoi: Strategy



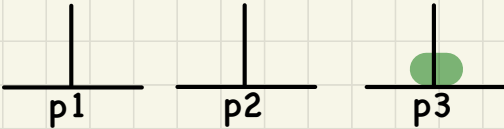
Consider 2 disks:  $A < B$

move  from p1 to p3



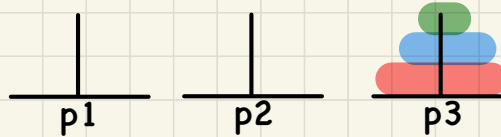
Consider 1 disk: A

move  from p1 to p3

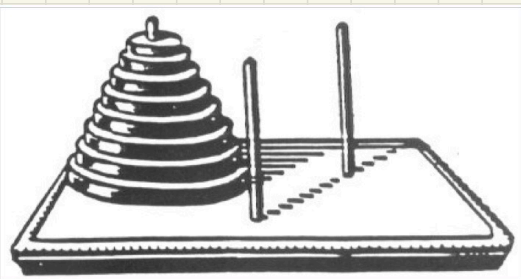


Consider 3 disks:  $A < B < C$

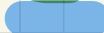
move  from p1 to p3

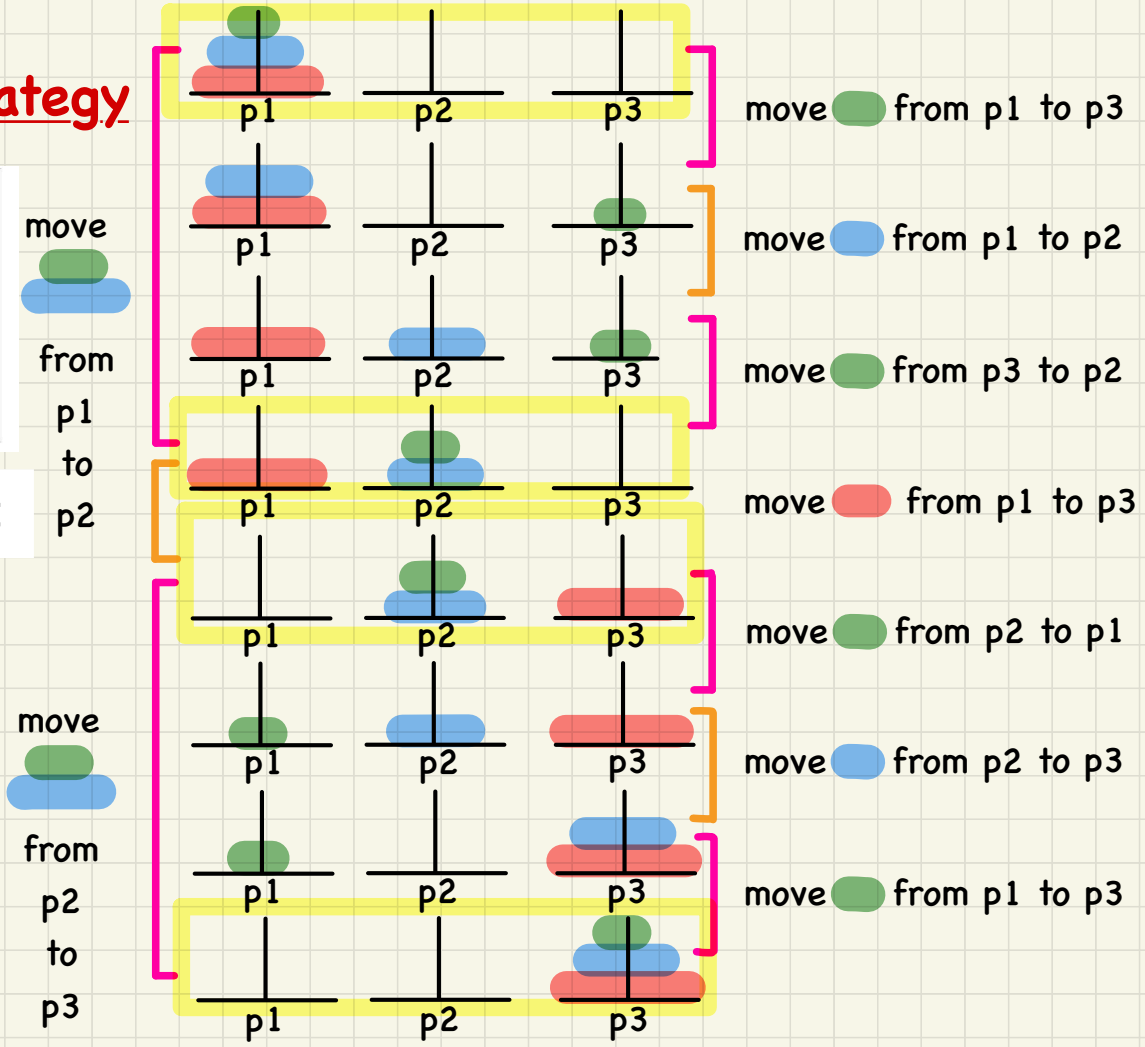


# Tower of Hanoi: Strategy



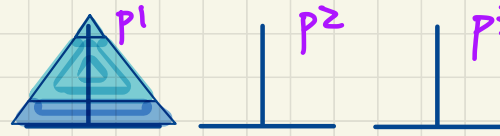
Consider 3 disks  $A < B < C$

move   from p1 to p3

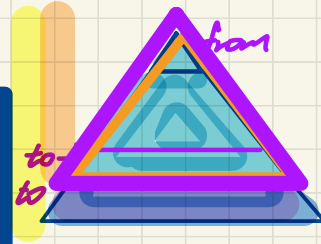




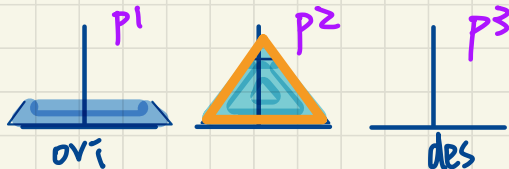
# Tower of Hanoi in Java



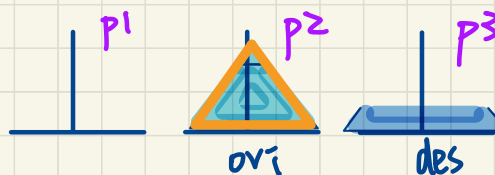
```
void towerOfHanoi(String[] disks) {
 toHelper(disks, 0, disks.length - 1, 1, 3);
}
// recursive helper method
void toHelper(String[] disks, int from, int to, int ori, int des) {
 if (from > to) { }
 else if (from == to) {
 // only one disk to move
 print("move " + disks[to] + " from " + ori + " to " + des);
 }
 else {
 // 1. 2. 3.
 int intermediate = 6 - ori - des;
 1 toHelper(disks, from, to - 1, ori, intermediate);
 2 print("move " + disks[to] + " from " + ori + " to " + des);
 3 toHelper(disks, from, to - 1, intermediate, des);
 }
}
```



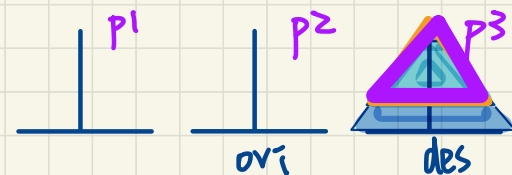
## Stage 1



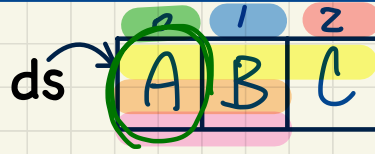
## Stage 2



## Stage 3



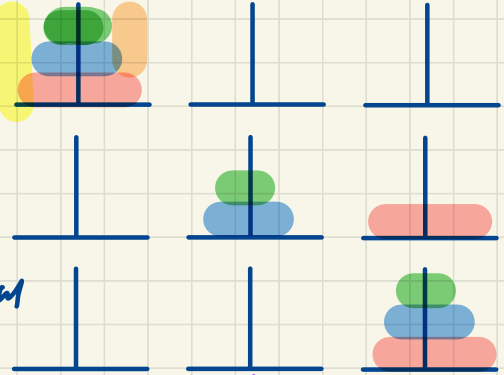
# Tower of Hanoi: Tracing



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

*ori*      *des.*

original problem  
subproblem #1  
subproblem #2



tohH(ds, 0, 1, pl, p2) <sup>intermediate</sup> <sup>④</sup> move C: pl to p3      tohH(ds, 0, 1, p2, p3) <sup>intermediate</sup> <sup>des</sup>

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

② move B: pl to p2

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

Exercise

⑤  
⑥  
⑦



# Tower of Hanoi: Tracing

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

$$\begin{aligned}
 tohH(ds, \underbrace{0, 2}_{\{A, B, C\}}, p1, p3) = & \left\{ \begin{aligned} & \text{Move } C: p1 \text{ to } p3 \\ & tohH(ds, \underbrace{0, 1}_{\{A, B\}}, p1, p2) = \left\{ \begin{aligned} & tohH(ds, \underbrace{0, 0}_{\{A\}}, p1, p3) = \left\{ \text{Move } A: p1 \text{ to } p3 \right. \\ & \text{Move } B: p1 \text{ to } p2 \\ & tohH(ds, \underbrace{0, 0}_{\{A\}}, p3, p2) = \left\{ \text{Move } A: p3 \text{ to } p2 \right. \end{aligned} \right. \\ & tohH(ds, \underbrace{0, 1}_{\{A, B\}}, p2, p3) = \left\{ \begin{aligned} & tohH(ds, \underbrace{0, 0}_{\{A\}}, p2, p1) = \left\{ \text{Move } A: p2 \text{ to } p1 \right. \\ & \text{Move } B: p2 \text{ to } p3 \\ & tohH(ds, \underbrace{0, 0}_{\{A\}}, p1, p3) = \left\{ \text{Move } A: p1 \text{ to } p3 \right. \end{aligned} \right. \end{aligned} \right.
 \end{aligned}$$



# Tower of Hanoi: Running Time

```

void towerOfHanoi(String[] disks) {
 tohHelper (disks, 0, disks.length - 1, 1, 3); $T(n)$
}
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; $T(n-1)$
 tohHelper (disks, from, to - 1, ori, intermediate); I
 print("move " + disks[to] + " from " + ori + " to " + des);
 tohHelper (disks, from, to - 1, intermediate, des); $T(n-1)$
 }
}

```

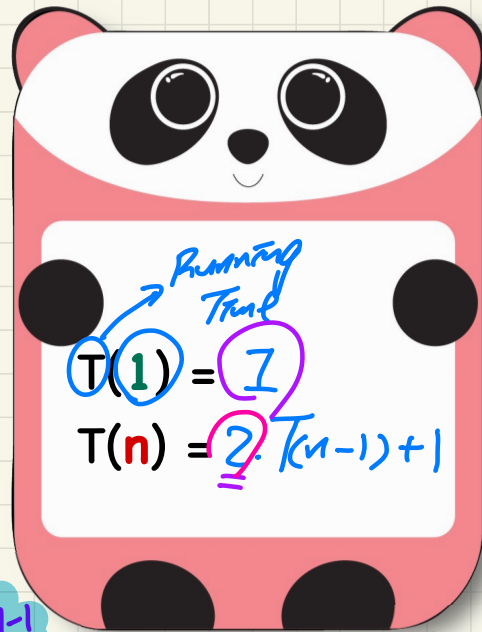
$$T(n) = 2 \cdot T(n-1) + 1$$

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

$$2^n = \frac{2 \cdot (2 \cdot (2 \cdot T(n-3) + 1) + 1) + 1}{2^3} + 3 = 2 \cdot (2 \cdot 2 \cdots T(1) + 1) + 1 + \cdots + (n-1)$$

$n - ? - 1$   
# unfolding

## Running Time as a Recurrence Relation



**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.

✓  
ext. → fail

no ext. → pass

SomeEx. → pass

no ext.  
ext other  
than SomeEx. → 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();
 obj.m();
 fail();
 }
 catch (SomeException e) {
 /* do nothing → pass */
 }
 }
}
```

SomeEx.  
(unexpected)

```

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

2nd call → expect

if exc. occurred → caught and "Fail" printed  
 if exc. not occurred → proceed with normal flow

reaching this line means the expected SomeExc did not occur.

SomeExc occurred (unexpectedly) → no fail!

```

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) {
 }
 }
}

```

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

```

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) {
 }
 }
}

```

SomeException

As soon as  
a first failure  
occurs,  
normal flow  
of exec should be  
interrupted

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

↓ (need to have nested try-catch)



```
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) {

 }
 }
}
```

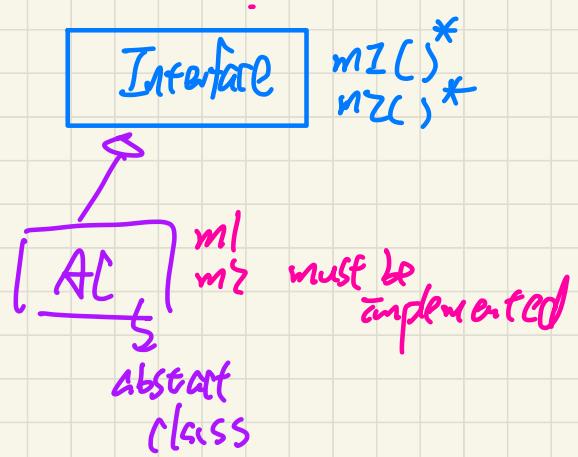
As soon as  
the first "fail"  
occurs, the flow is  
interrupted -  
(no need to have  
nested try-catch)



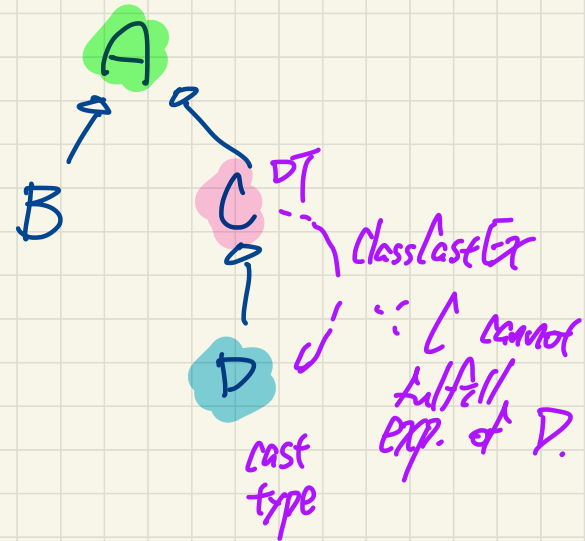
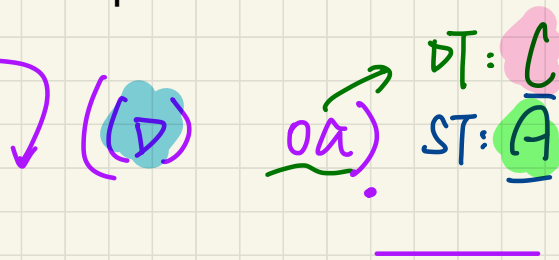
Past lab on recursion

↳ 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) to a, 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!

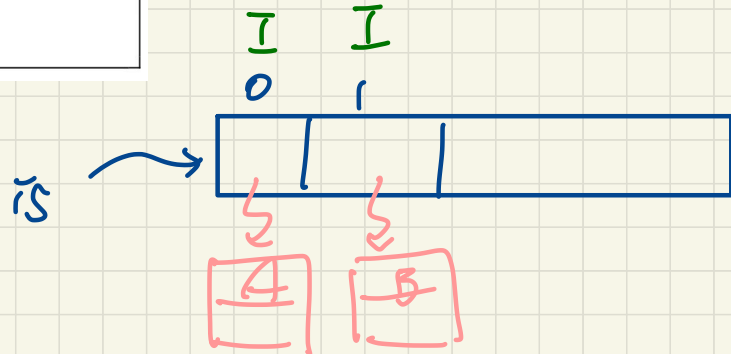
```

1 class Collector {
2 A[] as; int numberOfAs; I[] is;
3 B[] bs; int numberOfBs;
4 Collector() {
5 as = new A[10]; bs = new B[10];
6 void addA(A a) {
7 is as[numberOfAs] = a; numberOfAs++; }
8 void addB(B b) {
9 is 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 }

```

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

① c.addI(new A());  
 ② c.addI(new B());



I hope you enjoyed learning with me 



All the best to you! 