EECS2030 Fall 2022
Advanced Object-Oriented Programming

Lecture Notes

Instructor: Jackie Wang

<u>Lecture 1 - Sep. 7</u>

<u>Syllabus & Review on OOP</u>

Object Orientation
Classes vs. Objects on Eclipse

Course Learning Outcomes (CLOs)

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

iClicker.

- LaboP1

[ ]

↳ Eclipse (remotelabs)

↳ github (private).

↳ documents

↳ how to infer Java code from JUnit tests ([ ])

↳ $\boxed{\text{programming pattern}}$

↳ tutorial videos

submit exported project

IDE - Eclipse

integrated
development
environment

→ editor

→ debugger

```
class A {
    m(...) {
    
    }
}
```

this.a.c.d.e

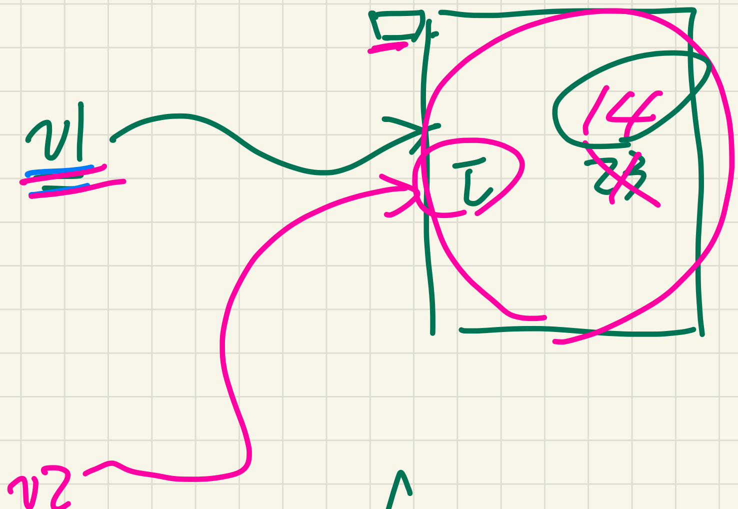
aliasing

v1.i = 14

v2.i 14.

EECS2031!

Java ref. variables are pseudo-pointer



C

i++ (pointer arithmetic)

int i = 23;
int j = 46;

i = j;

A v1
A v2
A v3

v1

v2

v3

i ~~23~~ ~~14~~ 67

i 46

v1.i
v2.i
23

v1.i = 14

v1.i = 14
v2.i = 14

v2 = v3
v1.i = 67

v1.i 67
v2.i 46

Lecture 2 - Sep. 12

Review on OOP

Object Orientation
Classes, Objects, Methods

- Lab0 Part 1
  + $\boxed{\text{Eclipse}}$: Your Machine vs. $\boxed{\text{RemoteLabs}}$
  + Tutorial Videos
  + PDF guides:
    * Inferring Java Classes from JUnit Tests
    * Programming Pattern: Array Attributes
- Scheduled Lab this Week: Optional Q&A
- Office Hours

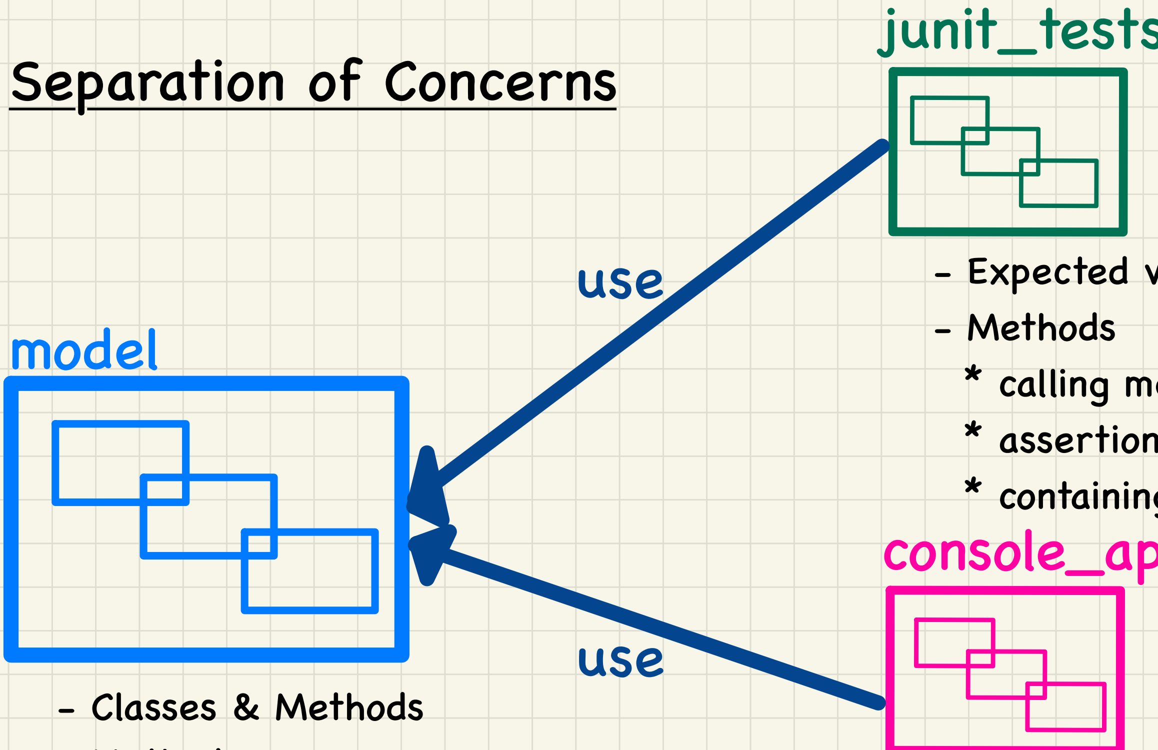
latest.

try.

eecs account.

Reading.. up to slide 49.

Separation of Concerns



junit_tests

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

model

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

attributes : should be private

methods : 1. helper methods : private

2. to be called by other classes:
public

class Person{

Attributes

}

default const. available

class Person{

atts.

Person(___ , ___){

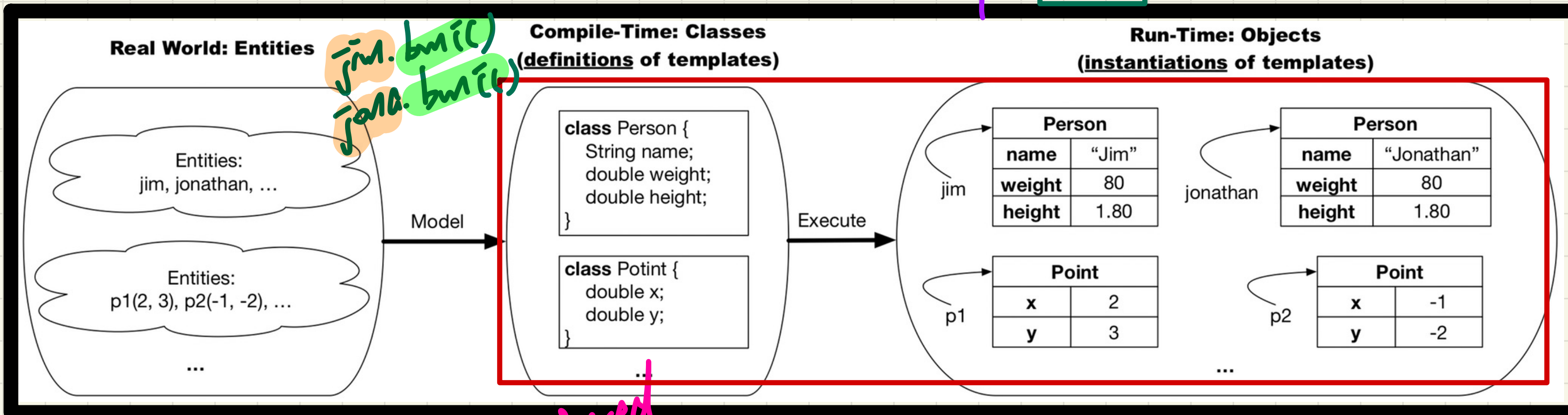
}

}

default const not. ava.

Observe-Model-Execute Process

Context objects p1 dist( )
p2 dist( )



shared by all Person objects

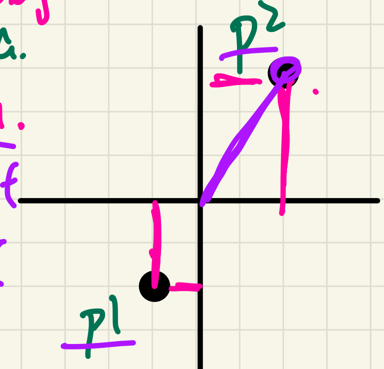
Entities: jim, jona.

Attributes: w., h.

Changes: gainWeight

Inquiries: getBMI

Template: Person



Entities: p1, p2

Attributes: x, y.

Changes: ↑x ↓y

Inquiries: dist

Template: Point

Modelling: from Entities to Classes

mutator/
setter

Identify Critical Nouns & Verbs

classes, attributes,

accessor/
getter

Example 1

Point
(class)

x, y
attributes

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

moveUp
Down
East West.

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.

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

Constructors not using this Keyword

model

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

    /*
     * Constructors
     */
    public Person() {

    }

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

JUnit

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

console

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

store address of some Person object

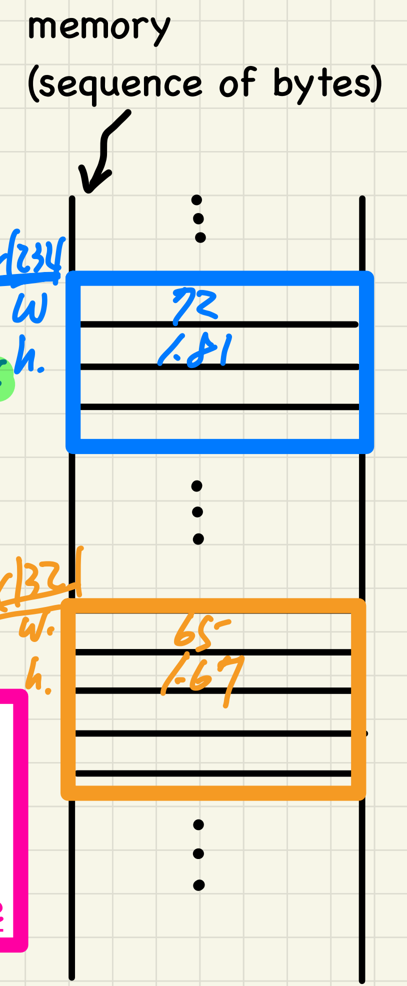
72 65 1.81 1.67

0x1234

jim

0x1324

jona.



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

Lecture 3 - Sep. 14

Review on OOP

Object Orientation
Tracing OO Programs, Aliasing, Arrays

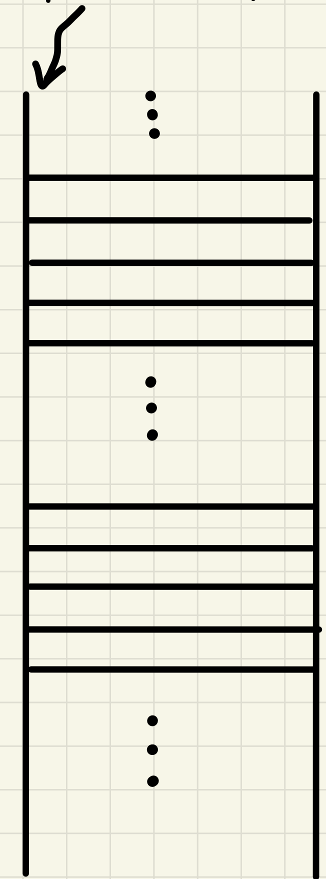
- Lab0 Part 1 Due Soon
- Lab0 Part 2 Released on Tuesday
- Lab1 to be released on Friday

1. ref. type
2. arrays

Constructors not using this Keyword

memory

(sequence of bytes)



model

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

    /*
     * Constructors
     */
    public Person() {

    }

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

define method

use method

JUnit

arg.

```
@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);
}
```

arg.

param.

console

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

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

Parameters vs. Arguments

Template Definition

```
class Point {
  Point(double x, double y) {…}

  double getDistanceFrom(Point other) {…}

  void move(char direction, double units) {…}
}
```

P.

P.

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);
  }
}
```

arg.

arg.

Q: Can parameters be used as arguments?

Can arg. be used as parameters? NO.

m (int i, . . .) {

param.

p1. m2 (i)

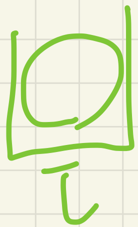
}

context objects

param. i is used as an argument to invoke method m2 char **

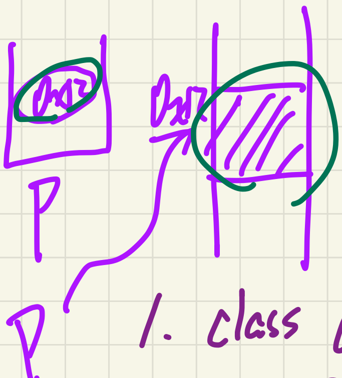
no correspondence in Java

boolean float double char



int i ; → primitive variable

at runtime, i can store an int. value



Person p ; → reference variable

1. class defined in your project
2. any Java library (String , ArrayList)

at runtime, p can store the address of a Person object

Constructors not using this Keyword

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

    /*
     * Constructors
     */
    public Person() {

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

Question

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

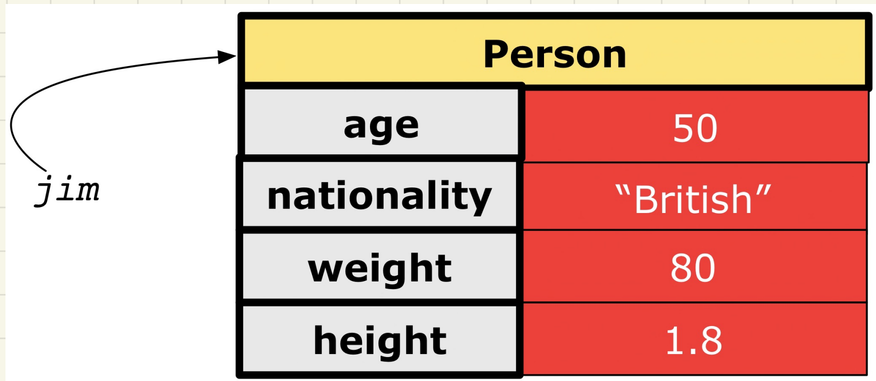
variable shadowing

Tracing OO Code: Visualizing Objects

Slides 24 - 28

To visualize an object:

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



Effects of Creating New Objects

model

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

    /*
     * Constructors
     */
    public Person() {

    }

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

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

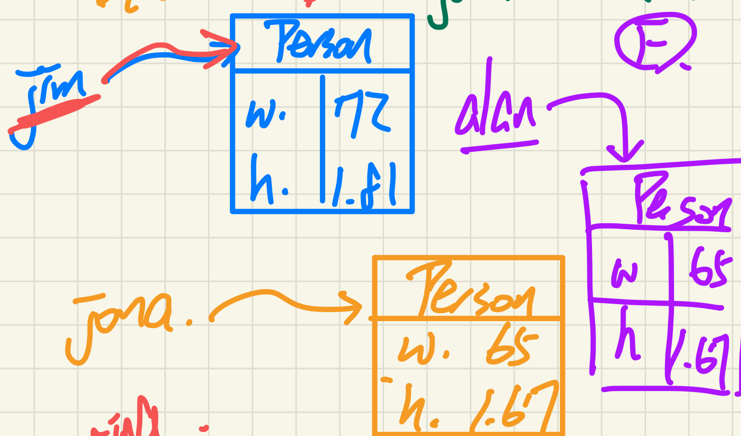
Person alan = new Person(65, 1.67);

JUnit

```
@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);
}
```

C.O.

jona. == alan



jona. weight = 65

address dereferencing (add. lookup)

BMI

$$\frac{\text{weight}^{\text{kg}}}{\text{height}^2}$$

meters.

Accessors/Getters

jim.getBMI();
c.o.

model

jona.getBMI()
c.o.

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

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

jim. jona.

c.o.s different

JUnit

expect

actual

tolerance (ε)

```java
@Test
public void test_2() {
    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);
}
```

same method called

jim

Person	
w.	72
h.	1.81

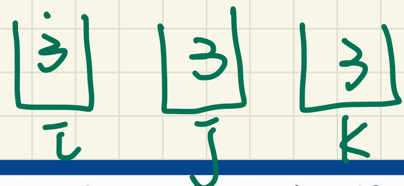
jonathan

Person	
w.	65
h.	1.67

Copying Primitive vs. Reference Values

Primitive

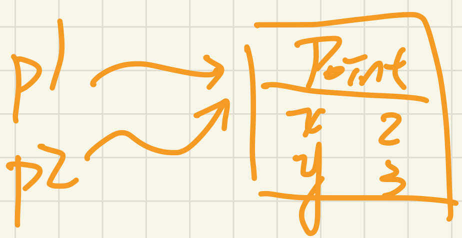
```
int i = 3;
int j = i;    System.out.println(i == j);/*true*/
int k = 3;    System.out.println(k == i && k == j);/*true*/
```

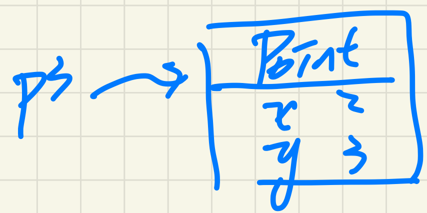


Reference

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

F F





Exercise

Person[] persons1; → stores the beginning address of the array

each index of the array stores the address of some Person object

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

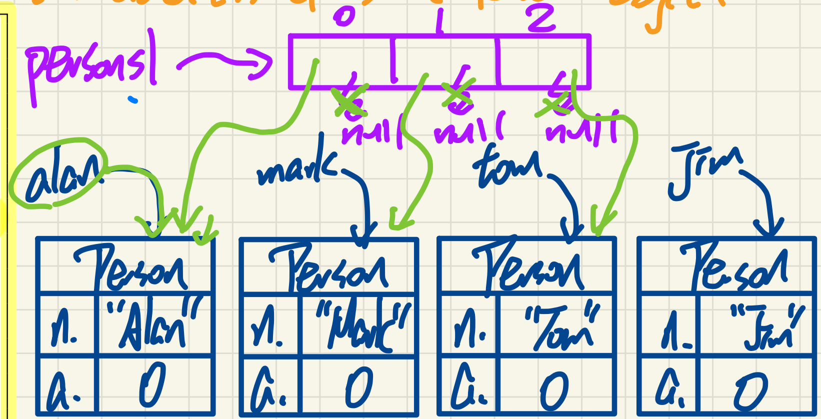
name
age

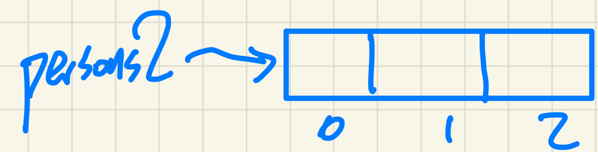
3

3

0
1
2
iterations

size of array





Person[] persons1 = new Person[3]; ✓

→ persons1[0] = alan; // copy add. stored in alan to index 0.
→ persons1[1] = mark;
→ persons1[2] = tom;

1st iteration

persons2[0] = persons1[0];

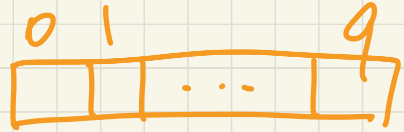
%

Review

↓

remainder

/ modulo op.



Person[] ps = new Person[MAX]; (MAX = 10)

① MAX (10) indices in the array

② Range of indices: 0 .. MAX-1.

③ ps.length == MAX

largest index: ps.length-1

Lecture 4 - Sep. 19

Review on OOP

Tracing OO Programs, Aliasing, Arrays
Attributes/Parameters/Return Types
Anonymous Objects

1. deadline
2. prog. req.

Announcements

- Lab1 released (scheduled lab sessions & office hours)
- Lab0 Part 2 Due on Friday
- WrittenTest1
  (**make sure** you try logging into **eClass** in WSC)
- ProgTest1

Exercise

Person[] persons1;

stores the beginning address of the array

each index of the array stores the address of some Person object

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

name

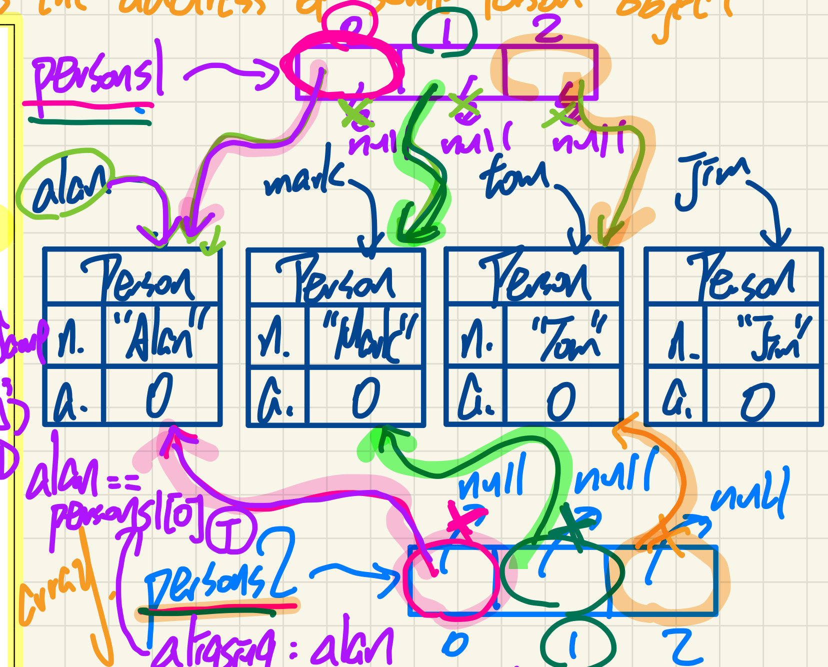
age

3

3 iterations: 0, 1, 2

assertSame(alan, p1[0])

alan == persons1[0]



Person[] persons1 = new Person[3]; ✓

→ persons1[0] = alan; // copy add. stored in alan to index 0.

→ persons1[1] = mark;

→ persons1[2] = tom;

1st iteration persons2[0] = persons1[0];

2nd persons2[1] = persons1[1];

3rd persons2[2] = persons1[2];

skip it

Exercise

Person[] persons1[]; → stores the beginning address of the array

each index of the array stores the address of some Person object

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

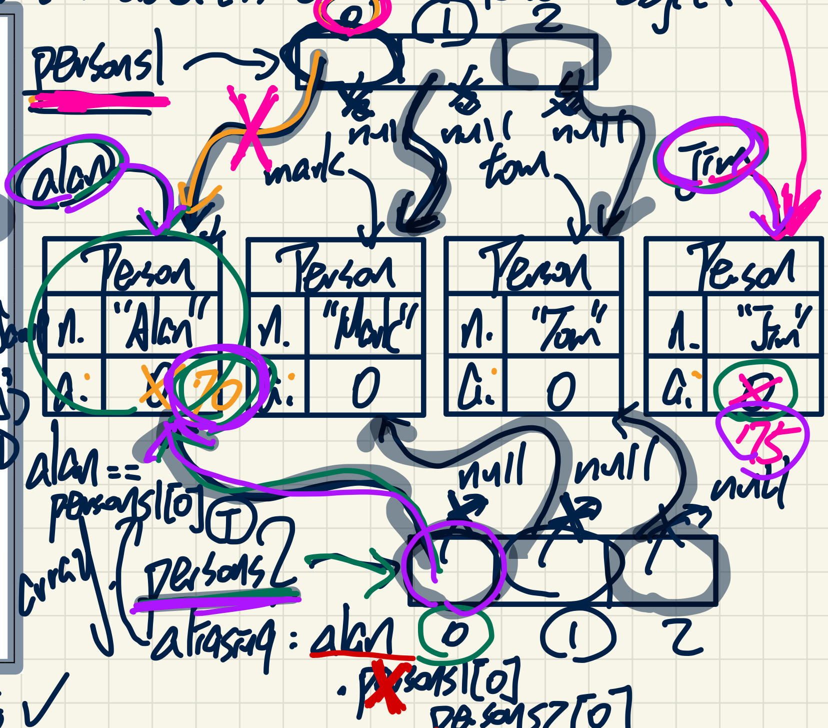
name
age

3

0 1 2 iterations

0
70
70

75
70
70



Person[] persons1 = new Person[3]; ✓

→ persons1[0] = alan; // copy add. stored in alan to index 0.

→ persons1[1] = mark;

→ persons1[2] = tom;

alan.toString() → address

Accessors/Getters vs. Mutators/Setters

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

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

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

jim jim jim

jon. jon. jon.

jim jon

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

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

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

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

jim

Person	
w.	~~72~~ 75
h.	1.81

jonathan

Person	
w.	~~65~~ 68
h.	1.67

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

Use of Accessors vs. Mutators


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

intend to use the mutator call as the argument value

- Calls to *mutator methods* *cannot* be used as values.
  - 1 e.g., `System.out.println(jim.setWeight(78.5));` void X
  - 2 e.g., `double w = jim.setWeight(78.5);` void X
  - 3 e.g., `jim.setWeight(78.5);` ✓
- Calls to *accessor methods* *should* be used as values.
  - 4 ✓ e.g., `jim.getBMI();`
  - 5 ✓ e.g., `System.out.println(jim.getBMI());` ✓
  - 6 e.g., `double w = jim.getBMI();`

Method Parameters

mainly for private helper methods

- **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.

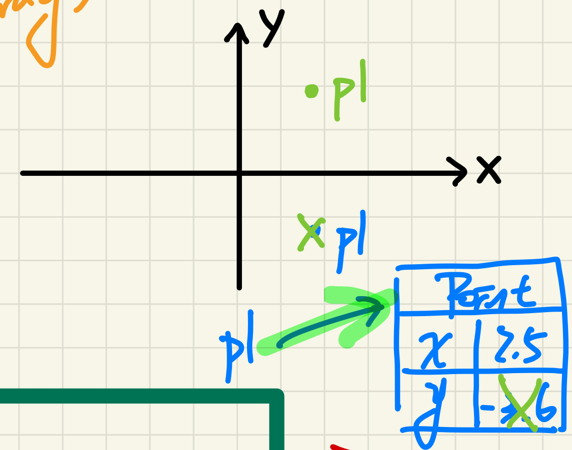
  p1.getDFO(); p1.getDF(p2);

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

Reference-Typed Return Values

class MyClass {
  — att;
  ~~type~~
  1. primitive
  2. ref type
    ↳ single-valued
    ↳ multi-valued (array)
}

class Person {
  Person spouse;
  Person[] children;
}



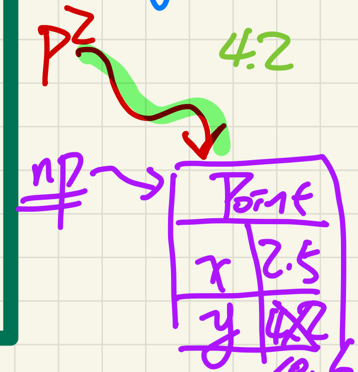

```
public class Point {
public void moveUpBy(int i) { y = y + i; }
Point movedUpBy(int x) {
  Point np = new Point(x, y);
  np.moveUp(x);
  return np;
}
}
```

modify this

- does not modify this
- modify some local var.

```
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);
  }
}
```

return np



Anonymous Objects


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

name

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

Anonymous exp.

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

name

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

anonymous obj.

Lab0 P2

overloading - **Exercise**

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

① treat this as a helper method.

this.addOrder(new Order(- - -));

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

this.orders[this.noo] = o;

this.noo++;

dup.

this.addOrder(o);

Lecture 5 - Sep. 21

Review on OOP

More Advanced Use of this
Static Variables

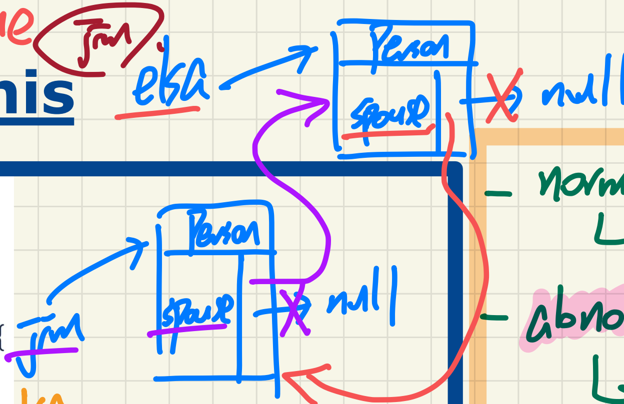
Announcements

- Lab1 released (scheduled lab sessions & office hours)
- Lab0 Part 2 Due on Friday
- WrittenTest1 → WSC
  - make sure you try logging into eClass in WSC
  - A guide and some practice questions released soon
- Programming Test 1 (60 to 65 min)

  WSC
  - Identical format as Lab1
  - Number of starter tests will be smaller
  - Guide, Practice Test, Mockup Test to be announced

jim.spouse.spouse.spouse.spouse.name (Jim)

Example: Reference to this




```
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
  public void marry(Person other) {
    if(this.spouse != null || other.spouse != null) {
      /* Error: both must be single */
    }
    else { this.spouse = other; other.spouse = this; }
  }
}
```

- normal
  ↳ marry
- abnormal
  ↳ e.g. can't marry one to themselves
  e.g. can't marry someone not single

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

c.o. / arg.

① jim != elsa — other

```
if (this == other
    || (this.spouse != null
    || other.spouse != null)) { ... }; // error
else {

}
```

✓

jim   elsa

~~this~~.spouse = ~~other~~;

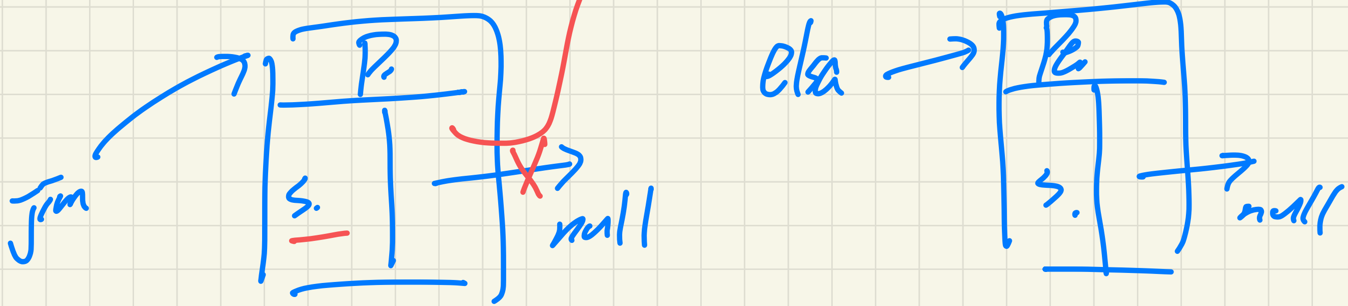
~~other.spouse = this;~~

T

① jim.spouse == elsa

② elsa.spouse == jim

F.



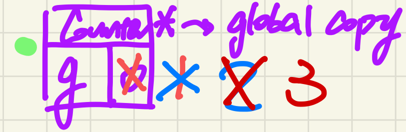
Managing Account IDs: Manual

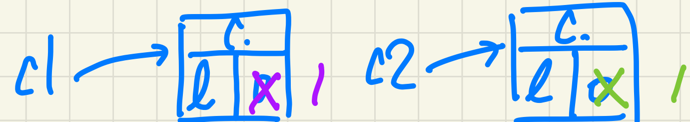

```
public class Account {
  private int id;
  private String owner;
  public int getID() { return this.id; }
  public Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

manual management of id's

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

Declaring Global Variables among Objects





static ↳ all Counter objects share the same copy.

non-static ↳ each Counter obj has its own copy

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

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

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

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

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

static att not initialized here

this.g++; ↳ not an error ↳ warning

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

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

        c1.incrementLocal();

        c2.incrementLocal();

        c1.incrementGlobal();

        c2.incrementGlobal();

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

warning: static var. should not be specific to an obj.

not necessary to create a C.O. for this

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

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

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

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

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

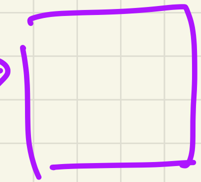
init. done only once

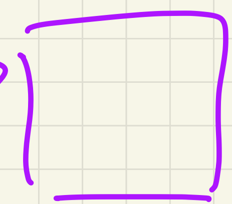
executed whenever a new Counter obj is created

Counter	
g	~~l~~

* 2

```java
println( Counter.g );      // 0
Counter c1 = new Counter();
println( Counter.g );      // 1
Counter c2 = new Counter();
println( Counter.g );      // 2
```

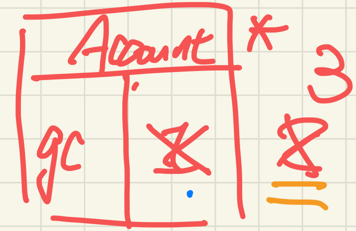
c1 → 

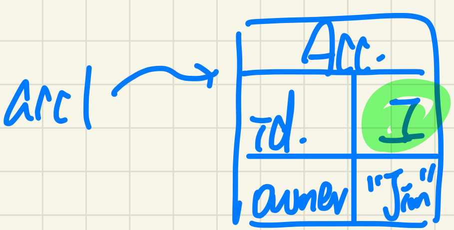
c2 → 

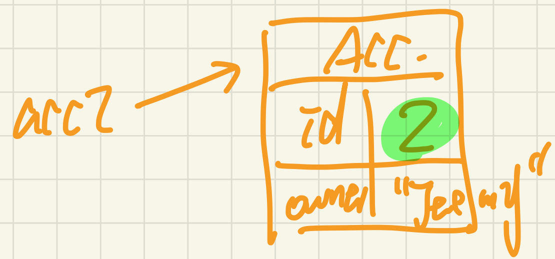
Managing Account IDs: Automatic

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

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







Misuse of Static Variables

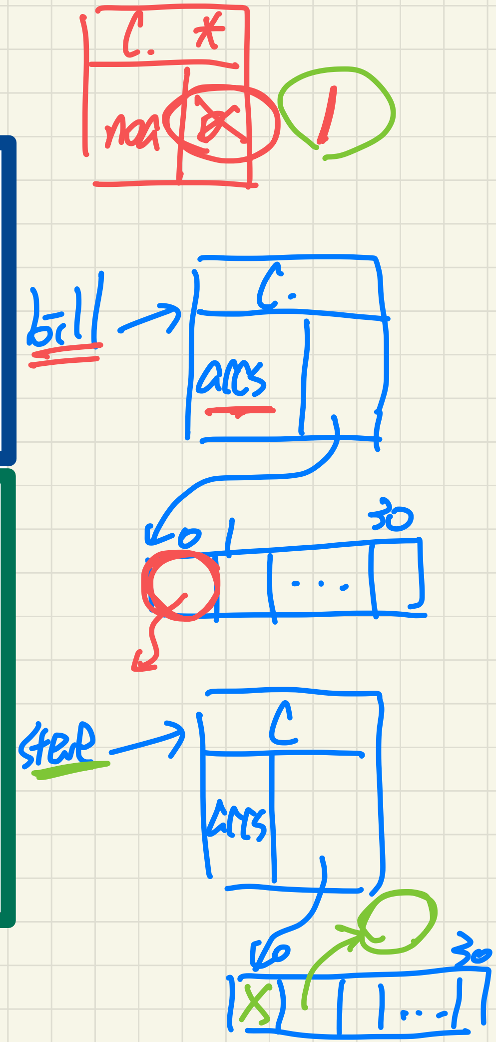
```java
public class Client {
  private Account[] accounts;
  private static int numberOfAccounts = 0;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  } }
```

bill steve

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

bill.addAccount(acc3);
steve.addAccount(acc4);

Exercise



Lecture 6 - Sep. 26

Review on OOP, Exceptions

Static Variables: Common Error
Caller vs. Callee
Error Handling using Console Messages

Announcements

- Lab1 due at 2pm this Wednesday
- WrittenTest1
  - make sure you try logging into eClass in WSC
  - A guide and some practice questions released
- Programming Test 1 (60 to 65 min)
  - Identical format as Lab1
  - Number of starter tests will be smaller
  - Guide, Practice Test, Mockup Test to be announced

Use of Static Variables: Common Error

```
public class Bank {
    private string branchName;
    public String getBrachName() { return this.branchName; }
    private static int nextAccountNumber = 0;
    public static String getInfo() {
        nextAccountNumber++;
        return this.branchName + nextAccountNumber;
    }
}
```

Branch

① ②

cannot be just replaced by class name

String s = "York" + 50

non-static
⇒ must have some C.O. to replace this

Use: Bank.getInfo()

(solution 1)

static

```
public class Bank {
    private string branchName;
    public String getBrachName() { return this.branchName; }
    private static int nextAccountNumber = 0;
    public static String getInfo() {
        nextAccountNumber++;
        return this.branchName + nextAccountNumber;
    }
}
```

non-static branch name ⇒ each Bank obj has its own branch

local

static branch name ⇒ all Bank objects share the same branch

global

↳ doesn't make real sense

(Solution 2).

```java
public class Bank {
    private string branchName;
    public String getBrachName() { return this.branchName; }
    private static int nextAccountNumber = 0;
    public static String getInfo() {
        nextAccountNumber++;
        return this.branchName + nextAccountNumber;
    }
}
```

Exercise.

Caller vs. Callee

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

caller

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

context of calling m2 from C2

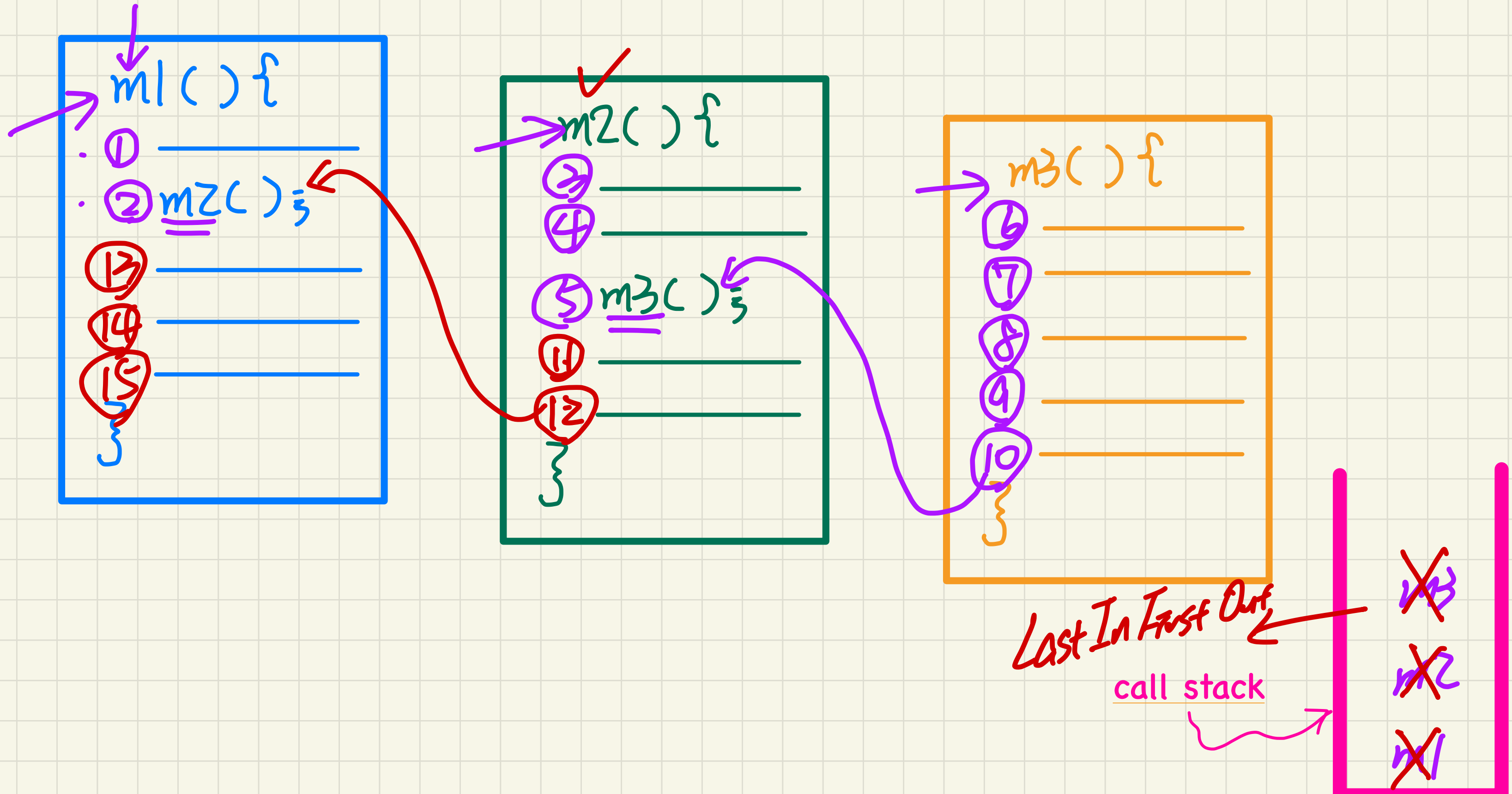
method being used (callee

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

① class C3 {
  void m3() {
    C1 o = new C1(); o.m1();
  }
}

② class C2 {
  void m2() {
    C3 o = new C3();
    o.m3();
  }
}

Visualizing a Call Chain using a Stack



Error Handling via Console Messages: Circles

```java
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 }
```

-10 -10 -10

exit;

can disrupt L5 & L6

Caller?

Callee?

```java
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 }
```

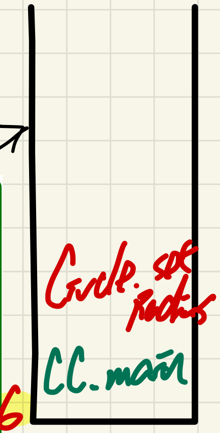
to continue

but it would not allow caller to handle the error (e.g. enter another #).

print error but would not stop the execution of L5 & L6

for error handling to be acceptable, these lines should here not be allowed to continue.

call stack



Error Handling via Console Messages: Banks

```java
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; }
  }
}
```

```java
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 );
      }
    } /
  } /*
}
```

Account

C.O.

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

Caller?

Callee?

call stack

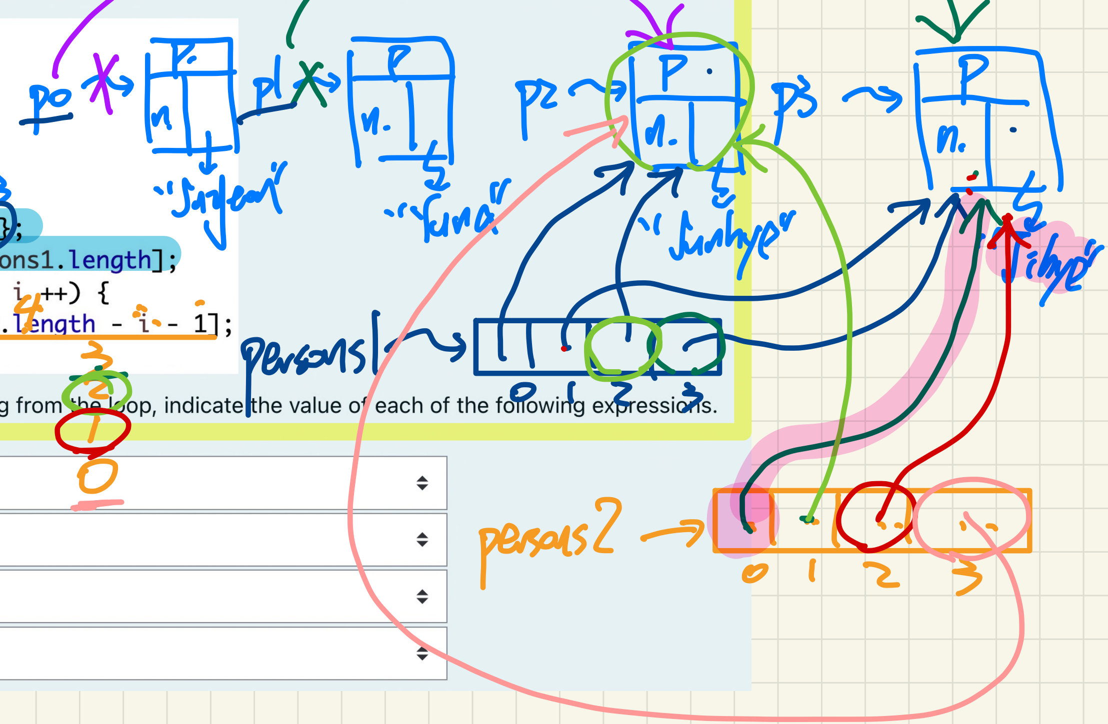
- Account withdraw
- Bank withdrawFrom
- BA.main

context	caller	callee
BA	main	Bank withdrawFrom
Bank	withdrawFrom	Account withdraw
Account	withdraw	N.A.

Practice Written Test 1

Assume that a Person class is already defined, and it has an attribute name, a constructor that initializes the person's name from the input string, and an accessor `getName` returning the person's name. Consider the following fragment of Java code (inside some main method):

```java
Person p0 = new Person("Suyeon");
Person p1 = new Person("Yuna");
Person p2 = new Person("Sunhye");
Person p3 = new Person("Jihye");
p0 = p2;
p1 = p3;
Person[] persons1 = {p0, p1, p2, p3};
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons2.length; i ++) {
    persons2[i] = persons1[persons2.length - i - 1];
}
```



Executing the above fragment of code, after exiting from the loop, indicate the value of each of the following expressions.

persons2[0].getName()	Choose... Jihye
persons2[1].getName()	Choose...
persons2[2].getName()	Choose...
persons2[3].getName()	Choose...

Practice Written Test 1

Assume a \`Person\` class declared with: a string attribute \`name\` and a constructor initializing that string attribute using the input parameter.

Now consider the following fragment code which implements the \`main\` method of some console application class:

```java
Person p1 = new Person("Alan");
Person p2 = new Person("Mark");
Person p3 = new Person("Alan");
Person p4 = p2;
p2 = p1;
p1 = p4;
p4 = p3;
p3 = p1;
System.out.println("Done!");
```

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

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

Lecture 7 - Sep. 28

Exceptions

To Handle or Not to Handle?
Error Handling using Exceptions

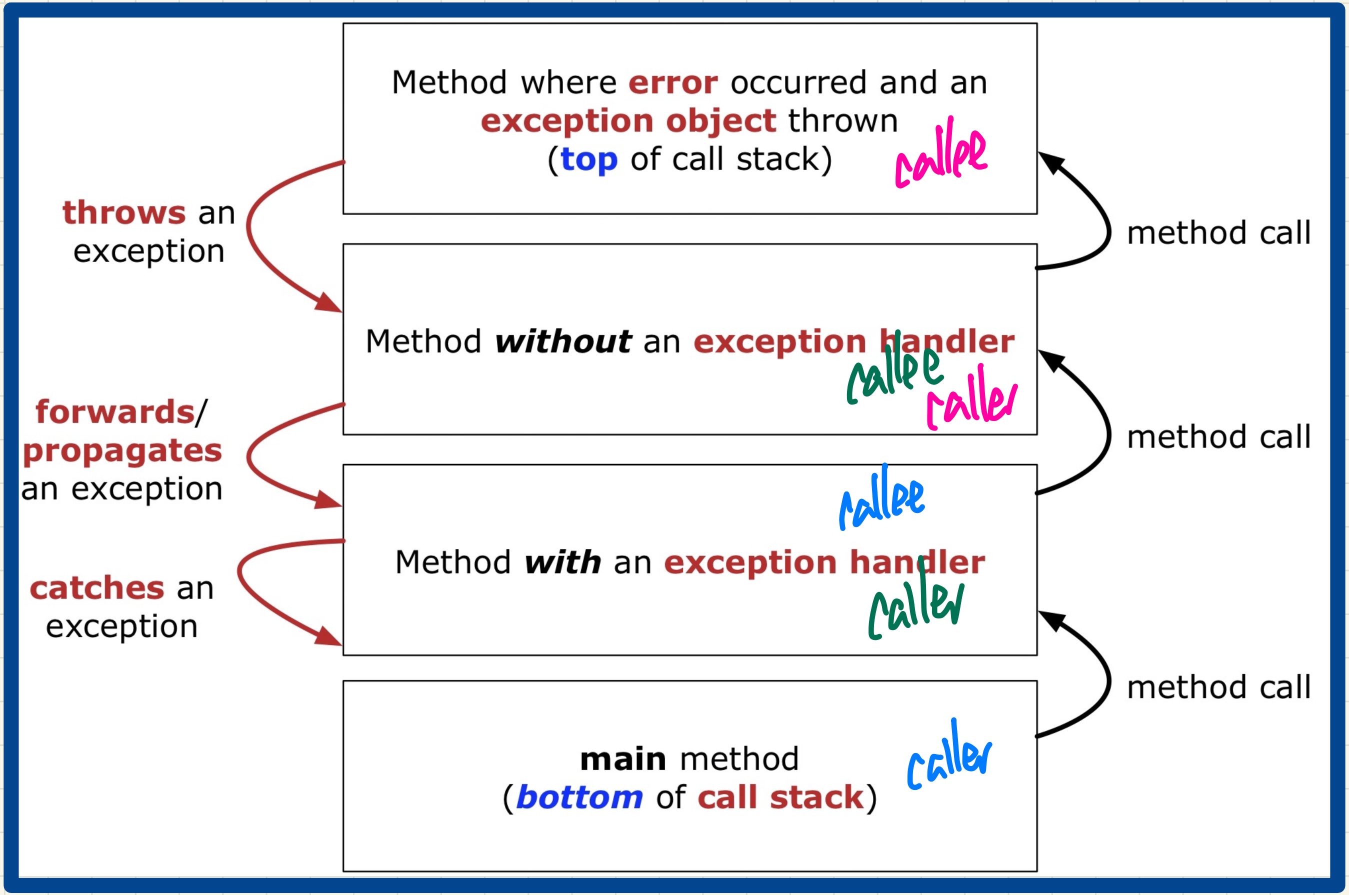
Announcements

- Lab1 due at 2pm today (Wednesday)
- WrittenTest1
    - Marks to be released on Friday
    - Visit my office hours to discuss questions if you wish
- Programming Test 1
    - Guide & Practice Test to be released (bteo Thursday)
    - A Short Mockup Test to be arranged

Exception Handler

```
try {
    ________
}
catch( ________ ) {
    ________
}
catch( ________ ) {
    ________
}
```

What to Do When an Exception is Thrown: Call Stack



Catch-or-Specify Requirement

to handle

The "Catch" Solution: A `try` statement that ***catches*** and ***handles*** the ***exception*** (**without** propagating that exception to the method's *caller*).

```
main(...) {
  Circle c = new Circle();
  try {
    c.setRadius(-10);
  }
  catch(NegativeRaidusException e) {
    ...
  }
}
```

has the potential of throwing an exception

how to handle that exception.

The "Specify" Solution: A method that specifies as part of its *header* that it may (or may not) ***throw*** the ***exception*** (which will be thrown to the method's ***caller*** for handling).

```
class Bank {
  Account[] accounts; /* attribute */
  void withdraw (double amount)
    throws InvalidTransactionException {
    ...
    accounts[i].withdraw(amount);
    ...
  }
}
```

1. some line in the body of imp may throw an exception

2. that excep from thrown will not be handle in current method.

Example: To Handle or Not To Handle?

context	caller	callee
Tester	main	B.mb
B	mb	A.ma
A	ma	n.a.

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

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

```java
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?  */
  } }
```

```java
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

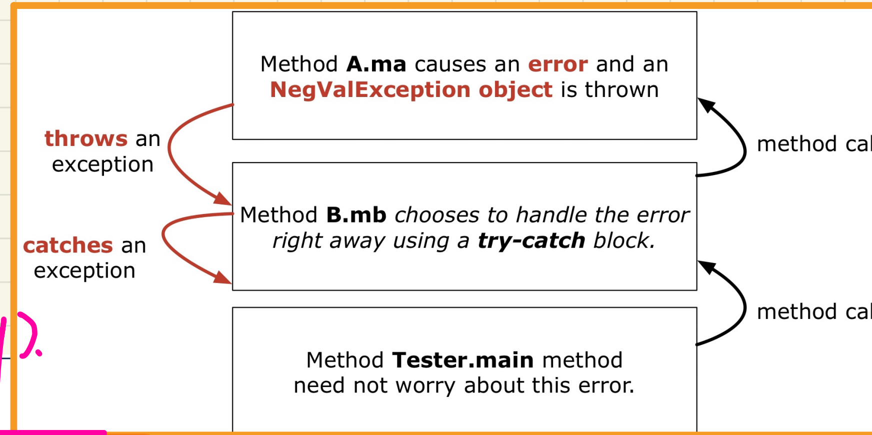
Version 1:

Handle the Exception in B.mb

to satisfy the catch or specify req. (specify).

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

this is where the error occurred



Normal : 20

Abnormal : -10

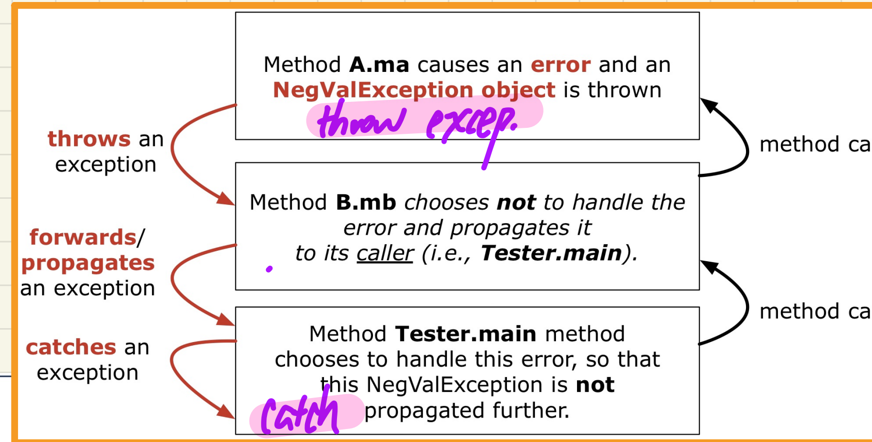
```
class B {
  mb(int i) {
    A oa = new A();
    try { oa.ma(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

throw NVE.

```
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



abnormal input: -10

```
class A { -10
  ma(int i) throws NegValException {      specify
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
} }
```

```
class B { -10
  mb(int i) throws NegValException {      specify
    A oa = new A();
    oa.ma(i); -> throws NVE
          -10
} }
```

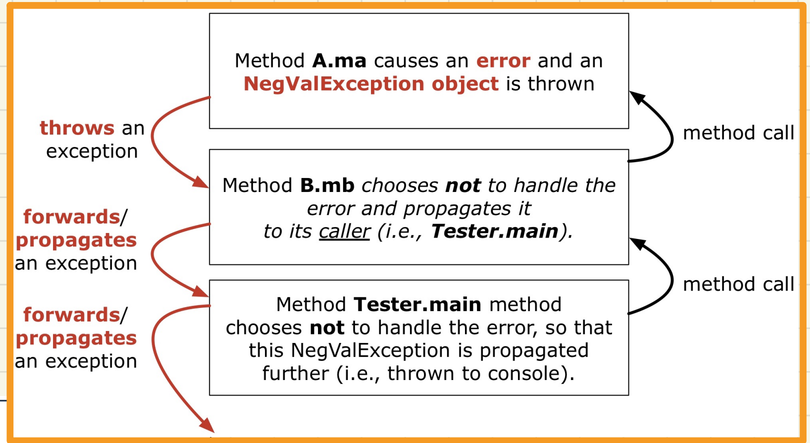
```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt(); -10
    B ob = new B();
    try { ob.mb(i); } -10
    catch(NegValException nve) { /* Do something. */ }
} }
```

this is where the exception gets handled.

exception handler

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. */ }
 } }
```

-20

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

specify

-20

```
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);
 } }
```

specify

-20

prop. to terminal

abnormal input: -20.

Lecture 8 - Oct. 3

Exceptions, TDD

Using Exceptions: Circles & Banks
Catching Multiple Exceptions
More Advanced Use of Exceptions

Announcements

- WrittenTest1 Marks Released
    - Visit my office hours to discuss questions if you wish
- Programming Test 1 (tomorrow, Tuesday)
    - Guide & Practice Test released
    - Arrange as many mock-up tests as you can
- Lab2 to be released shortly after PT1

Recap of Exceptions

- Catch-or-Specify Requirement

Normal Flow of Execution

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

When the exception does not occur

Abnormal Flow of Execution

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

When the exception occurs

Error Handling via Exceptions: Circles (Version 1)

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

```
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; }
}
```

specify

where the excep. is originated

⇒ (typically) just do specify.

Test Case 1:
User enters 10

Test Case 2:
User enters -5

```
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);
    }
  }
}
```

10 -5 → throw IRE

① Reaching this line means the IRE did not occur

② Not reaching this line means the IRE occurred

Error Handling via Exceptions: Circles (Version 2)

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

T

TRIV

Test Case:

User enters -5

Then user enters 10

```java
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; }
}
```

-5 10

as long as it's not the case that the last user-entered input is valid, keep executing body of loop.

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

initially, no valid radius entered

-5 10

no IRE thrown

IRE thrown

Enter a radius:
-5
Try again!
Enter a radius:
Circle with ...

① throw IRE if r < 0

② otherwise, no IRE thrown

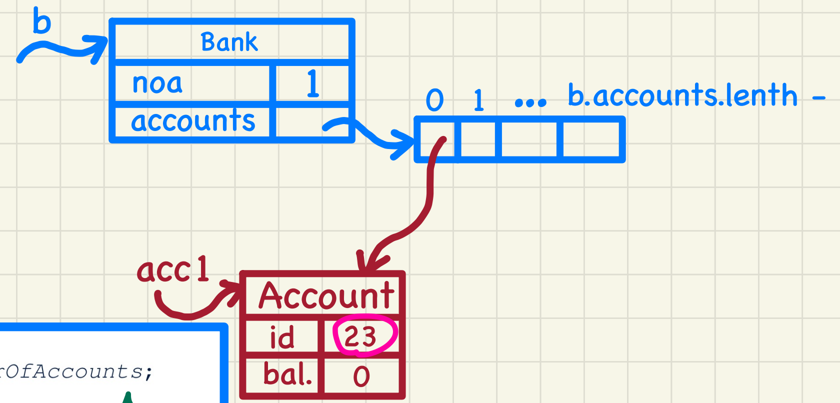
Error Handling via Exceptions: Banks

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

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

-5M

specify.




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

specify

-5M → may throw ITE

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

-5M

-5M → throw ITE → may throw ITE.

1. paper
2. Eclipse

Exercise: try input 20

Test Case:

User enters -5000000

More Example: Multiple Catch Blocks

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

100 -5M throws ITE

does not throw ITE

-5 throws NRE

more than one exceptions might be thrown

Test Case 1:

a: -5000000

r: 23

Test Case 2:

a: 100

r: -5

removing this block causes error: NRE not handled.

More Example: Parsing Strings as Integers

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

Test Case:

User Enters: twenty-three

User Then Enters: 23

"23" "twenty-three"

"23" "twenty-three"

throws NFE

NFE not thrown

may throw NFE.

Exercise: Type & debug in Eclipse!

X

T

vI

Enter an int:
twenty-three
Not valid.

Enter an int:
23

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

specify in where the exception is originated

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

Example 2: Method that calls another which throws the exception

specify.

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

may throw VTSE

Review: Specify-or-Catch Principle

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

throws VTSE

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

may throw VTSE

Error ∵ exception already handled.

catch option

Let's try to take your attendance:

A) I am here.
B) I am here.
C) I am here.
D) I am here.
E) I am here.
F) I am here.

Lecture 9 - Oct. 5

Testing Exceptions & TDD

Testing Exceptions: Console Testers
Testing Exceptions: JUnit Tests

Announcements

- Programming Test 1
- Lab2
- Reading Week

A Class for Bounded Counters

```java
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!
```

```java
/* 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 --; }
  }
}
```

Correct

Correct.

Manual Tester 1 from the Console

```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```

What if decrement is implemented correctly?

Expected Behaviour:

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


```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```


What if decrement is implemented incorrectly?

e.g., It only throws VTSE when c.value < 0

Running Console Tester 1 on Correct Implementation

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

Imp. (Correct)

```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter(); 0
    println("Init val: " + c.getValue());
    try {
      c.decrement(); -> throws VTSE
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```

test.

0

Running Console Tester 1 on Incorrect Implementation

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

-1

imp. (wrong).

```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```

expected VTSE not thrown

Manual Tester 2 from the Console

Test Case 3

- Nothing unexpected occurs.
- Everything expected occurs.

Test Case 1

VTLE thrown unexpectedly

Test Case 2

VTLE <u>not</u> thrown as expected

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try {
     c.increment(); c.increment(); c.increment();
     println("Current val: " + c.getValue());
     try {
      c.increment();
      println("Error: ValueTooLargeException NOT thrown.");
     } /* end of inner try */
     catch (ValueTooLargeException e) {
      println("Success: ValueTooLargeException thrown.");
     } /* end of inner catch */
    } /* end of outer try */
    catch (ValueTooLargeException e) {
     println("Error: ValueTooLargeException thrown unexpectedly.");
    } /* end of outer catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

no VTLE expected

0 1 2 3

VTLE expected

Running Console Tester 2 on (Correct) Implementation 1

3 == 3

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

Correct.

```java
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try {
      c.increment(); c.increment(); c.increment();
      println("Current val: " + c.getValue());
      try {
        c.increment();
        println("Error: ValueTooLargeException NOT thrown.");
      } /* end of inner try */
      catch (ValueTooLargeException e) {
        println("Success: ValueTooLargeException thrown.");
      } /* end of inner catch */
    } /* end of outer try */
    catch (ValueTooLargeException e) {
      println("Error: ValueTooLargeException thrown unexpectedly.");
    } /* end of outer catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

0 1 2 3

throw VTLE

Running Console Tester 2 on (Incorrect) Implementation 2

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

<

Incorrect Imp.

0

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try {
      c.increment(); c.increment(); c.increment();
      println("Current val: " + c.getValue());
      try {
        c.increment();
        println("Error: ValueTooLargeException NOT thrown.");
      } /* end of inner try */
      catch (ValueTooLargeException e) {
        println("Success: ValueTooLargeException thrown.");
      } /* end of inner catch */
    } /* end of outer try */
    catch (ValueTooLargeException e) {
      println("Error: ValueTooLargeException thrown unexpectedly.");
    } /* end of outer catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

0

throws VTLE

0

Running Console Tester 2 on (Incorrect) Implementation 3

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

incorrect

>

```java
public class CounterTester2 {
 public static void main(String[] args) {
  Counter c = new Counter();
  println("Current val: " + c.getValue());
  try {
   c.increment(); c.increment(); c.increment();
   println("Current val: " + c.getValue());
   try {
    c.increment();
    println("Error: ValueTooLargeException NOT thrown.");
   } /* end of inner try */
   catch (ValueTooLargeException e) {
    println("Success: ValueTooLargeException thrown.");
   } /* end of inner catch */
  } /* end of outer try */
  catch (ValueTooLargeException e) {
   println("Error: ValueTooLargeException thrown unexpectedly.");
  } /* end of outer catch */
 } /* end of main method */
} /* end of CounterTester2 class */
```

4 → expected VTLE not thrown

Exercise

say: incorrect so that VTLE thrown prematurely.

Question. Can this alternative to `ConsoleTester2` work (<u>without</u> nested `try-catch`)?

```
public class CounterTester2 {
  public static void main(String[] args) {
   Counter c = new Counter();
   println("Current val: " + c.getValue());
   try {
    c.increment(); c.increment(); c.increment();
    println("Current val: " + c.getValue());
   }
   catch (ValueTooLargeException e) {
    println("Error: ValueTooLargeException thrown unexpectedly.");
   }
   try {
    c.increment();
    println("Error: ValueTooLargeException NOT thrown.");
   } /* end of inner try */
   catch (ValueTooLargeException e) {
    println("Success: ValueTooLargeException thrown.");
   } /* end of inner catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

not skipped even if an error has been identified

if this line was also printed: contradiction!

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

A Manual, Iterative Console Tester

```java
import java.util.Scanner;
public class CounterTester3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String cmd = null; Counter c = new Counter();
    boolean userWantsToContinue = true;
    while(userWantsToContinue) {
      println("Enter \"inc\", \"dec\", or \"val\":");
      cmd = input.nextLine();
      try {
        if(cmd.equals("inc")) { c.increment() ; }
        else if(cmd.equals("dec")) { c.decrement() ; }
        else if(cmd.equals("val")) { println( c.getValue() ); }
        else { userWantsToContinue = false; println("Bye!"); }
      } /* end of try */
      catch(ValueTooLargeException e){ println("Value too big!"); }
      catch(ValueTooSmallException e){ println("Value too small!"); }
    } /* end of while */
  } /* end of main method */
} /* end of class CounterTester3 */
```

may throw VTLE

may throw VTSE

JUnit: Where an Exception is Expected (1)

JUnit Test

```java
@Test
public void testDecFromMinValue() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.decrement();
    fail("ValueTooSmallException is expected.");
  }
  catch(ValueTooSmallException e) {
    /* Exception is expected to be thrown. */
  }
}
```

do nothing; ∴ a JUnit test without (1) assertion failure or (2) exceptions would pass?

fail the current test right away

Console Tester

```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```

Lecture 10 - Oct. 17

Testing Exceptions & TDD

Console Testers vs. JUnit Tests
Regression Testing

Announcements

- Programming Test 1 Results: by the middle of next week
- Lab2 due this Friday
- Look ahead: WrittenTest2 & ProgTest2

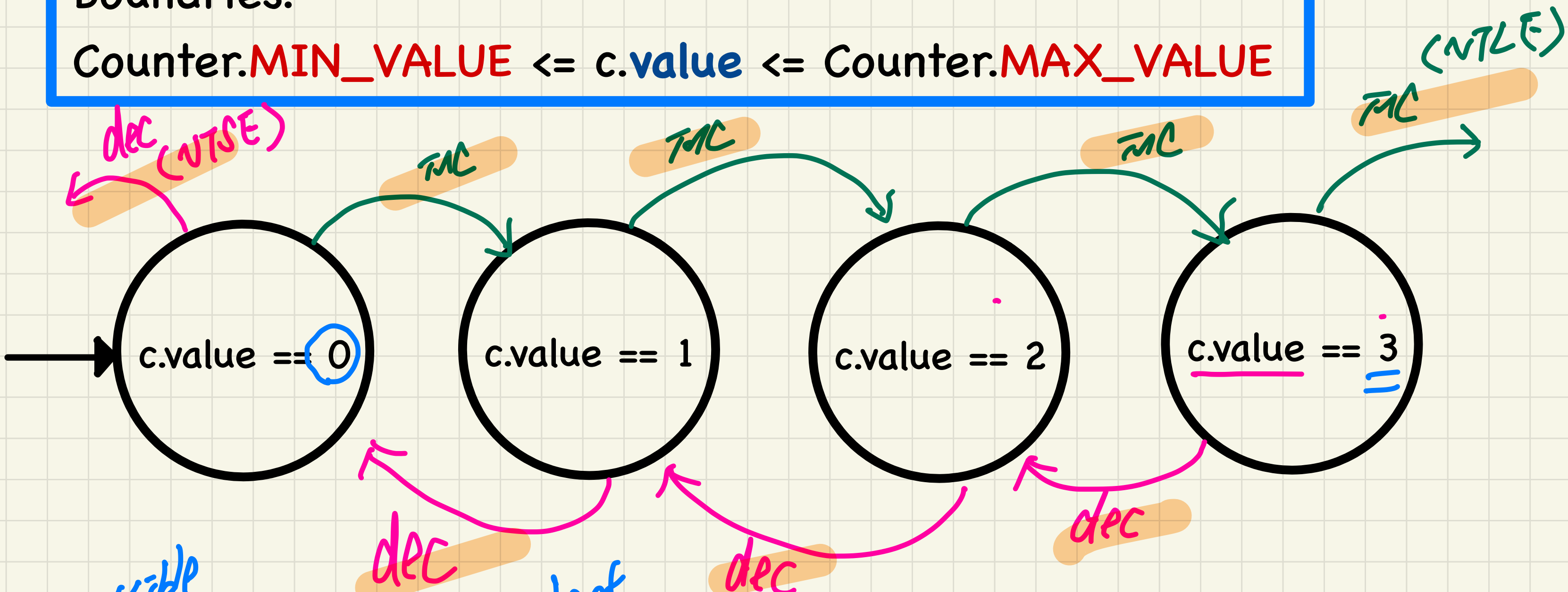
Coming Up with Test Cases: A Single, Bounded Variable

state transition diagram

EECS2001

Boundries:

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



4 possible states.

equivalent classes

c.increment()

c.decrement()

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; or
  - an *unexpected* exception (e.g., `NullPointerException`, `ArrayIndexOutOfBoundException`) is thrown.
- ***Success*** if neither assertion failures nor *unexpected* exceptions occur.

for a test to fail:
(1) exception or
(2) assertion failure.

e.g. ① NullPointerE.
② AIOBE

TestCounter.java

```java
package tests;
import static org.junit.Assert.*;
import org.junit.Test;
public class TestCounter {
  @Test
  public void test() {
    fail("Not yet implemented");
  }
}
```

disrupts exec. flow.

Compare with: System.out.println ("Error: ...")

(exec. flow not disrupted)

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

① NullPointerException

target object == null

________ . m(...)

<u>not</u> relevant

obj.m(...)

null ②

obj.att1.att2.m(...)

null ①

null ③

② IndexOutOfBoundsException

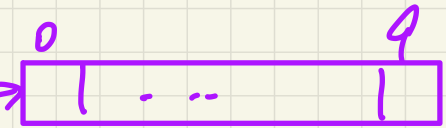
$a[i]$ → any integer expression

e.g. i

e.g. $i-1$

array

IOBE occurs if:



a

length 10

① $i < 0$ || $i >= a.length$

what if changing this to && ? (Exercise)

change this to || ?

✓

② $!(0 <= i$ && $i < a.length)$

Examples: JUnit Assertions (1)

Consider the following class:

```
class Point {
  int x; int y;
  Point(int x, int y) { this.x = x; this.y = y; }
  int getX() { return this.x; }
  int getY() { return this.y; }
}
```

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

```
Point p;
assertNull(p);  ✓                         null
assertTrue(p == null);   ✓
assertFalse(p != null);   ✓
assertEquals(3, p.getX());   × /* NullPointerException */
p = new Point(3, 4);
assertNull(p); [redacted]
assertTrue(p == null); [redacted]
assertFalse(p != null); [redacted]
assertEquals(3, p.getX()); [redacted]
assertTrue(p.getX() == 3 && p.getY() == 4); [redacted]
```

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);
assertTrue(36.2984, c.getArea(), 0.01);   ✓
```

Equals

Slides 38 - 39

JUnit: Where an Exception is Not Expected

```java
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment();
    assertEquals(1, c.getValue());
  }
  catch(ValueTooBigException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooBigException is not expected.");
  }
}
```

no exception thrown

What if increment is implemented correctly?

```java
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment();
    assertEquals(1, c.getValue());
  }
  catch(ValueTooBigException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooBigException is not expected.");
  }
}
```

exception thrown unexpectedly

What if increment is implemented incorrectly?

e.g., It only throws VTSE when c.value < 0

Slides 40 - 41

JUnit: Where an Exception is Expected (1)

JUnit Test

```java
@Test
public void testDecFromMinValue() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.decrement();
    fail("ValueTooSmallException is expected.");
  }
  catch(ValueTooSmallException e) {
    /* Exception is expected to be thrown. */
  }
}
```

throw VTSE as expected

Scenario 1:
dec implemented correctly

Scenario 2:
dec not imp. correctly

Console Tester

```java
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("Error: ValueTooSmallException NOT thrown.");
    }
    catch (ValueTooSmallException e) {
      println("Success: ValueTooSmallException thrown.");
    }
  } /* end of main method */
} /* end of class CounterTester1 */
```

```java
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment();
    assertEquals(1, c.getValue());
  }
  catch(ValueTooBigException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooBigException is not expected.");
  }
}
```

reaching this line means no exception happened unexpectedly.

VTBE happened unexpectedly.

```java
@Test
public void testDecFromMinValue() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.decrement();
    fail ("ValueTooSmallException is expected.");
  }
  catch(ValueTooSmallException e) {
    /* Exception is expected to be thrown. */
  }
}
```

the expected exception did not occur

↓ the expected exception occurred ⇒ pass

JUnit: where an Exception is Expected (2.1)

Console Tester

working

```java
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try {
      c.increment(); c.increment(); c.increment();
      println("Current val: " + c.getValue());
      try {
        c.increment();
        println("Error: ValueTooLargeException NOT thrown.");
      } /* end of inner try */
      catch (ValueTooLargeException e) {
        println("Success: ValueTooLargeException thrown.");
      } /* end of inner catch */
    } /* end of outer try */
    catch (ValueTooLargeException e) {
      println("Error: ValueTooLargeException thrown unexpectedly.");
    } /* end of outer catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

(expected)

JUnit Test

working

(unexpected)

```java
@Test
public void testIncFromMaxValue() {
  Counter c = new Counter();
  try {
    c.increment(); c.increment(); c.increment();
  }
  catch (ValueTooLargeException e) {
    fail("ValueTooLargeException was thrown unexpectedly.");
  }
  assertEquals(Counter.MAX_VALUE, c.getValue());
  try {
    c.increment();
    fail("ValueTooLargeException was NOT thrown as expected.");
  }
  catch (ValueTooLargeException e) {
    /* Do nothing: ValueTooLargeException thrown as expected. */
  }
}
```

JUnit: where an Exception is Expected (2.2)

Recall the alternative to ConuterTester2 that has un-nested try-catch blocks.

Why is the JUnit test logically correct but the Console Tester is not?

```java
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try {
      c.increment(); c.increment(); c.increment();
      println("Current val: " + c.getValue());
    }
    catch (ValueTooLargeException e) {
      println("Error: ValueTooLargeException thrown unexpectedly.");
    }
    try {
      c.increment();
      println("Error: ValueTooLargeException NOT thrown.");
    } /* end of inner try */
    catch (ValueTooLargeException e) {
      println("Success: ValueTooLargeException thrown.");
    } /* end of inner catch */
  } /* end of main method */
} /* end of CounterTester2 class */
```

Console Tester

VTLE thrown unexpectedly

unnested ⇒ not working.

Flow not disrupted.

```java
@Test
public void testIncFromMaxValue() {
  Counter c = new Counter();
  try {
    c.increment(); c.increment(); c.increment();
  }
  catch (ValueTooLargeException e) {
    fail("ValueTooLargeException was thrown unexpectedly.");
  }
  assertEquals(Counter.MAX_VALUE, c.getValue());
  try {
    c.increment();
    fail("ValueTooLargeException was NOT thrown as expected.");
  }
  catch (ValueTooLargeException e) {
    /* Do nothing: ValueTooLargeException thrown as expected. */
  }
}
```

JUnit Test

VTLE thrown unexpectedly

unnested ⇒ working

test fails right away

Exercise

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 }
```

if VTLE thrown ↳ it's unexpected

if VTLE thrown ↳ it's expected

it's not possible to know whether or not the VTLE is expected

Hint: Say Line 12 is executed,

is it clear if that ValueTooLargeException was thrown as expected?

Testing Many Values in a Single Test

Loops can make it effective on generating test cases:

```
@Test
public void testIncDecFromMiddleValues() {
  Counter c = new Counter();
  try {
    for(int i = Counter.MIN_VALUE; i < Counter.MAX_VALUE; i ++) {
      int currentValue = c.getValue();
      c.increment();
      assertEquals(currentValue + 1, c.getValue());
    }
    for(int i = Counter.MAX_VALUE; i > Counter.MIN_VALUE; i --) {
      int currentValue = c.getValue();
      c.decrement();
      assertEquals(currentValue - 1, c.getValue());
    }
  }
  catch(ValueTooLargeException e) {
    fail("ValueTooLargeException is thrown unexpectedly");
  }
  catch(ValueTooSmallException e) {
    fail("ValueTooSmallException is thrown unexpectedly");
  }
}
```

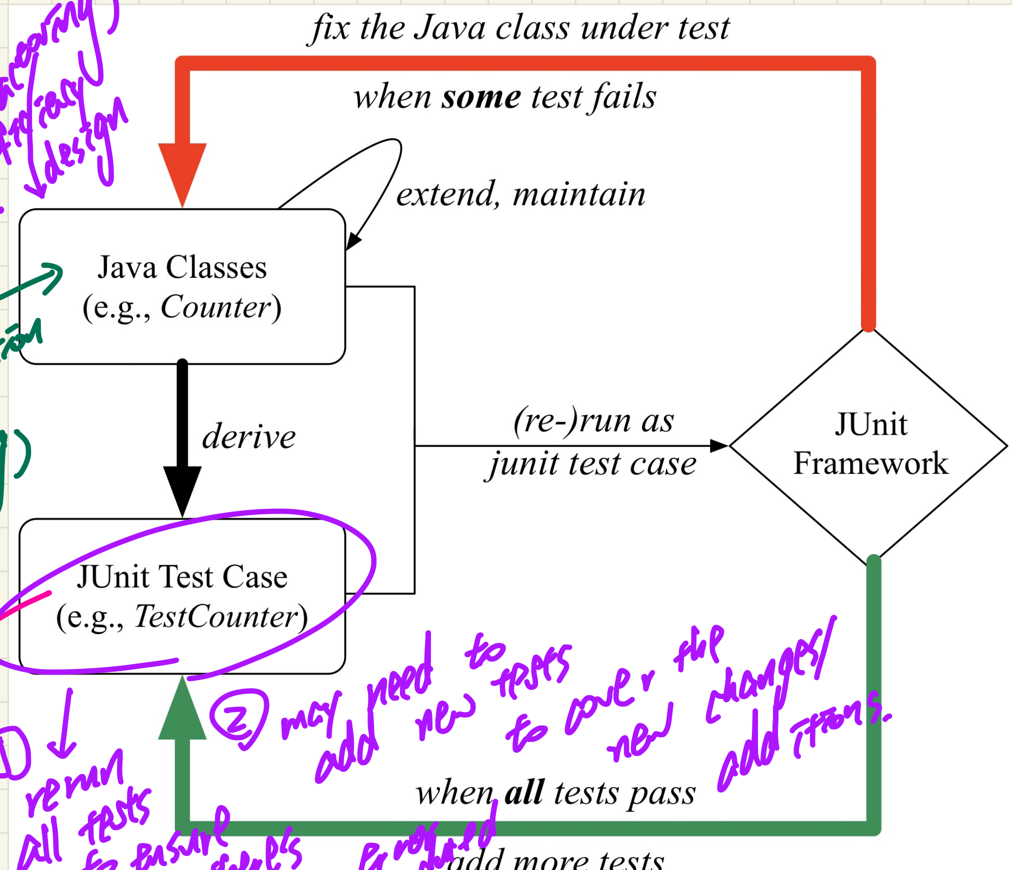
→ any unexpected VTLE thrown here will fail the test

what if the →MIN: 0
→MAX: 1000

Test-Driven Development (TDD): Regression Testing

some internal changes (refactoring) w/ 1. efficiency 2. design

"model" implementation (right vs. wrong)



define "correctness" of imp. under test (1. coverage 2. quality)

① rerun all tests to ensure there's no error introduced

② may need to add new tests to cover the new changes/additions.

Lecture 11 - Oct. 19

Object Equality

To Override or Not to Override
Overriding equals: 4 Phases

Announcements

- Lab2 due this Friday
- Look ahead: WrittenTest2 & ProgTest2
    + **Important Exercise**:
      Use debugger to explore execution paths
      in the console testers & JUnit tests

int i = ...;

int j = ...;

i == j → compare primitive value

Person p1 = new ...;

Person p2 = ...;

p1 == p2 → compare address values

obj1.equals(obj2)

obj1 → this

obj2 → obj

① each class has one parent class

super class / parent class

② each class may have multiple child classes

```
class Object{
    :
    equals
}
```

everything in Object is inherited / accumulated to any class

↳ Java library class

- classes in project.

inheritance

```
class Person{
    :
}
```

child class / sub class

$x == y$

① x, y are primitive vars.

② x, y are ref vars.

i.o.

x.equals(y)

① x, y are ref var

The equals Method: To Override or Not?

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

default version inherited to every class

extends

extends

default equals inherited

```
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;
  }
}
```

overrides/redefines the default version

The equals Method: Default Version

s → "(2, 3)"

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

p1 p1 p1

extends

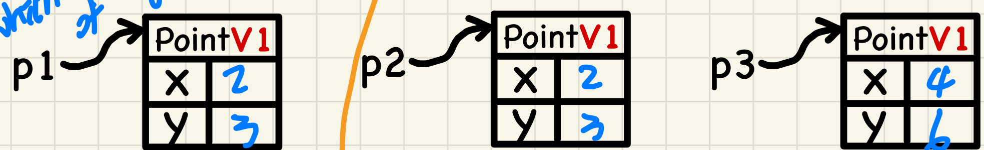
Strategy
① Find out the type of the C.O. obj
② See which version of equals.

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

```java
String s = "(2, 3)";
PointV1 p1 = new PointV1(2, 3);
PointV1 p2 = new PointV1(2, 3);
PointV1 p3 = new PointV1(4, 6);
System.out.println(p1 == p2); /* false */
System.out.println(p2 == p3); /* false */
System.out.println(p1.equals(p1)); /* true */
System.out.println(p1.equals(null)); /* false */
System.out.println(p1.equals(s)); /* false */
System.out.println(p1.equals(p2)); /* false */
System.out.println(p2.equals(p3)); /* false */
```

≡ p1 == p1

p1 == null



① boils down: p1 == s (F)

② p1 == s

The equals Method: Overridden Version

Phase 1

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

extends

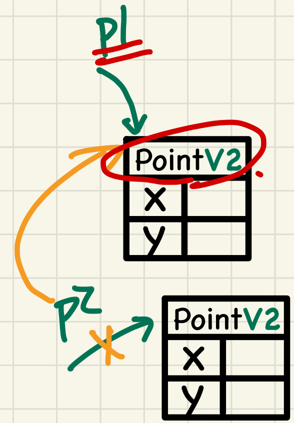
```java
public class PointV2 {
  private int x;
  private int y;
  public PointV2 (int x, int y) { ... }
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false }
    Point other = (PointV2)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

PointV2 p1 = new PointV2(..);

p1.equals(..);

p1 → C.O. dynamic ✓ is of type PointV2

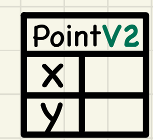
↳ overridden version is called



p2 = p1;

p1.equals(p2);

↓ since equals meth. is overridden, call that version.



The equals Method: Overridden Version

Phase 2

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

extends

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

Have we missed:

p1 → null

p2 → null

if (this == null && obj == null) { return true; }

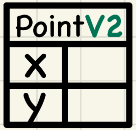
no need to consider this == null ∵ a NPE would've occurred

reaching this line means this != obj

p1.equals(p2)

Scenario 1: more to phase 3

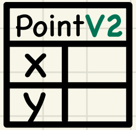
p1 != p2 and p2 is not null

p1 → 

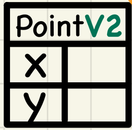
p2 → 

Scenario 2

(a non-null object is not equal to a null obj)

p1 → 

p2 → null



The equals Method: Overridden Version

Phase 3

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

extends

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

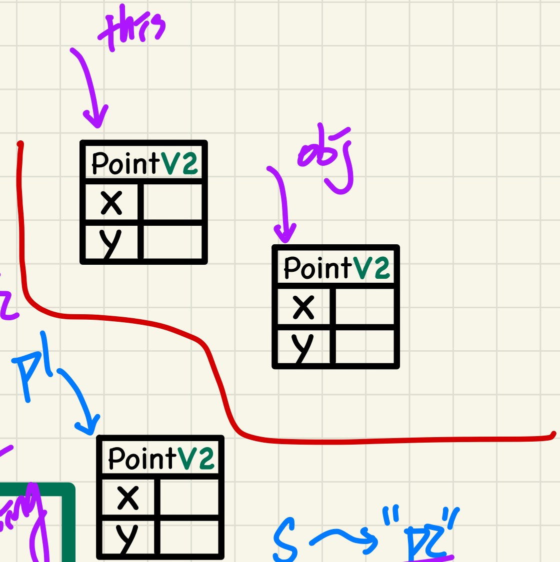
reaching this line means:
1. this != obj
2. obj != null

p1.getClass() != s.getClass()

PointV2

String

T: comparing objects of diff. types

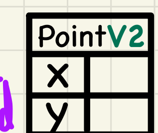


s ~> "P2"

p1.equals(s);

① returns the dynamic type of the C.O.

② Title of the object box pointed to by C.O.



The equals Method: Overridden Version

Phase 4

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

extends

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

reaching this line means:
① this != obj
② obj != null
③ comparing objects of the same type

static type restricts the range of methods that can be called on obj

this

PointV2	
x	
y	

obj

PointV2	
x	
y	

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

① we can only invoke methods/attrs declared in the ST of obj.

② x, y only declared in PointV2

PointV2	
x	
y	

review up to this line by Monday

Lecture 12 - Oct. 24

Object Equality

Overriding equals: Type Casting
JUnit Assertions for Object Equality
Short-Circuit Evaluation: && vs. ||

Announcements

- ProgTest1 grading finishing this week
- Lab2 solution video
- Exam confirmed by the registrar office:
  + **In-Person**: 7pm to 10pm, Monday, December 12
  + Last day of class: Monday, December 5
  + Review session(s)?
- WrittenTest2: Guide & Practice Questions by Thursday

The equals Method: Overridden Version

Phase 4

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

extends

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

static type "static" keyword nothing to do with

↳ declared type

↳ restricts the range of methods "callable" on obj

↳ obj.x ✗

obj.y ✗

parameter

PointV2 p3 = new PointV2(..);

PointV2 p4 = new PointV2(..);

p3.equals(p4);

argument

(PointV2) obj

↓

ST of the alias

① obj != this

② obj != null

③ this and obj have same dynamic type

other has ST PointV2 (∴ x and y can be called upon it)

p3 → PointV2 (x, y)

p4 → PointV2 (x, y)

other

obj ST: Object

ST: PointV2 alias created by cast

The equals Method: Overridden Version

Example 1: Trace L7

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

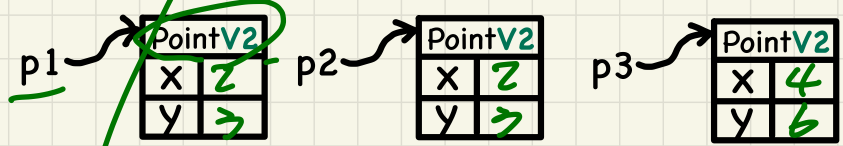
extends

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

overridden version

p1 p1 p1

```java
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));   /* [redacted] */
9  System.out.println(p1.equals(s));   /* [redacted] */
10 System.out.println(p1.equals(p2));   /* [redacted] */
11 System.out.println(p2.equals(p3));   /* [redacted] */
```



dynamic type is PointV2 ⇒ version of equals in PointV2 called

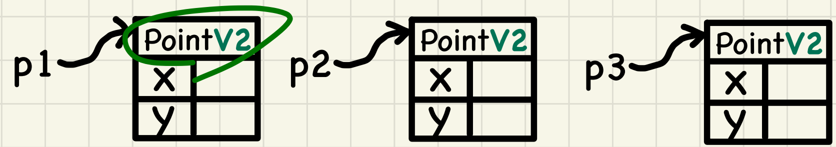
The equals Method: Overridden Version

Example 1: Trace L8

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

extends

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




```
public class PointV2 {
  private int x;
  private int y;
  public PointV2 (int x, int y) { ... }          null
  public boolean equals(Object obj) {
    if(this == obj) { return true; }        p1   null
    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;
  }
}
```

The equals Method: Overridden Version

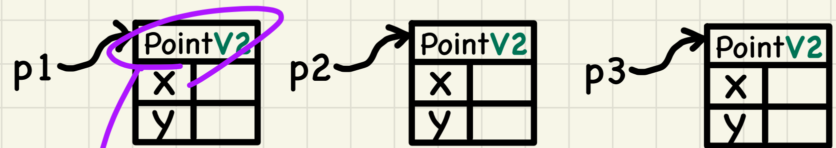
Example 1: Trace L9

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

extends

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

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



p1.getClass()

s.getClass() → String

The equals Method: Overridden Version

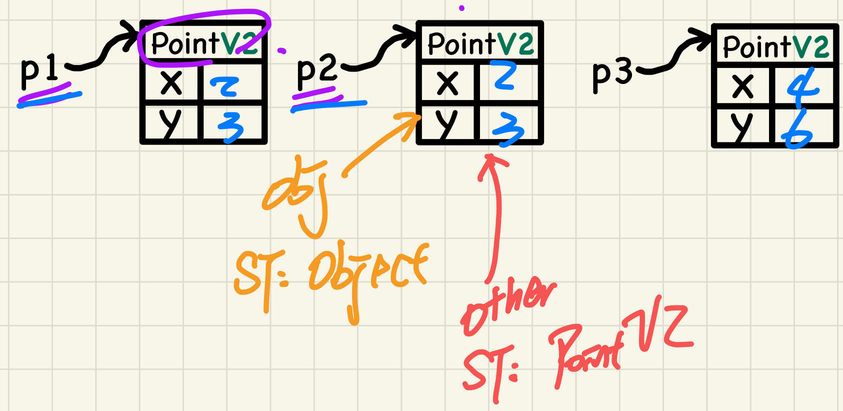
Example 1: Trace L10

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

extends

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

```
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;
  }
}
```



The equals Method: Overridden Version

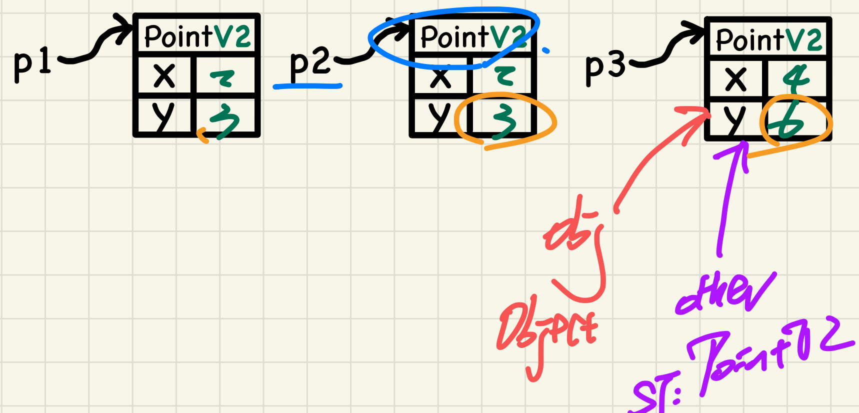
Example 1: Trace L11

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

extends

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

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



The equals Method: To Override or Not?

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

extends

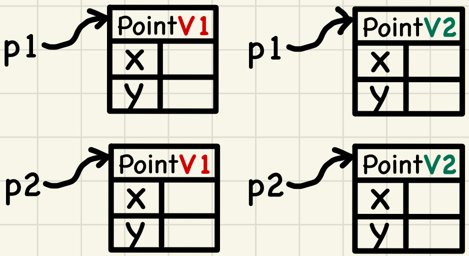
extends

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

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

```java
String s = "(2, 3)";
PointV1 p1 = new PointV1(2, 3);
PointV1 p2 = new PointV1(2, 3);
PointV1 p3 = new PointV1(4, 6);
System.out.println(p1 == p2);   /* false */
System.out.println(p2 == p3);   /* false */
System.out.println(p1.equals(p1));   /* true */
System.out.println(p1.equals(null));   /* false */
System.out.println(p1.equals(s));   /* false */
System.out.println(p1.equals(p2));   /* false */
System.out.println(p2.equals(p3));   /* false */
```

```java
String s = "(2, 3)";
PointV2 p1 = new PointV2(2, 3);
PointV2 p2 = new PointV2(2, 3);
PointV2 p3 = new PointV2(4, 6);
System.out.println(p1 == p2);   /* false */
System.out.println(p2 == p3);   /* false */
System.out.println(p1.equals(p1));   /* true */
System.out.println(p1.equals(null));   /* false */
System.out.println(p1.equals(s));   /* false */
System.out.println(p1.equals(p2));   /* true */
System.out.println(p2.equals(p3));   /* false */
```



The equals Method: Overridden Version

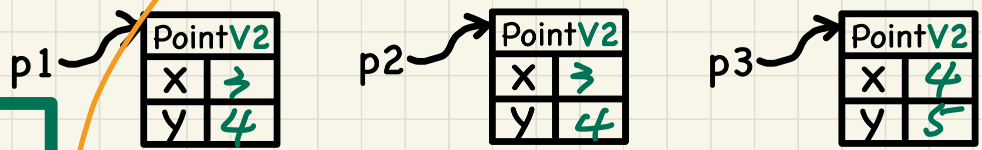
Example 2

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

extends

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

```java
PointV2 p1 = new PointV2(3, 4);
PointV2 p2 = new PointV2(3, 4);
PointV2 p3 = new PointV2(4, 5);
System.out.println(p1 == p1);        /* true */
System.out.println(p1.equals(p1));   /* true */
System.out.println(p1 == p2);        /* false */
System.out.println(p1.equals(p2));   /* true */
System.out.println(p2 == p3);        /* false */
System.out.println(p2.equals(p3));   /* false */
```



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

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

1 holds

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

2

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

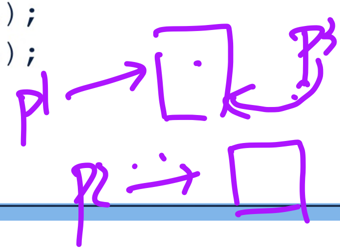
does not hold

assertSame vs. assertEquals

assertSame(exp1, exp2)

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

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



assertEquals(exp1, exp2)

- ≈ $\boxed{\texttt{exp1 == exp2}}$ if `exp1` and `exp2` are **primitive** type

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

reference types

assertEquals(x, y)

↳ x.equals(y)

assertEquals(y, x)

↳ y.equals(x)

① may make a diff.

$\neq$ x.getClass() != y.getClass()

② may not make a diff

$\neq$ otherwise

assertEquals: Reference Comparison or Not

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); × /* ∵ different PointV1 objects */
assertEquals(p2, p3); × /* ∵ different types of objects */
```

p1 == p2

p2 == p3

addresses

p2.e(p3)

p1.e(p2)

p1 → PointV1

p2 → PointV1

p3 → PointV2

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); [redacted]
assertEquals(p2, p3); [redacted]
assertEquals(p3, p2); × /* ≈ p3.equals(p2) ≈ p3.getClass()==p2.getClass() */
```

p3.equals(p2)

```
PointUI p1 = new PointUI(...);
PointUI p2 = new PointUI(...);

p1.equals(p2)
  ↳ p1 == p2
  ↳ assertSame(p1, p2).
```

Short-Circuit Evaluation: && conjunction

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

if any of the operands is F
&& → F

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");
    }
}
```

guarding cond. for div. by zero.

$\boxed{0 != 0}$ && (10/0 > 2)

F

not to be evaluated.

```java
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");
    }
}
```

evaluate this first

$\boxed{y / x > 2}$ &&

↳ crash

$x \neq 0$

Input:

$x = 0$, $y = 10$

$p \wedge q$

$\equiv q \wedge p$

Short-Circuit Evaluation: ||

disjunction

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

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

if true → RHS skipped

Test Inputs:

x = 0, y = 10

x = 5, y = 10

Ex 1. Swap order of ||

Ex 2. Compare the SCE condition between && and ||

if any one of the operands is true

|| → T

0 == 0 || 10/0 > 2

T

not evaluated!

Short-Circuit Evaluation: Common Errors

Test Inputs:

x = 0, y = 10

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

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

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

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

Short Circuit Evaluation

$\boxed{\begin{matrix} e1 \\ \\ e3 \end{matrix}}$ $\begin{matrix} \&\& & e2 \\ \\ \| & e4 \end{matrix}$

guarding condition

e2: prevented from being evaluated if e1 is F.

e4: prevented from being evaluated if e3 is T.

Lecture 13 - Oct. 26

Object Equality

Equality for Array-Typed Attributes
Call by Value

Announcements

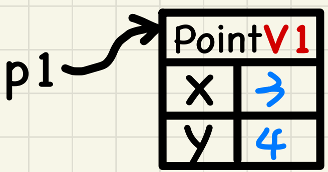
- ProgTest1 final processing: results expected by tmw
- Lab3 to be released on Monday

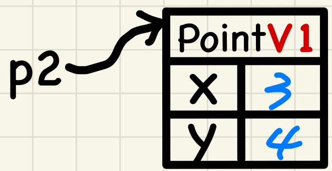
Testing Default Equality of Points in JUnit

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

p1 == p2 return false

assertSame(p1, p2); fail






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

p2 p1 p2

extends

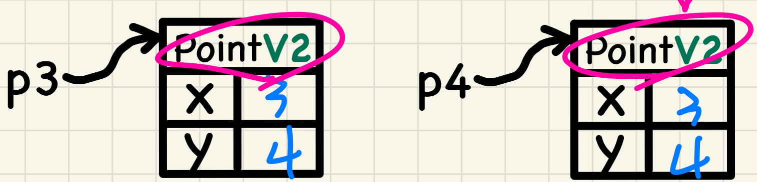
```java
public class PointV1 {
  private int x;
  private int y;
  public PointV1 (int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Testing Overridden Equality of Points in JUnit

```java
@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, p4); */ /* both fail */
  assertTrue(p3.equals(p4)); assertTrue(p4.equals(p3));
  assertEquals(p3, p4); assertEquals(p4, p3);
}
```

true

true




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

extends

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

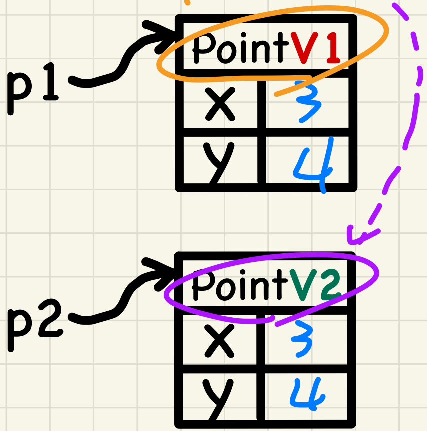
overridden

Testing Equality of Points in JUnit: Default vs. Overridden

```java
@Test
public void testEqualityOfPointV1andPointv2() {
  PointV1 p1 = new PointV1(3, 4); PointV2 p2 = new PointV2(3, 4);
  /* These two assertions do not compile because p1 and p2 are of different types. */
  /* assertFalse(p1 == p2); assertFalse(p2 == p1); */
  /* assertSame can take objects of different types and fail. */
  /* assertSame(p1, p2); */ /* compiles, but fails */
  /* assertSame(p2, p1); */ /* compiles, but fails */
  /* version of equals from Object is called */
  assertFalse(p1.equals(p2));
  /* version of equals from PointV2 is called */
  assertFalse(p2.equals(p1));
}
```

→ p1 == p2

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



extends

extends

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

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

int i = ...
int j = ...

~~assertSame(i, j);~~

assertSame(exp1, exp2);

are exp1 and exp2 storing the same object ref.

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

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

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 }
```

short-circuit evaluation

null

null

∵ obj. weight ✗

C.O. of type String

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

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

evaluated first

$$\boxed{\textbf{if}(\textbf{this}.getClass()\ != obj.getClass())}\ ||\ obj == \textbf{null})$$

null → NPE!

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 }
```

∵ obj.nop ✗

Phase 4

as soon as 'equal' becomes false, exit from the loop array

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

```
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);
  }
}
```

```
class PersonCollector {
    Person[] persons;
    ⋮
    ... equals(...) {
        ⋮
        this.persons[i].equals(other.persons[i]);
    }
}
```

PersonCollector

C.O.

version in Person class is invoked.

Testing Equality of Person/PersonCollector in JUnit (1)

```java
@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));
```

Person version

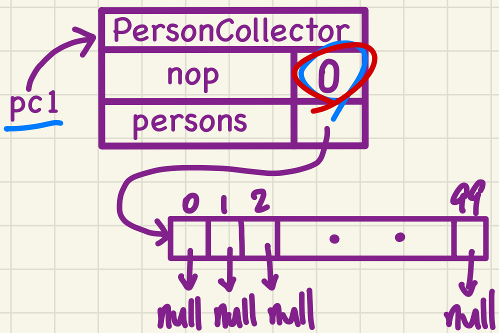

```java
public class Person {
  private String firstName; private String lastName;
  private double weight; private double height;
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null || this.getClass() != obj.getClass()) { return false; }
    Person other = (Person)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

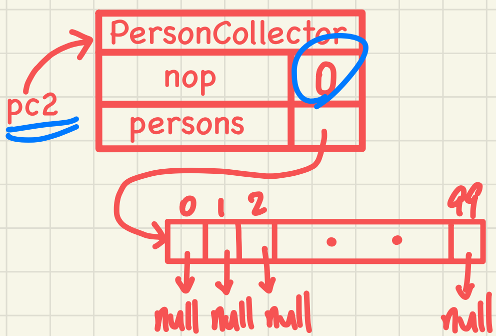
Testing Equality of Person/PersonCollector in JUnit (2)

(continued from testPersonCollector)

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

true (not even an iteration is run).





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

```java
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() { ... }
}
```

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

pc1 pc2

true equal

true && 0 < 0 F

Testing Equality of Person/PersonCollector in JUnit (3)

(continued from testPersonCollector)

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

↳ Person version.

Person

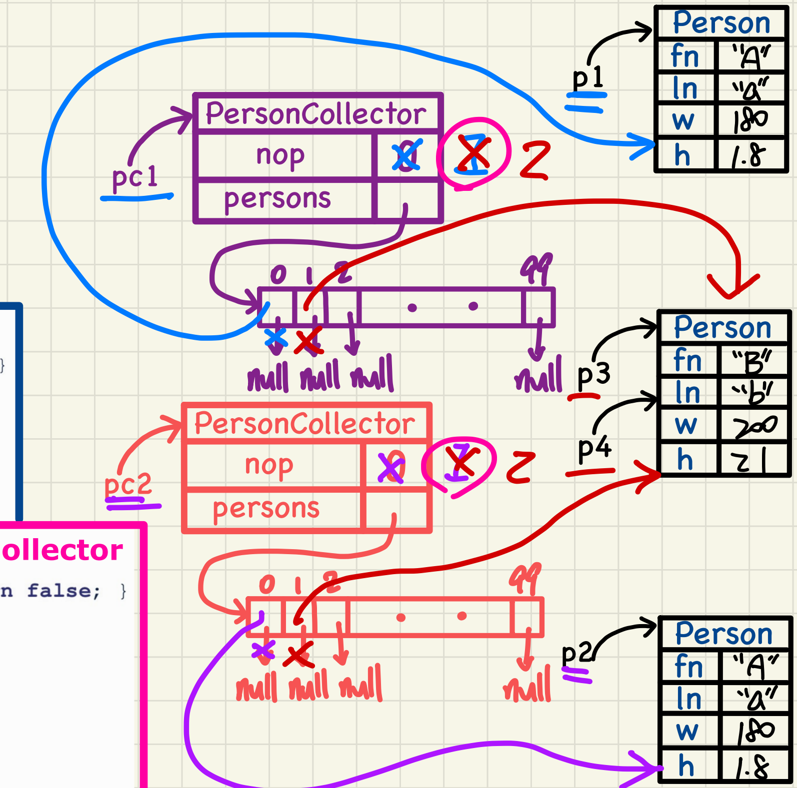
```
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  Person other = (Person) obj;
  return
       this.weight == other.weight
    && this.height == other.height
    && this.firstName.equals(other.firstName)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

PersonCollector

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

↳ Person version

false.

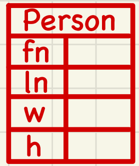


Testing Equality of Person/PersonCollector in JUnit (4)

(continued from testPersonCollector)

```java
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));
```

compare the two anonymous objects



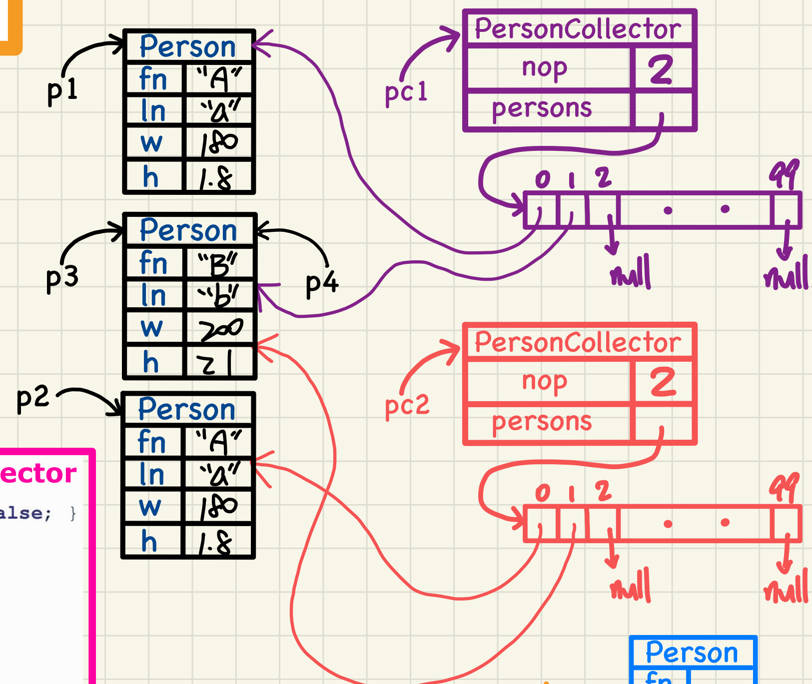

```java
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  Person other = (Person) obj;
  return
      this.weight == other.weight
   && this.height == other.height
   && this.firstName.equals(other.firstName)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

Person

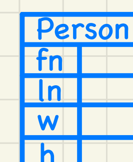
```java
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false; }
  PersonCollector other = (PersonCollector) obj;
  boolean equal = false;
  if(this.nop == other.nop) {
    equal = true;
    for(int i = 0; equal &&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

PersonCollector

In 3rd iteration (i==2)



(F)



the two anonymous objects are compared

Call by Value: Primitive Argument

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

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

call by value

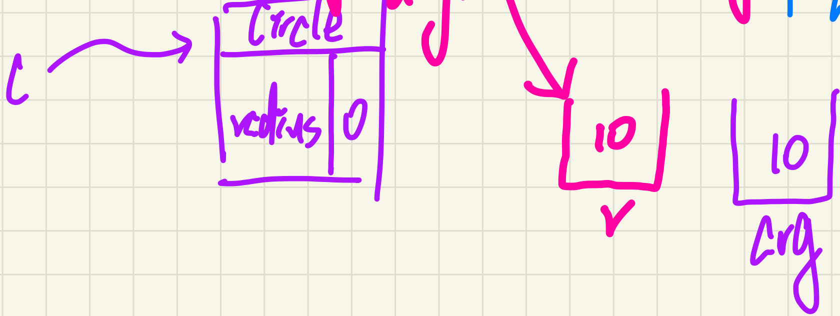
Primitive type

!! call by value

r is a copy of arg.

not referring to the original arg..

Primitive Argument



Call by Value: Reference Argument

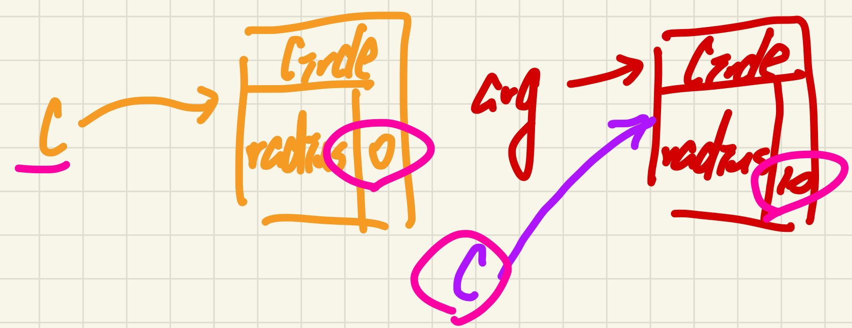
call by value

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

ref. type

10

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



referencing the same object as pointed to by arg.

copy of 'arg'

Lecture 14 - Oct. 31

Aggregation

Call by Value: Primitive vs. Reference Aggregations

Announcements

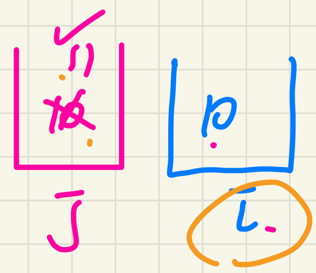
- ProgTest1: Visit office hours to discuss your solution
- Lab3 released (equals & copy constructor)
- WrittenTest2 tomorrow (guide & practice)

Call by Value: Re-Assigning Primitive Parameter

call by value

```java
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```java
@Test
public void testCallByVal() {
  Util u = new Util();
  int i = 10;
  assertTrue(i == 10);
  u.reassignInt(i);
  assertTrue(i == 10);
}
```



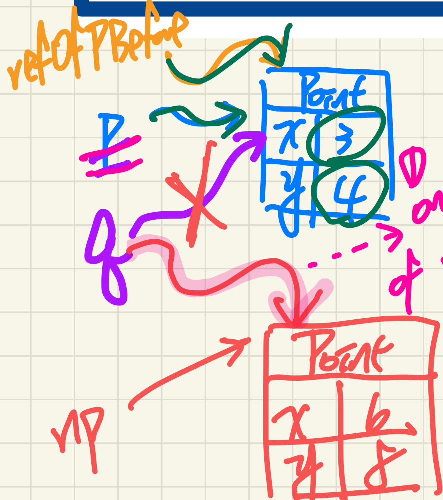
from the caller's point of view arg. i is not changed

Call by Value: Re-Assigning Reference Parameter

```java
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

call by value

```java
@Test
public void testCallByRef_1() {
  Util u = new Util();
  Point p = new Point(3, 4);
  Point refOfPBefore = p;
  u.reassignRef(p);
  assertTrue(p == refOfPBefore);
  assertTrue(p.getX() == 3);
  assertTrue(p.getY() == 4);
}
```



① only the copy of p is changed

a copy of address stored in p.

② p itself is not changed

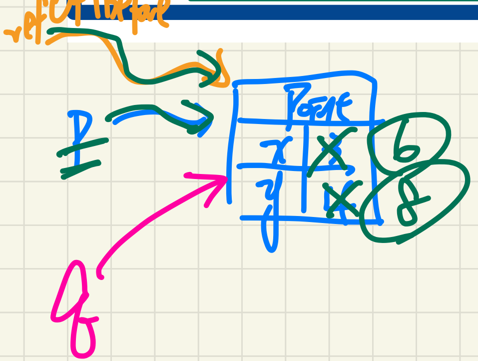
```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x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

Call by Value: Calling Mutator on Reference Parameter

```java
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4);  } }
```

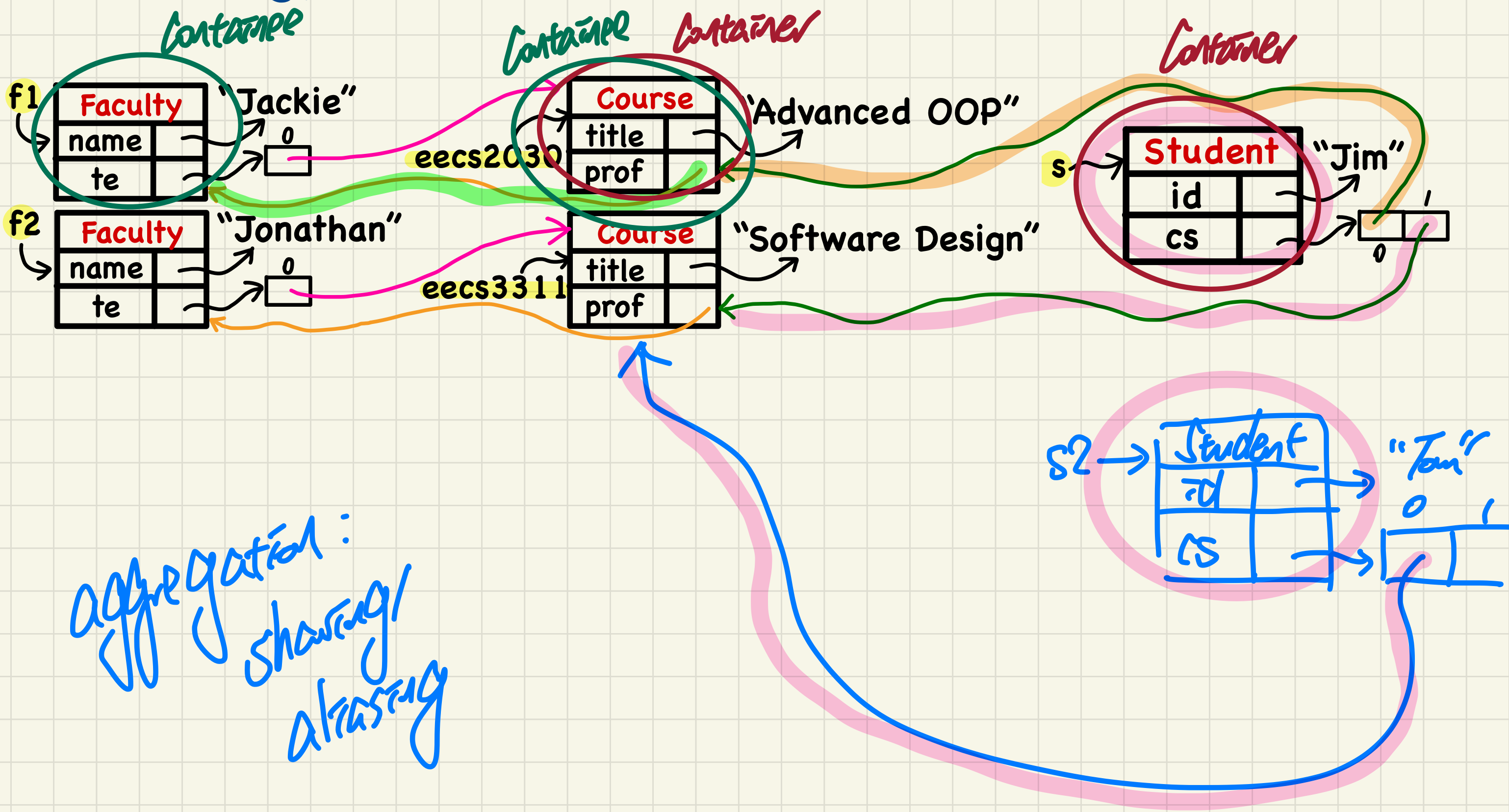
call by value

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




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

Terminology: Container vs. Containee

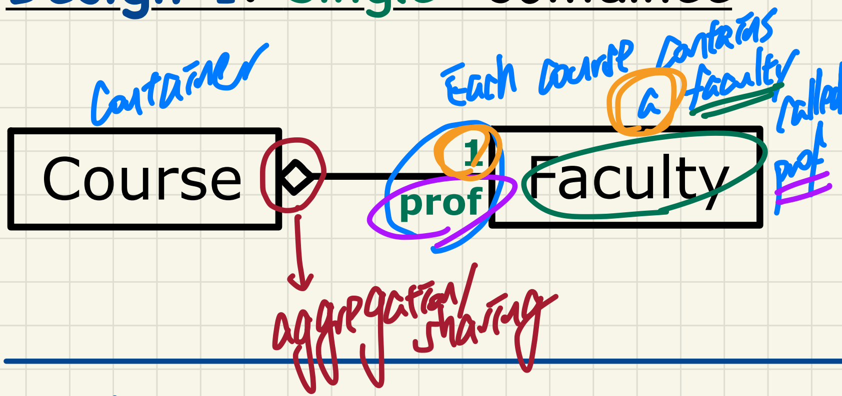


aggregation:
sharing/
aliasing

Aggregation: Design

Java Implementation

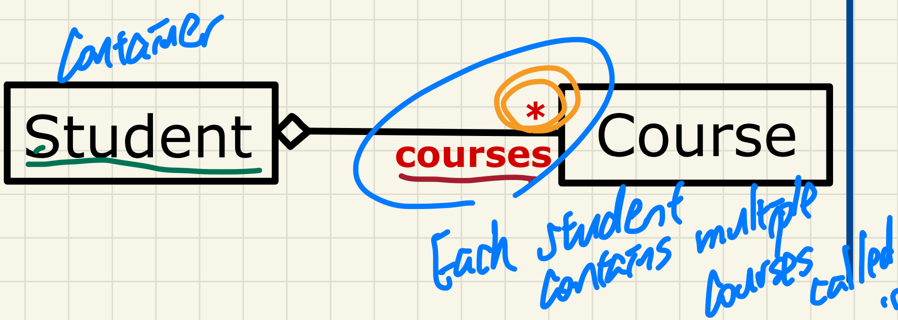
Design 1: Single Containee




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

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

Design 2: Multiple Containees




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

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

Aggregation (1)

Course	
title	
prof	

Faculty	
name	

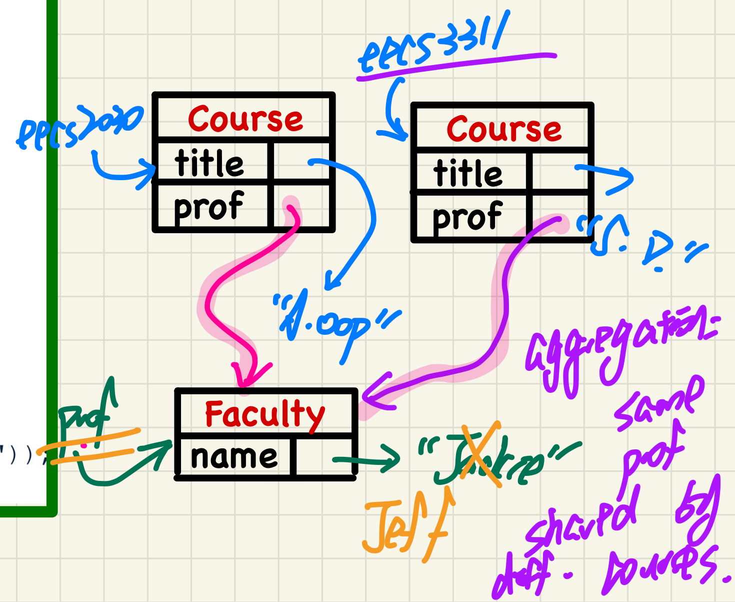
```java
class Course {
  String title;
  Faculty prof;
  Course(String title) {
    this.title = title;
  }
  void setProf(Faculty prof) {
    this.prof = prof;
  }
  Faculty getProf() {
    return this.prof;
  }
}
```

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

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

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

exercise



Aggregation (2)

Student	
id	
cs	

Faculty	
name	
te	

Course	
title	
prof	

(Exercise)

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

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

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

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

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

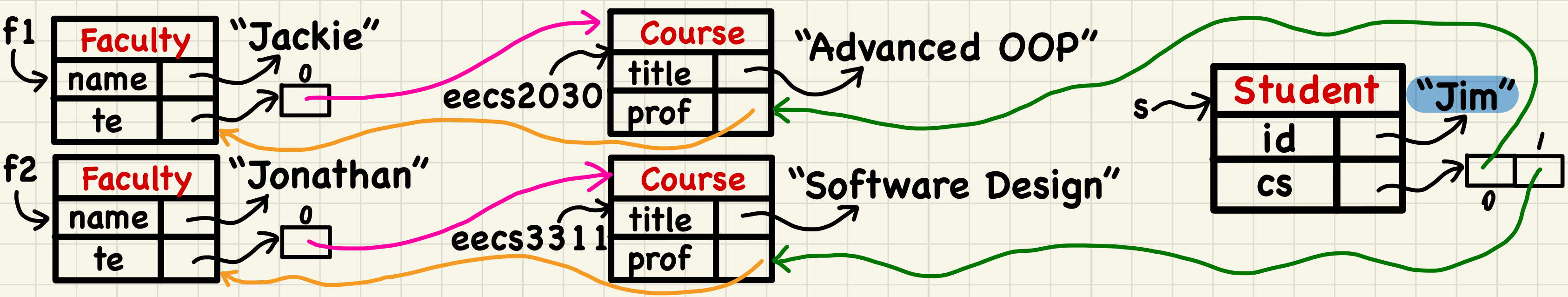
adding a container
(e.g. addPoint)

Runtime Object Structure: Student, Course, Faculty


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

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

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



Dot Notation for Navigating Classes (1)


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

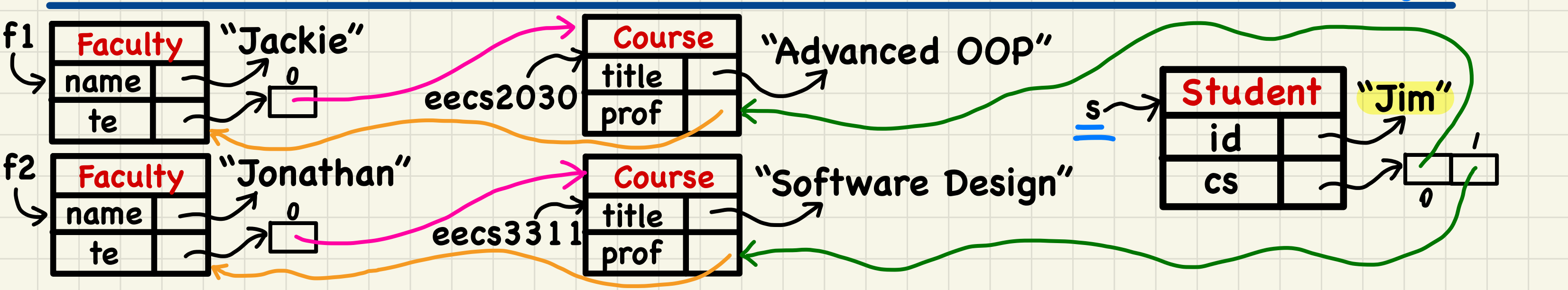
```
public class Course {
  private String title;
  private Faculty prof;
}
```

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

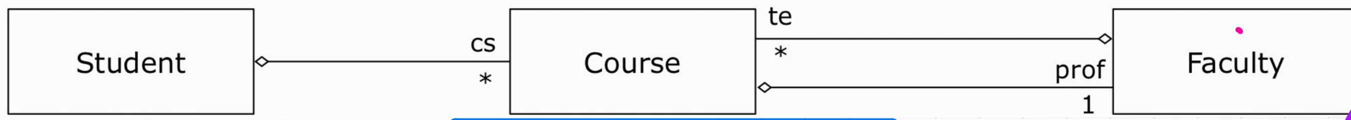
```
/* Get the student's id.
 */
String getID() {
  return this.id;
}
```

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

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



Dot Notation for Navigating Classes (2)




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

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

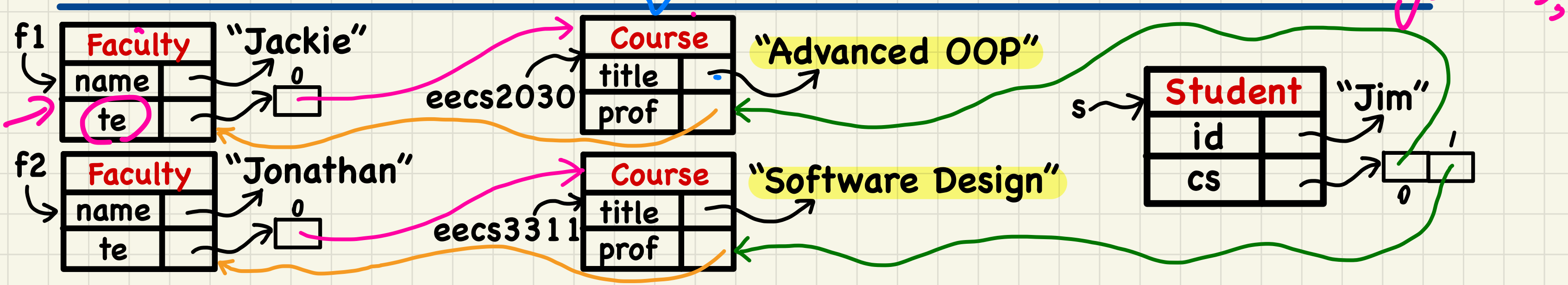
```java
public class Faculty {
  private String name;
  private Course[] te;
}
```

• te ?
• getTe() ?
wed:

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

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

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



Dot Notation for Navigating Classes (3)

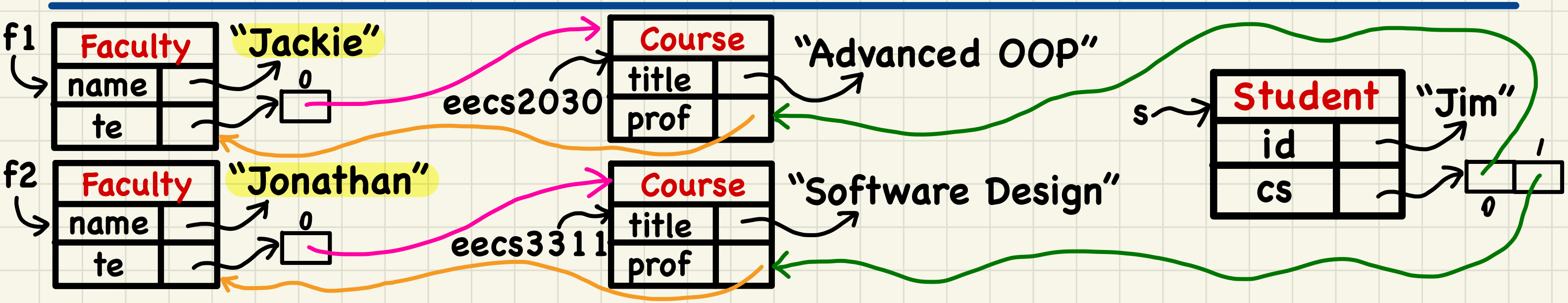

```java
public class Student {
  private String id;
  private Course[] cs;
}
```

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

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

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

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



Practice Written Test 2

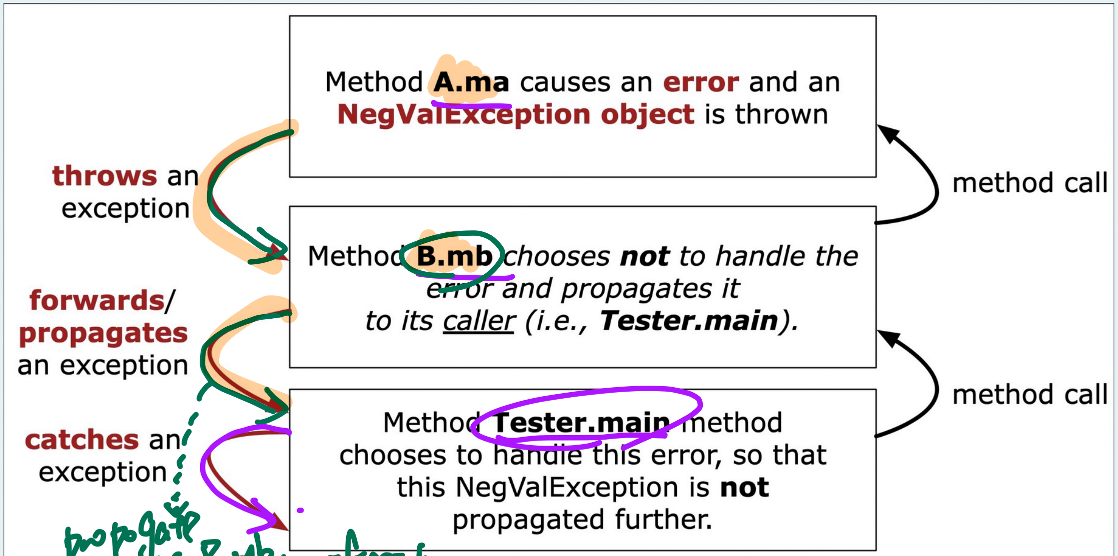
Q1

specify.

A.ma
B.mb
Tester.main

Call Stack

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



In the above call stack, upon satisfying the catch-or-specify requirement, how many methods opt for the **specify** option? Your answer must be an **integer** value.

Answer: 2.

Practice Written Test 2

Q2

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

Select one:

- True
- False

is m1 still subject to the try-catch req.

throws NegValException

callee of m2

caller of m2

call stack

implements a try-catch block

```
try {
  m3(...)
}
catch (NegValEx. ...) { ... }
```

Practice Written Test 2

Q3

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {                     1                2                 3
6       c.increment(); c.increment(); c.increment();
7       println("Current val: " + c.getValue());
8       try {
9  3     c.increment(); 4?
10       println("Error: ValueTooLargeException NOT thrown.");
11      } /* end of inner try */
12      catch (ValueTooLargeException e) {
13       println("Success: ValueTooLargeException thrown.");
14      } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17      println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19    } /* end of main method */
20  } /* end of CounterTester2 class */
```

Say the method `increment` is implemented correctly as explained above.

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any):

2nd line to execute (if any):

3rd line to execute (if any):

4th line to execute (if any):

5th line to execute (if any):

6th line to execute (if any):

7th line to execute (if any):

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

L13 of CounterTester2 | L17 of CounterTester2 | Execution Terminated

Practice Written Test 2

Q4

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown.");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeExcept
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

Say the *increment* method is implemented **incorrectly** as follows:

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

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any): _____

2nd line to execute (if any): _____

3rd line to execute (if any): _____

4th line to execute (if any): _____

5th line to execute (if any): _____

6th line to execute (if any): _____

7th line to execute (if any): _____

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

L13 of CounterTester2 | L17 of CounterTester2 | Execution Terminated

Practice Written Test 2

Q5

Recall the assumptions made on the counter example:

- The counter's maximum value is 3.
- A correct implementation of the *increment* method should throw a ValueTooLargeException when the counter's current value reaches the maximum.

Now consider the following console tester:

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown.");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeExce
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeExcepti
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

Say the *increment* method is implemented **incorrectly** as follows:

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

From the following lines of execution, drag and drop the **relevant** ones to indicate the corresponding runtime execution path.

Where the execution already terminates, drag and drop "Execution Terminated" to the execution line.

1st line to execute (if any): _____

2nd line to execute (if any): _____

3rd line to execute (if any): _____

4th line to execute (if any): _____

5th line to execute (if any): _____

6th line to execute (if any): _____

7th line to execute (if any): _____

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

L13 of CounterTester2 | L17 of CounterTester2 | Execution Terminated

Practice Written Test 2

Q6

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

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

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

For the above PointV2 class, assume that there is a constructor, like in PointV1, whi

Let's now assume the following object creations:

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

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

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

Practice Written Test 2

Q7

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

Consider the following fragment of code:

```
if(0 <= i && ns[i] % 2 == 1 && i < ns.length) {
   System.out.println("Outcome 1");
}
else {
   System.out.println("Outcome 2");
}
```

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

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

Lecture 15 - Nov 2

Composition, Inheritance

Dotted Notation vs. Private Attributes
Compositions
The Student Management Problem

Announcements

- **ProgTest1**: Visit office hours to discuss your solution
- **Lab3** due next Wednesday (equals & copy constructor)
- **WrittenTest2** to be released by early Friday
- **ProgTest2**: guide to be released soon

→ Lab2.

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).

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

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

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

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

class X {

C.O. of type Y

A.

private

context class A

X ∵ context class A does not match type of this.ob

≠ X == Y

↳ ok to reference the private attribute.

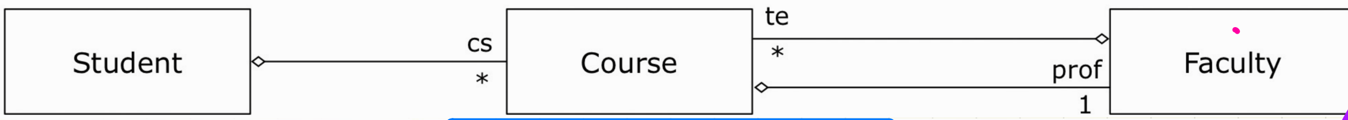
C.O. of type B

① ② ③

exercise

C.O. of .ai of type A matching context class A

Dot Notation for Navigating Classes (2)




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

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

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

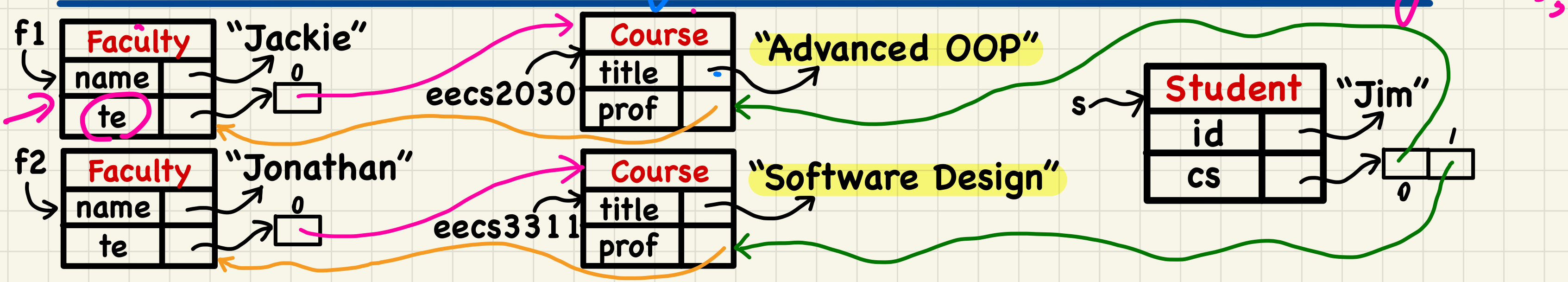
• te ?
• getTe() ?
Wed:

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

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

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

te?



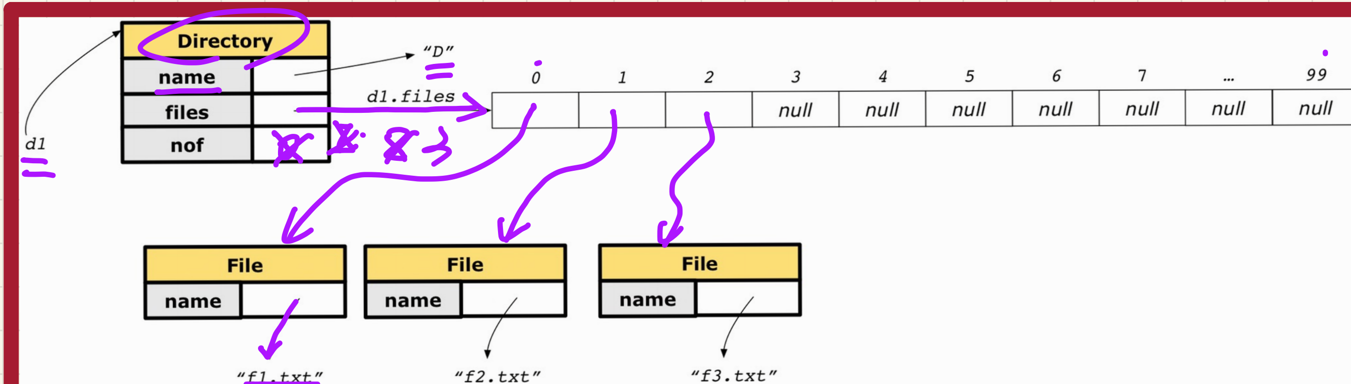
Composition: No Sharing

```
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;
}
```

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



```
class X {

    X ( X other ) {

    }

}
```

copy constructor

Composition: Copy Constructor (Shallow Copy)

calling the copy constructor

```
@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); /* violation of composition *
  d2.files[0].changeName("f11.txt");
  assertFalse(d1.files[0].name.equals("f1.txt")); }
```

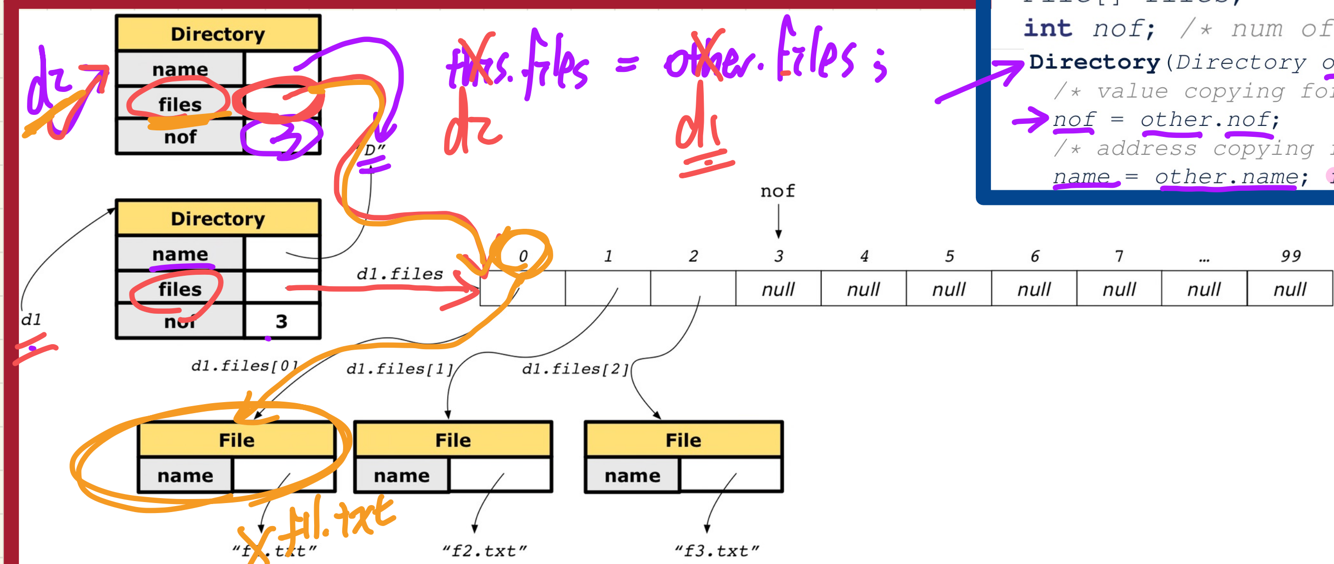
other

satisfied of composition was preserved

```
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; }
```

dir

copy the beginning address of the array.



Composition: Copy Constructor (Deep Copy)

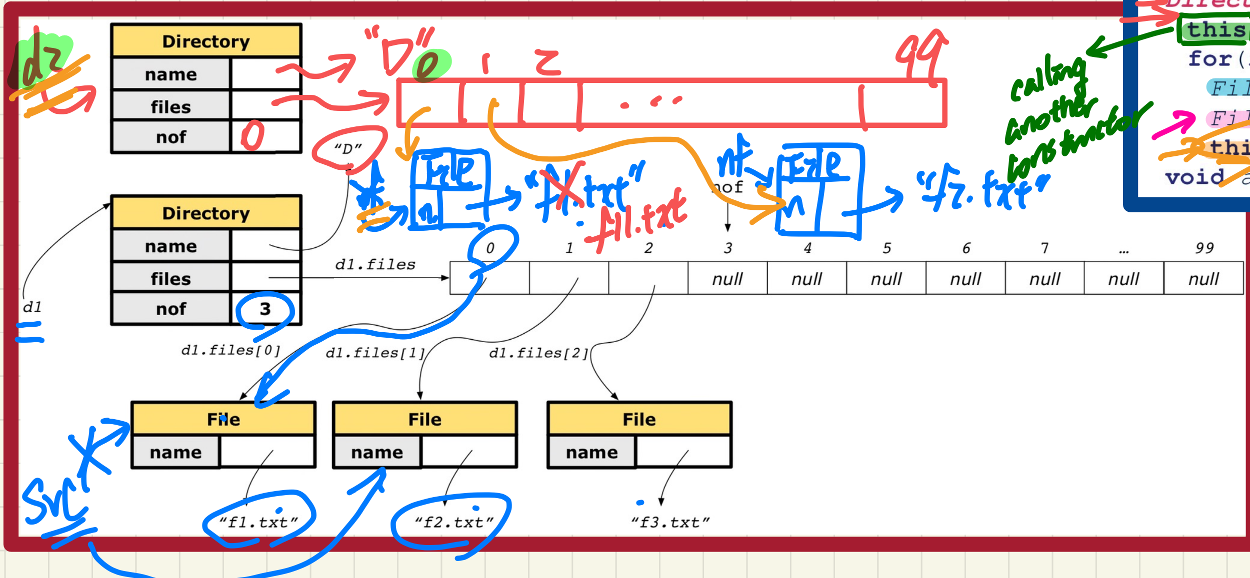
```java
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.files != d2.files); /* composition preserved */
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0].name.equals("f1.txt"));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

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

```java
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 < nof; i ++) {
     File src = other.files[i];
     File nf = new File(src);
     this.addFile(nf); } }
 void addFile(File f) { ... } }
```

other.nof

calling another constructor

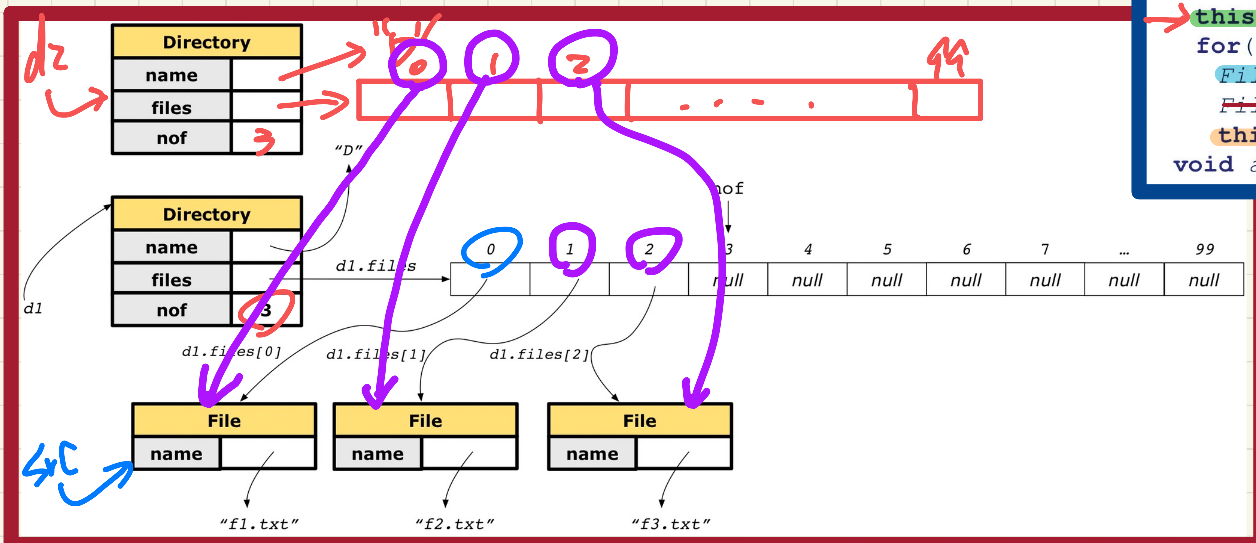


Exercise: Copy Constructor (Composition?)

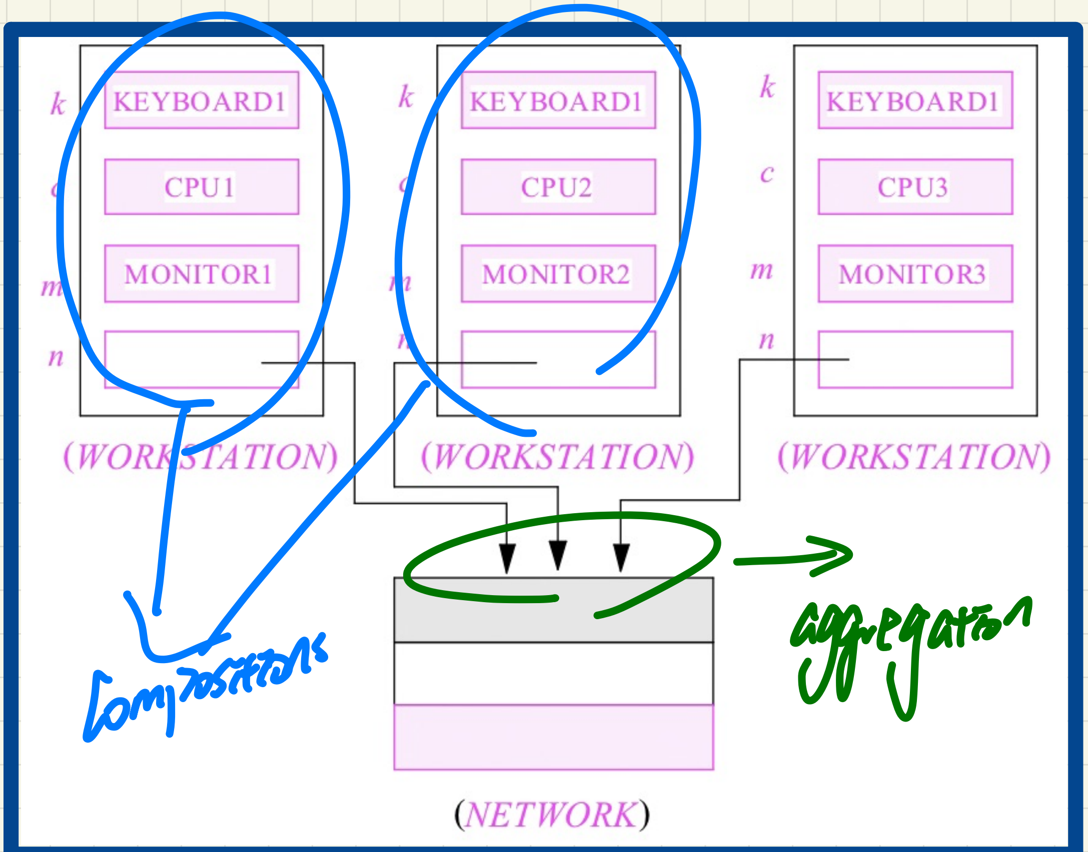
```java
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt"
  Directory d2 = new Directory(d1);
  assertTrue(d1.files != d2.files); [redacted]
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0]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

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

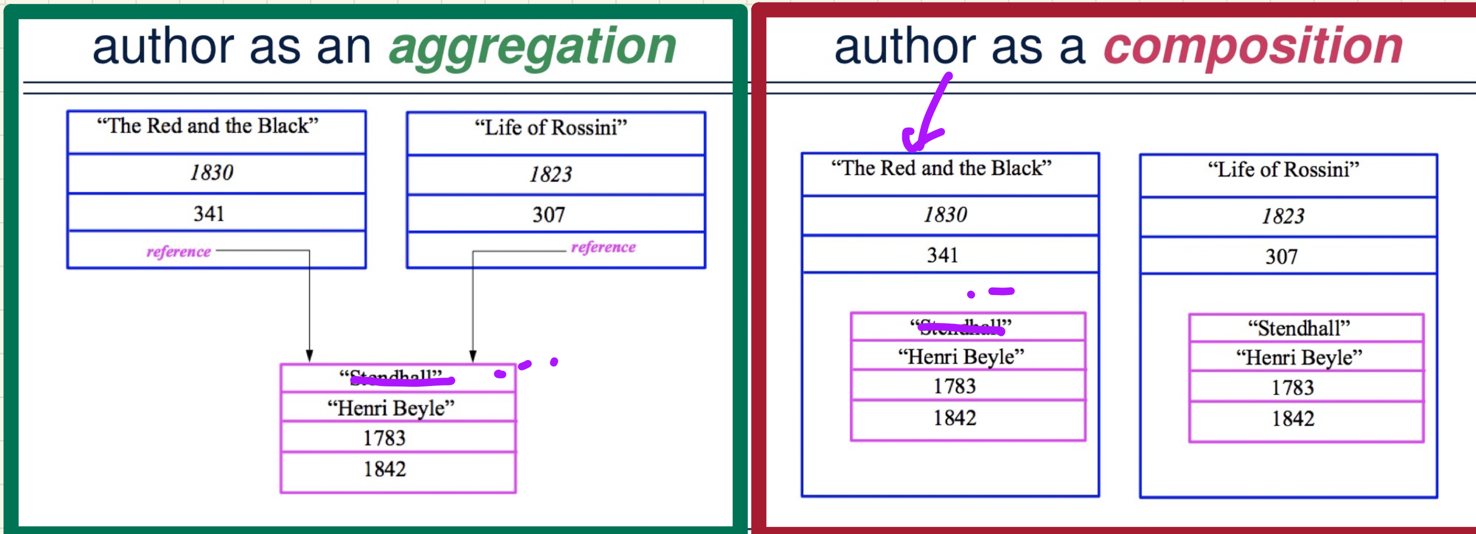
```java
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this (other.name);
    for(int i = 0; i < nof; i ++) {
      File src = other.files[i];
      ~~File nf = new File(src);~~
      this.addFile(~~nf~~ src);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



Modelling: Aggregation vs. Composition



Implementation: Aggregation or Composition



Hyperlinked author page | Physical printed copies

Inheritance: Motivating Problem

Student[ ]

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.

depends on the kind of student.

should not apply simultaneously!

Lecture 16 - Nov 7

Inheritance

SMS: Attempts without Inheritance
SMS: Use of extend, super
Visibility: Project, Package, Class

Lab2

Announcements

- ProgTest2: guide
- Lab3 due this Wednesday (equals & copy constructor)
- WrittenTest2 results released on Friday
- ProgTest1: Visit office hours to discuss your solution

First Design Attempt

design flaws

no implementation flaws

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

RS

NRS

Student rs = new Student(1);

Student nrs = new Student(2);

... rs.getTuition()

nrs.getTuition()

```
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;
  }
}
```

base

```
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

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

RS

NRS

should be separated to diff. classes

```java
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;
   }
}
```

```java
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 $\boxed{\text{Cohesion}}$

a class collects relevant att/methods to a common theme

Superman Class

all attr/methods for solving a problem go into this single class

First Design Attempt

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

compare with inheritance:
in inheritance the dynamic type is an automatic mechanism for managing student kind.

kind==3
int. student.

```java
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){ ... }
}
```

```java
public void register(Course c){
   int MAX = -1;
   if (this.kind == 1) { MAX = 6; }
   else if (this.kind == 2) { MAX = 4; }
   else if (this.kind == 3){
      ...
   }
   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 **obselete**?

no duplicates
if a change is needed
only one place needs to be changed

Testing Student Classes (without inheritance)

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

1000

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

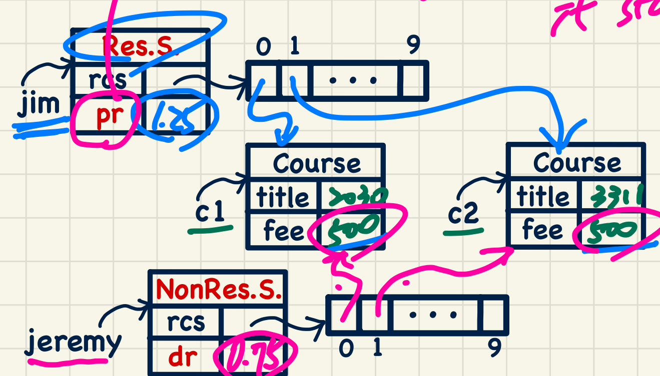
1000

assuming that all RSs have the same pr we may make it static.

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

1250

750



Student Classes (without inheritance): Maintenance (1)

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

add constraint

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

add constraint

Maintenance e.g., a new registration constraint:

```java
if(numberOfCourses >= MAX_ALLOWANCE) {
  throw new TooManyCoursesException("Too Many Courses");
}
else { ... }
```

Student Classes (without inheritance): Maintenance (2)

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

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

Maintenance e.g., a new tuition formula:

```java
/* ... can be premiumRate or discountRate */
...
return tuition * inflationRate * ...;
```

class RS

class NRS

class SMS {

}

Object[] students;

students[0] = new RS(...); ✓

students[1] = new NRS(...); ✓

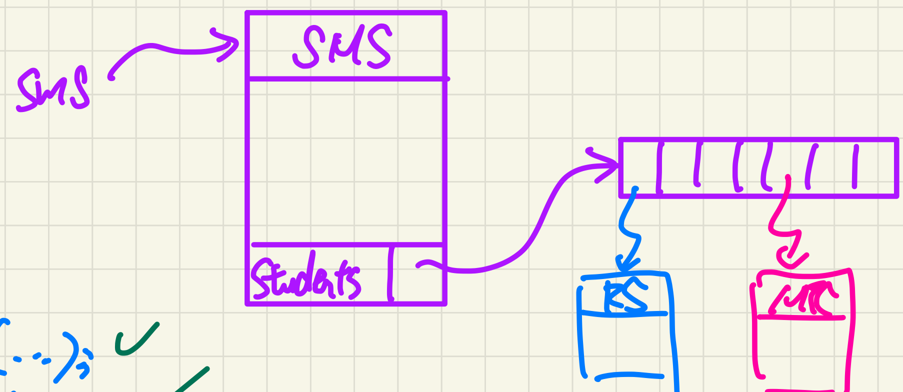
No No! poor design.

RS[] students = new RS[100];

students[0] = new RS(...); ✓

students[1] = new NRS(..); ✗

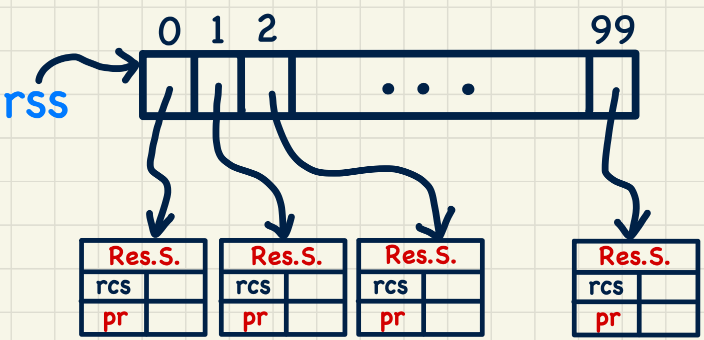
at runtime:

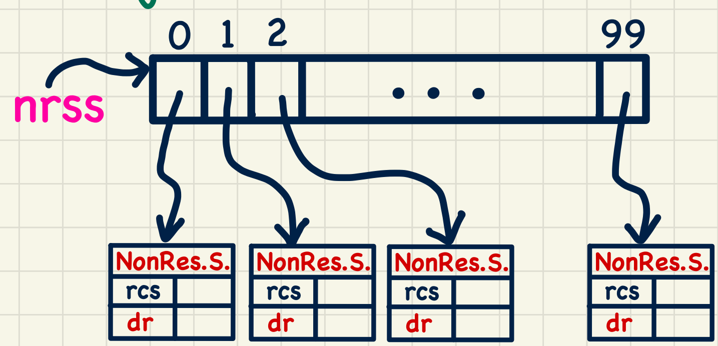


A Collection of Students (without inheritance)

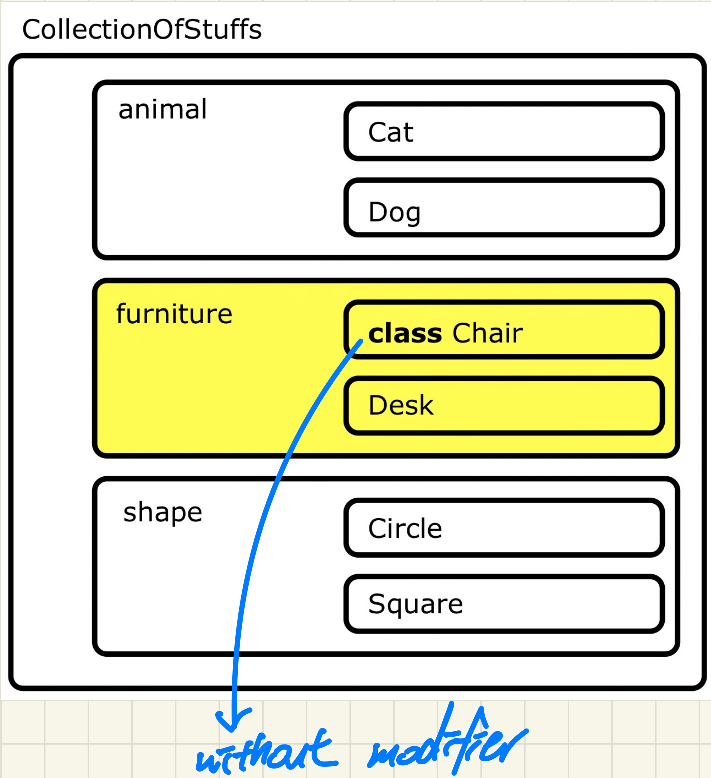
```
public class StudentManagementSystem {
  private ResidentStudent[] rss;
  private NonResidentStudent[] nrss;
  private int nors; /* number of resident students */
  private int nonrs; /* number of non-resident students */
  public void addRS(ResidentStudent rs){ rss[nors]=rs; nors++; }
  public void 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); }
  }
}
```

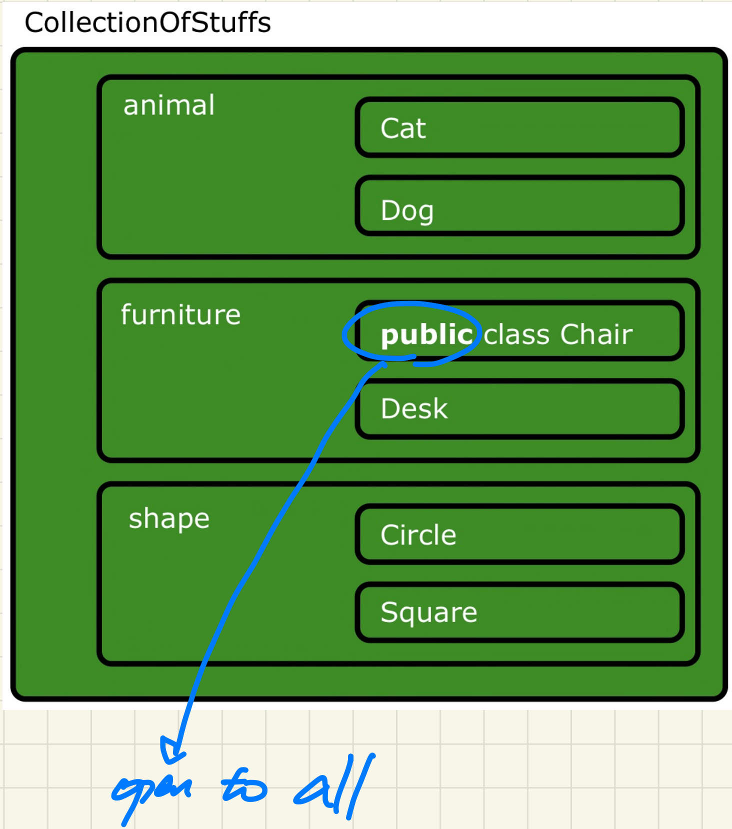
↳ multiple, duplicated loops are necessary ∵ multiple arrays



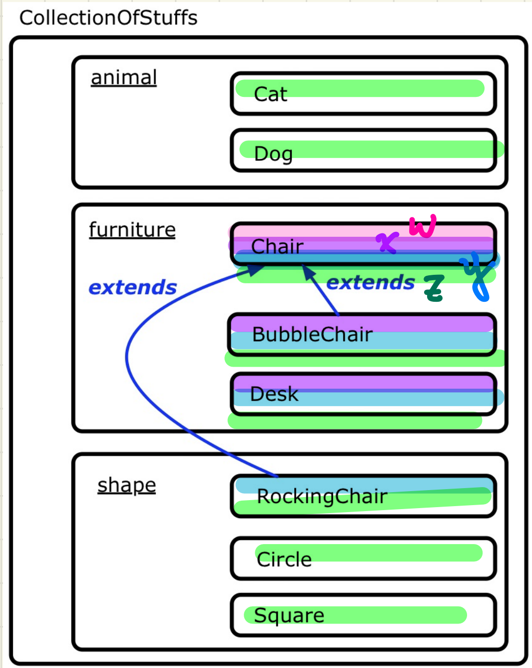


Visibility: Classes





Visibility: Attributes and Methods




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

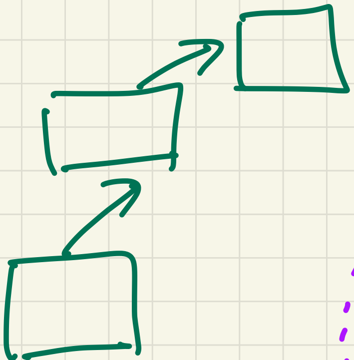
1. visible to subclasses in either the same package or other packages.

2. other classes in the same package

as ↑: no modifier + subclasses in other packages.

	CLASS	PACKAGE	SUBCLASS (same pkg)	SUBCLASS (different pkg)	NON-SUBCLASS (across Project)
public					
protected					
no modifier					
private					

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 */
  }
}
```

declare what's in common among subclasses
↳ RS
↳ NRS

base

parent version

base amt. calculation (shared among classes)

inherited, overridden version

no need to re-declare name, regCourses, nOC, register

inherit everything

parent

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

calling the constructor in parent 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;
  }
}
```

return the base amt. calculation from parent class

Lecture 17 - Nov 9

Inheritance

Code Reuse
Static Types & Expectation
Intuition: Polymorphism
Intuition: Dynamic Binding

Announcements

- **ProgTest2**: postponed to Tuesday, November 15
- **Lab3** due today at 2pm

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 */
  }
}
```

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

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

** new att & new meth declared in subclasses are not available in parent class.

* new method! void setPr(...) not inherited to parent class

inherited but not overridden

inherited & overridden

common code inherited to all subclasses

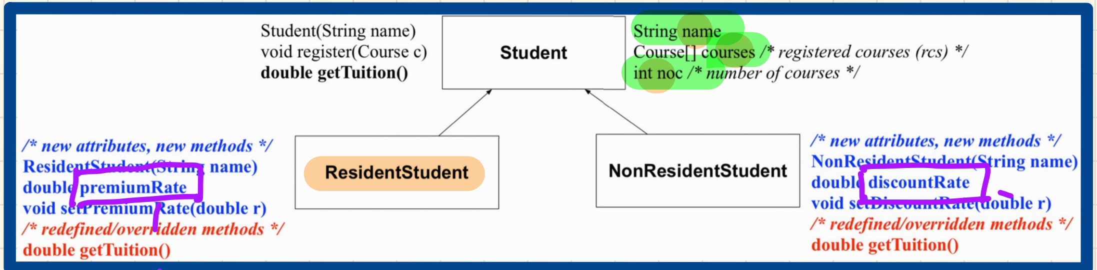
overriding inherited methods.

* new attributes

Student s = new Student(..);
s.setPremiumRate(1.25); X

outside expectation of Student.

Visualizing Parent and Child Objects



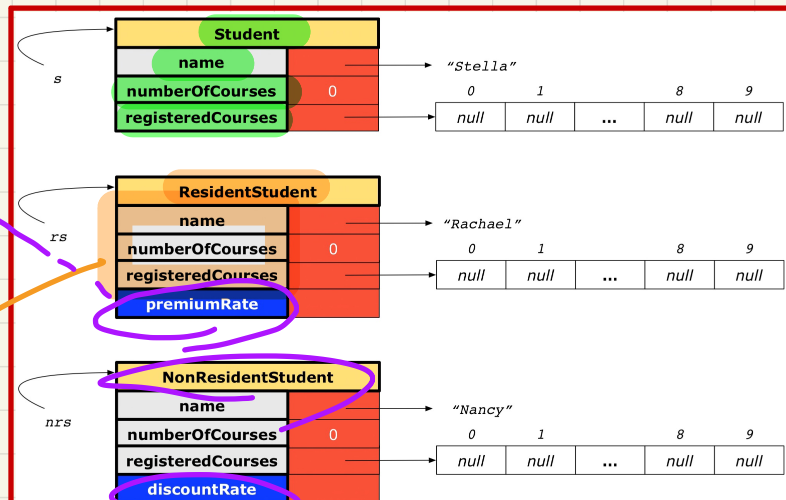
Inheritance Hirarchy

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

Declaring Static Types

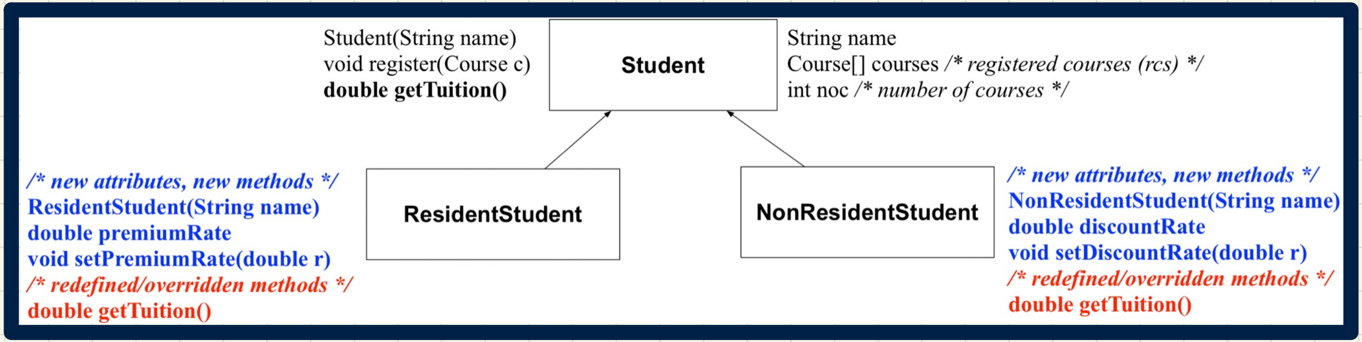
new attribute

Runtime Object Structure



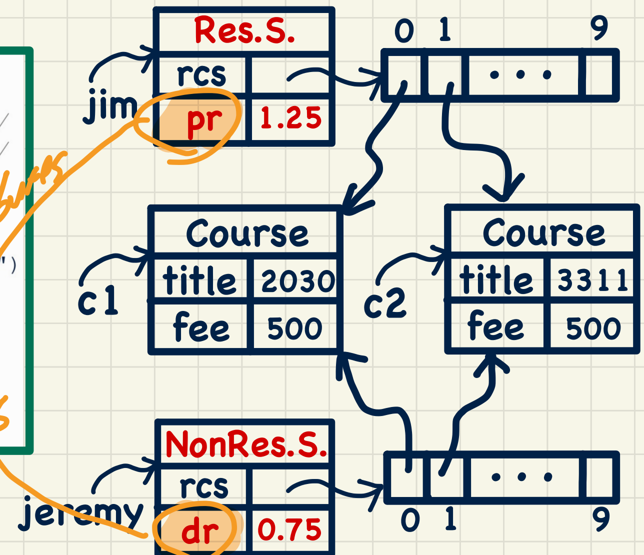
Inherited from Student class

Testing Student Classes (with inheritance)




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

new attributes declared in subclasses



what if: Student jim = new RS(..);

Student Classes (with inheritance): Expectations

dynamic types (not relevant).

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

Student

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

ResidentStudent

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

NonResidentStudent

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

declared/static types

each subclass' expectation is at least as much as its parent

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

sibling classes share expec. from parent

expectation

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

beyond parent's expec. & no overlap.

Intuition: Polymorphism

② expectation on rs:
rs.setPremiumRate(1.25).

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

Student

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

ResidentStudent

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

NonResidentStudent

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

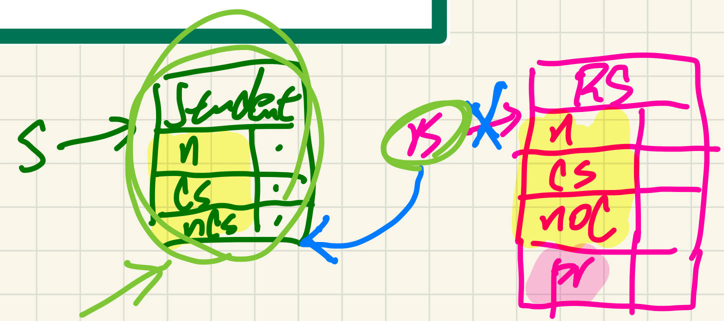
crash

↳ rs = s should be invalid

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

Assume rs = S was valid

↳ ① executing the assignment points rs to a Student obj.



Intuition: Polymorphism

② expectation on rs:
rs.setPremiumRate(1.25).

↳ crash

Student

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

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

ResidentStudent

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

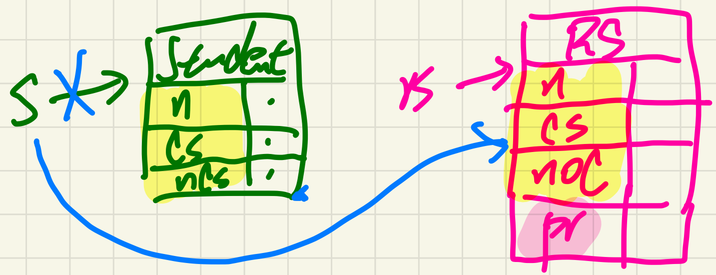
NonResidentStudent

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

↳ rs = s should be invalid

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

③ type casting can make this work



① s.setPremiumRate(1.25)
↳ not valid ∵ ST of s (Student) does not declared pr.

C_1 obj1 = C_2 obj2 ;

⋮

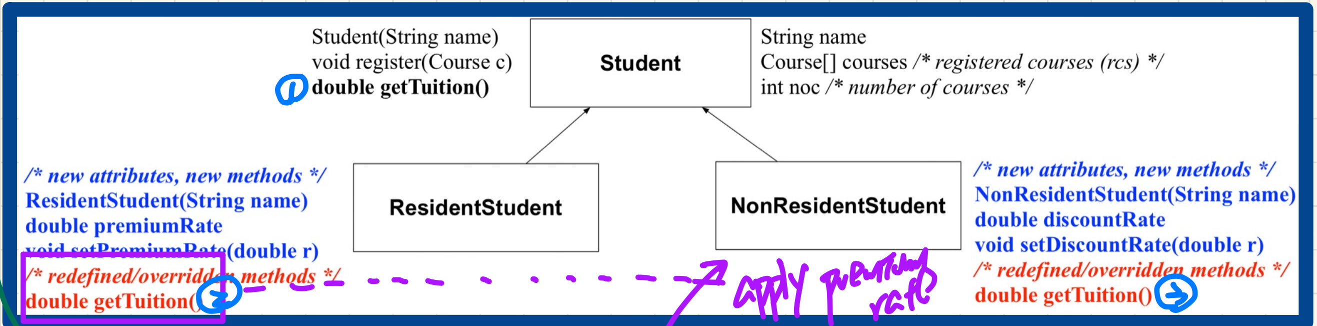
obj1 = obj2

↳ to be valid the ST of obj2 (C_2) should be a subclass of the ST of obj1 (C_1).

descendants class.

S: ↳ expectation determined by ST of C.O.

Intuition: Dynamic Binding



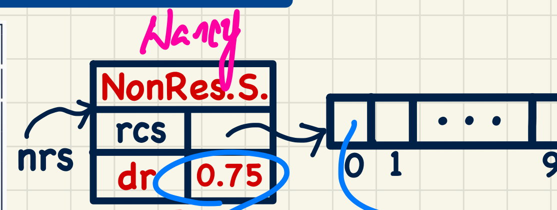

```
Course eecs2030 = new Course("EECS2030", 100.0);
Student s;
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
rs.setPremiumRate(1.25); rs.register(eecs2030);
nrs.setDiscountRate(0.75); nrs.register(eecs2030);
s = rs; System.out.println(s.getTuition()); [redacted]
s = nrs; System.out.println(s.getTuition()) [redacted]
```

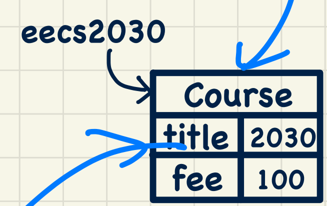
ST

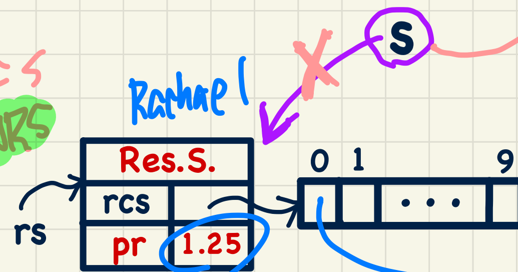
changes the dynamic type of S from RS to NRS

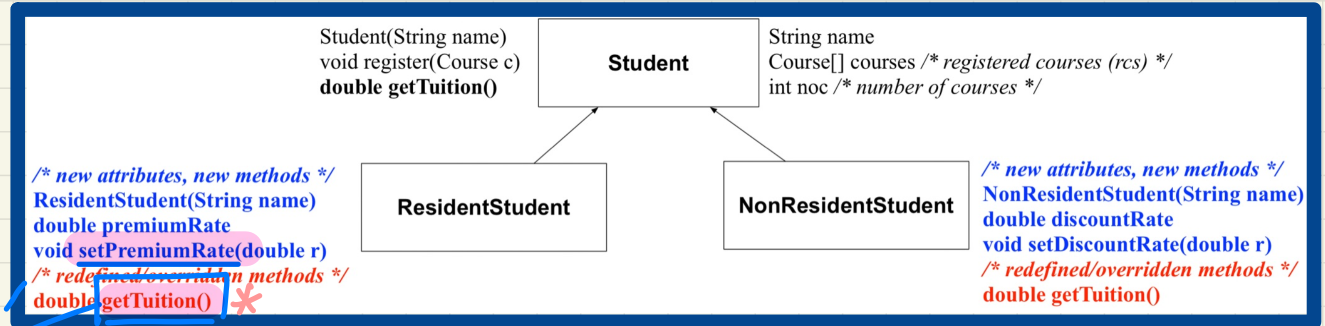
Dynamic type of S becomes RS

DT of S is NRS







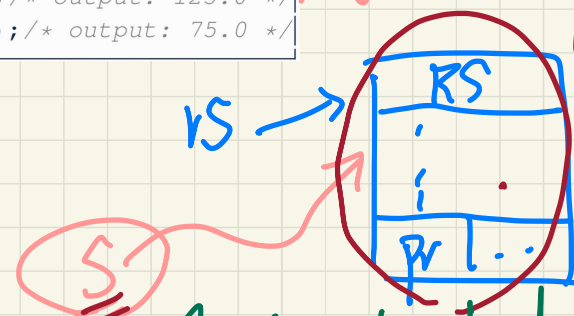



```java
Course eecs2030 = new Course("EECS2030", 100.0);
Student s;
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
rs.setPremiumRate(1.25); rs.register(eecs2030);
nrs.setDiscountRate(0.75); nrs.register(eecs2030);
s = rs; System.out.println( s .getTuition());/* output: 125.0 */
s = nrs; System.out.println( s .getTuition());/* output: 75.0 */
```

DT of s is RS

↳ version of getT() in RS will be invoked

```
getTuition ( ) {
  :
  this.setPremiumRate(..);
}
```



↳ implicitly call setP on RS object

On the other hand:
s.setPremiumRate X

PointV1 p1 = [. . .] .

PointV2 p2 = ...

① assertEquals(p1, p2)

↳ p1.equals(p2) → invoke default version in Object

↳ p1 == p2

② p1 == p2

1. Whether a line should compile?

Look at static type

2. Which version of method should be invoked?

Look at dynamic type

Lecture 18 - Nov 14

Inheritance

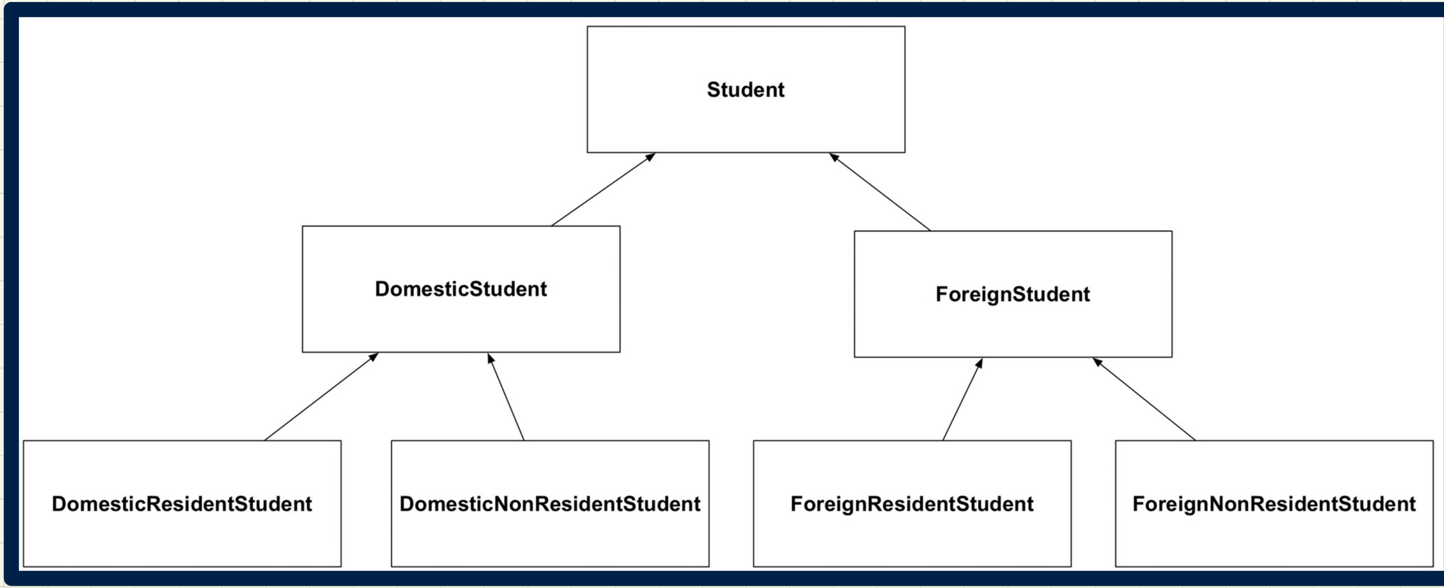
Rules of Substitutions
Static Types vs. Dynamic Types

Announcements

- ProgTest2: this Tuesday, November 15
- Lab4 released

ProgTest3

Multi-Level Inheritance Hierarchy: Students

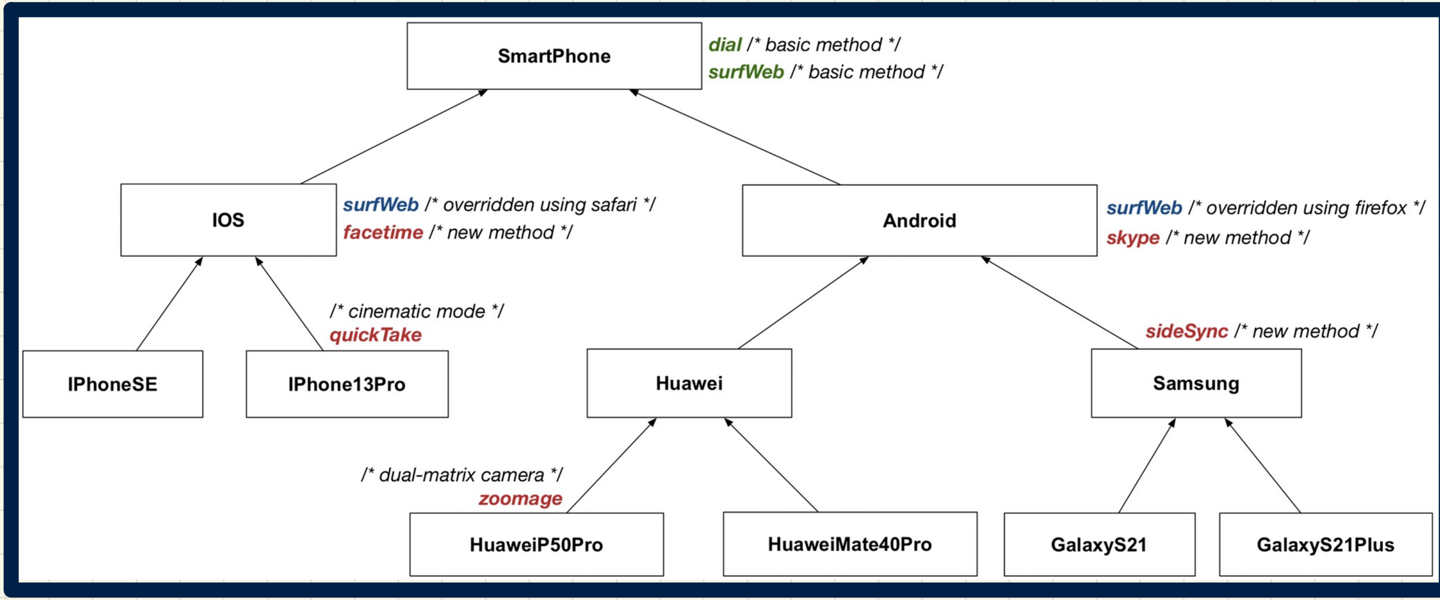


Reflections:

kind

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

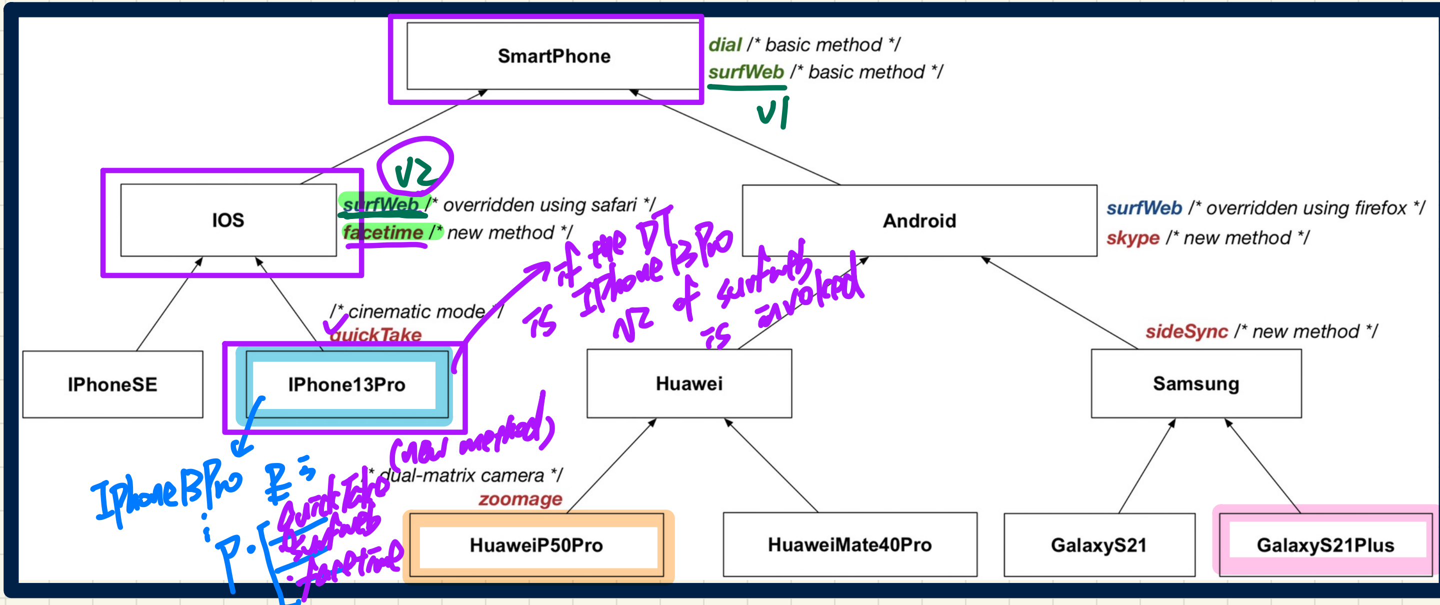
Multi-Level Inheritance Hierarchy: Smartphones



Reflections:

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

Multi-Level Inheritance Hierarchy: Smartphones



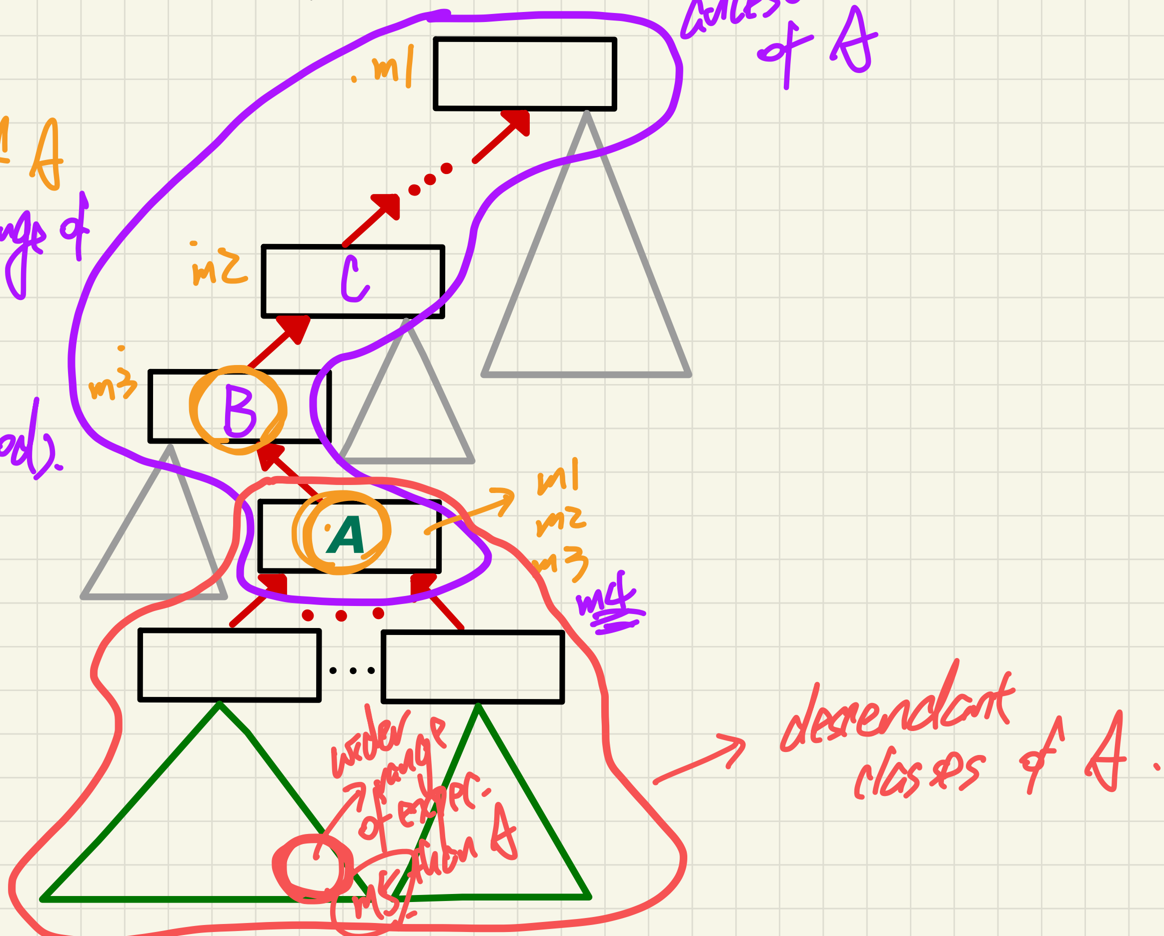
Exercise Compare the ranges of expectations of:

+ IPhone13Pro
+ HuaweiP50Pro
+ GalaxyS21Plus

exercise.

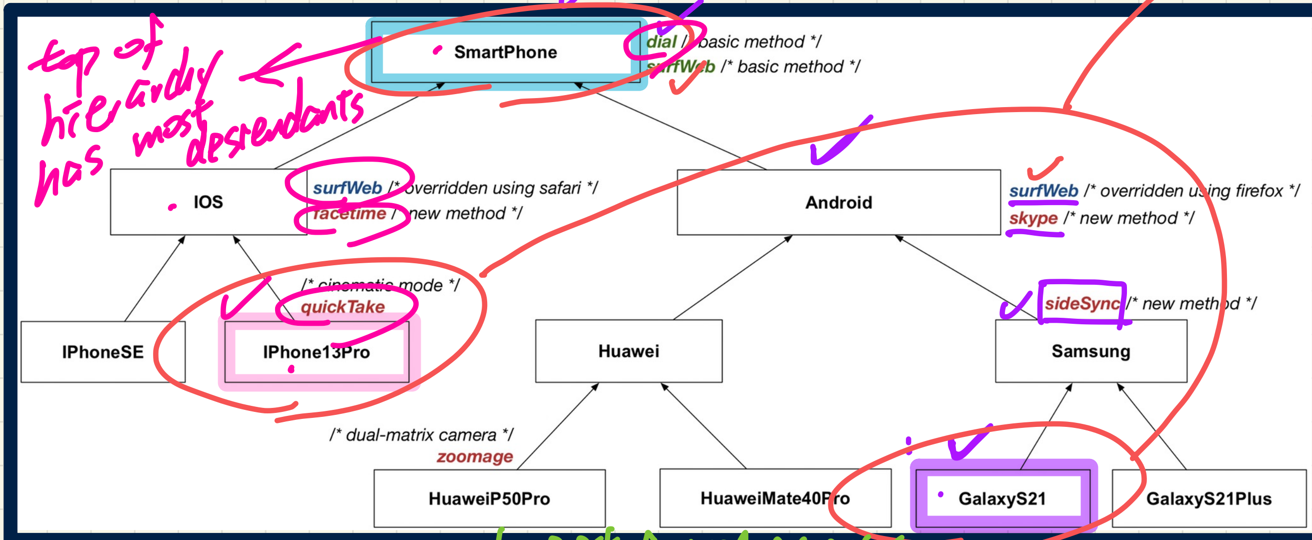
Inheritance Forms a Type Hierarchy

B is an ancestor of A
⇒ A has wider range of expectation than B
(e.g. m4
↳ new method).



Inheritance Accumulates Code for Reuse

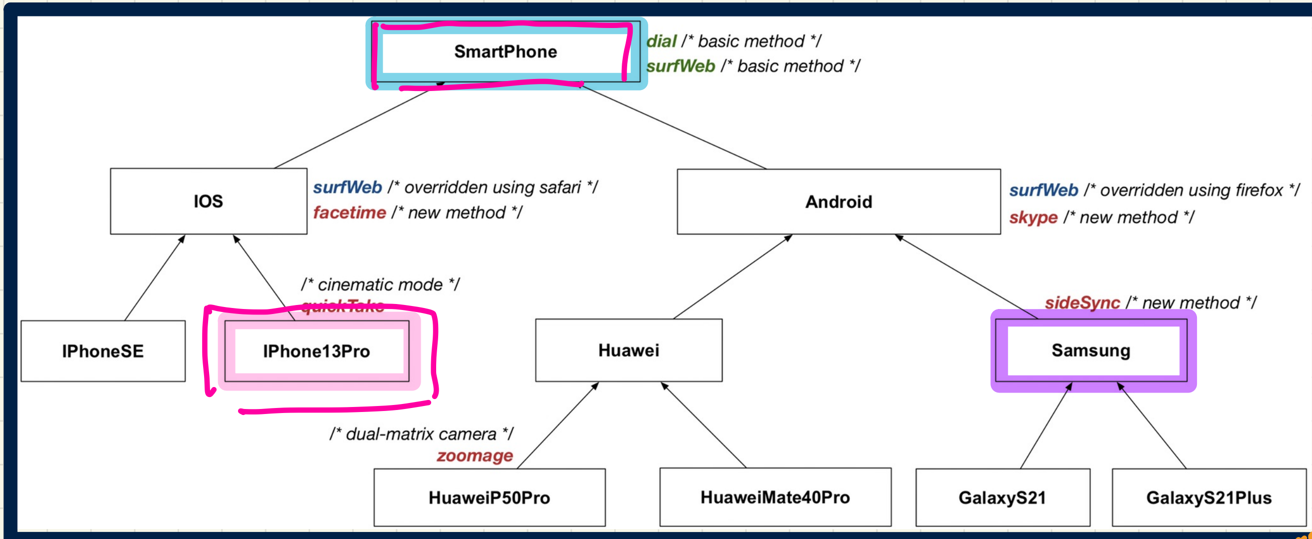
share expectations inherited from their lowest common ancestor



lowest common ancester

	ancestors	expectations	descendants
	S21, Sam., And., SP	sideSync, skype, surfWeb overridden, dial.	
	IP13Pro, IOS, SP	quickTake, facetime, surfWeb overridden, dial	exp. from LCA.
	exercise.		

Inheritance Accumulates Code for Reuse




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

SP.

sp1 SP

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

IP13Pro SP

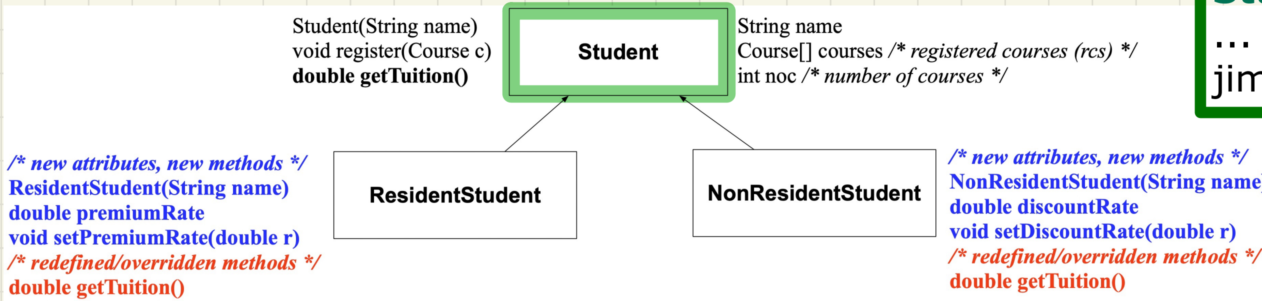
sp2 = sp1 X

sp2 = sp2 ✓

sp2 = sp3 X

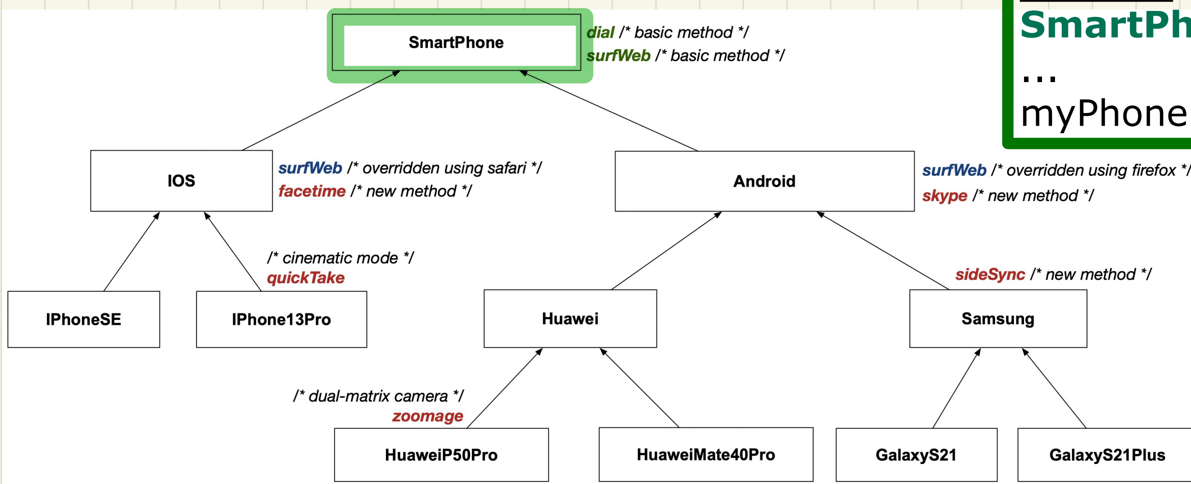
Static Types determine Expectations

Inheritance Herarchy: Students



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

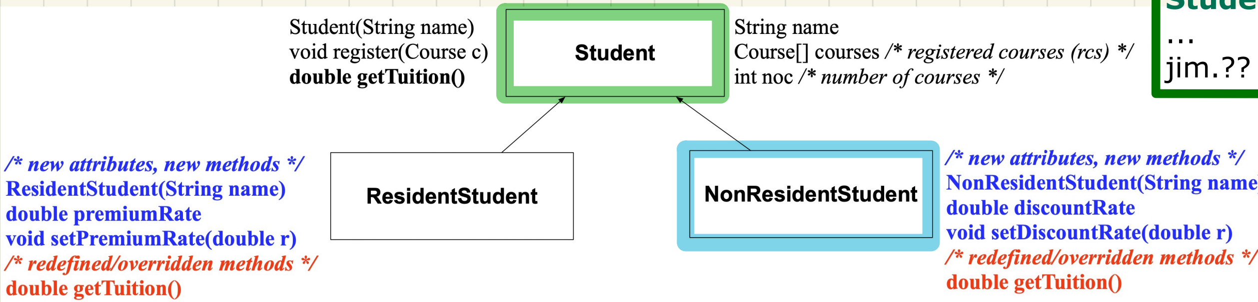
Inheritance Herarchy: Smart Phones



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

Static Types determine Expectations

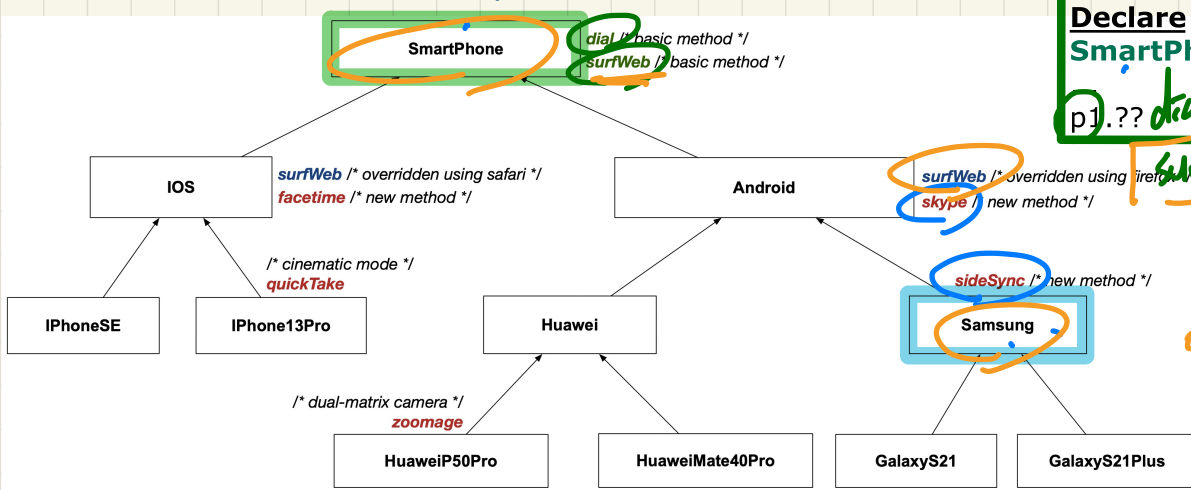
Inheritance Herarchy: Students



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

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

Inheritance Herarchy: Smart Phones



Declare:
SmartPhone p1;

p1.?? dial :
surfWeb :
not overridden

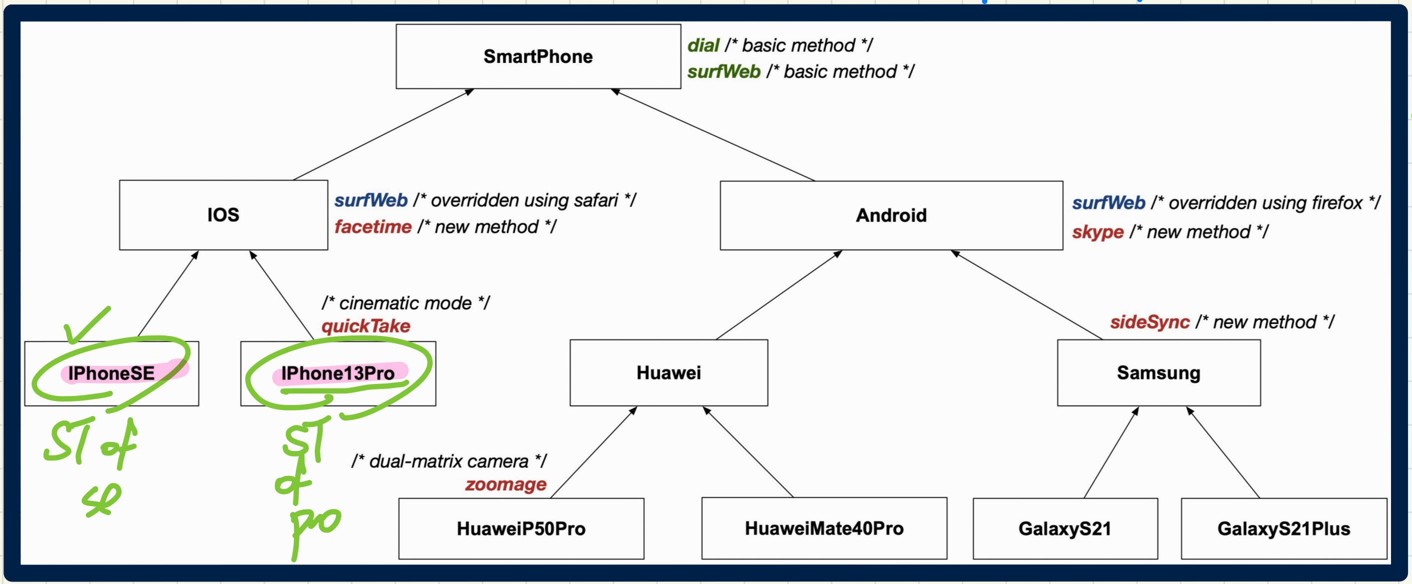
Declare:
Samsung p2;

p2.?? dial
surfWeb
skype
sideSync
overridden version.

Rules of Substitutions (1)

IPhoneSE se; ① se = pro; ✗

IPhone13Pro pro; ② pro = se; ✗ Is the ST of pro (IP13Pro) a descendant of the ST of se (IPhoneSE)?



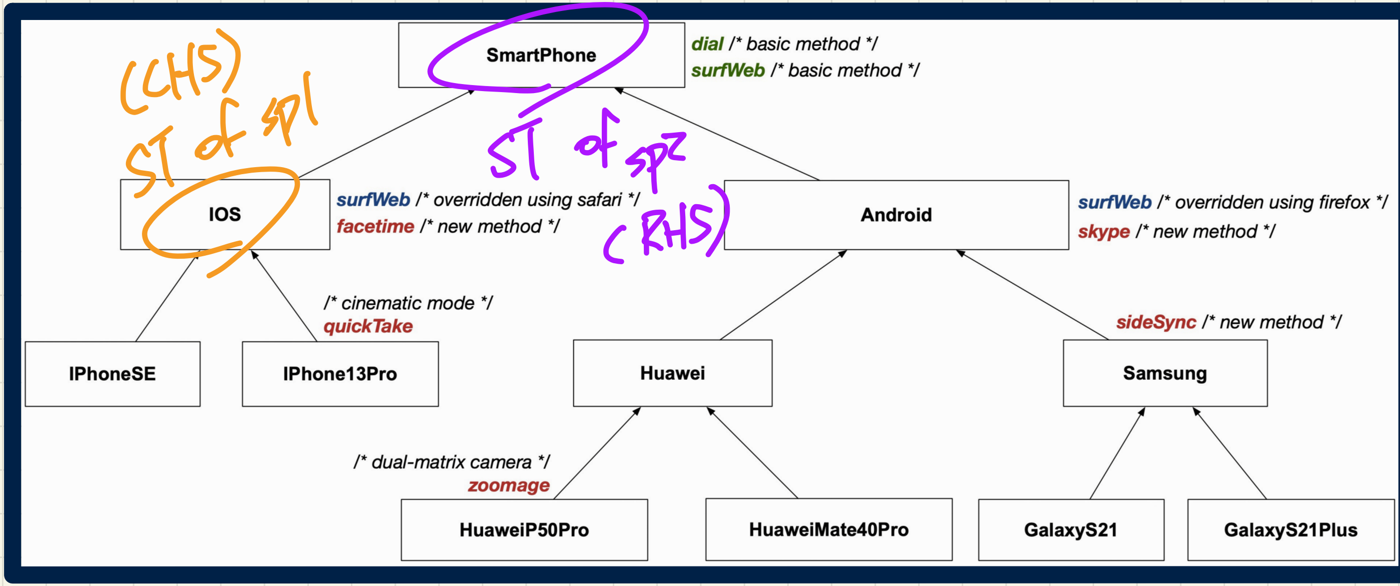
Declarations:

IOS sp1;
IPhoneSE sp2;
IPhone13Pro sp3;

Substitutions:

sp1 = sp2; ✓
sp1 = sp3; ✓

Rules of Substitutions (2)



Declarations:

IOS sp1;
SmartPhone sp2;

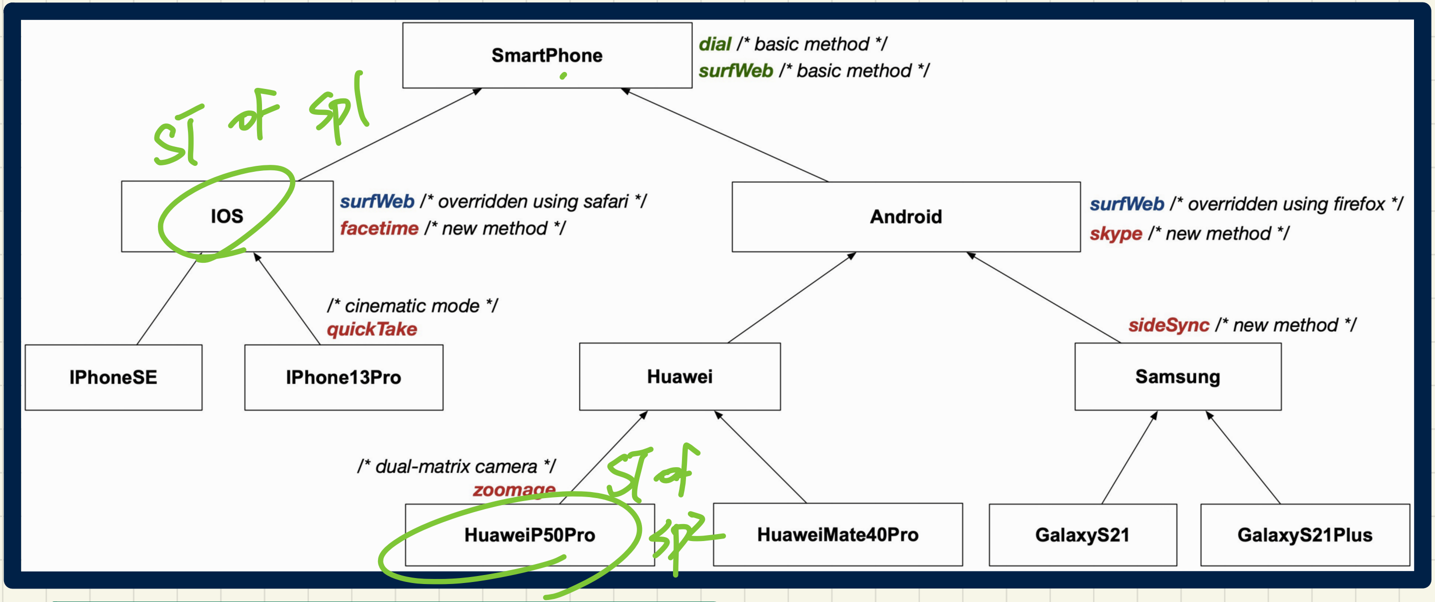
Substitutions:

sp1 = sp2; X

ST IOS

ST SP

Rules of Substitutions (3)



Declarations:

IOS sp1;

HuaweiP50Pro sp2;

Substitutions:

sp1 = sp2; X

Visualization: Static Type vs. Dynamic Type

Declaration:

```
Student s;
```

static type

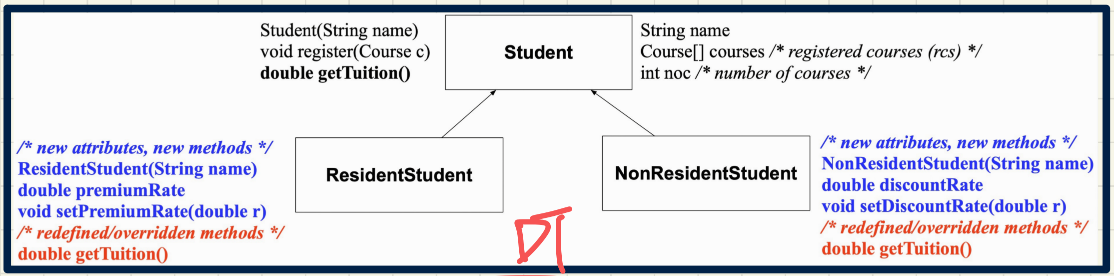
Substitution:

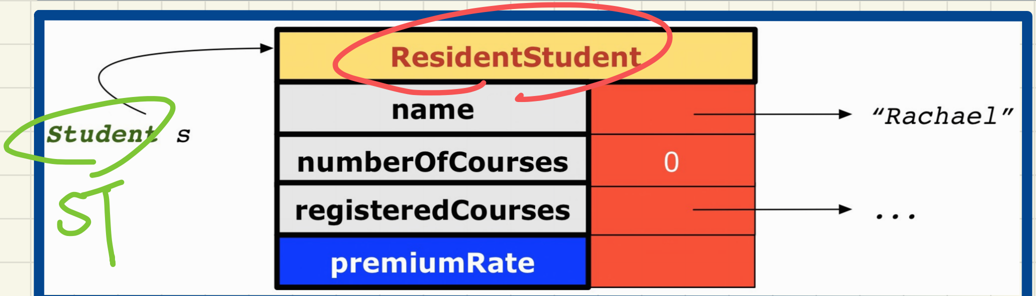
dynamic type

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

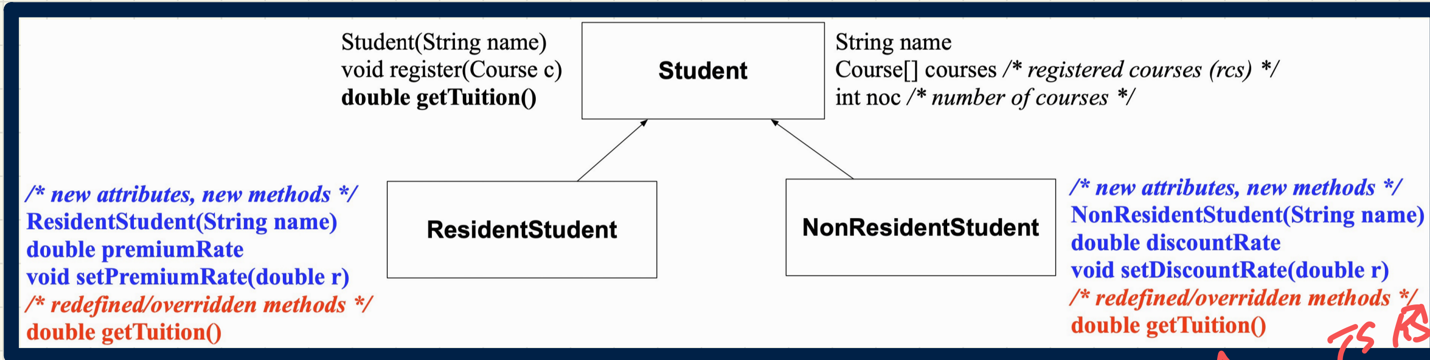
Static Type: Expectation

Dynamic Type: Accumulation of Code





Change of Dynamic Type (1.1)



Example 1:

DT1

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

DT2

DT of jim

jim

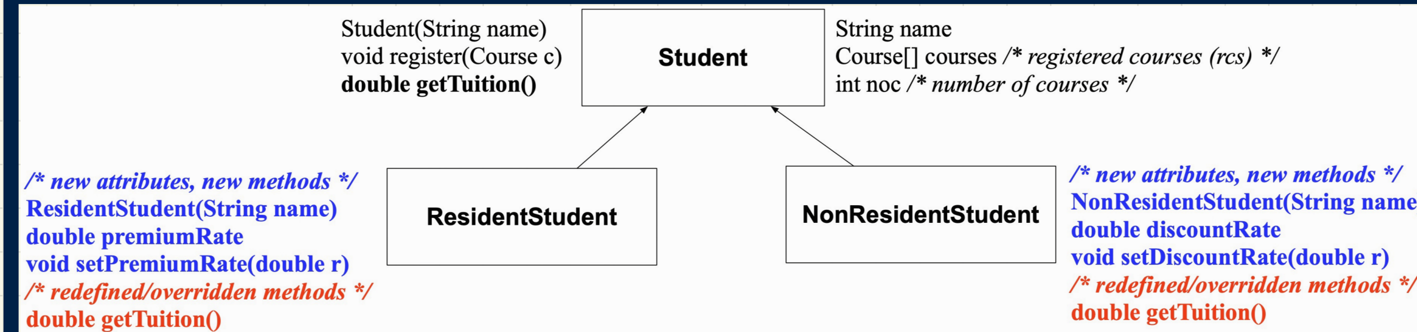
RS

NRS

DT of jim changes to NRS

Rule: New DT must fulfill the expectations on the ref. var's ST ⇒ new DT is a descendant of ref var's ST.

Change of Dynamic Type (1.2)



Example 2: ✓

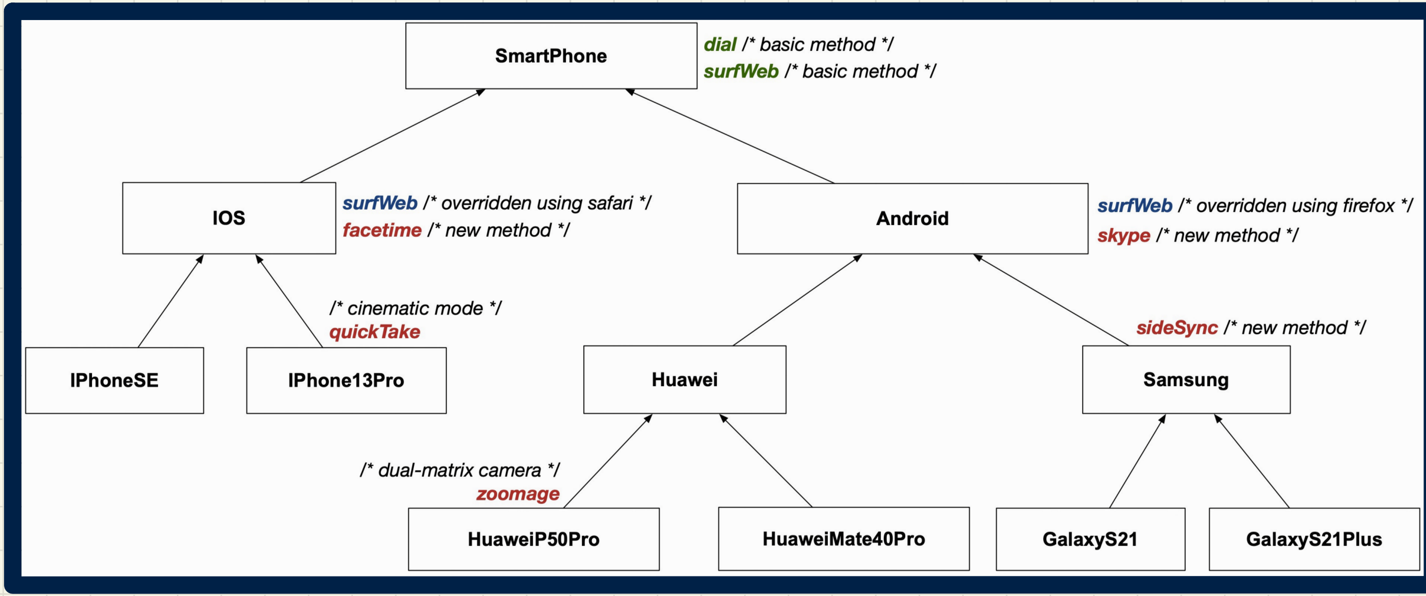
```
ResidentStudent jeremy = new Student(...);
```

Not valid

① Student is not a descendant of RS

② Student cannot fulfill exp. of RS.

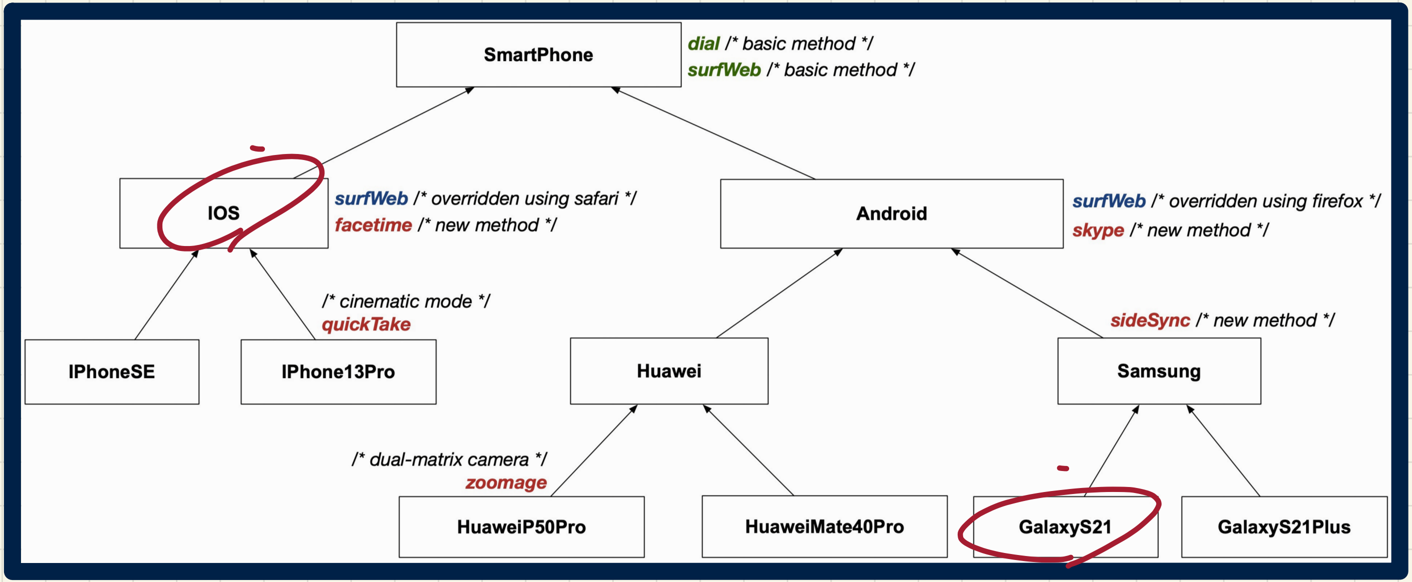
Change of Dynamic Type: Exercise (1)



Exercise 1:

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

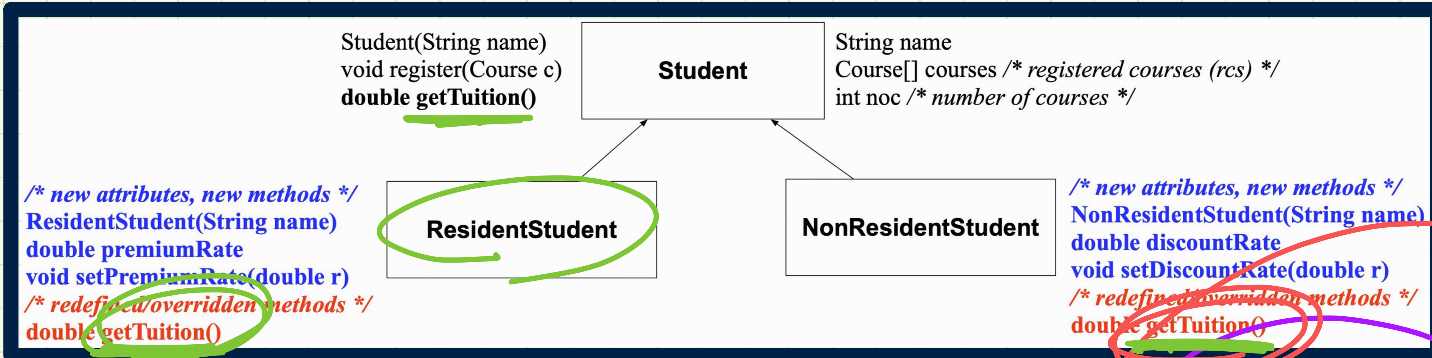
Change of Dynamic Type: Exercise (2)



Exercise 2:

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

Change of Dynamic Type (2.1)



Given:

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

jim rs nrs

Example 1:

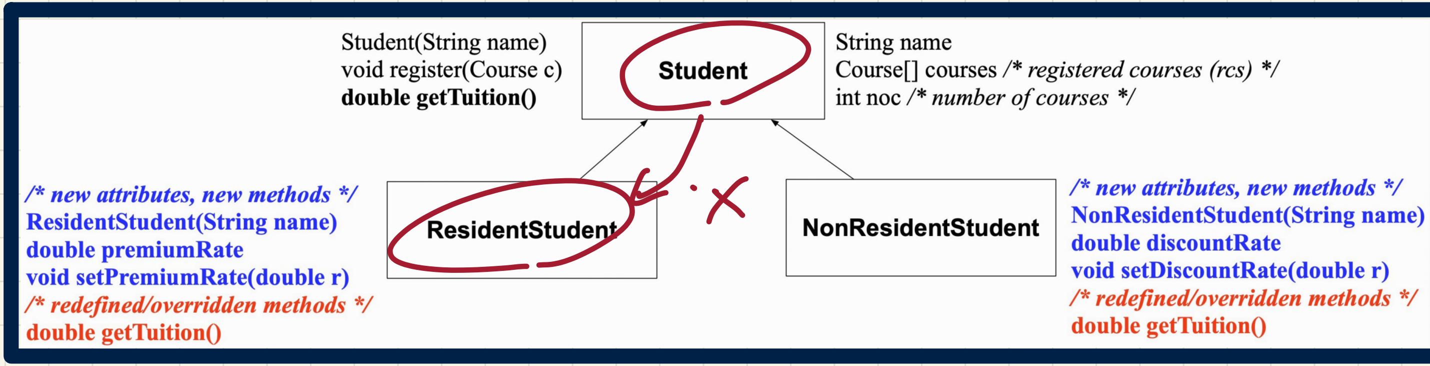
```
① jim = rs; ②
println(jim.getTuition());
jim = nrs; ③
println(jim.getTuition());
```

RS is DT

NRS is DT

	jim's ST	jim's DT
①	Student	Student
②	Student	RS
③	Student	NRS

Change of Dynamic Type (2.2)



Given:

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

Example 2:

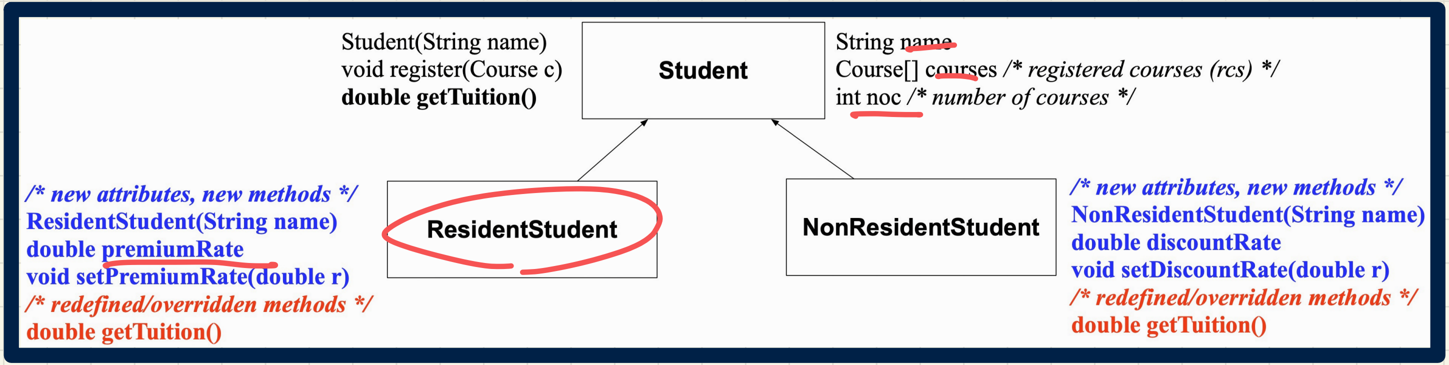
```
rs = jim;
println(rs.getTuition());
nrs = jim;
println(nrs.getTuition());
```

which version?

which version?

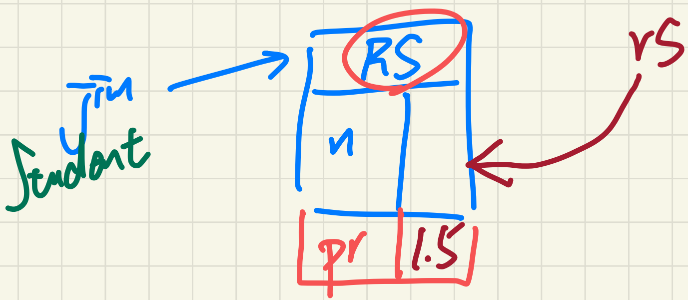
ST of jim (Student) not descendant of ST of rs (RS)

Type Cast: Motivation

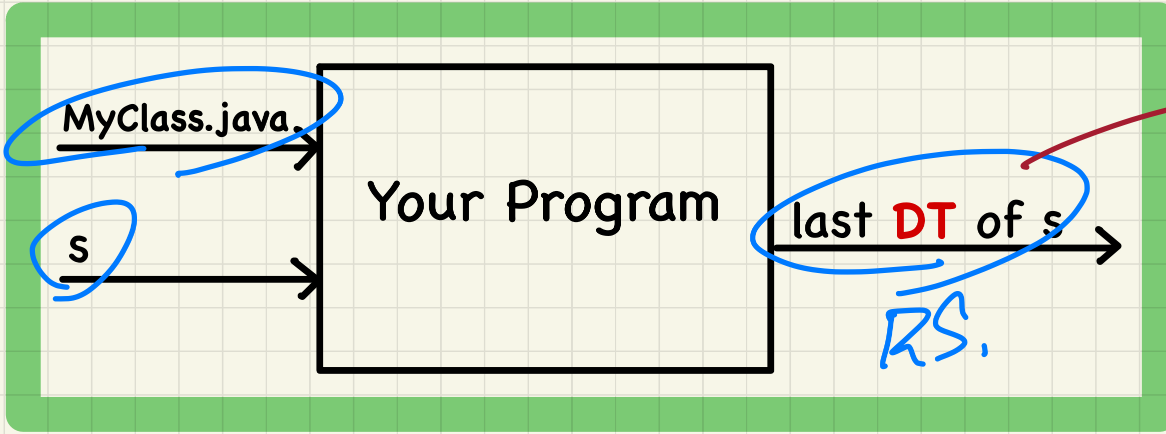



```
Student jim = new ResidentStudent("J. Davis");
ResidentStudent rs = jim;
rs.setPremiumRate(1.5);
```

ST: Student



An A+ Challenge: Inferring the DT of a Variable



undecidable to figure out the dynamic behaviour of a program

EECS 2001

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

last DT

Lecture 19 - Nov 16

Inheritance

Type Casting: Upward vs. Downward
Danger of Casts: ClassCastException

Announcements

- **Lab4** released

Recap: Static Types vs. Dynamic Types

```
C1 v1 = new C3(...);
C2 v2 = new C4(...);
v1.m();
v2.m();
v1 = v2;
v1.m();
v2.m();
```

static types

DT of v1 is C3

DT

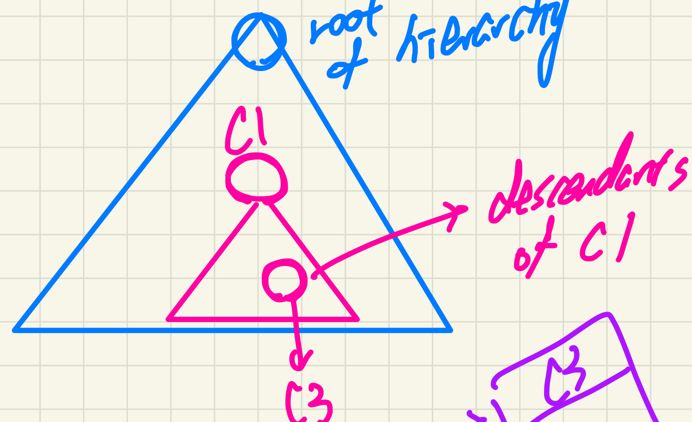
C3 should be a descendant of C1

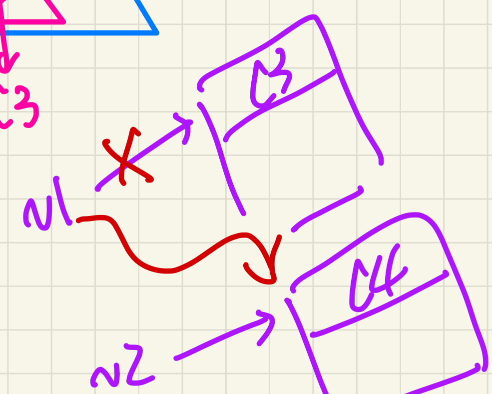
ST

↳ C3 can fulfill C1's expectation

DT of v1 changes to C4

v2's ST should be a descendant of v1's ST





Exercises on Eclipse:

+ SMS (variable assignments)
+ Smart Phones (hierarchy + variable assignments)

variable assignments

version of methods invoked.

Polymorphism and Dynamic Binding

Polymorphism:

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

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

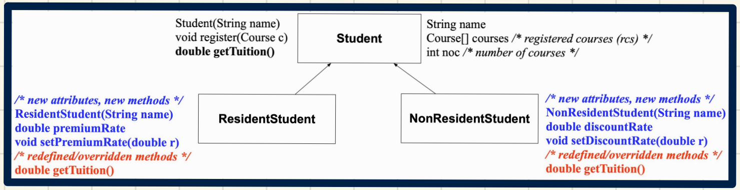
Dynamic Binding:

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

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

DT: RS

DT: NRS



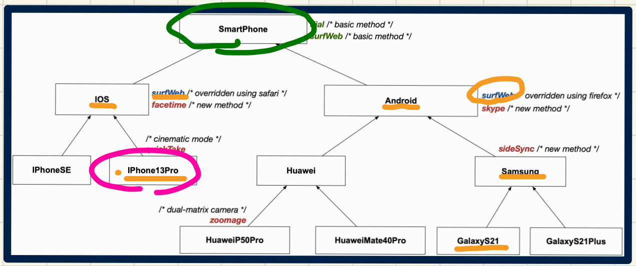

```
SmartPhone sp1 = new IPhone13Pro(...);
SmartPhone sp2 = new GalaxyS21(...);
sp1.surfWeb();
sp1 = sp2;
sp1.surfWeb();
```

→ DT of sp1 is IPhone13Pro.

DT: GalaxyS21

→ DT of sp1 is GalaxyS21

DT will become the DT of sp2.



Anatomy of a Type Cast

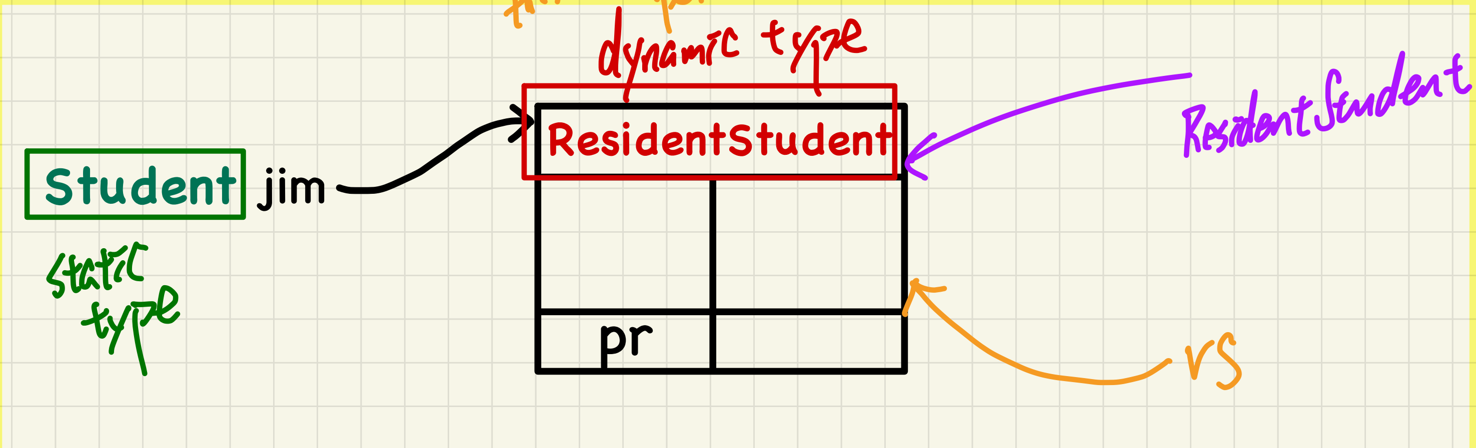
Student jim = **new** **ResidentStudent**("Jim");

type of the cast expression corresponds to the cast class: RS.

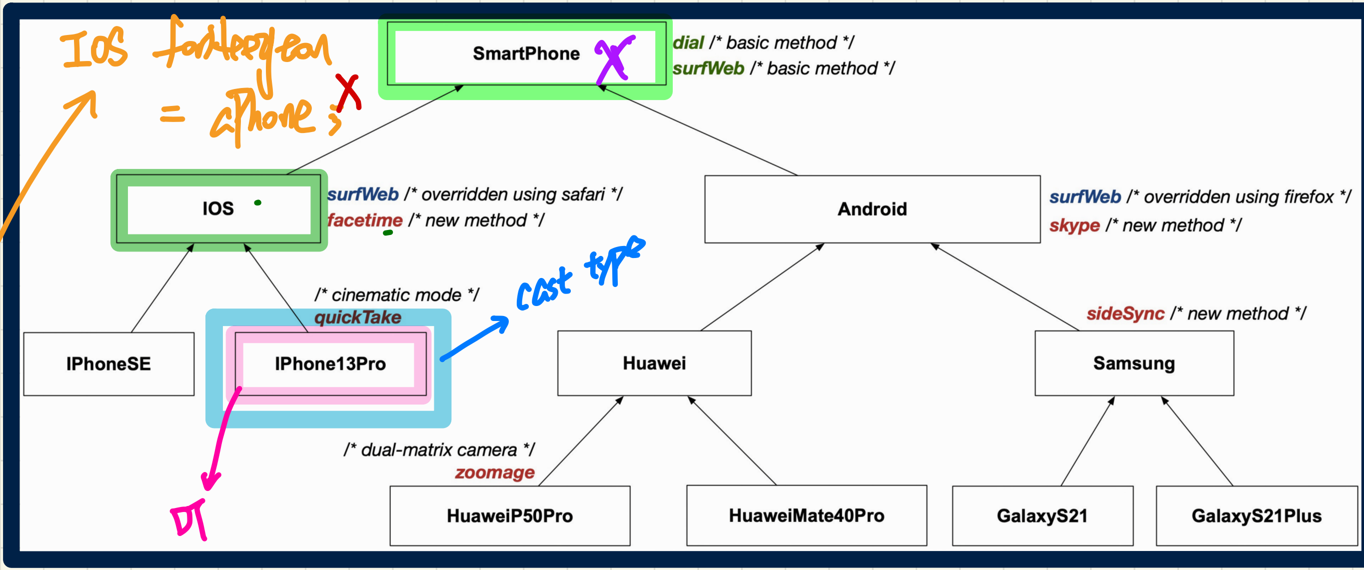
ST of a new alias

$$\overbrace{\textit{ResidentStudent rs}}^{\texttt{ST: ResidentStudent}}\ \overbrace{=}^{\texttt{valid substitution}}\ \underbrace{(\texttt{ResidentStudent})\ \overbrace{\textit{jim}}^{\texttt{ST: Student}}}_{\texttt{an alias whose ST is ResidentStudent}}\ ;$$

rs points to wherever this new alias points to



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();
```

ST aPhone → IPBP

IPBPro

ST: IOS includes facetime as exp

Anonymous Cast: Use the cast result directly.

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

Exercise

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

type: IP13Pro

① ((IPhone13Pro) aPhone).facetime();

② (IPhone13Pro) aPhone.facetime(); ST of aPhone only look at STs.

Compilable Casts: Upwards vs. Downwards

→ restrict expectation.

→ polymorphism

DT irrelevant for deciding if a cast compiles w.r.t. a variable's ST.

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

result of upward cast → fewer exp.

```
SmartPhone sp = (SmartPhone) myPhone;
```

(2)

```
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
```

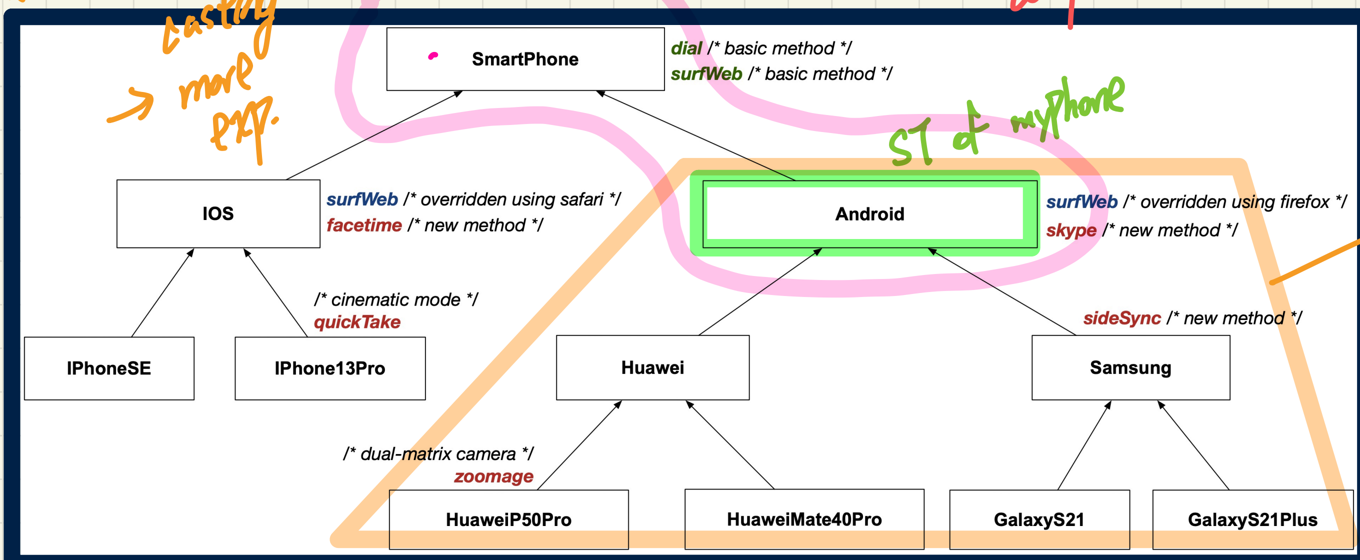
(3)

result of downward casting → more exp.

ancestor classes.

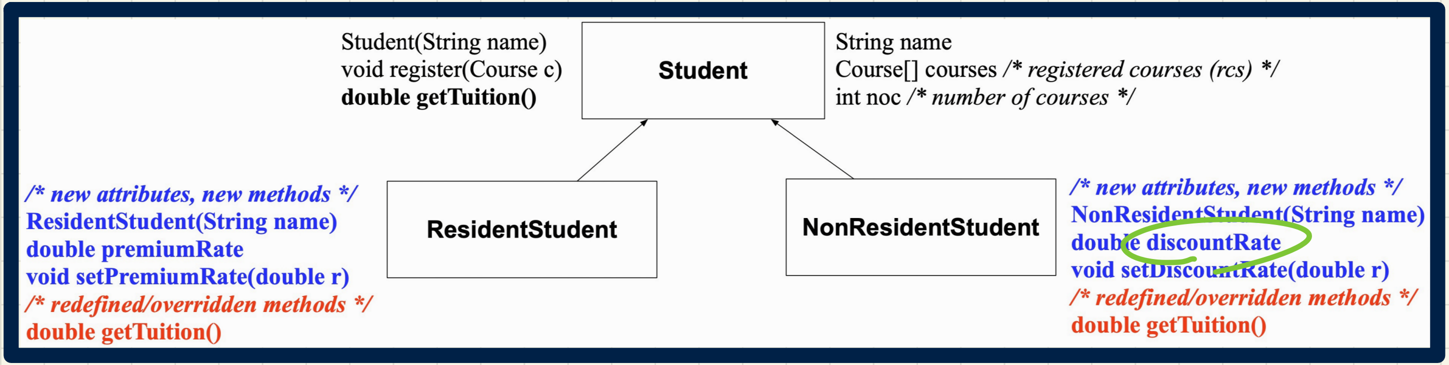
Expectations

	sp	myPhone	ga
dial	✓	✓	✓
surfWeb	✓	✓	✓
skype	✗	✓	✓
sideSync	✗	✗	✓
facetime	✗	✗	✗
quickTake	✗	✗	✗
zoomage	✗	✗	✗



polymorphism: descendants of myPhone's ST can be its DTs.

Compilable Type Cast May Fail at Runtime (1)




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

downward cast

② the DT of jim (NRS) cannot fulfill expectations of RS

ST: RS

① + ②
↳ ClassCastException

Student jim

NRS

dr [..

rs RS

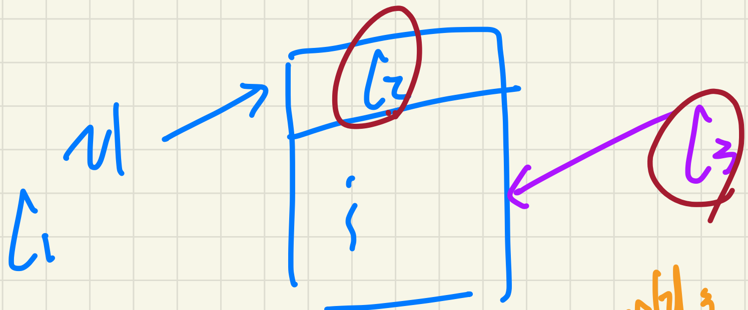
① by casting jim to an alias of ST RS, we intend to call methods within the exp. of RS

ST: C1
DT: C2

$$C1\ v1 = new\ C2(..);$$

$$C3\ v2 = \boxed{(C3)\ v1};$$

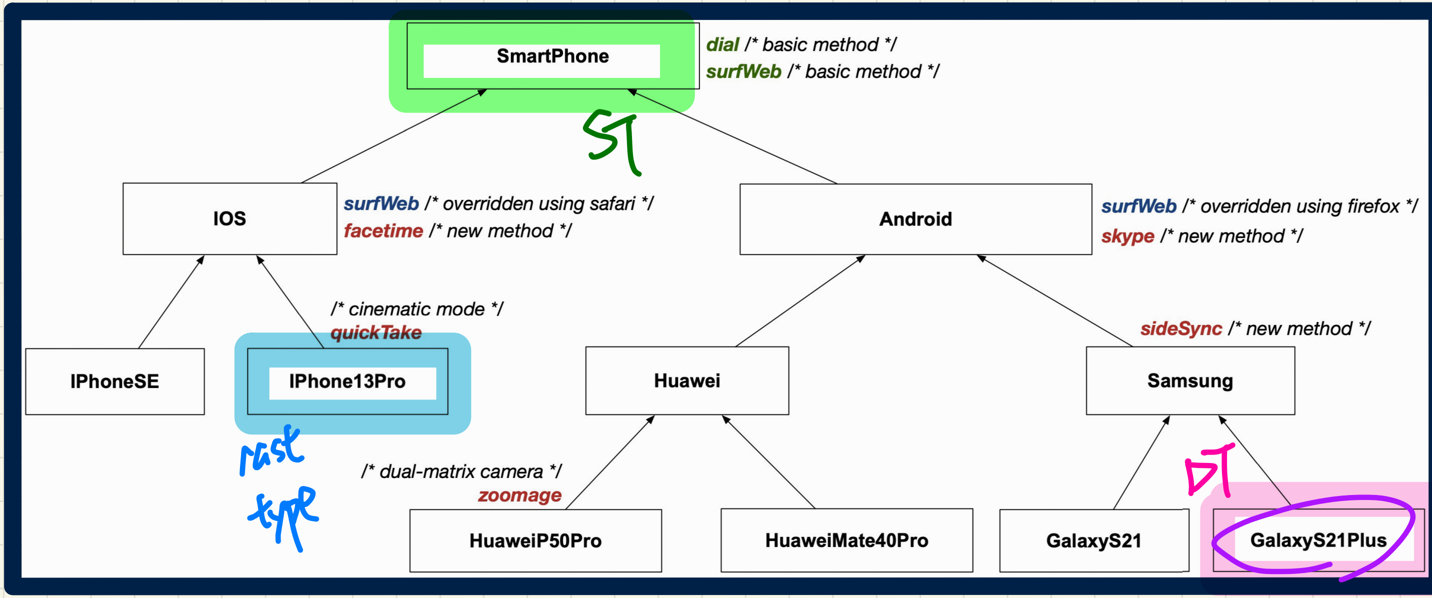
compiles if it's either upward or downward casting.



a ClassCastException occurs if DT of v1 (C2) cannot fulfill expectation of cast type (C3)

more precisely, when C2 is not a descendant of C3.

Compilable Type Cast May Fail at Runtime (2)




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

compiles ∵ downward cast

① DT of aPhone: GalaxyS21Plus

② cast type has expectation of IPhone13Pro

③ CCE ∵ DT cannot fulfill exp. of cast type.

Every line compiles!

within exp. of ST of c.o. (forHeeyeon)

Lecture 20 - Nov 21

Inheritance

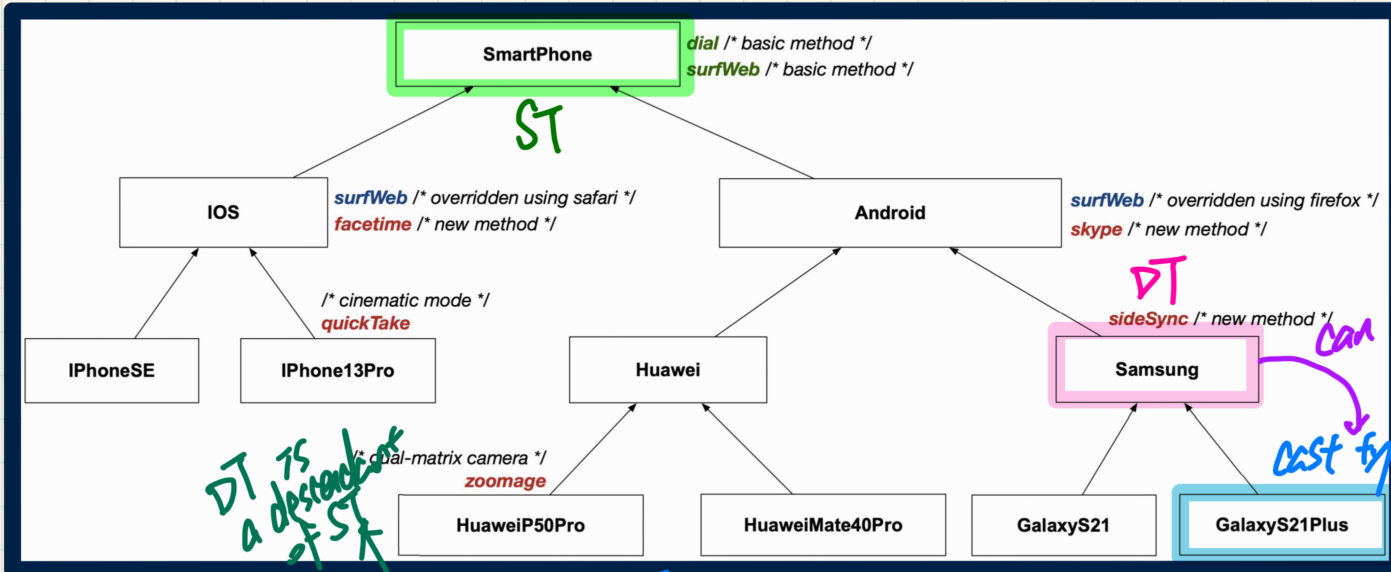
Type Cast: Compilable vs. Exception-Free
Checking DTs: instanceof Operator
Polymorphic Method Parameters

basis for ProgTest3

Announcements

- Lab4 due soon! tmrw.
- WT3 and ProgTest3 approaching...

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



ST DT

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

compiles

→ ① compile? ✓ downward cast

② ClassCastException ?

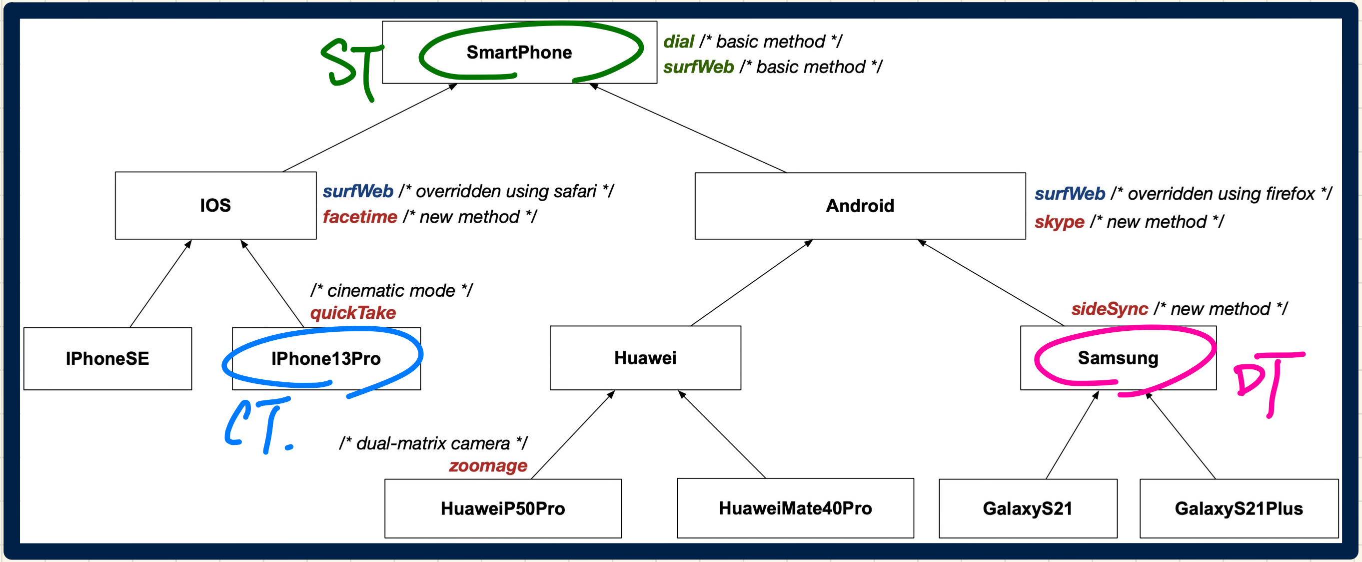
Compilable? **ClassCastException at runtime?**

∵ cast type is a descendant of the ST.

checked by instanceof operator

Can the DT fulfill the cast type?

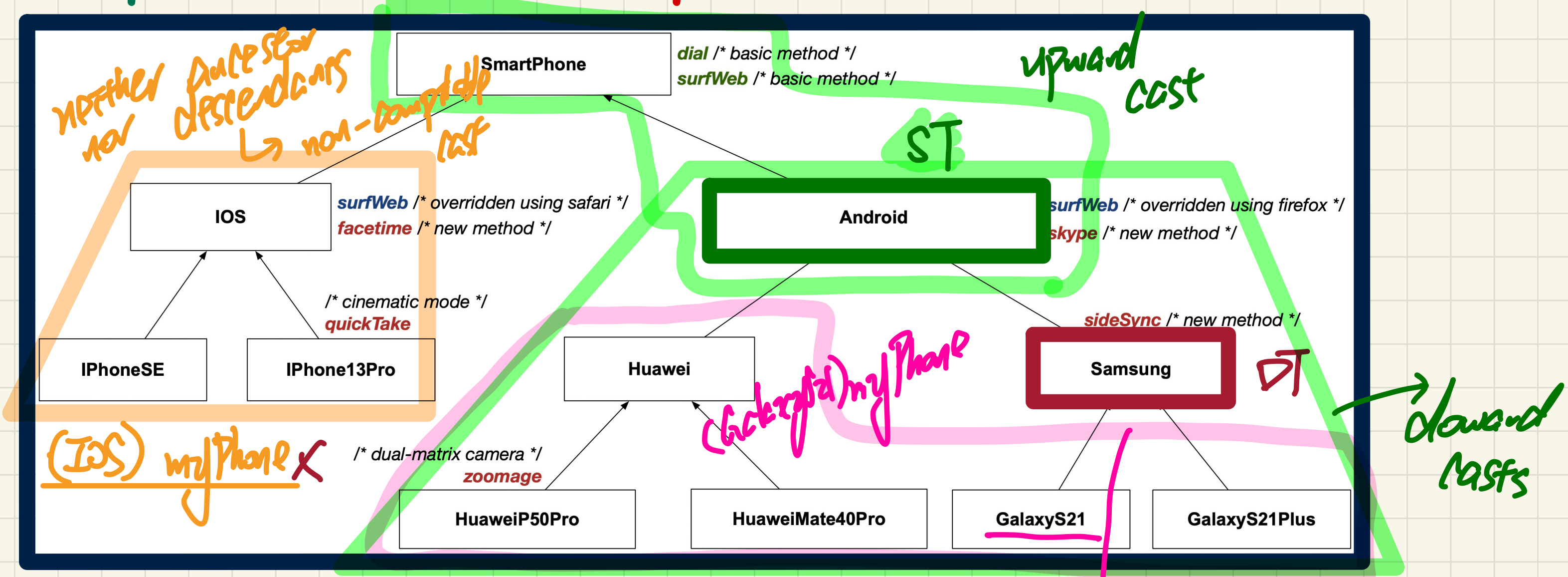
Exercise: Compilable Type Cast? Fail at Runtime? (2)




```
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



ST DT

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

compilable casts but ClassCastEx. at runtime

Compilable Casts ST

Non-Compilable Casts

Exception-Free Casts

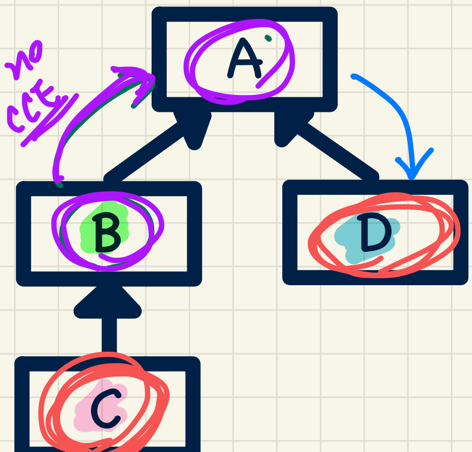
ClassCastException

Exercise: Compilable Cast vs. Exception-Free Cast

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

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

non-compilable cast.



at runtime ↳ CCE ∵ DT of b (which is C) is not a descendant of the cast type (D).

ST: A.

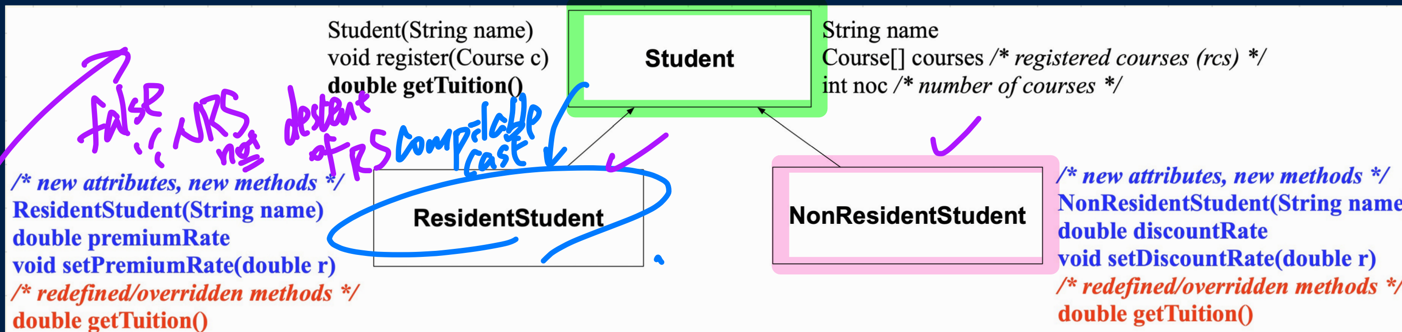
$(D)\ ((A)\ b)$

compilable!

↳ given any two classes X, Y: a . b

$(Y)\ ((Object)\ a)$

Checking Dynamic Types at Runtime (1)



false ∵ NRS not descendant of RS

compilable cast

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

a class name

ST: Student

DT: NRS

CCE prevented!

① ref. variable
② dot notation jim.spouse.

CCE ∵ ① DT NRS not descendant of RS
② DT NRS cannot fulfill expect of cast type RS

Boolean expression

$\boxed{\text{obj} \quad \text{instanceof} \quad \text{class}}$

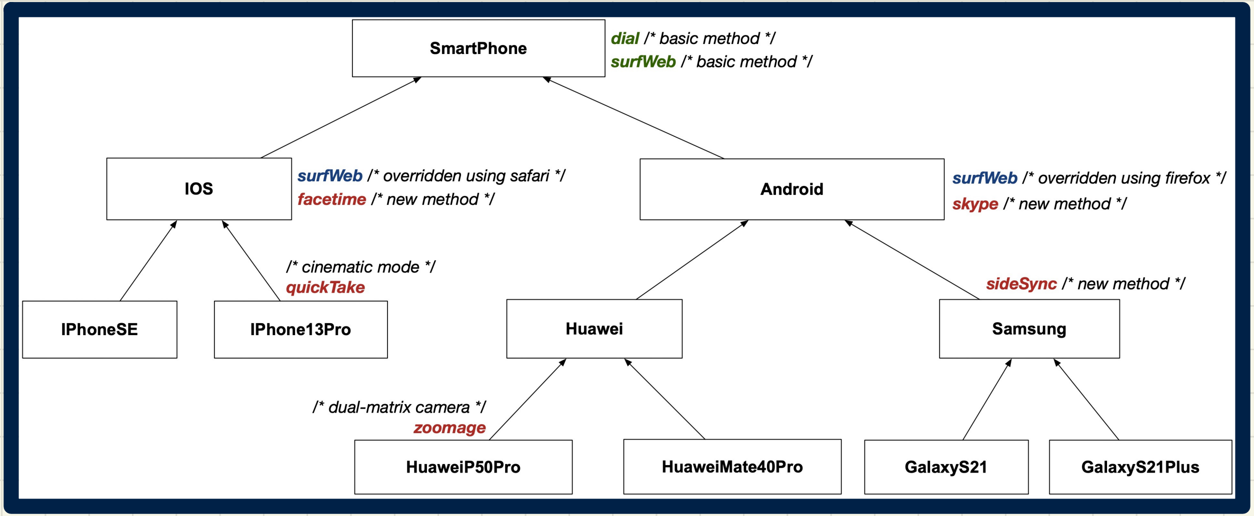
usually this class will be used subsequently to do a cast

(class) obj

↳ True if : ① DT of obj is a descendant of class

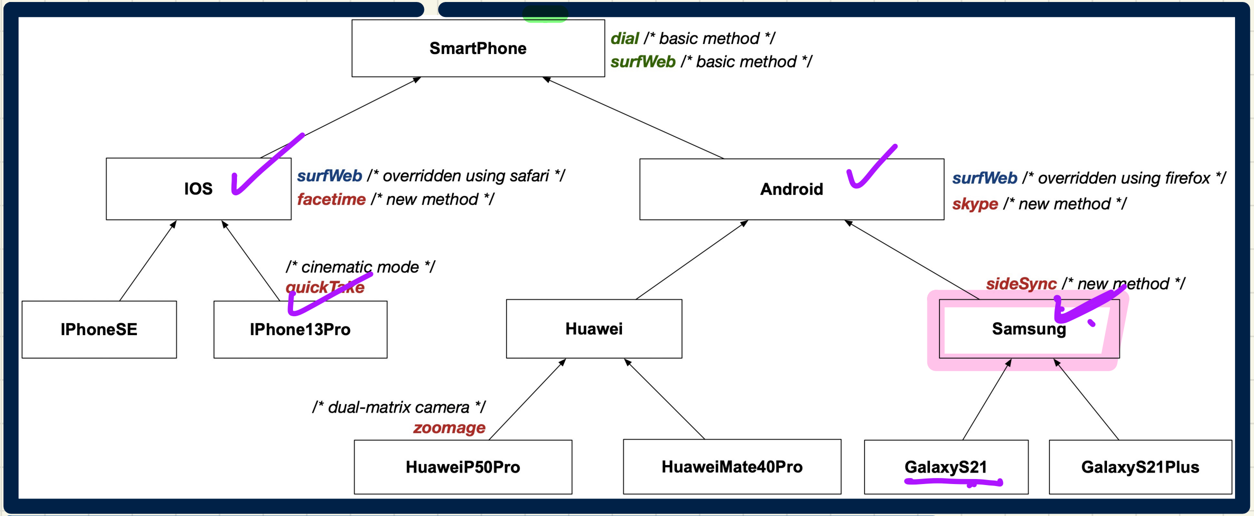
② DT of obj can fulfill expecation of class

Checking Dynamic Types at Runtime (2) (exercise)




```
SmartPhone aPhone = new GalaxyS21Plus();
if (aPhone instanceof IPhone13Pro ) {
  IOS forHeeyeon = ( IPhone13Pro ) aPhone;
  forHeeyeon.facetime();
}
```

Use of the instanceof Operator




```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android);
[redacted]
println(myPhone instanceof Samsung);
[redacted]
println(myPhone instanceof GalaxyS21);
[redacted]
println(myPhone instanceof IOS);
[redacted]
println(myPhone instanceof IPhone13Pro);
[redacted]
```

T

T

F

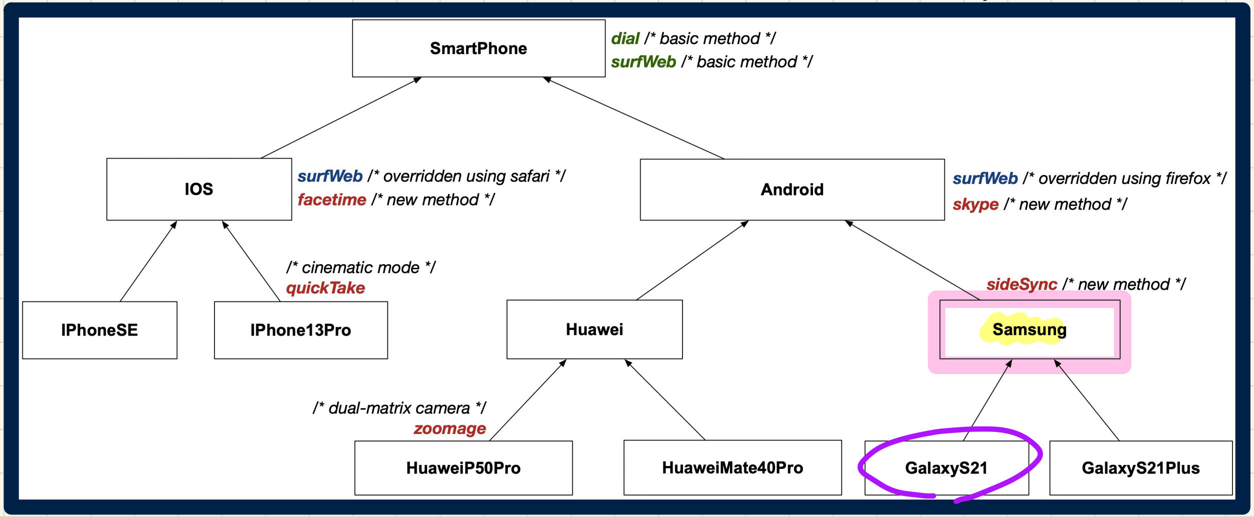
F

F

∵ DT of myPhone (Sam.) not descendant of GS21

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

Safe Cast via Use of the instanceof Operator




```java
SmartPhone myPhone = new Samsung();
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
if(myPhone instanceof Samsung) {
  Samsung samsung = (Samsung) myPhone;
}
if(myPhone instanceof GalaxyS21Plus) {
  GalaxyS21Plus galaxy = (GalaxyS21Plus) myPhone;
}
if(myphone instanceof HuaweiMate40Pro) {
  Huawei hw = (HuaweiMate40Pro) myPhone;
}
```

cast done without CCE!

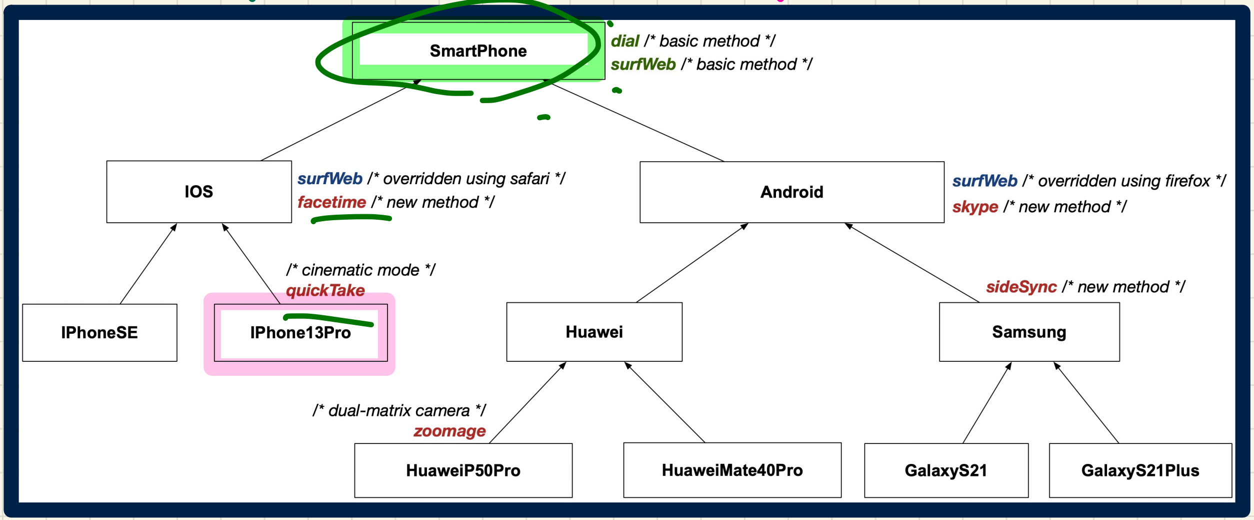
F

F

cast not reached ∵ instanceof evaluates to F

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

Static Types, Casts, Polymorphism (1)

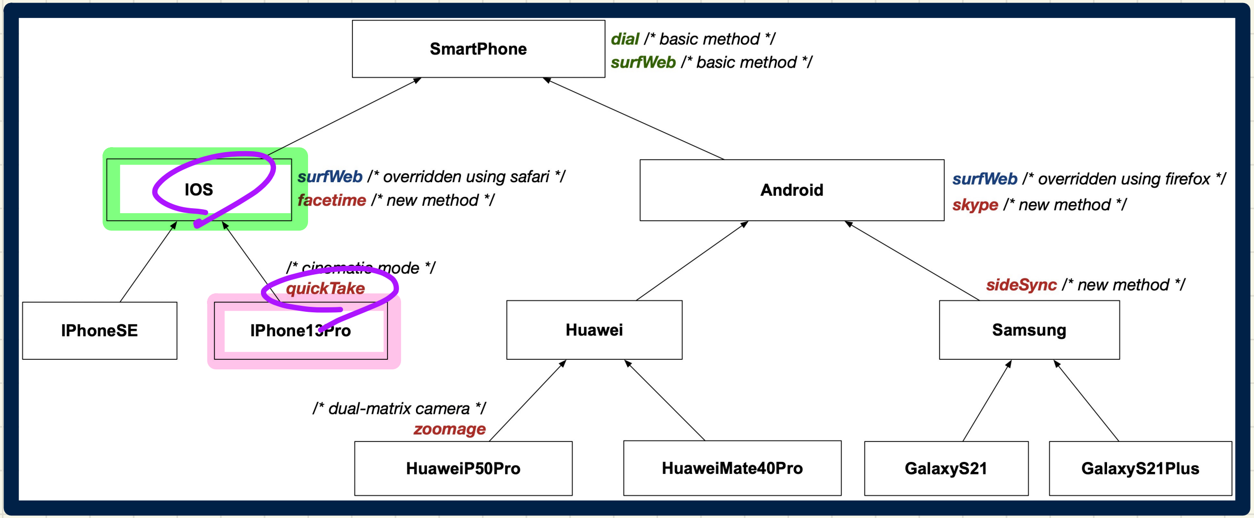



```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
SmartPhone sp = new IPhone13Pro();
sp.dial();        ✓
sp.facetime();    ×
sp.quickTake();   ×
```

ST: SmartPhone

Static Types, Casts, Polymorphism (2)

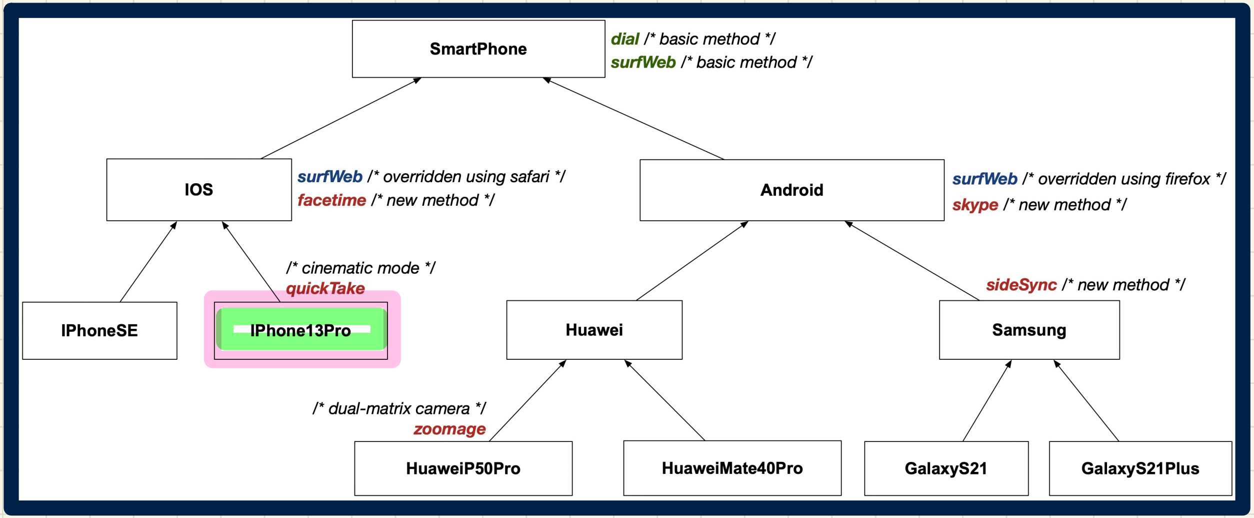



```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
IOS ip = new IPhone13Pro();
ip.dial();          ✓
ip.facetime();      ✓
ip.quickTake();     ×
```

→ ST IOS

Static Types, Casts, Polymorphism (3)

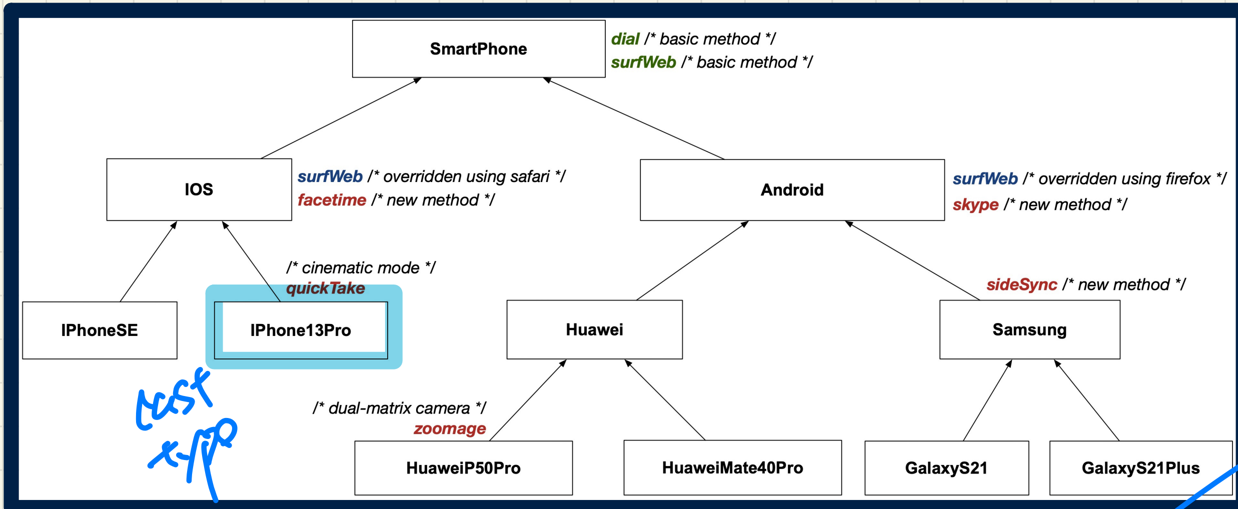



```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
IPhone13Pro ip6sp = new IPhone13Pro();
ip6sp.dial();        ✓
ip6sp.facetime();    ✓
ip6sp.quickTake();   ✓
```

allowed by ST.

Static Types, Casts, Polymorphism (4)




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

```
SmartPhone sp = new IPhone13Pro();   [redacted]
(IPhone13Pro) sp).dial();   [redacted]
(IPhone13Pro) sp).facetime();   [redacted]
(IPhone13Pro) sp).quickTake();   [redacted]
```

cast type

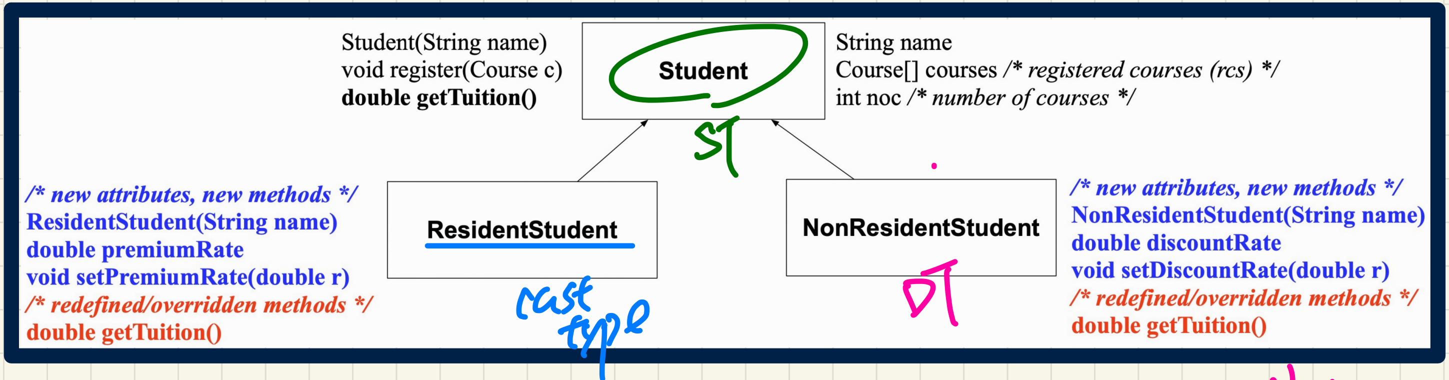
↑ a descendant

↳ no CCE.

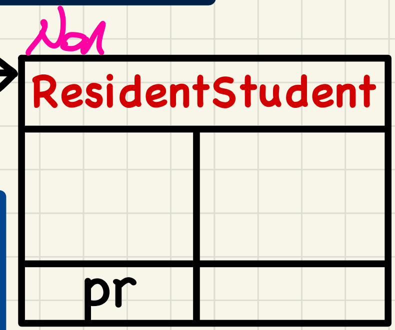
DT

creating an alias of ST IPhone13Pro.

Static Types, Casts, Polymorphism (5)



Student s



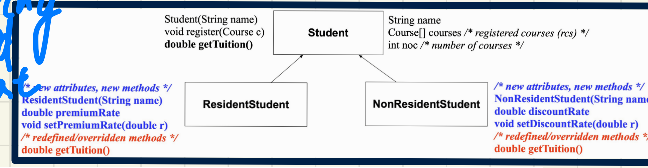

```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if (s instanceof ResidentStudent) {
  ((ResidentStudent) s).setPremiumRate(1.75);
  System.out.println(((ResidentStudent) s).getTuition());
}
```

NonResidentStudent → False

x

Polymorphic Parameters (1)

① it has an associated type. hierarchy
② each element in array has declared type Student




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

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

Student

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

ST: Student ST: RS

valid ∵ ST of RHS descendant of ST of LHS.

call by value: rs = o ; para. arg.

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

sms.addRS(o)

valid: ST of arg. o should be a descendant of para. rs

what should be the type of o?

Lecture 21 - Nov 23

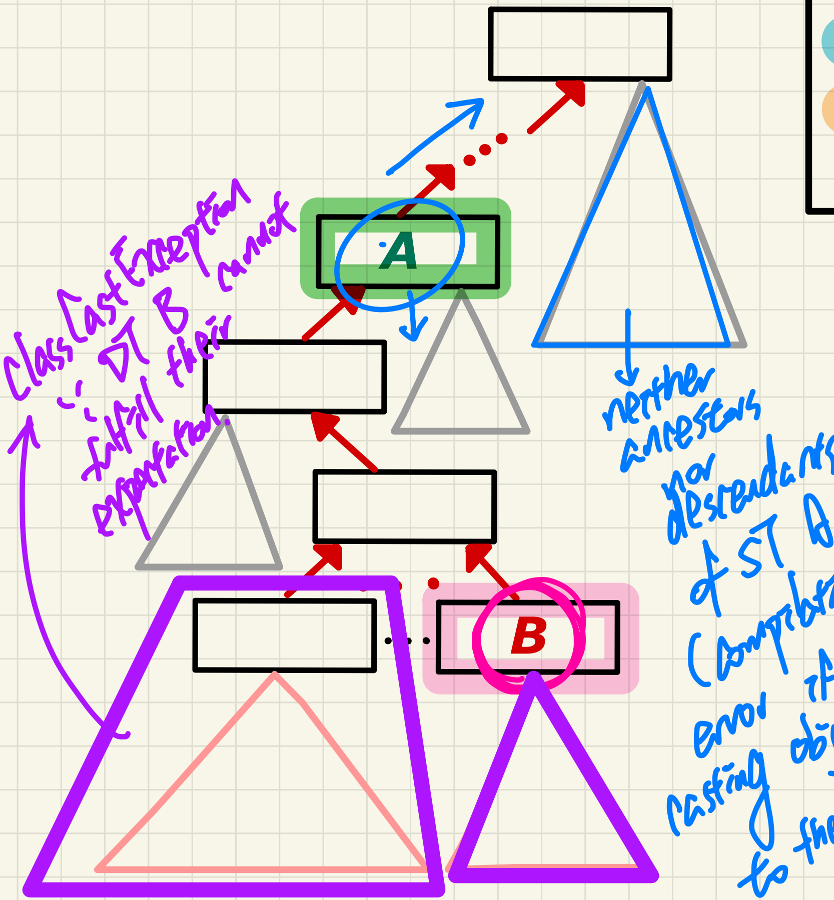
Inheritance

Polymorphic Method Parameters
Polymorphic Method Return Types

Announcements

- Lab4 due soon!
- ProgTest2 results to be released on Thursday
- WT3 and ProgTest3 approaching...

The instanceof Operator




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

True if obj's DT can fulfill the exp. of ??

cast type.

cast

- 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 ??.

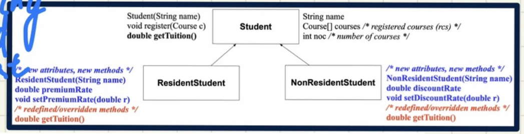
runtime
↳ DT!

- L2:
  - Evaluates to true if B can fulfill expectations on ??.

From last lecture...

Polymorphic Parameters (1)

① it has an associated type hierarchy
② each element in array has declared type Student




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

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

Student

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

valid ∵ ST of RHS descendant of ST of LHS.

ST: Student   ST: RS

call by value: rs = o; para. arg.

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

sms.addRS(o)

what should be the type of o?

valid: ST of arg. o should be a descendant of para. rs

Polymorphic Parameters (2)

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

call by value:

rs = s1

ST: RS

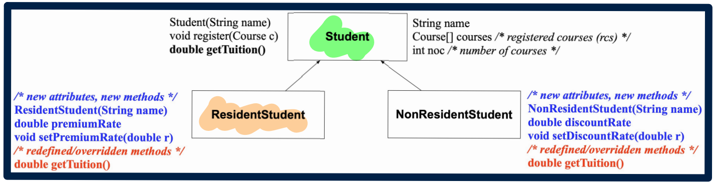
ST: S

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

addRS(RS rs)

addS(S s)

the higher the ST of parameter is, the more types of arguments it can accept



Casting Arguments

Student s;

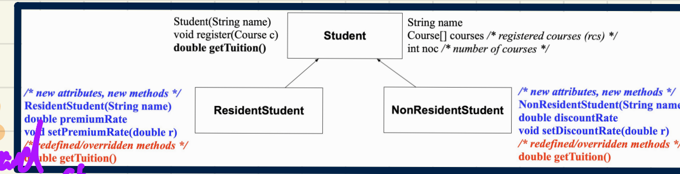
```
void addRS(ResidentStudent rs)
```

RS

sms.addRS(s); ✗

↗ cast exp. has ST : RS ✓ ∵ downward cast

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




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

① sms.addRS((RS) s);

DTs cannot fulfill exp. of cast type RS

ClassCastException?

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

② sms.addRS((RS) s);

ClassCastException?

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

③ sms.addRS((RS) s);

ClassCastException?

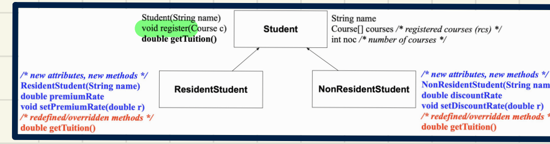
→ - compiles
- no CCE.

---

sms.addRS((ResidentStudent) nrs) compiles? → not compile ∵ cast type RS neither ancestor nor descendant of ST NRS

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

A Polymorphic Collection of Students




```
ResidentStudent rs = new ResidentStudent("Rachael");
rs.setPremiumRate(1.5);
NonResidentStudent nrs = new NonResidentStudent("Nancy");
nrs.setDiscountRate(0.5);
StudentManagementSystem sms = new StudentManagementSystem();
sms.addStudent(rs); /* polymorphism */
sms.addStudent(nrs); /* polymorphism */
Course eecs2030 = new Course("EECS2030", 500.0);
sms.registerAll(eecs2030);
for(int i = 0; i < sms.numberOfStudents; i ++) {
  /* Dynamic Binding:
   * Right version of getTuition will be called */
  System.out.println(sms.students[i].getTuition());
}
```

parameter type: Student accepting arguments of its descendant classes

dynamic binding

0, 1

```
class StudentManagementSystem {
  Student[] students;
  int numOfStudents;

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

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

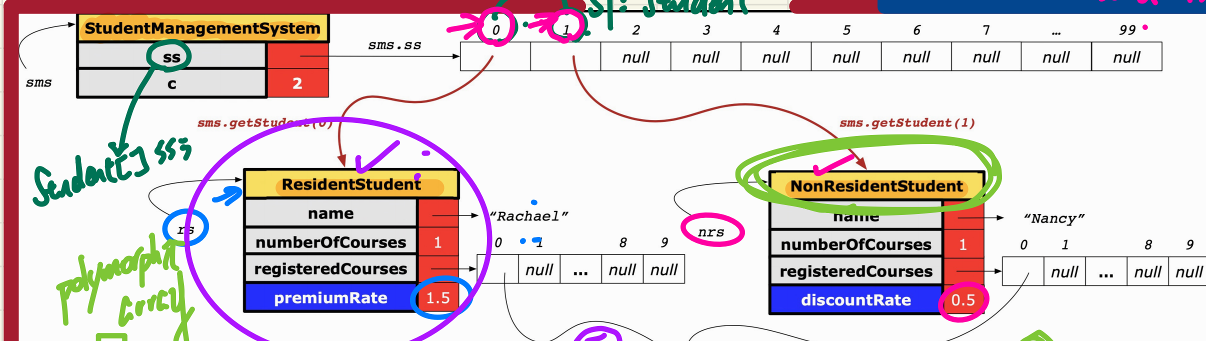
what if: students[i].getTuition(); ?

dynamic binding: which version of register method is invoked?

0, 1

students[0].register(c);
students[1].register(c);

ST: Student

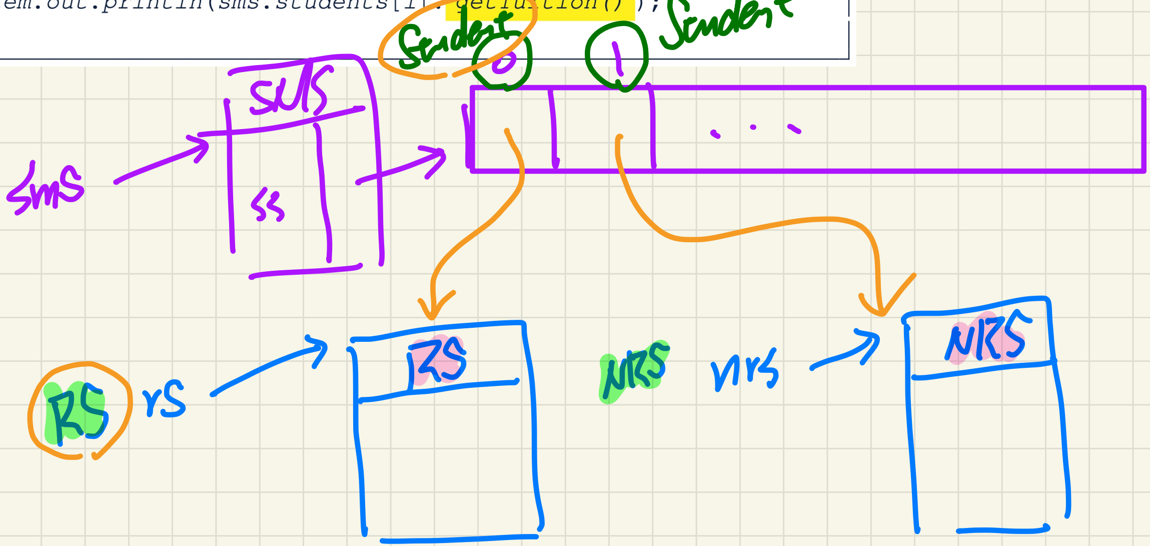


sms.ss[0] instanceof NonResidentStudent F
sms.ss[0] instanceof ResidentStudent T
sms.ss[0] instanceof Student T

sms.ss[1] instanceof NonResidentStudent T
sms.ss[1] instanceof ResidentStudent F
sms.ss[1] instanceof Student T

Polymorphic array:
- same ST for each elem.
- diff. DTs for elements

```java
ResidentStudent rs = new ResidentStudent("Rachael");
rs.setPremiumRate(1.5);
NonResidentStudent nrs = new NonResidentStudent("Nancy");
nrs.setDiscountRate(0.5);
StudentManagementSystem sms = new StudentManagementSystem();
sms.addStudent(rs); /* polymorphism */
sms.addStudent(nrs); /* polymorphism */
Course eecs2030 = new Course("EECS2030", 500.0);
sms.registerAll(eecs2030);
for(int i = 0; i < sms.numberOfStudents; i ++) {
  /* Dynamic Binding:
   * Right version of getTuition will be called */
  System.out.println(sms.students[i].getTuition());
}
```



Polymorphic Return Types

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

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

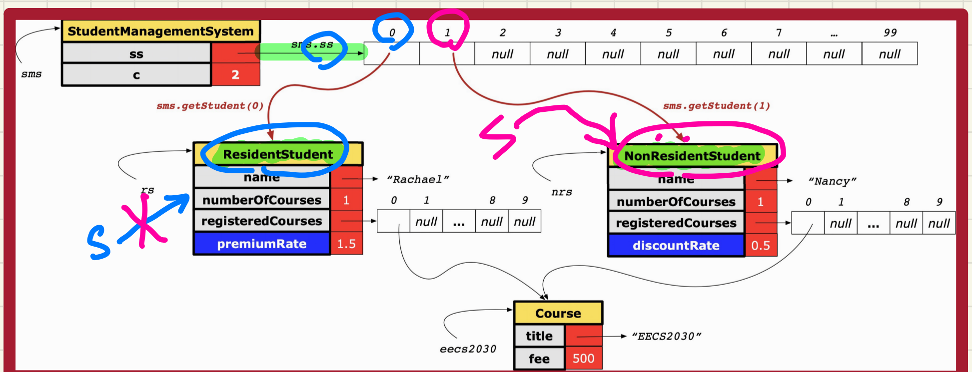
ss is a polymorphic array

① ss[i] has ST: Student

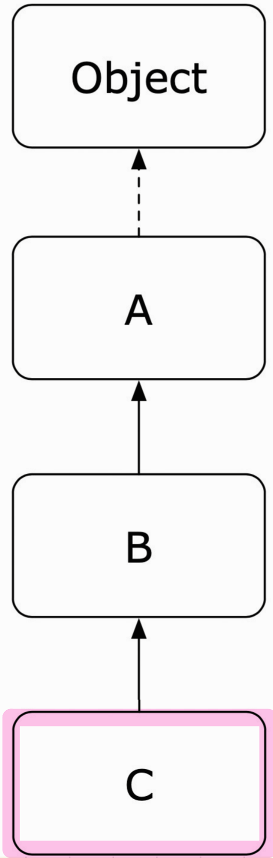
② ss[i] has DT a descendant of ST

ss[0]

ss[1]



Overridden Methods and Dynamic Binding (1)



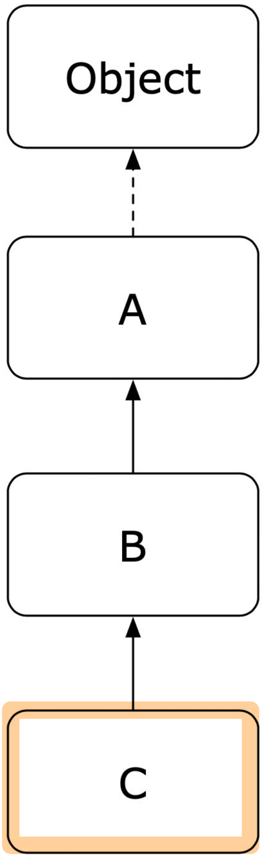

```
boolean equals (Object obj) {
  return this == obj;
}
```

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

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

L3 calls which version of equals? [Object]

Overridden Methods and Dynamic Binding (2)




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

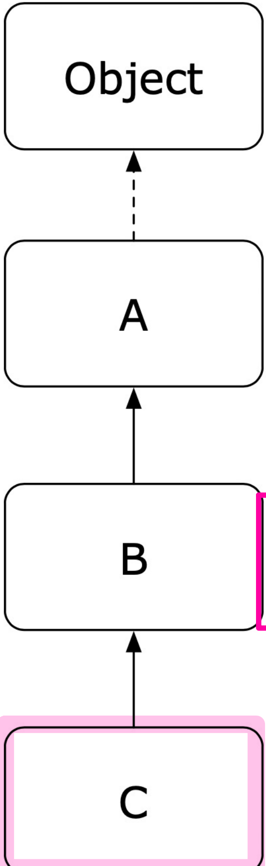
```
boolean equals (Object obj) {
  /* overridden version */
}
```

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

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

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

Overridden Methods and Dynamic Binding (3)




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

boolean equals(Object obj) {...}

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

overridden version of closest ancestor.

```
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]

Lecture 22 - Nov 28

Inheritance, Recursion

Type-Checking Rules
Solving Problems Recursively: Fac vs. Fib
Recursions on Strings: Palindrome

Announcements

- **Lab5** to be released on Wednesday

Static Types and Anticipated Expectations

```
class A {
  void m1() { ... }
}
class B extends A { }


class C extends A {}
```

①
```
B obj1 = new A();

A = obj2 = new A();
```
②
```
B obj3 = (B) obj2;
```

② A cannot fulfil the exp. of B

① not compile ∵ DT A not a descendant of the ST of obj1 (B)

CCE ∵ DT A can't fulfill exp. of cast type

ST: A

↳ ∵ downward cast

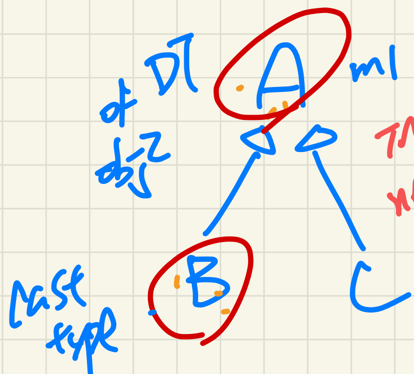
2. CCE?

at the moment, no new methods have been introduced

⇒ B & C & A have identical exp.

are anticipated.

in the future, new methods



Summary: Type Checking Rules

CODE	CONDITION TO BE TYPE CORRECT
`x = y`	Is `y`'s ***ST*** a **descendant** of `x`'s ***ST***?
`x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***?
`z = x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***? Is ***ST*** of `m`'s return value a **descendant** of `z`'s ***ST***?
`(C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***?
`x = (C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is `C` a **descendant** of `x`'s ***ST***?
`x.m((C) y)`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is method `m` defined in `x`'s ***ST***? Is `C` a **descendant** of `m`'s parameter's ***ST***?

param = arg.

call by value

3

1

2

Solving a Problem Recursively

base

Given a **small** problem:		Solve it **directly**:	

recursive case

Given a **big** problem:	i $n!$		
Divide it into **smaller** problems:	j $(n-1)!$	k	l
Assume solutions to **smaller** problems:	$(n-1)!$		
Combine solutions to **smaller** problems:	$n \cdot (n-1)!$		

$j < i \Rightarrow$ solving a strictly smaller problem

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

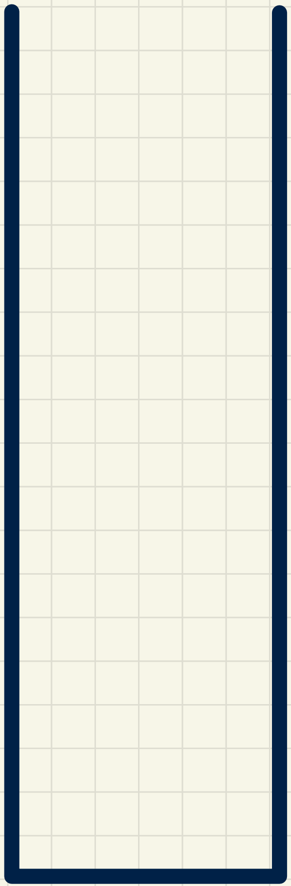
calling itself with some arg.

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 $\boxed{\text{current point of execution}}$.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

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



Runtime Stack

Recursive Solution: factorial

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

base case

problem

is this recursive?

↳ No!

∵ the problem (!) is not reduced into smaller problem(s) in the def

Recursive Solution

① Base Cases : $0! = 1$

② Recursive Cases : $n! = \boxed{(n-1)!} \cdot n$

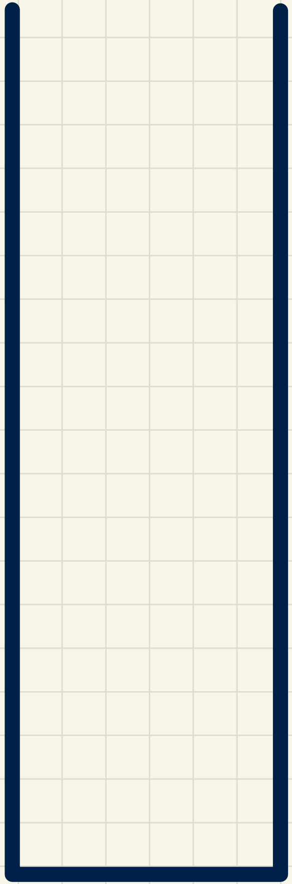
→ solution to a strictly smaller problem

Recursive Solution in Java: factorial

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

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

Example: factorial(3)



Runtime Stack

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;
}
```

3 2 1 0

3 * fac(2) 2· 6

2 * fac(1) 1 2

1 * fac(0) 1

= fac(1)

Example: factorial(3)

In order for the call stack not to grow indefinitely, we need to make sure the base case is reached ultimately.

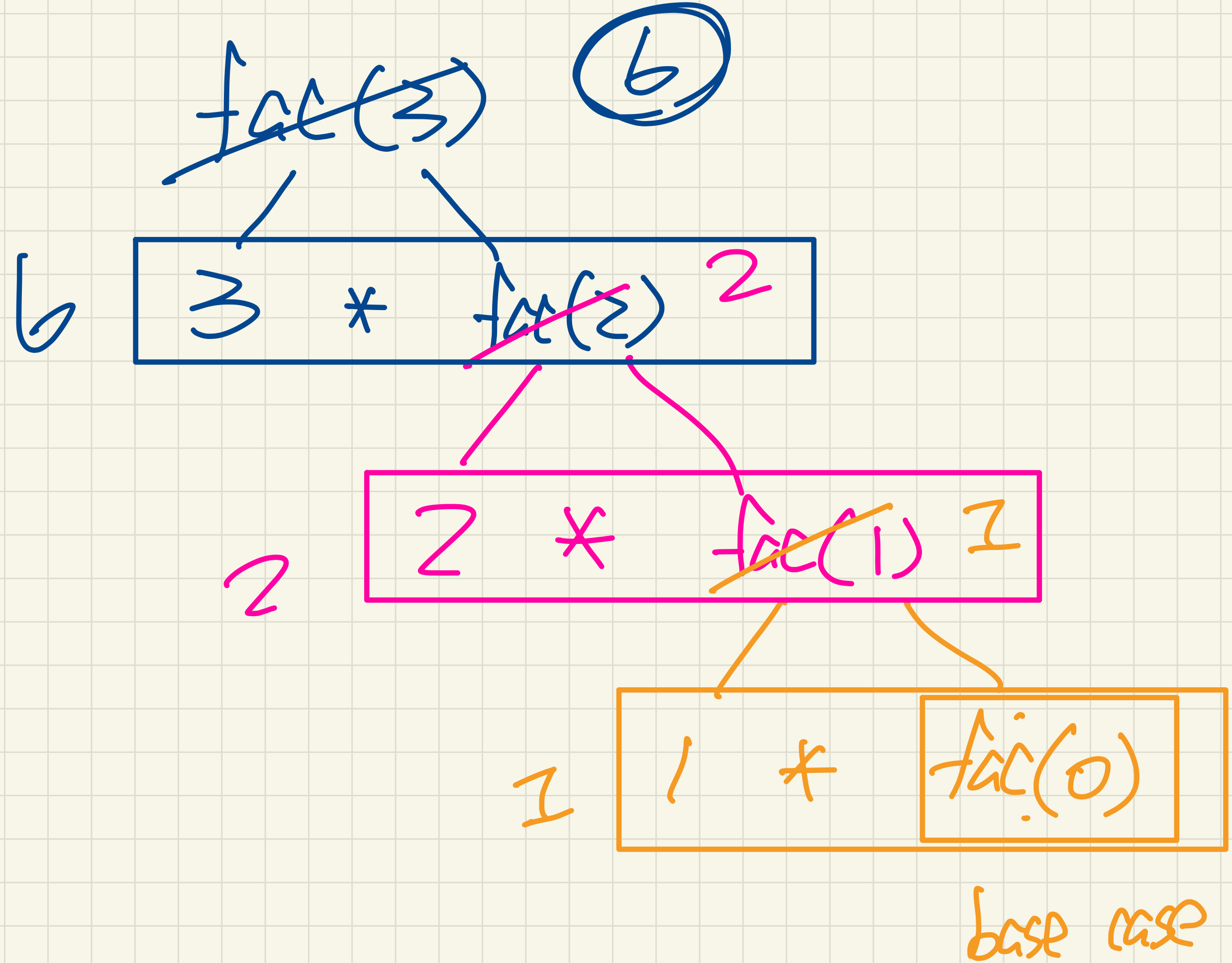
fac(0)

fac(1) 1

fac(2) 2

fac(3) 6

Runtime Stack



fac(3)
6
6
3 * fac(2)
2
2
2 * fac(1)
1
1
1 * fac(0)
base case

Common Errors of Recursion (1)

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

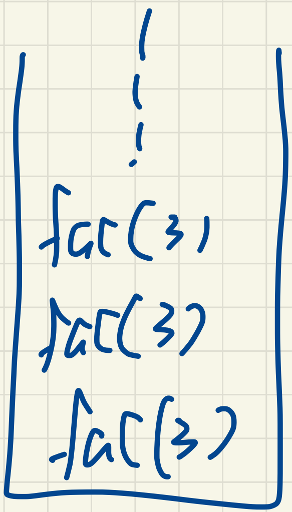
fac(-1)
fac(0)
fac(1)
fac(2)
fac(3)

StackOverflowException

→ always put at least one base case

Common Errors of Recursion (2)

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



StackOverflow Error.

→ when making a recursive call, make sure to call the method on a strictly smaller input.

Recursive Solution: Fibonacci Numbers

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

$F_1 \quad F_2 \quad F_3 \quad F_4 \quad \ldots \quad F_{n-2} \quad F_{n-1} \quad F_n$

F_n

$n \geq 1, 2$

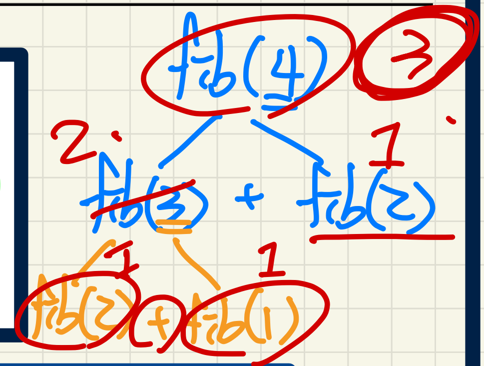
$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Recursive Solution in Java: Fibonacci Numbers

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




```java
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;
}
```

solution to a smaller problem

solution to another strictly smaller problem

Example: fib(4)

Runtime Stack

Use of String

substring (i, j) [2, 5) = 2, 3, 4

[i, j)

0 1 2 3

s → "a b c d"

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

empty string

[0, 0)

[0, 4) = [0, 3]

entire string

Recursions on Strings

Palindrome

“racecar”

“aracecars”

“raceacar”

compare the 1st and last characters

C1: Same

↳ ______

C2: Diff

↳ not palindrome

palin(“aracecars”)

= ‘a’ == ‘s’ &&

palin(“racecar”)

strictly smaller problem.

Reversal

“abcd”

Number of Occurrences

“abca”

‘a’

‘b’

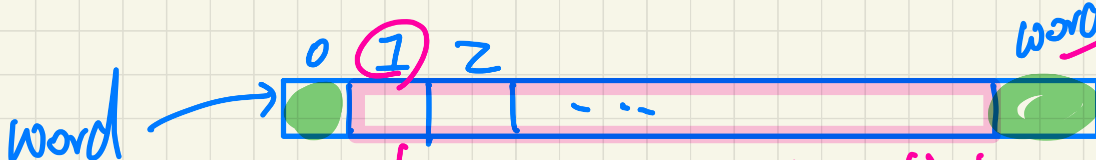
Problem: Palindrome

```java
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);
  }
}
```

Empty string or string of length 1 ⇒ calculate right away

recursive call on a strictly smaller problem.

word.length() - 1



word.substring(1, word.length() - 1)

Lecture 23 - Nov 30

Recursion

Tracing Recursions: Faibonacci
Recursions on Strings: Reverse
Recursions on Arrays

Announcements

- Lab5 to be released by the end of today
- ProgTest3 next Tuesday (based on Lab4)

Recursive Solution: Fibonacci Numbers

F_7 F_8 F_9

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

Base Cases

$F_1 = 1$

$F_2 = 1$

Recursive Cases

$\boxed{F_n} = F_{\boxed{n-1}} + F_{n-2}$

$n > 2$

strictly smaller than n

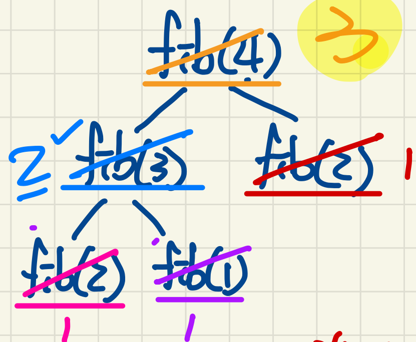
$< n$

solved recursively by two recursive calls

$F_9 = F_7 + F_8$.

Recursive Solution in Java: Fibonacci Numbers

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



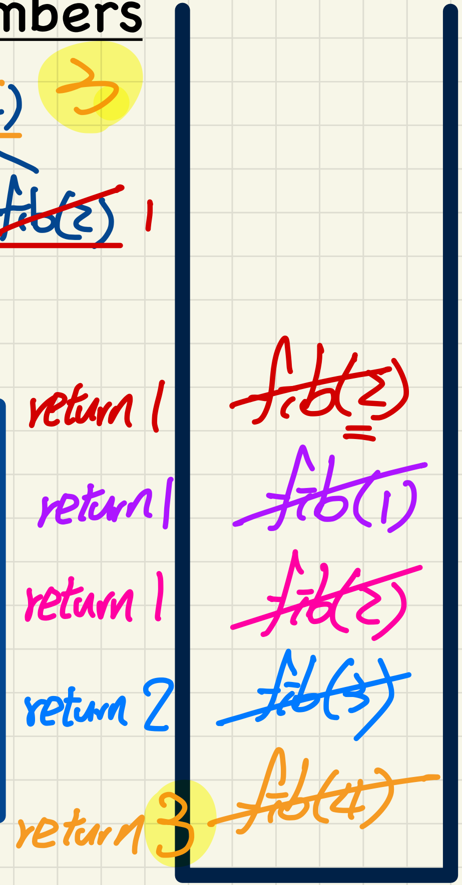

```
int fib (int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib (n - 1) + fib (n - 2);
  }
  return result;
}
```

2 ~~fib(3)~~ + ~~fib(2)~~ 1 = 3

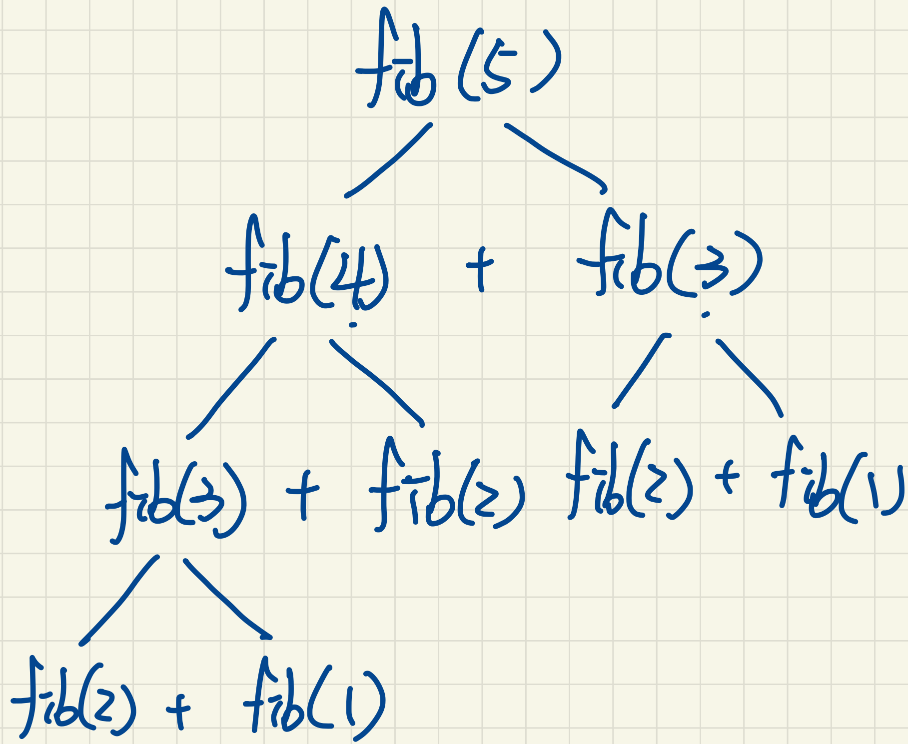
~~fib(2)~~ + ~~fib(1)~~ = 2

1 1

Example: fib(4)



Runtime Stack

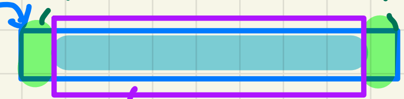

fib(5)
fib(4) + fib(3)
fib(3) + fib(2)
fib(2) + fib(1)
fib(2) + fib(1)

Recursions on Strings

Palindrome

input string

== ?



strictly smaller problem

"racecar" → T

"aracecars" → F

"raceacar" → F

Reversal

"abcd"

strictly smaller problem

reverseOf

dcba

solution to strictly smaller prob.

Number of Occurrences

"abca"

'a' $2 = 1 + 1$

'b' $1 = 0 + 1$

# of occurrences of the char in tail of input string.

↳ is the char equal to the head of the string

Problem: Reverse of a String

```java
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;
  }
}
```

base cases

↳ recursive call to solve a strictly smaller problem.

dcba

reverseOf(abcd)

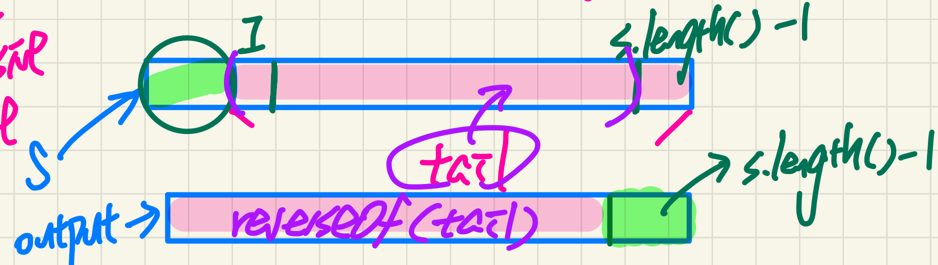
dcb ↳ reverseOf(bcd) + a

dc ↳ reverseOf(cd) + b

↳ reverseOf(d) + c

d

recursive case



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);
    }
  }
}
```

base case

recursive case

what if s is "a"?

↳ ""

s → 0

head

tail

occurrencesOf(tail, c)

c == head → 1

c != head → 0

occ(abca, a) = 2

a==a → 1 + occ(bca, a)

a!=b → 0 + occ(ca, a)

a!=c → 0 + occ(a, a)

a==a → 1 + occ("", a) → 0

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[0] = a[i - 1]; }
    m(sub) } }
```

base cases

recursive case

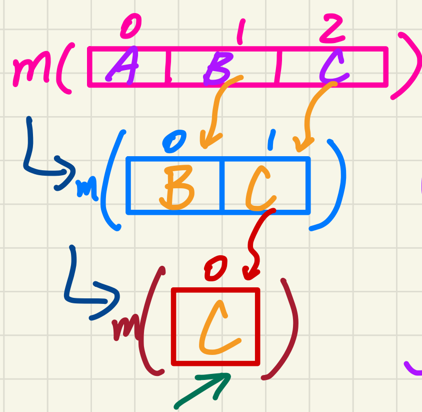
i-1

i

sub[0] = a[1]

Say a1 = {}, consider m(a1) → execute the base case

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



not space-efficient (for each r.c., a new array is created)

Recursion on an Array: Passing Same Array Reference

array of length 1.

```
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) } }
```

base cases

recursive case

empty array

[0, -1] → empty range.

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

→ m(a1, 0, -1)

from to

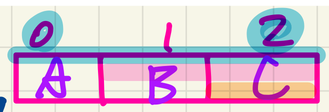
↳ min index

max index

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

3

a2



m(a2, 0, 2)

↳ m(a2, 1, 2)

strictly smaller problem (elements from indices 1 to 2)

↳ m(2, 2)

strictly smaller problem (last item in array).

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);
  }
}
```

max index

max index

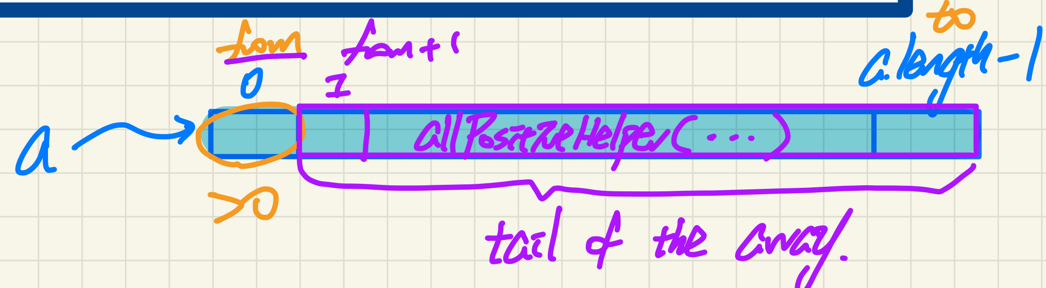
recursive helper method

empty array

array of length 1

base cases

recursive case



Tracing Recursion: allPositive

Say a = {}

allPositive(a)

allPH(a,0,-1)

a.length - 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)

{4}

allPH(a,0,0)

a.length - 1

a[0] > 0

True

0

4

from

to

a

```
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);
  }
}
```

True

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);
  }
}
```

allPositive(a)

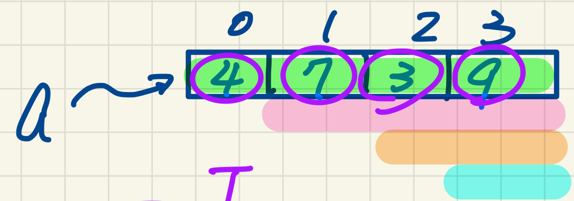
allPH(a,0,3) T

a[0] > 0 T && allPH(a,1,3) T

a[1] > 0 T && allPH(a,2,3) T

a[2] > 0 T && allPH(a,3,3)

a[3] > 0 T



Tracing Recursion: allPositive

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

- allPositive(a)
  - allPH(a,0,3)
    - a[0] > 0
    - allPH(a,1,3)
      - a[1] > 0
      - allPH(a,2,3)
        - a[2] > 0
        - allPH(a,3,3)

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

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

Exercise: Trace!

Problem: Are Numbers Sorted?

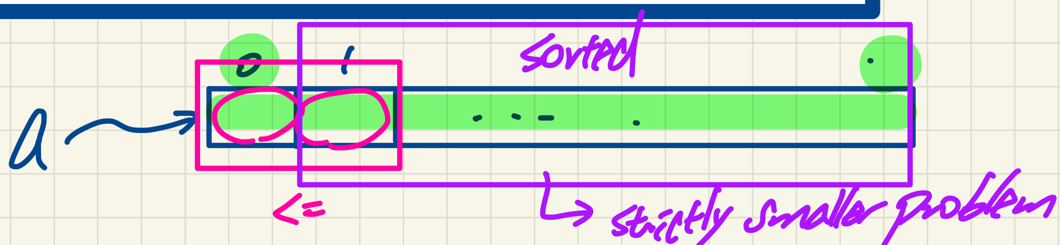
```java
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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);
  }
}
```

recursive helper method.

base cases

recursive case



Tracing Recursion: isSorted

Say a = {}

isSorted(a) {}

isSH(a,0,-1)

```java
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***

{4}

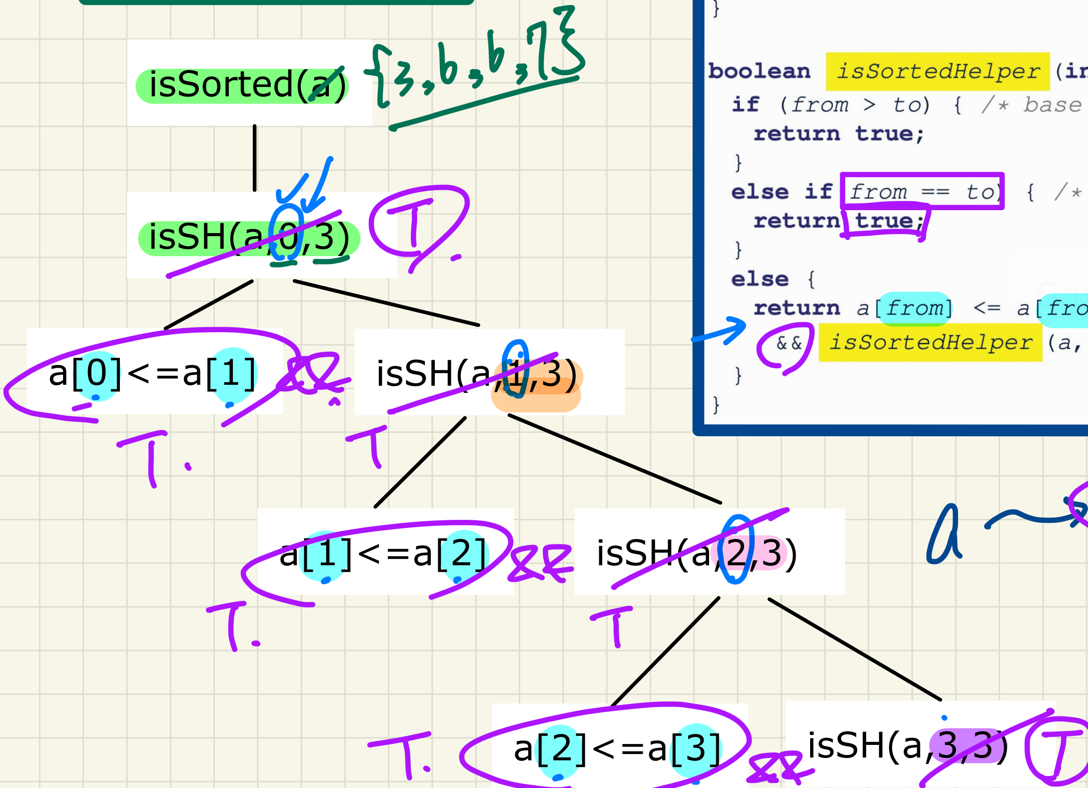
0

```java
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);
  }
}
```

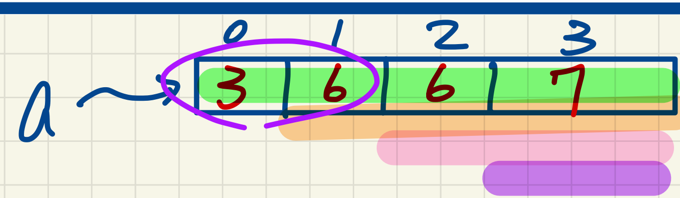
Tracing Recursion: isSorted

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



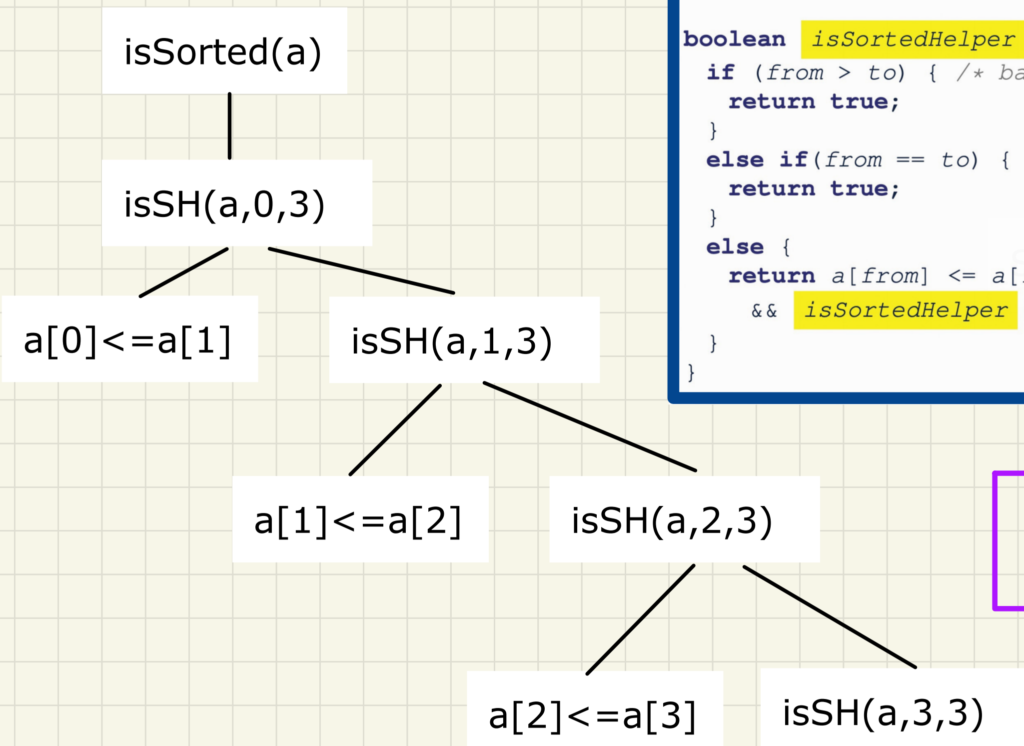

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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} → F




```
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);
  }
}
```

Exercise: Trace

Lecture 24 - Dec 5

Wrap-Up

Topics Covered
Exam Review Session

resubmission

3pm?

Announcements

- Lab5 already released
- WrittenTest3 to be released by the end of today
- ProgTest3 tomorrow (based on Lab4)
- Makeup Test on Wednesday (based on Lab2 & Lab3)
- Exam Review (Q&A) on Thursday, December 8?

✓ 1:00 – 2:30

✓ 4:00 – 5:30

Exam Info

- When: $\boxed{\text{7pm to 10pm}}$, Monday, December 12
- Where: TC Sobeys
- Coverage: Everything (lecture materials & labs) ① lectures ② Labs (③ CodingBat)
- $\boxed{\text{Format:}}$ Multiple Choice & $\boxed{\text{Written}}$
  - ↳ 1. scantron 2. pencil (B?)
  - ↳ 1. program recursively 2. justification code, why or why not there's a CCE - 3. output. ↳ instanceof
- Restrictions:
  + No data sheet
  + No sketch paper (Exam booklet includes it)
- What you should bring:
  + $\boxed{\text{Valid Photo ID (strict)}}$
  + Water/Snack

- some practice questions (by Friday) - PDF guide.

That's all!

I hope you enjoyed the learning journey with me.

Best of luck with your future endeavours!

Jackie

Dec. 7, 2022