# EECS2030 (Sec. G) Fall 2025
# Advanced Object-Oriented Programming

## Lecture Notes

Instructor: Jackie Wang

# Lecture 1 - Sep 4

## Syllabus & Introduction

***Professional Engineers: Code of Ethics***
***Software Development Process***

# Course Learning Outcomes (CLOs)

heados

| | |
|---|---|
| CLO1 | Implement an Application Programming Interface (API) |
| CLO2 | Test the implementation. JUnit. (Eclipse) |
| CLO3 | Document the implementation. |
| CLO4 | Implement aggregations and compositions. ref. aliasing. |
| 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. |

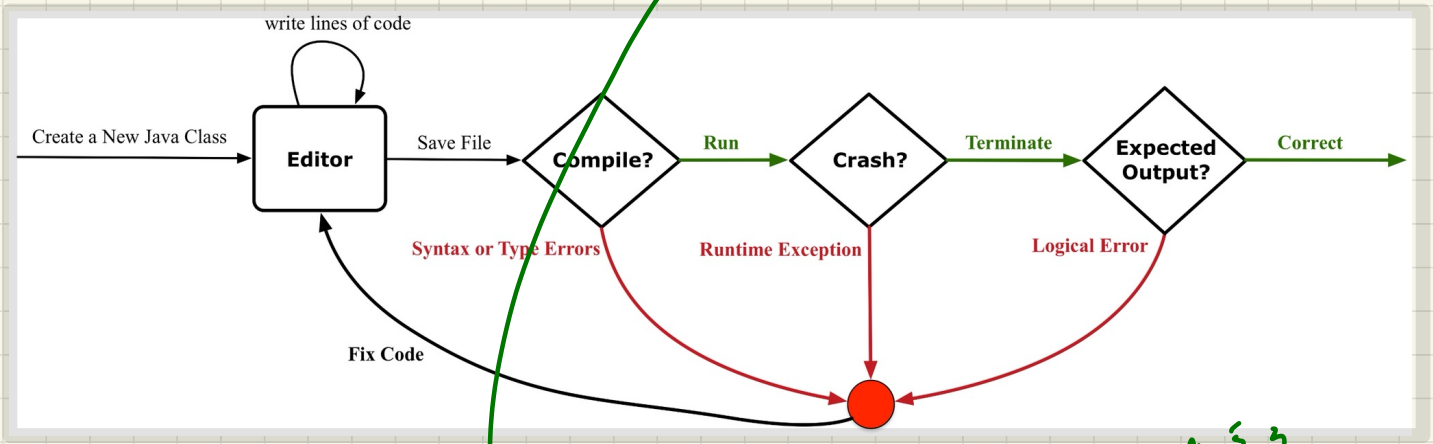
1. programming.
2. polymorphism
3. dynamic binding

- primitive arrays
- TOH.

2101

2101

# Software Development: Compile Time vs. Run Time




editing Java code
↳ save
↳ syntax error ( missing ";" "class" )

type errors
↳ manipulating values inconsistently
e.g. ① int i = "20 30";
② int i = 23;
int j = 7;
unchecked

# Lecture 2 - Sep 8

## OOP Review

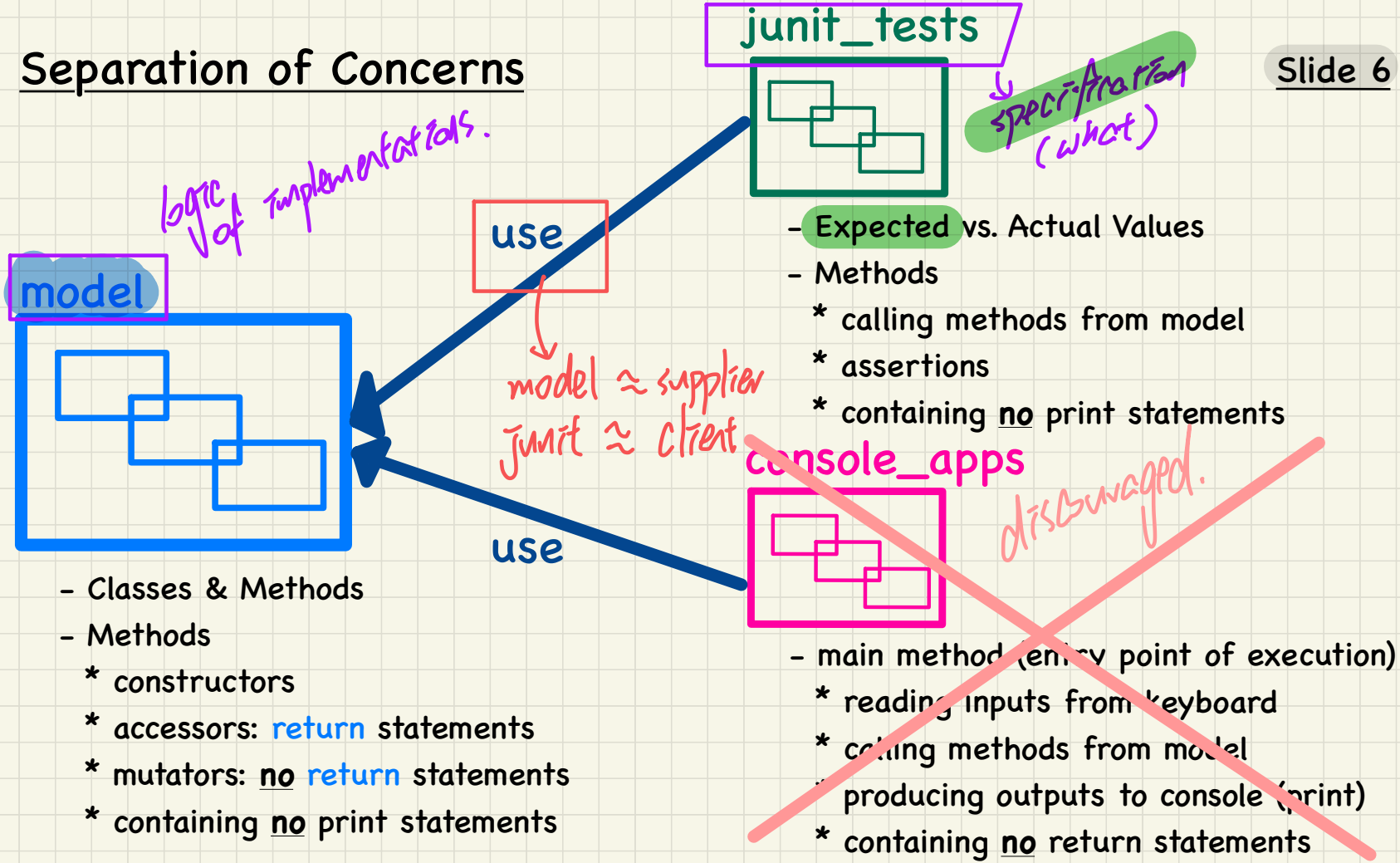
***Separation of Concerns: Model vs. Tests***
***OO: Observe-Model-Execute Process***
***Java Data Types: Primitive***

## Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Priorities:
  - + Lab0P1 (tutorial videos + PDFs)
  - + Due: This Tuesday (Sep 8)
- Lab0P2 to be released this Tuesday (Sep 8)
- Lab this Wednesday (Sep 10): Demo on helper methods?
- Lab next Wednesday (Sep 17):
  - + [≈ 40 min] Mock-up Programming Test 0
  - + [≈ 40 min] Q&As (TAs, Jackie)

# Separation of Concerns






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

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

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

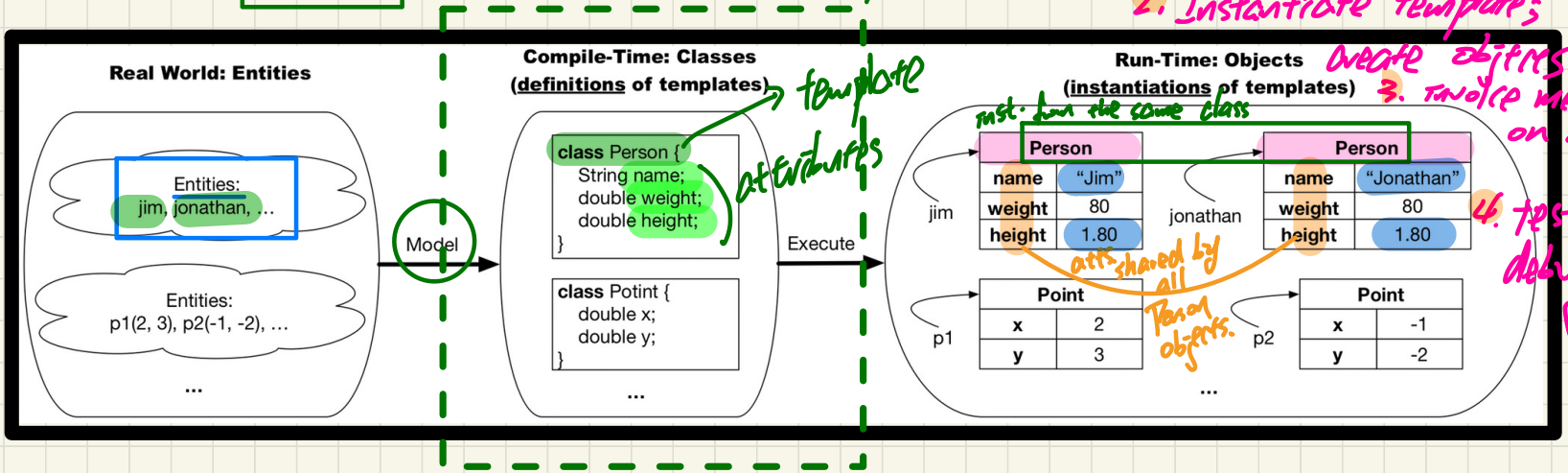
# Observe-Model-Execute* Process

Java

Slide 7

Compile time.

1. Assume: Everything compiles
2. Instantiate templates
3. Invoke meth. on objs.
4. test & debug.




template

attributes

inst. from the same class

attr. shared by all Person objects.

Jim

Jonathan

Entities → Persons.

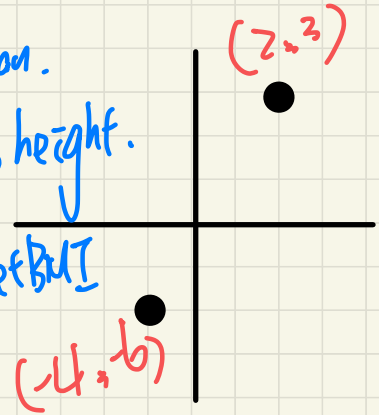
Entities: Jim, Jonathan.

Attributes: weight, height.

Changes: growTaller

Inquiries: getW, getBMI

Template: Person




Entities:

Attributes:

Changes:

Inquiries:

Template:

OO

templates

instances

changes

invariants

OOP (Java, C#)

classes

objects

void setH(...)

mutators/setters

accessors/getters not void.

String getN()

double getW()

Jonathan, Jim

Person

- Entities of the same $\boxed{\text{kind}}$ share the same set of attributes.

  ↳ weight, height

- Values of attributes among entities (of the same kind) may be different.

Jim.weight vs. Jonathan.weight

```
class Person {
    double weight;
    double height;
    Person (double w) { ... }
    Person (double h) { ... }
}
```

<double>

<double>

compilation error ∵ the list of parameter types is not distinct

```
Person jim = new Person
        (72.6);
```

ambiguous: which version to call?

```
class Person {

    double weight;
    double height;

    Person(double w) { ... }
    Person(double h) { ... }

}
```

<double, double>

double w, double h

① Person jim = new Person(173.2);

② Person jonathan = new Person(84.5, 185.7);

jim

| Person | |
|---|---|
| w. | 0 |
| h. | 173.2 |

jonathan

| Person | |
|---|---|
| w | 84.5 |
| h. | 185.7 |

# 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

private attrs methods

modify the internal workings without any public/external users realizing it.

(e.g. helper methods)

obj1.m1().m2().m3()

chain of method calls.

Slides 9 ~ 47

assigned readings!

```
Person jim = . . . ;
Person jon = . . . ;
```

* ref. aliasing

different objects

① jim.getWeight() → 72

② jon.getWeight() → 85

✓ ③ jim.gainWeight(2) → +2 to jim's weight

④ jon.gainWeight(2) → +2 to jon's weight

⑤ jim.getWeight() → 74

⑥ jon.getWeight()

* may ref the same object

① vs. ②

③ vs. ④

① vs. ⑤. (not same ∵ ③)

② vs ⑥ (not same ∵ ④)

- In Java, each variable <u>must</u> be declared with a type.

compile time

static-typed PL

dynamic-typed PL e.g. Python.

{..., -2, -1, 0, 1, 2, 3, ...}

int

e.g. int i;

declared type for i (p.t.)

at r.t., i can store any value that's a member of int.

# Lecture 3 - Sep 11

## OOP Review

***Java Data Types: Reference***
***Parameters vs. Arguments***
***Variable Scopes, Tracing Object Creations***

## Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Priorities:
  - + LabOP2 (tutorial videos + PDFs)
  - + Due: Next Tuesday (Sep 16)
- Lab1 to be released next Tuesday (Sep 16)
- Lab yesterday (Sep 10): Demo on helper methods
- Lab next Wednesday (Sep 17):
  - + [≈ 40 min] Mock-up Programming Test 0
  - + [≈ 40 min] Q&As (TAs, Jackie)

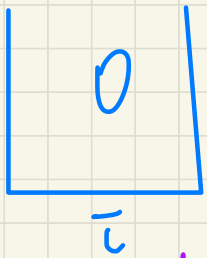
- In Java, each variable must be declared with a type.

compile time

static-typed PL

dynamic-typed PL e.g. Python.

{..., -2, -1, 0, 1, 2, 3, ...}




e.g. int i;

declared type for i (p.t.)

at r.t., i can store any value that's a member of int.

```
Person p1 = ...;
:
String s = "2030";
p1 = s;
```

× ∵ cannot store string address to a Person-specific var storing Person address

1. a class declared within your proj.

2. external lib. class imported (e.g. ArrayList)

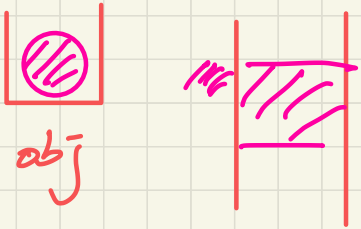
e.g. Person p;

at r.t. can store the address of some Person object

memory block reserved for storing Person-specific info.

# NullPointerException

→ runtime error




when context object of some method call is null.

undefined address

## method call

obj.m(...)

look up the address that's stored in the C.O. and go there.

context object (C.O.)

apply the def. of method to the targetted mem. block

If the mem. block does not exist (∵ address of C.O. is null) → NPE

Execution: → R

break point & debugger

(1) co may be null

(2) co.m1()

(3) co.m1().m2() may be null

co.m1().m2().m3() → Compiles

→ at r.t. NPE.

Q. How many possible causes for NPE? 3

# memory lookups using

Expression

Q. What are the exact causes?

(1)

(2)

(3)

A declared type of some variable $\underline{\underline{x}}$ denotes the set of values that is allowed to be stored in $\underline{\underline{x}}$.

arithmetic allowed

$\boxed{\text{numbers}}$ 1, 2, -1, -2, 0, 23.6, ...

address. 0x123 → means to uniquely identify a memory location (no arithmetic)

~~hex~~ base-8 base-16 number

Person@

# Parameters vs. Arguments

def. of methods

use/call methods

parameters

## Template Definition

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

  double getDistanceFrom(Point other) {...}

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

## Method Usages

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

arguments.

# Scope of Variables in a Class

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

  [ Assume for now: non-**static** ]
- Method-level parameters may be accessed within the declared method only.
- Method-level (local) variables may be accessed/modified within the declared method.

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

attributes (class-level variables)

fields

parameters

local variables (method-level)

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

p2 = n; ✓

call by value.

lvalue (change)

r value (use)

① p2 = n;

② n = p2;

# 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;
    }
}
```

all atts left with default values.

**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);
}
```

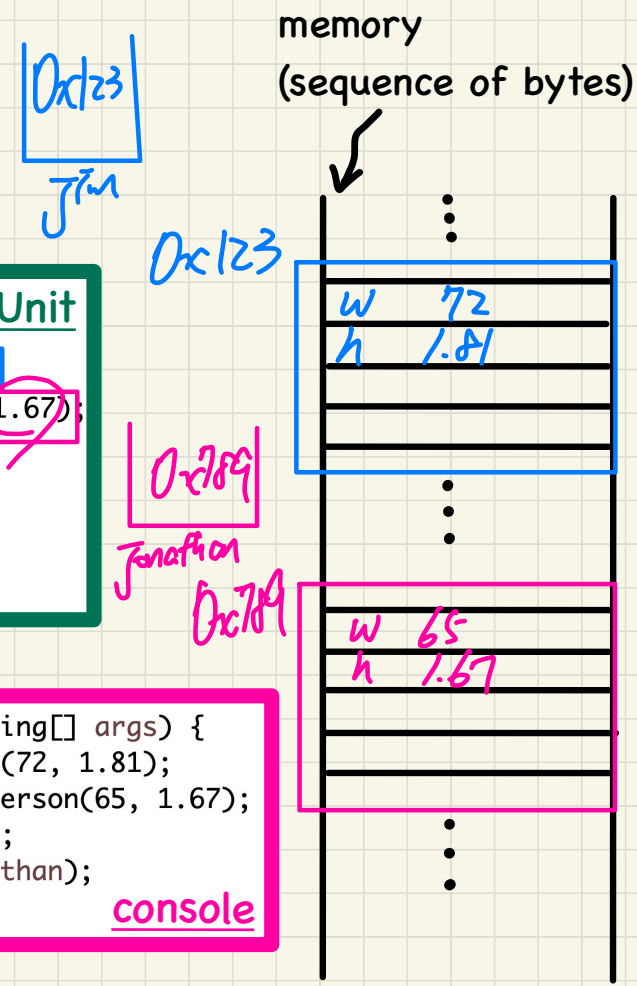
T

F

**console**

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

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



(e.g. assertTrue, assertFalse)

JUnit assertions : pass vs fail

(e.g. jon == jon vs jon != jon)

boolean expressions : True vs. false

assertTrue( boolean exp )

- true → pass
- false → fail

assertFalse( )

- true → fail
- false → pass

# Use of Accessors vs. Mutators

non-void

void

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

- Calls to *mutator methods* *cannot* be used as values.
  - e.g., `System.out.println(jim.setWeight(78.5));` ✗
  - e.g., `double w = jim.setWeight(78.5);` ✗
  - e.g., `jim.setWeight(78.5);` ✓
- Calls to *accessor methods* *should* be used as values.
  - e.g., `jim.getBMI();` ✗
  - e.g., `System.out.println(jim.getBMI());` ✓
  - e.g., `double w = jim.getBMI();` ✓

compiles but useless

# Lecture 4 - Sep 16

## OOP Review

***Tracing OOP: New Obj. & Method Calls***
***Variable Shadowing, Use of this***
***Reference Aliasing***

# Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  - + Lab0P2 (tutorial videos + PDFs)
  - + Due: This Tuesday (Sep 16)
- Lab1 released today (Sep 16)
- Lab last Wednesday (Sep 10): Demo on helper methods
- ProgTest0 (next Wednesday, Sep 24), Guide released
- Lab this Wednesday (Sep 17):
  - + Please go to your enrolled session.
  - + [≈ 40 min] Mock-up Programming Test 0
  - + [≈ 40 min] Q&As (TAs, Jackie)

# Constructors not using this Keyword

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

    /*
     * Constructors
     */
    public Person() {

    }

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

**model**

weight height weight height

leave attrs default values

ref. to the parameter "weight"

variable shadowing

Use "this" keyword to denote the context object

## Question

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

# Tracing OO Code: Visualizing Objects

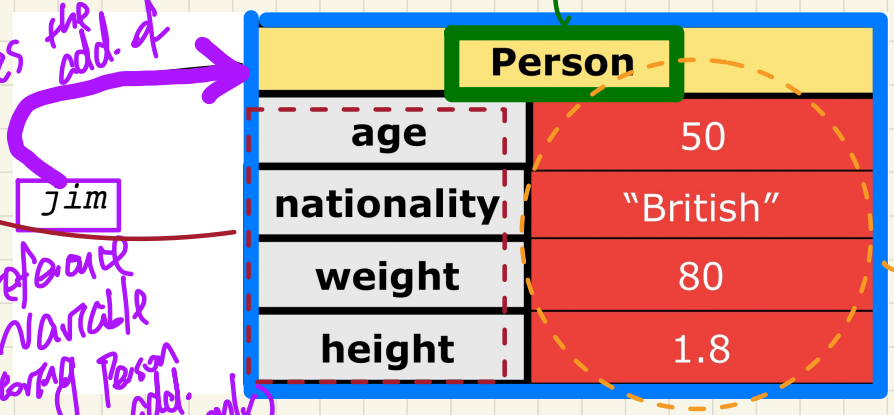
type of object

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*.

(selected) list of attributes

stores the add. of




reference variable (storing Person add. only)

object in memory.

instance-specific att. values

# Effects of Creating New Objects

**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 weight, double height) {
        this.weight = weight;
        this.height = height;
    }
}
```

overloading

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

72 65 1.81 1.67

this

jonathan

| Person | |
|---|---|
| w | 65 |
| h. | 1.67 |

**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);
}
```

T F

* Person jim = new Person(72, 1.81);

① ③ ②

① declares a ref. var. "jim" that stores add/ref of some Person obj.

jim

this

② 

| Person | |
|---|---|
| w. | 72 |
| h. | 1.81 |

③ stores add. of new obj. to the ref. var.

# 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 jon. jim jon jim jon jim jon 3 3

| Person | |
|---|---|
| w. | 72 |
| h. | 1.81 |

jim 72

| Person | |
|---|---|
| w. | 65 |
| h. | 1.67 |

jonathan 68

```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);
}
```

x

assertEquals( x, y, 0.01 )

double double

ε (tolerance)

↳ assertTrue( $|x - y| \le 0.01$ )

# Copying Primitive vs. Reference Values



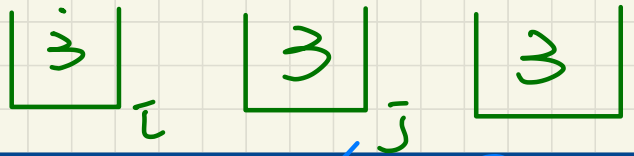
## Primitive

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

copy what's stored in i, and store that value to j.

T T

and → T




## Reference

①

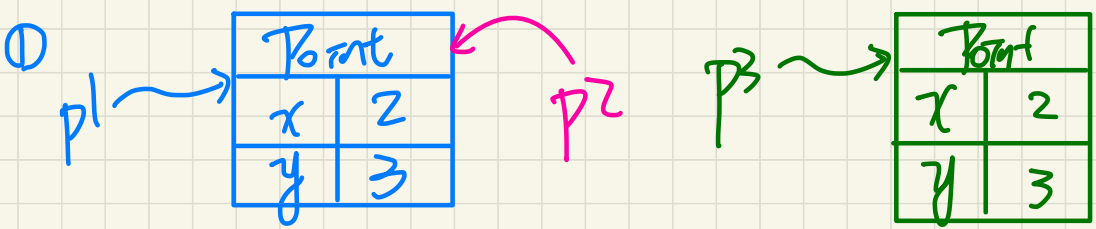
```
Point p1 = new Point(2, 3);
Point p2 = p1;
Point p3 = new Point(2, 3);
Systme.out.println(p3 == p1 || p3 == p2);
Systme.out.println(p3.x == p1.x && p3.y == p1.y);
Systme.out.println(p3.x == p2.x && p3.y == p2.y);
```

**

F T

or F

①




** After exec. this var. assignment, p1 and p2 point to same obj.

# Copying Primitive Values

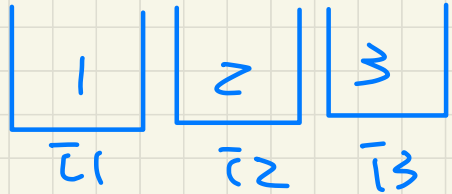


```
int i1 = 1;
int i2 = 2;
int i3 = 3;
int[] numbers1 = {i1, i2, i3};
int[] numbers2 = new int[numbers1.length];
for (int i = 0; i < numbers1.length; i ++) {
  numbers2[i] = numbers1[i];
}
numbers1[0] = 4;
System.out.println(numbers1[0]);
System.out.println(numbers2[0]);
```

ref. var. stores the starting point of an array, where each element is an int.

0 1 2 3

4 1




i

0 ns2[0] = ns1[0]

1 ns2[1] = ns1[1]

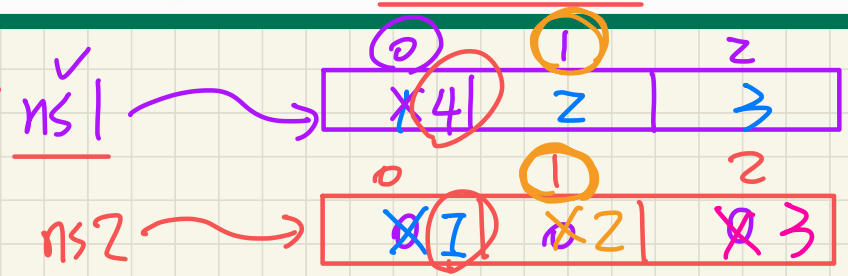
2 ns2[2] = ns1[2]

3

3 < 3

F exit

ns1.length !! 3



# Lecture 5 - Sep 18

## OOP Review

***Exercise: Reference Aliasing & Arrays***
***Manual Management of Account IDs***
***Use of Static vs. Non-Static Variables***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Review Lab0
  + Complete Lab1
  + Due: Next Tuesday (Sep 30)
- ProgTest0 (next Wednesday, Sep 24), Guide released
- Mock-up Programming Test 0 to be released

** copy the add. stored in alan into index 0 of array ps1.

# Copying Reference Values: Aliasing



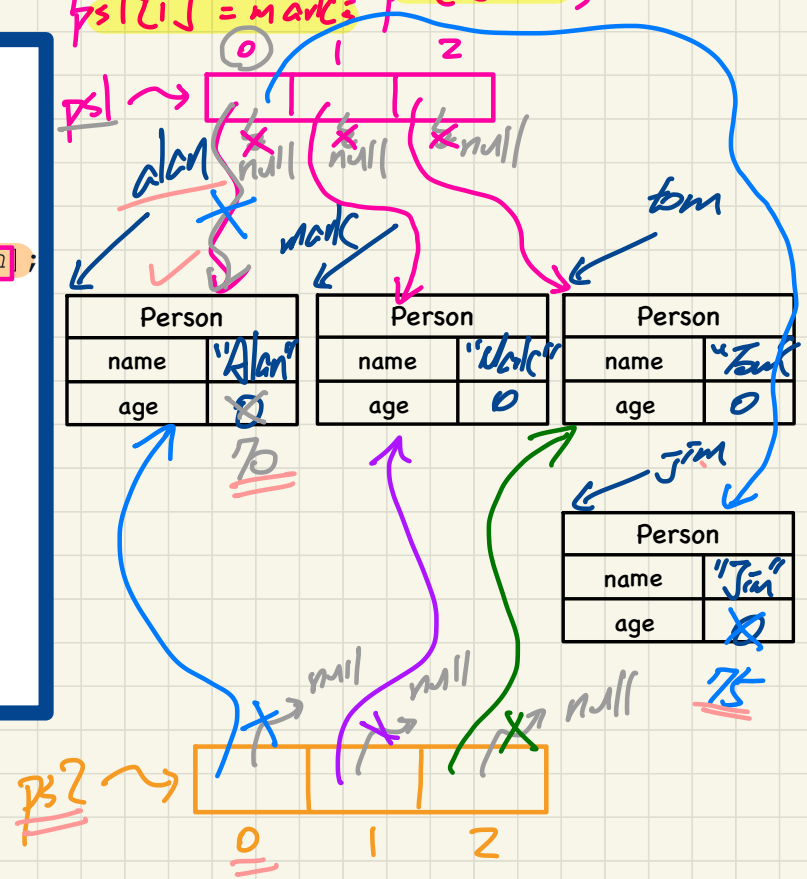
```
Person alan = new Person("Alan");
Person mark = new Person("Mark");
Person tom = new Person("Tom");
Person jim = new Person("Jim");
Person[] persons1 = {alan, mark, tom};
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons1.length; i ++) {
  persons2[i] = persons1[i]; }
persons1[0].setAge(70);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

*** i

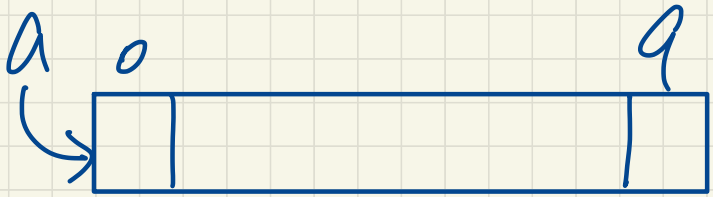
0 ps2[0] = ps1[0];

1 ps2[1] = ps1[1];

2 ps2[2] = ps1[2];



$a[i]$




① $a \; != \; null$

$a.length == 10$

② $0 \le i \le a.length - 1$

# Copying Arrays

```
Person[] ps1;
Person[] ps2;
```

①

```
ps1 = new ...
ps2 = new ...
```

②

```
ps2 = ps1;
```

Creating aliases at the array level.

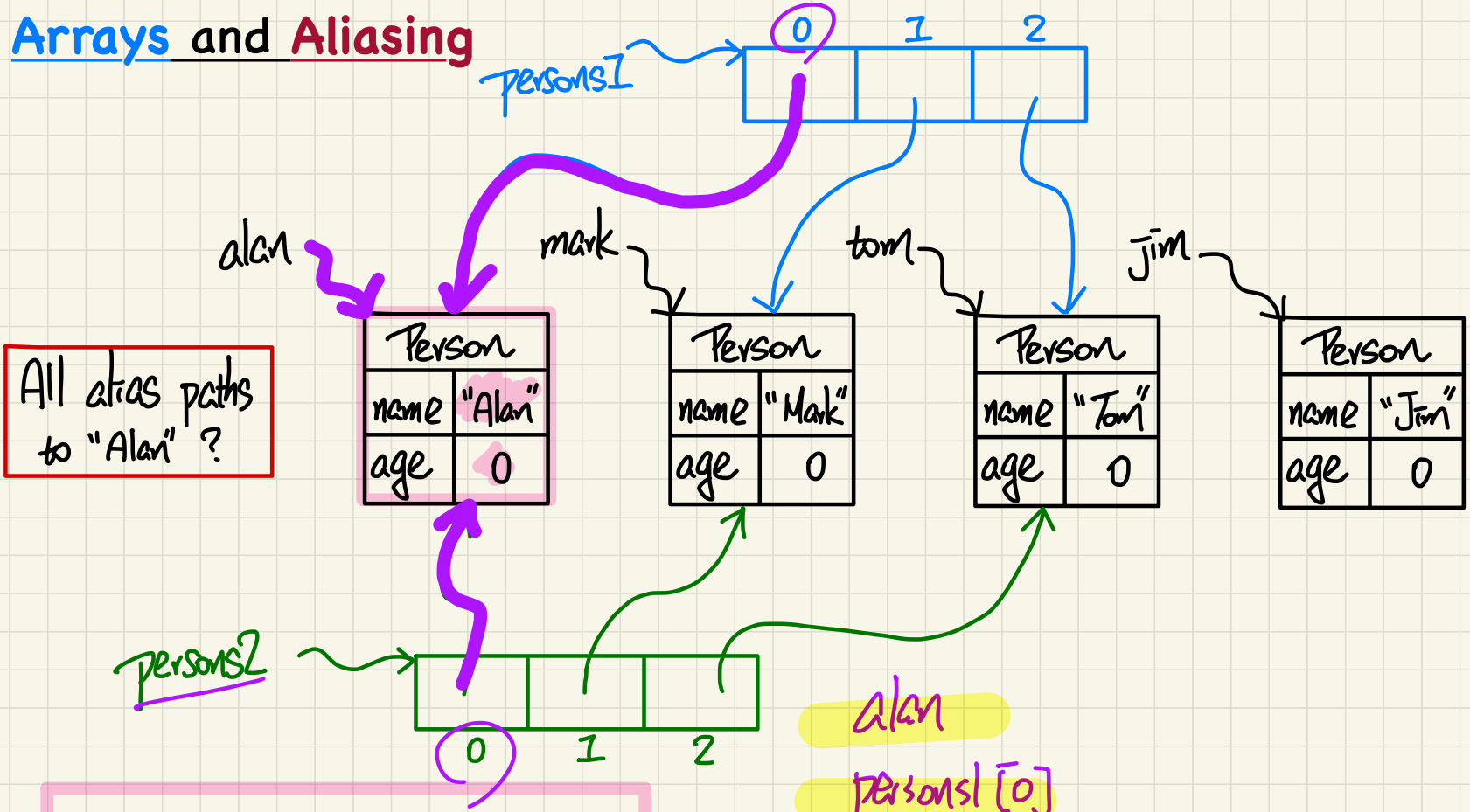
③

```
ps1 = null;
```

④

| | ① | ② | ③ | ④ |
|---|---|---|---|---|
| ps1==null | T | F | F | T |
| ps2==null | T | F | F | F |
| ps1==ps2 | T | F | T | F |

# Arrays and Aliasing




All alias paths to "Alan"?

alan

persons1[0]

persons2[0]

Q. All alias paths to "Alan"?

# 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;
  }
}
```

explicit parameters presumably should not clash with existing id's

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

Goal: auto. generate & manage id's

2. when id's are duplicated, not a compilation error (logical error)

1. burden on the Account user to specify unique id's.

# Declaring Global Variables among Objects

* Each Counter object has its own copy (instance-specific value).

** All Counter objects share a single/unique copy (global var.)

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

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

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

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

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

* non-static variable (attribute)

static variable

** static var init. as soon as its decl.

non-static var init. in constructor.

c1 c2

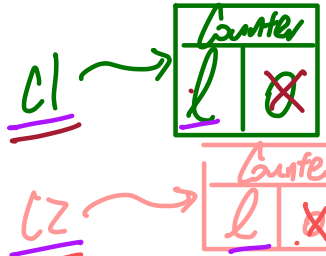
c1 c2

this.g ++; (compiles, with a warning).

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

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

    ① c1.incrementLocal();
    ② c2.incrementLocal();
    ③ c1.incrementGlobal();
    ④ c2.incrementGlobal();
    ⑤ Counter.g = Counter.g + 1; // Counter.global ++;
    }
}
```

Counter.g

Accessing static variables: ClassName . nameOfStaticVar.

# Lecture 6 - Sep 23

## OOP Review

***Use and Misuse of Static Variables***
***Use of this: Marriage Example***
***Reference-Typed Return Values***
***Anonymous Objects***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Review Lab0
  + Complete Lab1
  + Due: Next Tuesday (Sep 30)
- ProgTest0 (this Wednesday, Sep 24):
  + Guide & Mockup Test released
  + **Required**: Go to your **enrolled** session.
  + A physical **photo ID** is required.

Alt String id_str = "Id" + id;

id_str
"Id1"
"Id2"
!

# Managing Account IDs: Automatic



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

global

local

acc1 acc2

list of parameters not including id anymore (not burden of user of Account const.)

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

0

1 1 2

acc1 → c.o.

| Account | |
|---|---|
| id | 1 |
| owner | "Jim" |

acc2 → c.o.

| Account | |
|---|---|
| id | 2 |
| owner | "Jeremy" |

0

| Account | |
|---|---|
| gC* | ~~1~~ |

~~2~~

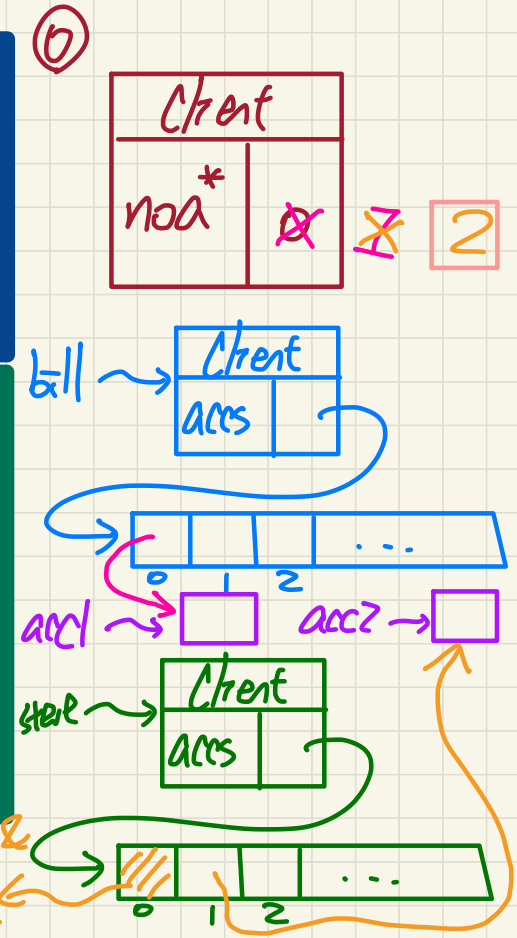
3

# 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 ++;
  } }
```

```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] */
}
```



# Use of Static Variables: Common Error



```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;
    }
}
```

Compilation Error: non-static var (branchName) used in static context (getInfo)

Expected Usage: Bank.getInfo

not a context object → Bank (not an object!).branchName (non-static)

Fix 1

```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;
    }
}
```

remove the usage of non-static var.

return value of global counter.

Fix 2

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

not a good design :(

branch name should be instance-specific.

# Example: Reference to this



recursive type

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

spouse is null by default.

* this.spouse == null && other.spouse == null

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

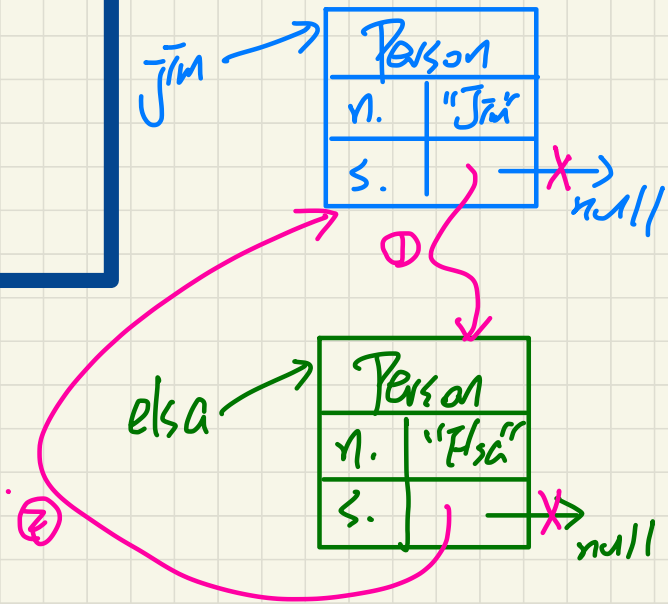
C.O. other

this other

① jim.spouse = elsa;

② elsa.spouse = jim;

other this



| Math | Programming |
| --- | --- |
| (conjunction, and) $\wedge$ | && |
| (disjunction, or) $\vee$ | \|\| |
| (negation) $\neg$ | ! |

short-circuit evaluation.

## Exercise

illegal ≡ !(legal).

**De Morgan**

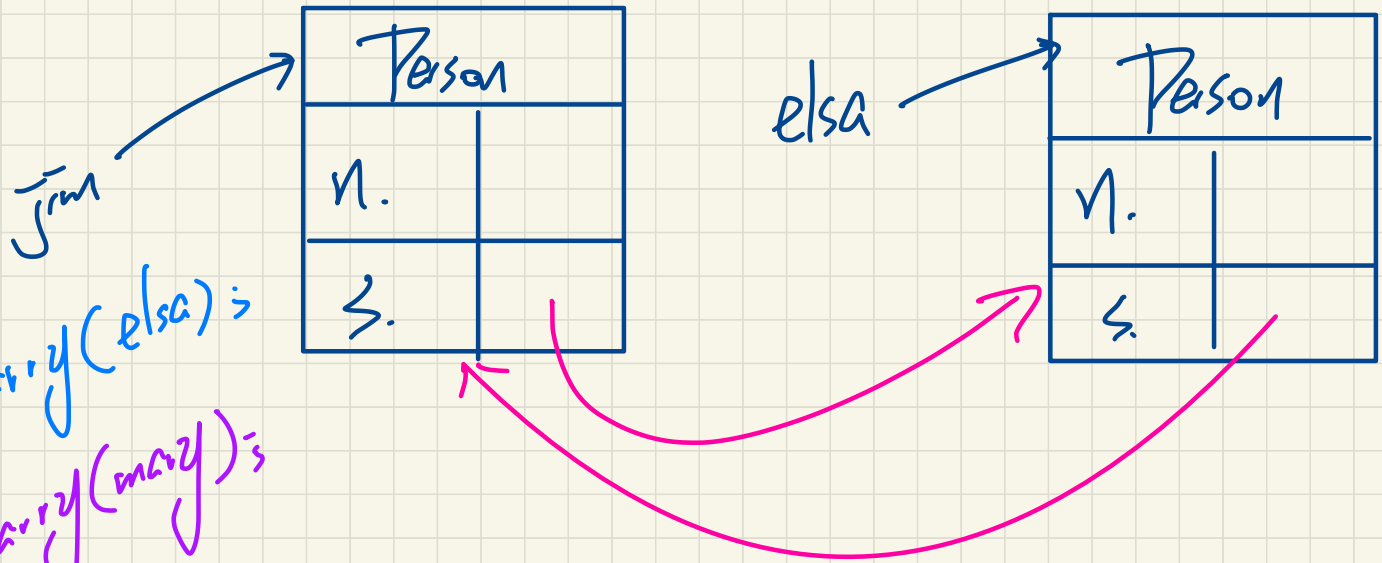
$\neg(p \wedge q) \equiv \neg p \vee \neg q$

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

this.spouse == null

other.spouse == null

```
public void marry(Person other) {
    if (* proposed marriage is illegal) { /* error */ }
    else {
        ①
        ②.
    }
}
```




Jim.marry(elsa);

Jim.marry(mary);

Jim → not single; should be some error.

# Reference-Typed Return Values



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

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

```java
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);
   }
}
```

# Anonymous Objects



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

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

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

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

## Exercise

```java
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) {



  }
}
```

# Lecture 7 - Sep 25

## OOP Review

***Caller vs. Callee***
***Tracing Chain of Method Calls via a Stack***
***Catch-or-Specify Requirement***
***To Handle or Not to Handle: V1***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Review Lab0
  + Complete Lab1; Due: Next Tuesday (Sep 30)

method that calls another method → callee

# Caller vs. Callee

method that's called by another method → caller.

* Each method call defines a caller-callee relation.

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

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

caller

type of c.o.

c.o.

caller-callee relation.

Exercise: Make C2.m2 a caller.

callee: C2.m2 → does not mean that m2 is static!

caller: C1.m1

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

(Alt 1) Make C1.m1 a callee.

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

callee

caller

```
class C {
    void m( ) {
        this.m( );
    }
}
```

C.m is also the callee.

# Visualizing a Call Chain using a Stack

entry point of execution

m1 ( ) {

① ② m2( );

③ ④ ⑮

} m1 terminates

caller: m1
callee: m2

when m2 is being executed, m1's exec is suspended

m2( ) {

③ ④ ⑤ m3( );

⑪ ⑫

} m2 terminates

caller: m2
callee: m3

m3( ) {

⑥ ⑦ ⑧ ⑨ ⑩

}

m3

m2

m1

called to exec. for latest.

called to execute the evalst

chronological order of methods terminating

chronological order of method calls

call stack

m3 finishes its execution (terminates)
↳ removed from stack

stack ops

add/push

remove/pop

first item to be removed

top of stack

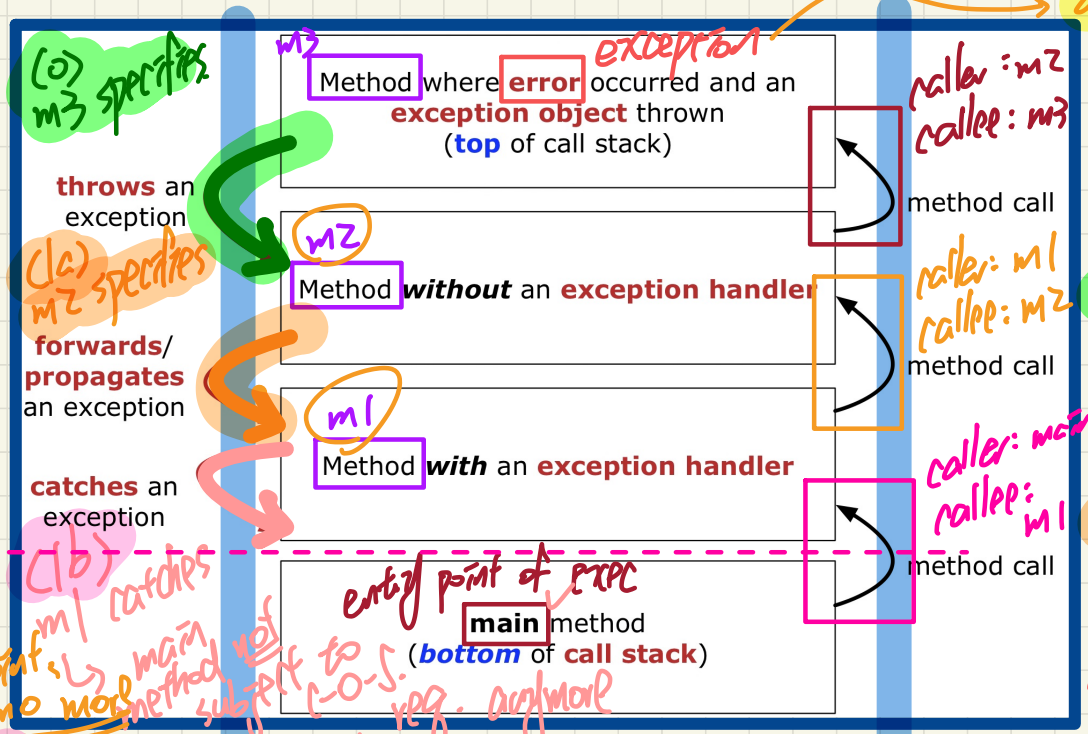
bottom of stack

(Last-In, First-Out)
LIFO

order of additions

order of removals.

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




as soon as an exception object is thrown, the call stack steps going.

## Catch-or-Specify Req.

(0) Where the exception is originated, specify it.

(1a) A method may specify the exception, which is propagated to its caller.

(1b) A method may catch the exception.

↳ beyond this method, all subsequent callers not subject to C-O-S Req.

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

```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.

| context | caller | callee |
|---|---|---|
| A | / | / |
| B | B.mb | A.ma |
| Tester | Tester.main | B.mb |

call stack

A.ma
B.mb
Tester.main

# Lecture 8 - Sep 30

## Exceptions

***To Handle or Not to Handle: Versions 2, 3***
***Generalizing Exception Handling***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Complete Lab1; Due: This Tuesday (Sep 30)
  + Lab2 to be released
- ProgTest1
  + guide released
  + PracticeTest1 released
  + In-Person Review Session at 2 PM, Friday, Oct 3 (CLH C)

# Version 1: B.mb Catches

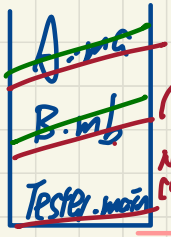
```
class A {
  A() {}

  void ma(int i) throws NegValException {
    if(i < 0) {
      println("abnormal exec of A.ma");
      throw new NegValException("Neg Val: " + i);
    }
    else {
      println("normal exec of A.ma");
    }
  }
}
```

```
class B {
  B() {}

  void mb(int i) {
    A oa = new A();
    try {
      oa.ma(i);
      println("From B.mb: Calling A.ma did not cause NVE.");
    }
    catch(NegValException nve) {
      println("From B.mb: Calling A.ma caused NVE.");
    }
  }
}
```

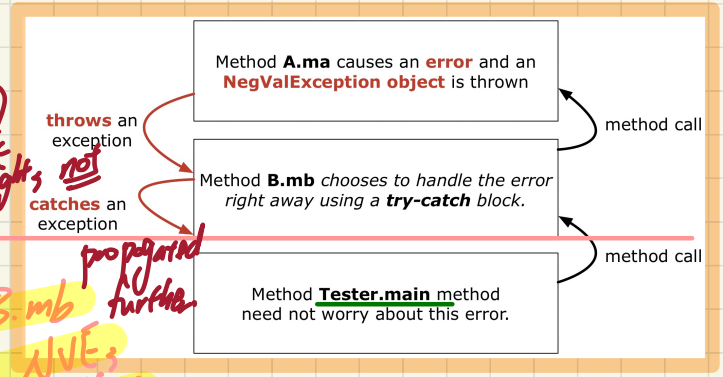
```
class Tester {
  main(...) {
    int i;
    B ob = new B();
    ob.mb(i);
    println("From Tester.main: After calling B.mb.");
  }
}
```

normal exec of A.ma
B.mb: calling A.ma did not cause NVE.
Tester.main: after calling B.mb.

abnormal exec. of A.ma
B.mb: calling A.ma caused NVE.
Tester.main: After calling B.mb.




NVE caught, not propagated further




Since B.mb catches NVE, subsequent callers no longer subject to c-or-s req.

**Q1**. In B.mb, is calling oa.ma subject to **catch-or-specify req**?

Yes ∵ its callee A.ma specifies TE.

**Q2**. In Tester.main, is calling B.mb subject to **catch-or-specify req**?

No ∵ its callee B.mb catches that.

# Version 2: B.mb Specifies, Tester.main Catches

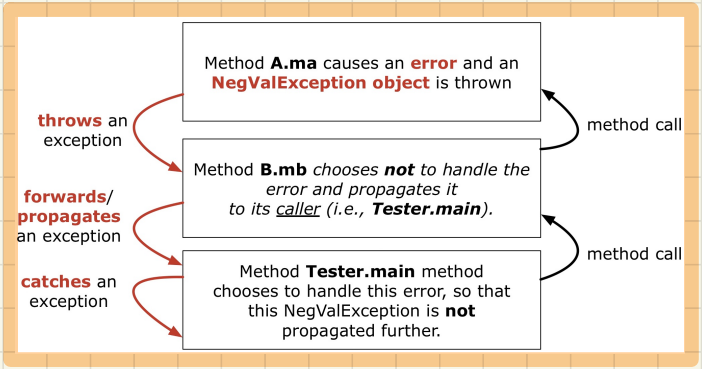
```
class A {
  A() {}

  void ma(int i) throws NegValException {
    if(i < 0) {
      println("abnormal exec of A.ma");
      throw new NegValException("Neg Val: " + i);
    }
    else {
      println("normal exec of A.ma");
    }
  }
}
```

```
class B {
  B() {}

  void mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
    println("From B.mb: Calling A.ma did not cause NVE.");
  }
}
```

```
class Tester {
  main(...) {
    int i;
    B ob = new b();
    try {
      ob.mb(i);
      println("Tester.main: Calling B.mb did not cause NVE.");
    }
    catch(NegValException nve) {
      println("Tester.main: Calling B.mb caused NVE.");
    }
  }
}
```

4
-5

could be specify option (version 3)




**Q1**. In B.mb, is calling oa.ma subject to **catch-or-specify req**?

Yes ∵ A.ma specifies it.

**Q2**. In Tester.main, is calling B.mb subject to **catch-or-specify req**?

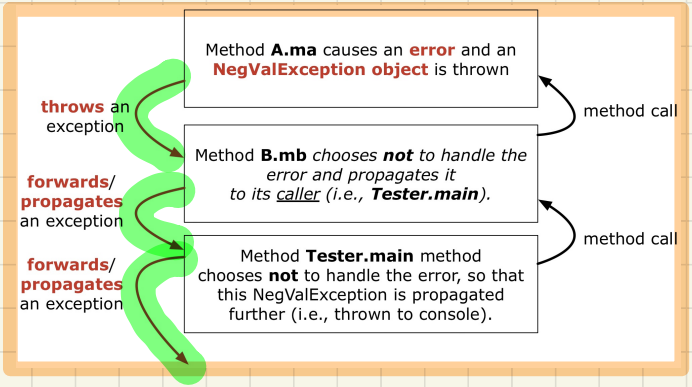
Yes ∵ B.mb specifies it

# Version 3: B.mb Specifies, Tester.main Specifies

```
class A {
  A() {}

  void ma(int i) throws NegValException {
    if(i < 0) {
      println("abnormal exec of A.ma");
      throw new NegValException("Neg Val: " + i);
    }
    else {
      println("normal exec of A.ma");
    }
  }
}
```

```
class B {
  B() {}

  void mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
    println("From B.mb: Calling A.ma did not cause NVE.");
  }
}
```

```
class Tester {
  main(...) throws NegValException {
    int i;
    B ob = new b();
    ob.mb(i);
    println("Tester.main: Calling B.mb did not cause NVE.");
  }
}
```




**Q1**. In B.mb, is calling oa.ma subject to **catch-or-specify req**?

Yes. A.ma specifies it

**Q2**. In Tester.main, is calling B.mb subject to **catch-or-specify req**?

Yes. B.mb specifies it.

# Catch-or-Specify Requirement: Call Stack

**Q1**. Origin of Exception:

$C_1.m_1$

**Q2**. Methods subject to CoS Req:

$i$ methods ($i-1$ methods specify, 1 method catch).

**Q3**. Methods free from CoS Req:

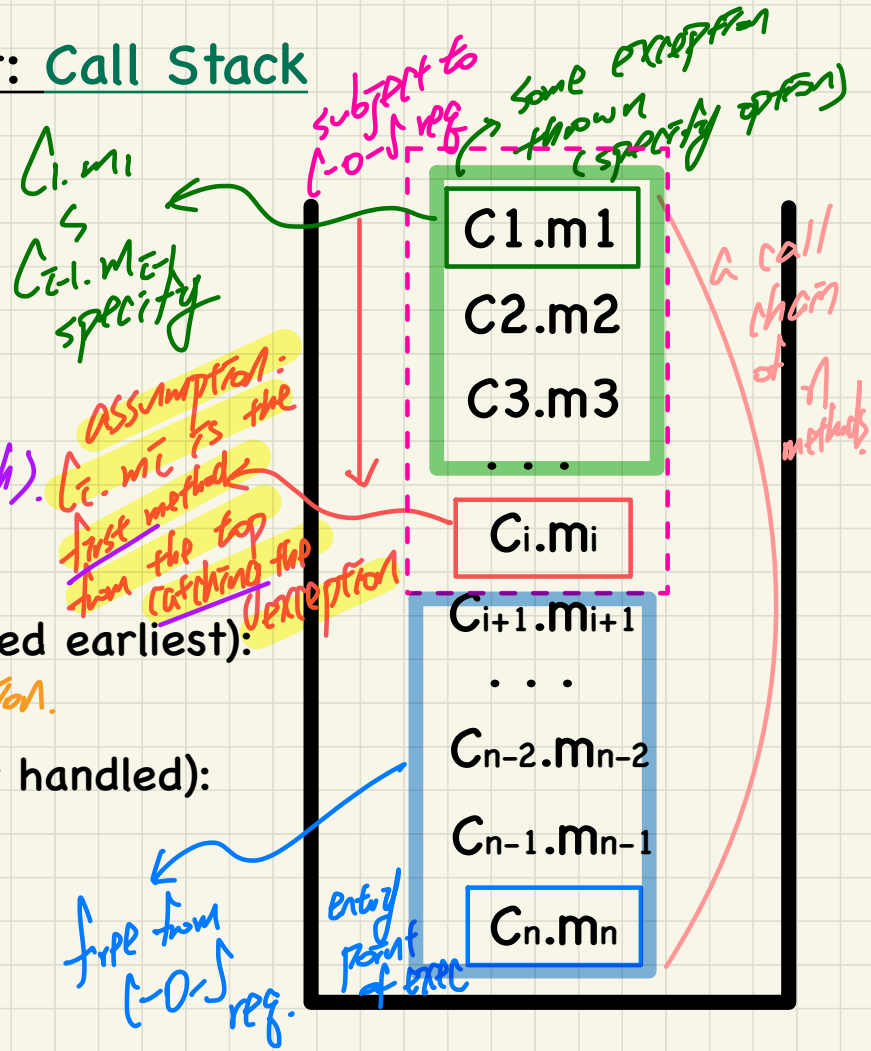
$n-i$ methods

**Q4**. Extreme Case 1 (exception handled earliest):

$C_2.m_2$ catches the exception.

**Q5**. Extreme Case 2 (exception never handled):

$C_1.m_1 \sim C_n.m_n$ all specify.



# Lecture 9 - Oct 2

## Exceptions

***Execution Flows: Normal vs. Abnormal***
***Examples: Circle, Bank, ParseInt***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Lab1 solution video released
  + Lab2 released
- ProgTest1
  + guide released
  + PracticeTest1 released
  + In-Person Review Session at 2 PM, Friday, Oct 3 (CLH C)

# Catch-or-Specify Requirement: Execution Flows (1)

**Scenario**: Current caller chooses to **catch/handle** the exception.

## Normal Flow of Execution

```
① ... /* before, ouside try-catch block */
  try {
② o.m(...); /* may throw SomeException */
③ ... /* rest of try-block */
  }
  catch (SomeException se) {
    ... /* rest of catch-block */
  }
④ ... /* after, ouside try-catch block */
```

exception not thrown

When the exception does not occur

bypassed
∵ no exception occurred

## Abnormal Flow of Execution

```
① ... /* before, ouside try-catch block */
  try {
② o.m(...); /* may throw SomeException */
  ... /* rest of try-block */
  }
  catch (SomeException se) {
③ ... /* rest of catch-block */
  }
④ ... /* after, ouside try-catch block */
```

exception thrown

When the exception occurs

rest of try block bypassed
∵ exception occurred.

# Catch-or-Specify Requirement: Execution Flows (2)

**Scenario**: Caller chooses to **specify/propogage** the exception.

**Normal** Flow of Execution

```
class C1 {
  void m1 throws SomeException {
  1 ... /* some code */
  2 ... /* some code */
  3 C2 o = new C2();
  4 o.m();
  5 ... /* some code */
  6 ... /* some code */
  }
}
```

exception not occurred

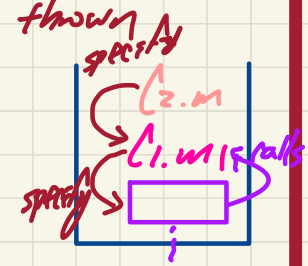
When the exception does not occur

**Abnormal** Flow of Execution

```
class C1 {
  void m1 throws SomeException {
  1 ... /* some code */
  2 ... /* some code */
  3 C2 o = new C2();
  4 o.m();
    ... /* some code */
    ... /* some code */
  }
}
```

exception thrown

exception disrupted the flow and thrown to the caller of C1.m1




When the exception occurs

# Error Handling via Exceptions: Circles (Version 2)

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

**Test Case:**

User enters -5

Then user enters 10

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

terminates setR met. abnormally

terminates setR normally

true

~~false~~

TRIV

~~-5~~ 10

Exercise 1 23

Exercise 2 -4, -3, 10

```
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!"); }
    } } }
```

-5 → IRE thrown

10 → IRE not thrown

Console

```
Enter a radius:
-5
Try Again
Enter a radius:
10 Area is ...
```

~~C.setR~~

~~CC2.main~~

!C

while ( B ) {

}

repeated as long as B is true.

repeated as long as C is false.

# Error Handling via Exceptions: Circles (Version 1)

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

**Test Case 1:**

User enters 10

**Test Case 2:**

User enters -5

```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;
}
```

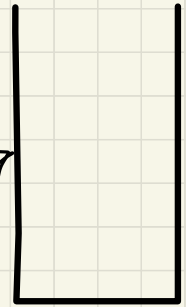
| caller | callee |
|---|---|
| | |
| | |

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

Caller?

Callee?

call stack



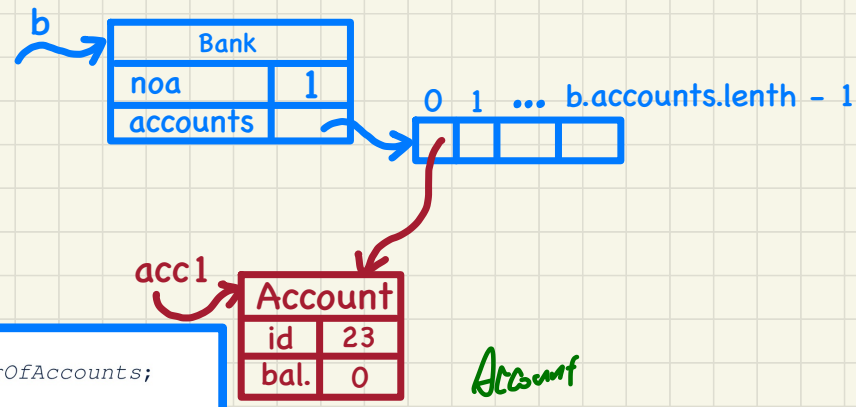
# Error Handling via Exceptions: Banks

```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; }
  }
}
```

```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 */ }
```

```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); } } }
```




Account

accounts[i].withdraw(..)

Account[]

Account.withdraw

Bank.withdraw

BA.main

**Test Case:**

User enters -5000000

# More Example: Multiple Catch Blocks

Assumption
Customized exceptions are unrelated to each other
↳ their catch blocks can be arranged in any order

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

may throw ITE → not thrown

-5 may throw NRE → thrown

**Test Case 1:**

a: -5000000

r: 23

**Test Case 2:**

a: 100

r: -5

# More Example: Parsing Strings as Integers

~~false~~ true

vI

```
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"

NFE thrown

Enter an integer

twenty-three

Not valid

Enter an integer:

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

```
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

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

# Review: Specify-or-Catch Principle

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

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

# A Class for Bounded Counters

start.

0 1 2 3

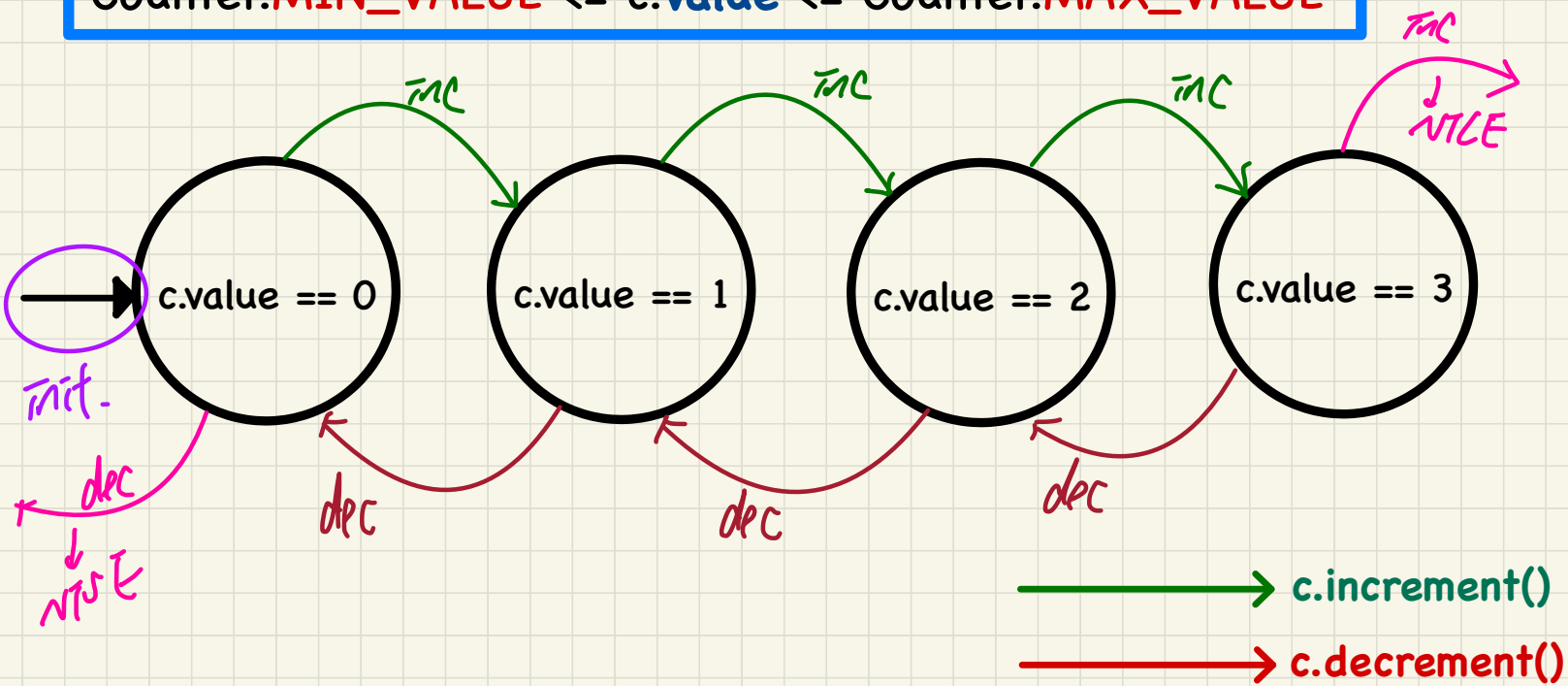
```java
public class Counter {
  public final static int MAX_VALUE = 3;
  public final static int MIN_VALUE = 0;
  private int value;
  public Counter() {
    this.value = Counter.MIN_VALUE;
  }
  public int getValue() {
    return value;
  }
  ... /* more later!
```

```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 --; }
  }
}
```

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

Boundries:

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



# Lecture 10 - Oct 7

## TDD with JUnit

***Regression Testing***
***Example Test 1: VTLE Not Expected***
***Example Test 2: VTSE Expected***

## Announcements/Reminders

- Today's class: notes template posted
- Priorities:
  + Lab1 solution video released
  + Lab2 released
- ProgTest1
  + guide released
  + PracticeTest1 released
  + Last Friday's Review session materials posted

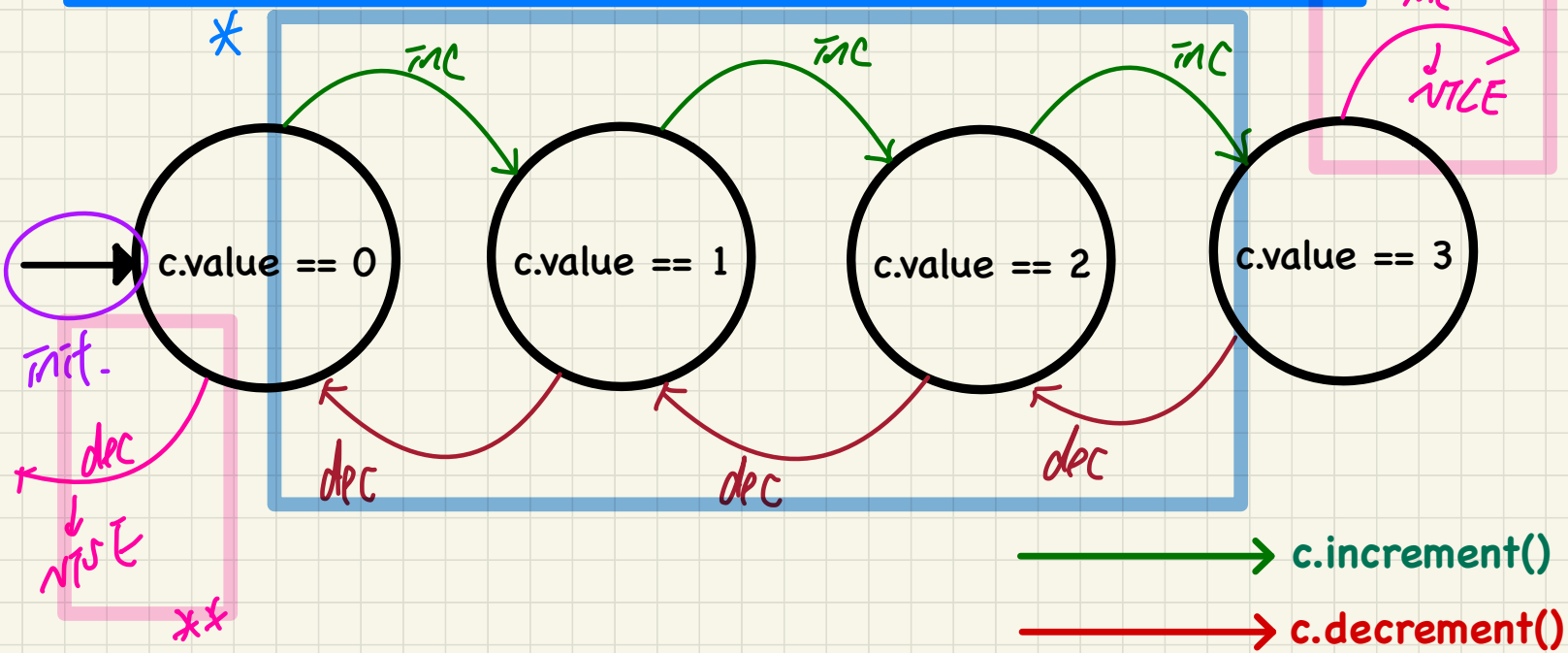
* VTCE or VTSE not expected.

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

** VTCE expected for inc, VTSE expected for dec

Boundries:

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



# JUnit Test Method vs. Method Under Test

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

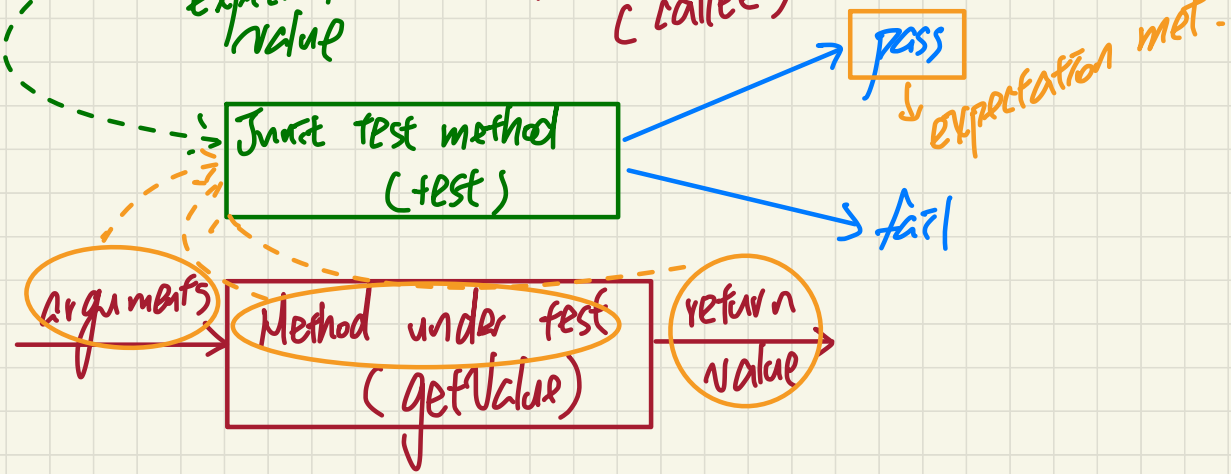
(JUnit) test method (caller)

value produced by the implementation

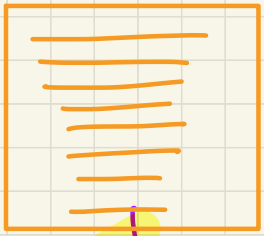
expected value

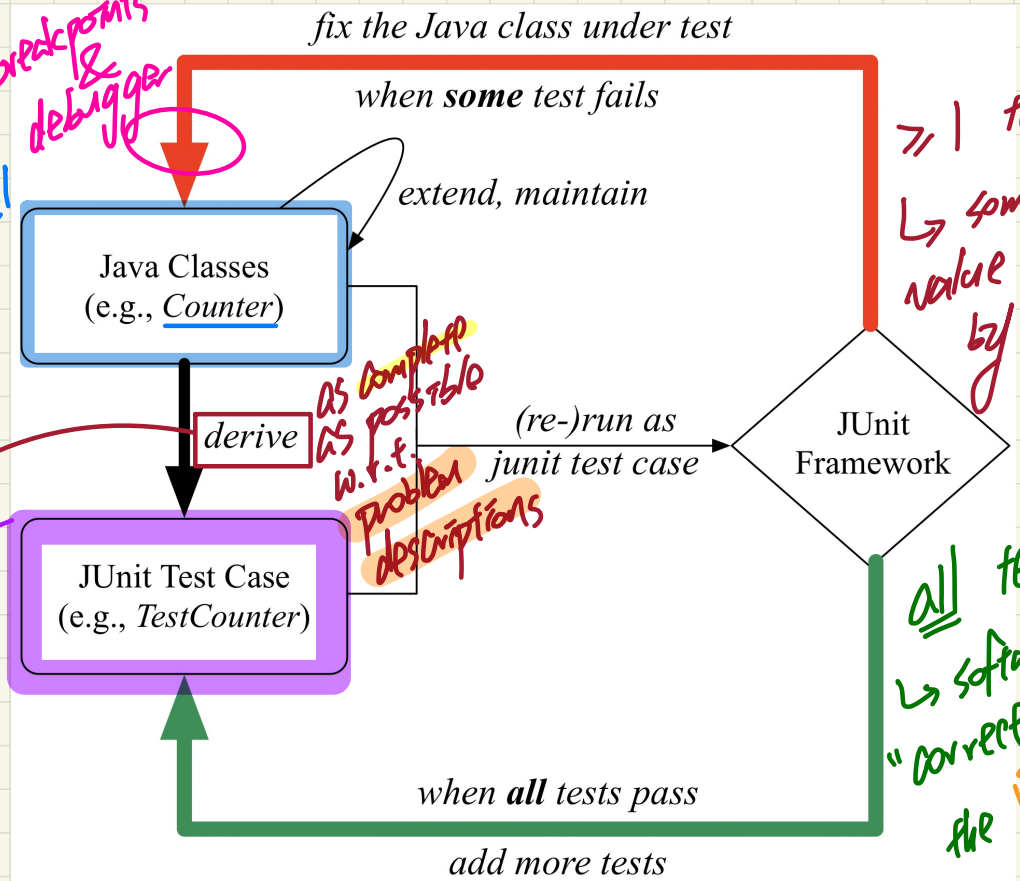
expected value

method under test (callee)



# Test-Driven Development (TDD): Regression Testing






breakpoints & debugger

model

incremental development.

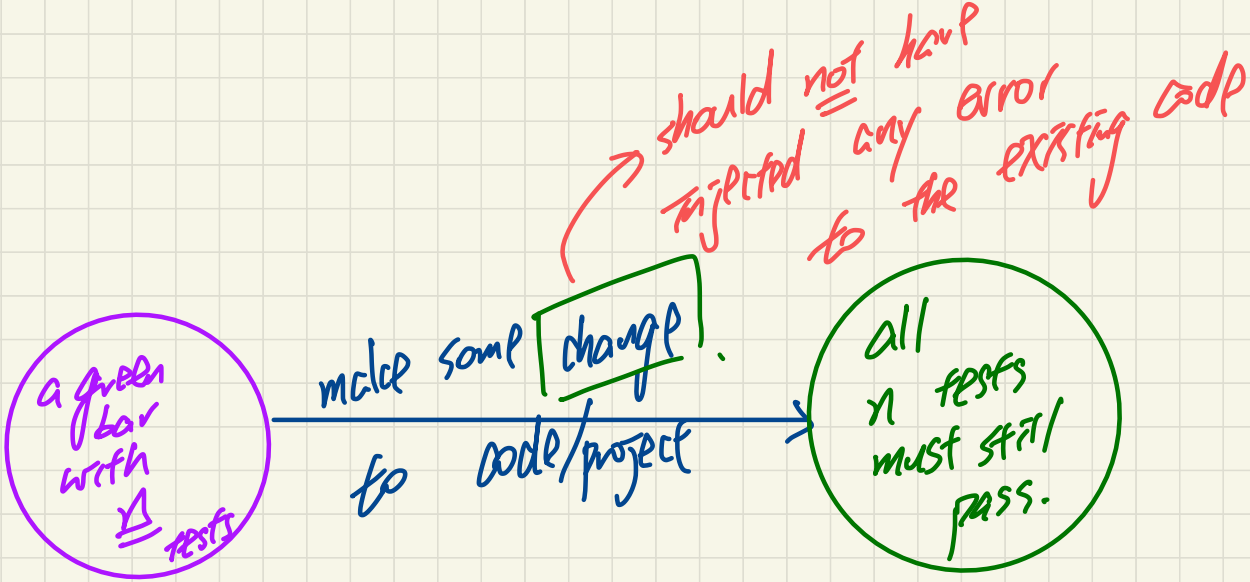
⇒ as soon as a unit of implementation is completed, write & run test cases right away.

EECS4313

as complete as possible w.r.t. problem descriptions

≥ 1 test failed.
↳ some expected value not met by implementation
* all scenarios of inputs.

all tests pass
↳ software is "correct" relative to the tests run.
↳ diverse enough to cover *


a green bar with n tests
make some change to code/project
should not have injected any error to the existing code
all n tests must still pass.

## Self Studies

Creating & Running JUnit Test Cases: Slides 12 to 21

Common Assertions in JUnit: Slide 22

Examples on JUnit: Slides 23 - 25

# A Default Test Case that Fails

slide 16

The result of running a test is considered:

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

fail() fail("...")

fail the test right away unconditionally.

slide 19

TestCounter.java

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

doing nothing
↳ test passes trivially!

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

# Design of a Test

Test

**Expectation**: certain kind of exception occurs

- ↳ that kind of exception is thrown → pass
- ↳ that kind of exception not thrown → fail

method under test.

Test

**Expectation**: certain kind of exception not thrown

- ↳ that kind of exception is thrown → fail
- ↳ that kind of exception not thrown → pass

# JUnit: An Exception 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(ValueTooLargeException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooLargeException is not expected.");
  }
}
```

may or may not throw VTLE

VTLE not thrown (correct)

pass.

What if increment is implemented correctly?

test pass

may or may not be true?

Expected Behaviour: correctly

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

expectation of test.

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

VTLE thrown (incorrect)

fail.

What if increment is implemented incorrectly?

test fail

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

# Running JUnit Test 1 on Correct Implementation

Expectation

VTLE not thrown.

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

0 → 1.

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

c.v == 0

VTLE not thrown

pass

test passed

∵ imp. does not throw VTLE as expected.

# Running JUnit Test 1 on Incorrect Implementation

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

unexpected.

Expectation

VTLE not thrown

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

c.v == 0

VTLE thrown unexpectedly

fail

test failed

∵ Test has thrown the VTLE unexpectedly.

# JUnit: An Exception Expected

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

VTSE thrown (correct)

pass.

What if decrement is implemented correctly?

VTSE thrown → pass

## Expected Behaviour:

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

expectation: VTSE should be thrown.

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

VTSE not thrown (incorrect).

fail.

What if decrement is implemented incorrectly?

VTSE not thrown → fail

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

# Running JUnit Test 2 on Correct Implementation

Expectation
VTSE thrown.

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

Correct behaviour!

```
@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. */
  }
}
```

C.V == 0

VTSE thrown

pass

test passed
∵ Imp. has thrown VTSE as expected.

# Running JUnit Test 2 on Incorrect Implementation

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

0 → −1

Expectation
VTSE thrown

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

1 c.v == 0

0

→ VTSE not thrown!

fail

test fail

∵ Imp. did not throw VTSE as expected.

# JUnit: Exception Sometimes Expected, Somtimes Not

```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. */
  }
}
```

c.v == 0

1 2 3

4?

P1: VTLE not expected

P2: VTLE expected

## Expected Behaviour:

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

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

# Lecture 11 - Oct 9

## TDD with JUnit

***Example Test 3:***
***  Some Exceptions Expected, Some Not***
***Console Tester vs. JUnit Test***
***Using a for-Loop for***
***Call by Value: Principle, Circle Examples***

## Announcements/Reminders

- Today's class: notes template posted
- Office Hour during Reading Week to be announced

# Running JUnit Test 3 on Correct Implementation

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

0 → 1

1 → 2

2 → 3

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

c.v == 0

c.v == 1

c.v == 2

c.v == 3

3

3

VTLE thrown

pass

test passed

-( imp.

sat expectations of P1 & P2.

# Running JUnit Test 3 on Incorrect Implementation

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

incorrect loh.

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

C.V == 0

VTLE thrown

fail

test failed

-i imp.

failed to sat

exp. of PI

# Running JUnit Test 3 on Incorrect Implementation

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

0 → 1

1 → 2

2 → 3

3 → 4

```
@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. */
  }
}
```

c.v == 0

c.v == 1

c.v == 2

c.v == 3

→ VTLE not thrown →

fail.

test failed

if imp. even though sat P's exp, failed to sat P2's exp.

# Exercise: Console Tester vs. JUnit Test

WTI Example question (nested try-catch)

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

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

Console tester

c.v == 0

VTLE thrown

would not disrupt the exec. flow.

c.v == 0

VTLE thrown

Witness

if (value < max) {
throw VTLE(...)

Console

Error.

Success.

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

# Exercise: Combining catch Blocks?

**Q**: Can we rewrite `testIncFromMaxValue` to:

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

P1

P2

VTLE caught.

from P1 → not expected → fail

from P2 → expected → pass

ambiguous

**Hint**: Say Line 12 is executed,

is it clear if that ValueTooLargeException was thrown as expected?

# Testing Many Values in a Single Test

what if
min : 0
max : 1000

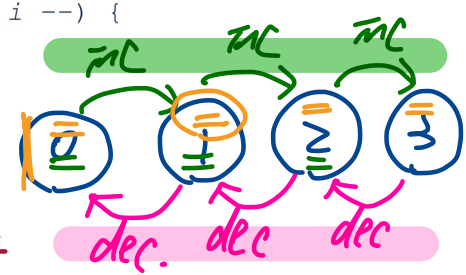
Loops can make it effective on generating test cases:

```
@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");
  }
}
```

0 3 before inc after inc

3 0 before dec after dec

i: 0 1 2



# Method Call: Callee vs. Caller

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

declaration of the caller's method parameter

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

caller: B.n

callee: A.m

value used to call the callee's method

call by value.

Before the 1st line of method m is executed, there's an implicit variable assignment exec:

param = arg

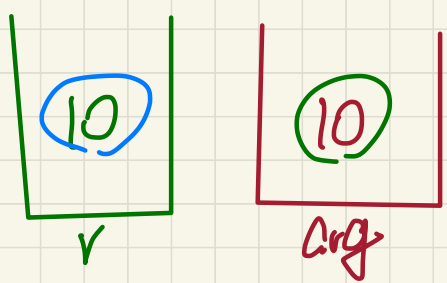
# Call by Value: Primitive Argument

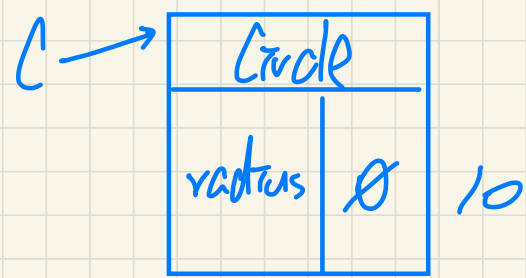
```
class Circle {
   int radius;
   void setRadius(int r) {
   this.radius = r;
   }
}
```

```
class CircleUser {
   ...
① Circle c = new Circle();
② int arg = 10;
③ c.setRadius(arg);
   }
}
```

call by value:

r = arg



# Call by Value: Reference Argument

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

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

call by value: c = arg

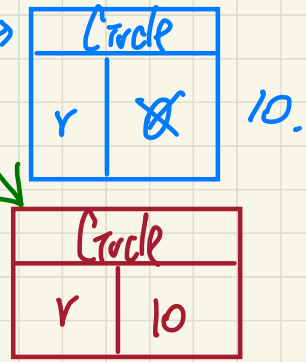
c.o.

c.o.

c

(param) c

arg



# Lecture 12 - Oct 21

## Object Equality

***Call by Value: Prim. vs. Ref. Types***
***Equality: == vs. equals***
***Default Version of equals in Object Class***

## Announcements/Reminders

- Today's class: notes template posted
- ProgTest0, ProgTest1 grading process:
  + TAs are due to return results to me Tuesday afternoon.
  + I will take one day or two to finalize the results.
- WrittenTest1 next Wednesday (October 29)
  + Some practice questions on eClass
  + Lecture 12 and Lecture 13 this week **are** covered.
  + Zoom Review Session: 4 PM on Saturday, October 25
- Lab3 to be released

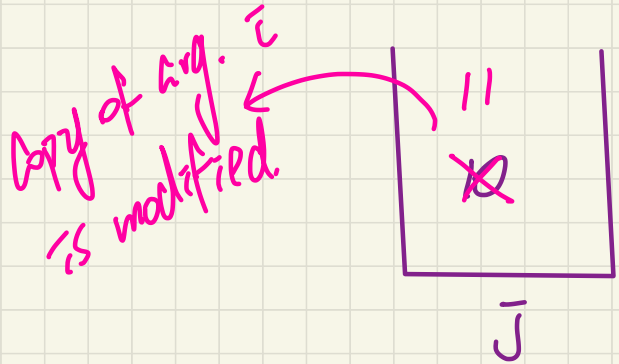
# Call by Value: Re-Assigning Primitive Parameter

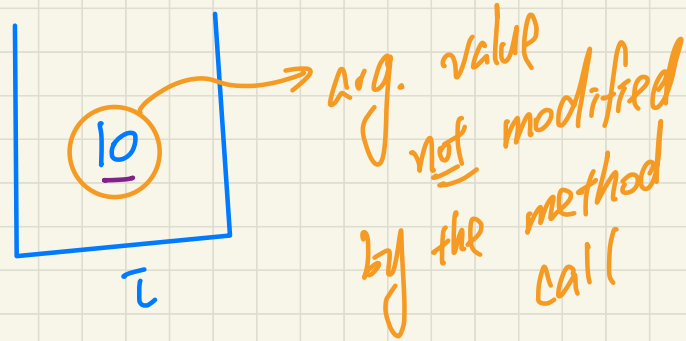
```
public class Util {
  void reassignInt(int j) {
  j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

c.b.v.

⑤ j = i

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



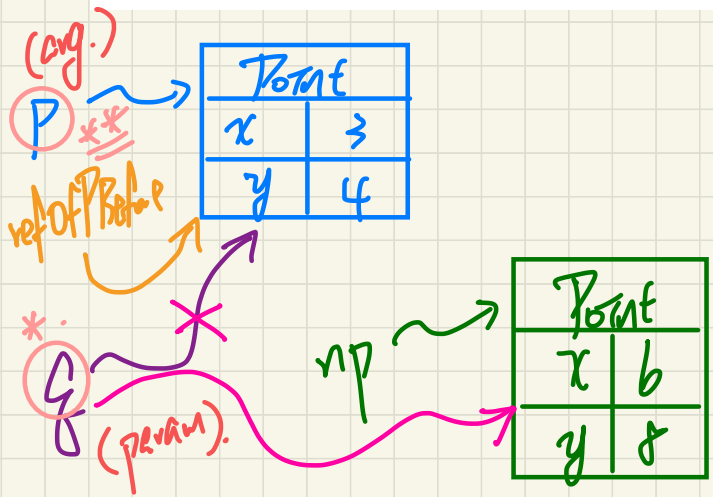
# Call by Value: Re-Assigning Reference Parameter

* param. (q) is re-assigned to np.

** arg (p) stays the same.

```
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); } }
```

```
@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);
}
```

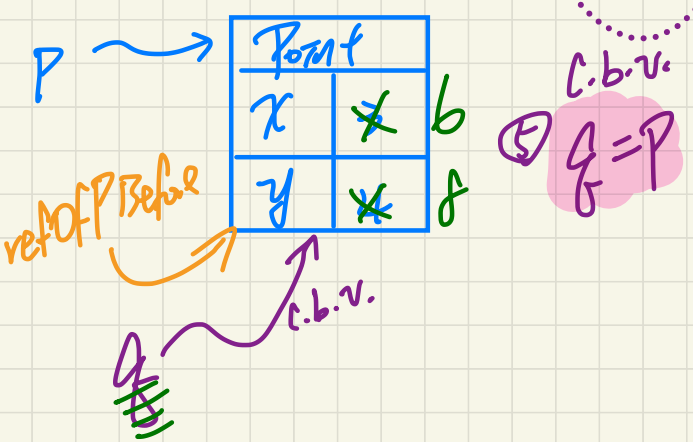



```
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x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

# Call by Value: Calling Mutator on Reference Parameter

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

```java
@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; }
}
```

# Equality

- `==`
- equals

## Primitive

1. `==` (value)
2. equals ✗

```
int i = 23;
int j = 46;
```

invalid C.O.

`i.equals(j)` ✗

↳ compilation error

false

`i == j`

## Reference

1. `==` (address/ref)
2. equals
   ↳ default version? customized version?

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

false ∵ diff. addresses

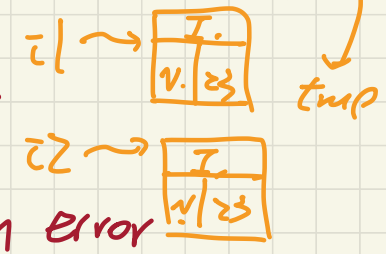
`p1 == p2`

`p1.equals(p2)`

↳ compares. ↳ result depends on how the Point class defines equals.

```
Integer i1 = new I.(23);
Integer i2 = new I.(23);
i1 == i2
i1.equals(i2)
```

false ∵ diff. refs.



# Object Equality: First Example

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

```
class PointTester {
  ... main(...) {
    PointV1 p1 = new PointV1(3, 4);
    PointV1 p2 = new PointV1(3, 4);
    System.out.println(p1.equals(p2));
  }
}
```

boil down into p1 == p2

this

Q1:

Does PointTester compile, given that **equals** is **not** declared in PointV1?

YES.

Q2:

Where does **equals** come from?

↳ Object

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

p1 p2 p2

1. not explicitly defined in PointV1
2. default version from Object is called.*

# The equals Method: To Override or Not?

all methods inherited to

parent/ super class

child/ sub class

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

3. overridden/redefined version is used.

extends

extends

1. explicit def. of equals method

2. default version of equals no longer applicable

1. no explicit def. of equals method

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

2. default version of equals from Object is used

```
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;
   }
}
```

redefined overridden version

# The equals Method: Default Version

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

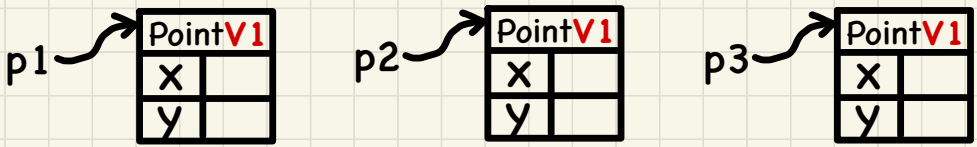
default version inherited to all classes.

extends

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

Since PointV1 does not explicitly override/redefine equals → default version inherited from Object is used.

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



# Lecture 13 - Oct 23

## Object Equality

***Call by Value: Array Indexing***
***Deciding the Version of equals Method***
***Short-Circuit Evaluation***

## Announcements/Reminders

- Today's class: notes template posted
- WrittenTest1 next Wednesday (October 29)
  + Some practice questions on eClass
  + **Past review session** materials released
  + Lecture 12 and Lecture 13 this week **are** covered.
  + Zoom Review Session: 4 PM on Saturday, October 25
    * We can focus on revisiting parts of the covered topics.
- Lab3 released

# Call by Value: Array Indexing

```
class Student {
  String[] courses; int noc;
  void register(String course) { ... }
}
```

```
RygSystem sys = new RygSystem();
Student s1 = new Student("alan");
Student s2 = new Student("mark");
sys.add(s1); sys.add(s2);
sys.register(s1, "EECS1090");
sys.register(s2, "MATH1025");
sys.registerAll("EECS2030");
```

RygSystem sys = new RygSystem()

c.b.v.

s = s1;

s = s2;

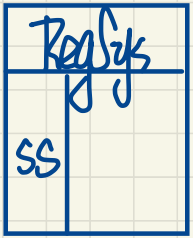
```
class RegSystem {
  Student[] ss; int nos;
  void register(Student s, String course) {
    s.register(course);
  }
  void registerAll(String course) {
    for(int i = 0; i < nos; i ++) {
      this.register(this.ss[i], course);
    }
  }
}
```

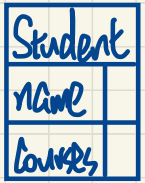
s = this.ss[i]

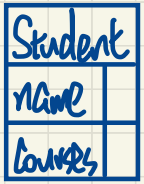
c.b.v.

course = "EECS2030"

c.b.v. happens for every iteration.



# Method Call: When Multiple Versions are Available

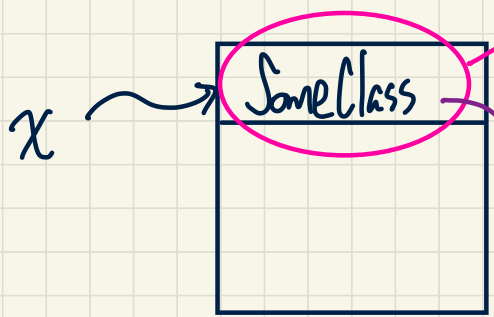
C.O. is all that matters to determine the version of equals method being called

`x.equals(y)`

which version?
1. default version (from object class)
2. overridden version (if exists).

**Principle:**

+ **Trace** the **dynamic type** of object "pointed to" by **reference variable x** at the point of executing this line of method call.
+ Is the **equals** method **overridden** in this **dynamic type**?
  * Yes -> Invoke the **overridden** version.
  * No -> Invoke the **default version** (inherited from **Object**).




dynamic type of x

Look up the def. of SomeClass:

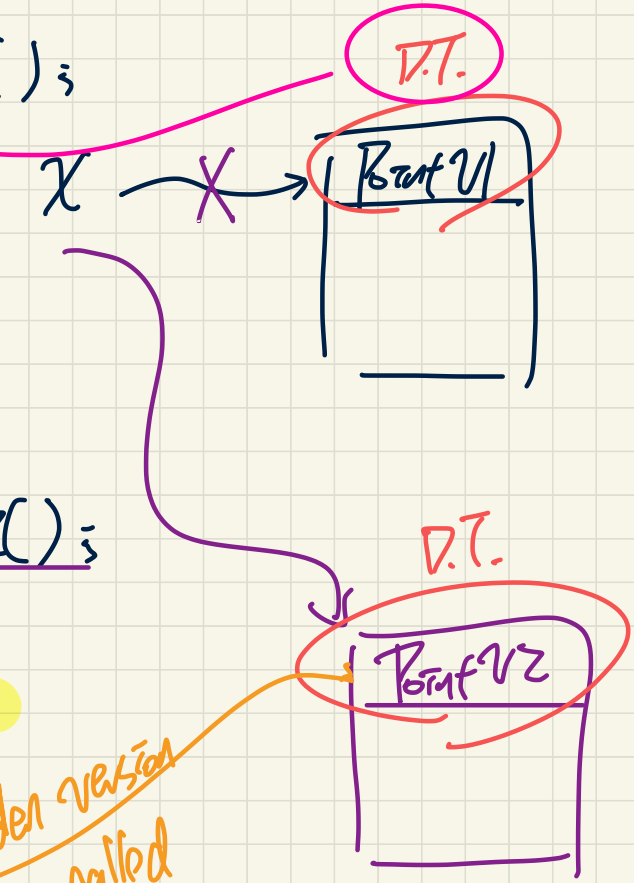
1. If SomeClass does not override equals, call the default version.

2. If SomeClass overrides equals, call this version.

```
x = new PointV1();

x.equals(...).

:

x = new PointV2();

x.equals(...).
```

default version called.

overridden version called



# The equals Method: Default Version

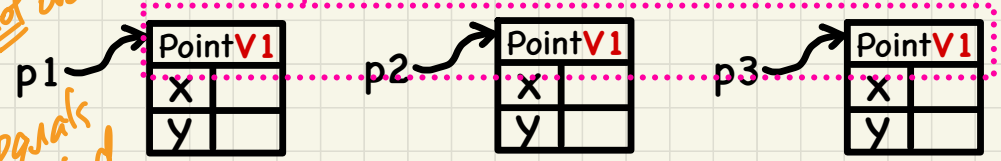
```
public class Object {
  ...
  public boolean equals(Object obj) {
    return this == obj;
  }
}
```

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

extends

1. all OO.s have the same D.T. PointV1
2. PointV1 has not overridden equals method
3. ∴ default version from Object invoked.

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




L7: p1 == p1 → T

L8: p1 == null → F.

L9: "p1 == s" → F.

PointV1 p1 = new PointV1 . . . ;

String s = " . . . " ;

declared types different

↳ forbidden to compare their addresses directly.

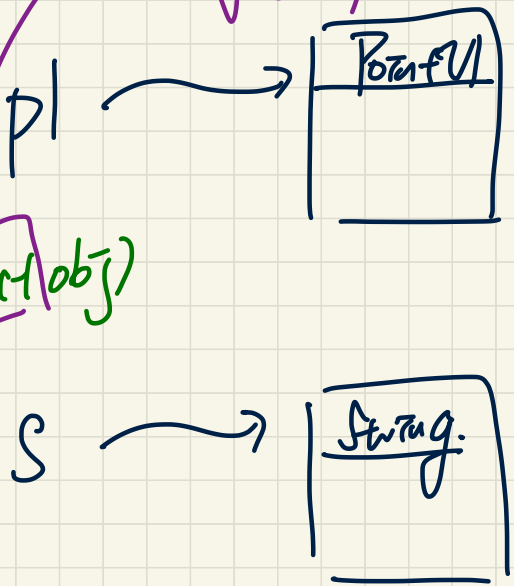
any type of argument is acceptable.

① p1.equals(s); → F

↳ "p1 == s"

→ Compiles ∵ boolean equals(Object obj)

② p1 == s → compilation error



# Call by Value: Invoking the Overridden equals

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

extends

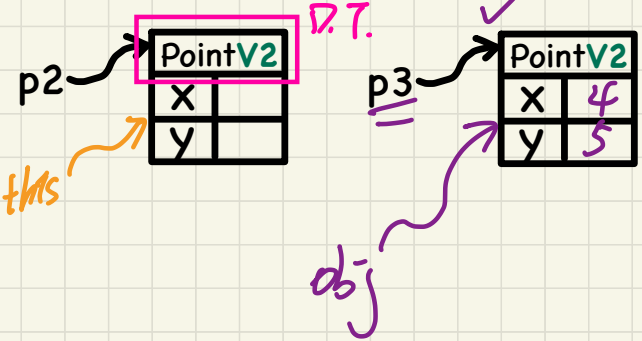
```
public class PointV2 {
private int x;
private int y;
public PointV2 (int x, int y) { ... }
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null) { return false; }
  if(this.getClass() != obj.getClass()) { return false }
  PointV2 other = (PointV2)obj;
  return this.x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

c.b.v.

obj = p3

```
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 */
```

D.T.



# Short-Circuit Evaluation: && and, conjunction

| Left Operand op1 | Right Operand op2 | op1 && op2 |
|---|---|---|
| 2.1 true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

the only way for && to be T is when both operands are true.

**Test Inputs:**

x = 0, y = 10

x = 5, y = 10

op1 && op2

```
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");
    }
}
```

Division By Zero!

if x == 0

F: 0 != 0 && 10/0 > 2 skipped

T: 5 != 0 && 10/5 > 2 ✓

1. Evaluate L to R

2.1. If op1 eval. to T
↳ eval op2 to decide the final result of &&

2.2. if op1 eval. to F
↳ skip eval. of op2 the final result of && is F

# Short-Circuit Evaluation: || or, disjunction

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

2.1

2.2.

the only way to make || F is when both operands are false.

**Test Inputs:**

x = 0, y = 10

x = 5, y = 10

op1 || op2

1. Evaluate L to R

2.1. If op1 eval. to F
↳ eval op2 to decide the final result of ||

2.2. If op1 eval. to T
↳ skip eval. of op2
the final result of || is T

```
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");
}
```

## Math

$p \wedge q$

$p \vee q$

Commutativity

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

## Java

$p \;\&\&\; q$ → $p \;\&\&\; q$

$q \;\&\&\; p$

not identical

↳ short-circuit evaluation

$p \;||\; q$

guarding condition (without passing it's skip 2nd operand)

some expression to protect.

$x != 0$ && ... $y/x$ ...

$x == 0$ || ... $y/x$ ...

logical negation of each other.

error condition (if it's true, skip 2nd operand).

a.length == 10

0 . . 9

a : array of values

Is this always guarded?

No.

T

10

$i \geq 0$ && $a[i] == v$ && $i \leq a.length - 1$

g1

g2

i == length of array (10)

10

a[10] → IOOBE!

Correct: g1 && g2 && ... a[i]

Exercise: use ||

# 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 */ }
```

# Lecture 14 - Oct 28

## Object Equality

***Overridden equals: Phases 1, 2, 3***
***Static vs. Dynamic Types, Type Casting***

## Announcements/Reminders

- Today's class: notes template posted
- ProgTest0 and ProgTest1 marks and feedback released
- WrittenTest1 this Wednesday (October 29)
  + Some practice questions on eClass
  + Past review session materials released
  + Lecture 12 and Lecture 13 this week are covered.
- Lab3 released

# The equals Method: Overridden Version

Phase 1

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

extends

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

if C.O. and param. obj are the same object → the grp equal

C.O. p1 equals (p2) arg. → T / → F

p1 C.O.

this

(param) obj

obj = p2 p2 (arg.)

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

# The equals Method: Overridden Version

Example 1: Trace L7

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

extends

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

obj = p1

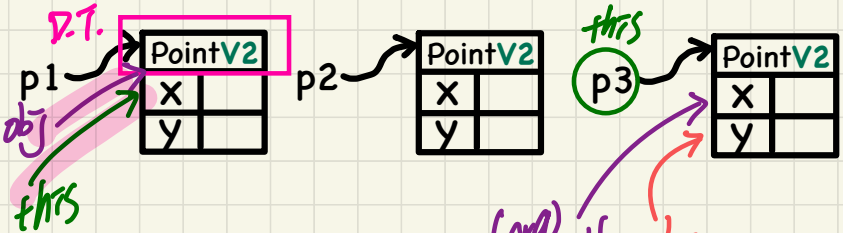
∵ p1 and p1 are the same object.

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

D.T.

c.o.

arg




e.g. p4 = p3;

(arg) p4 obj (param)

p3 equals (p4);

arg

obj = p4

m. true

∵ p3 and p4 same obj.

# The equals Method: Overridden Version

p1.equals(p2)

this == null → NPE.

Phase 2

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

this

obj

extends

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

reaching this tmp means: this != obj

F

unreachable :( if this were null, then a NPE would have already occurred.

* alt. phase 2

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

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

# The equals Method: Overridden Version

Example 1: Trace L8

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

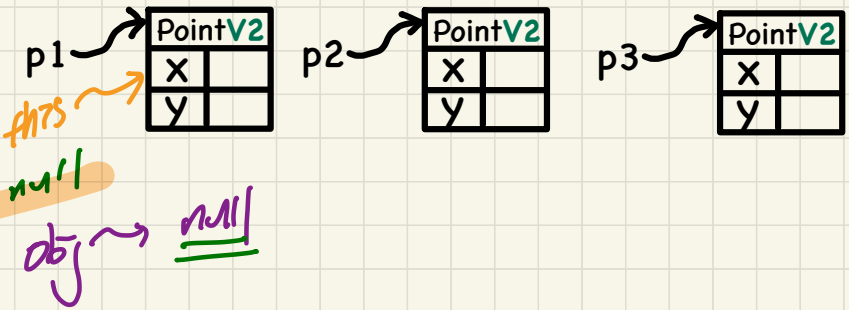
extends

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

E

rn. false : p1 is not null but obj is null

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



may be changed runtime type : determines the version of method called.

# Static Type, Dynamic Type, Type Casting

fixed

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

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

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

declared type

determines the range of methods & attributes that can be invoked on that ref. var.

| var | ST | exp. |
|---|---|---|
| o1 | C1 | m1, equals |
| o2 | C2 | m2, equals |
| o3 | Object | equals |

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

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

ST: C1

ST: C2

```
C1 o1 = new C1();
C2 o2 = new C2();
o1.m1();
o1.m2();   X
o2.m1();   X
o2.m2();
Object o3;
o3 = o1;
o3.m1;   X
o3.m2;   X
((C1) o3).m1();   ✓
((C1) o3).m2();   X
o3 = o2;
((C2) o3).m1();   X
((C2) o3).m2();   ✓
```

R.T.

create a new copy of address with a different S.T.

∵ m2 is not an exp. of o1's ST (C1)

R.T. of o3 is C1

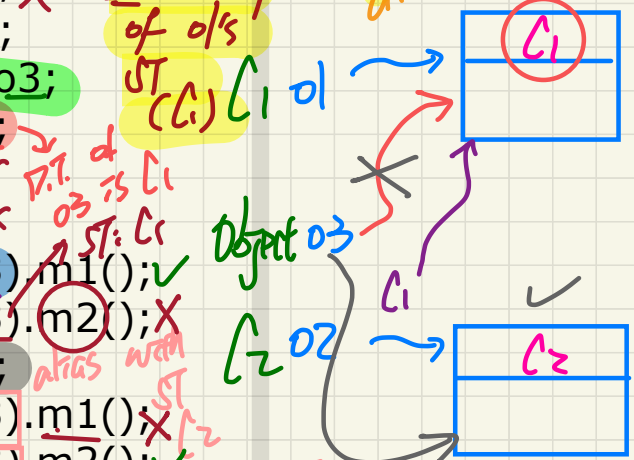
ST: C1

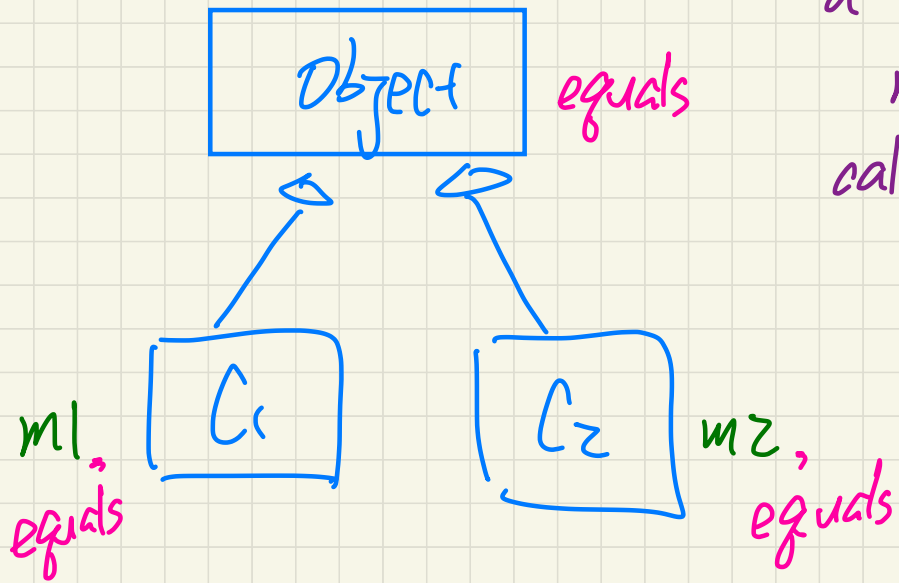
alias with ST C2

alias with ST C2

not compile

∵ ST of o3 is Object, whose expectation does not include m1.






Expectation of a type/class:
range of meth./attrs callable on that type

Given a line of method call:

obj.m(...)

(1) Does it compile?

↳ look at the $\boxed{\text{S.T.}}$ of C.O. obj.

(2) If it compiles, which version of m is called?

↳ look at the $\boxed{\text{D.T}}$ of C.O. obj.

parent of every class

can be any class

```
Object obj = new SomeClass();

Object obj = obj2;
```

declared type can be any class.

force the cast to happen first.

$((C_1)\ \boxed{o3}).m1$

"casting o3 to C1"

"creating an alias of o3 with ST. C1"

$(C_1)\ o3.m1$

✗ not compile

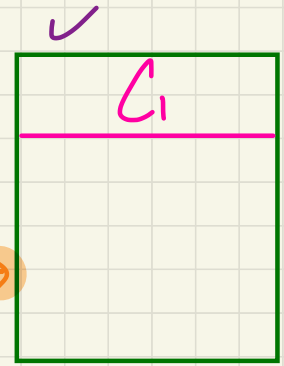
Object o3 = ...

Object o3 = ... not going to change

$\boxed{(C_1)\ o3}$ = when o3's pointing to.

exp: equals

$\boxed{\text{Object}}$ o3




o3's ST: object

$((C_1)\ o3).m1$

o3's ST: object

exp: m1, equals.

$(C_1)\ o3$

alias with ST. C

# The equals Method: Overridden Version

Phase 3

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

extends

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

this != obj

obj != null

return the D.T. of the context objects

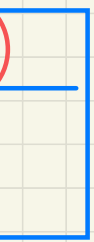
objects with different D.T.s are not equal to each other.

this.getClass()

this

obj.getClass()

obj



# The equals Method: Overridden Version

Example 1: Trace L9

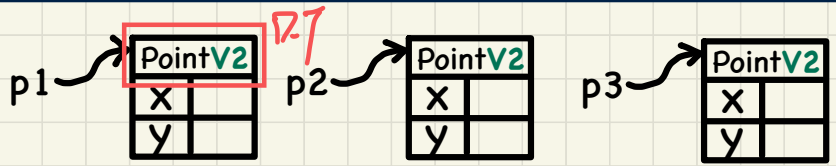
```java
public class Object {
   ...
   public boolean equals(Object obj) {
      return this == obj;
   }
}
```

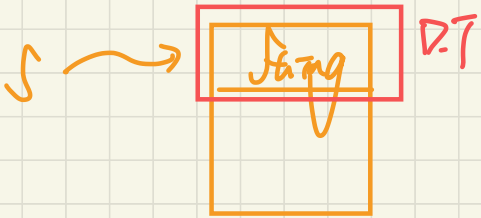
extends

```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
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);   /* [redacted] */
System.out.println(p2 == p3);   /* [redacted] */
System.out.println(p1.equals(p1));   /* [redacted] */
System.out.println(p1.equals(null));   /* [redacted] */
System.out.println(p1.equals(s));   /* [redacted] */
System.out.println(p1.equals(p2));   /* [redacted] */
System.out.println(p2.equals(p3));   /* [redacted] */
```

rt. false.



# Lecture 15 - Oct 30

## Object Equality

***Overridden equals: Phase 4***
***Short-Circuit, Logical Implication***
***JUnit: assertSame vs assertEquals***

## Announcements/Reminders

- Today's class: notes template posted
- ProgTest0 and ProgTest1 marks and feedback released
- Lab3 released

# The equals Method: Overridden Version

Phase 4

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

extends

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

* PointV2 other = (PointV2) obj;

d.t. of C.O. this

static type of obj.

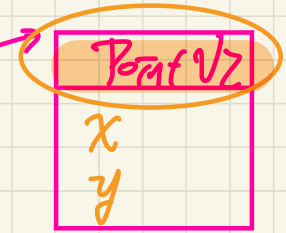
this != obj

obj != null

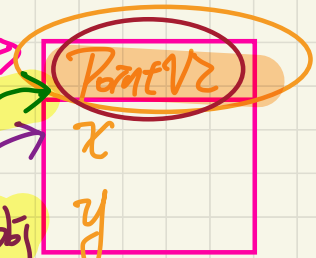
this.getClass() == obj.getClass()

Compile : ST of other is PointV2.

this → C.O.




PointV2 other obj




(PointV2) obj

alt. case exp. of ST. PointV2

this.x == ~~obj.x~~ x ∵ ST of obj is object.

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

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

# The equals Method: Overridden Version

Example 1: Trace L10

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

extends

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

bypassed

Object obj

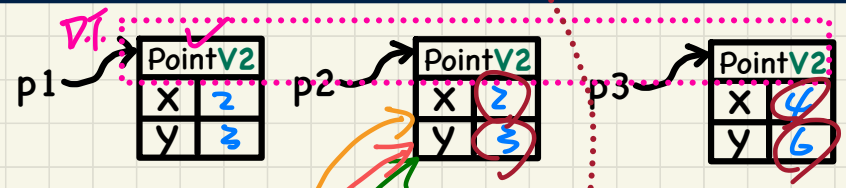
PointV2 other

(PointV2) obj

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

true.

D.T.




return false

Exercise

# The equals Method: Overridden Version

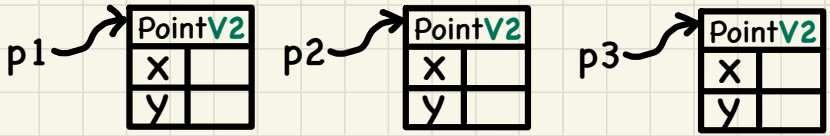
Example 1: Trace L11

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

extends

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

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



# The equals Method:

# To Override or Not?

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

extends

extends

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

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

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

D.T.

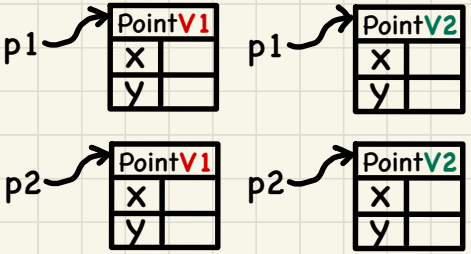
default version from Object is called

↳ p1 == p2

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

D.T.

overridden version from PointV2 is called



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

① ② risk of NPE

a ☐ ~~if (① && ②) { return false; }~~

b ☐ ~~if (② && ①) { rn. false; }~~

c ☑ if (① || ②) { rn. false; }

↳ if true, skip eval. ②

d ☐ if (② || ①) { rn. false; }

if obj is null NPE would've occurred before evaluating ①.

# The equals Method: Overridden Version

## Example 2

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

extends

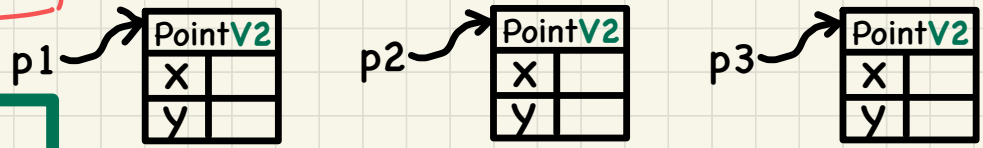
```java
public class PointV2 {
  private int x;
  private int y;
  public PointV2 (int x, int y) { ... }
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false }
    PointV2 other = (PointV2)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

obj1

obj2

```java
1  PointV2 p1 = new PointV2(3, 4);
2  PointV2 p2 = new PointV2(3, 4);
3  PointV2 p3 = new PointV2(4, 5);
4  System.out.println(p1 == p1);   /* [redacted] */
5  System.out.println(p1.equals(p1));   /* [redacted] */
6  System.out.println(p1 == p2);   /* [redacted] */   F
7  System.out.println(p1.equals(p2));   /* [redacted] */   T
8  System.out.println(p2 == p3);   /* [redacted] */
9  System.out.println(p2.equals(p3));   /* [redacted] */
```

(A) (B)




obj1 == obj2

(A) Two objects are reference-equal.

(B) Two objects are contents-equal.

obj1.eq.(obj2)

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

not valid.

# Logical Implication

$p \boxed{\rightarrow} q$

$p \boxed{\Rightarrow} q$

false when:
p is true
q is false.

it rains $\rightarrow$ outdoor ground wet

it rains T
outdoor ground wet F.

$\rightarrow$ implication is false.

# assertSame vs. assertEquals

any objects

pass if: exp1 == exp2
fail if: exp1 != exp2

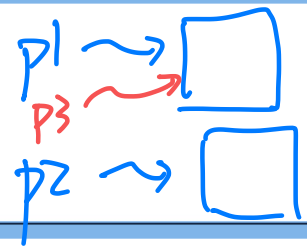
***assertSame***(exp1, exp2)

- 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);
```

p1 == p3 → pass

p2 == p3 → fail




***assertEquals***(exp1, exp2)

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

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

i == j → fail

ref types, any type

assertEquals( exp1 , exp2 )

↳ assertTrue( exp1.equals(exp2) )

① assertEquals( exp1 , exp2 )

↳ assertTrue( exp1.equals( exp2 ) )

version in exp1's D.T.

② assertEquals( exp2 , exp1 )

↳ assertTrue( exp2.equals( exp1 ) )

version in exp2's D.T.

# assertEquals: Reference Comparison or Not

* assertT(p1.equals(p2))
** assertT(p2.equals(p3))

**assertEquals**(exp1, exp2)

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

p1 == p2

"p2 == p3"

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

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

false

false


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

** phase 3

both rn. false but for diff reason

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

false
false

p1.eq.(p2)
p2.eq.(p3)

* p3.eq.(p2)
↳ overridden ver.
↳ rn. false '∵ diff. D.T.' **

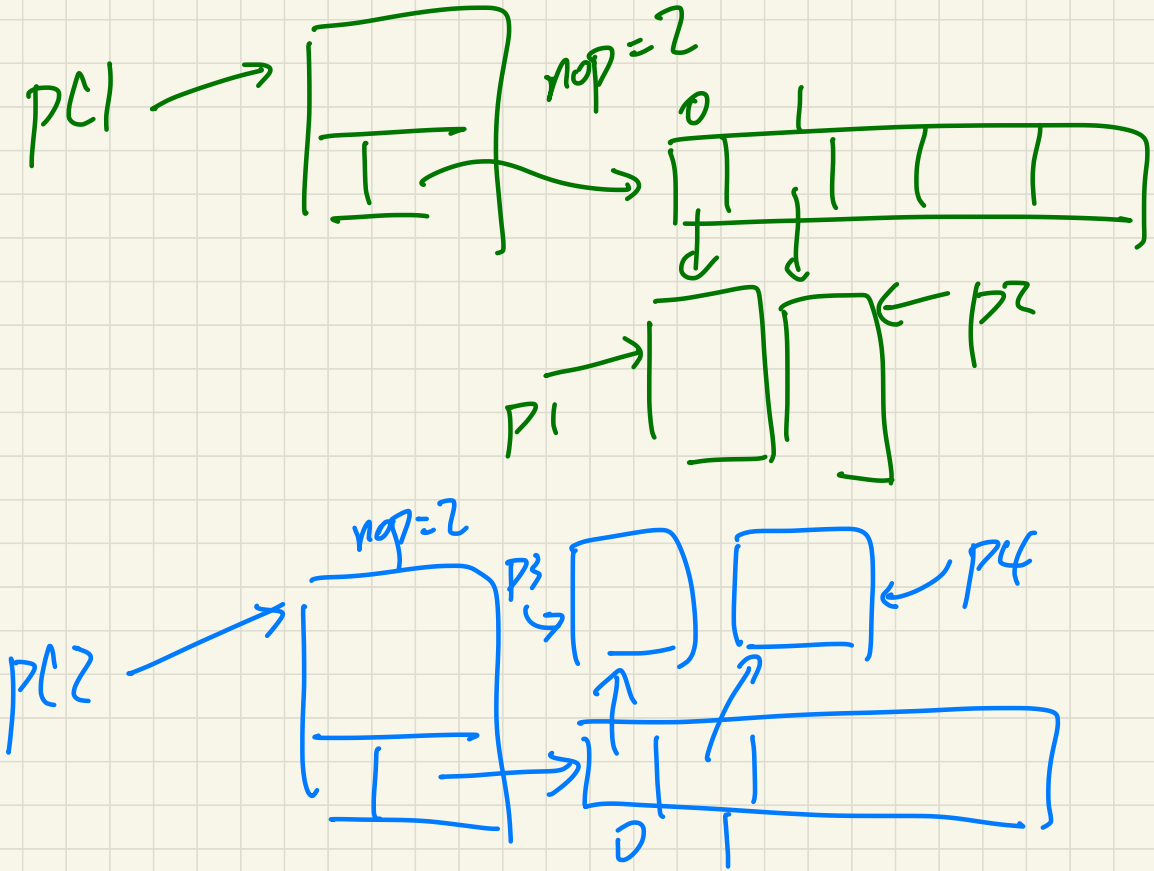
# Exercise: PersonCollectors are equal if their arrays of persons are equal

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

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

```java
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 }
```

```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);
  }
}
```


pC1
nop=2
0
1
p1
p2
nop=2
p3
p4
pC2
0
1

# Lecture 16 - Nov 4

## Inheritance

***Design Principles: Cohesion, SCP***
***SMS: Design Attempt 1***
***SMS: Design Attempt 2***
***Implementing Inheritance in Java***

## Announcements/Reminders

- Today's class: notes template posted
- WrittenTest1 marks and feedback released
- ProgTest2 this Wednesday (November 5)
  + Guide released
  + PracticeTest2 released
  + Lab3 solution released
- Tracing Exercises: assertEquals and Person, PersonCollector

# Inheritance: Motivating Problem

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

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

ls cd pwd

# Design Principles

1. Cohesion

Unix principle
↳ each command should do one things and do it well

↳ Attributes and methods in a single class should serve the same purpose.

be under a unifying theme

2. Single Choice Principle

violation: duplicates.

↳ A change to the software should impact a minimum number of places

not necessarily 1, but no more than necessary.

methods, classes.

# First Design Attempt

how to simulate OO in a non-OO language (e.g. C).

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

32

==1 RS
==2 NRS

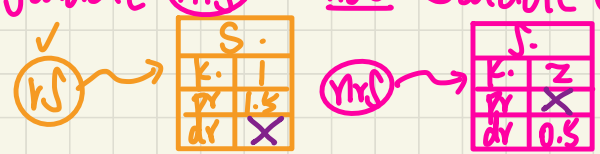
RS

NRS

1 2

rs. getTuition()
nrs. getTuition()

Student rs = new Student(1);
Student nrs = new Student(2);




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

base amount

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

mutually exclusive

should be put in separate 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 Cohesion

# 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;
   }
   ...
}
```

```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 (kind == 3) { ... }
```

dup1

dup2

```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 ++;
   }
}
```

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

## Good design?

Judge by Single Choice Principle

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

kind == 3 ↳ international

# Testing Student Classes (without inheritance)

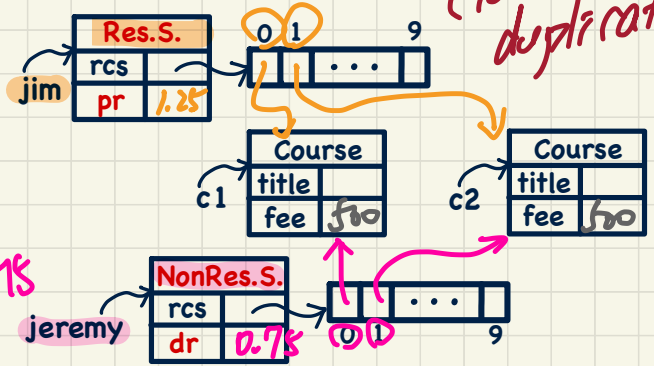
```java
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a m
  public ResidentStudent (String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc]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

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

cohesion (pr and dr put in different classes)

SCP violated (lots of duplicates)

```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()); 1000 * 1.25
    System.out.println("Jeremy pays " + jeremy.getTuition()); 1000 * 0.75
  }
}
```



# 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 ;
  }
}
```

```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 ;
  }
}
```

every duplicate has to be changed!

## Maintenance e.g., a new registration constraint:

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

how many methods should be impacted?

# 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 ;
  }
}
```

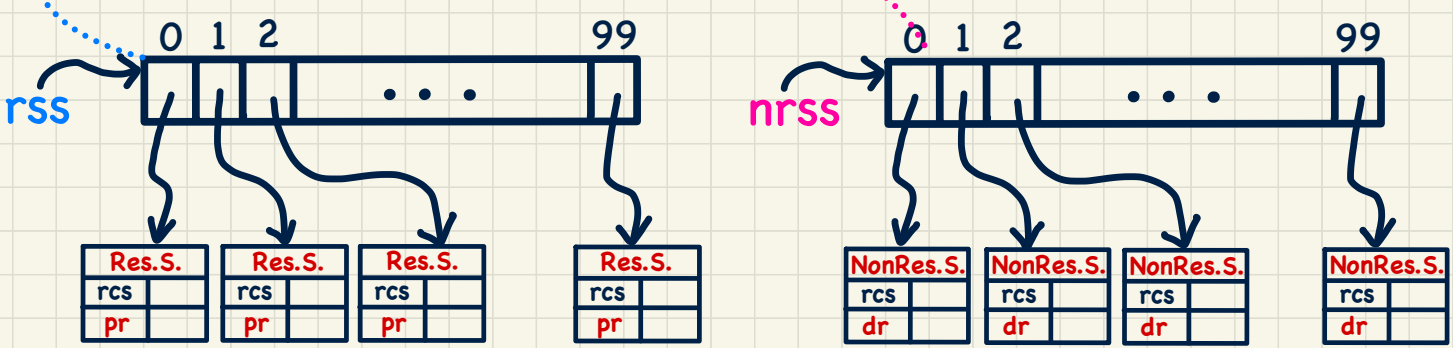
## Maintenance e.g., a new tuition formula:

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

# 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); }
  }
}
```

# arrays = # kinds of students



# Recall: Student Classes (with inheritance)

child/sub-class extends parent/super class

RS NRS → Student.

super vs. this

parent class — current class/object obj

```
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 */
  }
}
```

"super" relative to RS and NRS

*

parent version

Common super/parent class

Call non-private attributes/methods are inherited to child/sub classes)

↳ sat. SCP.

base amt. calculation

child/sub class specific attributes

↳ cohesion is sat.

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

* student

RS overridden version

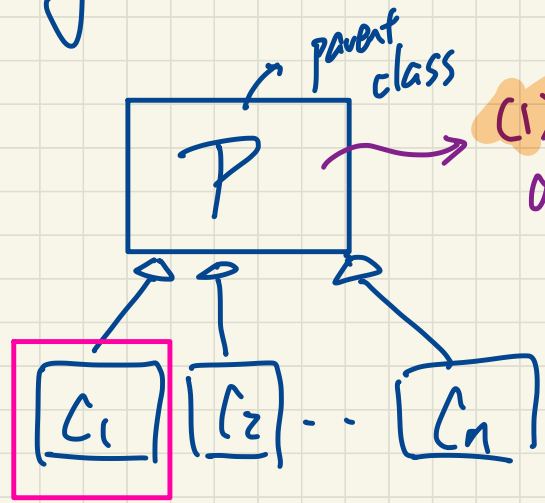
recursing the parent version of the method.

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

* student

NRS overridden version

When programming with inheritance (OOP):




(1) declares att./methods common to all subclasses (SCP).

(2a) declare additional atts/methods applicable to this subclass only.

(2b) override/redefine the inherited methods

# Lecture 17 - Nov 6

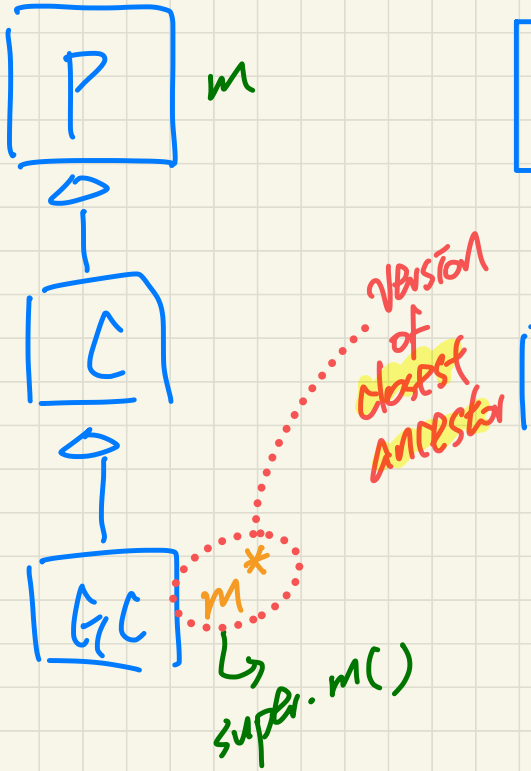
## Inheritance

***Modifiers in Java***
***Static Types and Expectations***
***Polymorphism, Dynamic Binding***

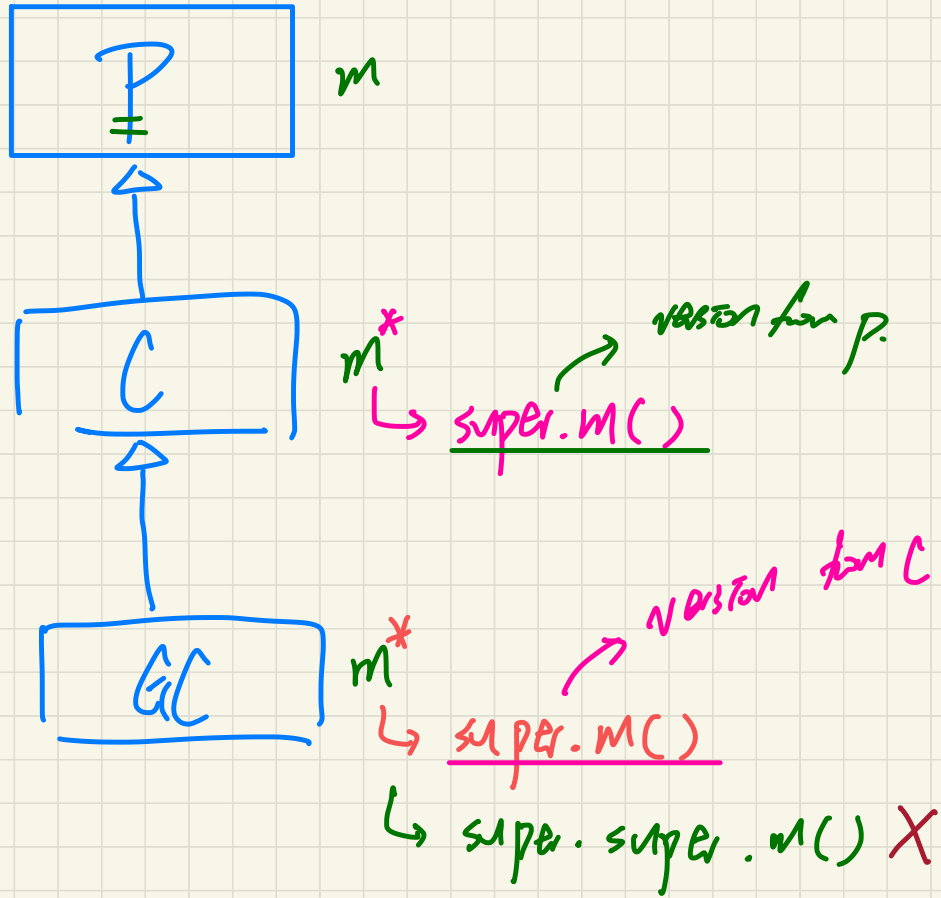
## Announcements/Reminders

- Today's class: notes template posted
- Lab4 ~~solution~~ released
- Tracing Exercises: assertEquals and Person, PersonCollector

(scenario)



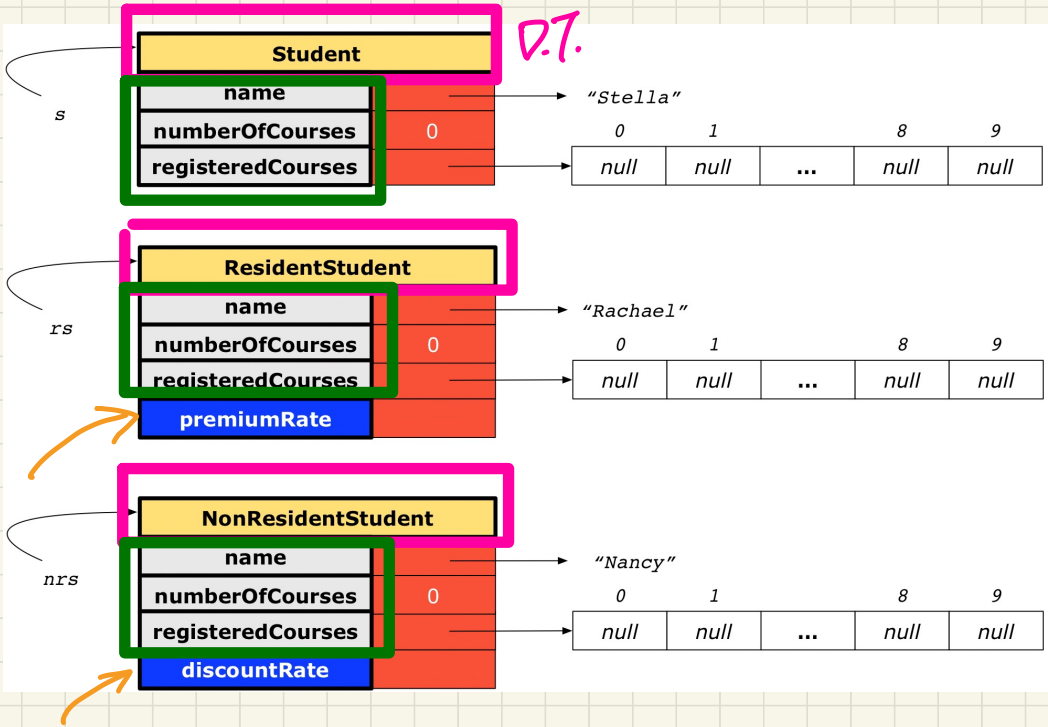

(scenario 1)

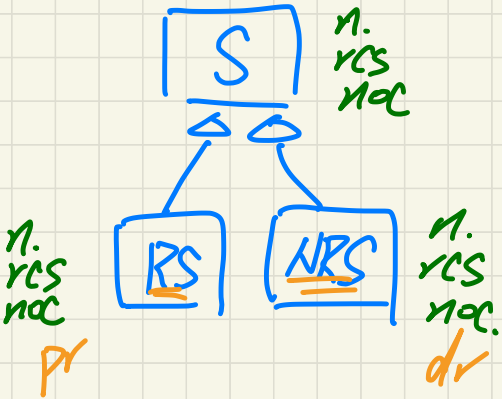


# Visualizing Parent and Child Objects

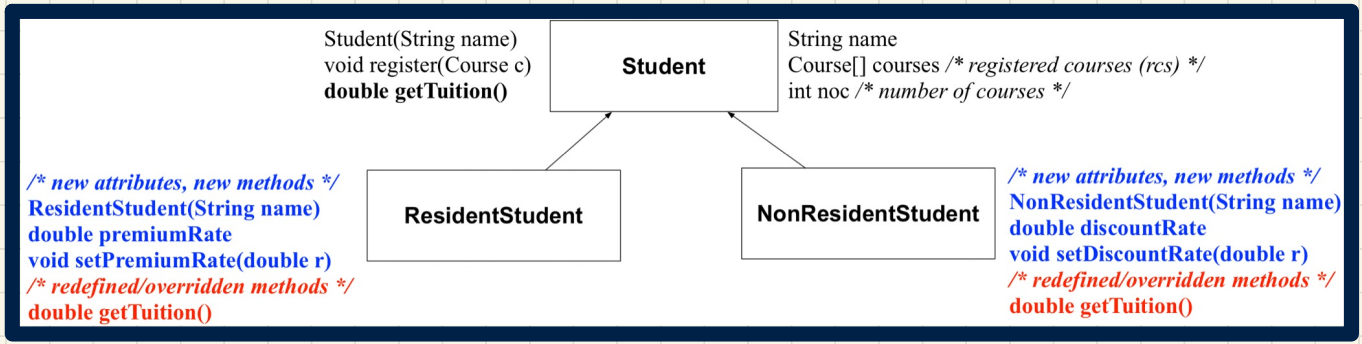
dynamic types

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



# Testing Student Classes (with inheritance)



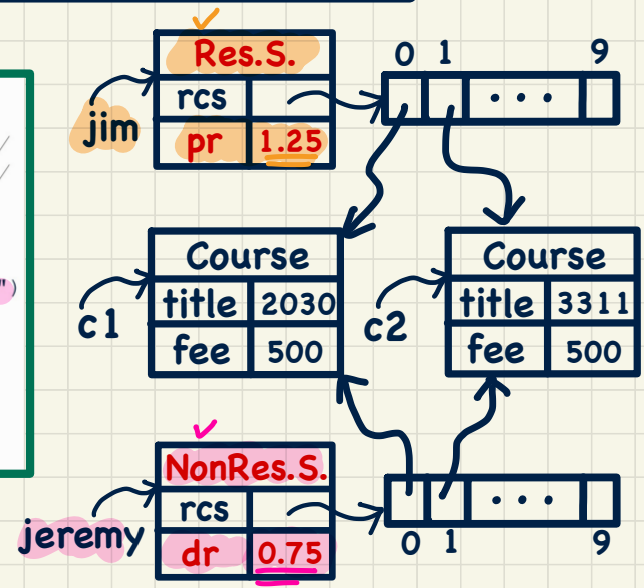

```java
public class StudentTester {
  public static void main(String[] args) {
    Course c1 = new Course("EECS2030", 500.00); /* title and fee */
    Course c2 = new Course("EECS3311", 500.00); /* title and fee */
    ResidentStudent jim = new ResidentStudent("J. Davis");
    jim.setPremiumRate(1.25);
    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());
  }
}
```

D.T.: RS

version from RS (pr)

D.T.: NRS

version from NRS (dr)



# Modifiers in Java

visibility.

① private

② no modifier

③ protected

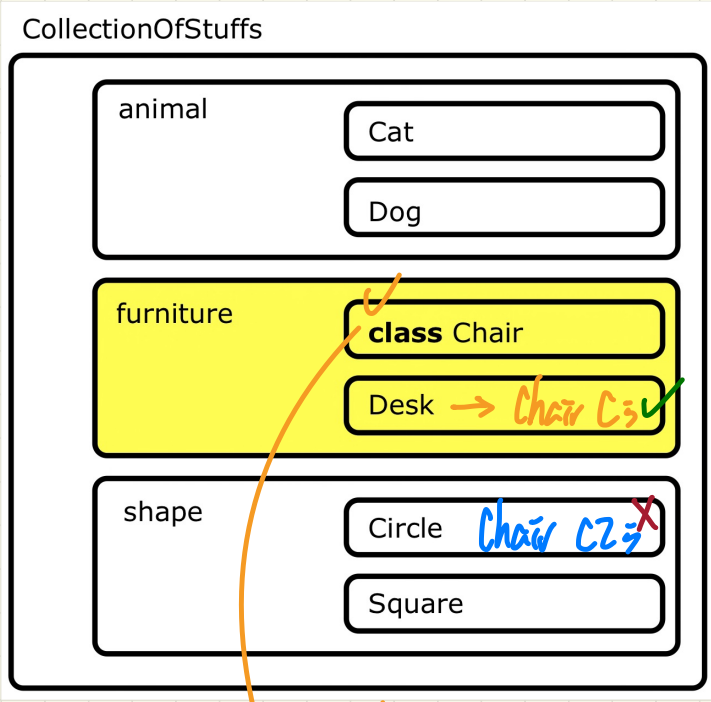
④ public

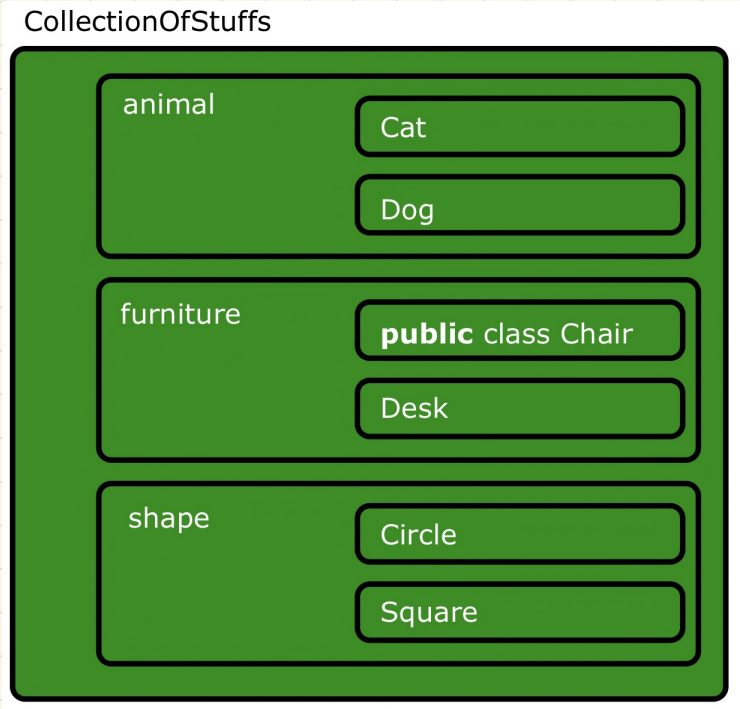
attributes/methods.

inheritance (e.g. Lab4)

classes.

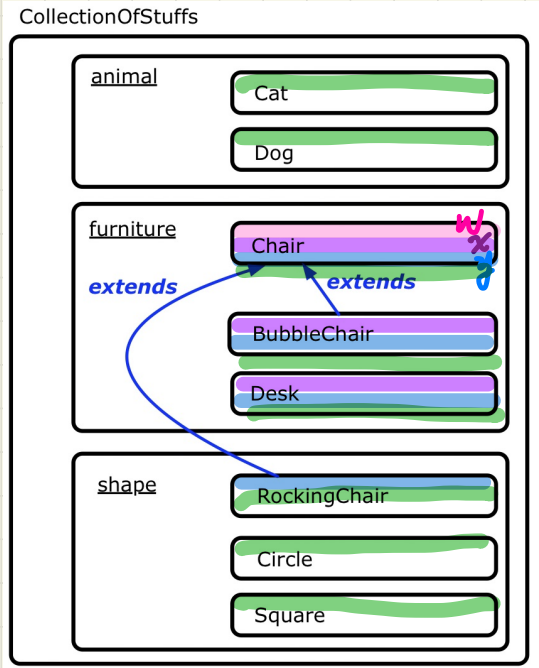
# Visibility: Classes




no modifier
↳ visible within the residing package

# Visibility: Attributes and Methods




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

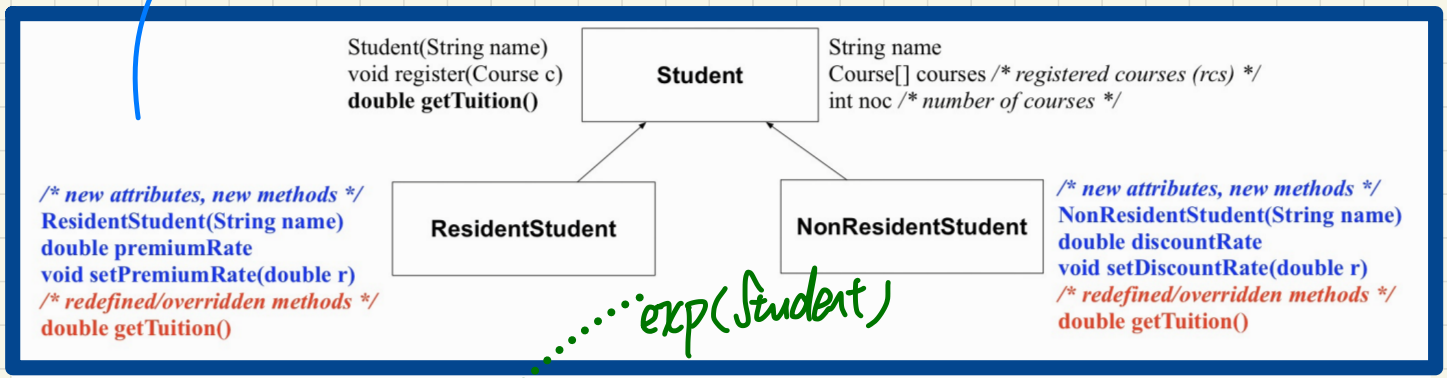
1. package level
2. subclasses (not in the same package)

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

assume: all classes in same package.

# Student Classes (with inheritance): Expectations

determined by static types




exp(Student)

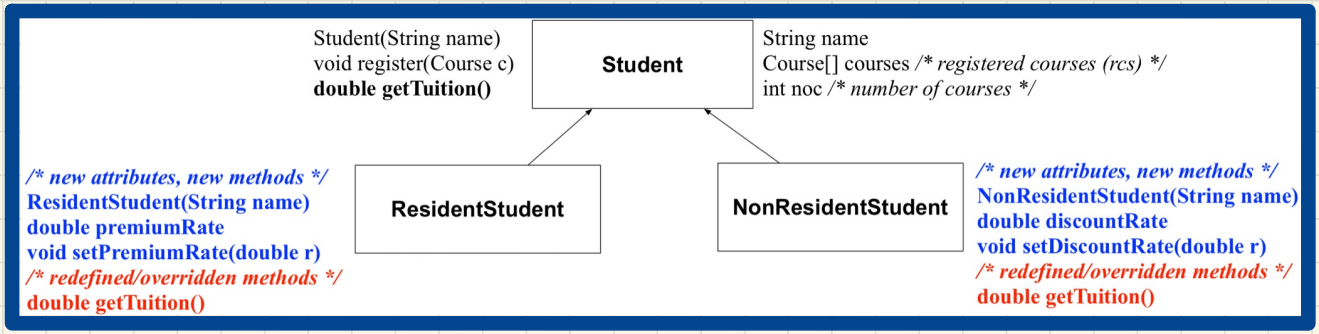
```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

additional exp. for child classes

| | name | rcs | noc | reg | getT | pr | setPR | dr | setDR |
|---|---|---|---|---|---|---|---|---|---|
| ST: S s. | | | | | | | | | |
| ST: RS rs. | | | | | | | | | |
| ST: NRS nrs. | | | | | | | | | |

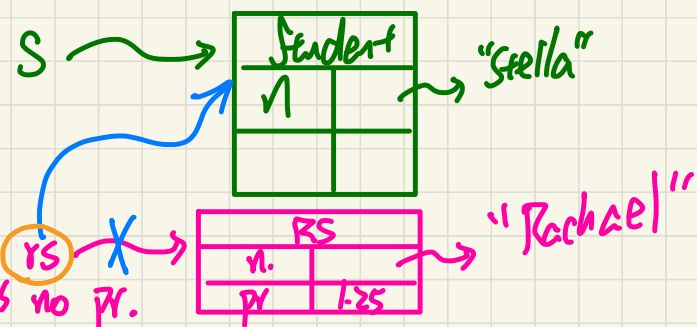
# Intuition: Polymorphism

4. assumption is wrong ∵ rs = s should not compile.

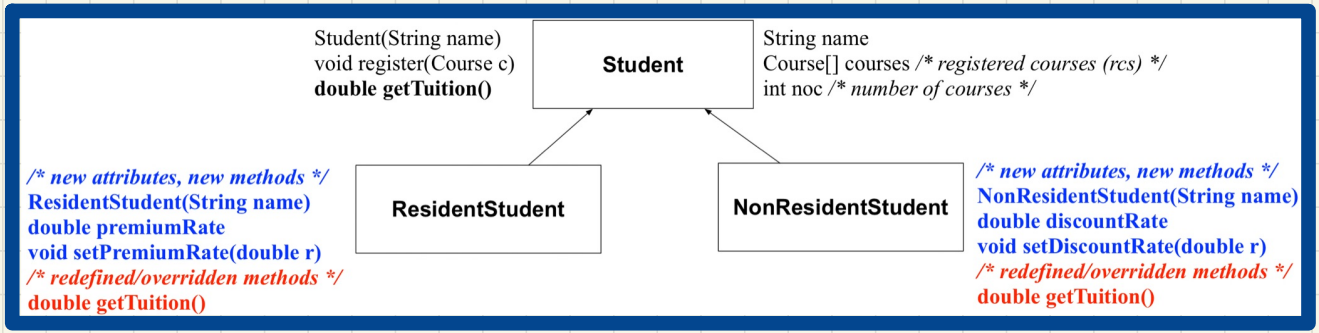



```
1  Student s = new Student("Stella");
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.25);
4  s = rs; /* Is this valid? */
5  rs = s; /* Is this valid? */
```

1. Assume rs = s compiles.
2. Execute the var assignment.
3. Exp of rs' ST (RS) includes: pr
   rs.pr → crash ∵ Student has no pr.



# Intuition: Dynamic Binding




var s

| step | ST | DT |
|---|---|---|
| 1 | Student | null |
| 2 | | RS |
| 3 | | RS |
| 4 | | NRS |

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

① ② ③ ④

ST: Student

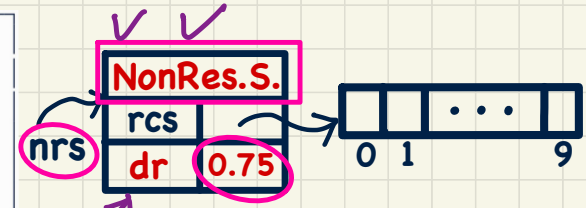
ST: RS and NRS.

DT: RS apply pr

125

DT: NRS apply dr

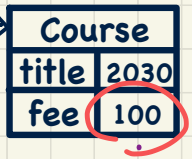
75



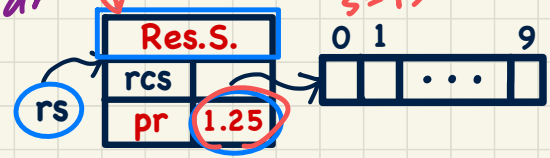

s=nrs

Student s → null

eecs2030




s=rs



Compilation : static type

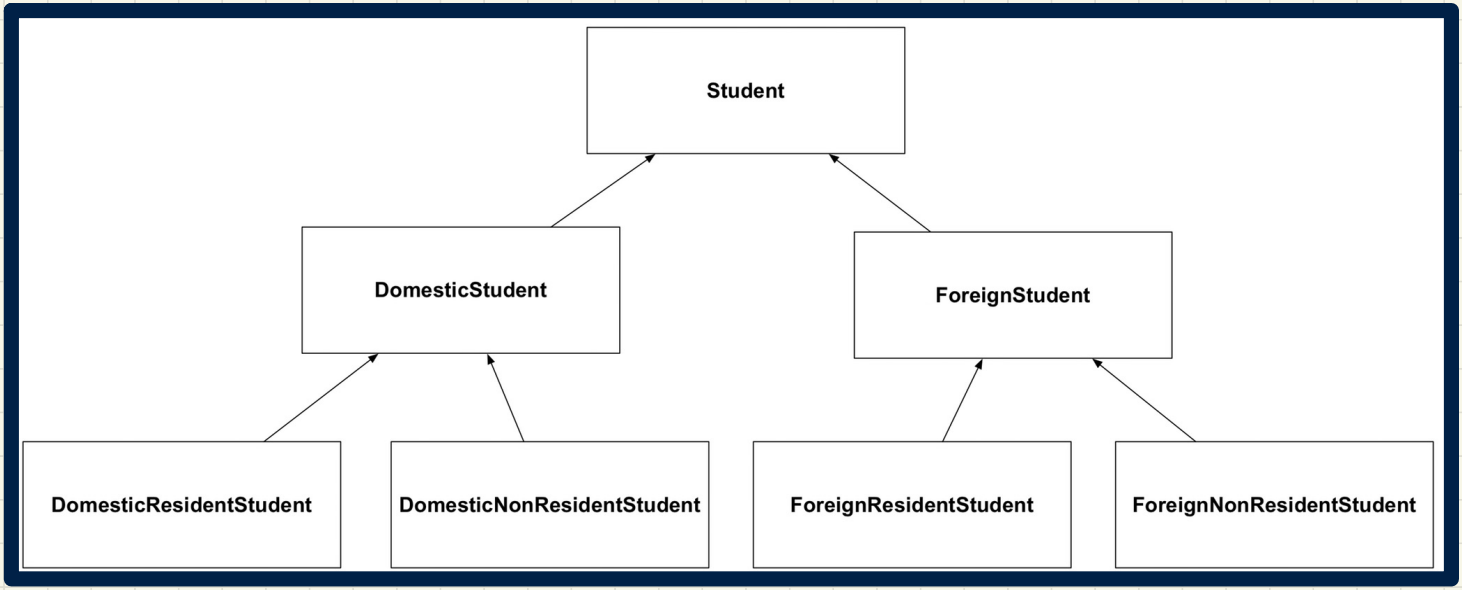
<u>runtime</u> : dynamic type

e.g. version of method called

e.g. exception

↓ ClassCastException.

# Multi-Level Inheritance Hierarchy: Students




**Reflections**:

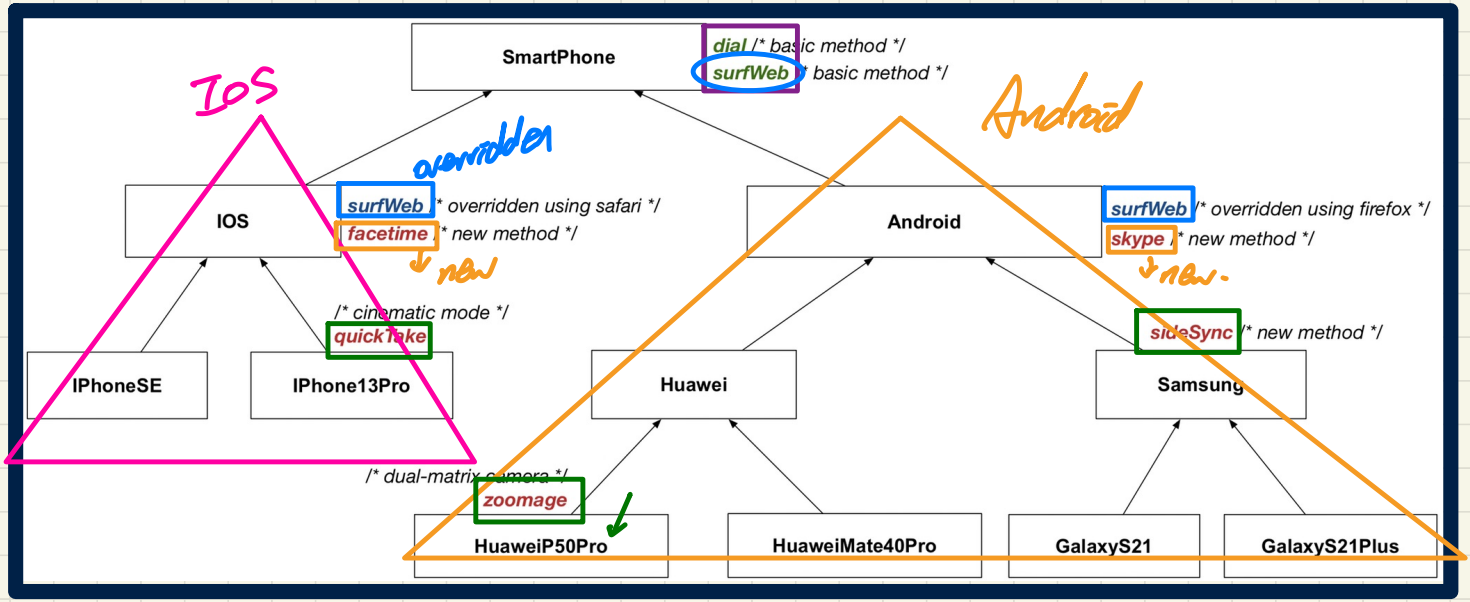
read var?

how many classes

- 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?

→ extends?

# 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?

# Lecture 18 - Nov 11

## Inheritance

***Ancestors, Descendants, Expectations***
***Rules of Substitutions***

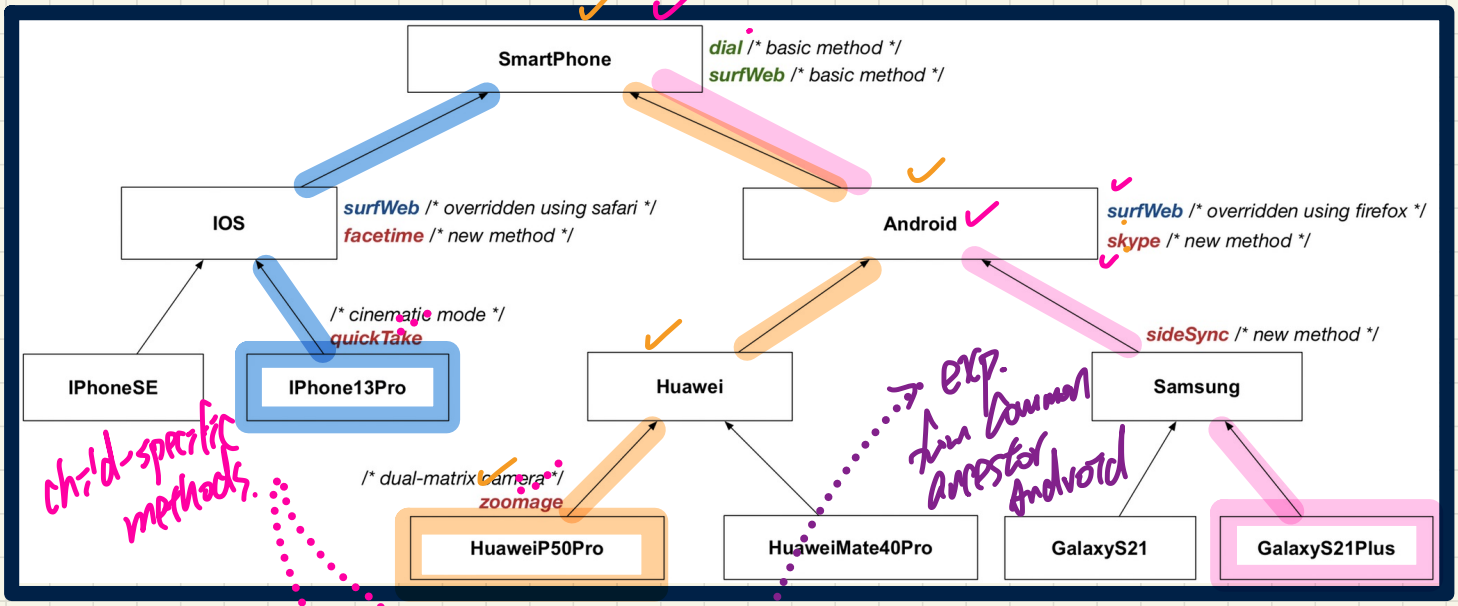
WSC105 -

## Announcements/Reminders

- Today's class: notes template posted
- Lab4 released
- Tracing Exercises: assertEquals and Person, PersonCollector

# Multi-Level Inheritance Hierarchy: Smartphones

expectations ≈ callable method.




child-specific methods.

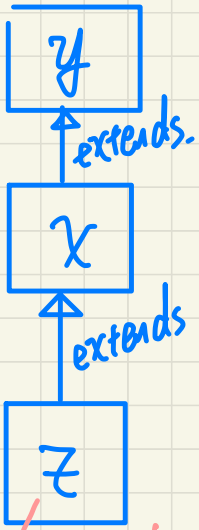
exp. from common ancestor Android

**Exercise** Compare the ranges of expectations of:

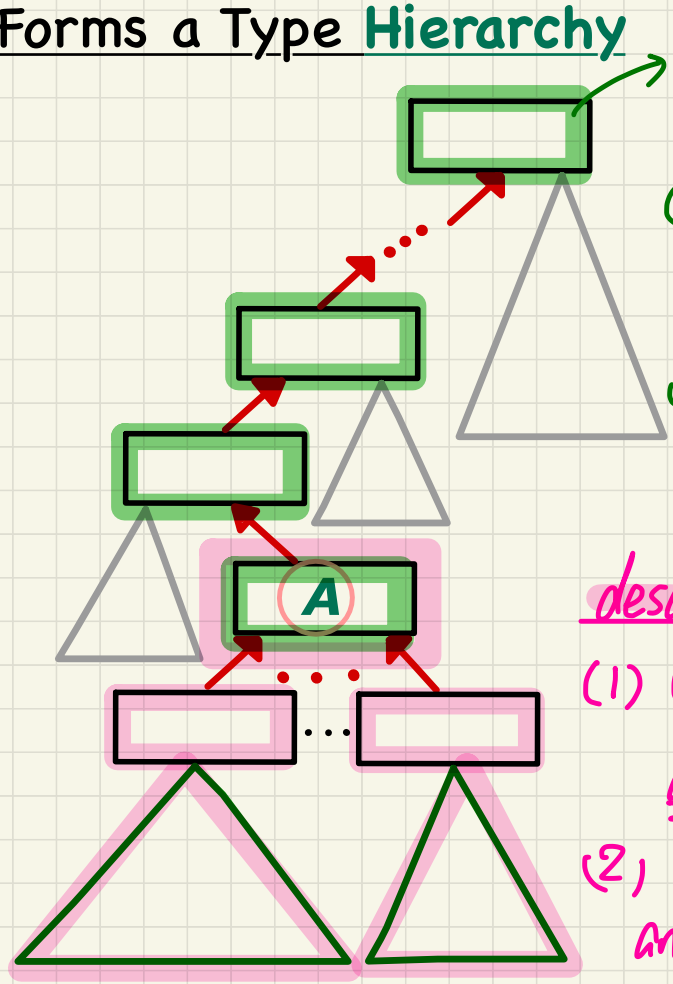
+ IPhone13Pro — qT, facetime, surfWeb, dial
+ HuaweiP50Pro — zoomage, skype, surfWeb, dial
+ GalaxyS21Plus — sideSync, skype, surfWeb, dial

common exp inherited from SmartPhone

# Inheritance Forms a Type Hierarchy




z indirectly extends y.




**ancestors of A**

(1) A accumulates all atts/mths. from its ancestors

expectation

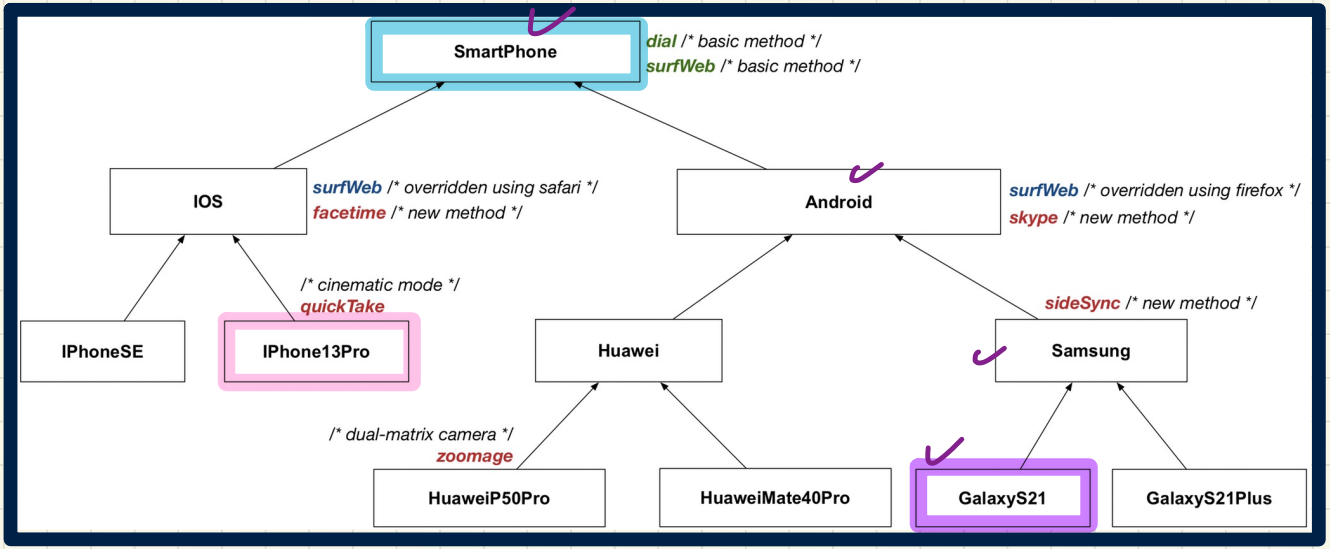
(2) exp(A) ≥ exp of any ancestor class

**descendants of A**

(1) exp(A) inherited to all of A's descendants.

(2) exp(A) ≤ exp of any decendant class.

# Inheritance Accumulates Code for Reuse




| | ancestors | expectations | descendants |
|---|---|---|---|
| | GS21 → S. → A. → SP | | GS21 |
| | | Exercise | |
| | SP. | | all classes |

assumption: there's some inheritance hierarchy existing

$\boxed{var1} = \boxed{var2}$

ST: $T_1$ ST: $T_2$

does this var. assignment compile?

substitution

Rule of substitutions (inheritance).

$T_2$ must be a descendant of $T_1$

$exp(T_2) \geq exp(T_1)$

# Rules of Substitutions

does a var. assignment compile? (static types; inheritance)

ST of oa

ST of ob

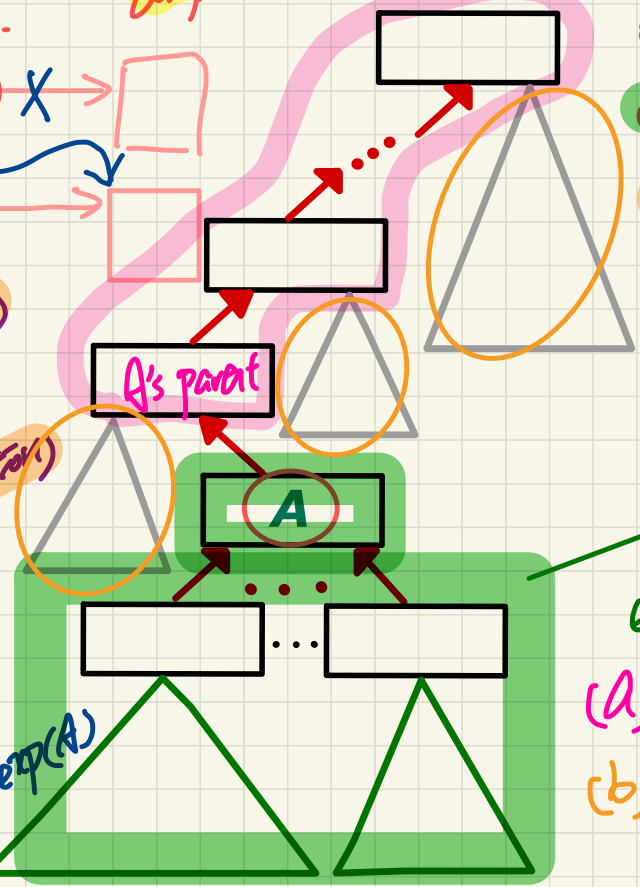
```
A oa = ...;
? ob = ...;
oa = ob;
```

ST: A.

oa → X

ob →

what can ? (ST of ob) be in order for this var. assignment (substitution) to compile?




descendants of A

exp ≥ exp(A).

(a) ancestors of A's parent

(b) neither ancestor nor descendant

* oa's expectations must be fulfilled by the ST of ob.

exp(?) ≥ exp(A)

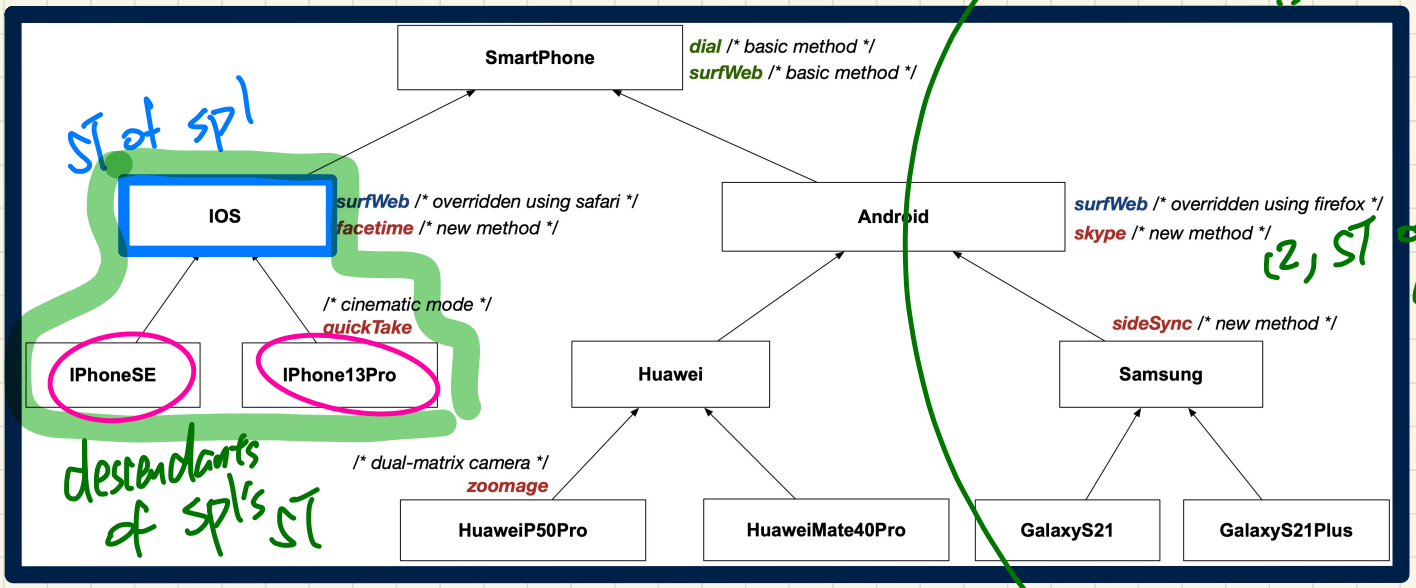
↳ ? must be a descendant of A.

- substitute oa by ob

exp(A)

- after the substitution we expect: *oa.

# Rules of Substitutions (1)




Compiles: (1) ST of sp2 (IPSE) is a descendant of the ST of sp1 (IOS).

(2) ST of sp2 can fulfill the exp of sp1's ST.

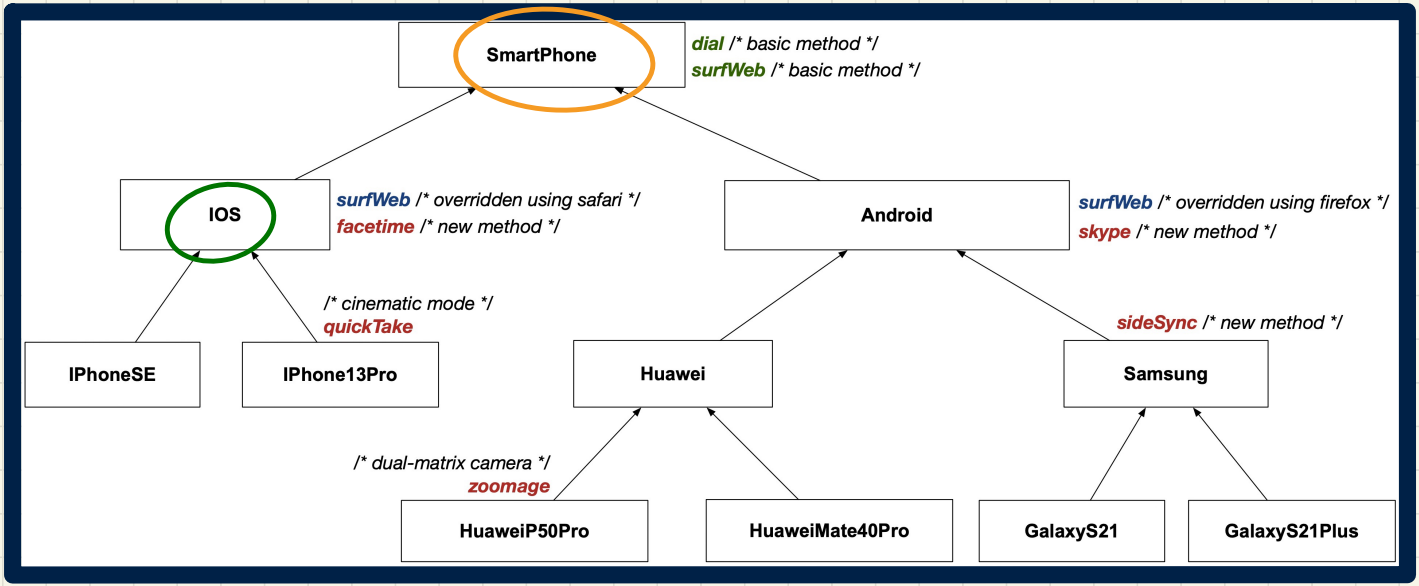
**Declarations:**

**IOS** sp1;
**IPhoneSE** sp2;
**IPhone13Pro** sp3;

**Substitutions:**

sp1 = sp2;
sp1 = sp3;

# Rules of Substitutions (2)




**Declarations:**

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

**Substitutions:**

```
sp1 = sp2;
```

fail to compile
-( SP is not descendant of IOS

# Visualization: Static Type vs. Dynamic Type

**Static Type**: Expectation

**Dynamic Type**: Accumulation of Code

**Declaration:**

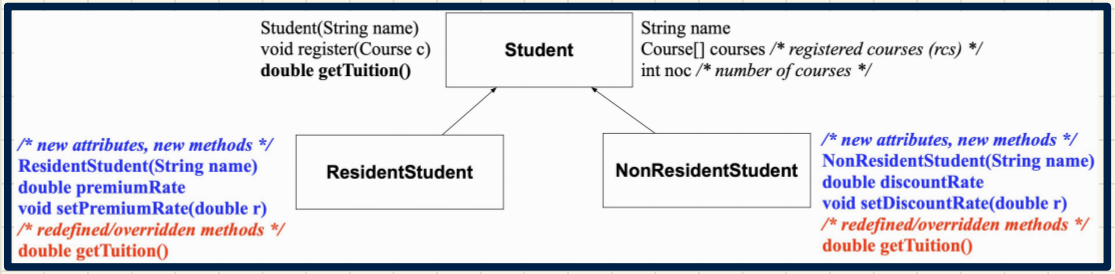
**Student** s;

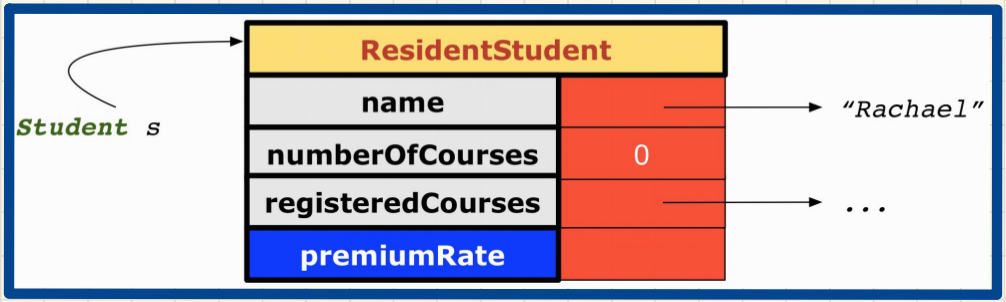
Compiler :( (1) RS is a des. of the ST of s (Student)

**Substitution:**

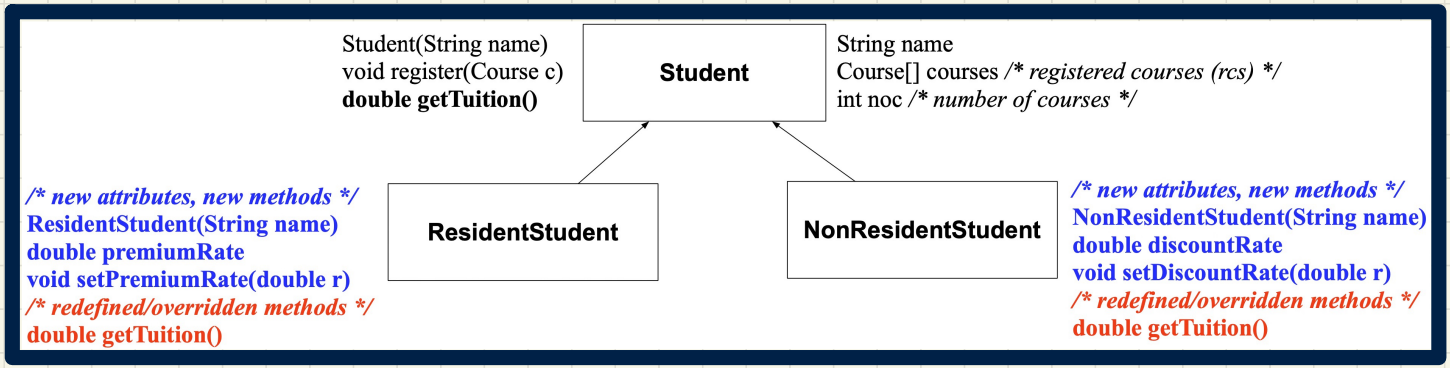
s = **new ResidentStudent**("Rachael");

(2) RS can fulfill the exp. of s' ST (Student)



# Change of Dynamic Type (1.1)

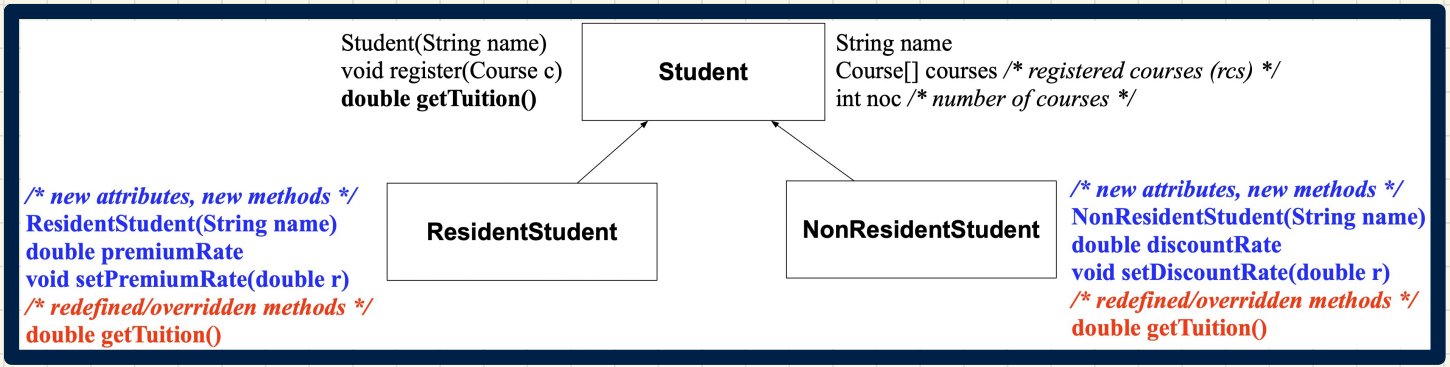



**Example 1:**

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

Compile : RS and NRS are descendants of jim's ST (Student).

# Change of Dynamic Type (1.2)

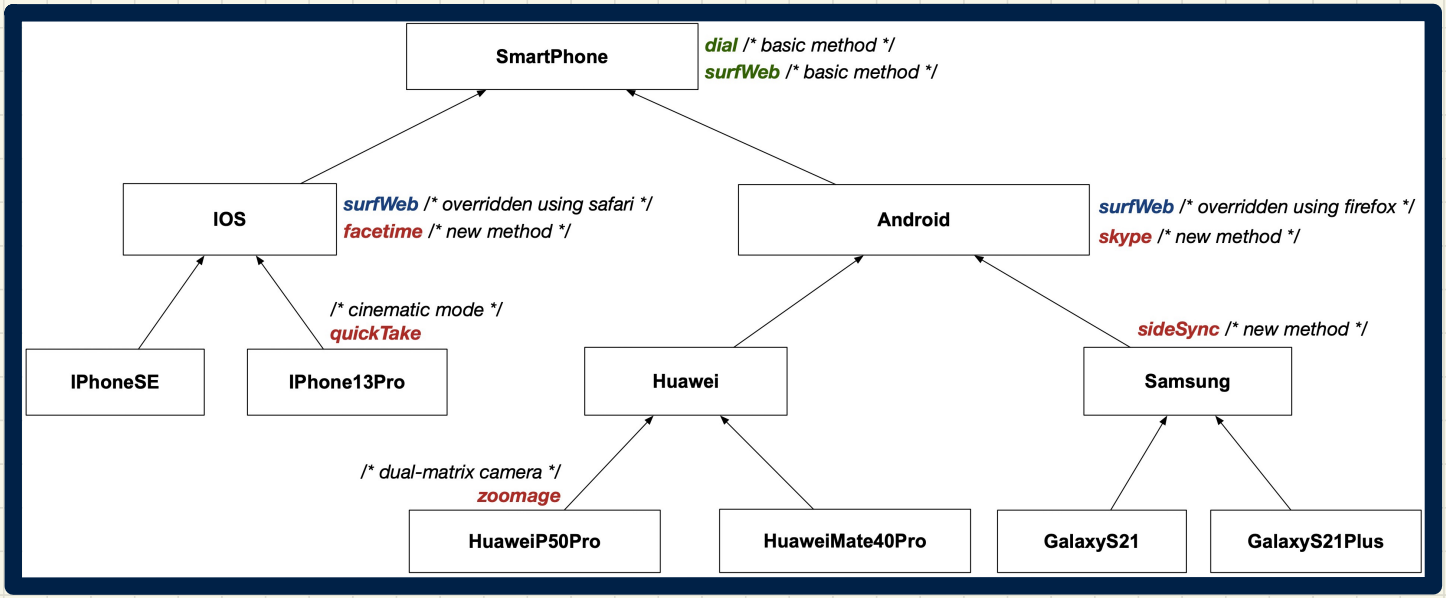



**Example 2:**

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

fail to compile
∵ Student cannot fulfil exp of jeremy's ST (RS).

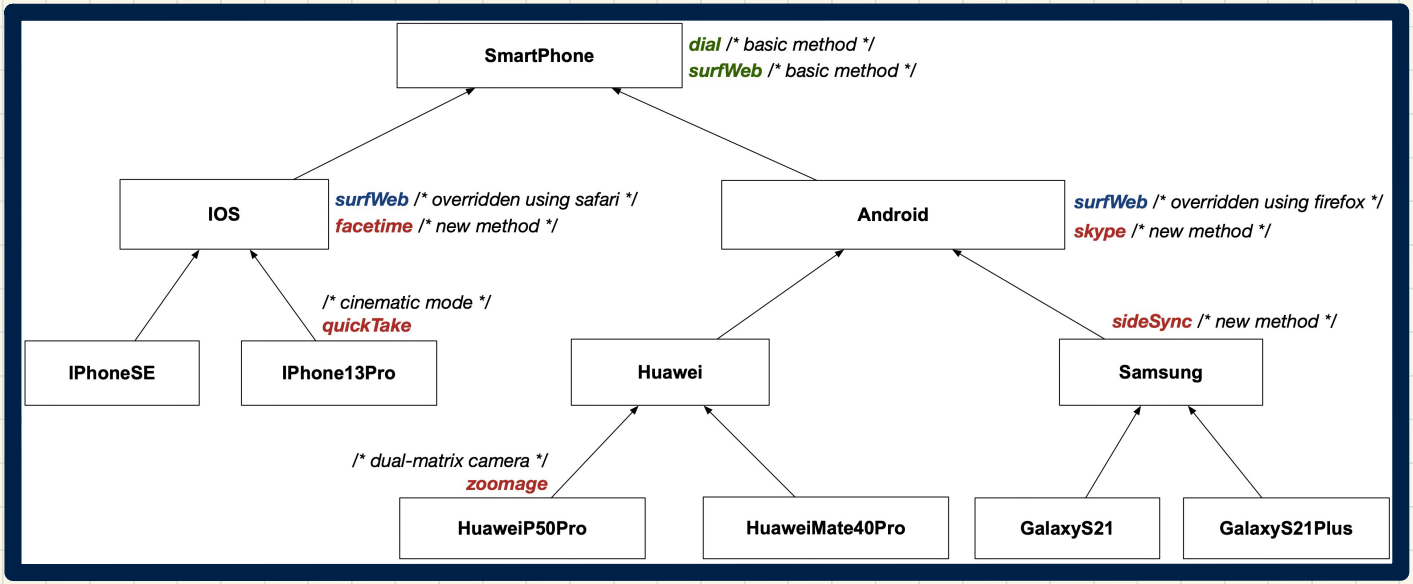
# Change of Dynamic Type: Exercise (1)




**Exercise 1:**

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

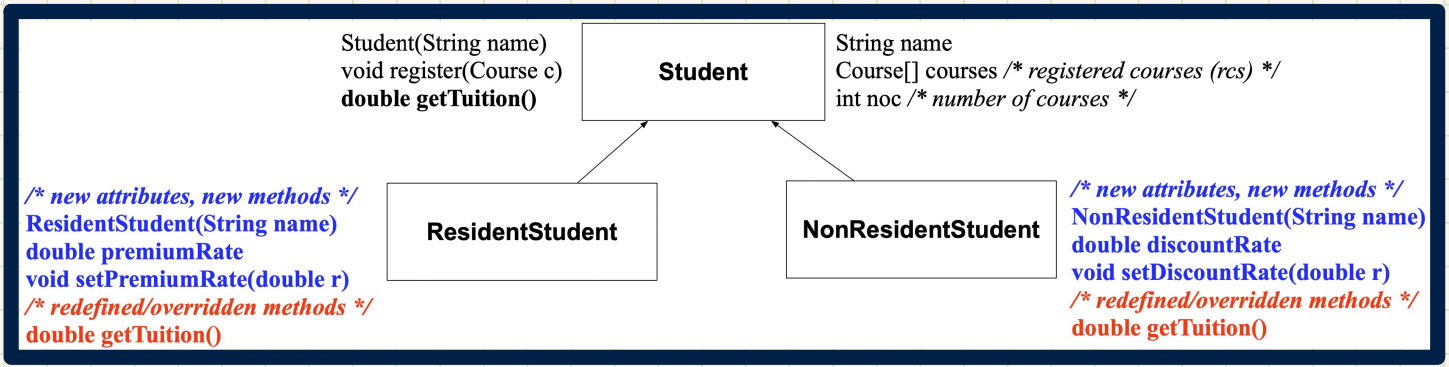
# Change of Dynamic Type: Exercise (2)




**Exercise 2:**

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

# Change of Dynamic Type (2.1)




**Given:**

```
① Student jim = new Student(...);
② ResidentStudent rs = new ResidentStudent(...);
③ NonResidentStudent nrs = new NonResidentStudent(...);
```

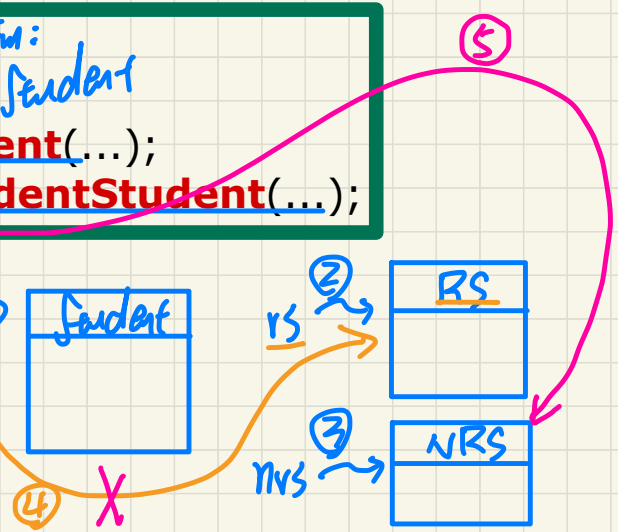
DT of jim: Student

**Example 1:**

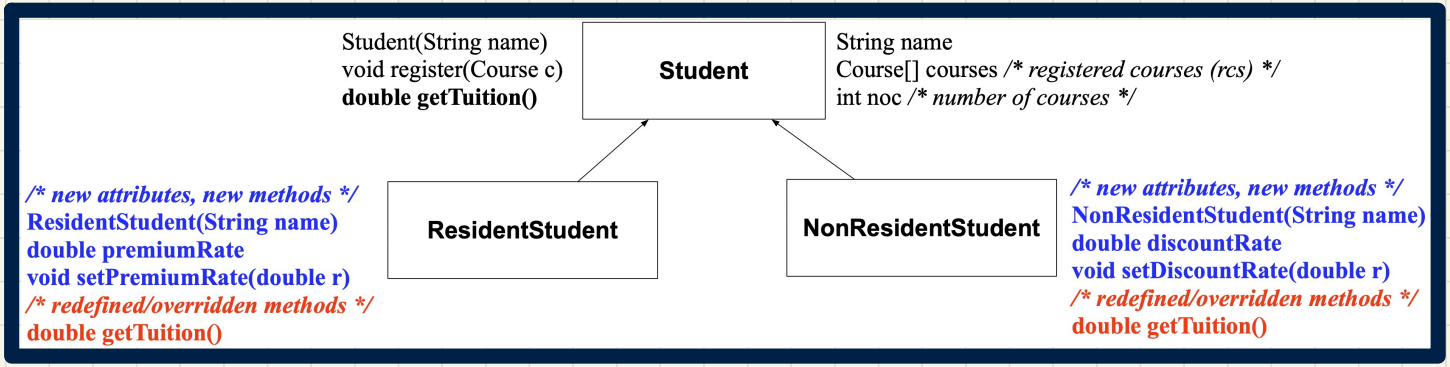
```
jim = rs;
println(jim.getTuition());
jim = nrs;
println(jim.getTuition());
```

④ DT of jim: changed to D.T. of rs (ResidentStudent)

⑤ DT of jim: changed to DT. of nrs (NonR.S.)



# 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());
```

not compile

∵ Student (jim's ST) not descendant of ST of LHS

# Lecture 19 - Nov 13

## Inheritance

***Type Casts: Motivation, Syntax, Visual***
***Compilable Casts: Upward vs. Downward***
***Casts at Runtime: ClassCastException***

## Announcements/Reminders

- Today's class: notes template posted
- Lab4 released
- WrittenTest2 guide and example questions released
- Notes on Type Casting

many shapes

# Polymorphism and Dynamic Binding

$\frac{T_1}{ST}$ var = $\underbrace{\text{new } T_2()}_{DT.}$;

$T_2$ must fulfill the exp($T_1$)

**Polymorphism**:

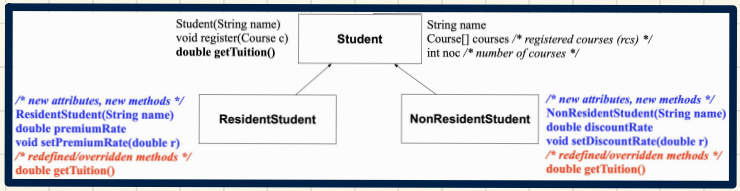
An object's **static type** may allow **multiple** possible **dynamic types**.

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

**Dynamic Binding**:

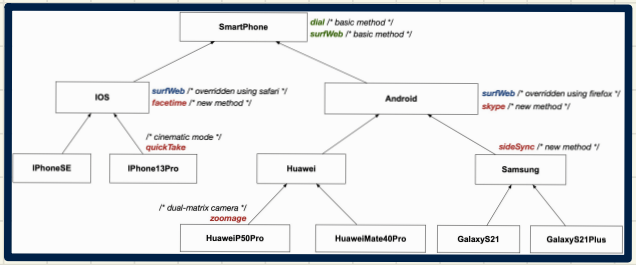
An object's **dynamic type** determines the **version** of method being invoked.

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

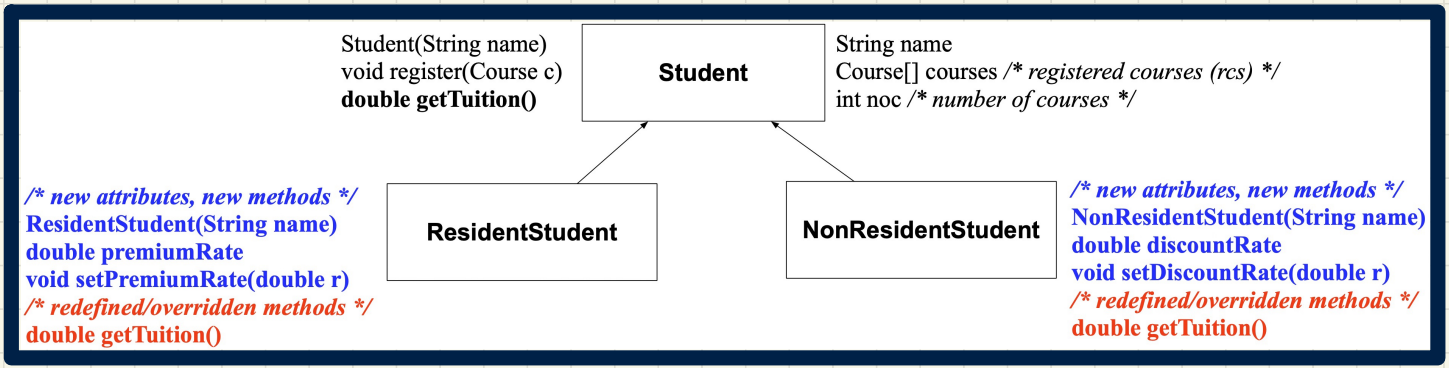



```
SmartPhone sp1 = new IPhone13Pro(...);
SmartPhone sp2 = new GalaxyS21(...);
sp1.surfWeb();
sp1 = sp2;
sp1.surfWeb();
```

after this substitution, DT of sp1 changes to DT of sp2.



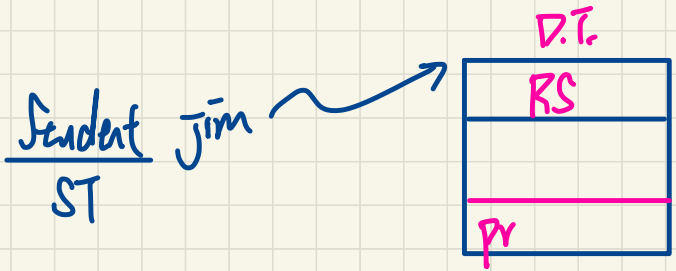
# Type Cast: Motivation



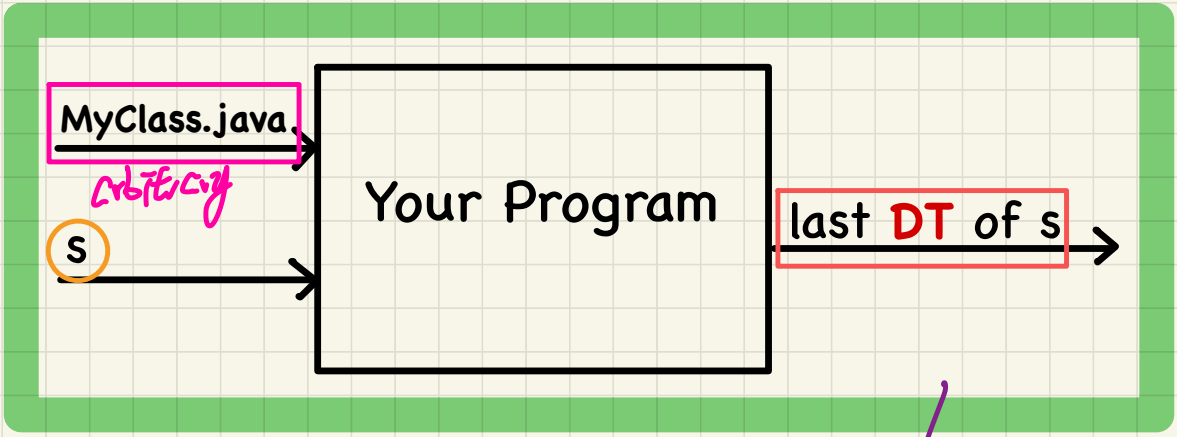

```
1  Student jim = new ResidentStudent("J. Davis");
2  ResidentStudent rs = jim;
3  rs.setPremiumRate(1.5);
```

ST: Student

not compile : Student (jim's ST) is not descendant of RS.



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




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

last DT of s

undecidable problem

(writing a program to infer the dynamic behavior of another arbitrary program)

# Anatomy of a Type Cast

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

ST: *ResidentStudent* — valid substitution — ST: Student

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

an alias whose ST is *ResidentStudent*

cast type

type cast expression

ST: ResidentStudent

jim.setPr ✗

rs.setPr. ✓

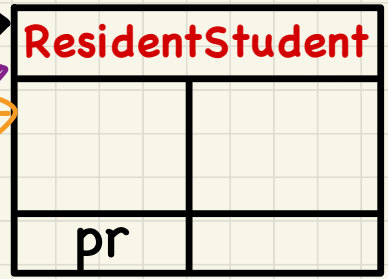
Student jim ✓

rs

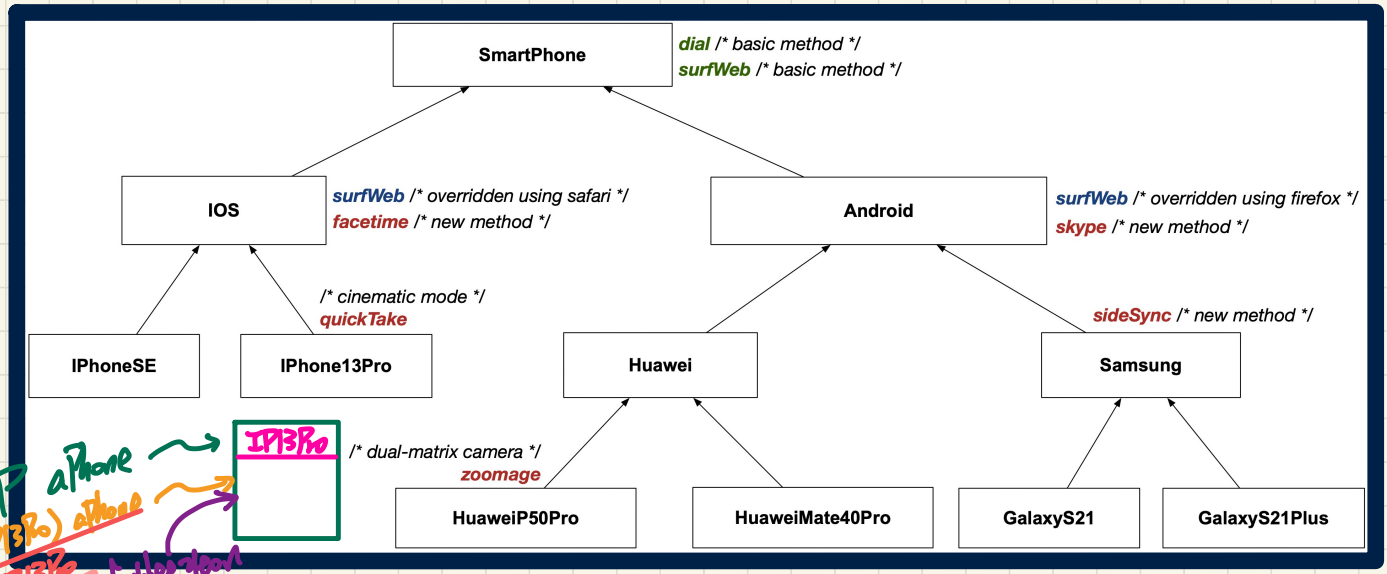
Resident Student

(RS) jim

ST: ResidentStudent



# Type Cast: Named vs. Anonymous




SP aPhone

(IP13Pro) aPhone

ST: IP13Pro

IOS forHeeyeon

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

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

**Anonymous Cast**: Use the cast result directly.

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

using type cast exp. as the C.O.

## Exercise

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

as if: (IP13Pro)(aPhone.facetime())

ST: SP ✗

static type consideration only

# Compilable Casts: Upwards vs. Downwards

dts regarding DT.

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

SmartPhone sp = (SmartPhone) myPhone;

GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
```

upward cast

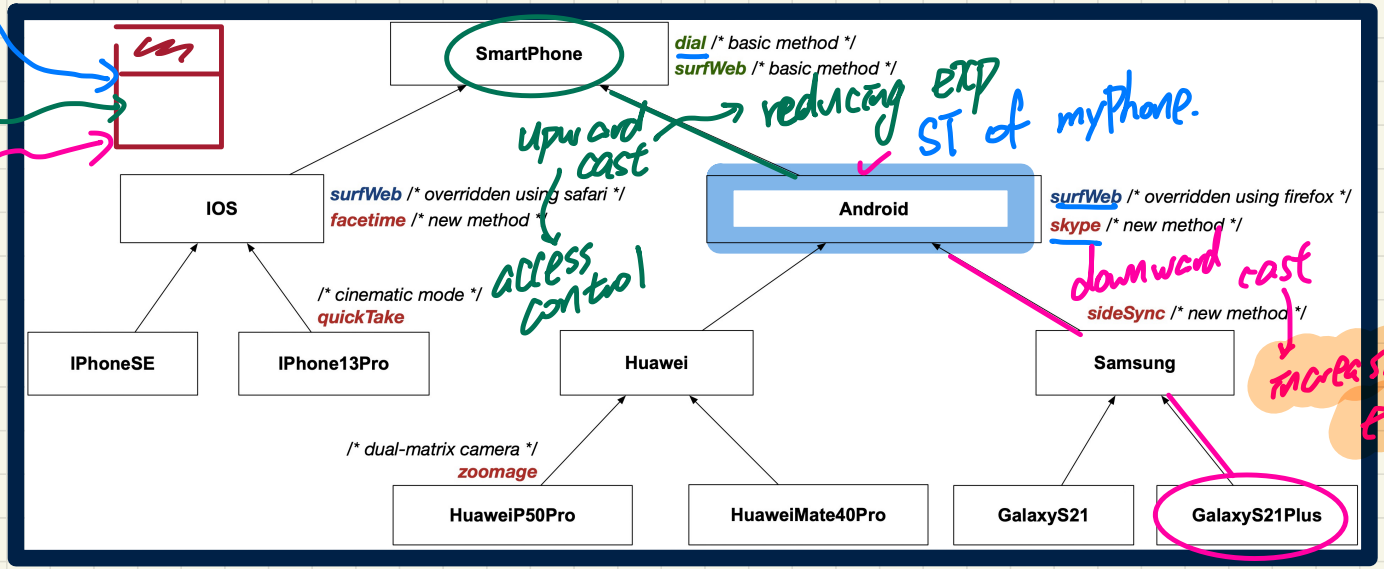
## Expectations

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

myPhone

sp

ga




upward cast

reducing exp ST of myPhone.

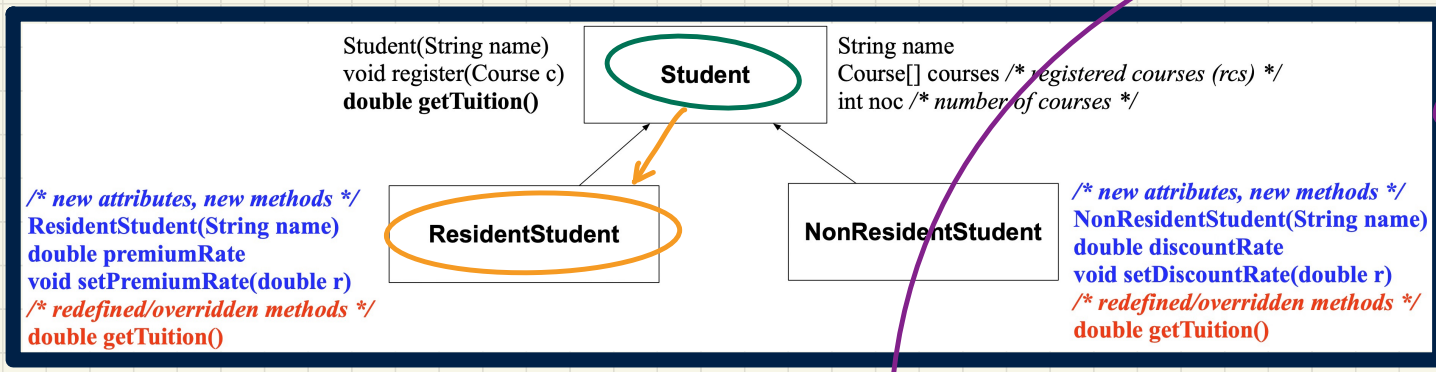
access control

downward cast

increasing exp

# Compilable Type Cast May Fail at Runtime (1)

trigger: ClassCastException (CCE)




(1) when the DT (e.g. NRS) cannot fulfill the exp. of cast type. e.g. RS

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

downward cast

(2) when the DT is not a descendant of cast type.

exp (RS) → pr.

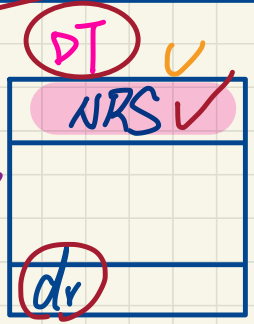
ST: Student

RS

jim

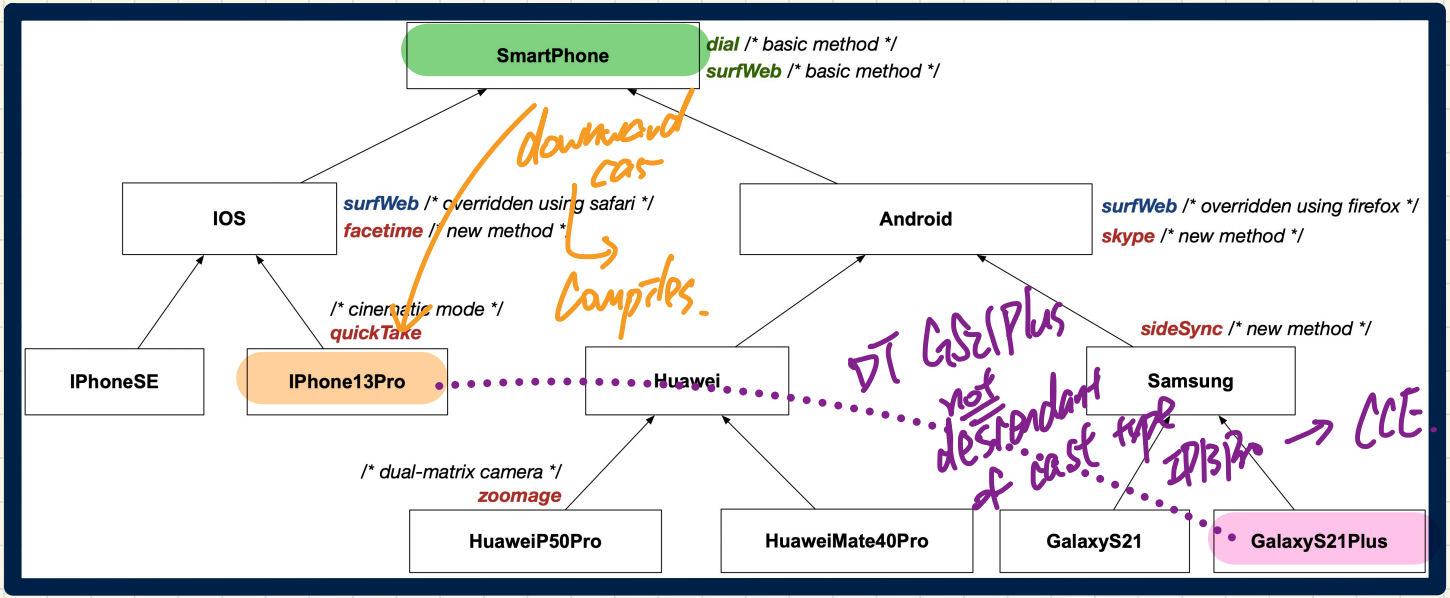
rs

(RS) jim



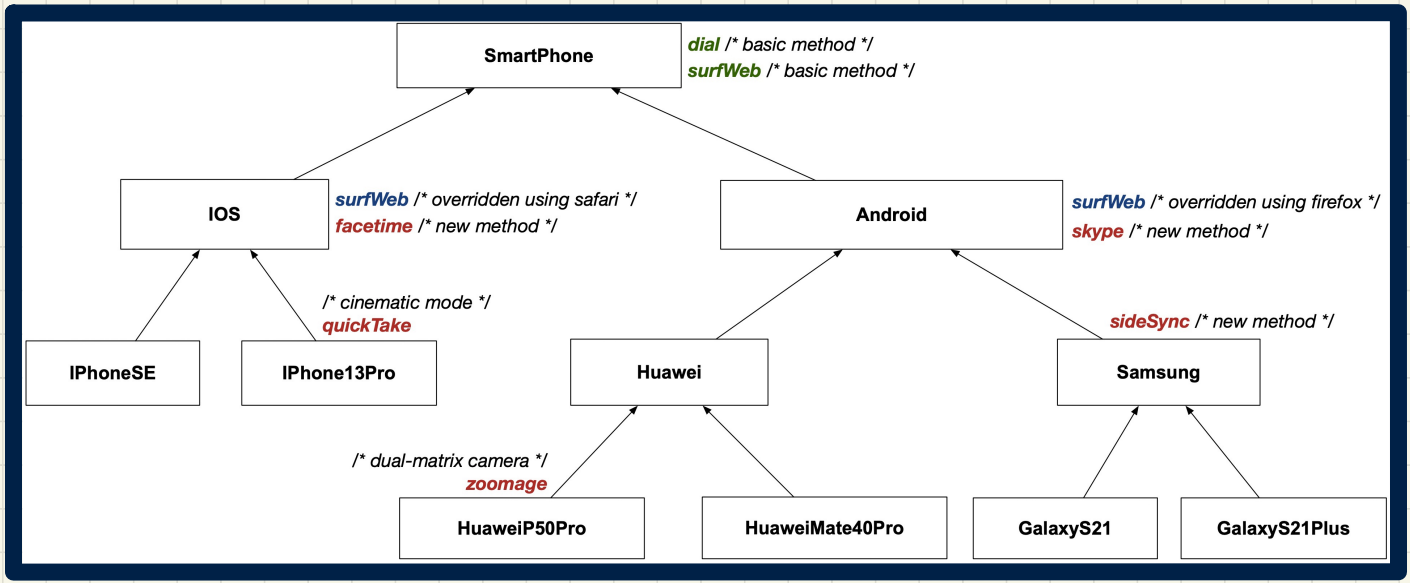

issue: DT NRS cannot fulfill the exp (RS)

# Compilable Type Cast May Fail at Runtime (2)




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

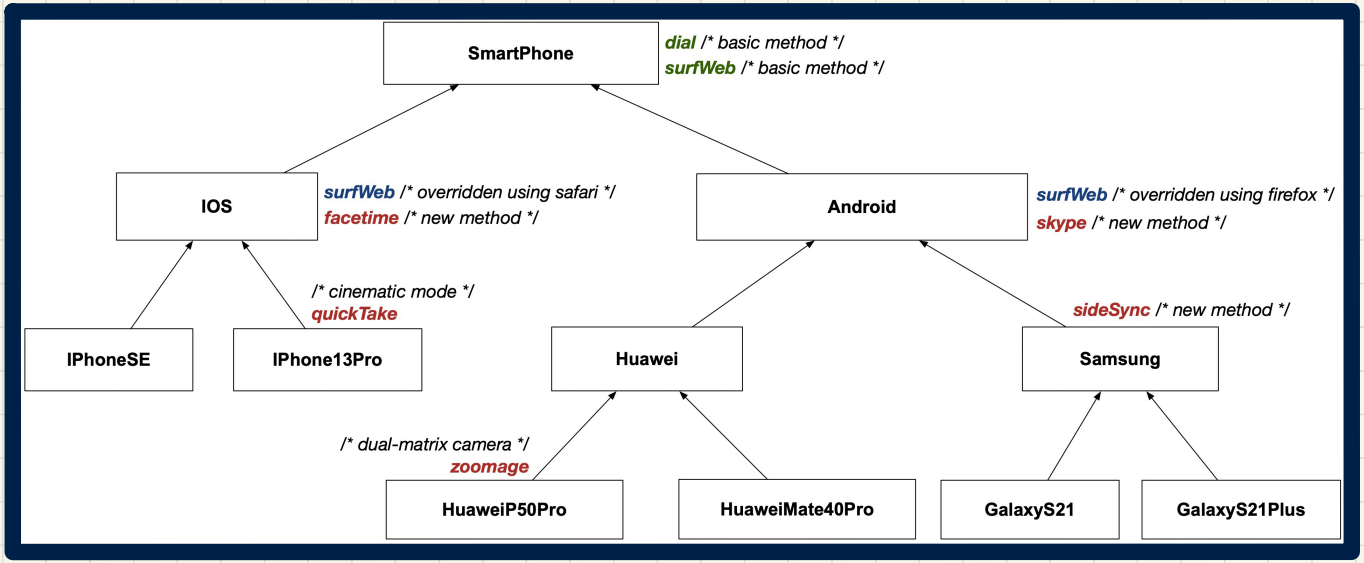
# Exercise: Compilable Type Cast? Fail at Runtime? (1)




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

**Compilable? ClassCastException at runtime?**

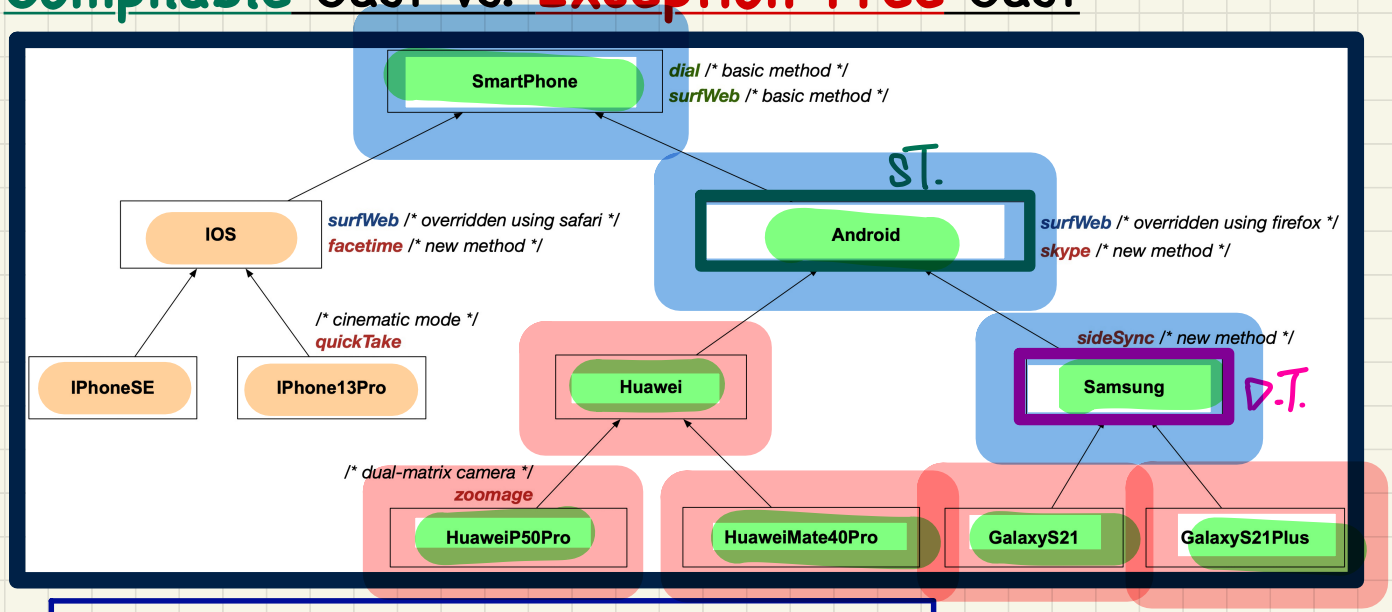
# Exercise: Compilable Type Cast? Fail at Runtime? (2)




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

**Compilable? ClassCastException at runtime?**

# Compilable Cast vs. Exception-Free Cast




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

Compilable Casts

Exception-Free Casts

Non-Compilable Casts

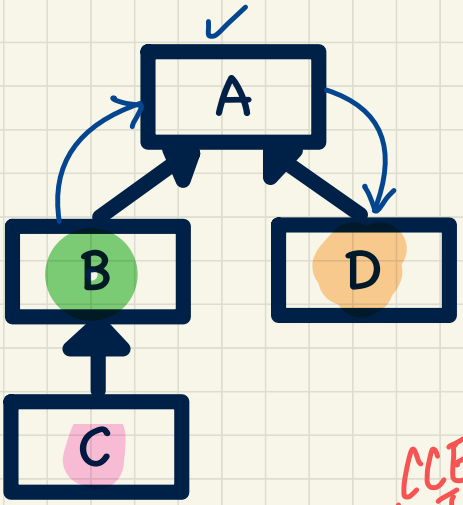
ClassCastException

# Exercise: Compilable Cast vs. Exception-Free Cast

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

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

× compile ∵ cast type D is neither ancestor nor descendant of b's ST (B)




downward cast

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

no CCE ∵ DT C is descendant of cast type A.

B b

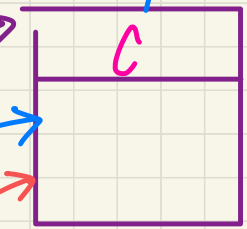
upward cast

ST: A.

CCE ∵ DT C cannot fulfill exp(D) cast type.

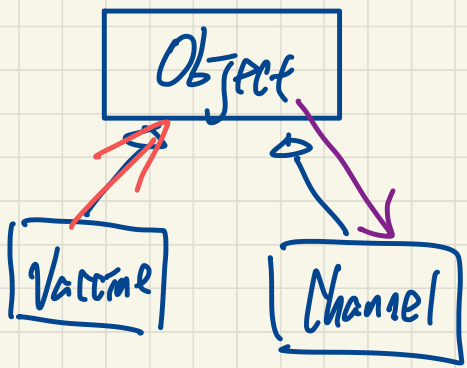
(A) b

(D) ((A) b)



Vaccine v = ... ;

Channel c = (Channel)((Object) v)



# Lecture 20 - Nov 18

## Inheritance

***instanceof vs. Compilable Cast***
***instanceof vs. ClassCastException Freedom***
***Call by Value and Inheritance***

## Announcements/Reminders

- Today's class: notes template posted
- Lab4 due soon
- WrittenTest2 guide and example questions
  + **In-person** review session: 3 PM on Tuesday @ R N203
  + Notes on Type Casting
- Online Course Evaluation

can fulfill, dynamically, exp. of

D.T.

$x$ | instanceof | Y

an expression denoting some object

e.g., $x$ : $a[i]$ → Y

a class name.

usually used ver. before a subsequent cast.

Compiles iff

$(Y)\,x$ compiles

upward or downward cast.

no CCE!!

$x$ ⇝ [ $x$'s DT ]

↳ evaluates to Boolean ( True or False )

↳ True if (1) DT of $x$ can fulfill exp(Y)

(2) DT of $x$ is descendant of Y

(3) Y is ancestor of DT of $x$

false otherwise

↳ if true, then it's safe to execute $(Y)\,x$

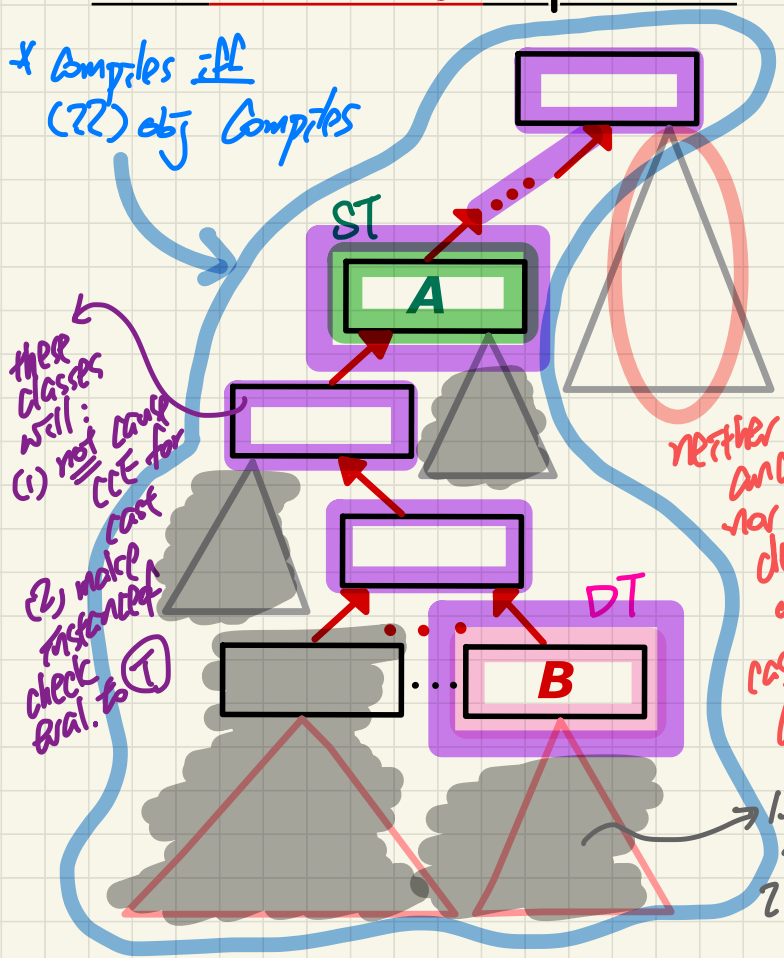
Assume: both exp. compile

| | DT of $x$ fullfill exp(Y) | ¬(DT of $x$ fullfill exp(Y)) |
|---|---|---|
| $x$ instanceof Y | true | false. |
| (Y) $x$ | type cast succeeds, creating an alias of ST Y | CCE! |

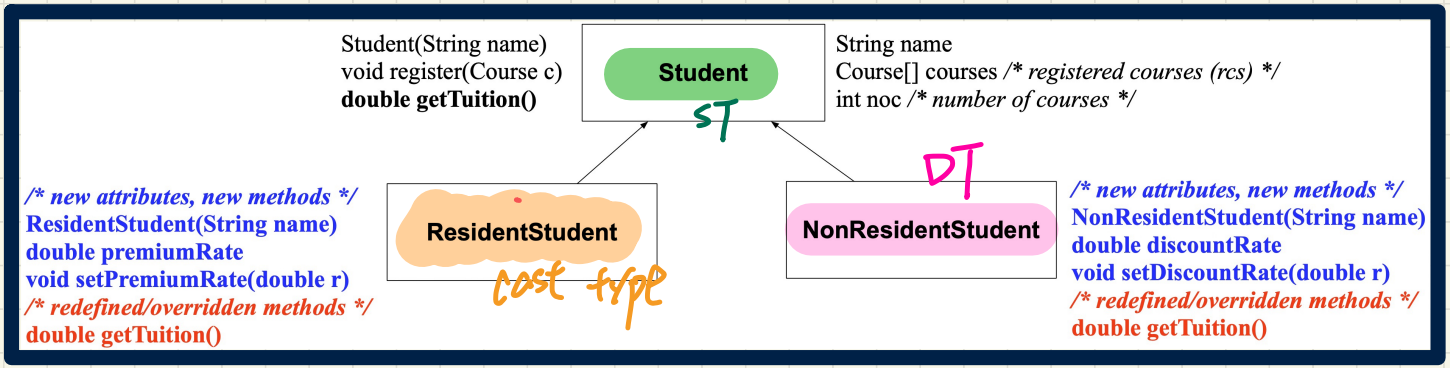
# The instanceof Operator

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

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



# Checking Dynamic Types at Runtime (1)




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

1. compile? downward cast
2. CCE!
∵ DT NRS not descendant of cast type RS.

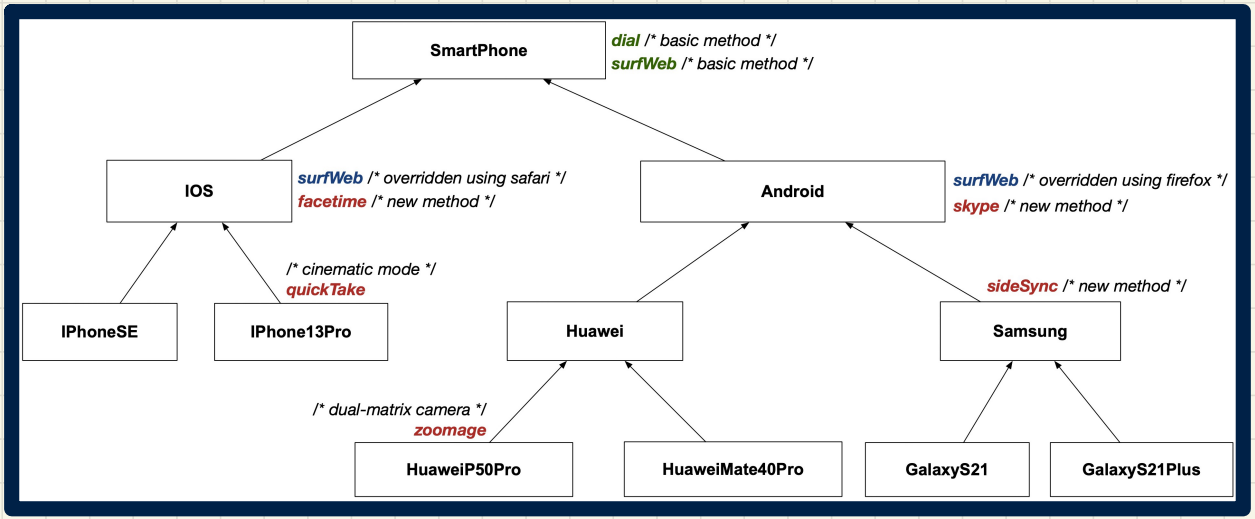
.....bypassed, preventing CCE from happening!!

Can Jim's DT fulfill the exp (RS)

↳ NO → false

# Checking Dynamic Types at Runtime (2)

Exercise

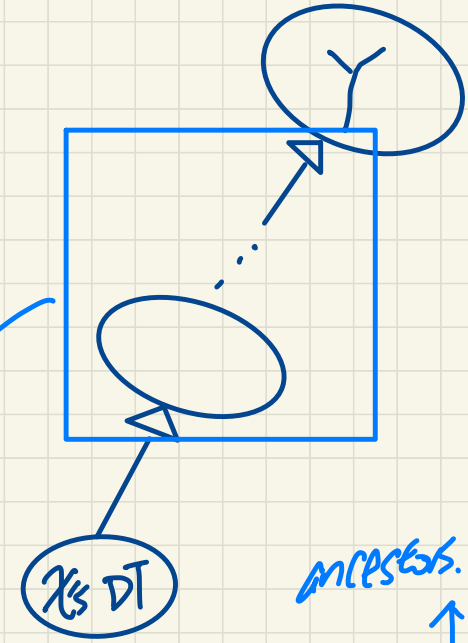



```java
SmartPhone aPhone = new GalaxyS21Plus();
if (aPhone instanceof IPhone13Pro) {
  IOS forHeeyeon = (IPhone13Pro) aPhone;
  forHeeyeon.facetime();
}
```

$\underbrace{x.\text{getClass}()}_{\text{DT of } x} == Y$

$x$ instanceof $Y$

DT of $x$

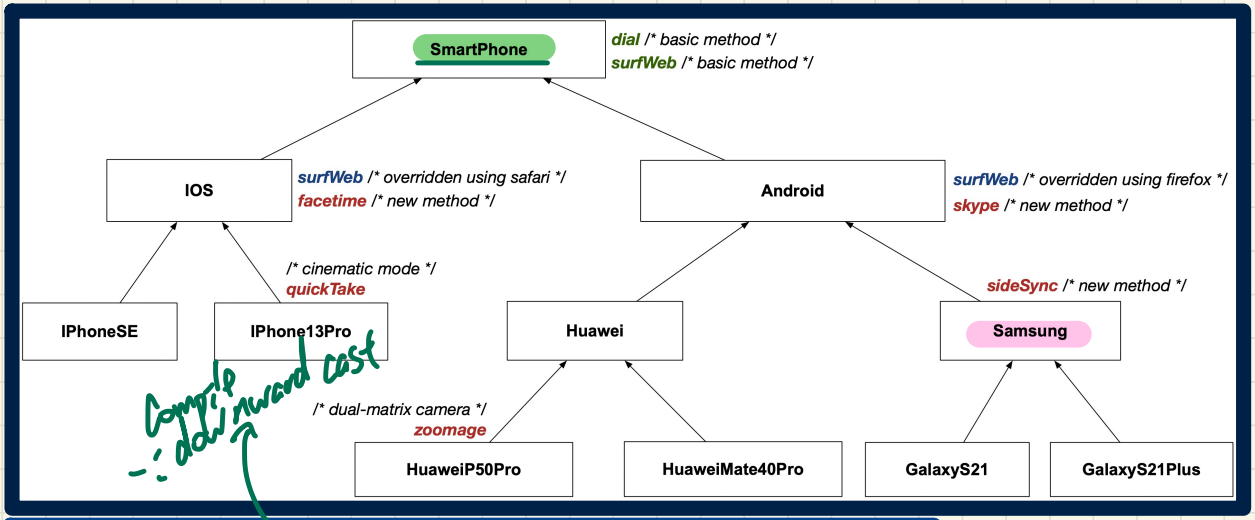



ancestors.

$x$ instanceof $Y$ eval. to T

Tuples that $x$'s DT can also fullfill these "close"!

# Use of the instanceof Operator



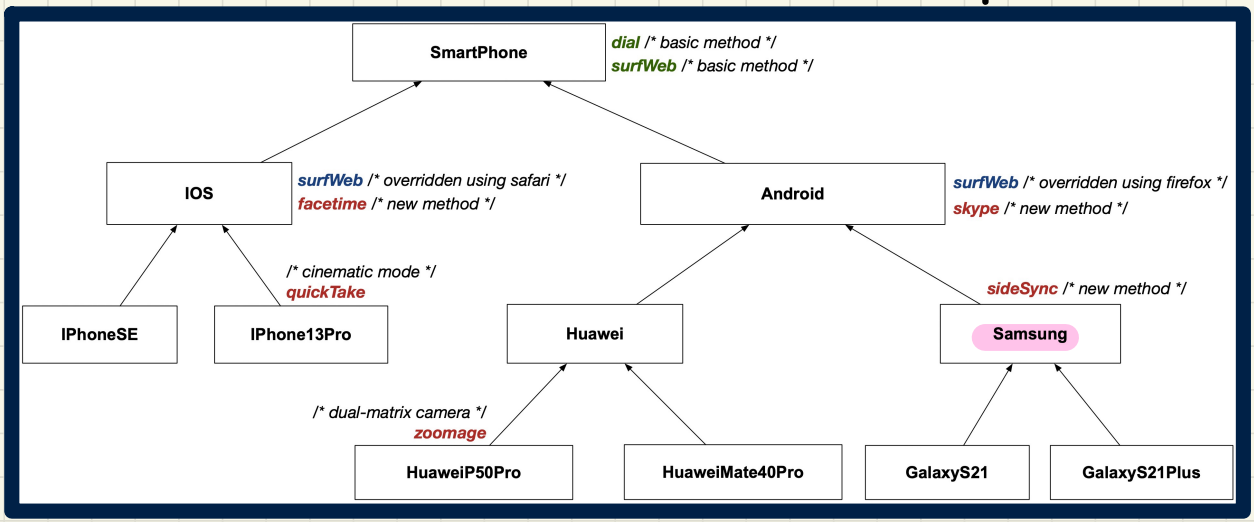

Compile-time error

```
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

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

# Safe Cast via Use of the instanceof Operator




```
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;
}
```

T

no CCE

E

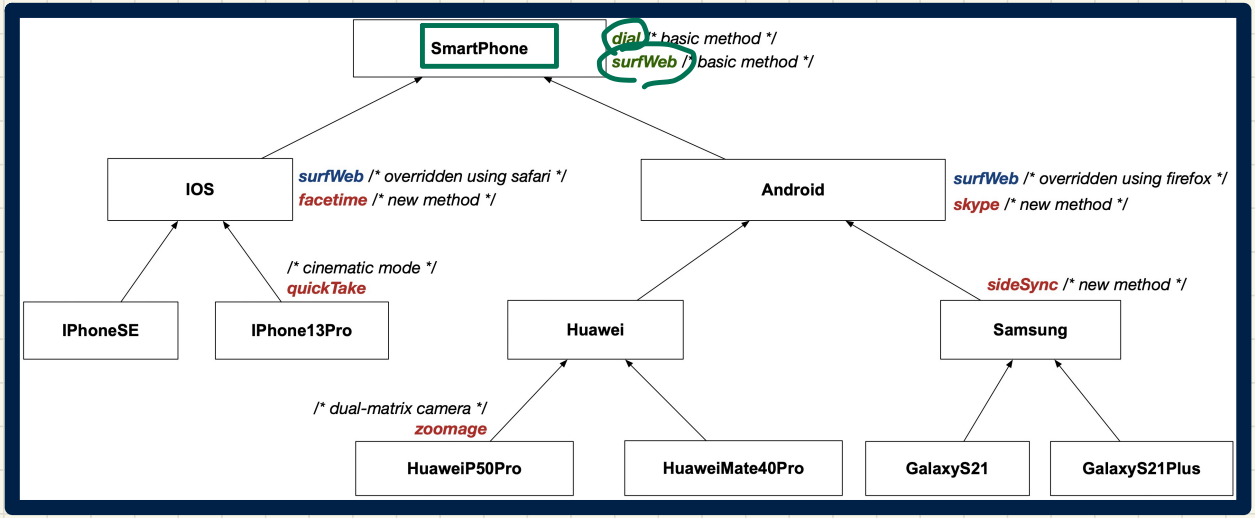
bypassed

E

CCE prevented!

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

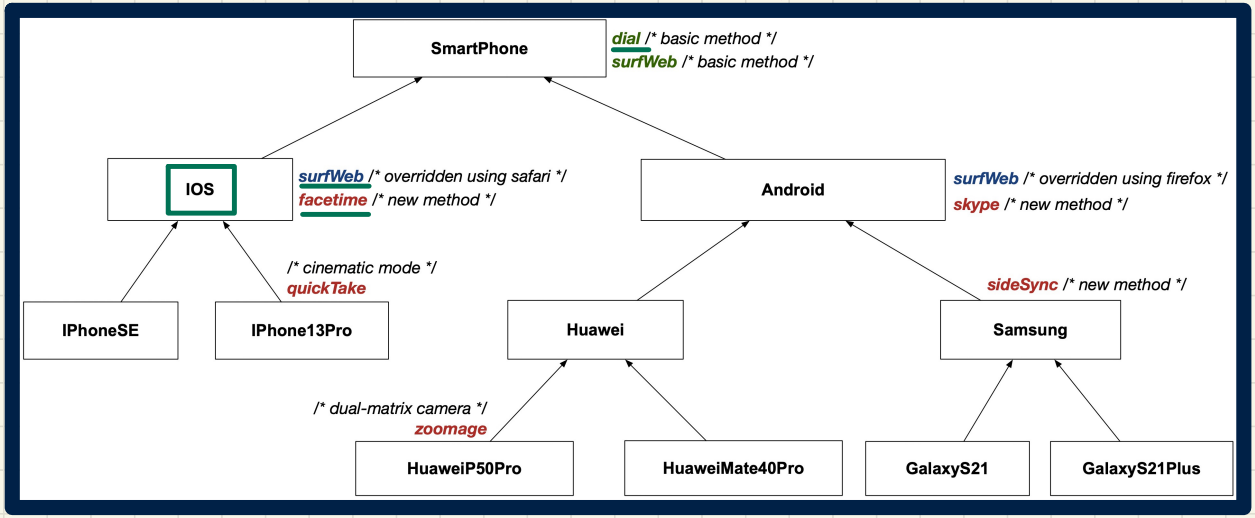
# Static Types, Casts, Polymorphism (1)




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

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

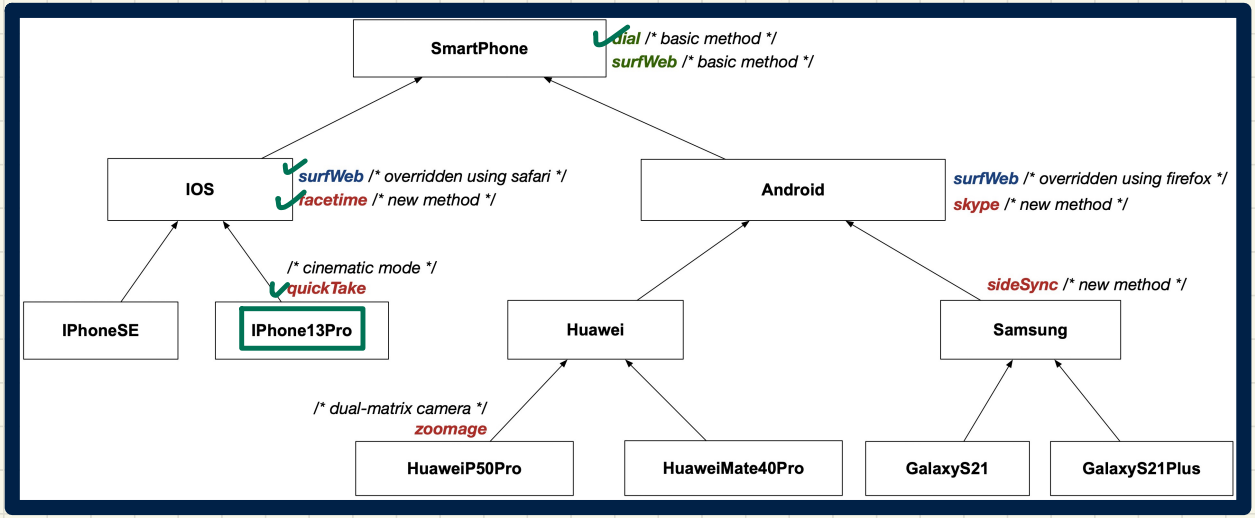
# Static Types, Casts, Polymorphism (2)




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

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

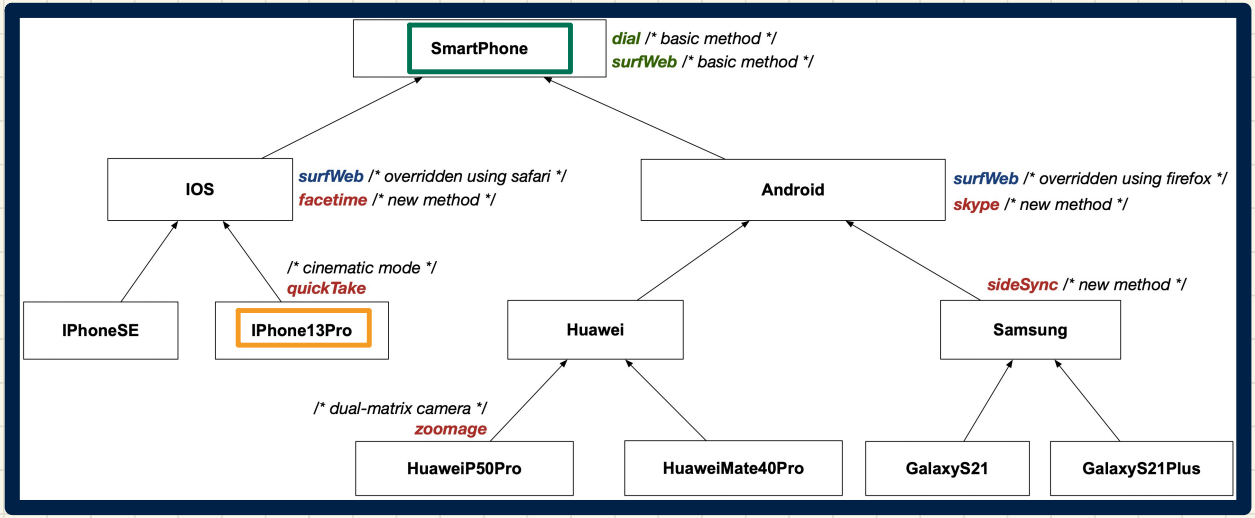
# Static Types, Casts, Polymorphism (3)




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

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

# Static Types, Casts, Polymorphism (4)

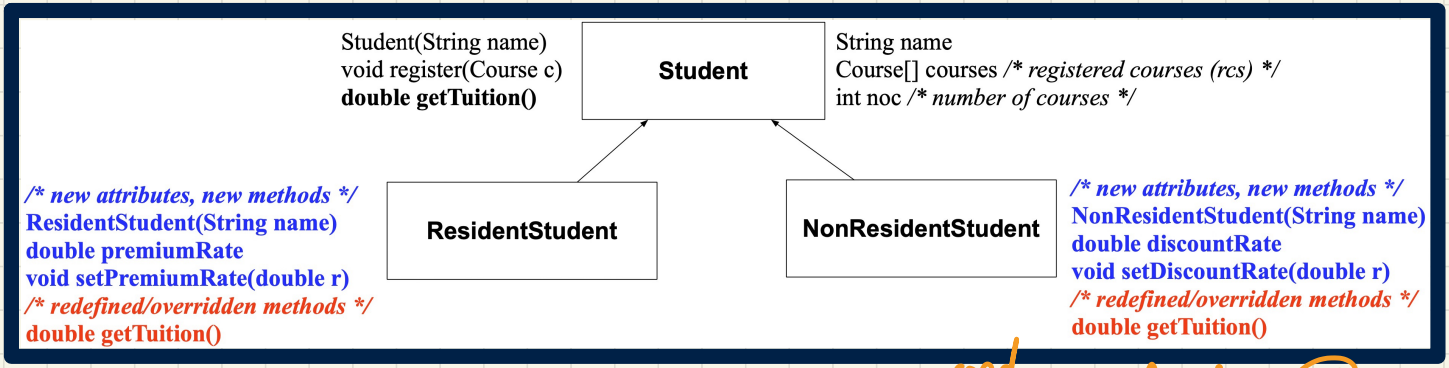



```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

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

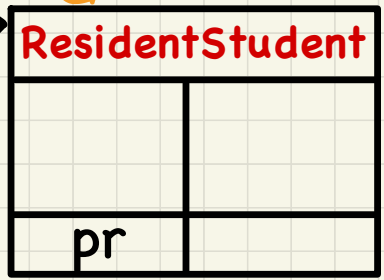
cast compiles
( exp has ST: IP13Pro)

# Static Types, Casts, Polymorphism (5)




safety guaranteed ∵ instanceof check evaluates to T.

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());
}
```

true if it's safe to cast s to RS

# Call by Value

Callee

```
class C {

void m(T1 param){
    ...
  }
}
```

call by value:*

param = arg

ST: T1 ST: T2

Caller

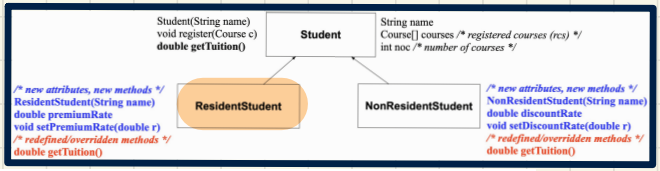
```
C obj = new C();
T2 arg = ... ;
obj.m(arg);
```

*Rule of substitutions

(1) T2 fulfills exp(T1)

(2) T2 descendant of T1

# Polymorphic Parameters (1)




each element of ssy has ST 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]?

ST: Student.

call by value:

rs = 0

ST: RS ST: ?

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

ST: Student ST: RS

compiles if ? is descendant of RS.

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

Student No. sms.addRS(o) → under what condition does this method call compile?

# Polymorphic Parameters (2)

call to this method only compiles if the ST of arg. rs descendant of ST of param (RS)

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

arg with ST a descendant of student compiles

```
Student s1 = new Student();
Student s2 = new ResidentStudent();
Student s3 = new NonResidentStudent();
ResidentStudent rs = new ResidentStudent();
NonResidentStudent nrs = new NonResidentStudent();
StudentManagementSystem sms = new StudentManagementSystem();
① sms.addRS(s1); ✗
② sms.addRS(s2); ✗
③ sms.addRS(s3); ✗
④ sms.addRS(rs); ✓
⑤ sms.addRS(nrs);
sms.addStudent(s1); ✓
sms.addStudent(s2); ✓
sms.addStudent(s3); ✓
sms.addStudent(rs); ✓
sms.addStudent(nrs); ✓
```

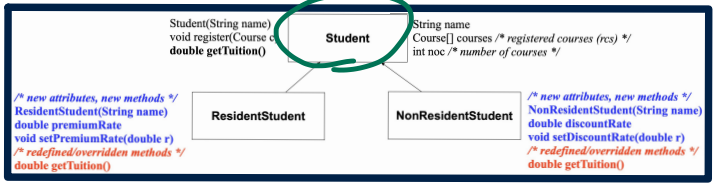
ST: RS → ST: S

call by value : rs = s1

ST: RS → ST: RS

rs = rs

c.b.v : rs = nrs ✗



# Lecture 21 - Nov 20

## Inheritance

***Casting Method Arguments***
***Polymorphic Arrays and Return Types***
***Recursion: Resources***

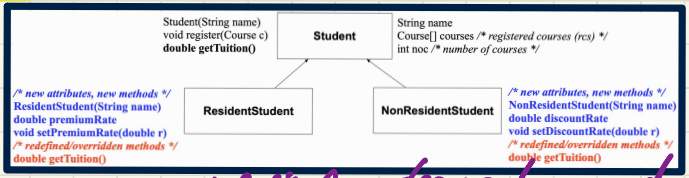
## Announcements/Reminders

- Today's class: notes template posted
- Lab4 solution released
- ProgTest3 guide and example questions released
- Online Course Evaluation

# Casting Arguments

c.b.v. rs = (RS) s

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




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

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

c.b.v. rs = s

fixed

* cast compiles ∵ downward cast
* cast exp. has ST: RS

ClassCastException?

YES!

∵ DT Student not descendant of cast type RS

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

fixed

ClassCastException?

YES!!

∵ DT NRS cannot fulfill exp(RS)

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

fixed

ClassCastException?

No.

CCE free

fail to compile -(RS) nrs

sms.addRS( (ResidentStudent) nrs) compiles?

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

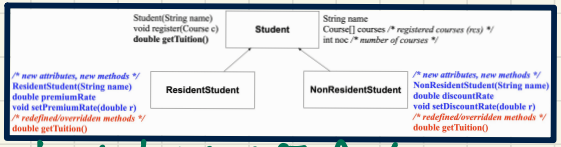
if (nrs instanceof RS) { ... }

fails to compile

neither upward nor downward cast → compile ×

ST of elements: Student. DTs of elements: descendants of ST.

# 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());
}
```

| I | DT |
|---|---|
| 0 | RS |
| 1 | NRS |

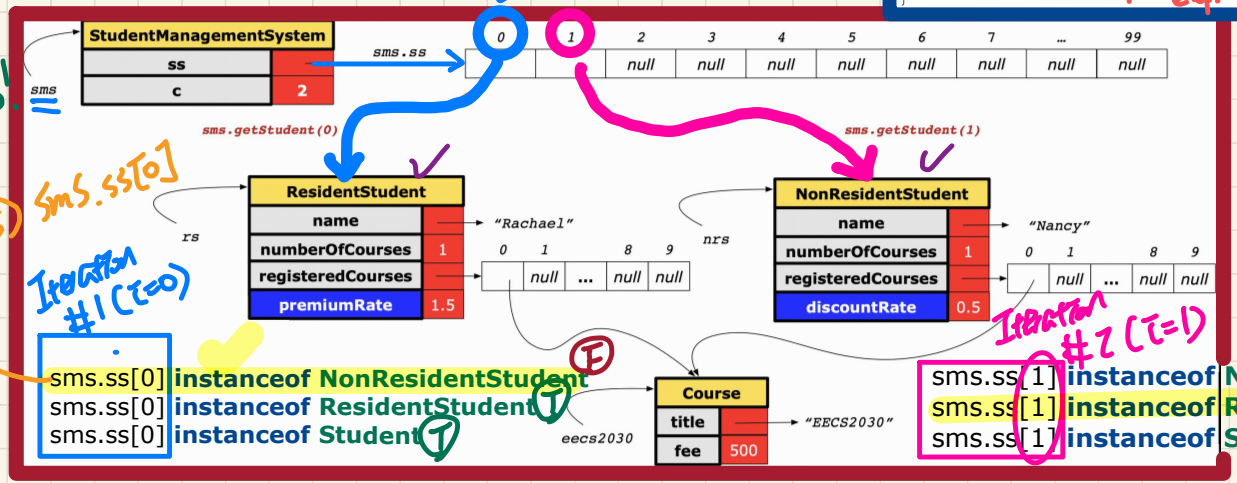
elements have same ST: Student.

```
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)
    }
  }
}
```

any arg. with ST a descendant of Student can be passed.

ST: Student. part of exp.

| I | DT of ss[I] |
|---|---|
| 0 | RS |
| 1 | NRS |




compiles! (NRS) sms.ss[0]

Iteration #1 (I=0)

Iteration #2 (I=1)

# 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? */
```

DT: RS

ST: Student

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? */
```

DT: NRS

ST: Student

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); ×
```

ST: Student

ST of return value

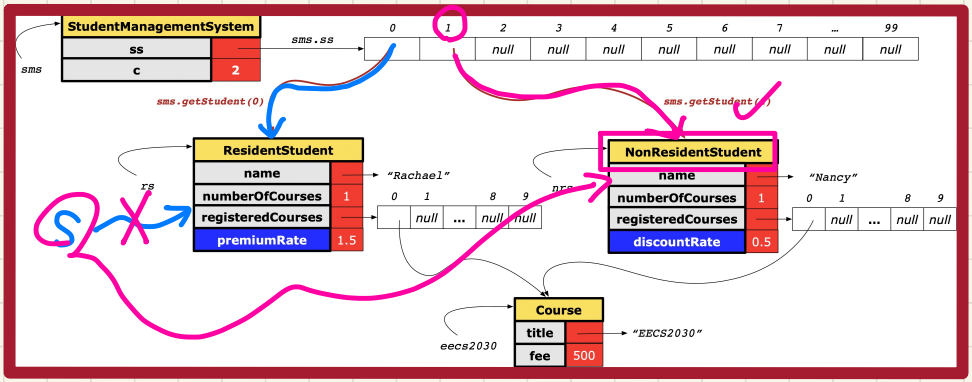
↳ DTs of return values are descendants of Student.

```
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;
  } }
```

0 1

polymorphic array.

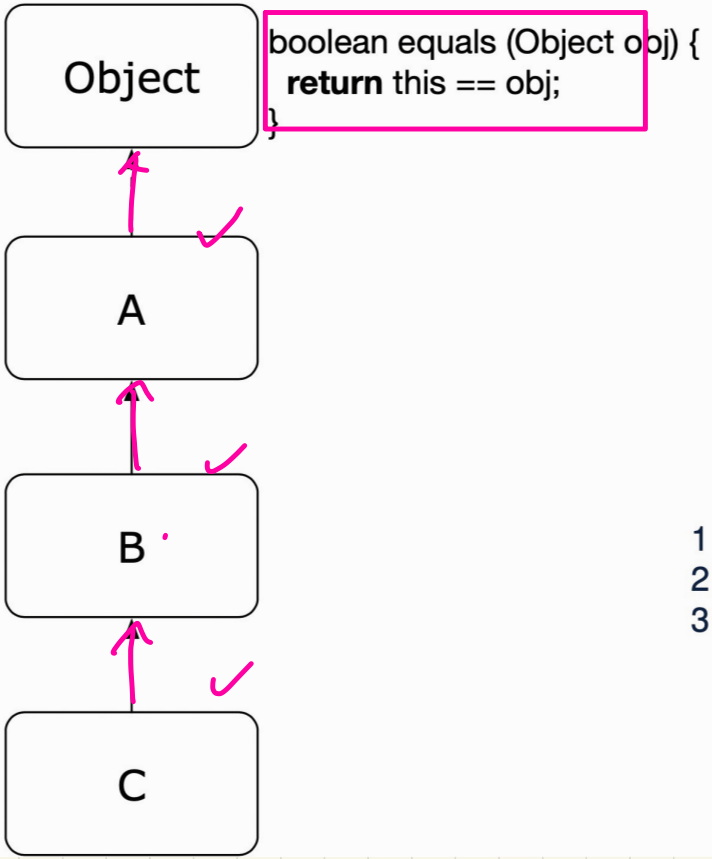
| i | DTs |
|---|---|
| 0 | RS |
| 1 | NRS |



# 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***?<br>Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***? |
| `z = x.m(y)` | Is method `m` defined in `x`'s ***ST***?<br>Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***?<br>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***?<br>Is `C` a **descendant** of `x`'s ***ST***? |
| `x.m((C) y)` | Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***?<br>Is method `m` defined in `x`'s ***ST***?<br>Is `C` a **descendant** of `m`'s parameter's ***ST***? |

# Overridden Methods and Dynamic Binding (1)



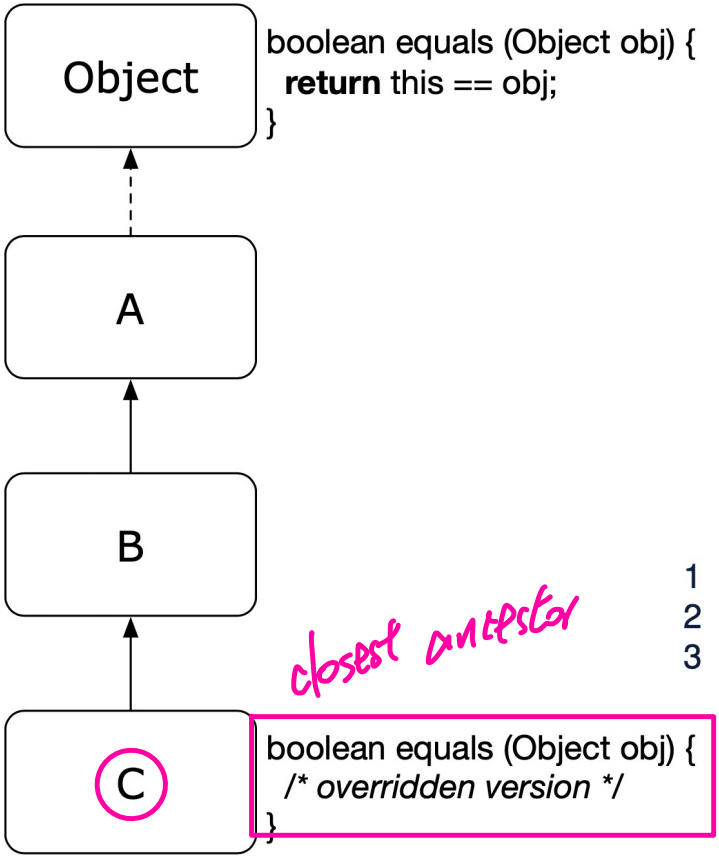

```
boolean equals (Object obj) {
  return this == obj;
}
```

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

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

**L3** calls which version of `equals`? Object.

# Overridden Methods and Dynamic Binding (2)



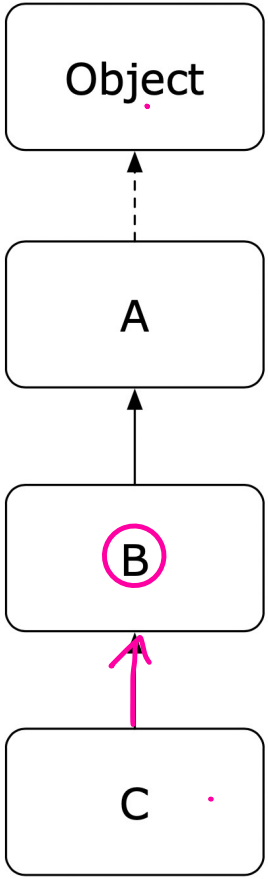

```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
```

```
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) {
  /* overridden version */
}
```

```
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 25

## Recursion

***Recursion: Base vs. Recursive Cases***
***Tracing Recursion: Stack vs. Tree***
***Examples: fac, fib, isPalindrome***

## Announcements/Reminders

- Today's class: notes template posted
- ProgTest2 results and feedback released
- Lab5 released (deadline: Dec 2; grace period until Dec 9)
  + Required background study: abstract classes & interfaces

* In Java, make recursive call(s), assume correct results returned.

# Solving a Problem Recursively

Computed & returned right away.

base case(s)

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

Given a **big** problem:

divide

Divide it into **smaller** problems: $SP_1$ $SP_2$ $SP_3$

*

Assume solutions to **smaller** problems: $S_1$ $S_2$ $S_3$

Combine solutions to **smaller** problems:

combiner.

direct recursion calls itself

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

$j < i$.

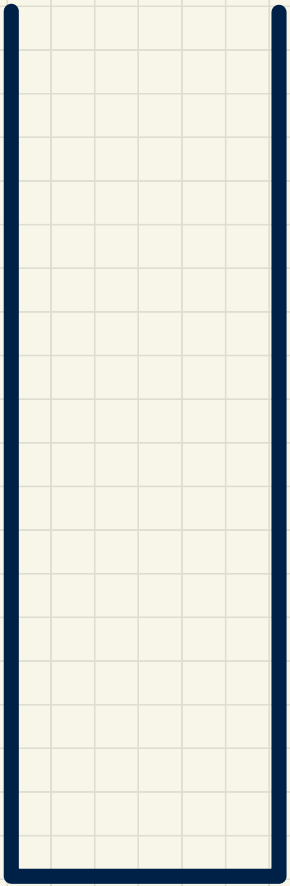
a recursive call

# Tracing Recursion via a Stack

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

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

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



Runtime Stack

$4! = 4 * 3 * 2 * 1 = 24.$

# Recursive Solution: factorial

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

not a recursive def.

the function being defined (i.e., "!") is not recused/called.

to define $n!$, we use the def. of $\boxed{(n-1)!}$ → strictly smaller input.

## Recursive Def.

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

base case

recursive case

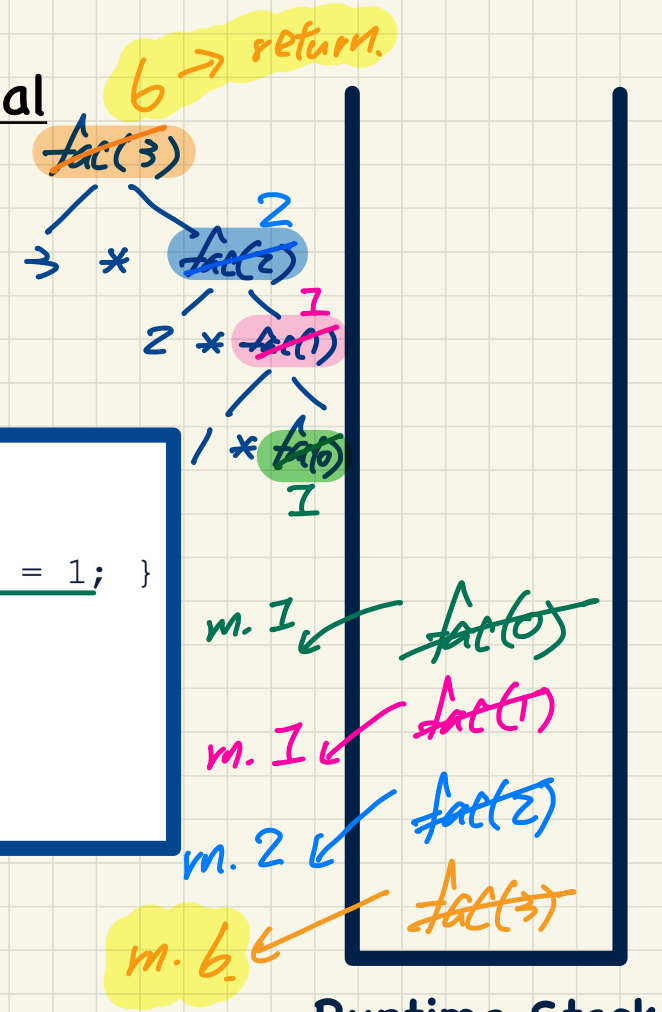
recursive call.

# 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;
}
```

**Example:** factorial(3)



# Tracing Recursive Methods

(1) Using a call stack

Pro: simulates how computer works at runtime

Con: for complicated tasks, using stacks can make it hard to gain intuition about the correctness of recu. method.

(2) Using a recursion tree

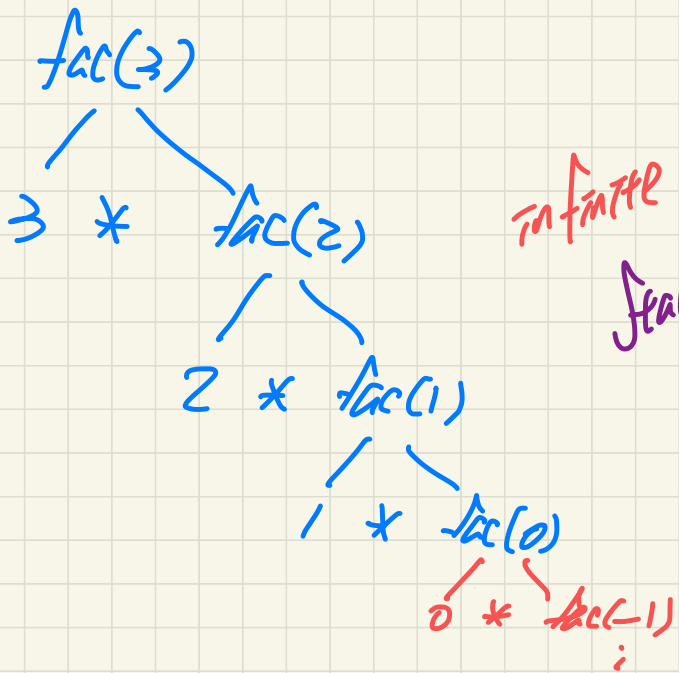
Con

Pro.

# Common Errors of Recursion (1)

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

missing base case



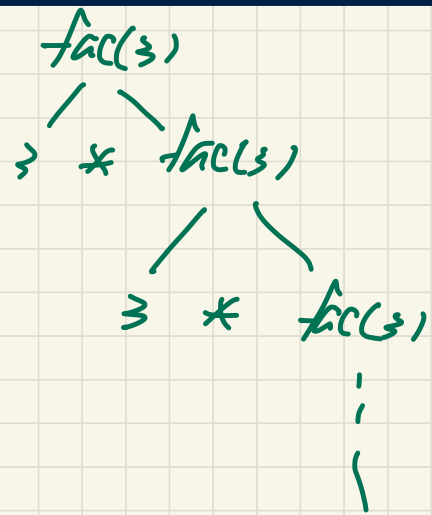

infinite recursion.

StackOverflowException.

# Common Errors of Recursion (2)

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

recursive call not on strictly smaller input.



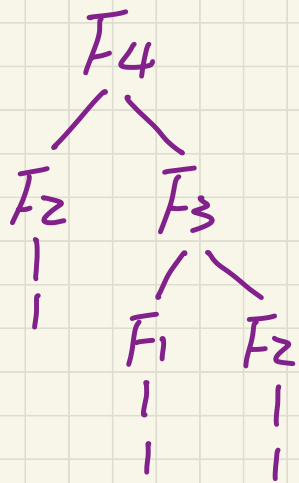

infinite recursion.

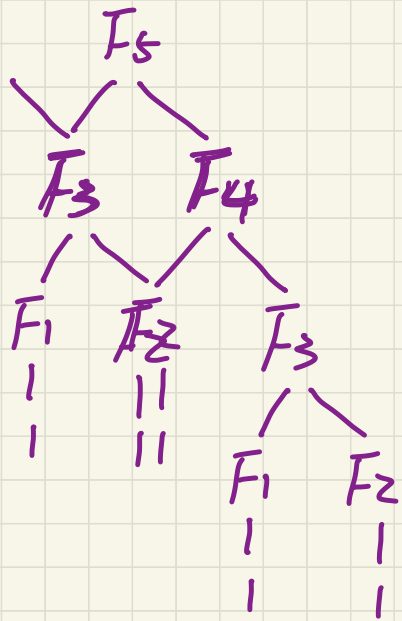
# Recursive Solution: Fibonacci Numbers

1 2 3 4 5 6 7 8 9 10 11

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

$F_1$ $F_2$




base cases.

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

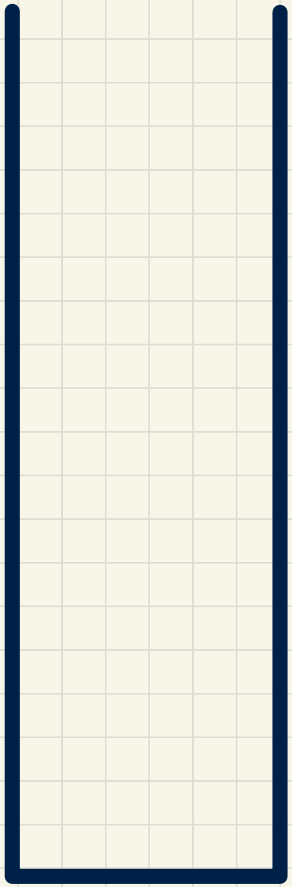
two recursive calls.

# Recursive Solution in Java: Fibonacci Numbers

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

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

**Example**: fib(4)



Runtime Stack

# Use of String

isEmpty("␣␣␣") → false

S → | a (0) | b (1) | c (2) | d (3) |

```
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();
  }
}
```

0123

[0, -1] size: (-1)-0+1 = 0

0-3

1-2

0, 1

2, 3

subs. (3, 2) → ~~empty~~ or IOBE ← exception.

assumption: $i \le j$

S.substring(i, j)

↓

return a string including characters from indices: i to j-1

S → | String | val → |

$[1, 3] \rightarrow \underline{1, 2, 3}$ size: 3

$[1, 3) \rightarrow 1, 2$

$[0, 10] \rightarrow$ size 11

substring$(i, j)$

$\hookrightarrow [i, j)$

$[i, j-1]$

assumption: $a \leq b$

$[a, b]$

size: $b - a + 1$

# Recursions on Strings

## Palindrome

"racecar"

"racecars"

"racecaar"

1st & last chars same

there's a chance the substring, minus 1st & last chars, is a palindrome.

not a palindrome.

already diff → false.

## Reversal

"abcd"

## Number of Occurrences

"abca"

'a'

'b'

# Lecture 23 - Nov 27

## Recursion

***Examples: reverseOf, occurrencesOf***
***Recursion on Arrays: Creating Sub-Arrays***
***Recursion on Arrays: Call by Value***

## Announcements/Reminders

- Today's class: notes template posted
- Lab5 released (deadline: Dec 2; grace period until Dec 9)
  + Required background study: abstract classes & interfaces

# Recursions on Strings

## Palindrome

"racecar"

"racecars"

"racecacar"

1st & last chars same

there's a chance the substring, minus 1st & last chars, is a palindrome.

not a palindrome.

already diff → false.

## Reversal

rev("abcd")

↓

"dcba"

reverse of.

## Number of Occurrences

"abca"

occ("abra", 'a')

'a' 1 + occ("bcb", 'a')

'b' 0 + occ("bcb", 'b')

# Problem: Palindrome

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

recursive case.

if(fC == lC){
return isP(middle);
}
else { return false; }

isP(abcba)

a==a && isP(bcb)

T

b==b && ~~isP(c)~~ true.

T

T.

1) LHS is true → eval RHS

2) LHS is false → skip RHS

word

middle.

0 1 . . . word.length() - 1

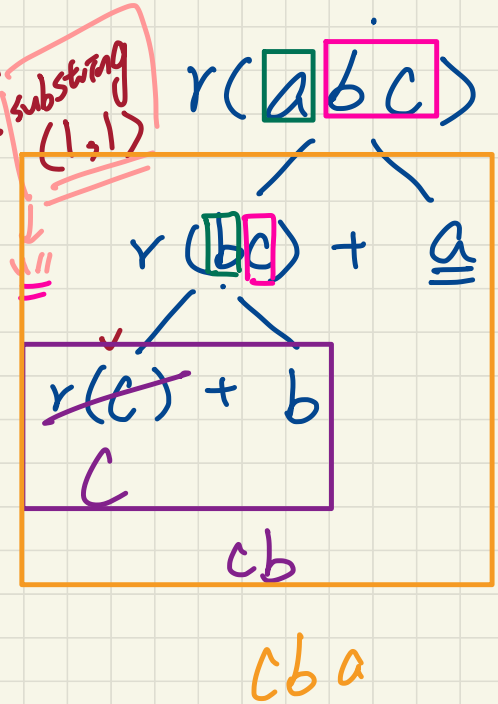
firstC. lastC.

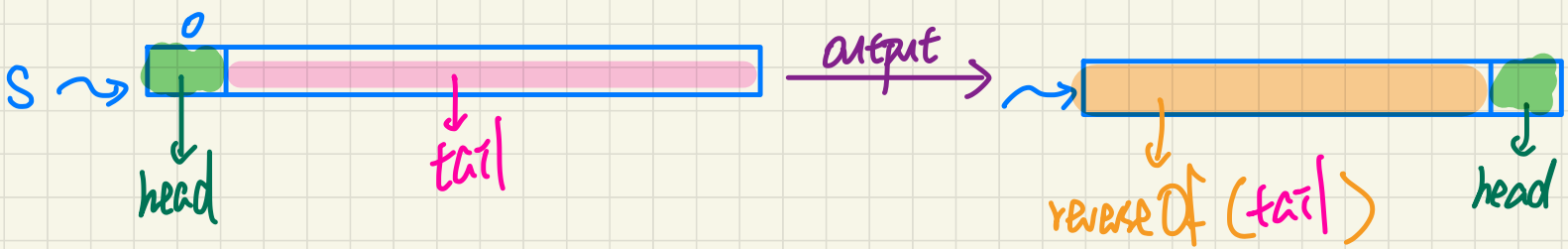
# Problem: Reverse of a String

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

base cases

what if s.length == 1 "c". substring (1, 1)



a b c d

d c b a

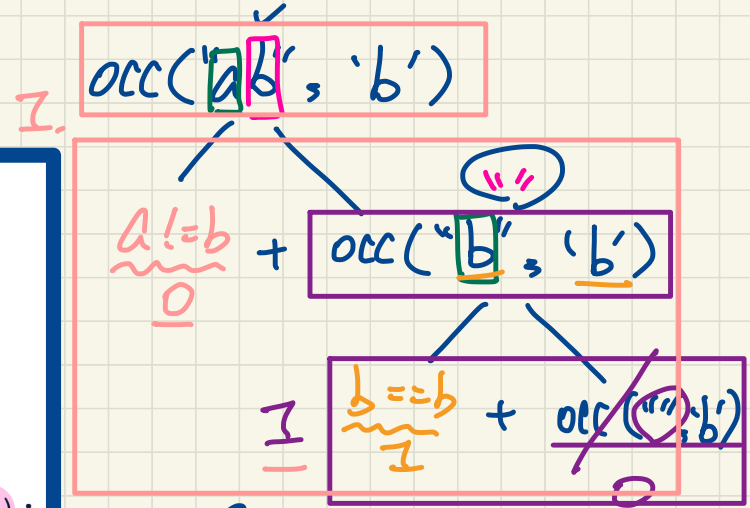
alternative approach for implementing reverseOf.

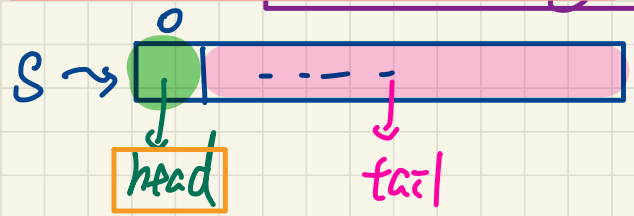
# Problem: Number of Occurrences

```
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.

return head == c ? 1+ occ(tail, c) : occ(tail, c)

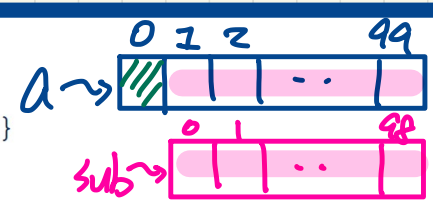



1 if head == c
0 if head != c
+ occ(tail, c).

# Recursion on an Array: Passing new Sub-Arrays

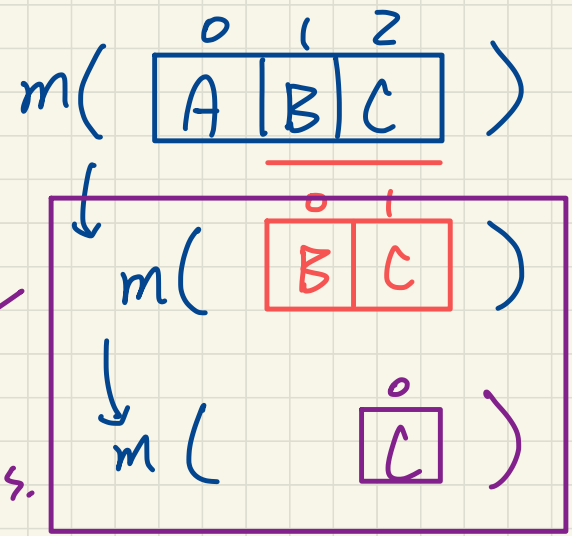
```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```




Say a1 = {}, consider m(a1)

↳ base case #1.

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




loop time & sub-arrays space wasted to create sub-arrays between R.C.s.

# Recursion on an Array: Passing Same Array Reference

"sub-arrays"

range of indices to inspect in array 'a'

```
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) } }
```

3, 2

2, 2

sub-array of size 0

sub-array of size 1

recursive call


* call by value : passing the ref. of 'a' around.

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

a1.length == 0

1st index

last index

from > to

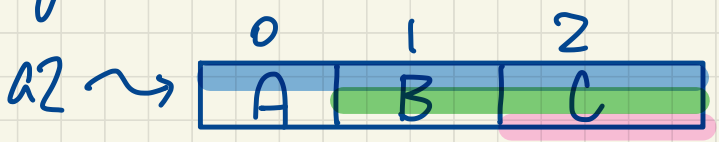
empty range of indices to look at

↳ empty sub-array.

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

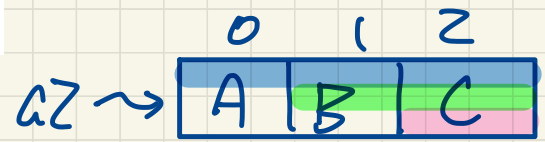
a2.length == 3

2



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

3




m(a2, 0, 2)

↳ input array of length 3

m(a2, 1, 2)

↳ sub-array of length 2

m(a2, 2, 2)

base case (as if sub-array of length 1).

as recursive calls are made, narrow down the intervals.

# Lecture 24 - Dec 2

## Recursion

***Recursion on Arrays: allPositive, isSorted***
***Course Wrap-Up***
***Assigned Readings: Tower of Hanoi***

## Announcements/Reminders

- Today's class: notes template posted
- Lab5 released (deadline: Dec 2; grace period until Dec 9)
  + Required background study: abstract classes & interfaces
- Review session survey active now!
- Exam guide, example questions to be released

universal property

# Problem: Are All Numbers Positive?

$> 0$

↳ 0 is not positive.

Say a = {}

$\forall x \in \phi \cdot P(x) \equiv$ True. ) there's no witness to show some violation

$x > 0$

existential property: $\exists x \in \phi \cdot P(x) \equiv$ False ) there's no witness to show some satisfaction.

$x > 0$

Say a = {4}

T

0 1 2 3

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

T

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

F.

allP( a, 0, 3)

a[0] > 0 && allP( a, 1, 3).

4 > 0

T.

# Problem: Are All Numbers Positive?

public

```
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);
  }
}
```

call to a recursive helper method.

private

# Tracing Recursion: allPositive

Say a = {}

allPositive(a)

allPH(a,0,-1)

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

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

# Tracing Recursion: allPositive

Say a = {4}

allPositive(a)

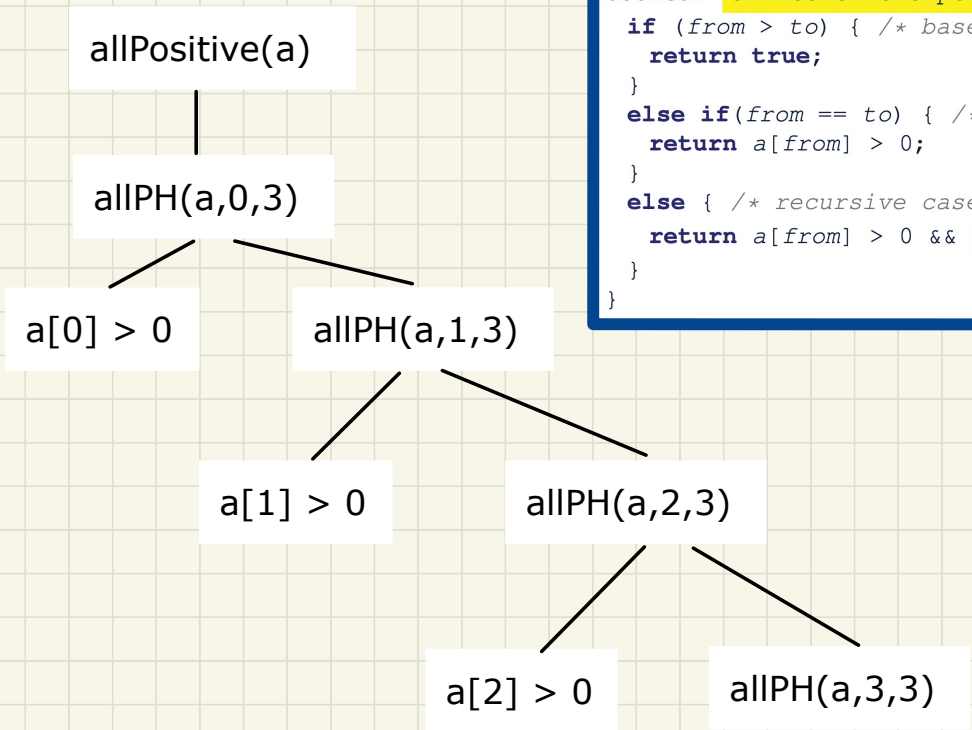
allPH(a,0,0)

a[0] > 0

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

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

# Tracing Recursion: allPositive

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




```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);
  }
}
```

# Tracing Recursion: allPositive

a.length == 4

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

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

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

F. allPositive(a)

F. allPH(a,0,3)

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

5 > 0

T.

1st R.C.

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

3 > 0

T

2nd R.C.

F

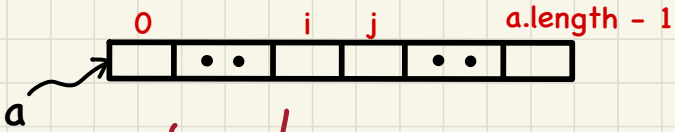
-2 > 0

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

s.t.B: bypass RHS

F

# Sorting Orders of Arrays




not, ¬, !

$\boxed{\text{non}}$-descending

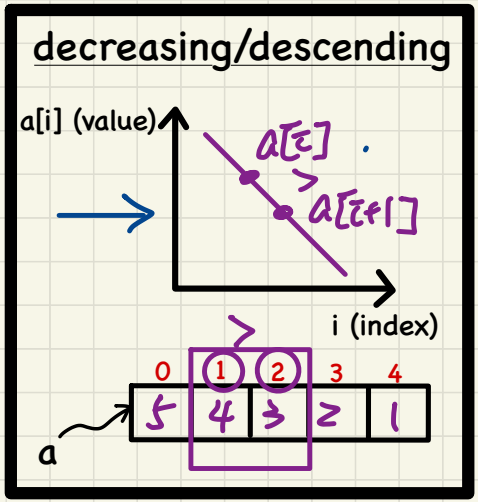
$\triangleq \neg(\text{descending})$

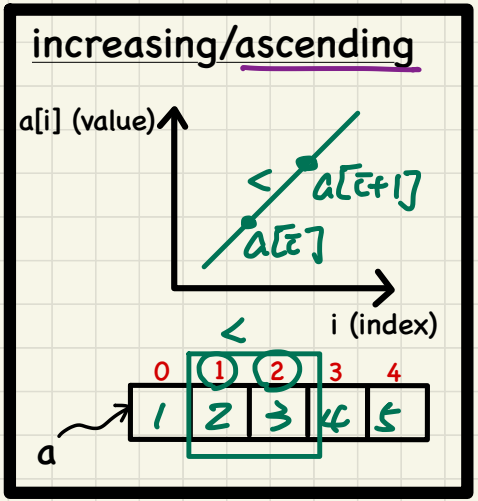
$\equiv \neg(a[i] > a[i+1])$

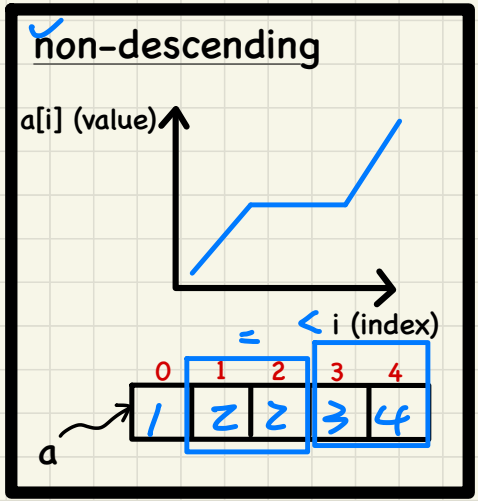
$\equiv a[i] \leq a[i+1]$

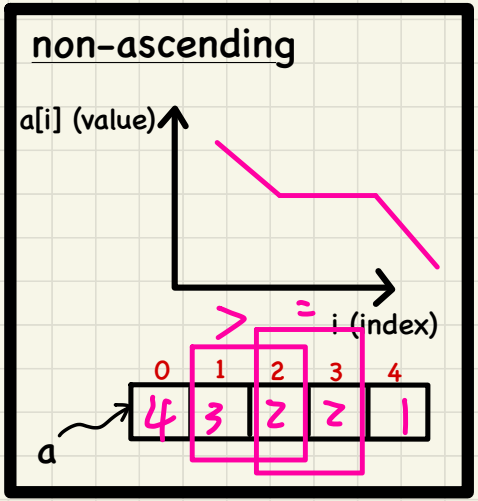
ascending

equal to



# Problem: Are Numbers Sorted?

Say a = {}

true (∵ cannot find violation).

Say a = {4}

true

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

0 1 2 3

true

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

false.

$a[0] \le a[1]$

&& $a[1] \le a[2]$

&& $a[2] \le a[3]$

from

isSorted(a, 0, 3)

$a[0] \le a[1]$ && isSorted(a, from+1, to)

# Problem: Are Numbers Sorted?

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

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

# Tracing Recursion: isSorted

Say a = {}

isSorted(a)

isSH(a,0,-1)

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

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

# Tracing Recursion: isSorted

Say a = {4}

isSorted(a)

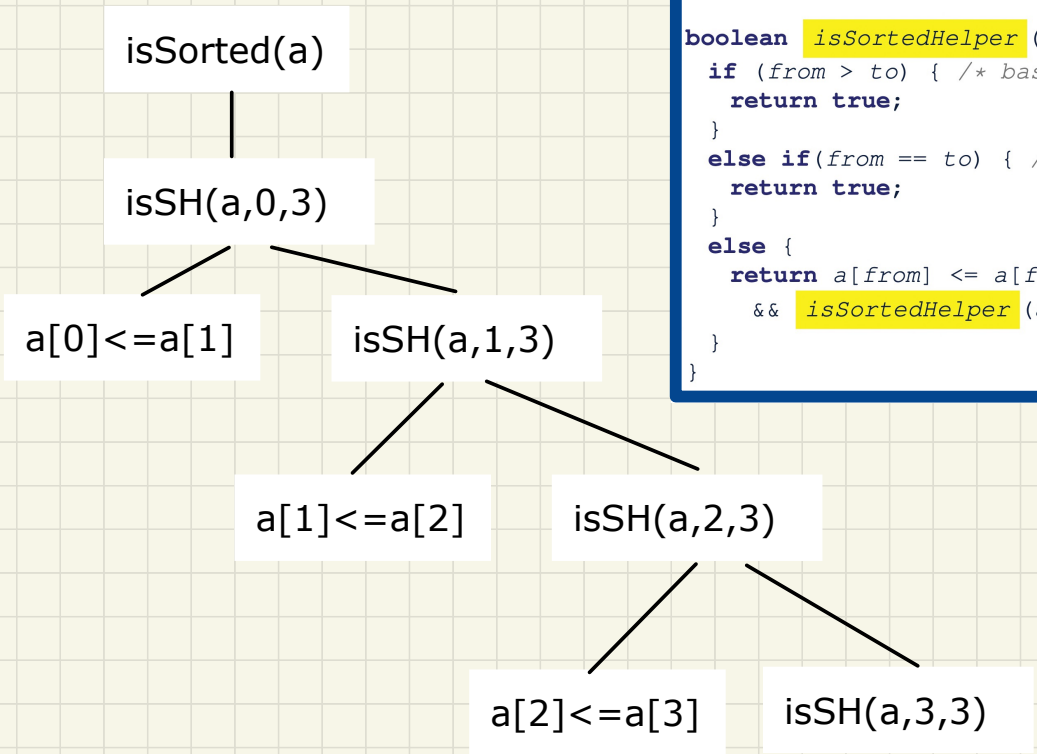
isSH(a,0,0)

**return *true***

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

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

# Tracing Recursion: isSorted

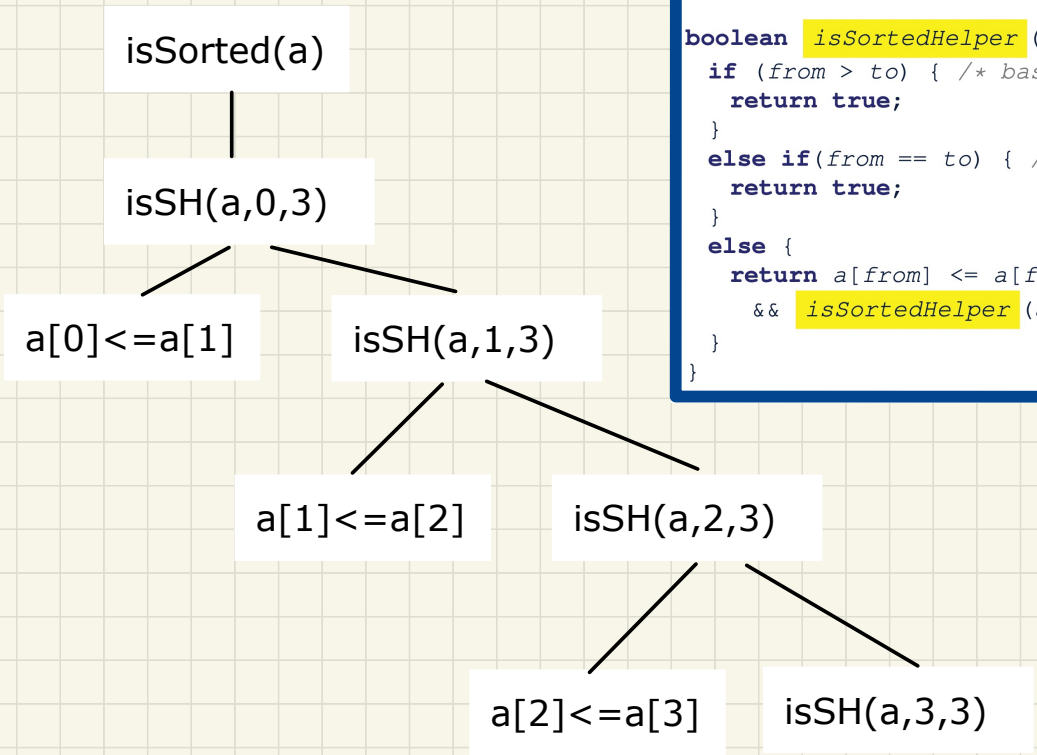
Say a = {3,6,6,7}




```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 = {3,6,5,7}




```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);
  }
}
```

# Exam Info

- When: 9am to 12pm, Tuesday, December 16 (TM Tait)
- Format: Some Multiple Choice & Mostly Written
  1. def.
  2. explanations
  3. tracing.
  4. coding.
- Coverage: **Everything** (lecture materials & labs)
  + slides, iPad notes, code examples
  + resources on recursion
- Even problems that look **challenging** at first are built on the **same** **foundational techniques** you've learned and practiced in **lectures** and **labs**. A **solid, reflective** grasp of the basics will take you **farther** than memorizing examples.
- Restrictions:
  1. q. booklet
  2. answer booklet.
  + No data sheet + No sketch paper (Exam booklet includes it)
- What you should bring:
  + Valid, Physical Photo ID (strict) + Water/Snack

I hope you enjoyed learning with me 😊

All the best to you !