

EECS2030 (Sec. B) Fall 2025

Advanced Object-Oriented Programming

Lecture Notes

Instructor: Jackie Wang

Lecture 1 - Sep 3

Syllabus & Introduction

Professional Engineers: Code of Ethics
Software Development Process

Course Learning Outcomes (CLOs)

CLO1

Implement an Application Programming Interface (API)

CLO2

Test the implementation.

CLO3

Document the implementation

CLO4

Implement aggregations and compositions.

CLO5

Implement inheritance.

CLO6

Use recursion.

CLO7

Implement linked lists.

CLO8

(Informally) prove that recursive algorithms are correct and terminate.

CLO9

(Informally) analyse the running time of (recursive) algorithms.

labs, proj, tests

JUnit.

ref. algos.

method headers

polymorphism
dynamic binding.

2/0/.

FFS 2/0/1, 3/0/1

Progtest

Given

Starter.java

test_1

test_2

⋮

test_n

develop

Java
classes from
scratch
pass tests

17

24

writing code in IDE.

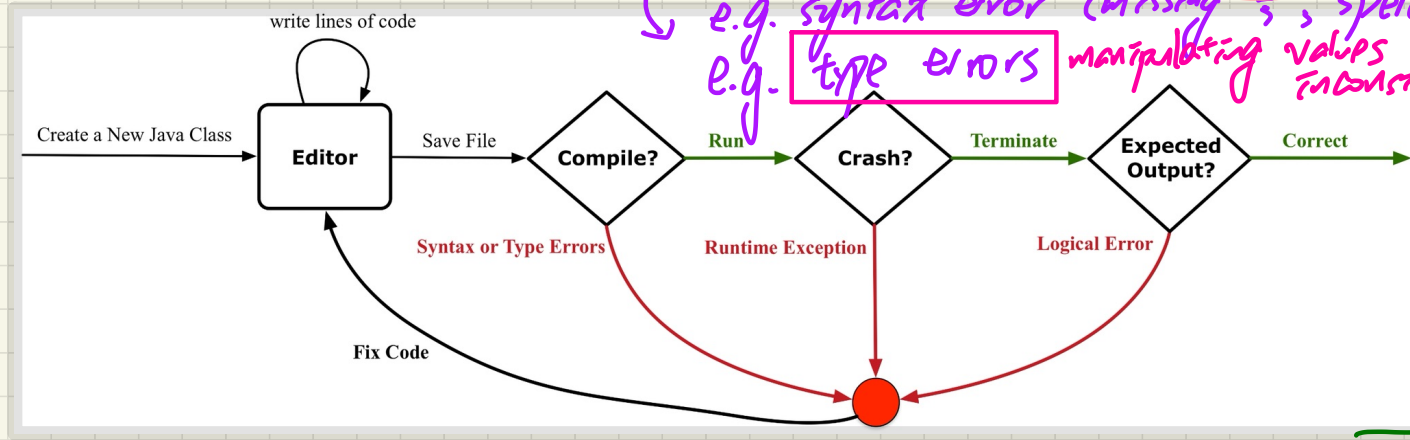
Software Development:

Compile Time

vs. **Run Time**

- assuming code compiles - behaviour.

e.g. syntax error (missing , , spelt "class")
e.g. type errors
manipulating values inconsistently



Type Errors

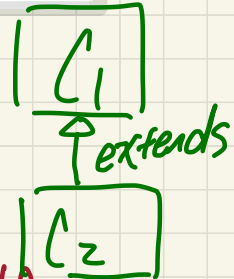
① X `int i = "2030";`

② X `int i = 2030;`
`int j = x;` → undeclared

③

`C1 obj = new C2();`

only compiles if C1 and C2 are related in the right way through inheritance.



Runtime Errors

① Standard

1. ArrayIndexOutOfBoundsException

2. NullPointerException

3. ClassNotFoundException

Inherited.

②. Assertion

Output doesn't meet expectation (JUnit).

Lecture 2 - Sep 8

OOP Review

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

Announcements/Reminders

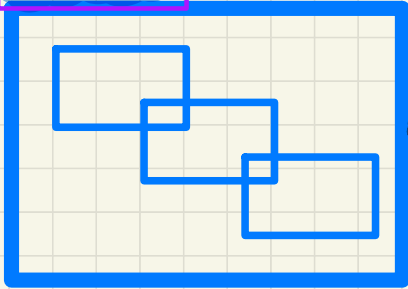
- First Class (Syllabus) recording & notes posted
- Today's class: [notes template](#) posted
- Priorities:
 - + **LabOP1** (tutorial videos + PDFs)
 - + Due: This Tuesday (Sep 8)
- **LabOP2** to be released this Tuesday (Sep 8)
- Lab this Wednesday (Sep 10): Demo on [helper methods?](#)
- Lab next Wednesday (Sep 17):
 - + [≈ 40 min] Mock-up **Programming Test 0**
 - + [≈ 40 min] Q&As (TAs, Jackie)

Separation of Concerns

Slide 6

logic of implementations.

model



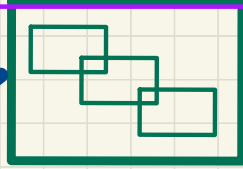
- Classes & Methods
- Methods
 - * constructors
 - * accessors: **return** statements
 - * mutators: **no return** statements
 - * containing **no** print statements

use

*model $\approx$ supplier
junit $\approx$ client*

use

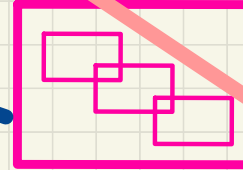
junit_tests



specification (what)

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

console_apps

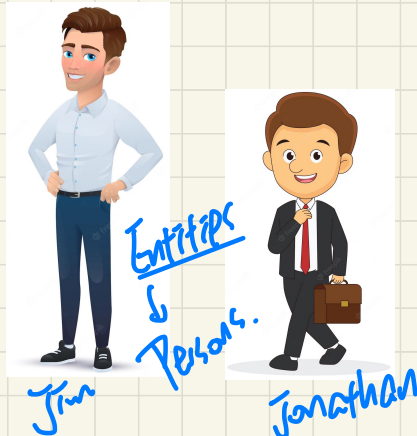
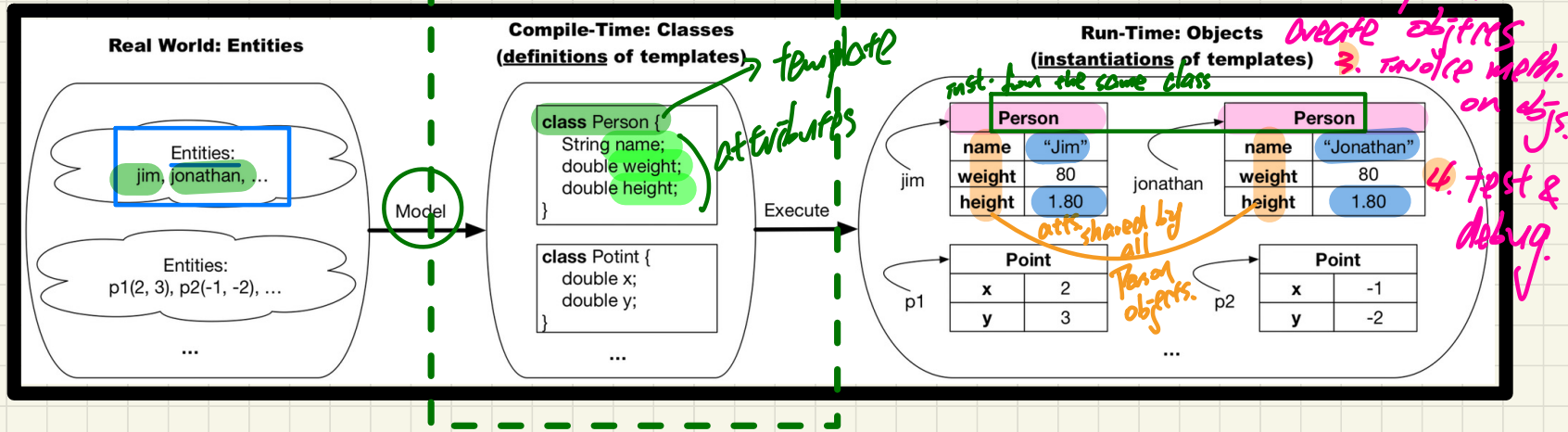


- main method (entry point of execution)
 - * reading inputs from keyboard
 - * calling methods from model
 - * producing outputs to console (print)
 - * containing **no** return statements

discouraged.

Observe-Model-Execute Process

Slide 7



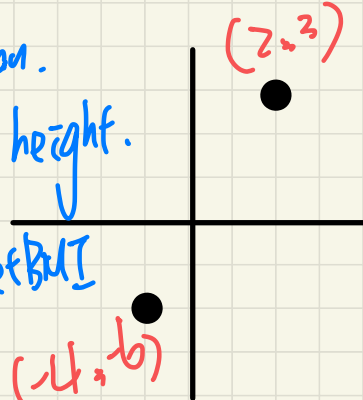
Entities: Jim, Jonathan.

Attributes: weight, height.

Changes: grow taller

Inquiries: getW, getH, getBMI

Template: Person



Entities:

Attributes:

Changes:

Inquiries:

Template:

1. Assume: Everything compiles
2. Instantiate template;
3. create objects on obj.
4. test & debug.

OO

templates

instances

changes

inquiries

(Java, C#)
OOP

classes

objects

mutators/setters

accessors/getters not void.

String getN()
double getW()

void set(...)

- Entities of the same kind share the same set of attributes.

Jonathan, Jim → Person

↳ 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) { ... }
```

```
}
```

↓
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

Jim

Person	
w.	0
h.	173.2

✓

① Person Jim = new Person (173²);

②

Person Jonathan =
new Person (84.5,
185.7);

Jonathan

Person	
w	84.5
h.	185.7

Object Oriented Programming (OOP)

Slide 8

- 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 attr methods
modify the internal workings
without any public/external users realizing it.
(e.g. helper methods)
obj.m1().m2().m3()
chain of method calls.

Slides 9 ~ 47

assigned readings!

Person jim = ... ; * ref. aliasing

Person jon = ... ;
↑ different objects

① jim.getWeight() → 72

② jon.getWeight() → 85

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

* may ref. the same object
④ jon.gainWeight(2) → +2 to jon's weight

⑤ jim.getWeight() → 74

⑥ jon.getWeight()

① vs. ②

③ vs. ④

① vs. ⑤. (not same
" ③)

② vs ⑥
(not same
" ④)

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

compile time. ← static-typed
PL
dynamic-typed python.
PL e.g. 1, 0, 3
{... 7, 2, 3...}

e.g. `int` `i` ;

↓
declared type
for `i`
(p.t.)

↓
at r.t. → store any value
`i` can store a member of `int`.

Lecture 3 - Sep 10

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 today (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.

compiled time. ← static typed

PL dynamic-typed python. e.g. 1, 0, 7, -1, 3 { ... 7, -1, 0, 3 } 1, 2, 3, ...

e.g. `int i;`

`int` is declared type for `i` (P.t.)

`i` at v.t. can store any value that's a member of `int`.

1. a class declared within your proj.
2. external lib. class imported

e.g. `Person p;`

At v.t. the address of some Person object

memory block reserved for storing Person-specific info.

```
Person p = ...;
String s = "230";
p.x = s;
```

i cannot store string address in a var that's meant to store Person obj.

A declared type of some variable x

denotes the set of values that is
allowed to be stored in x .

NullPointerException → runtime error

when context object of some method call is null

undefined address
breakpoint → debugger

e.g. `CO.m().mz().mz()`
(1) may be null
(2) `CO.m()` may be null
→ Say NPE!
(3) `CO.m().mz()` may be null

method call

`obj.m(...)`

Context object

look up the address that's stored in the i.o.

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

if the mem. block doesn't exist (i.e. address is null)
→ NPE!

Q. How many possible causes for NPE? 3

(3)

(2)

(1)

10

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

Parameters

vs.

Arguments

def. of methods

use/call methods

parameters

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

Template Definition

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

i = p2 ✓

* each "new" indicates a new block of memory, with a default address, to memory

Constructors not using this Keyword le created (sequence of bytes)

```
public class Person {  
    /*  
    * Attributes.  
    * Person instances have the same attribute names.  
    * Person instances have specific attribute values.  
    */  
    double weight;  
    double height;  
    /*  
    * Constructors  
    */  
    public Person() {  
           
    }  
    public Person(double newWeight, double newHeight) {  
        weight = newWeight;  
        height = newHeight;  
    }  
}
```

model

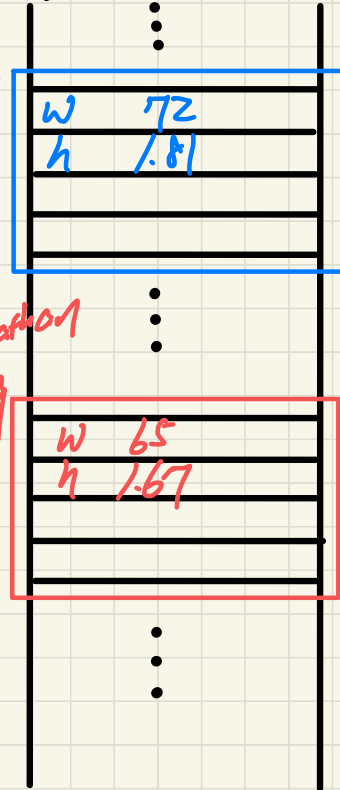
```
@Test  
public void test_1() {  
    ** Person jim = new Person(72, 1.81);  
    Person jonathan = new Person(65, 1.67);  
    assertTrue(jim != jonathan); true  
    assertFalse(jim == jonathan); false  
    assertNotSame(jim, jonathan);  
    assertNotEquals(jim, jonathan);  
}
```

JUnit

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

console

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



(e.g. assertTrue , assertFalse)
JUnit assertions : pass vs. fail

(e.g. $\text{jim} == \text{jon}$,
 $\text{jim} != \text{jon}$)
boolean expressions : true vs. false

assertTrue ($\frac{\text{boolean exp}}{\downarrow}$)

- true → pass

- false → fail

assertFalse ($\frac{\text{boolean exp}}{\downarrow}$)

- true → fail

- false → pass

$\text{AssertSame}(\underline{\text{obj1}}, \underline{\text{obj2}})$

↳ $\text{AssertTrue}(\text{obj1} == \text{obj2})$

↳ $\text{AssertFalse}(\text{obj1} != \text{obj2})$

$\text{AssertNotSame}(\text{obj1}, \text{obj2})$

↳ $\text{AssertFalse}(\text{obj1} == \text{obj2})$

↳ $\text{AssertTrue}(\text{obj1} != \text{obj2})$

Use of **Accessors** vs. **Mutators**

returning non-void
returning void.

Slide 48

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

- Calls to **mutator methods** *cannot* be used as values. *void*

- e.g., `System.out.println(jim.setWeight(78.5));` ×
- e.g., `double w = jim.setWeight(78.5);` × *void*
- e.g., `jim.setWeight(78.5);` ✓

- Calls to **accessor methods** *should* be used as values.

- e.g., `jim.getBMI();` ✓ *not useful* *double*
- e.g., `System.out.println(jim.getBMI());` ✓
- e.g., `double w = jim.getBMI();` ✓ *double*

× *completes but not useful*

In-Lab Demo - Sep 10

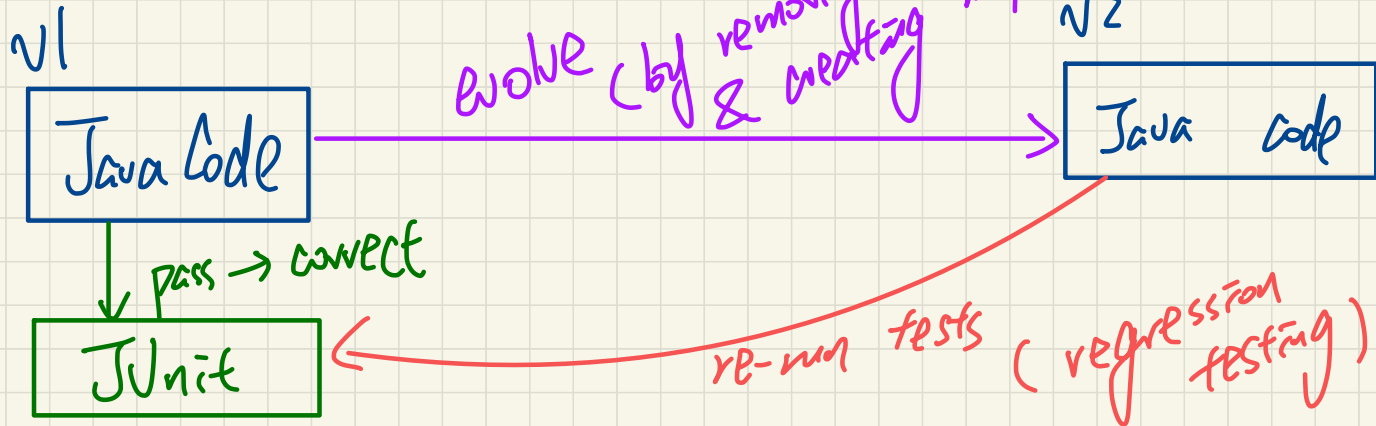
Helper Methods

Code Smell
Refactoring
Breakpoints & Debugger

Factoring Out

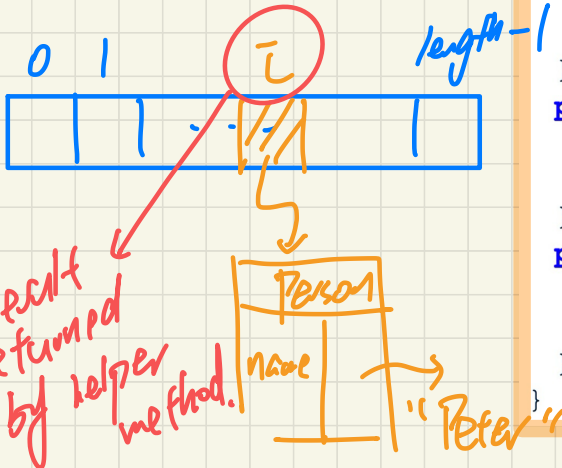
$$x \cdot y + x \cdot z$$

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



PersonCollector: without a Helper Method

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



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

searching repeated.

PersonCollector: with a Helper Method

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

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

break $\Rightarrow$ continue,

return prematurely
 $\downarrow$
middle of loop

~~spaghetti~~
code

$\hookrightarrow$ valid

$\hookrightarrow$ error-prone for bigger, more complicated project

Lecture 4 - Sep 15

OOP Review

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

Announcements/Reminders

- Today's class: [notes template](#) posted
- Priorities:
 - + **LabOP2** (tutorial videos + PDFs)
 - + Due: This Tuesday (Sep 16)
- **Lab1** to be released this Tuesday (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

Use "this" keyword
to denote the context
object (which allows you to ref. attributes).
Question

```
public class Person {  
    /*  
     * Attributes.  
     * Person instances have the same attribute names.  
     * Person instances have specific attribute values.  
     */  
    double weight;  
    double height;  
  
    /*  
     * Constructors  
     */  
    public Person() {  
  
    }  
  
    public Person(double newWeight, double newHeight) {  
        this.weight = newWeight; weight  
        height = newHeight; height  
    }  
}
```

model

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

clashes with the attribute

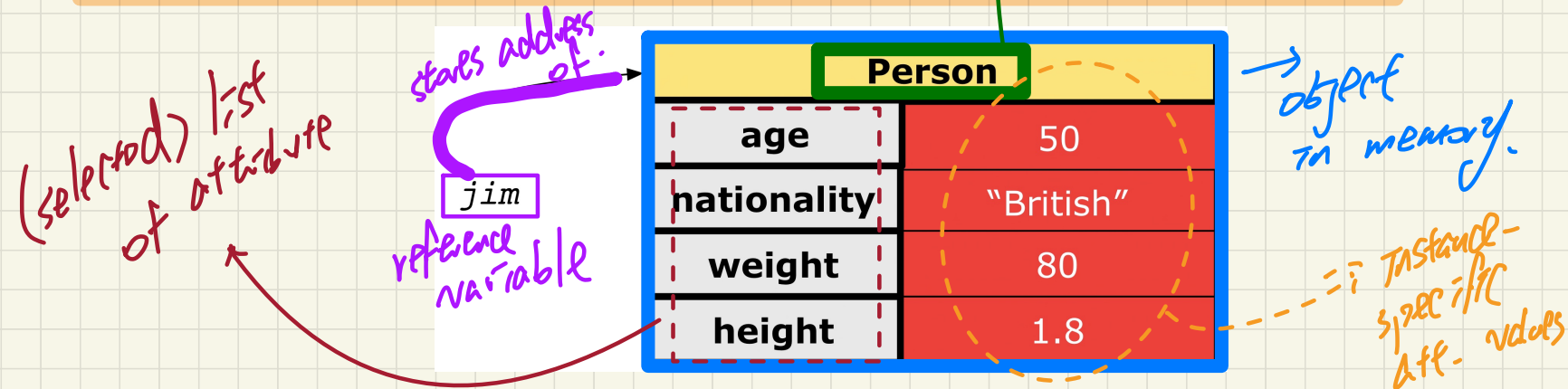
(variable shadowing).

not referring to the intended attribute (it's shadowed by the parameter name)

Tracing OO Code: Visualizing Objects

To visualize an object:

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



to different objects.

Effects of Creating New Objects

```
public class Person {  
    /*  
     * Attributes.  
     * Person instances have the same attribute names.  
     * Person instances have specific attribute values.  
     */  
    double weight;  
    double height;  
  
    /*  
     * Constructors  
     */  
    public Person() {  
  
    }  
  
    public Person(double weight, double height) {  
        this.weight = weight;  
        this.height = height;  
    }  
}
```

model

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

72
1.81
72
1.81
Jonathan

Person	
w.	65
h.	1.67

```
@Test  
public void test 1() {  
    Person jim = new Person(72, 1.81);  
    Person jonathan = new Person(65, 1.67);  
    assertTrue(jim != jonathan);  
    assertFalse(jim == jonathan);  
    assertNotSame(jim, jonathan);  
    assertNotEquals(jim, jonathan);  
}
```

JUnit

* Person jim = new Person(72, 1.81);
① ② ③

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

Jim

②

this

Person	
weight	72
height	1.81

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

Accessors/Getters vs. Mutators/Setters

```
public class Person {  
    /*  
     * Attributes.  
     * Person instances have the same attribute names.  
     * Person instances have specific attribute values.  
     */  
    double weight;  
    double height;  
  
    /* Accessors/Getters */  
    public double getBMI() {  
        double bmi = this.weight / (this.height * this.height);  
        return bmi;  
    }  
  
    /* Mutators/Setters */  
    public void gainWeightBy(double amount) {  
        this.weight = this.weight + amount;  
    }  
}
```

Handwritten annotations for the Person class:

- Blue arrows point to `getBMI()` and `gainWeightBy()`.
- Orange arrows point to `this.weight` and `this.height` in the `getBMI()` method.
- Blue and orange text "jim" and "jonathan" are written next to the `this` keyword in the `getBMI()` method.
- Blue and orange text "jim" and "jonathan" are written next to the `this` keyword in the `gainWeightBy()` method.

Handwritten "jim" with an arrow pointing to the table.

Person	
w.	72 75
h.	1.81

Handwritten "jonathan" with an arrow pointing to the table.

Person	
w.	65 68
h.	1.67

```
@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);  
}
```

Handwritten annotations for the test code:

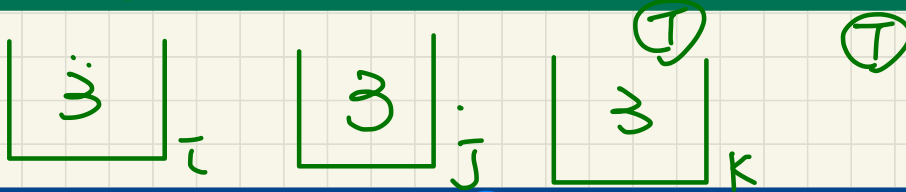
- Blue circles around `72` and `1.81` in `new Person(72, 1.81)`.
- Orange circles around `65` and `1.67` in `new Person(65, 1.67)`.
- Blue circles around `jim` and `jonathan` in the `assertEquals` calls.
- Blue circles around `3` in `gainWeightBy(3)`.
- A pink box highlights the final two `assertEquals` calls.

Copying Primitive vs. Reference Values

Slide 50

Primitive

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

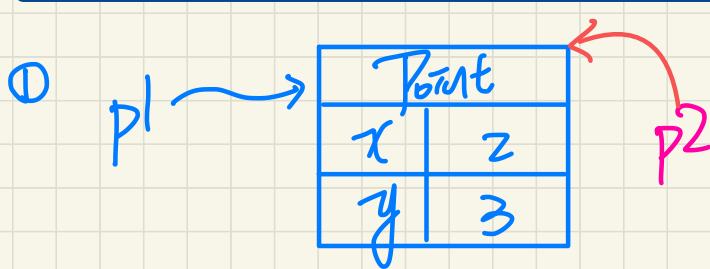


```
Point p1 = new Point(2, 3);  
Point p2 = p1; System.out.println(p1 == p2);  
Point p3 = new Point(2, 3);  
System.out.println(p3 == p1 || p3 == p2);  
System.out.println(p3.x == p1.x && p3.y == p1.y);  
System.out.println(p3.x == p2.x && p3.y == p2.y);
```



Reference

* Copy the ref./add. in `p1` and store it into `p2`.



** `p1` and `p2` store the same address
↳ they're aliases of the same object.

Copying Primitive Values

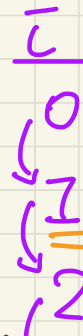
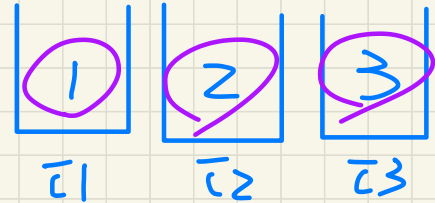
Slide 51

```
int i1 = 1;  
int i2 = 2;  
int i3 = 3;
```

stores the starting add. of an arry

```
int[] numbers1 = {i1, i2, i3};  
int[] numbers2 = new int[numbers1.length];  
for(int i = 0; i < numbers1.length; i++) {  
    numbers2[i] = numbers1[i];  
}
```

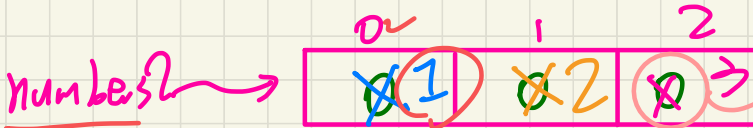
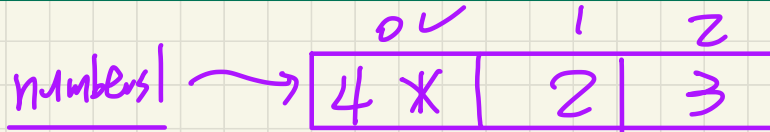
```
numbers1[0] = 4;  
System.out.println(numbers1[0]);  
System.out.println(numbers2[0]);
```



ns2[0] = ns1[0]

ns2[1] = ns1[1]

ns2[2] = ns1[2]



3 < ns1.length
3 < 3
3 < 3
no iteration

default value for int, not null !!

Lecture 5 - Sep 17

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
- Lab this Wednesday (Sep 17):
 - + Please go to your enrolled session.
 - + [≈ 40 min] Mock-up **Programming Test 0**
 - + [≈ 40 min] Q&As (TAs, Jackie)

Copying Reference Values: Aliasing

```

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

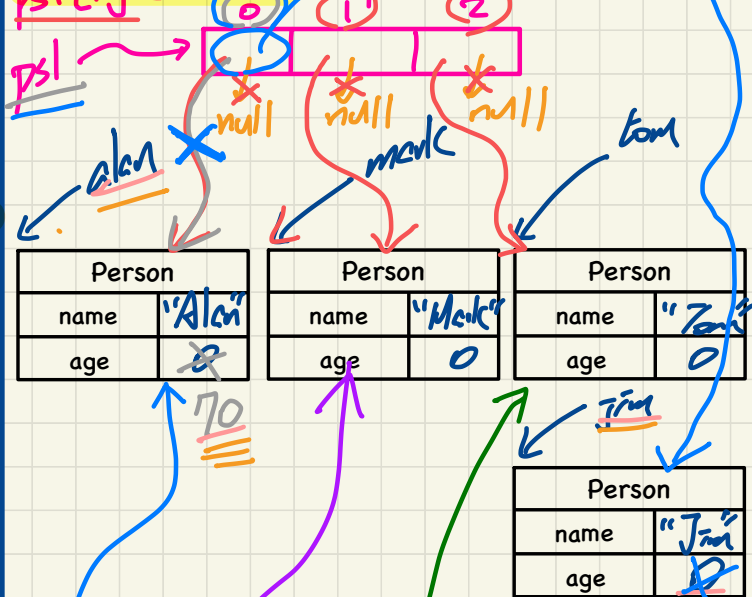
```

* `Person[] ps1 = new Person[3];`

`ps1[0] = alan;`

`ps1[1] = mark;`

`ps1[2] = tom;`

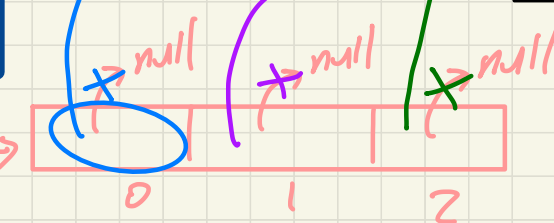


Person	
name	"Alan"
age	0

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75



Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Person	
name	"Jim"
age	75

Person	
name	"Mark"
age	0

Person	
name	"Tom"
age	0

Copying Arrays

Person[] ps1;

Person[] ps2; ①

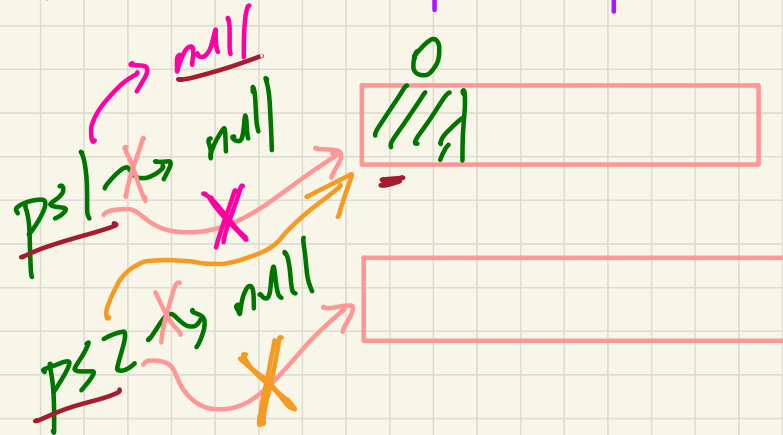
ps1 = new ...

ps2 = new ... ②

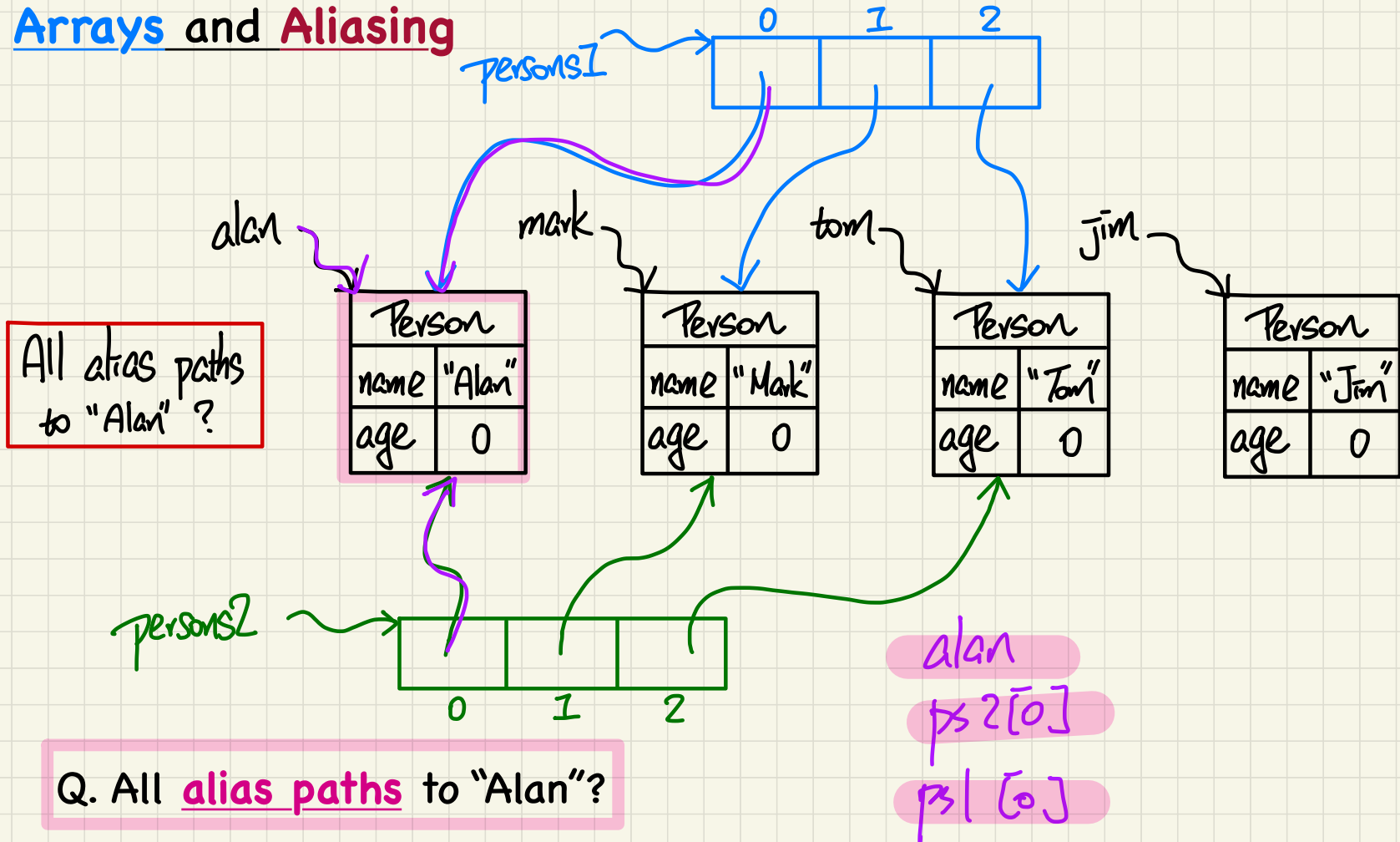
✓ ps2 = ps1; *creating alias at the array level.* ③

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



Managing Account IDs: Manual

Slide 75

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

goal: id's among
Account objects are
unique.

explicit parameter,
presumably should not
clash with
existing id's.

Goal:

Auto. ID

generation &
management

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

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; // *non-static variable (attribute)  
    static int g = 0; // **static variable (attribute)  
    public Counter() {  
        this.l = 0;  
    }  
    public int getLocal() {  
        return this.l;  
    }  
    public void incrementLocal() {  
        this.l++;  
    }  
    public void incrementGlobal() {  
        g++;  
    }  
}
```

Annotations for Counter class:

- `private int l;`: non-static variable (attribute)
- `static int g = 0;`: static variable (attribute)
- `this.l = 0;`: non-static var init. as soon as its decl.
- `return this.l;`: non-static var
- `g++;`: static var

```
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 ++;  
    }  
}
```

Annotations for CounterTester class:

- `Counter c1 = new Counter();`: ①
- `Counter c2 = new Counter();`: ②
- `c1.incrementLocal();`: ③
- `c2.incrementLocal();`: ④
- `c1.incrementGlobal();`: ⑤
- `c2.incrementGlobal();`: ⑥
- `Counter.g = Counter.g + 1;`: ⑦

Diagram illustrating the state of Counter objects and the global variable:

- Object c1: local variable l = 1, global variable g = 1.
- Object c2: local variable l = 1, global variable g = 2.
- Global variable g: 0.

Accessing static variables: ClassName.nameOfStaticVar

Lecture 6 - Sep 22

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.
- This Wednesday's class:
 - + We'll finish at about 12:30.
 - + 20 minutes to be covered in Section G's recording

Alt. String `id_str = "Id" + id;` Managing Account IDs: Automatic

Slides 76 - 77

```
class Account {  
    private static int globalCounter = 1;  
    private int id; String owner;  
    public Account(String owner) {  
        this.id = globalCounter;  
        globalCounter ++;  
        this.owner = owner; } }  
// acc1  
// acc2
```

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

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

①

Account	
gc*	*

~~2~~
3

acc1
c.o.

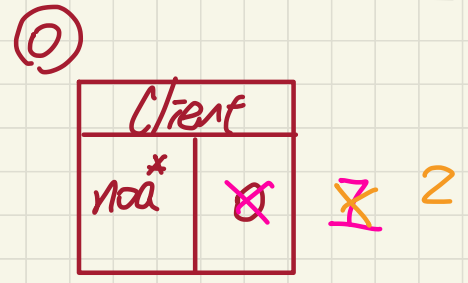
Account	
id	1
owner	"Jim"

acc2
c.o.

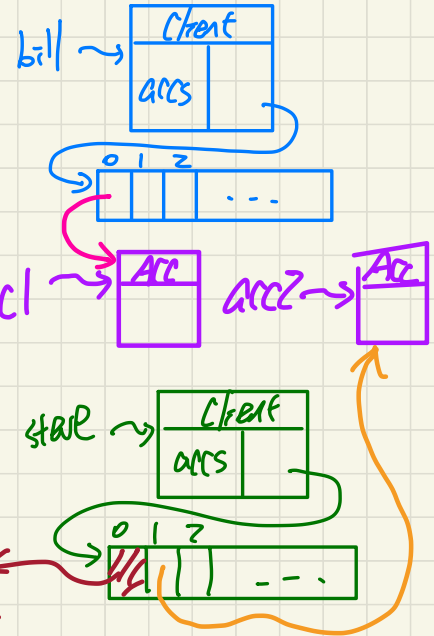
Account	
id	2
owner	"Jeremy"

Misuse of Static Variables

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



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



can. addAccount(acc3); →

0	
1	2

skipped & wasted!

Use of Static Variables: Common Error

Slides 80 - 82

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

Compilation Error: non-static var. used in static context.

Expected Usage: Bank.getInfo.

not a context object → Bank.branchName X

not an object → non-static

Fix 1

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

↓ remove the usage of non-static var
↳

return # branches only.

Fix 2

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

↓ ok to use a static var here.

↳ not good design : branch name should be instance-specific.

Example: Reference to **this**

Slide 59 - 61

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 */ }  
    }  
}
```

when a person is created, spouse == null as default.

*

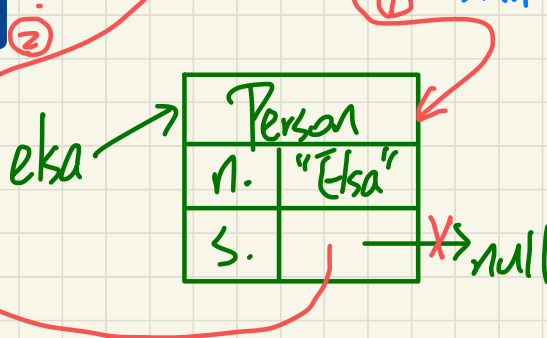
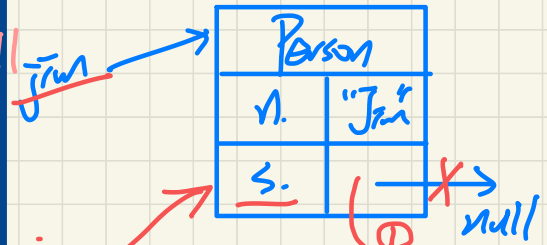
this.spouse == null

~~other.spouse == null~~

C.O. (this)

① jim.spouse = elsa;

② elsa.spouse = jim;
other. C.O. (this)



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

C.O.

C.O.

Math

(conjunctions
and) $\wedge$

(disjunctions
or) $\vee$

(negation,
not) $\neg$

Programming



short-circuit
evaluation

!

Exercise

illegal $\equiv$! (legal)

De Morgan

$\neg (p \wedge q) \equiv \neg p \vee \neg q$ this == null
 $\neg (p \vee q) \equiv \neg p \wedge \neg q$ other == null

```
public void marry(Person other) {
```

```
    if ( * proposed marriage is illegal ) { /* error */ }
```

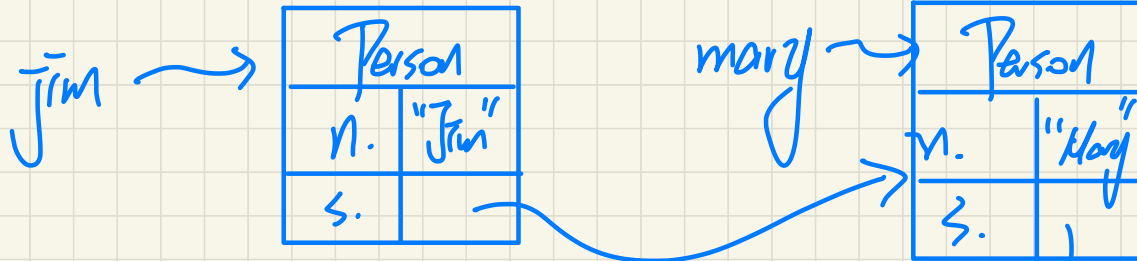
```
    else {
```

①

②

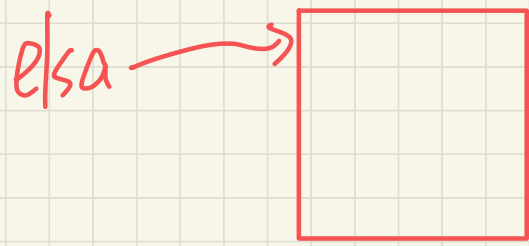
```
    }
```

```
}
```



Jim.marry(mary);

Jim.marry(elsa);



Reference-Typed Return Values

Slide 53

```
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;  
    }  
}
```

```
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

Slide 56 - 58

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

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

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

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

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

Exercise

Lecture 7 - Sep 24

OOP Review

Caller vs. Callee

Tracing Chain of Method Calls via a Stack
Catch-or-Specify Requirement

Announcements/Reminders

- Today's class: notes template posted
- Priorities:
 - + Review Lab0
 - + Complete Lab1; Due: Next Tuesday (Sep 30)
- Today's class:
 - + We'll finish at about 12:30.
 - + 20 minutes to be covered in Section G's recording

A method that calls another method

Caller

vs.

Callee

method that's called by another method

- 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: C1.m1

callee: C2.m2

not indicating that m1 is static

EXERCISE
Make C1.m1 a caller in another context.

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

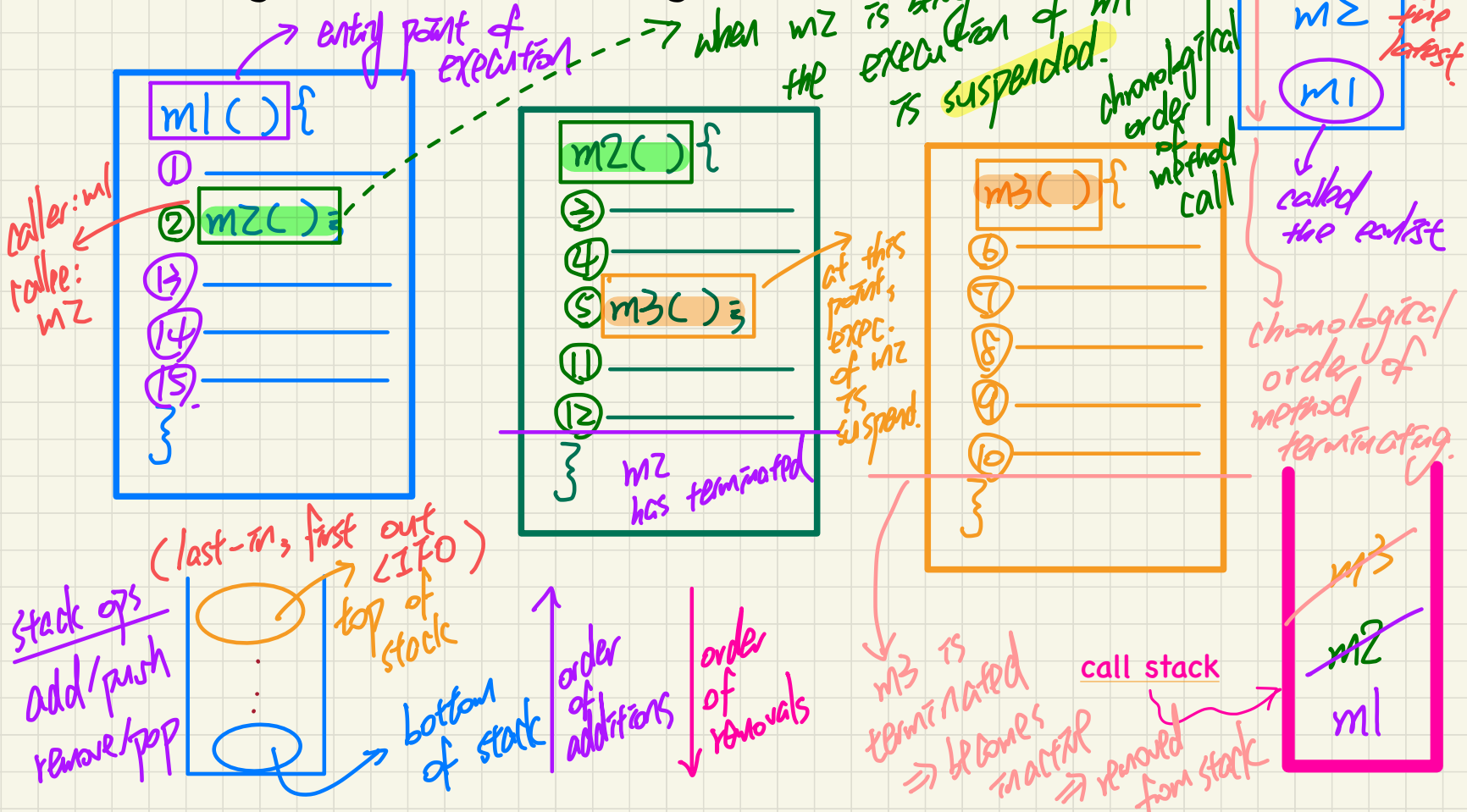
↓ YES. Make C1.m1 a callee in another context

(Alt 1) class C1 {
 void m3() { this.m1(); }
}

(Alt 2) class C3 {
 void m1() { C1 o = new C1();
 o.m1();
 }
}

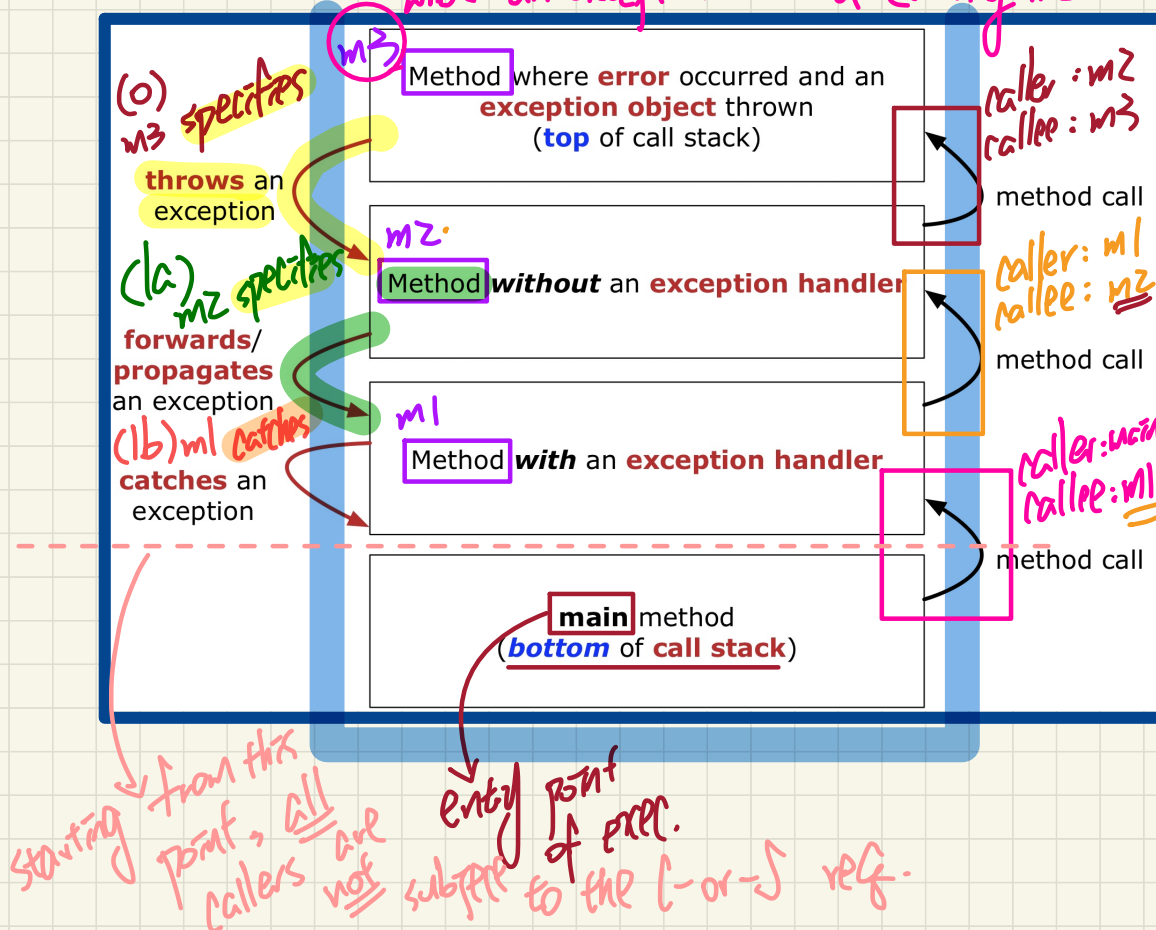
C1.m1 is a callee
o.m1()

Visualizing a Call Chain using a Stack



What to Do When an Exception is Thrown: Call Stack

where an exception occurred (causing the call stack to stop growing)



Catch-or-Specify Ref.

(o) Where an exception is thrown, just specify and propagate to the next caller.

(la) a caller may specify an exception (prog. from its callee)

(lb) a caller may catch an exception (prog. from its callee)

Catch-or-Specify Requirement

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

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

≈ parameter decl
store the add.ress of the caught exception

exception to be caught

try {

handling the caught excep.
catch(...){...}
catch(...){...}

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

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

part of the method header

Example: To Handle or Not To Handle?

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

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

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

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

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

Version 1:

Handle it in B.mb

Version 2:

Pass it from B.mb and handle it in Tester.main

Version 3:

Pass it from B.mb, then from Tester.main, then throw it to the console.

call
stack

A.ma
B.mb
Tester.main

Lecture 8 - Sep 29

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() {}

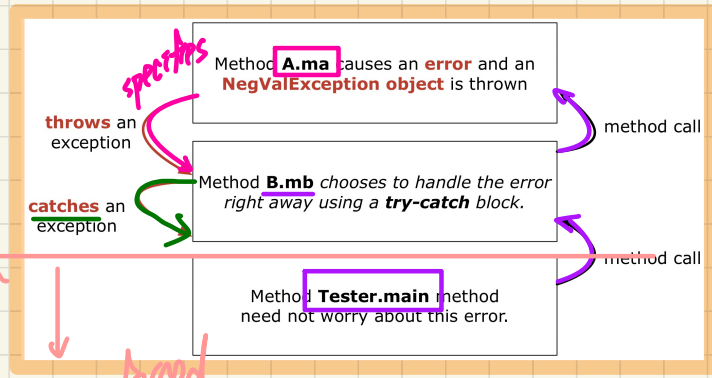
    void ma(int i) throws NegValException {
        5-4
        if(i < 0) {
            ⑥ ⑦
            println("abnormal exec of A.ma");
            throw new NegValException("Neg Val: " + i);
        }
        else {
            ⑦
            println("normal exec of A.ma");
        }
    }
}
```

```
class B {
    B() {}

    void mb(int i) {
        ④ ⑤
        A oa = new A();
        try {
            ⑤ ⑥
            oa.ma(i);
        } catch (NegValException nve) {
            ⑦
            println("From B.mb: Calling A.ma caused NVE.");
        }
    }
}
```

```
class Tester {
    main(...) {
        ① ②
        int i;
        ③ ④
        B ob = new B();
        ob.mb(i);
        ④ ⑤
        println("From Tester.main: After calling B.mb.");
    }
}
```

~~A.ma~~
~~B.mb~~
Tester.main



C-or-
req
no longer entered

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

YES ∵ A.ma specifies the exception

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

no ∵ B.mb catches the exception.

normal exe of A.ma

From B.mb:
Calling A.ma did not...

From Tester.main

abnormal excc of A.ma

From B.mb:
Calling A.ma caused...

From Tester.main ...

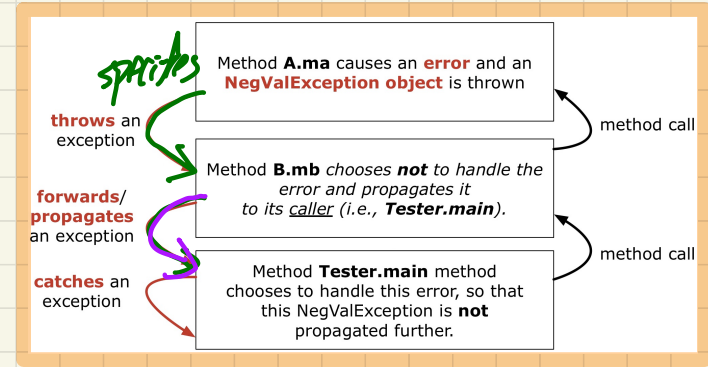
Version 2: B.mb Specifies,

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

Tester.main Catches

```
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.");  
        }  
    }  
}
```



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

YES. B.mb specifies NVE.

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

YES. Tester.main specifies NVE.

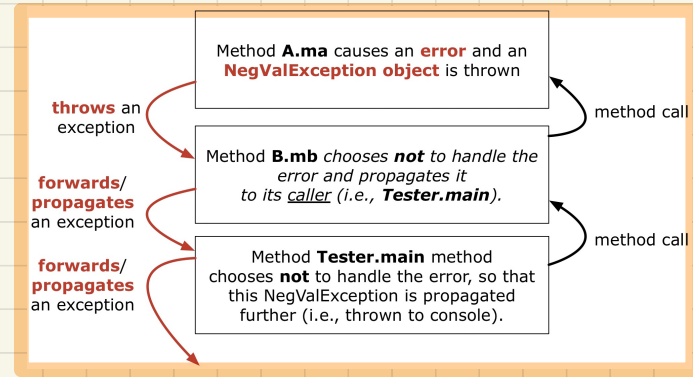
Version 3: B.mb Specifies,

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

Tester.main Specifies

```
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; = -10;  
        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. B.mb specifies NVE.

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

YES. T.m specifies NVE.

Catch-or-Specify Requirement: Call Stack

Q1. Origin of Exception:

$C1.m1$

Q2. Methods subject to CoS Req:

$\bar{\tau}$ methods ($\bar{\tau}-1$ methods specify $\bar{\tau}$ method catches)

Q3. Methods free from CoS Req:

$n-\bar{\tau}$ methods.

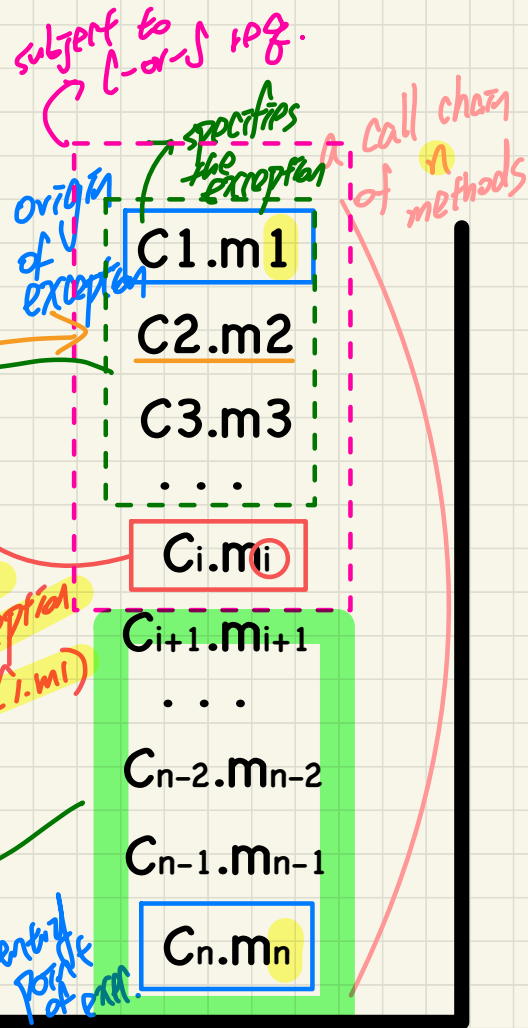
Q4. Extreme Case 1 (exception handled earliest):

$C2.m2$ catches $\tau\epsilon$

Q5. Extreme Case 2 (exception never handled):

No method catches
(All methods specify)

$C_{i+1}.m_{i+1}$
 $\hookrightarrow$
 $C_n.m_n$
($n-\bar{\tau}$ methods)



Lecture 9 - Oct 1

Exceptions

Execution Flows: Normal vs. Abnormal
Examples: Circle, Bank, parseInt

Announcements/Reminders

- Today's class: [notes template](#) posted
- Priorities:
 - + **Lab1** solution video to be released
 - + **Lab2** to be 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 does not occur

end of try

bypassed

When the exception does not occur

*bypassed
∴ exception
did not occur.*

Abnormal Flow of Execution

```
① .. /* before, ouside try-catch block */
try {
  ② o.m(...); /* may throw SomeException */
  X ... /* rest of try-block */
}
catch (SomeException se) {
  ④ ... /* rest of catch-block */
}
⑤ .. /* after, ouside try-catch block */
```

exception occurred!

*bypassed
∴ exception
occurred.*

When the exception occurs

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

Scenario: Caller chooses to specify/propagate 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 did not occur

When the exception does not occur
→ exception did not disrupt the flow of exec

Abnormal Flow of Execution

caller

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

caller

throw

C2.m

C1.m1

exception occurred

C1.m1 terminates abnormally

exception thrown disrupted the flow.

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; }  
}
```

✓ true
~~false~~
IRIV

~~-5~~
✓

10 Exercise
① 23
② -5, -10, 46

Console

Enter a radius:

-5

Try again

Enter a radius:

Circle with area ...

~~Circle.setR~~
CLZ.main

```
public class CircleCalculator2 {  
    public static void main(String[] args) {  
        ① Scanner input = new Scanner(System.in);  
        ② boolean inputRadiusIsValid = false;  
        ③ while (!inputRadiusIsValid) {  
            ④ System.out.println("Enter a radius:");  
            ⑤ double r = input.nextDouble();  
            ⑥ Circle c = new Circle();  
            ⑦ try { c.setRadius(r); }  
            ⑧ catch (InvalidRadiusException e) { print("Try again!"); }  
            ⑨ inputRadiusIsValid = true;  
            ⑩ System.out.print("Circle with radius " + r);  
            ⑪ System.out.println(" has area: " + c.getArea()); }  
    }  
}
```

! true →
false exit.

IRE thrown
IRE not thrown

⑩ X

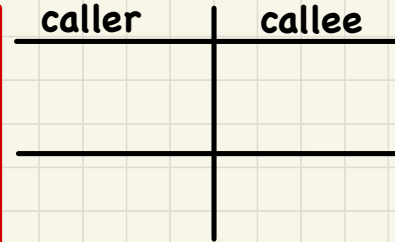
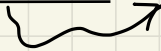
Error Handling via Exceptions: Circles (Version 1)

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

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

Caller?
Callee?

call stack



Test Case 1:

User enters 10

Test Case 2:

User enters -5

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

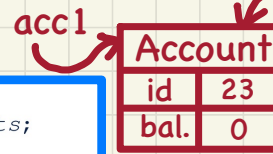
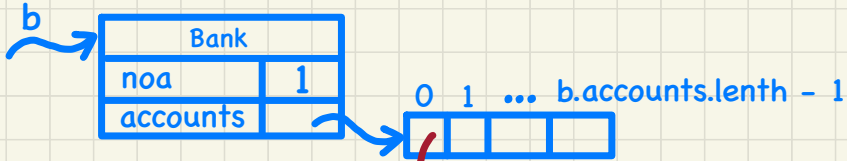

Error Handling via Exceptions: Banks

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

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

```
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 */  
    }  
}
```

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



Account.withdraw
Bank.withdraw
BA.main

accounts[i].withdraw
Account

Test Case:

User enters **-5000000**

Exercise

More Example: Multiple Catch Blocks

Assumption

All the customized exceptions are unrelated to each other.

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

Handwritten notes:

- ① *SM*
- ② *SM*
- ③ *SM*
- ④ *SM*
- throw ITE* (pointing to ②)
- may throw ITE* (pointing to ③)
- may throw NRE* (pointing to ④)

Test Case 1:

a: -5000000

r: 23

Test Case 2:

a: 100

r: -5

→ order of catch blocks does not matter.

More Example: Parsing Strings as Integers

false

NI

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

Test Case:

User Enters: twenty-three

User Then Enters: 23

"twenty-three"

"23"

"twenty-three"

NFE thrown

"23"

NFE not thrown

Enter an integer:
twenty-three

Try again

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);  
        }  
    }  
}
```

↓ excep. originated

Example 2: Method that calls another which throws the exception

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

✓
↓ caller may throw NSE

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) { ... }  
    }  
}
```

no need
to specify
VTSE.

→ called may throw VTSE

A Class for Bounded Counters

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

valid values
for counter:
0, 1, 2, 3

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

* VTSE and VTLE not expected $\rightarrow$ if any exception occurred $\rightarrow$ fail the test

Coming Up with Test Cases: A Single, Bounded Variable

** VTSE or VTLE expected $\rightarrow$ if the expected exception occurred $\rightarrow$ pass the test

Boundries:

0

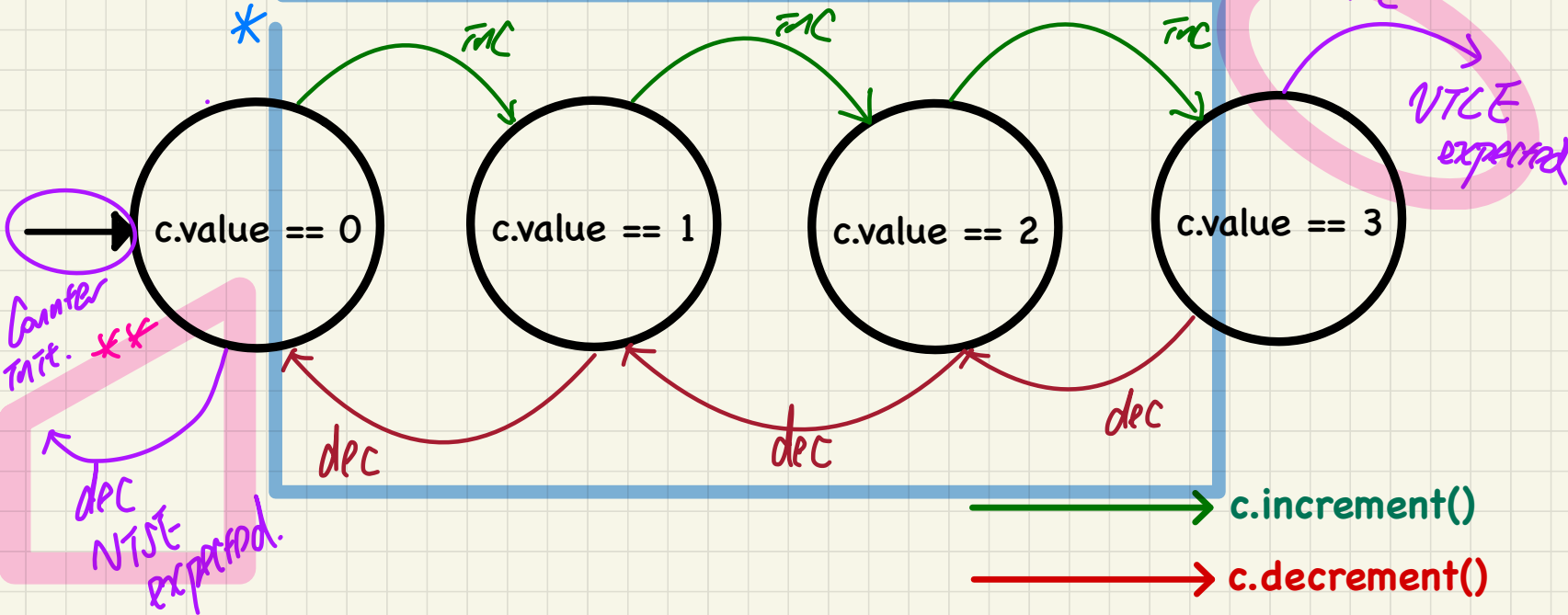
3

Counter: **MIN_VALUE** $\leq$ c.value $\leq$ **MAX_VALUE**

**

inc

VTLE expected



Lecture 10 - Oct 6

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

JUnit Test Method vs. Method Under Test

@Test

public void test() {

MyClass o = new MyClass();

assertEquals(23, o.getValue());

}

(JUnit) Test Method
(caller)

actual value

Implementation

expected value

method under test
(callee)

expected value

JUnit test method
(test)

pass
(imp. correct)

fail
(imp. incorrect)

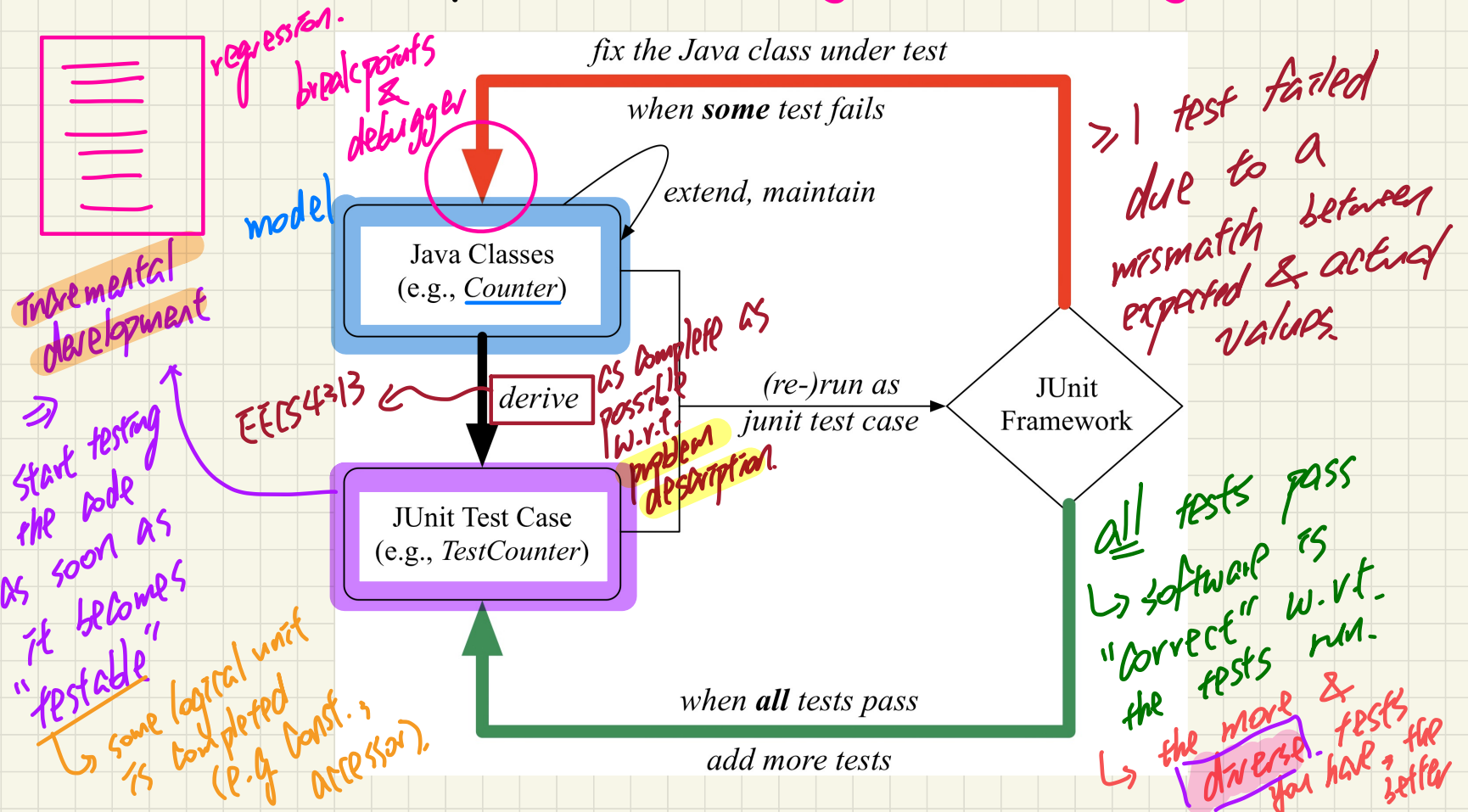
arguments

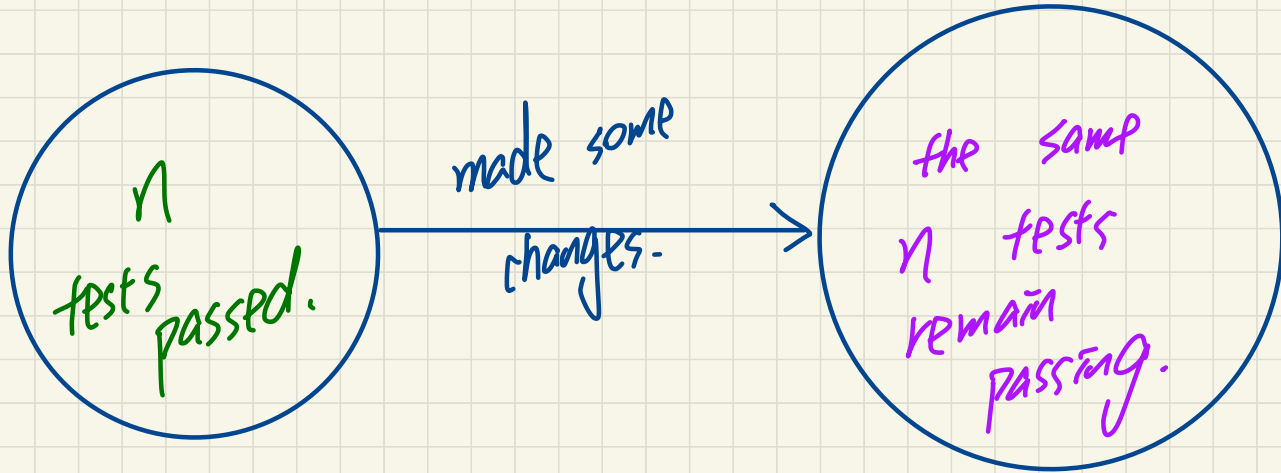
Method under test
(getValue)

return value

Test-Driven Development (TDD): Regression Testing

slide 11





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`, `ArrayIndexOutOfBoundsException`) thrown
- **Success** if neither **assertion failures** nor (unexpected) **exceptions** occur.

fail() → cause the

test to unconditionally fail & immediately.

slide 19

*no exception failure occurred.
no assertion failure occurred.
After commenting out fail assertion, the test passes trivially.*

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

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

Design of a (Unit) Test

assess if
some implementation
is correct.

(e.g. inc when counter value is max)

→ **Expectation**: certain exception occurs

↳ that kind of exception is thrown by imp. → pass

↳ that kind of exception is not thrown by imp. → fail
not expected

→ **Expectation**: certain exception does not occur
(e.g. inc when counter value is 0)
not expected

↳ that kind of exception is thrown → fail

↳ that kind of exception is not thrown → pass

JUnit: An Exception Not Expected

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

→ may throw VTLE
→ VTLE did not occur
pass the test

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

→ may throw VTLE
→ VTLE occurred
fail the test

What if increment is
implemented correctly?

↓
does not throw VTLE

Expected Behaviour:

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

↓
not expected.

What if increment is
implemented incorrectly?

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

Running JUnit Test 1 on Correct Implementation

Expectation

VTLException does not occur

$c.v == 0$

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

$0 \rightarrow 1$

test passes
because the
given imp.
is correct.

```
1  @Test  
2  public void testIncAfterCreation() {  
3      ① Counter c = new Counter(); |  $c.v == 0$ .  
4      ② assertEquals(Counter.MIN_VALUE, c.getValue());  
5      try {  
6          ③ c.increment(); |  $\rightarrow$  may throw VTLException 0.  
7          ④ 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 }
```

PASS

Running JUnit Test 1 on Incorrect Implementation

$C.V == 0$

Expectation
VTLCE not occurring
incorrect implementation.

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

VTLCE thrown unexpectedly.

the same test
fails :-
the given implementation
is incorrect.

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

fail the test case

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 }
```

Handwritten notes:
- A green oval around lines 6-9 with the text "may throw VTSE" and "if the VTSE is thrown as expected".
- A green arrow points from the comment in line 10 to the catch block.
- The word "PASS" is written in green below line 12.

What if decrement is implemented **correctly**?

Expected Behaviour:

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

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

Handwritten notes:
- A red oval around lines 6-9 with the text "may throw VTSE" and "if VTSE is not thrown".
- A red arrow points from the comment in line 10 to the catch block.
- The word "fail the test" is written in red below line 9.
- A red "X" is written next to line 10.
- The text "∴ the expected VTSE is not thrown!" is written in red below line 12.

What if decrement is implemented **incorrectly**?

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

Running JUnit Test 2 on Correct Implementation

c.v == 0

Expectation

*VTSE
occurs.*

```
public void decrement() throws ValueTooSmallException {  
    ④ if (value == Counter.MIN_VALUE) {  
        ⑤ throw new ValueTooSmallException("counter value is " + value);  
    }  
    X else { value --; }  
}
```

VTSE is expected

*the passes
∴ the given
imp. is correct
throws VTSE
as expected.*

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

pass

Running JUnit Test 2 on Incorrect Implementation

$C.V == 0$

Expectation
VTSF occurs

Incorrect Implementation.

```
public void decrement() throws ValueTooSmallException {  
    ④ if (value == Counter.MIN_VALUE) {  
        X throw new ValueTooSmallException("counter value is " + value);  
    }  
    else { ⑤ value --; }  
}  
0 → -1
```

the test
fails :-
the given imp.
is Incorrect
VTSF not
thrown when
dec. counter
with value 0.

```
1 @Test  
2 public void testDecFromMinValue() {  
3     ① Counter c = new Counter(); |  $C.V == 0$   
4     ② assertEquals(Counter.MIN_VALUE, c.getValue());  
5     try {  
6         ③ c.decrement(); ~→ VTSF not thrown  
7         ⑥ fail("ValueTooSmallException is expected.");  
8     }  
9     X catch (ValueTooSmallException e) {  
10         /* Exception is expected to be thrown. */  
11     }  
12 }
```

fail

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

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

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

→ P1: **VTLÉ** not expected

→ P2: **VTLÉ** expected

Lecture 11 - Oct 8

TDD with JUnit

Example Test 3:

***Some Exceptions Expected, Some Not
Console Tester vs. JUnit Test
Using a for-Loop for***

Announcements/Reminders

- Today's class: [notes template](#) posted
- I need to invigilate Section G's ProgTest1 at 1 PM.
 - + We'll finish at ¹²1:30 PM
 - + Please study the last 20 minutes of Section G's Thursday lecture recording.

Running JUnit Test 3 on Correct Implementation

```
public void increment() throws ValueTooLargeException {
    ④ ③ if (value == Counter.MAX_VALUE) {
        X ② throw new ValueTooLargeException("counter value is " + value);
    }
    else X ④ { value++; }
    }
    0 → 1
```

correct behaviour

the test passed
imp. correctly
sat. expectations
in P1 & P2

```
1 @Test
2 public void testIncFromMaxValue() {
3     ① Counter c = new Counter(); | C.V == 0
4     try {
5         ② c.increment(); | → VTL not thrown
6         | ⑤ 1 → 2 | ④ 2 → ③ | → VTL not thrown.
7     } catch (ValueTooLargeException e) {
8         X fail("ValueTooLargeException was thrown unexpectedly.");
9     }
10    ⑦ assertEquals(Counter.MAX_VALUE, c.getValue());
11    try {
12        ⑧ c.increment(); → VTL thrown
13        X fail("ValueTooLargeException was NOT thrown as expected.");
14    }
15    catch (ValueTooLargeException e) {
16        ⑨ /* Do nothing: ValueTooLargeException thrown as expected. */
17    }
18 }
```

pass!

Running JUnit Test 3 on Incorrect Implementation

```
public void increment() throws ValueTooLargeException {  
    ③ if (value < Counter.MAX_VALUE) {  
        ④ throw new ValueTooLargeException("counter value is " + value);  
    }  
    X else { value++; }  
}
```

Incorrect behaviour

the test failed
∴ imp. failed to sat.
PI expectation

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

→ will throw

bypassed ∴ the test already failed

Running JUnit Test 3 on Incorrect Implementation

```
public void increment() throws ValueTooLargeException {  
    ③ ④ ⑥ ③ if (value > Counter.MAX_VALUE) {  
        XXX throw new ValueTooLargeException("counter value is " + value);  
    }  
    ④ ⑦ ⑩ ④ else { value++; }  
}
```

0 → 1
1 → 2
2 → 3
3 → 4

Incorrect behaviour

```
1 @Test  
2 public void testIncFromMaxValue() {  
3     ① Counter c = new Counter(); | C.V == 0  
4     try { | C.V == 1 | ⑤ | C.V == 2 | C.V == 3  
5         ② c.increment(); | c.increment(); | c.increment();  
6     }  
7     catch (ValueTooLargeException e) {  
8         fail("ValueTooLargeException was thrown unexpectedly.");  
9     }  
10    ⑩ assertEquals(Counter.MAX_VALUE, c.getValue());  
11    try {  
12        ⑫ c.increment(); → VLE not thrown  
13        ⑬ fail("ValueTooLargeException was NOT thrown as expected.");  
14    }  
15    catch (ValueTooLargeException e) {  
16        /* Do nothing: ValueTooLargeException thrown as expected. */  
17    }  
18 }
```

fail.

test fail
∴ imp. sat. P1 exp.
but failed to
sat. P2 exp.

Exercise: Console Tester vs. JUnit Test

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

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

*Incorrect. Witness
(if (value < 3. max) {
throw ... 3.*

*Ante-
dictory
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 }
```

pass or fail?

VTLException comes from P1 → fail
VTLException comes from P2 → 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

MIN: 0
MAX: 3

what if the range of valid values is wrap

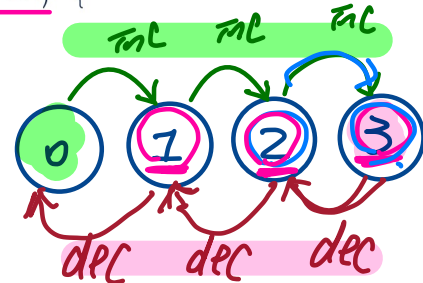
Loops can make it effective on generating test cases:

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

→ i: 0, 1, 2

c.v before inc.

c.v after inc



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`

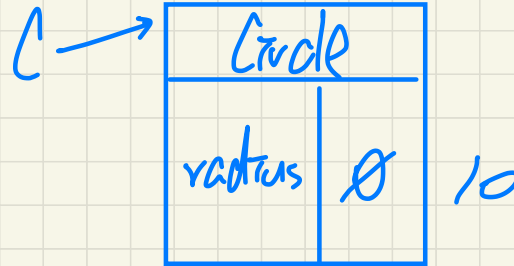
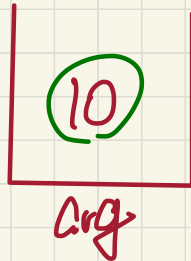
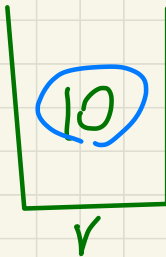
Call by Value: Primitive Argument

```
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



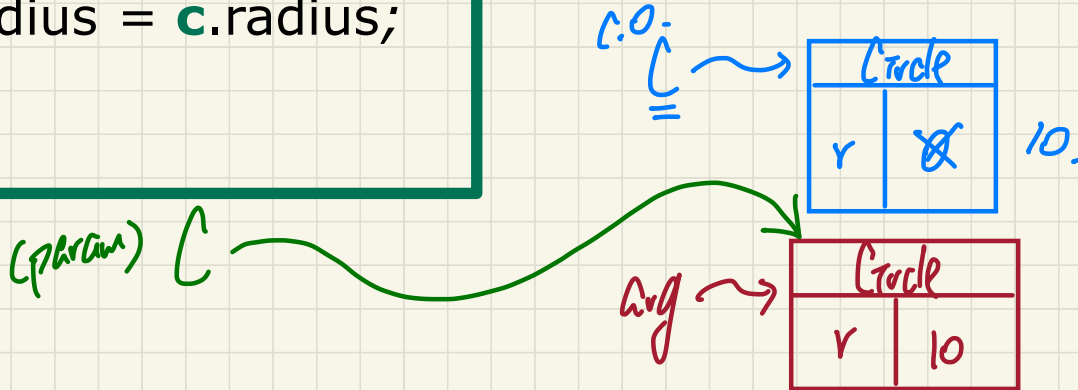
Call by Value: Reference Argument

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

Call by value: c = arg

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

c.0.



Lecture 12 - Oct 20

Object Equality.

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

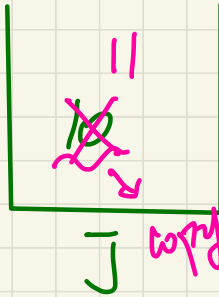
Announcements/Reminders

- 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 (and deadline shifted accordingly)

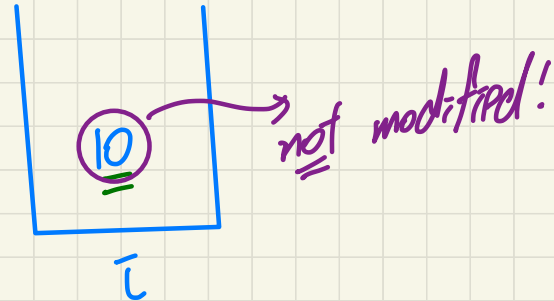
Call by Value: Re-Assigning Primitive Parameter

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

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



call by value:
⑤ $j = 11$



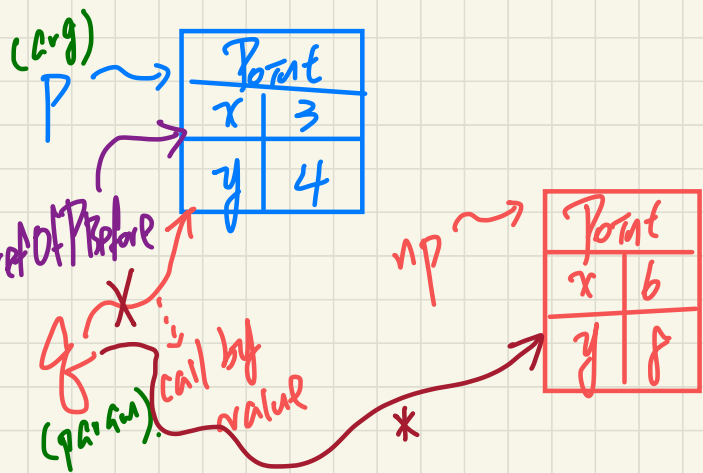
* only f , a copy of p , is modified. p itself is not modified.

Call by Value: Re-Assigning Reference Parameter

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

```
1 @Test
2 public void testCallByRef_1() {
3     Util u = new Util();
4     Point p = new Point(3, 4);
5     Point refOfPBefore = p;
6     u.reassignRef(p);
7     assertTrue(p == refOfPBefore);
8     assertTrue(p.getX() == 3);
9     assertTrue(p.getY() == 4);
10 }
```

important to export that p is re-assigned to np .

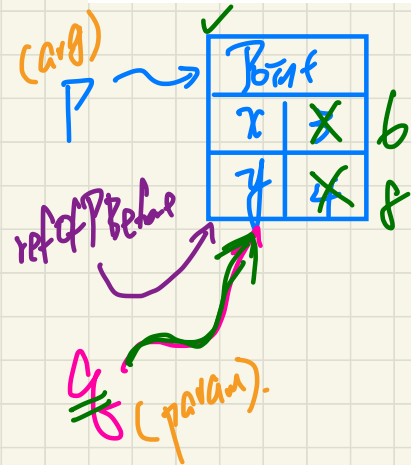


```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return this.x; }
    public int getY() { return this.y; }
    public void moveVertically(int y) { this.y += y; }
    public void moveHorizontally(int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

Call by Value: Calling Mutator on Reference Parameter

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

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



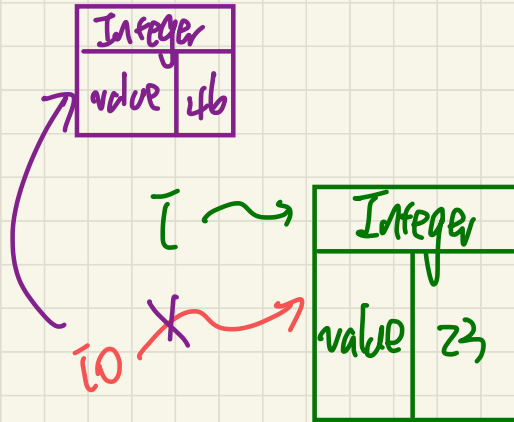
④ b.v.
⑤ q = p

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

```
void m(Integer i0) {
    ④ i0 = new Integer(46);
    [ ]
}
```

c.b.v.
③ $i0 = i$;

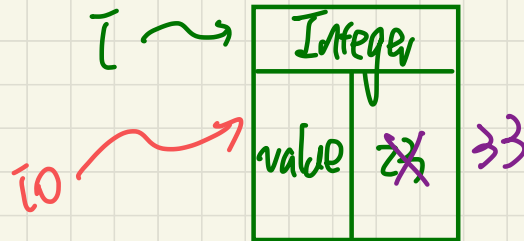
```
① Integer i = new Int.(23);
    ② m(i);
    ;
```



```
void m(Integer i0) {
    i0.add(10);
}
```

c.b.v.
③ $i0 = i$;

```
① Integer i = new Int.(23);
② m(i);
;
```



Equality < == equals

Primitive

int i = 23;

int j = 46;

i == j

i.equals(j)

1. == (value).
2. equals X compilation error

Reference

Point p1 = new Point(3, 4);

Point p2 = new Point(3, 4);

p1 == p2

false
↳ different
add. values

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);  
        ① System.out.println(p1.equals(p2));  
    }  
    ② println(p1 == p2);  
}
```

Q1:

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

Yes '∵ equals method is implicitly defined in every class.

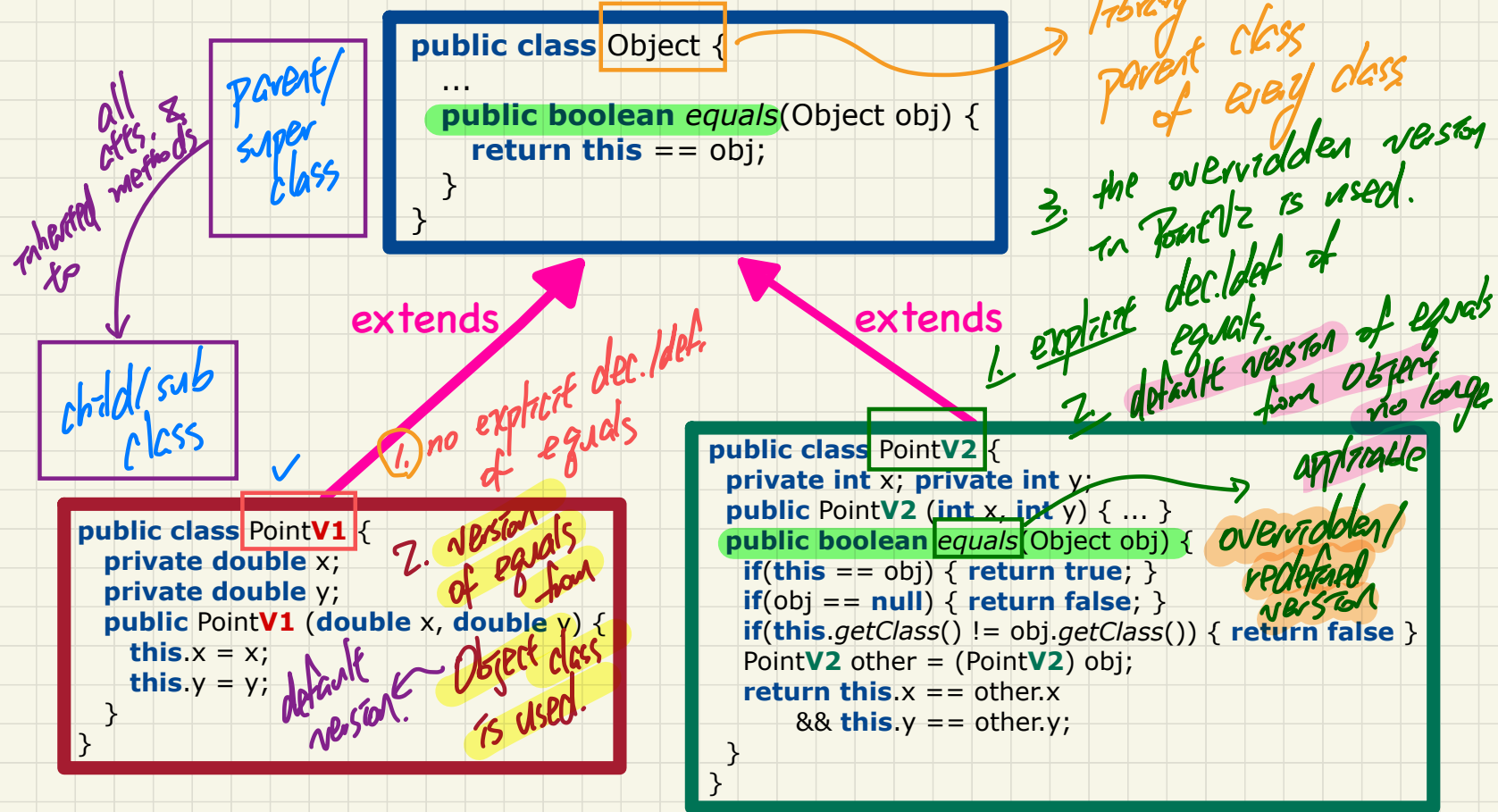
Q2:

Where does **equals** come from?

effectively the same
- i.e. no explicit
def. of equals method in PointV1

→ Eclipse: you may have to attach the source of your JDK installation
→ step into: Object class' equals method being called
→ p1 == p2
this obj.

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



* inherited from Object class *

The equals Method: Default Version

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

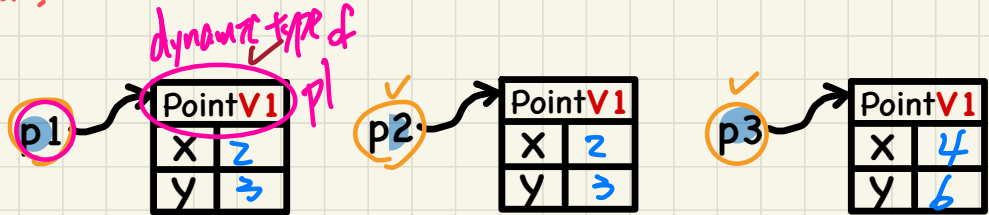
extends

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

no explicit redefinition/overriding of equals.

inherited to PointV1 class.

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



L7: C.O. p1 points to an object of type PointV1
dynamic type of p1 is PointV1
↳ the equals method comes from the one *

Lecture 13 - Oct 22

Object Equality.

***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

Method Call: When Multiple Versions are Available

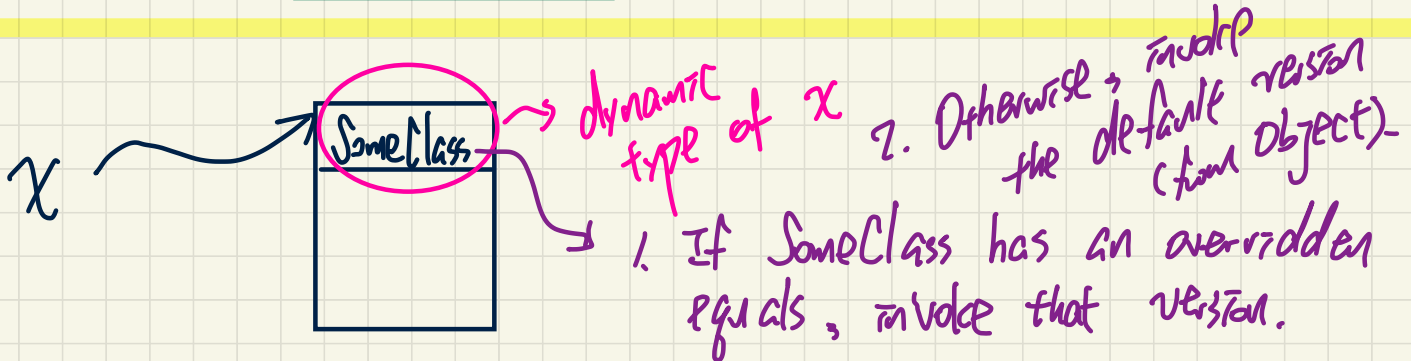
C.O. is all that matters to determine the version of equals to be invoked.

x.equals(y)

1. default version (from Object)?
2. overridden version (if it 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**).

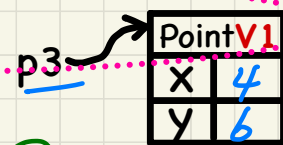
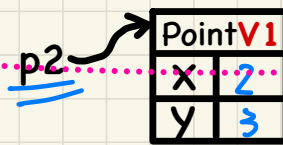
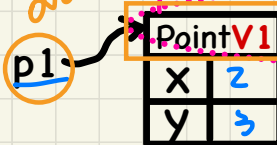


The equals Method: Default Version

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

```
1 String s = "(2, 3)";  
2 PointV1 p1 = new PointV1(2, 3);  
3 PointV1 p2 = new PointV1(2, 3);  
4 PointV1 p3 = new PointV1(4, 6);  
5 System.out.println(p1 == p2); /*  
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)); /*
```

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



27: → p1 == p1 (T)

28: → p1 == null (F)

29: → "p1 == s" (F)

3. Invoke the default version from Object

extends

1. dynamic type of p1 is PointV1.
2. PointV1 has no overridden equals.

dynamic type

s → "(2, 3)"
same dynamic type

Point V1 $p1 = \underline{\text{new}} \dots \Rightarrow$

String $s = \dots$

① $p1.\text{equals}(s)$

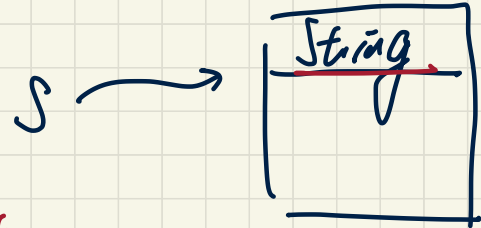
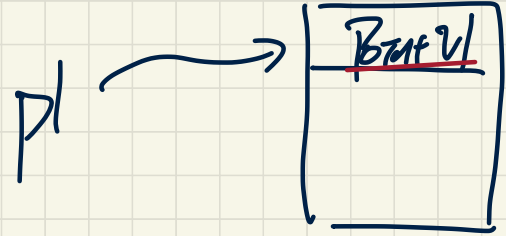
$\hookrightarrow "p1 == s"$

Compiler: argument of equals is simply expecting an object.

② $p1 == s$

$\leadsto$ compilation error!
because comparing addresses of two different types

String $s1 = \text{new String}("a");$
String $s2 = \text{new String}("a");$
 $s1 == s2$ (F)
 $s1.\text{equals}(s2)$ (T)



Call by Value: Invoking the Overridden equals

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

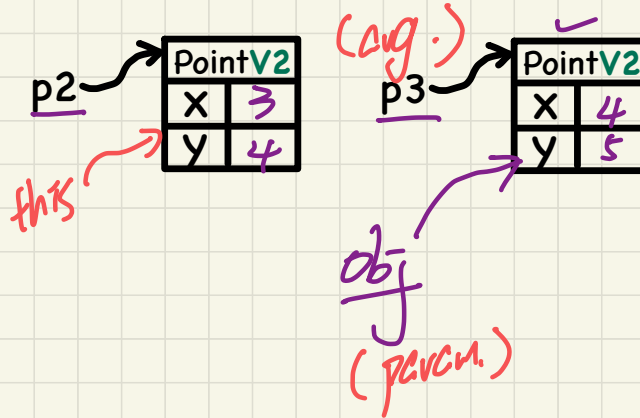
extends

C.b.V.
obj = p3

overridden
version

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

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



Short-Circuit Evaluation: && and/ conjunction

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

Test Inputs:

x = 0, y = 10

x = 5, y = 10

if at least one operand is false, && is false 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");
    }
}
    
```

if $x = 0$ → Division by Zero exception.
 expression should not fail to eval.
 if $x \neq 0$ && $10/0 > 2$ skipped
 if $x = 5$ && $10/5 > 2$ → $5 \neq 0$ && $10/5 > 2$ → $2 > 2$

1. Evaluation is from L to R.

2.1 If op1 eval. to T, eval op2, and its result is the final

2.2. If op1 eval. to F, result skip eval. of op2
 ↳ overall && is (F)

$$\underline{x \neq 0} \quad \checkmark \quad \&\&$$

Guarding condition.

↳ 1. If $x \neq 0$ eval. to T,
Continue to eval. y/x

2. If $x \neq 0$ eval to F $\rightarrow x = 0$

↳ skip the eval. of RHS.

DBFE $\neq x == 0$

$$y/x > 2$$

RHS

Short-Circuit Evaluation: $\parallel$ or/ disjunction.

Left Operand op1	Right Operand op2	op1 op2
z.2. false	false	false
true	false	true
z.1. false	true	true
true	true	true

Test Inputs:
 $\checkmark x = 0, y = 10$
 $x = 5, y = 10$

if at least one operand is true, then true.

```

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

T $0 == 0 \parallel 10/0 > 2$
 F $5 == 0 \parallel 10/5 > 2$
 skip.

1. Eval: L to R

2.1 If eval. op1 is T

skip eval. of op2, overall result of $\parallel$ is T.

2.2 If eval. op1 is F

eval. op2, and its result is the overall result of $\parallel$.

Math

$P \wedge Q$
 $P \vee Q$

Commutativity

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

Java

$P \&\& Q$
 $P \parallel Q$

← doesn't apply in runtime!

$$\frac{x \neq 0}{\text{guarding cond.}}$$

$$\frac{x == 0}{\text{error cond.}}$$

??

If it's
true eval. 2nd op. some
expression to protect.

$$y/x \dots$$

$$y/x \dots$$

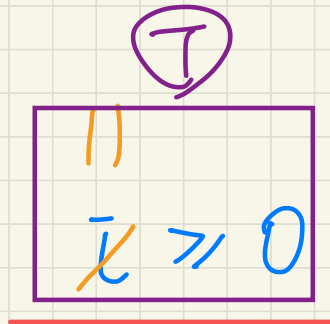
||

logical
negation
of each
other.

error cond.

if this is false,
eval. 2nd op.

a : array of values



may trigger
IOOBE.

$a[i] == v$

Assumption:
 $a.length == 10$
 $0..9$

$i \leq a.length - 1$

$g!$ may fail.

WTF pass : $i == 1$

IOOBE !!

Correct

Eraser Use "op.

$g!$ && $g?$ && ..
 $a[i]$

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 */ }
```

Oct 25

Written Test 1 Q&A

Assume the following method declaration: ^{in Person class}

```
public void growWeight( double amount) { -- }
```

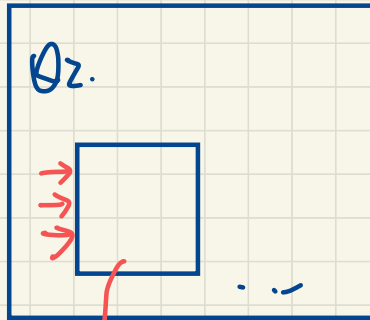
Write a line a java code that grows pl,
ref variable of typer Person, weight by 23.6.

```
pl.growWeight(23.6);
```

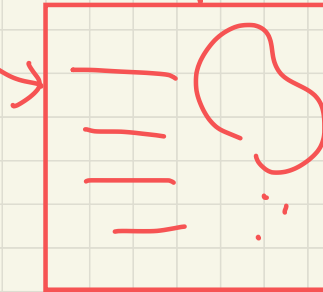
Guide

familiar with concepts
covered in Lab 0.

Screen

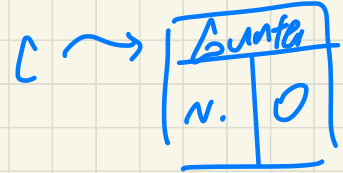


sketch paper



Running JUnit Test 1 on Correct Implementation

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



init: $v == 0$
assertEquals(0, 0)
↳ pass
if (0 == 3)
↳ not thrown
else
↳ v: 0 → 1
assertEq(1, 1) → pass

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

Lecture 14 - Oct 27

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

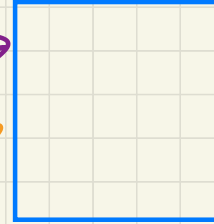
```
public class PointV2 {  
    private int x;  
    private int y;  
    public PointV2(int x, int y) { ... }  
    public boolean equals(Object obj) {  
        if (this == obj) { return true; }  
        if (obj == null) { return false; }  
        if (this.getClass() != obj.getClass()) { return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

✓
p1. equals(p2)
C.O. arg

(arg. p2) ~

obj
(param.)

(C.O.)
p1 ~
this ~



anything
is equal to
itself.

PointV2	
x	
y	

The equals Method: Overridden Version

Example 1: Trace L7

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

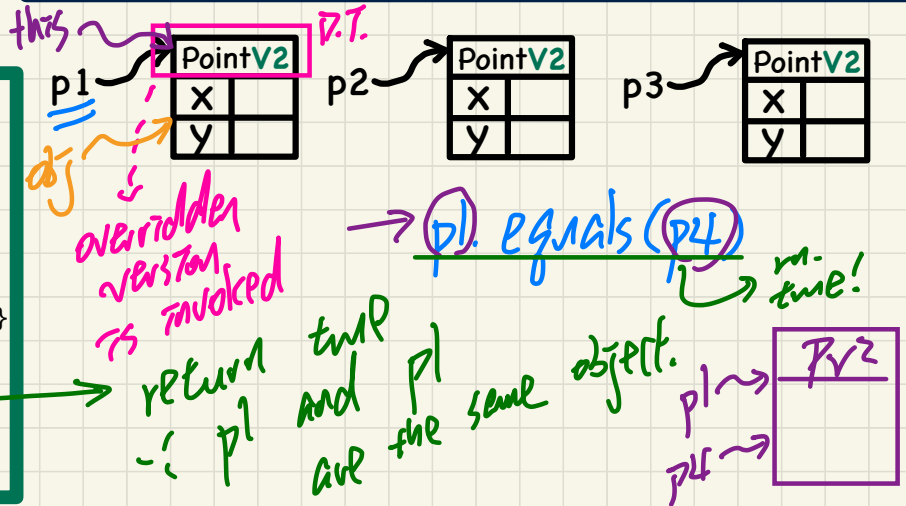
extends

```
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;
    }
}
```

```

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(p1 == 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] */

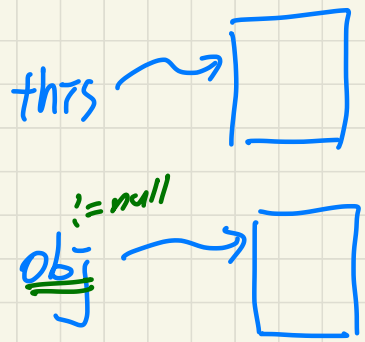
```



The equals Method: Overridden Version

Phase 2

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



not readable,
NPE
would be
occurred!!

extends
Ⓟ equals (Ⓟ?)
↳ if p1 == null
↳ NPE even before

```
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;  
    }  
}
```

realizing
this !=
obj.

we step into the
body of equals
method

```
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;  
    }  
}
```

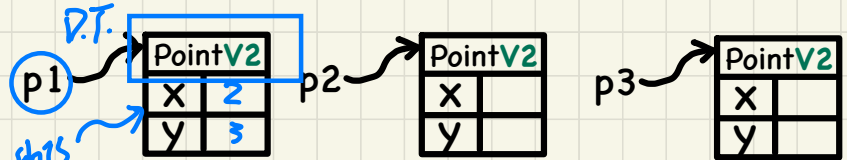
extends

```
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;  
    }  
}
```

∵ p1 is not null but it's compared to null for equality.

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

False



obj is null

Static Type, Dynamic Type, Type Casting

Var.	ST	exp
o1	C1	m1, eg
o2	C2	m2, eg
o3	Object	equals.

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

```

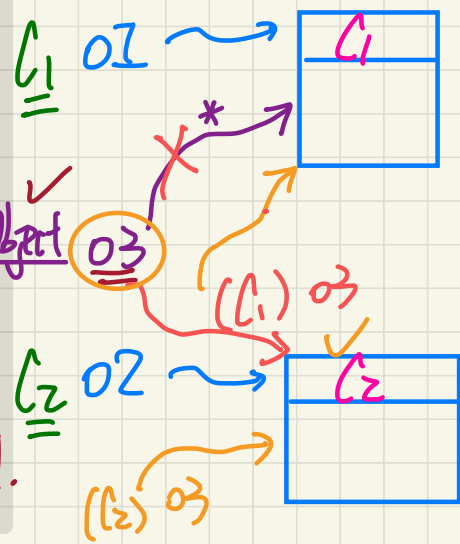
class C1 {
    ...
    public void m1() {...}
}

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

```

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

create a new copy of address with a different expectation



determines the range of methods that can be invoked on a ref. variable

(C1) o3

a new alias that:
 1. points to the same object as o3
 2. this new alias has ST. same as o3

do not compile
 ∵ ST of o3 remains Object
 ∵ cast type is C1.

Given a line of method call:

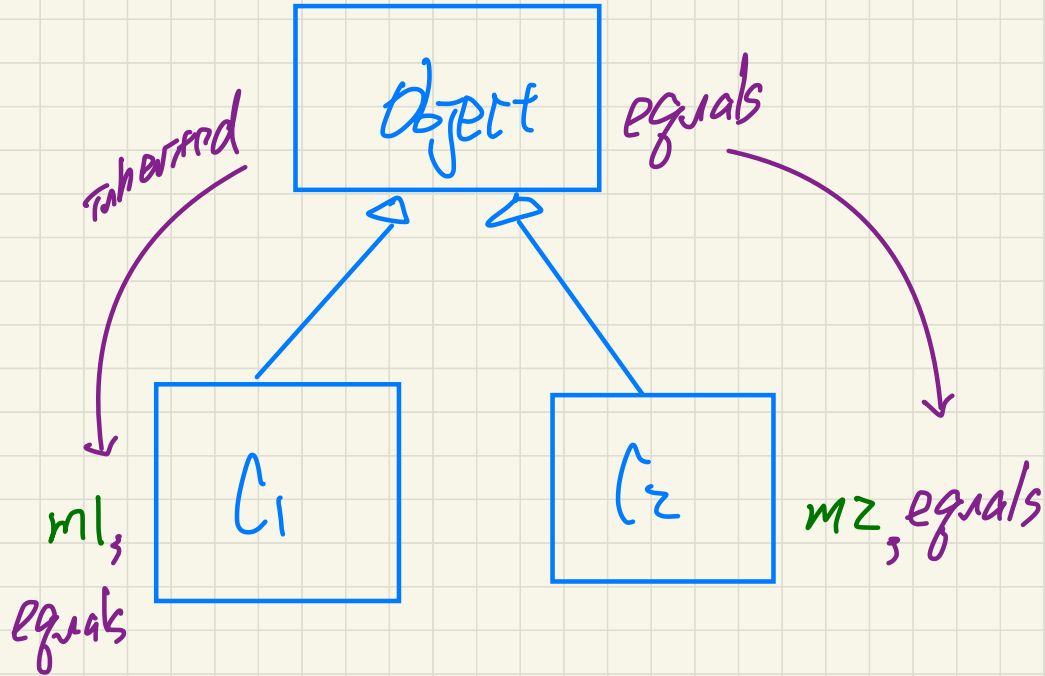
$obj.m(\dots)$

(1) Does the line compile?

↳ look at ST. of C.O. obj.

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

↳ look at M.T. of C.O. obj.



parent of every class

Object

always
compile

can be
any class

obj

=

new

SomeClass();

Object

obj

=

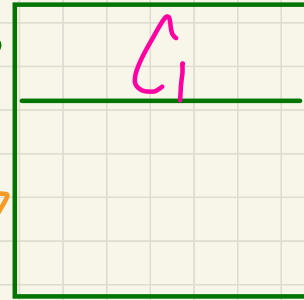
obj2;

↓
declared type
can be any class

$(C_1 \checkmark \boxed{03}) \cdot m1$
 this exp. has the ST
 C1, meaning
 that we can invoke
 methods from C1
 on it.

$\times (C_1) 03 \cdot m1$

type cast first,
 then call, m1.
 object.03



$(C_1) 03$
 this alias has
 ST. C1

The equals Method: Overridden Version

Phase 3

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

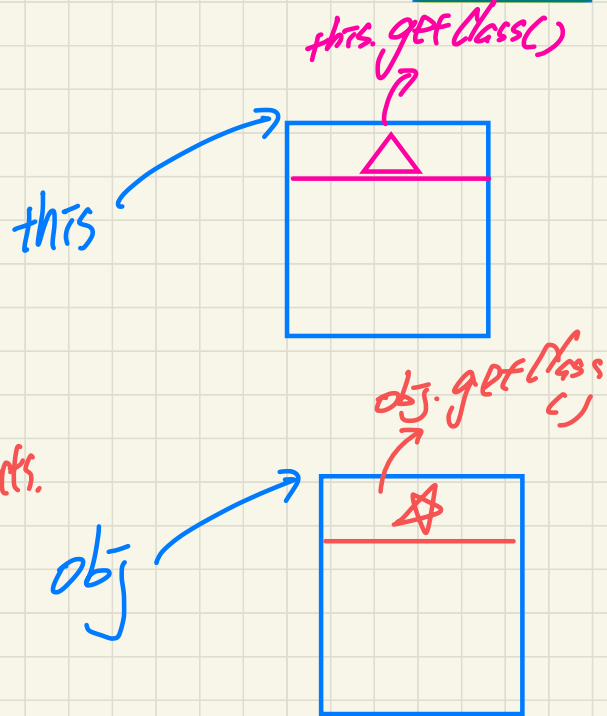
extends

```
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 context objects.

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



PointV2	
x	
y	

The equals Method: Overridden Version

Example 1: Trace L9

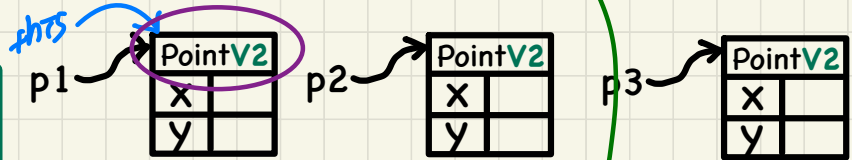
```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
}
```

extends

PointV2
!=
String (T)

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

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



rn. false
∴ p1 and s have different ref. T.s.

The equals Method: Overridden Version

Phase 4

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

extends

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

1. $this \neq obj$
2. $obj \neq null$
3. *

Can we write this?

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

PointV2

x	..
y	..

PointV2

x	..
y	..

PointV2	
x	
y	

not compile
obj's
ST does not support x & y.

obj has
ST: Object

Lecture 15 - Oct 29

Object Equality

Overridden equals: Phase 4
Short-Circuit, Logical Implication
JUnit: assertEquals 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;  
    }  
}
```

* PointV2 other =
(PointV2) obj;

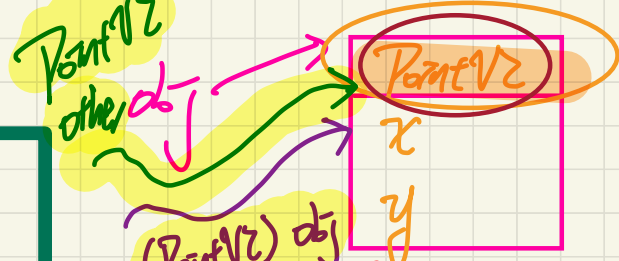
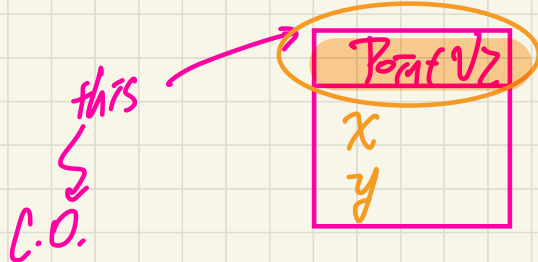
extends

d.t. of C.O. this

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

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

Compile '1' ST of other is PointV2.



alt. cost exp of ST. PointV2
this.x == obj.x
&& this.y == obj.y

ST of obj is object.

PointV2	
x	
y	

The equals Method: Overridden Version

Example 1: Trace L10

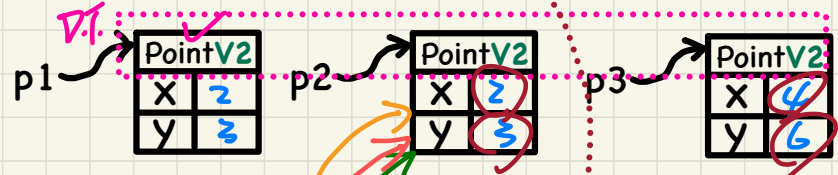
D.T.

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

extends

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

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



Object obj

PointV2 other

(PointV2) obj

return false
Exception

The equals Method: **Overridden** Version

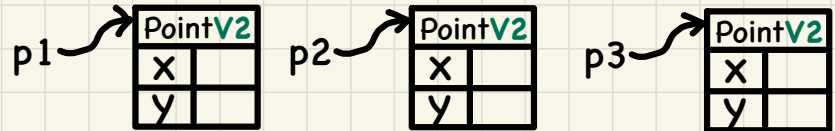
Example 1: Trace L11

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

extends

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

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



The equals Method:

To Override or Not?

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

extends

extends

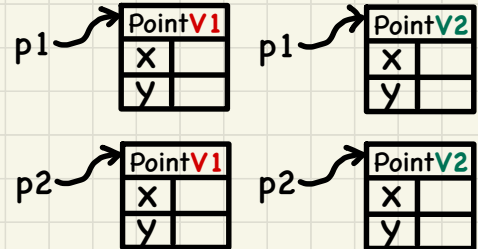
```
public class PointV1 {  
    private int x;  
    private int y;  
    public PointV1(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
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;  
    }  
}
```

default
version from
Object
is called
↳
p1 == p2
overridden
version from
PointV2
is called

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

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



```

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;
    }
}

```

..... → risk of NPE

a ☐ ~~if (① && ②) { return false; }~~

b ☐ ~~if (② && ①) { rn. false; }~~

c ☒ if (① || ②) { rn. false; }

d ☐ if (② || ①) { rn. false; }

↘ if true, skip eval. ②
 ↓ if obj is null NPE would've occurred before evaluating ①.

The equals Method: Overridden Version

Example 2

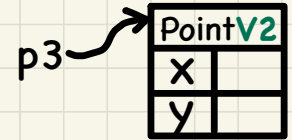
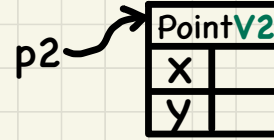
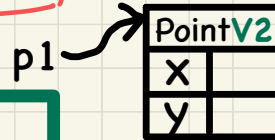
```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
}
```

extends

obj1
obj2

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

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



obj1 == obj2

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

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

obj1.equals(obj2)

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

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

not valid.

A → B

B → A

Logical Implication

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

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

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

assertSame(exp1, exp2) ^{Any objects}

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

≈ **assertTrue**($exp1 == exp2$)

≈ **assertFalse**($exp1 != exp2$)

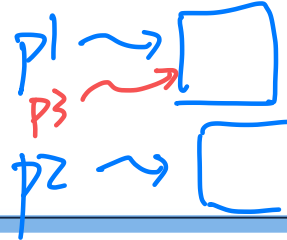
```
Point v1 p1 = new Point v1(3, 4);
```

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

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

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

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



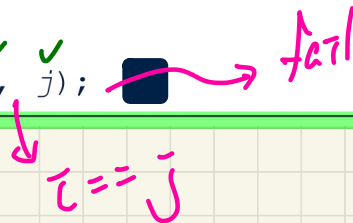
assertEquals(exp1, exp2)

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

```
int i = 10;
```

```
int j = 20;
```

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



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

* $\text{assertT}(\text{p1.equals(p2)})$
** $\text{assertT}(\text{p2.equals(p3)})$

assertEquals(exp1, exp2)

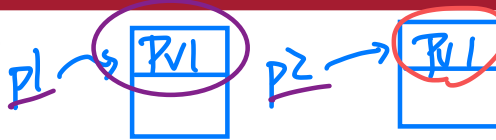
◦ $\approx \text{exp1.equals(exp2)}$ if exp1 and exp2 are **reference** type

Case 1: If equals is **not** explicitly overridden in exp1's declared type
 $\approx \text{assertSame}(\text{exp1}, \text{exp2})$

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

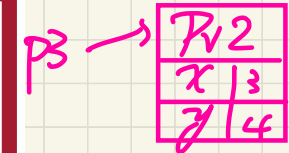
assertEquals(p1, p2);

assertEquals(p2, p3);



$p1 == p2$

" $p2 == p3$ "



Case 2: If equals is explicitly **overridden** in exp1's declared type
 $\approx \text{exp1.equals(exp2)}$

** phase 3

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);

p1.eg. (p2)
p2.eg. (p3)

* p3.eg. (p2)

↳ overridden ver.

↳ rn. false 'c' diff. r.t.

**

both
rn.
false
but
for
diff.
reasons

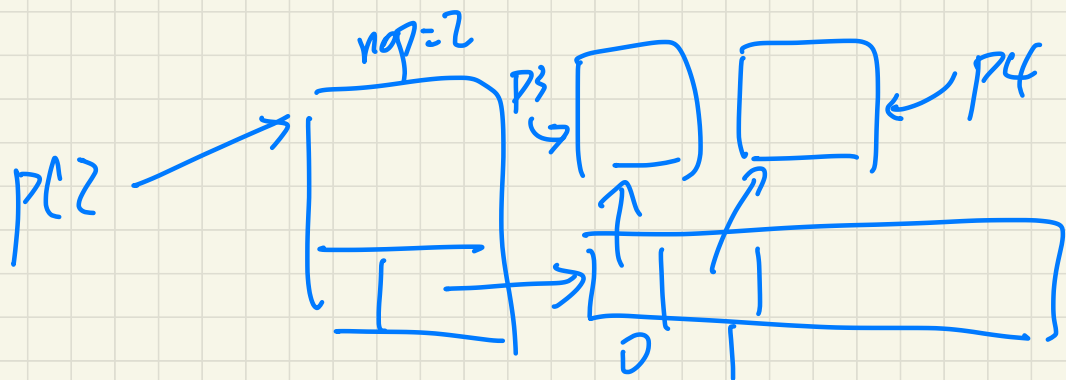
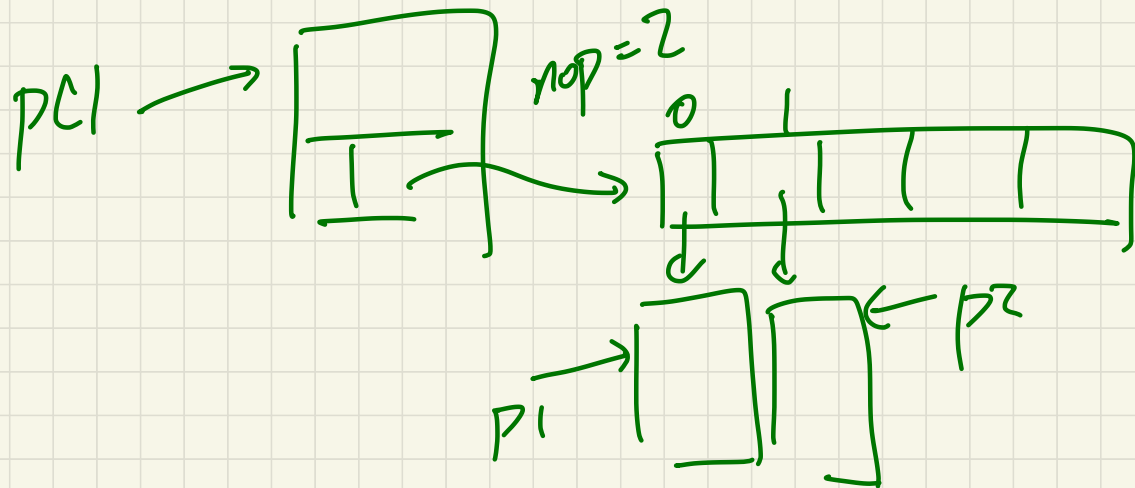
Exercise: PersonCollectors are **equal** if their arrays of persons are **equal**

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

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

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

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

Lecture 16 - Nov 3

Inheritance

Design Principles: Cohesion, SCP

SMS: Design Attempt 1

SMS: Design Attempt 2

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

→ arrays > collection.

Problem: A student management system stores data about ^{RS} students. There are two kinds of university ^{NRS} 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.

Design Principles

1. Cohesion

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

be under a unifying theme.

Unix principle
↳ each command should do one thing and do it well
↳ cd pwd

2. Single Chore Principle

↳ A change to be made should impact a minimum number of places

not necessarily 1,
but no more than necessary.

classes,
methods.

violation:
duplicates

First Design Attempt

→ how to simulate OO usage 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;
```

32 bits → 2³² kinds max encoded
RS
NRS

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

1 or 2

RS

NRS

```
}  
...  
}
```

rs. getTuition()
nrs. getTuition()

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

rs →

	S.
K	1
N	1.2
d	X

nrs →

	S.
K	2
N	X
d	0.8

```
public double getTuition(){
```

```
    double tuition = 0;  
    for(int i = 0; i < this.noc; i++){  
        tuition += this.courses[i].fee;  
    }
```

base amount.

```
    if (this.kind == 1) {  
        return tuition * this.premiumRate;
```

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

```
public void register(Course c){
```

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

First Design Attempt

```
public class Student {  
    private Course[] courses;  
    private int noc;  
  
    private int kind;  
    private double premiumRate;  
    private double discountRate;  
  
    public Student (int kind){  
        this.kind = kind;  
    }  
    ...  
}
```

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

```
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

*indicates cohesion
:- for RS, dr
is not applicable
for MRS, pr
is not applicable*

First Design Attempt

```
public class Student {  
    private Course[] courses;  
    private int noc;  
  
    private int kind;  
    private double premiumRate;  
    private double discountRate;  
  
    public Student (int kind){  
        this.kind = kind;  
    }  
    ...  
}
```

No violating SCP.

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

dupl.

dupl.

dup 3.

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

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

Good design?

Judge by Single Choice Principle

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

*kind == 3
↳ International.*

Testing Student Classes (without inheritance)

separate, unrelated classes

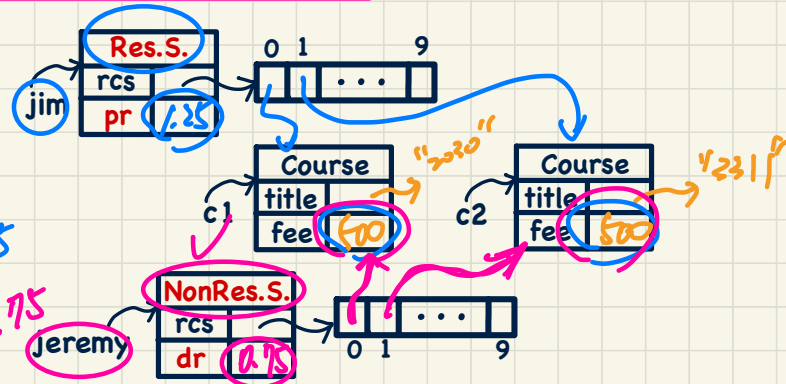
```
public class ResidentStudent {
    private String name;
    private Course[] courses; private int noc;
    private double premiumRate; /* assume a m
    public ResidentStudent (String name) {
        this.name = name;
        this.courses = new Course[10];
    }
    public void register(Course c)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

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

composition.

duplicates
→ violating SCP!

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



Student Classes (without inheritance): Maintenance (1)

violates SCP.

```
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;
    }
}
```

✓ change 1

```
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;
    }
}
```

✓ change 2

plans
to make
change is
not min.

Maintenance e.g., a new registration constraint:

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

↳ violates
SCP.

Student Classes (without inheritance): Maintenance (2)

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

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

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

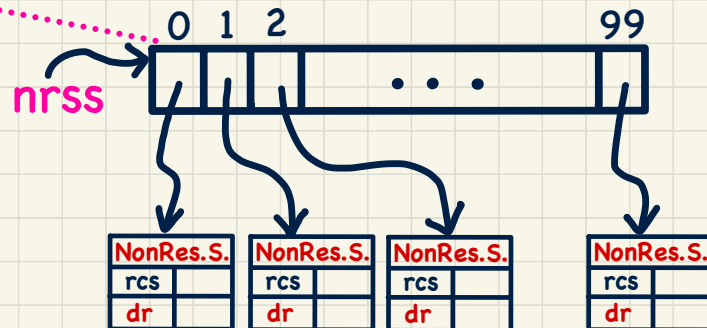
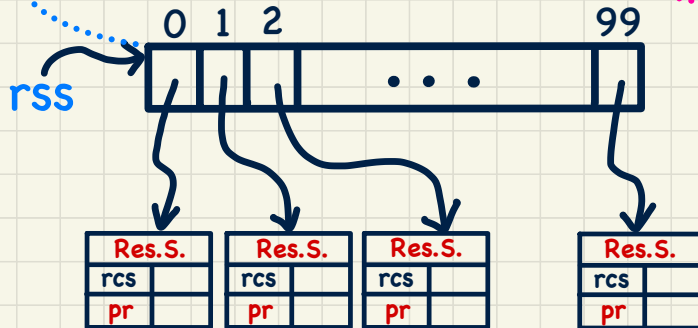
```
/* ... can be premiumRate or discountRate */
...
return tuition * inflationRate * ...;
```

A Collection of Students (without inheritance)

unrelated

```
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); }  
    }  
}
```

kinds of students
→ # arrays
→ # loops to write



Student Classes (with inheritance)

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

child/sub class

extends

parent/super class

Common super/parent class

Student is the parent class and common of RS and NRS

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

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

Lecture 17 - Nov 5

Inheritance

Modifiers in Java
Static Types and Expectations

Announcements/Reminders

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

Student Classes (with inheritance)

thrs vs. super
 ↳ current class
 ↳ common super / parent class
 ↳ follow extends

class
Common
super/parent
class

follow
extends to
parent class

A student is the class
Common Parent of RS and NRS

base
ant.

back
and -

overriden
versions

child/sub class $\xrightarrow{\text{extends}}$ parent/super class

child/sub class
specific attributes

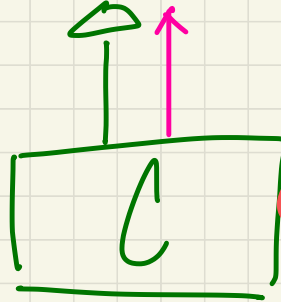
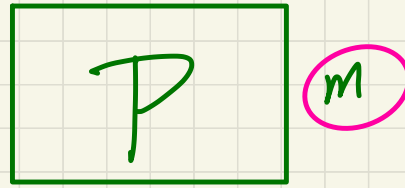
no need
→ to redeclare
attrs / methods
from parent
class (inheritance)

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

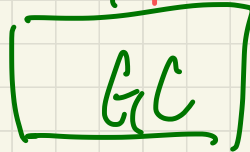
→ apply pr to base cnt

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

appx dr to
3000
4m f

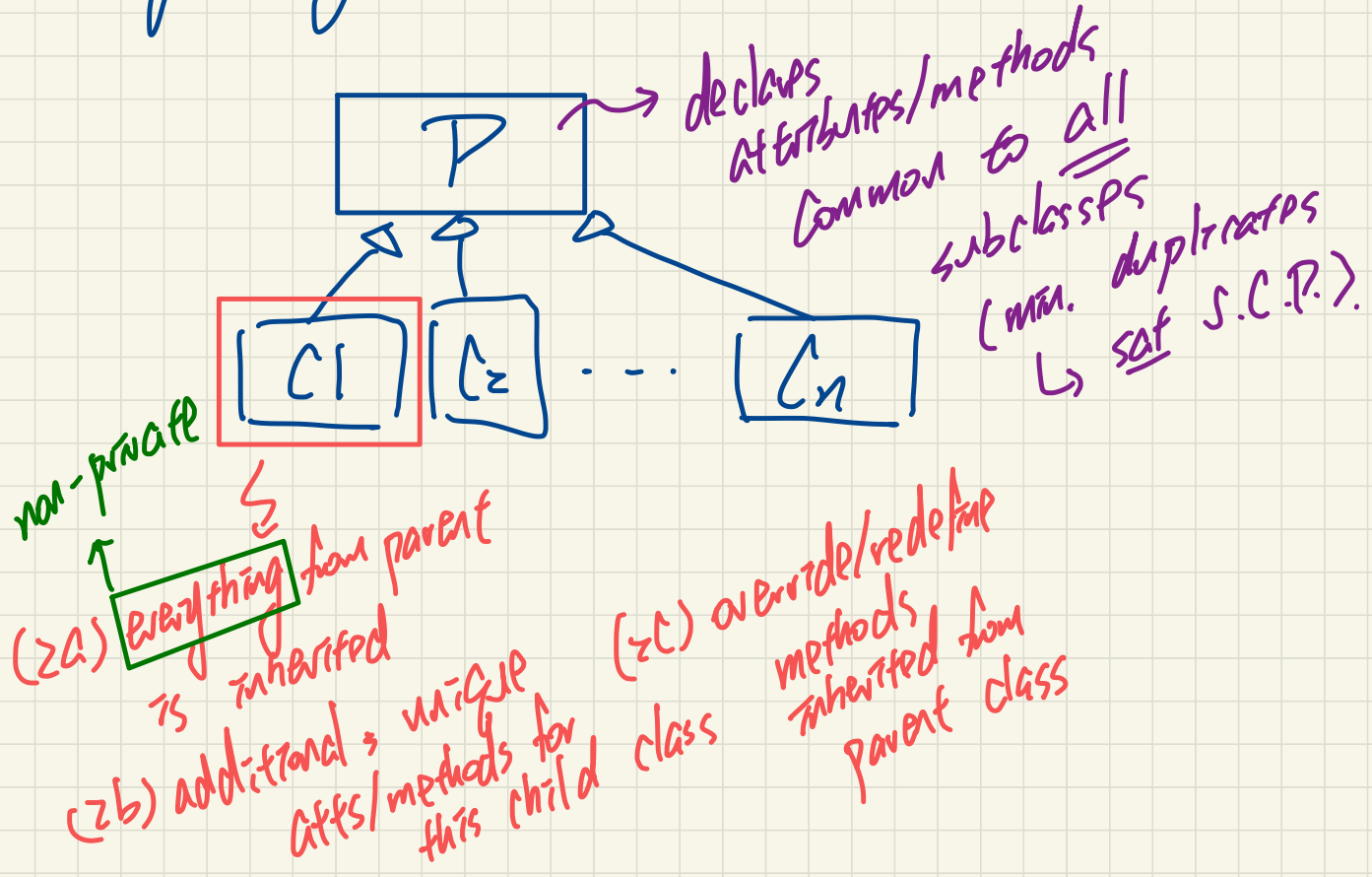


m^* overrides
↳ $\text{super.m}()$ version in P.



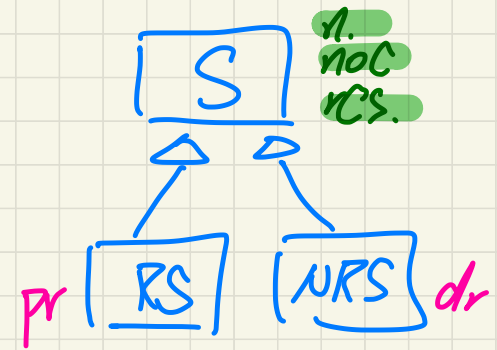
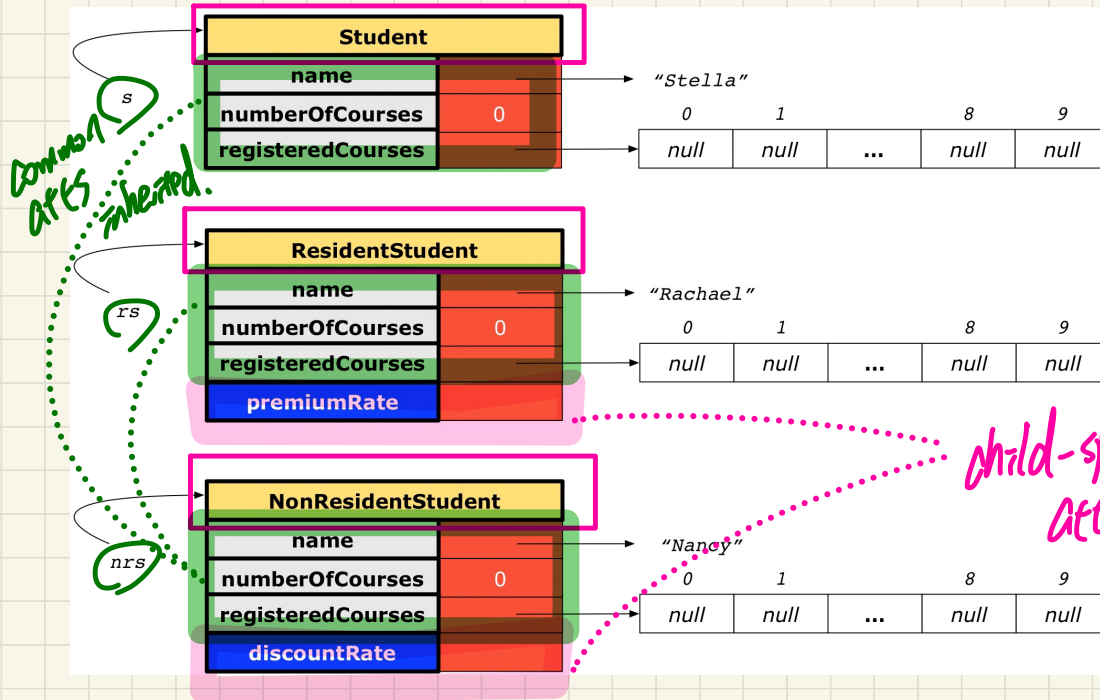
m^* overrides
↳ $\text{super.m}()$ version in C.
↳ ~~$\text{super.super.m}()$~~ ;

When programming with inheritance:



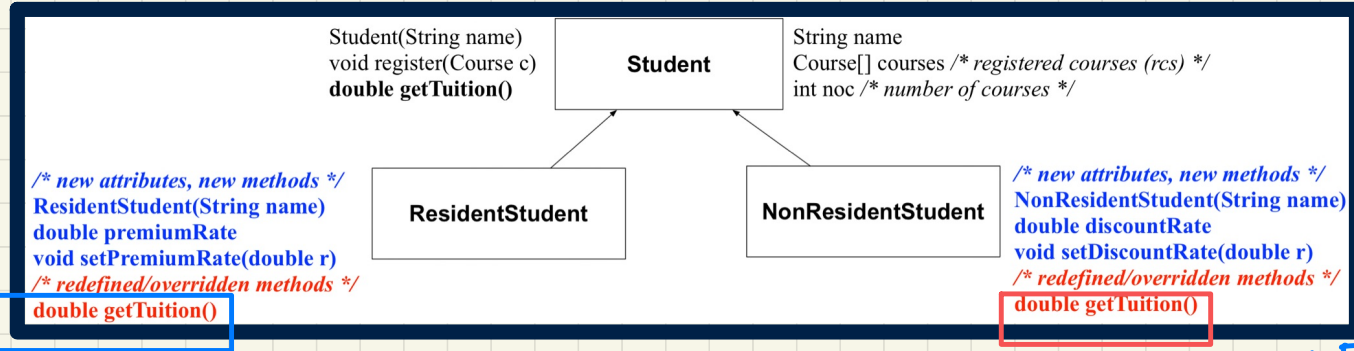
Visualizing Parent and Child Objects D.T.s

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



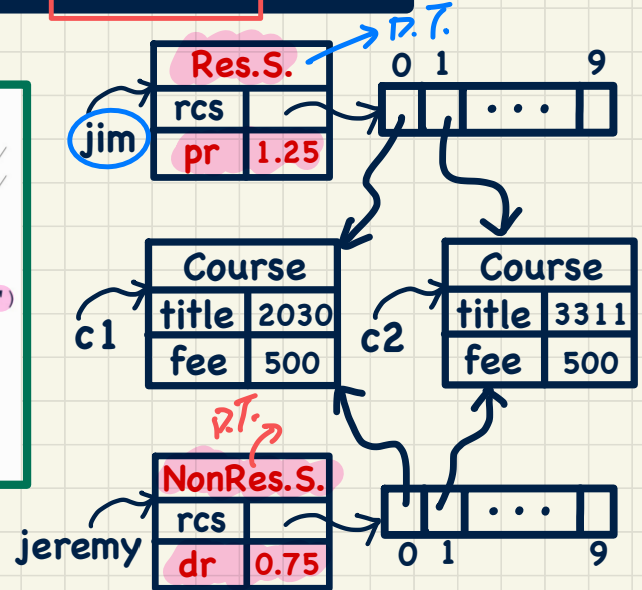
child-specific attributes

Testing Student Classes (with inheritance)



```

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



Modifiers in Java

visibility.

① private

② protected

③ no modifier

④ public

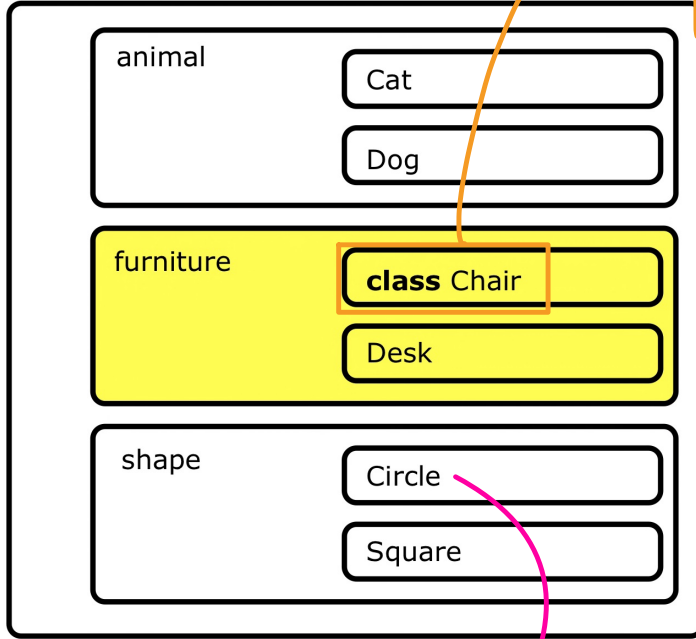
inheritance
(e.g. Lab 4)

classes.

attributes / methods

Visibility: Classes

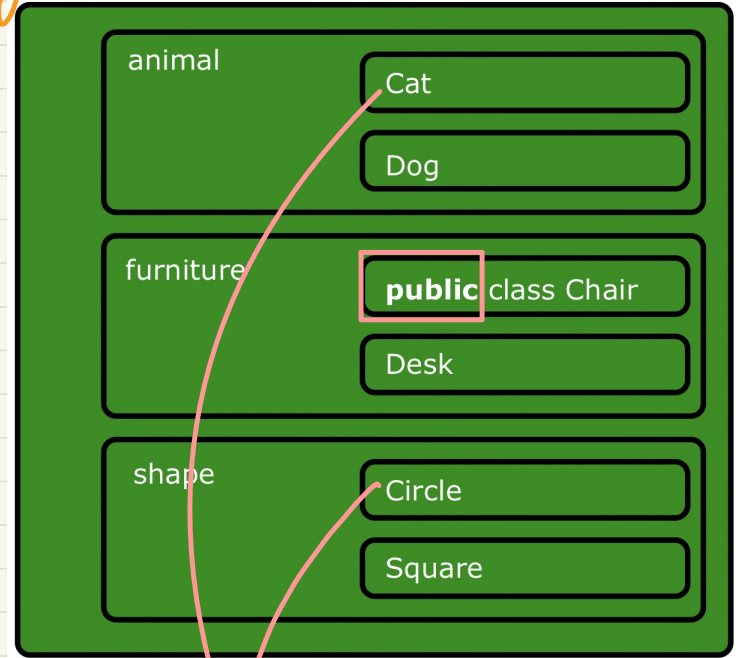
CollectionOfStuffs



visible in the resident package (package level)

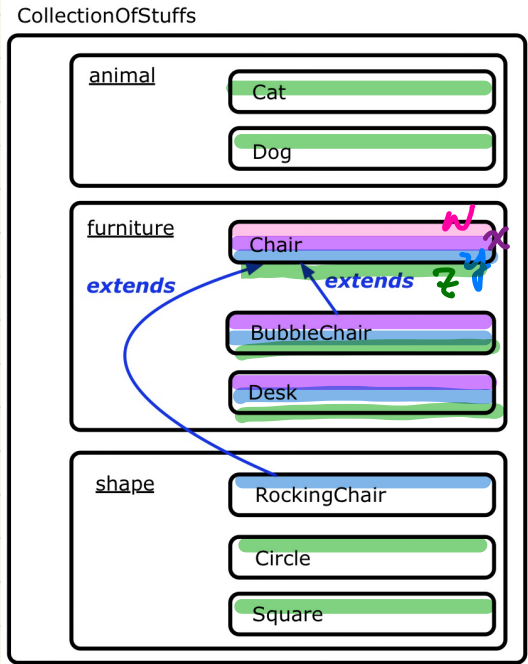
X Chair C's

CollectionOfStuffs



Chair C's ✓

Visibility: Attributes and Methods

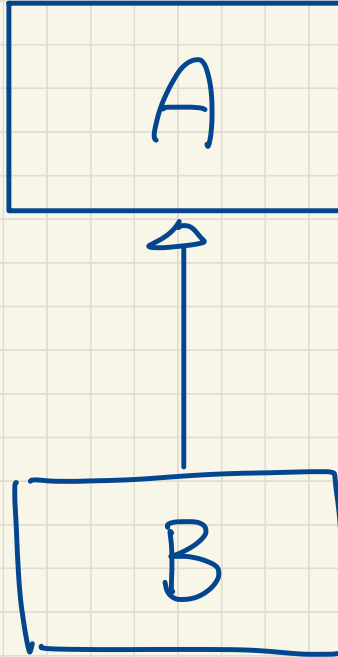


```

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

package level + inheritance (same or not package)

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



(1) declare all attributes to be

private

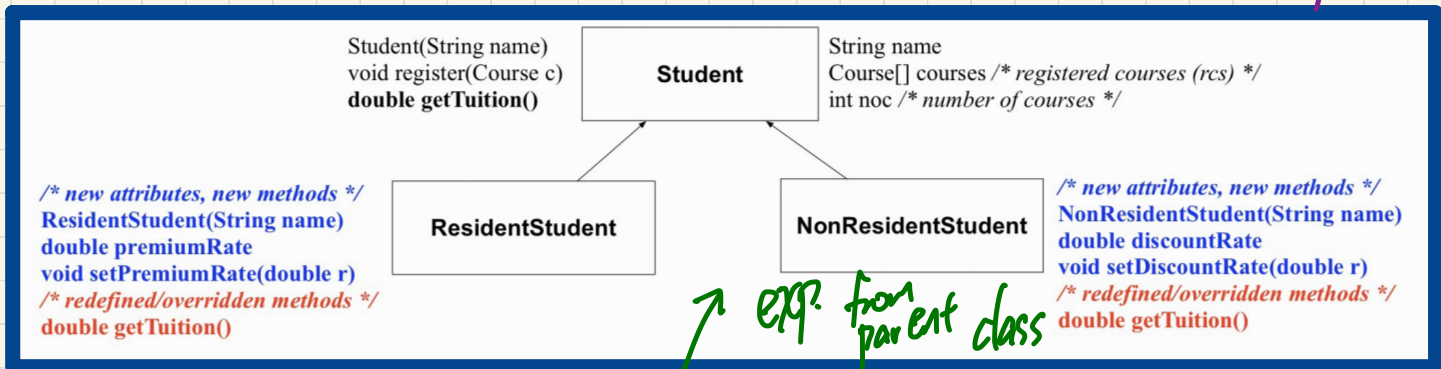
↳ not inherited.

(2) those attributes that should be inherited to B, make them protected.

↓
or no modifiers.

* each child class inherits exp. of its parent. range of attr/method callable static/declared type

Student Classes (with inheritance): Expectations



expectations of a ref. var. are determined by its static type

```

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

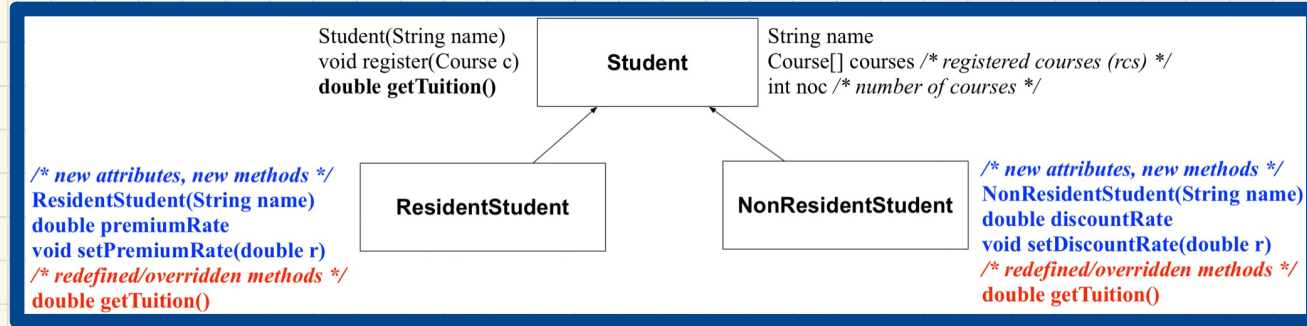
st: S.	name	rcs	noc	reg	getT	pr	setPR	dr	setDR
S.									
rs.									
nrs.									

st: S.
rs: RS
nrs: NRS

child-specific exp.

Intuition: Polymorphism

4. assumption is wrong
! $rs = s$ should not compile.

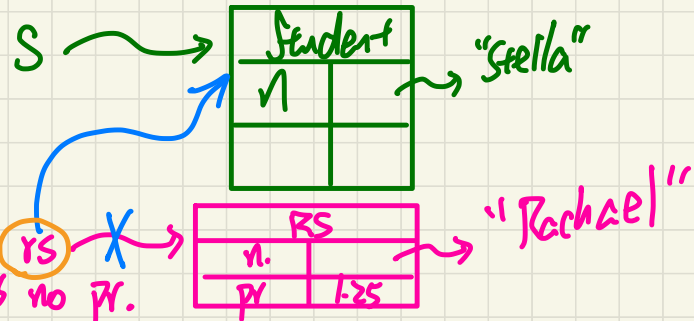


```
1 Student s = new Student("Stella");
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.25);
4 s ✓ rs; /* Is this valid? */
5 rs ✗ s; /* Is this valid? */
```

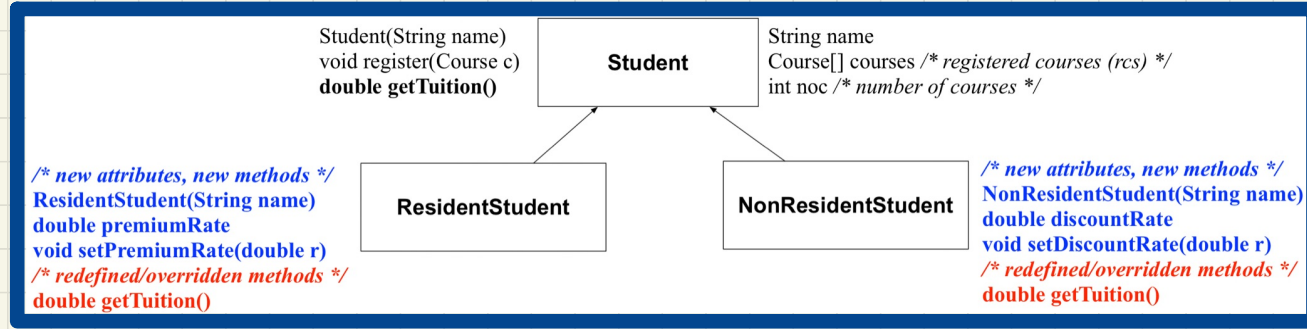
1. Assume $rs = s$ compiles.

2. Execute the var assignment.

3. Exp of rs ' ST (RS) includes: pr
 $rs.pr \rightarrow$ crash ! Student has no pr.



Intuition: Dynamic Binding

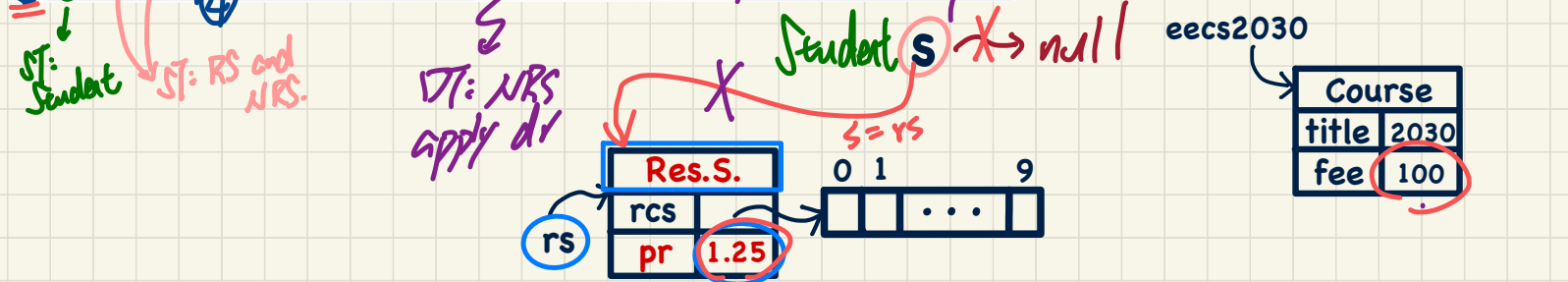


var s

step	ST	DT
①		null
②		RS
③	Student	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());
    
```



Compilation: static type

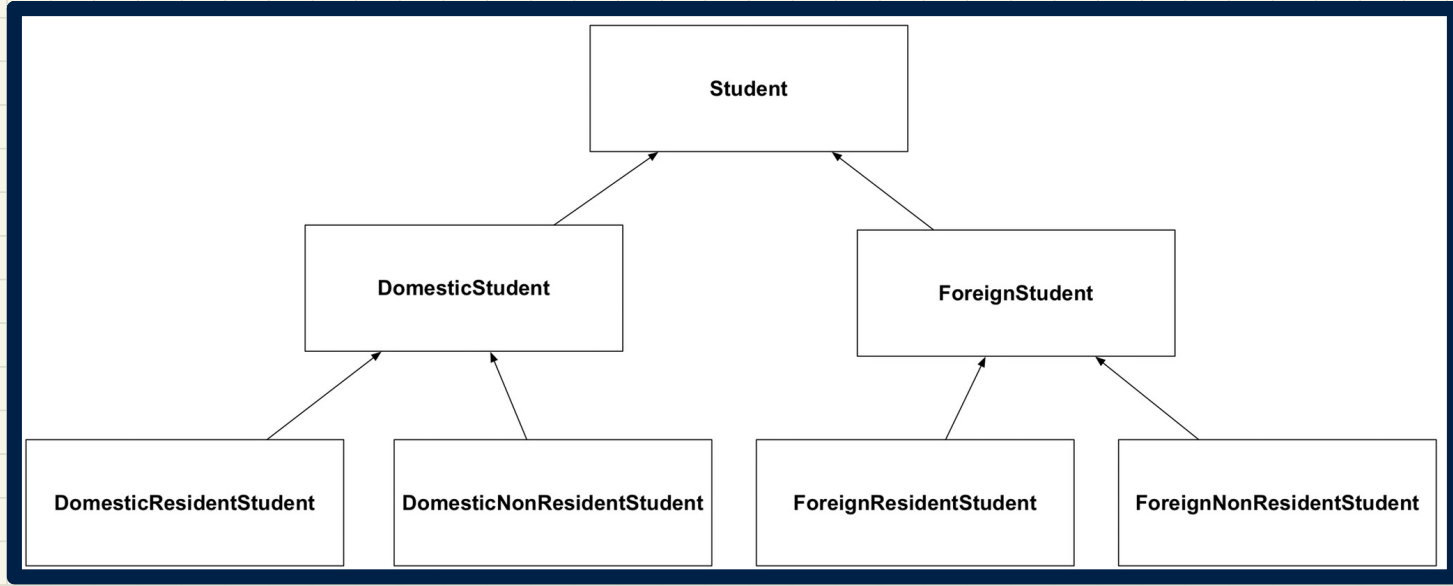
runtime: dynamic type

e.g. version
of method
called

e.g. exception

↓ `ClassNotFoundException`.

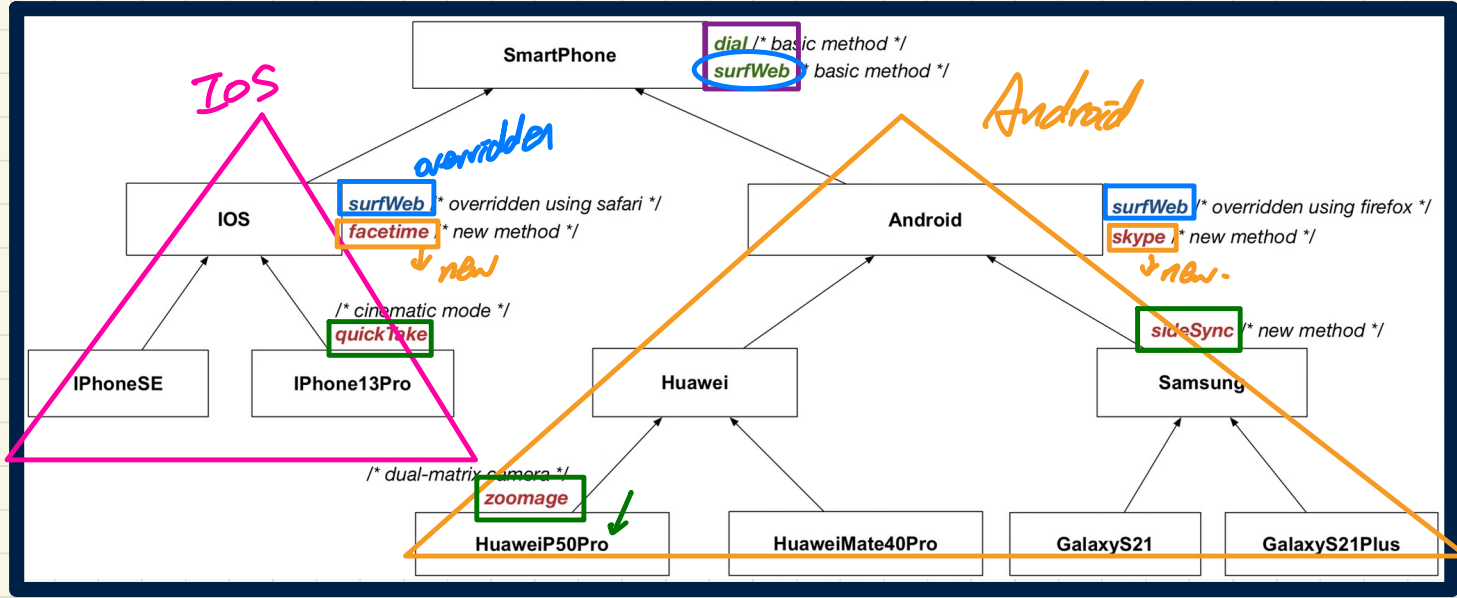
Multi-Level Inheritance Hierarchy: Students



Reflections:

- For **Design 1**, how many encodings to check for each method?
 - For **Design 2**, how many arrays to store for SMS?
 - For **Design 3**, where are common attributes/methods stored?
- kind var?*
how many classes
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 10

Inheritance

***Ancestors, Descendants, Expectations
Rules of Substitutions
Polymorphism vs. Dynamic Binding***

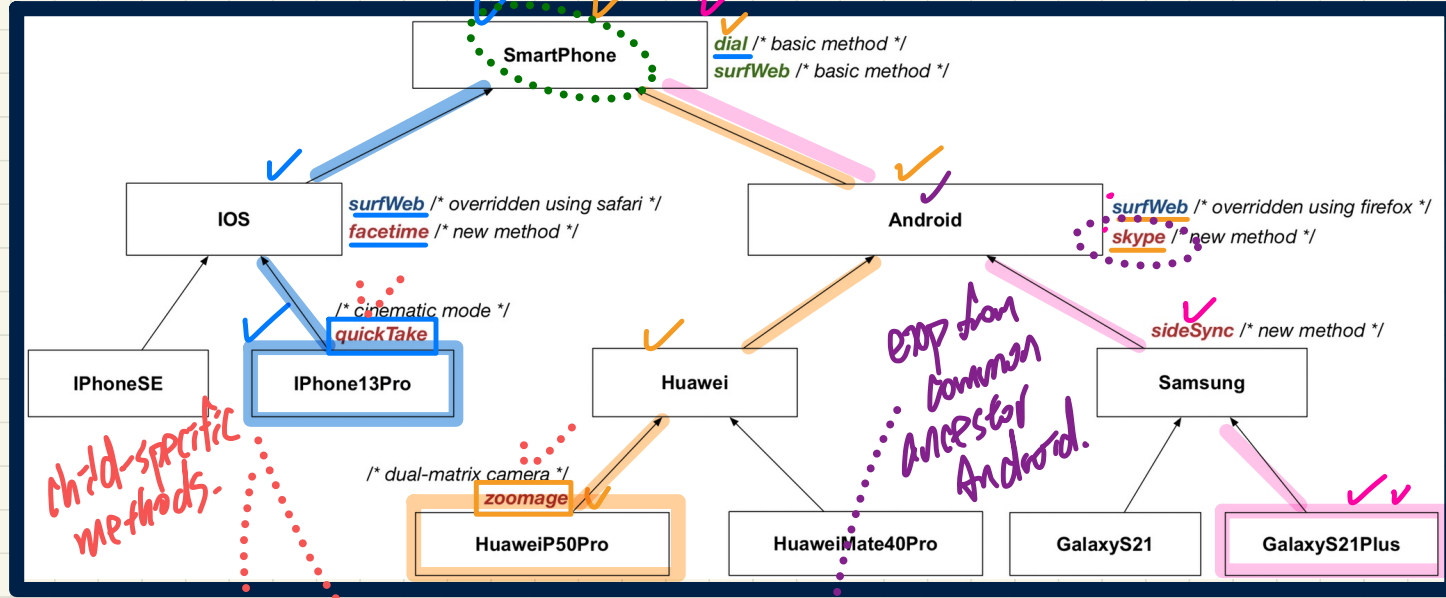
WSC 105

Announcements/Reminders

- Today's class: notes template posted
- Lab4 released
- WrittenTest2 guide and example questions to be released

Wed

Multi-Level Inheritance Hierarchy: Smartphones



Exercise Compare the ranges of expectations of:

- + iPhone13Pro
- + HuaweiP50Pro
- + GalaxyS21Plus

iPhone13Pro: *quickTake, facetime, surfweb, dial*
 HuaweiP50Pro: *zoomage, skype, surfweb, dial*
 GalaxyS21Plus: *side sync, skype, surfweb, dial*

→ common exp.
 inherited from Smartphone

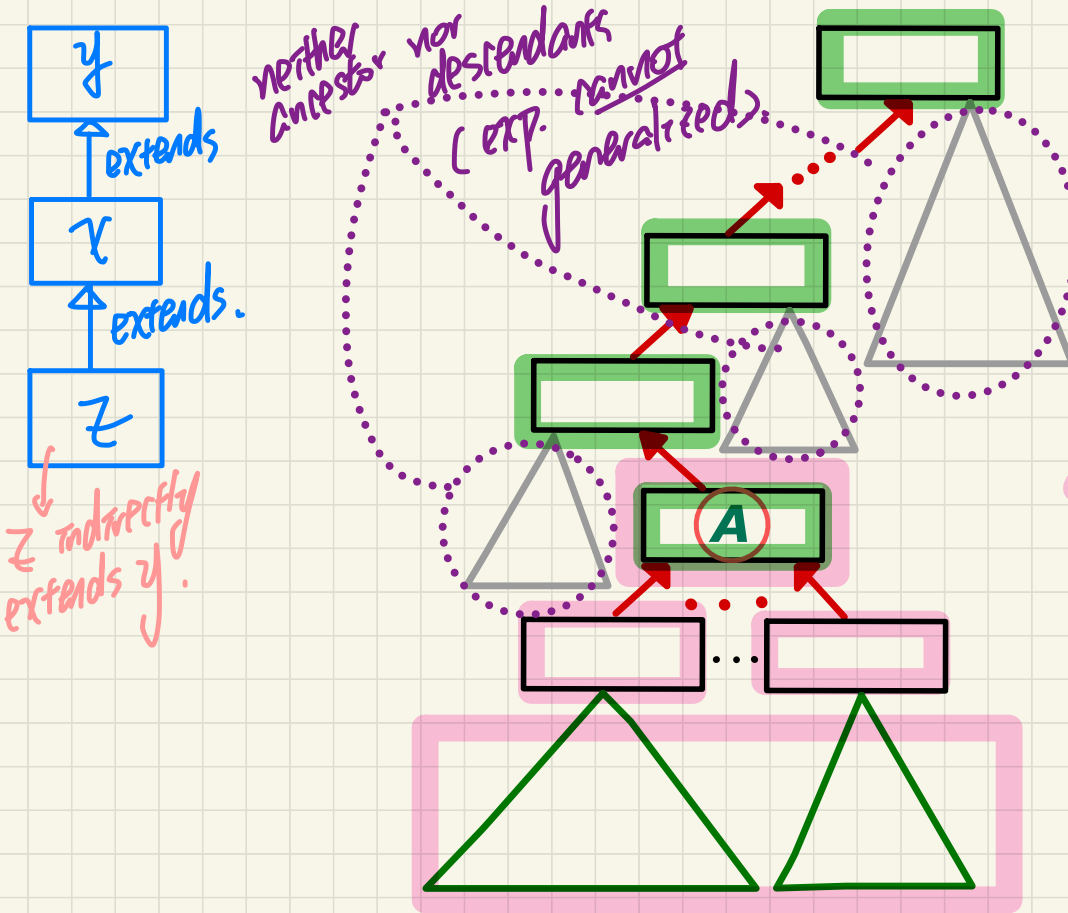
Inheritance Forms a Type Hierarchy

Ancestors of A

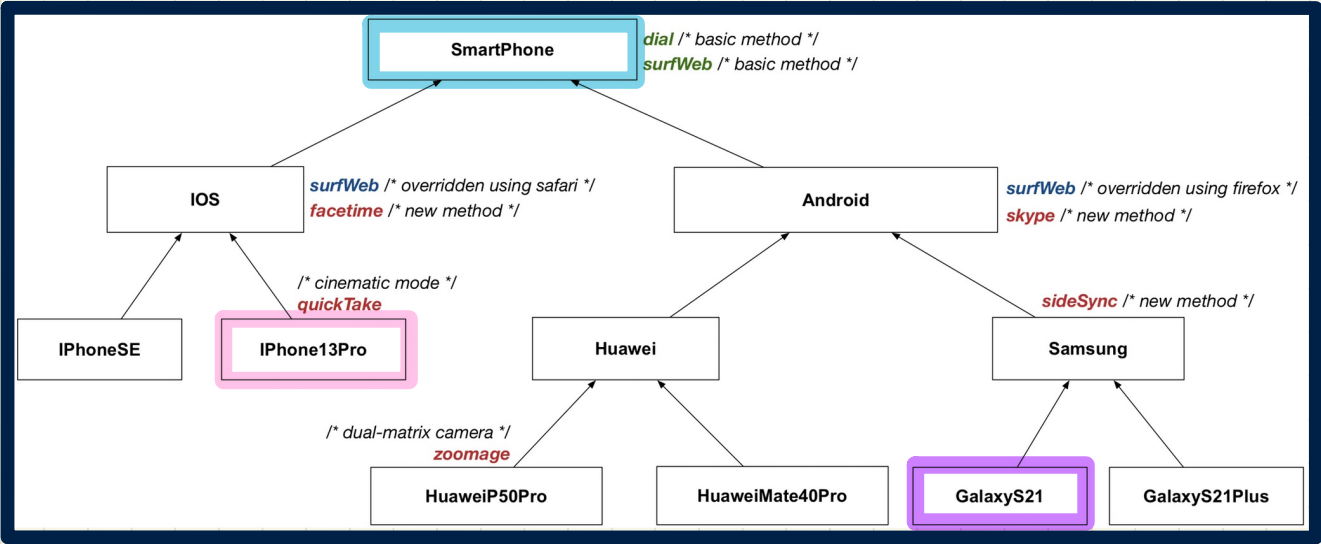
- (1) A accumulates the att./methods from its ancestors. exp.
- (2) $\text{exp}(A) \geq \text{exp}$ of any ancestor class

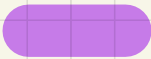
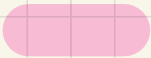
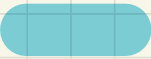
Descendants of A

- (1) $\text{exp}(A)$ inherited to all A 's descendants
- (2) $\text{exp}(A) \leq \text{exp}$ of any descendant class



Inheritance Accumulates Code for Reuse



	ancestors	expectations	descendants
	Gal, Sam, An, SP	 EXPECT.	Gal
			
	SP		all classes.

Rules of Substitutions

Compilation (static types).

* **OA's** expectation must be fulfilled by **ob's** ST: ?

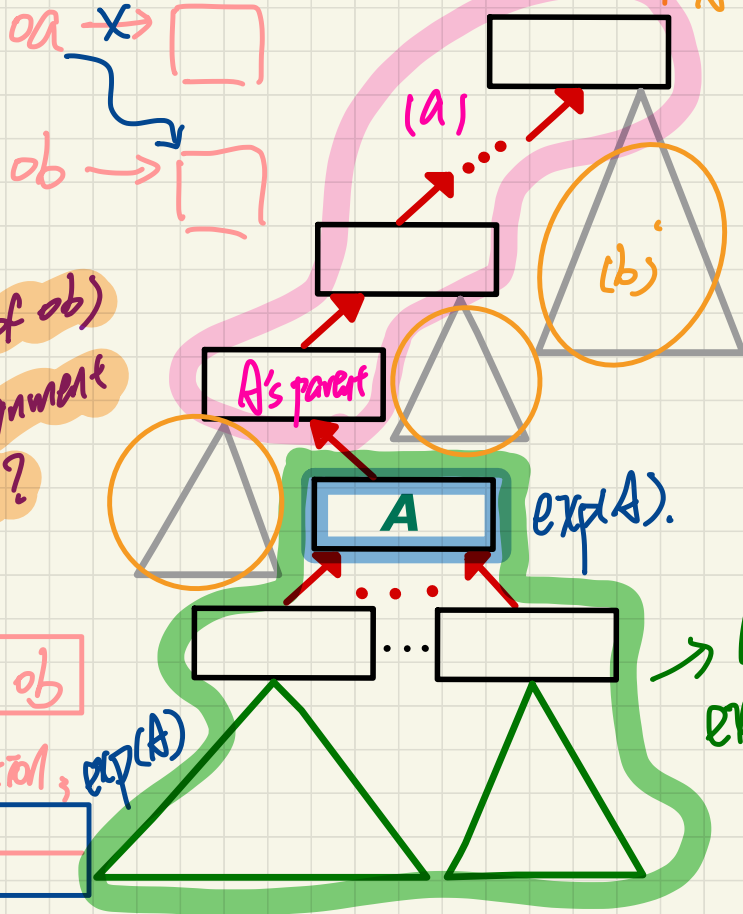
exp. : $\text{exp}(?) \geq \text{exp}(A)$

↳ ? must be a descendant of A

descendants of A
 $\text{exp} \geq \text{exp}(A)$
 if ? is one of these
 → fast to compile
 (a) ancestors of A's parent
 (b) neither an. nor des

ST of oa
 A oa = ...;
 ? ob = ...;
 oa = ob;
 ST of ob.
 what can ? (ST of ob) be in order for this var. assignment to compile?

• substitute **oa** by **ob**
 • after the substitution, we expect: oa.*



does this
var assignment
(substitution)
compile?

var1

$=$

var2

$\text{ST: } T_1$

$\text{ST: } T_2$

(inheritance).

Rule of substitutions

$\text{exp}(T_2) \supseteq \text{exp}(T_1)$

↓
 T_2 must be a descendant of T_1

Inheritance Accumulates Code for **Reuse**

* Q. What var can appear on the RHS

without compilation error?

(A1)

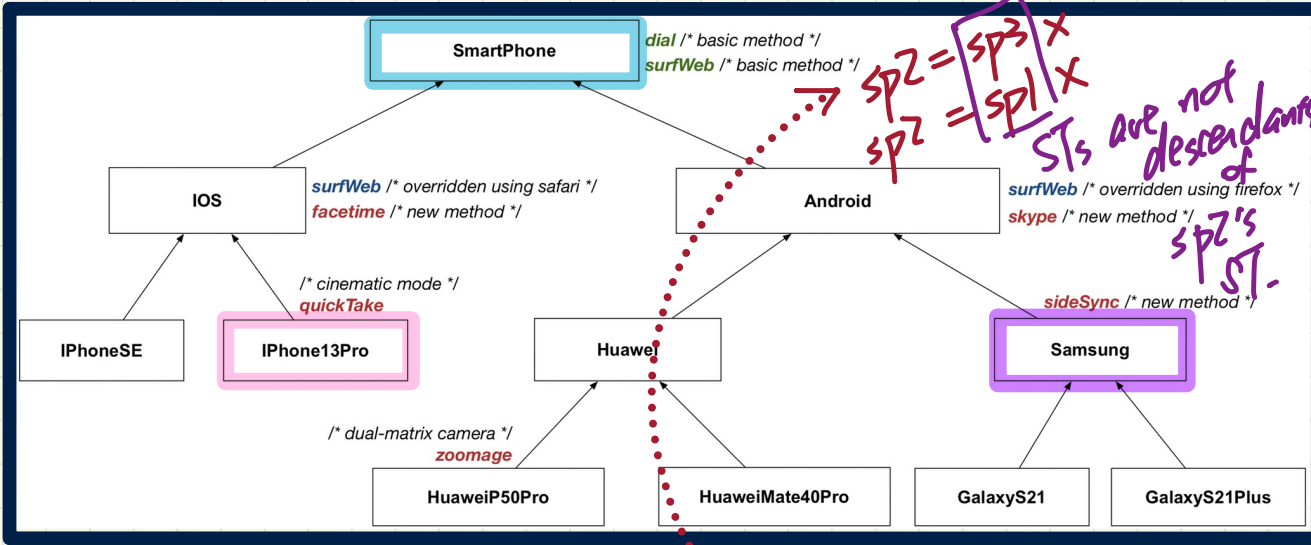
Any var whose ST

is a descendant

of spl's ST can appear on RHS.

(A2) to substitute

spl, its ST must have at least as large the exp of



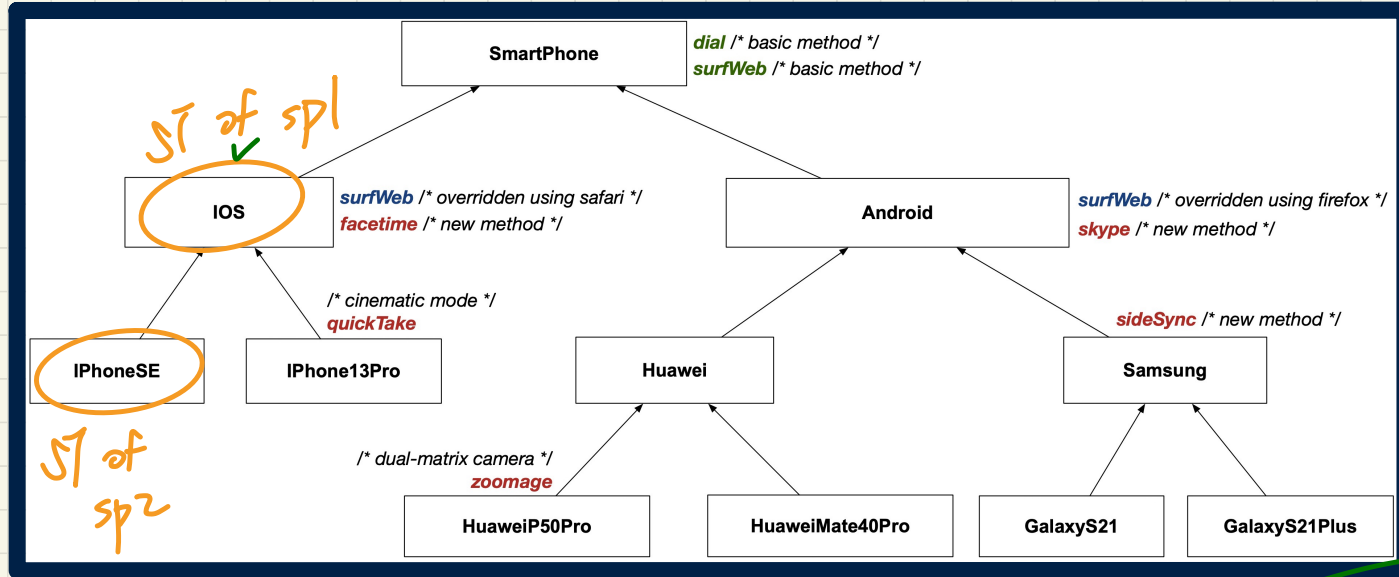
Handwritten notes: $sp2 = sp3$, $sp2 = sp1$, and "STs are not descendants of sp2's ST."

SmartPhone sp1;
 iPhone13Pro sp2;
 Samsung sp3;

Handwritten code snippets: $sp1 \oplus ?$, $sp2 \oplus ?$, $sp3 = ?$.

spl's ST exp.

Rules of Substitutions (1)



sp2's ST can fulfill the exp of sp1's ST.

Declarations:

IOS sp1;

iPhoneSE sp2;

iPhone13Pro sp3;

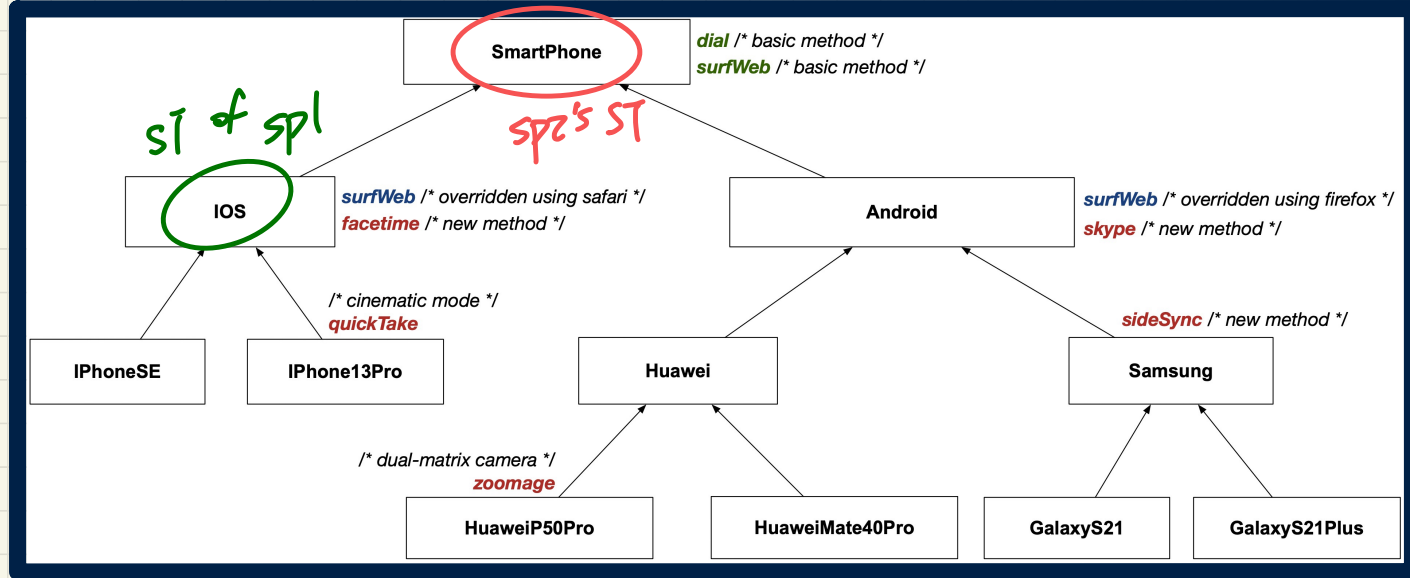
Substitutions:

sp1 = sp2;

sp1 = sp3;

Compiles
"∵ ST of sp2 (IPSE) is a descendant of ST of sp1 (IOS).

Rules of Substitutions (2)



Declarations:

IOS sp1;
SmartPhone sp2;

Substitutions:

sp1 = sp2;

↓
fail to compile.

Visualization: **Static** Type vs. **Dynamic** Type

Declaration:

Student s;

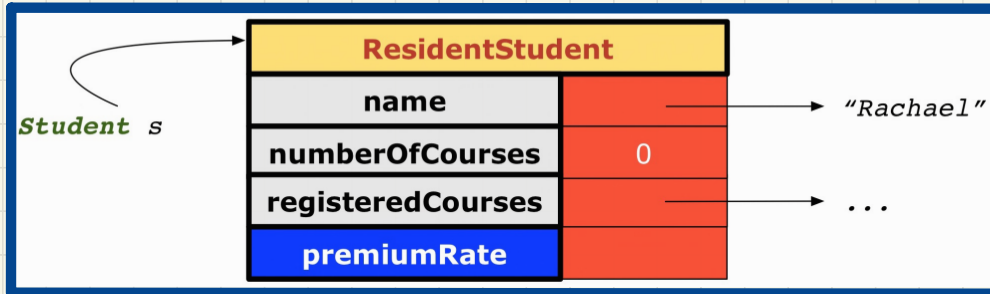
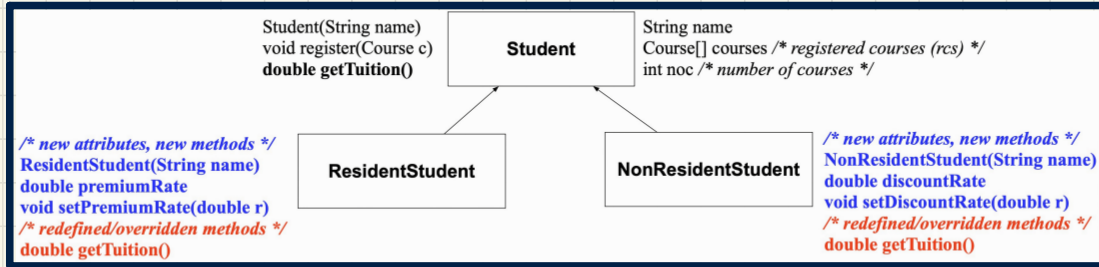
Substitution:

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

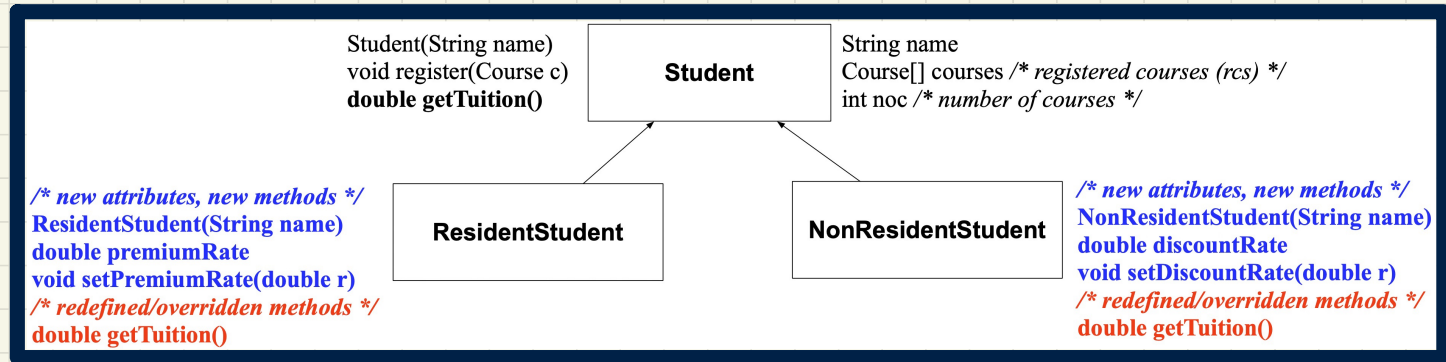
Static Type: Expectation

Dynamic Type: Accumulation of Code

new dt. for valid S → if RS can fulfill the exp. of the ST of s (Student).



Change of Dynamic Type (1.1)



Example 1:

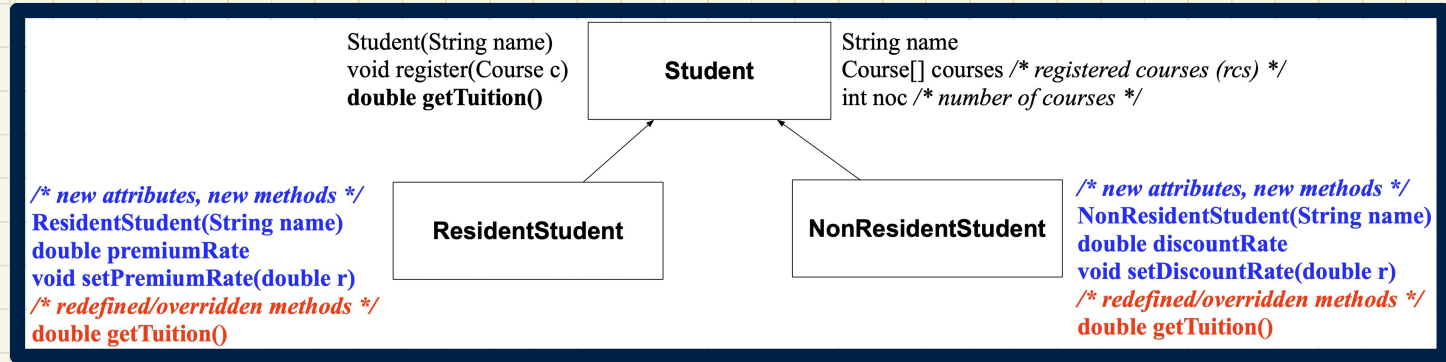
✓ *compiles - RS is a descendant of Student*

Student jim = new ResidentStudent(...);

jim = new NonResidentStudent(...);

*compiles
- NRS is a descendant
of Student*

Change of Dynamic Type (1.2)

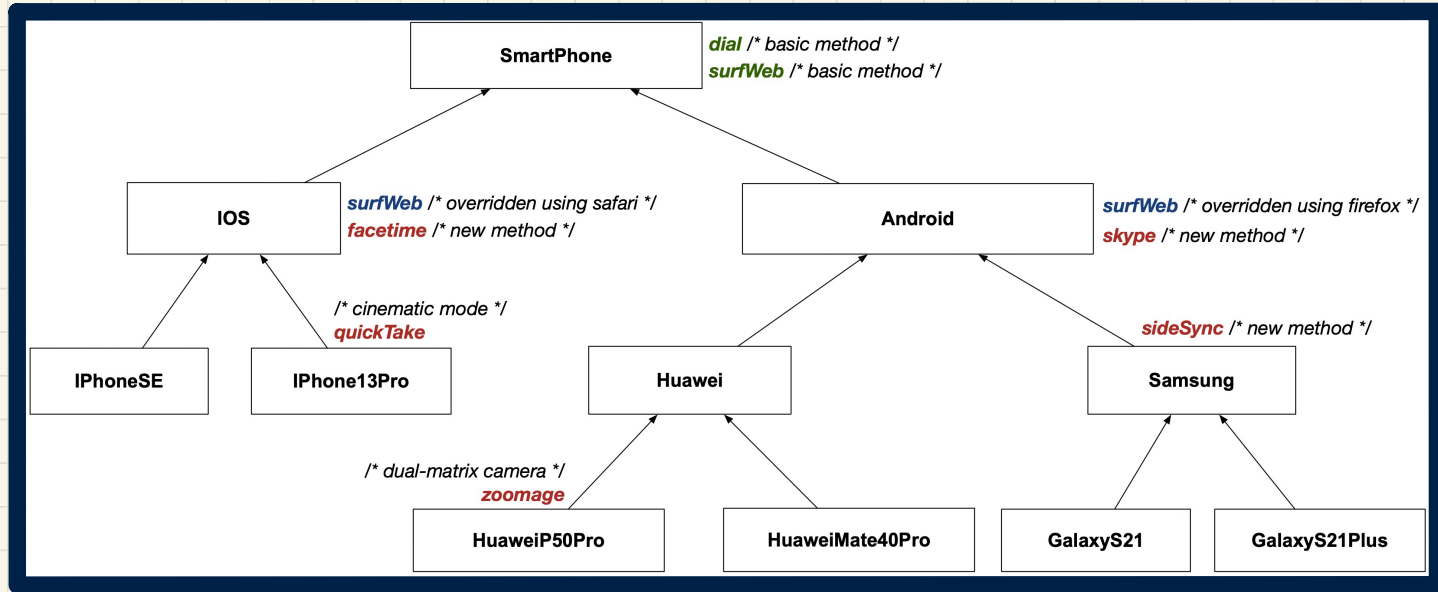


Example 2:

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

fail to compile
∴ **Student** is not a descendant of **RS**
proposed ST of jeremy.

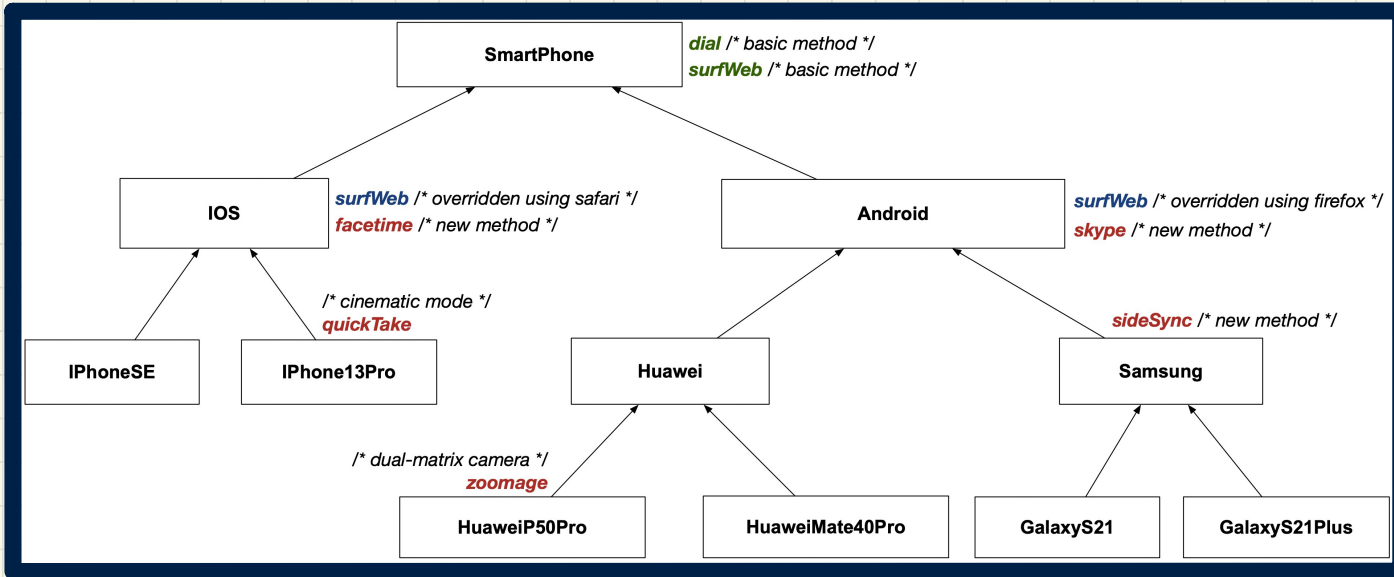
Change of **Dynamic** Type: Exercise (1)



Exercise 1:

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

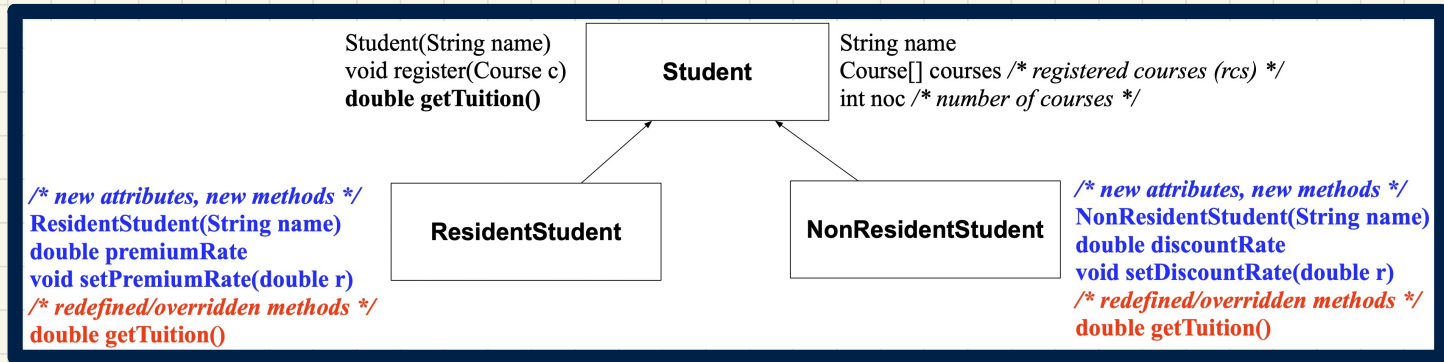
Change of Dynamic Type: Exercise (2)



Exercise 2:

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

Change of Dynamic Type (2.1)



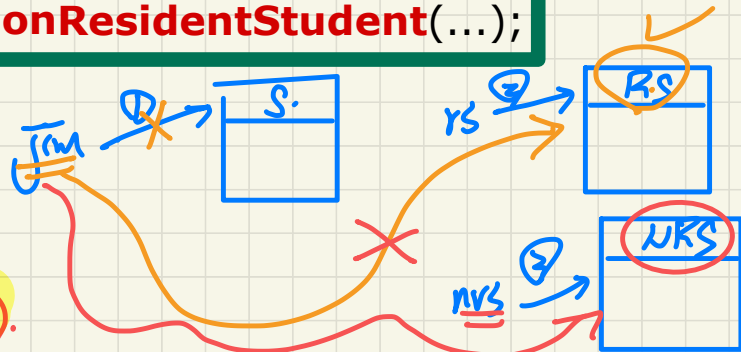
Given:

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

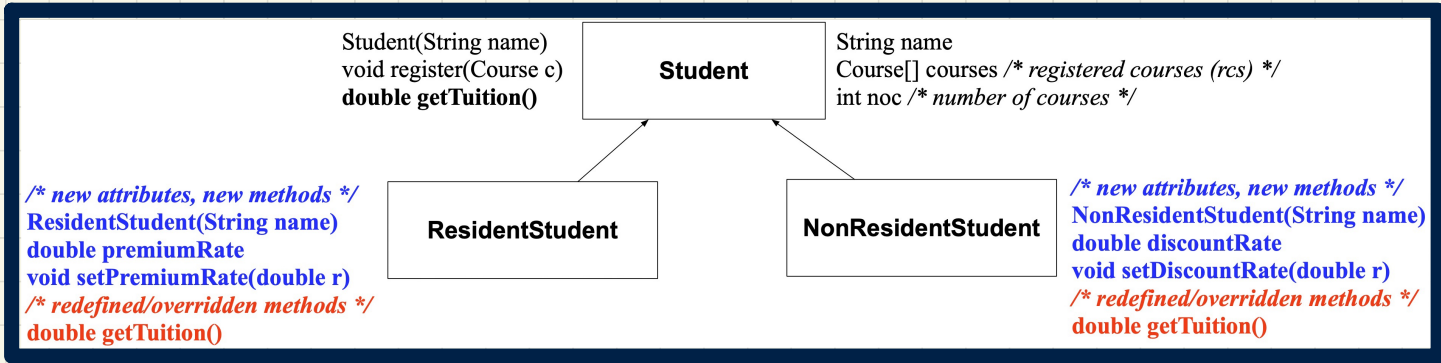
Example 1:

④ jim = rs; | DT of jim: changed to the DT of rs (RS)

⑤ jim = nrs; | DT of jim: changed to the DT of nrs (NRS).



Change of Dynamic Type (2.2)



Given:

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

Example 2:

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

not compile
∵ jim's ST (Student)
not descendant of
rs' ST (RS)
cannot fulfill
exp of.

Polymorphism and Dynamic Binding

meny

Polymorphism: shapes (DTs).

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

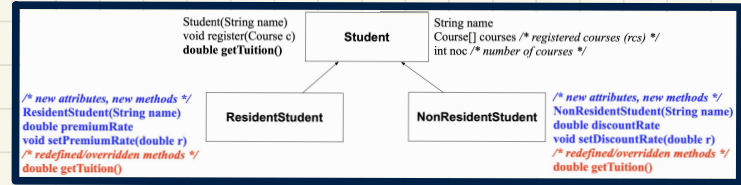
⇒ Each **dynamic type** has its **version** of method.

Dynamic Binding:

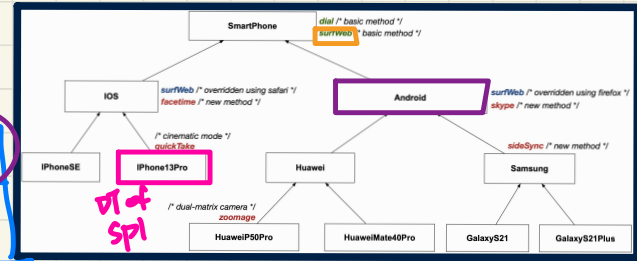
An object's **dynamic type** determines the **version** of method being invoked.

each DT may have its own version of method.

```
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();
```



sp1 → IP13P
sp2 → GS21
↓ version Line Android

Lecture 19 - Nov 12

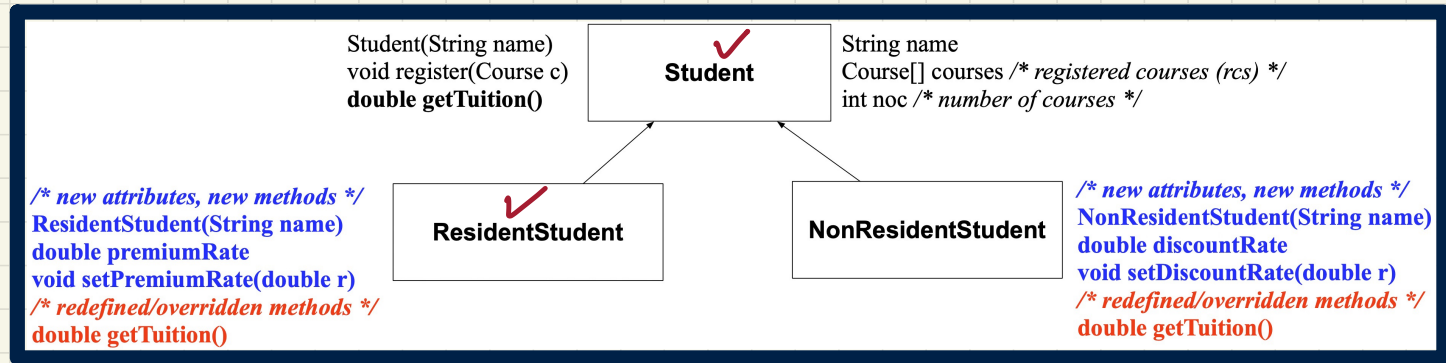
Inheritance

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

Announcements/Reminders

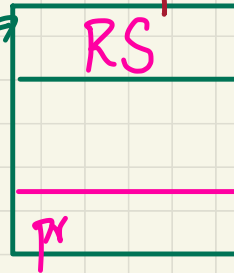
- Today's class: [notes template](#) posted
- **Lab4** released
- **WrittenTest2** guide and **example questions** released
- Notes on Type Casting

Type Cast: Motivation



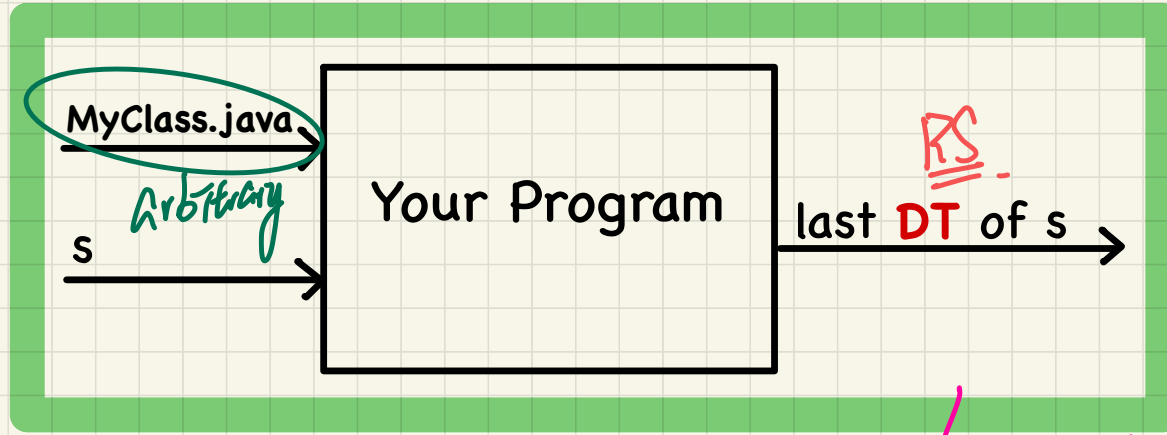
```
1 Student jim = new ResidentStudent("J. Davis");
2 ResidentStudent rs ✗ jim;
3 rs.setPremiumRate(1.5);
```

Student jim



x compile : jim's ST (S) not descendant of RS.

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



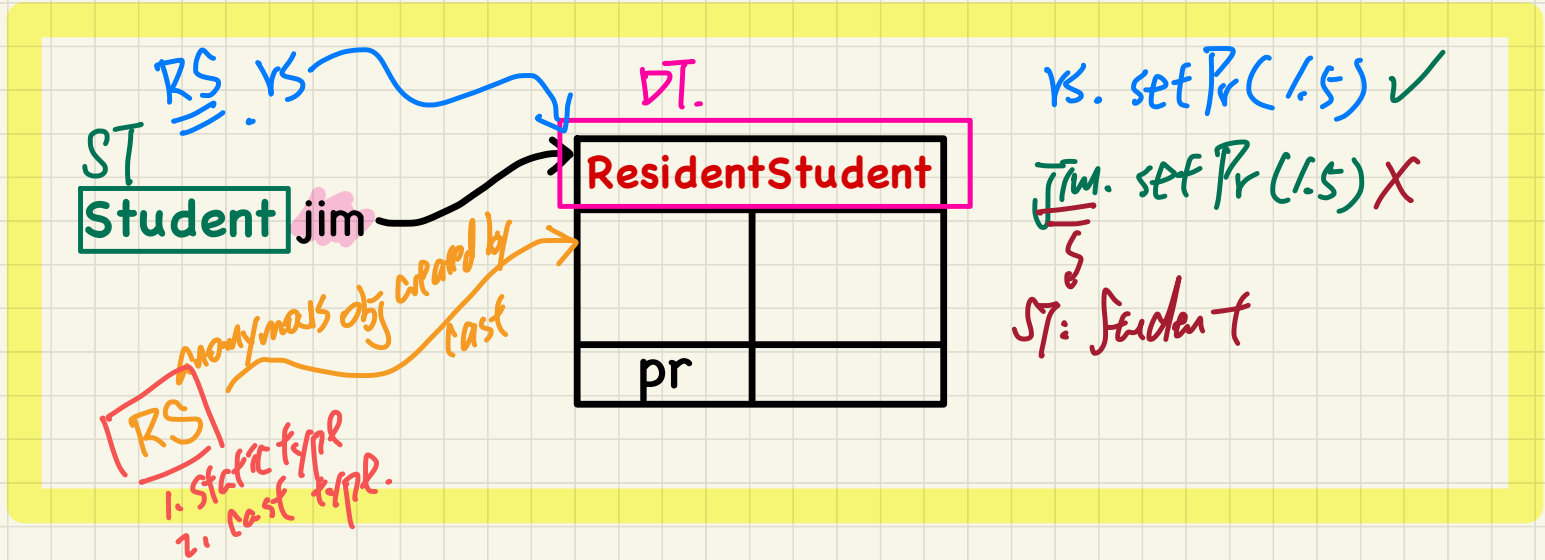
may not terminate

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

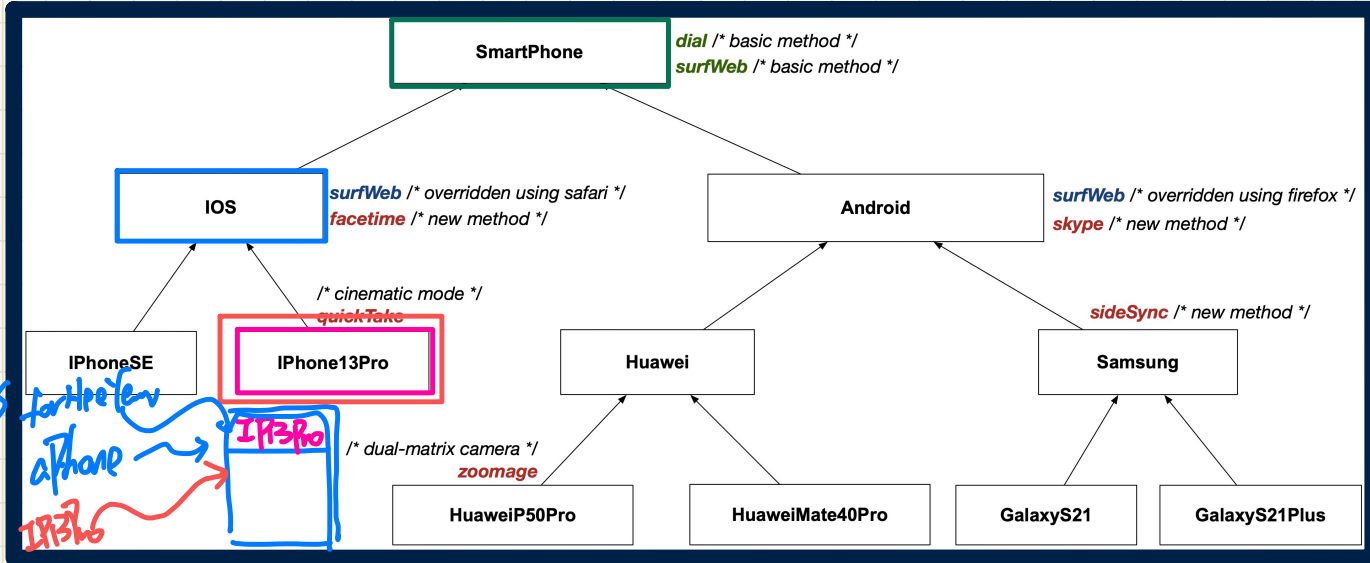
undecidable problem.
(not feasible to write a prog that infers the dynamic behaviour of another program)

Anatomy of a Type Cast

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



Type Cast: Named vs. Anonymous



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

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

Exercise

```
SmartPhone aPhone = new iPhone13Pro();
(IPhone13Pro) aPhone.facetime();
```

Anonymous Cast: Use the cast result directly.

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

↓ C.O. is a type cast with ST IP3Pro.

AS if: (IP3Pro) (aPhone).facetime()

Compilable Casts: Upwards vs. Downwards

Expectations

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

Android myPhone = new GalaxyS21Plus();

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

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

sp. skype X

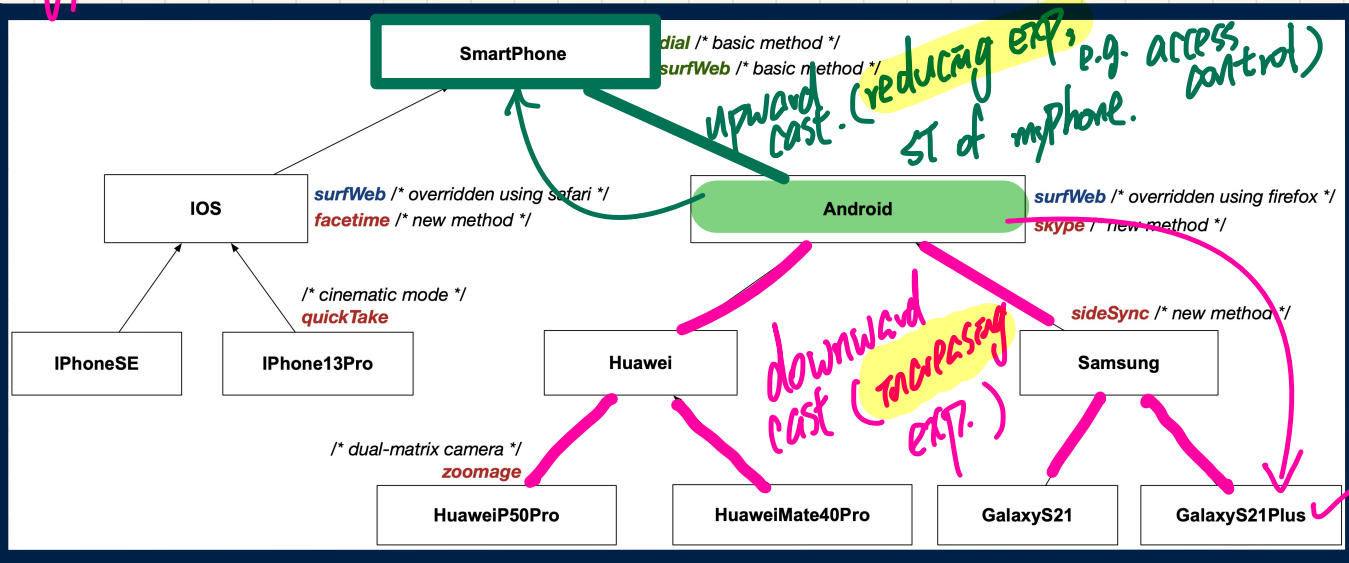
downward

x to consider compilable cast; disregard DT.

upward

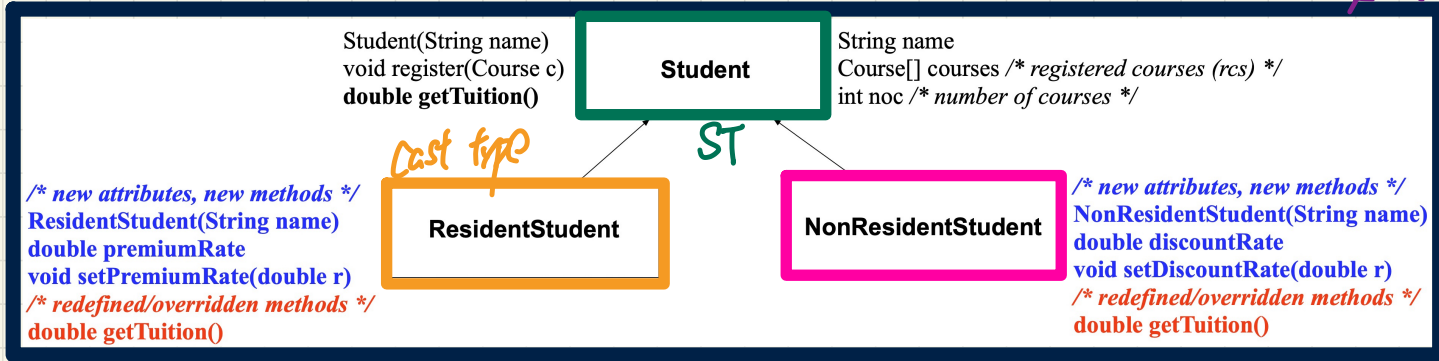
upward cast. (reducing exp, e.g. access control) ST of myPhone.

downward cast (increasing exp.)



Compilable Type Cast May Fail at Runtime (1)

(b) DT of `Jim` (NRS) is not a descendant of cast type (RS)

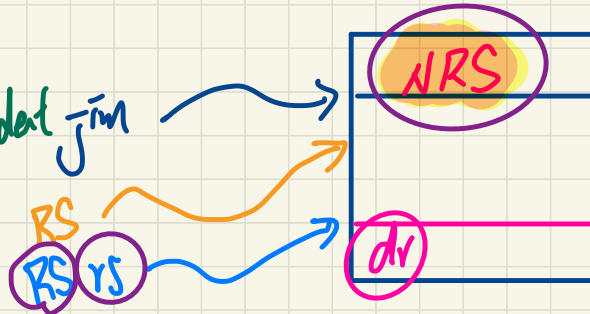


```

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

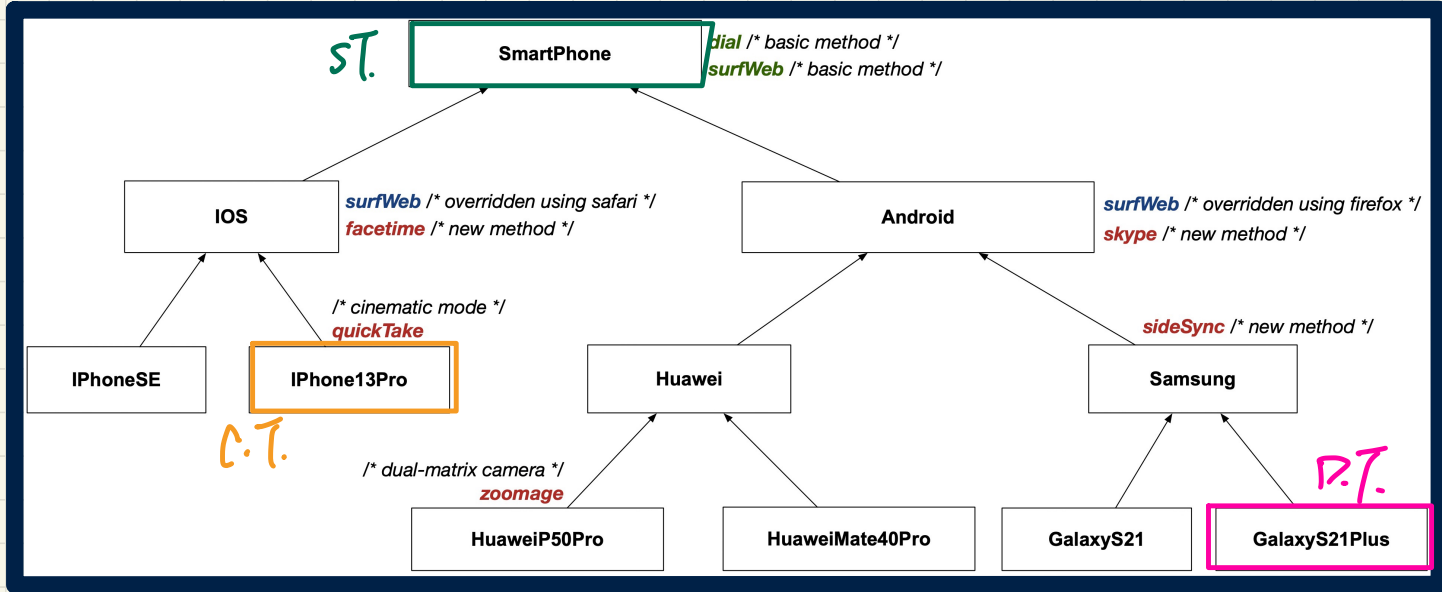
Compiles: downward cast

problematic
 DT NRS
 does not support pr.



(1) Compiles: downward cast
 (2) at runtime: ClassCastException
 (a) DT of jim (NRS) cannot fulfill exp. of cast type (RS)

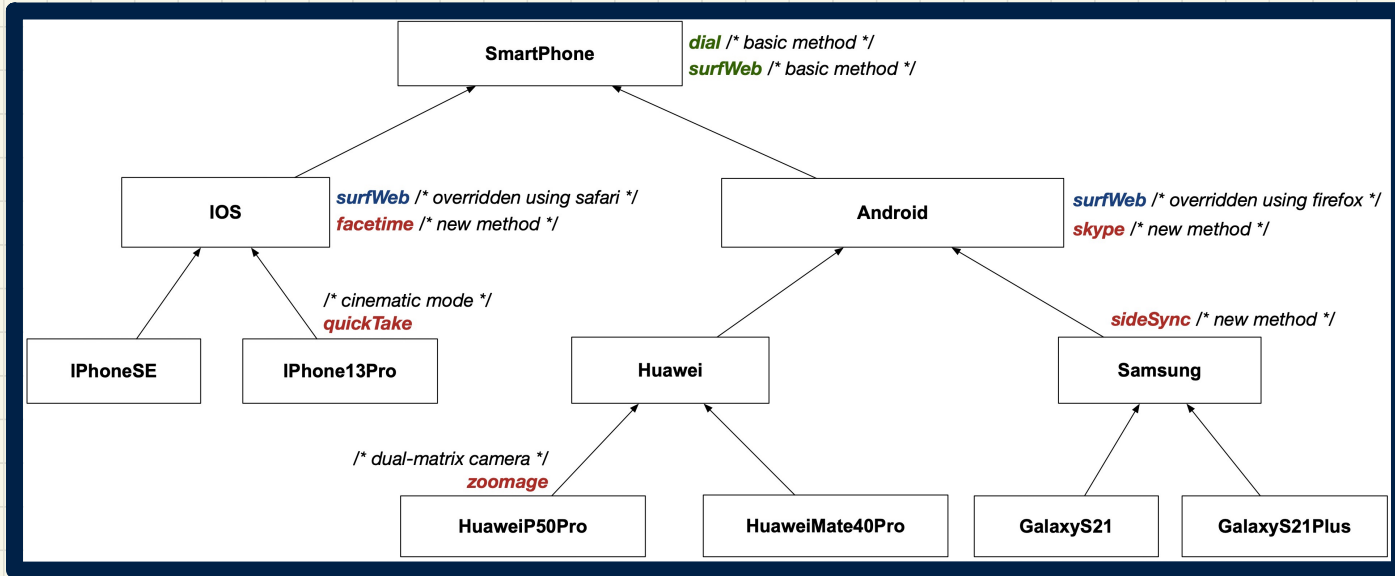
Compilable Type Cast May **Fail** at Runtime (2)



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

↓
Compiles but CCE
C.T. D.T. GalaxyS21Plus not descendant of C.T. iPhone13Pro

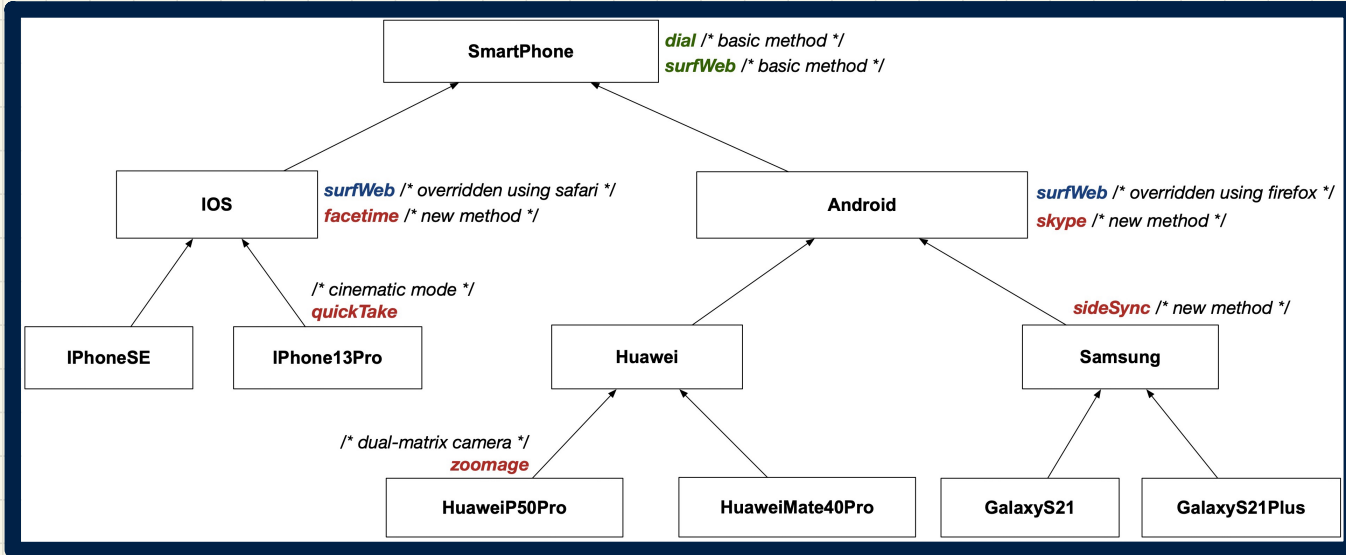
Exercise: Compilable Type Cast? **Fail** at Runtime? (1)



```
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)



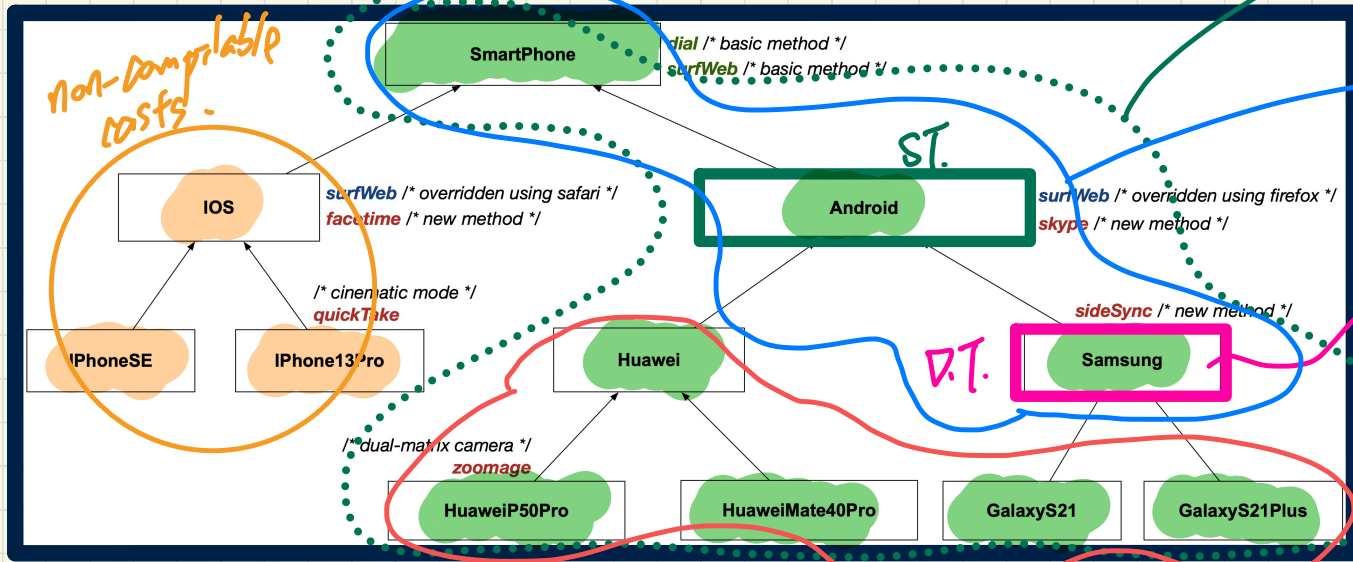
```
SmartPhone myPhone = new Samsung();  
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */  
iPhone13Pro ip = (iPhone13Pro) myPhone;
```

Compilable? ClassCastException at runtime?

Compilable Cast vs. Exception-Free Cast

ancestors of Samsung has $\leq \text{exp}(\text{Samsung})$

compilable casts.



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

Compilable Casts

Non-Compilable Casts

Exception-Free Casts

ClassCastException

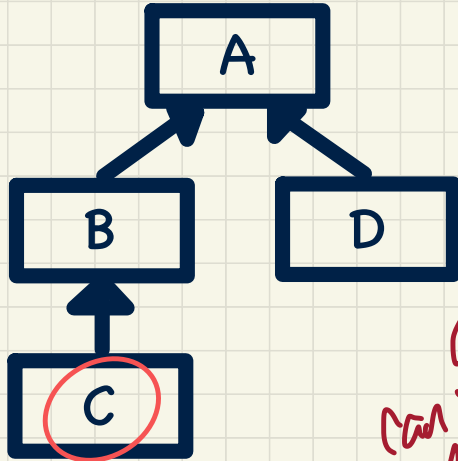
not ancestors of D.T. cannot full fill their exp.

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;
```

x compile: D is neither ancestor nor descendant of ST. of b (B)



$D \ d = (D) \ (A) \ b$

upward cast

downward cast

No. cast type D is not an ancestor of P.T. C. $\rightarrow$ CCE!!
Can P.T. C fulfill D? $\rightarrow$ Compiles!

$(D) \ (A) \ b$

P.T.: C
can C exp(A)?

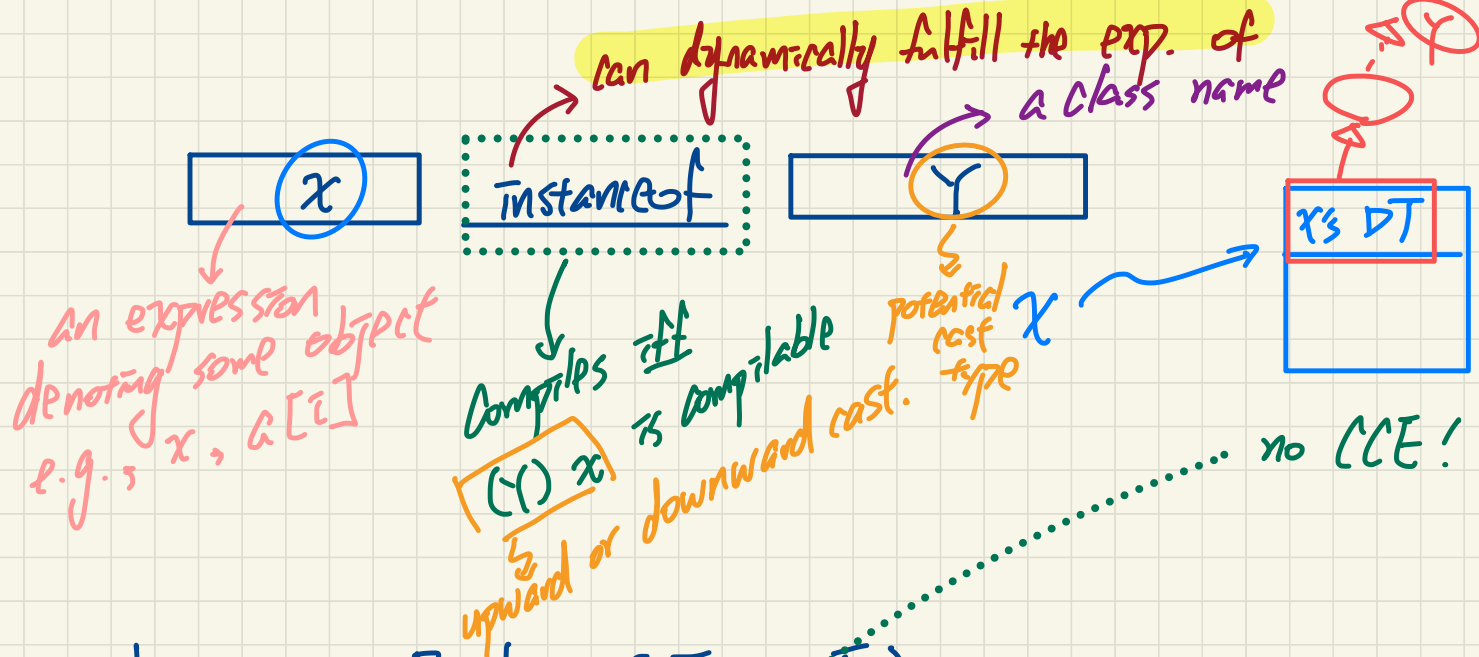
Lecture 20 - Nov 17

Inheritance

instanceof vs. Compile 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



↳ evaluates to Boolean (T or F)

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

(2) DT of x is descendant of Y

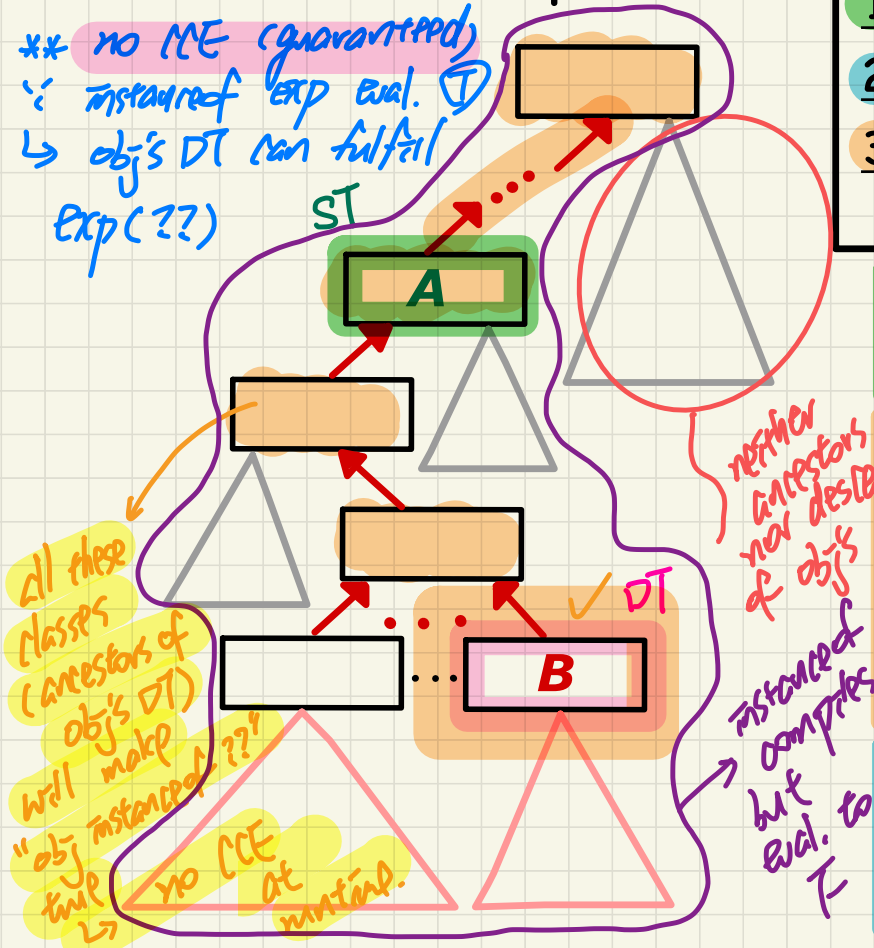
(3) Y is an ancestor of x 's DT

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

false otherwise

The instanceof Operator

** no ME (guaranteed)
 ∴ instanceof exp eval. ①
 ↳ obj's DT can fulfill
 exp(??)



* True iff obj's DT can fulfill exp(??)

```

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

assume: obj instanceof ??
 obj instanceof to true.

- L1 compiles if **B** can fulfill expectations of **A**.

- L3:

- Compiles if

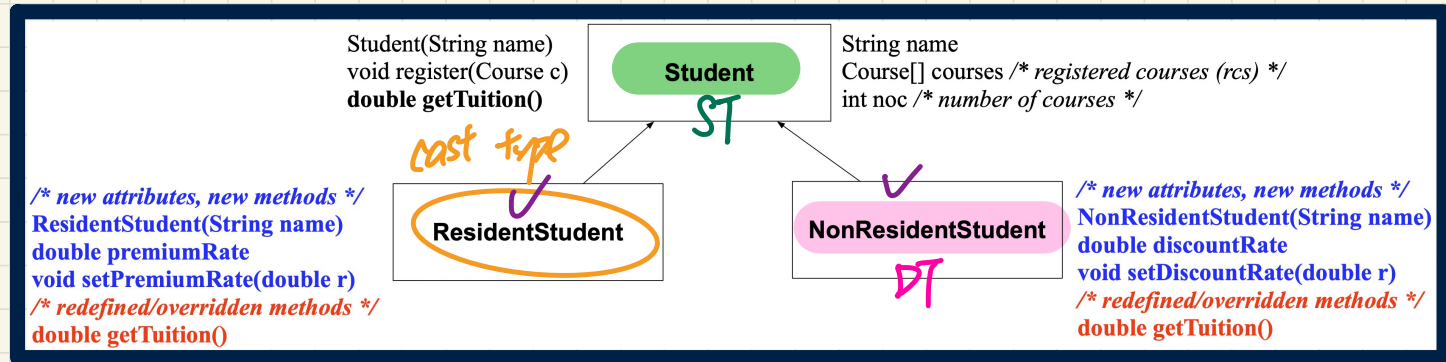
Up or Down cast w.r.t. **A**.

- ClassCastException if **B** cannot fulfill expectations on ??.

- L2:

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

Checking Dynamic Types at Runtime (1)



```

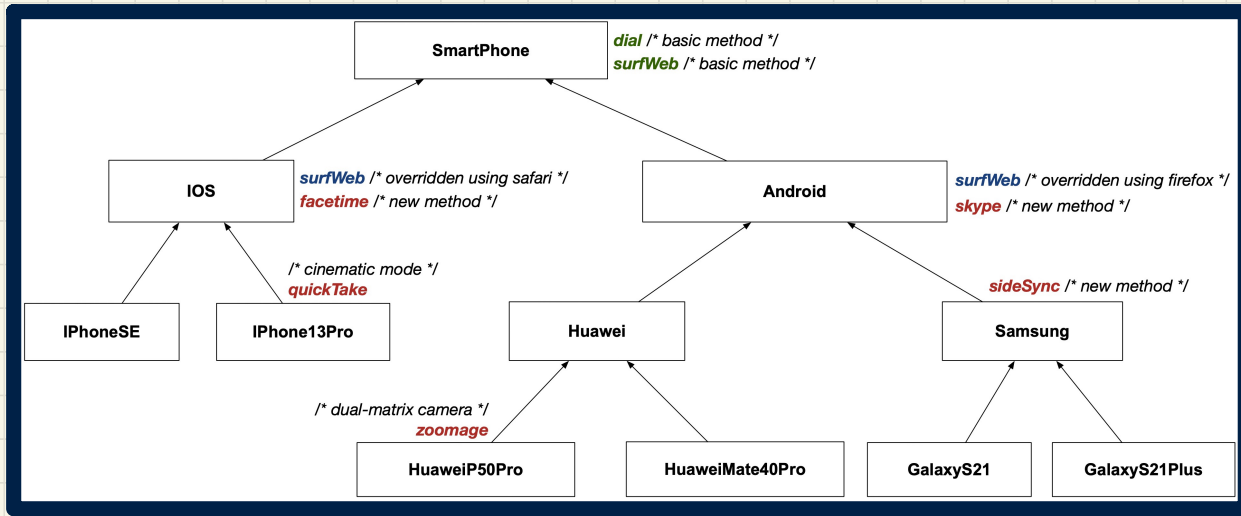
1 Student jim = new NonResidentStudent("J. Davis");
2 if (jim instanceof ResidentStudent) {
3     ResidentStudent rs = (ResidentStudent) jim;
4     rs.setPremiumRate(1.5);
5 }
  
```

bybassed → CCE prevented.

1. instanceof exp. (RS) jim. compiles
2. eval. to false
- jim's DT not descendant of RS
1. compiles -> downward cast
2. CCE -> DT of jim (NRS) cannot fulfil the mask type (RS).

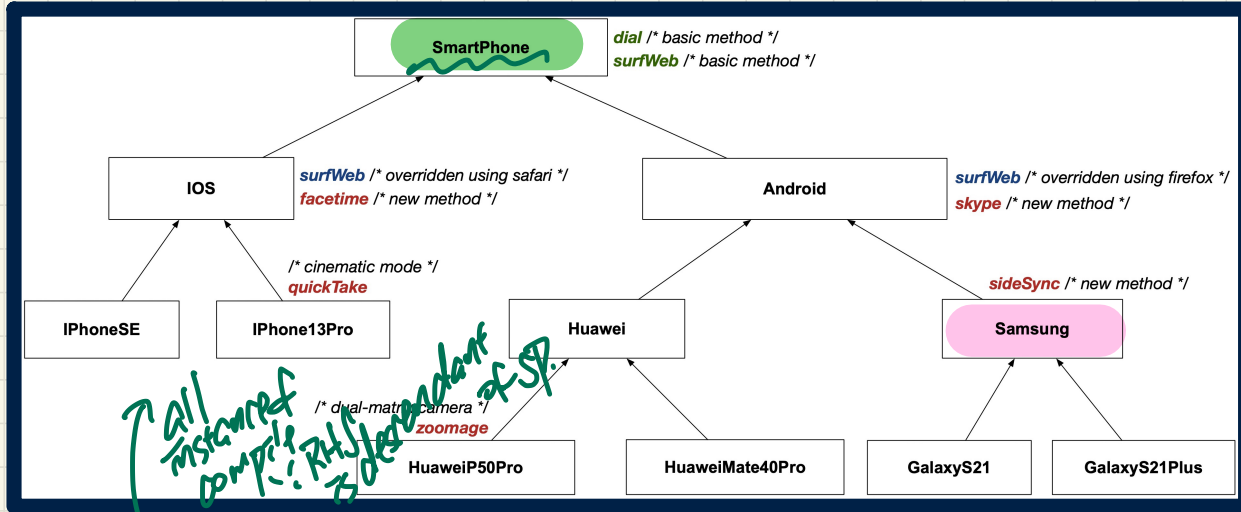
Checking Dynamic Types at Runtime (2)

Exercise



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

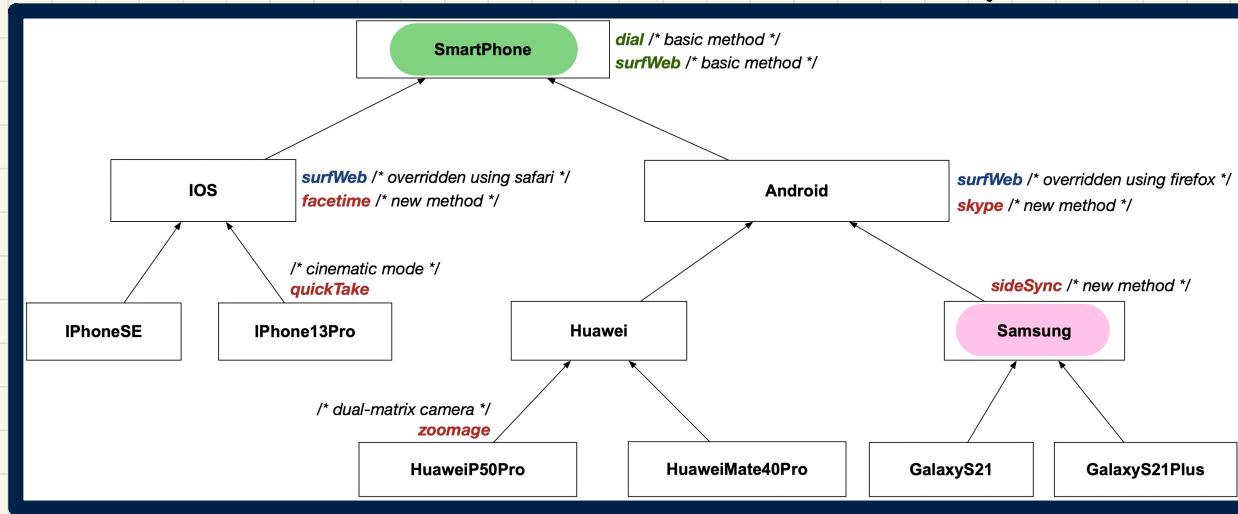
Use of the instanceof Operator



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

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

Safe Cast via Use of the instanceof Operator



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

by passed
prevent
CCE.

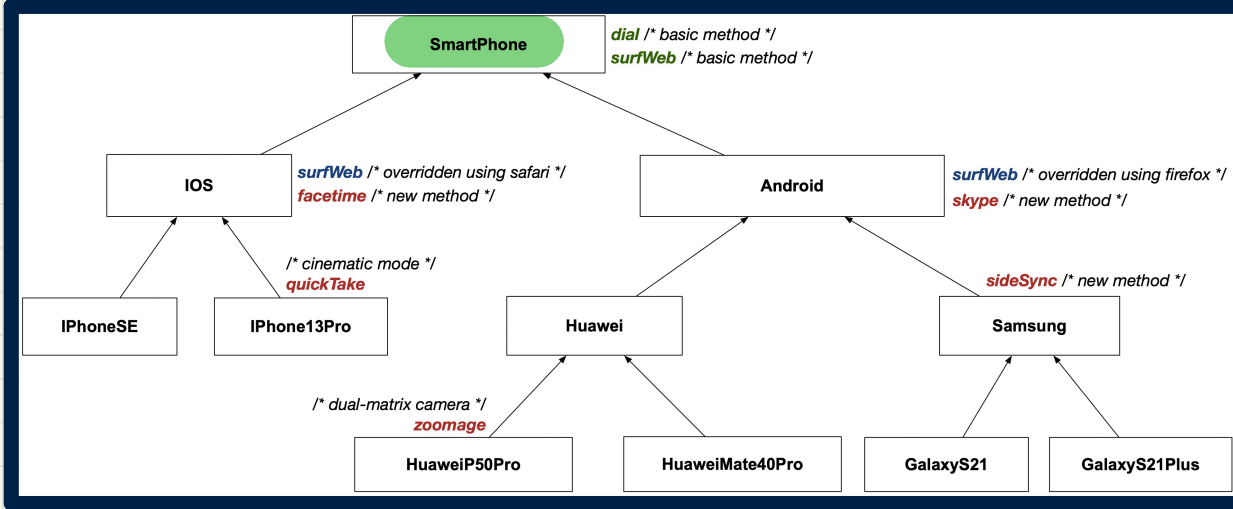
→ T → no CCE!

→ False

→ False

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

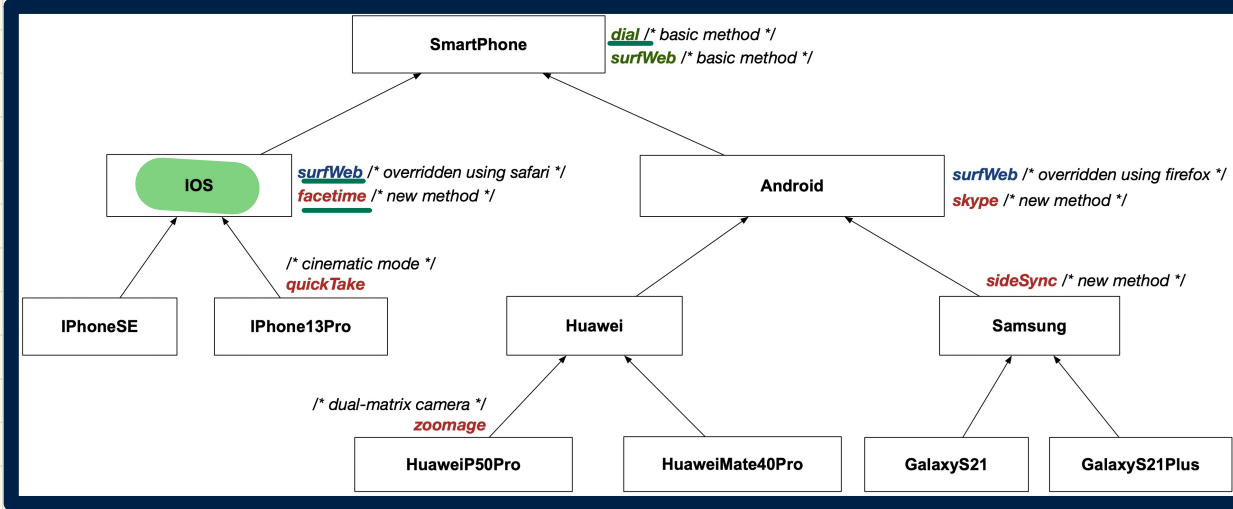
Static Types, Casts, Polymorphism (1)



```
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)

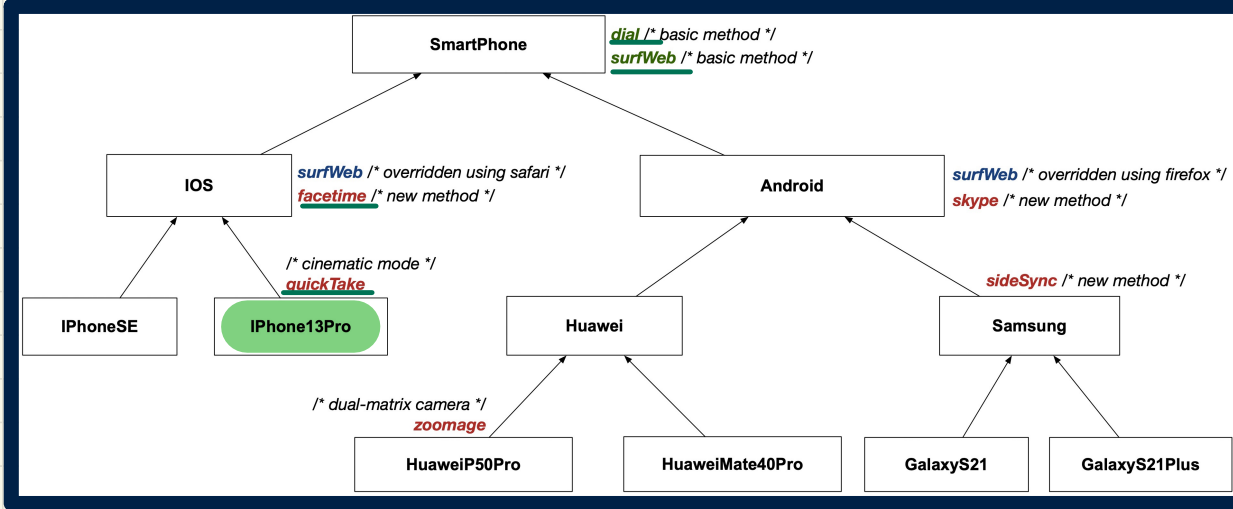


```
class SmartPhone {
    void dial() { ... }
}
class IOS extends SmartPhone {
    void facetime() { ... }
}
class iPhone13Pro extends IOS {
    void quickTake() { ... }
}
```

```
1 IOS ip = new iPhone13Pro();
2 ip.dial();
3 ip.facetime();
4 ip.quickTake();
```

ST: IOS.

Static Types, Casts, Polymorphism (3)

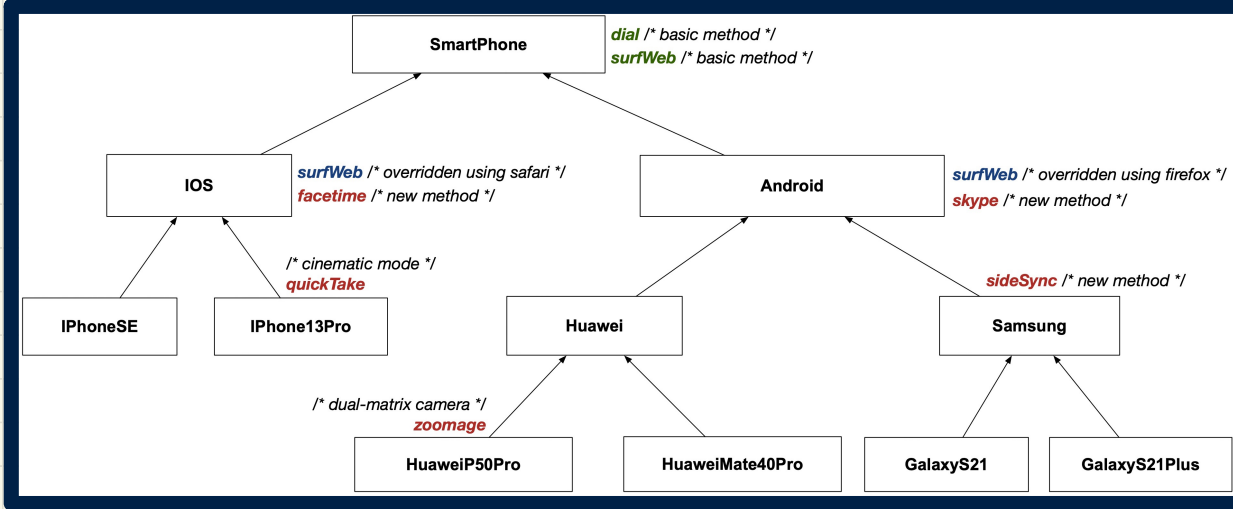


```
class SmartPhone {
    void dial() { ... }
}
class IOS extends SmartPhone {
    void facetime() { ... }
}
class iPhone13Pro extends IOS {
    void quickTake() { ... }
}
```

```
1 iPhone13Pro ip6sp = new iPhone13Pro();
2 ip6sp.dial(); ✓
3 ip6sp.facetime(); ✓
4 ip6sp.quickTake(); ✓
```

↓
ST: IP13Pro.

Static Types, Casts, Polymorphism (4)

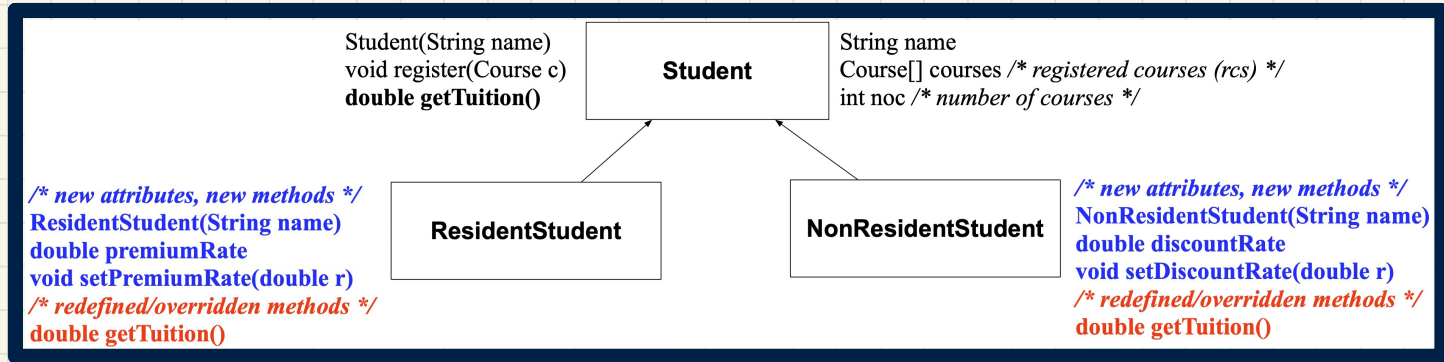


```
class SmartPhone {
    void dial() { ... }
}
class IOS extends SmartPhone {
    void facetime() { ... }
}
class iPhone13Pro extends IOS {
    void quickTake() { ... }
}
```

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

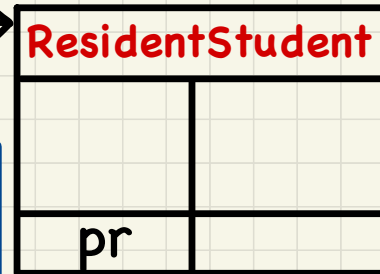
type cast exp.
of ST IP13Pro

Static Types, Casts, Polymorphism (5)



lme *CCE loop guaranteed C: instance of check*

```
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());
}
```



Call by Value

Caller

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

call by value: $\text{param} = \text{arg}^*$

Caller

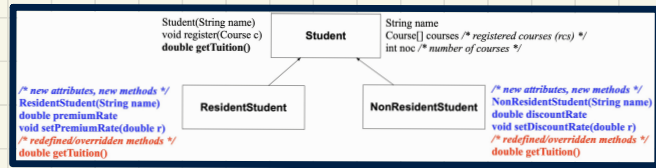
```
C obj = new C();  
T2 arg = ...;  
obj.m(arg);
```

* Rule of substitutions

- (1) T_2 a descendant of T_1
- (2) T_2 fulfills $\text{exp}(T_1)$

Polymorphic Parameters (1)

ST of each element in the array.



```

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

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

call by value: $rs = 0$

ST: S. ST: S. ST: S.

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

ST: RS

ST must fulfil RS (descendant of).

ST: Student

ST: RS

Yes "is descendant of Student."

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

ST: SMS

sms.addRS(0)

in order for method call to compile what should be about argument "0"?

Polymorphic Parameters (2)

for a method call to compile, the arguments ST must:

1. fulfill exp. of param's ST (CRS)
2. be a descend. of RS.

Ind.
RS = RS

```

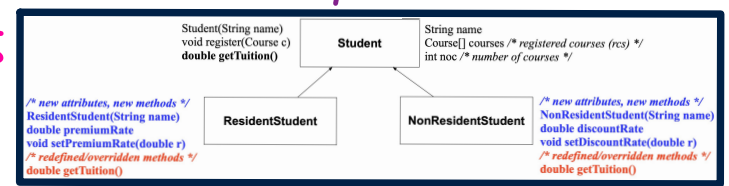
1 class StudentManagementSystem {
2     Student [] ss; /* ss[i] has static type Student */ int c;
3     void addRS(ResidentStudent rs) { ss[c] = rs; c++; }
4     void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5     void addStudent(Student s) { ss[c] = s; c++; } }
    
```

```

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);
- 6 sms.addStudent(s1);
- 7 sms.addStudent(s2);
- 8 sms.addStudent(s3);
- 9 sms.addStudent(rs);
- 10 sms.addStudent(nrs);

∴ args ST Student cannot fulfill ST of param RS
 * ∴ addStudent has param's ST top of
 RS = RS → ST: RS
 RS = nrs → ST: NRS
 ST: RS



inh. htr.,
my arg.
whose ST is
a descendant
until compile.

Lecture 21 - Nov 19

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

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

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);
```

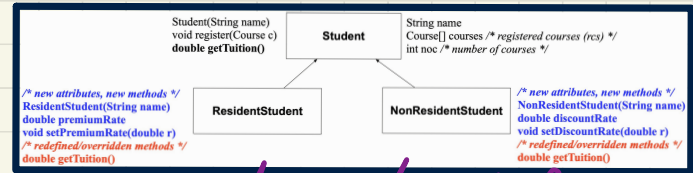
ST, RS = ST, Student

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

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

sms.addRS((ResidentStudent)nrs) compiles?

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



* downward cast: $S \rightarrow RS$
 * ST of cast exp: RS.
 ClassCastException?
 YES.
 ∵ DT of s (Student) cannot fulfil exp(RS)
 ClassCastException?
 YES.
 ∵ DT of s (NRS) not descendant of cast type(RS)
 ClassCastException? type(RS)
 No.
 CCE-free!

** not compiling
 ∵ neither upward nor downward cast.

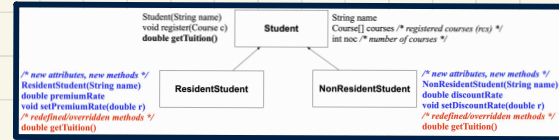
elements in array may have different DTs

A Polymorphic Collection of Students

```

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

$\frac{1}{0} \rightarrow \text{DT: RS}$
 $\frac{1}{1} \rightarrow \text{DT: NRS}$



```

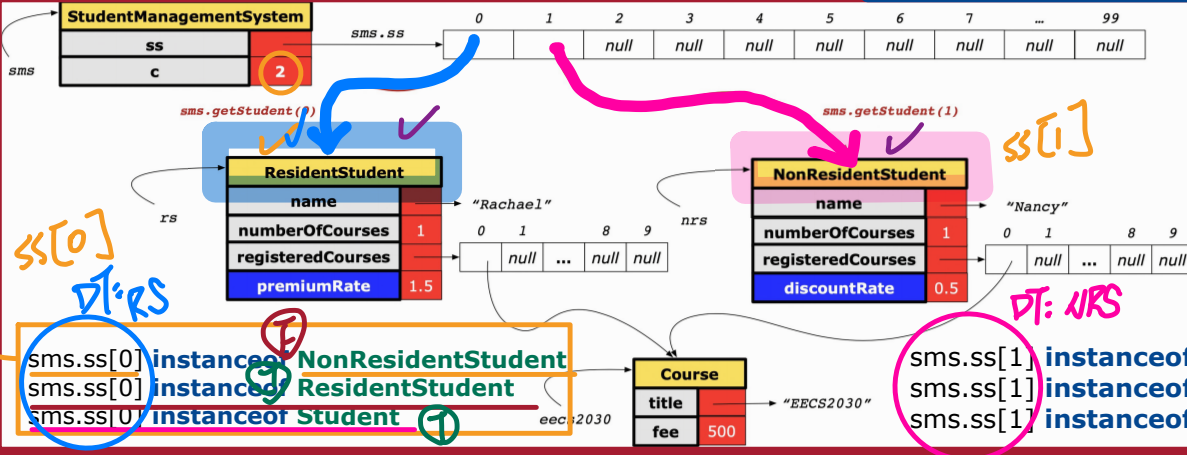
class StudentManagementSystem {
    Student[] students;
    int numofStudents;

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

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

arguments whose DTs are descendants of this can be passed

$0 \rightarrow \text{DT: RS}$
 $1 \rightarrow \text{DT: NRS}$



Polymorphic Return Types

```

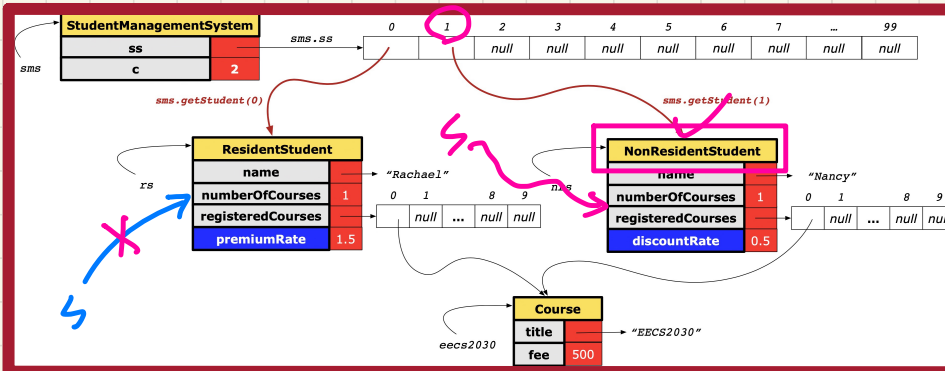
Course eecs2030 = new Course("EECS2030", 500);
ResidentStudent rs = new ResidentStudent("Rachael");
rs.setPremiumRate(1.5); rs.register(eecs2030);
NonResidentStudent nrs = new NonResidentStudent("Nancy");
nrs.setDiscountRate(0.5); nrs.register(eecs2030);
StudentManagementSystem sms = new StudentManagementSystem();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

Handwritten notes:
 dt: RS (for s)
 DT: NRS (for s)
 (NRS)
 ST: RT (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 index cond.");
        }
        else {
            s = ss[i];
        }
        return s;
    }
}
    
```

Handwritten notes:
 static type of return value
 DT of return value can be
 i < 0 || i >= c
 i → ss[i]: DT NRS
 0 → ss[0]: DT RS



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*/  
}
```

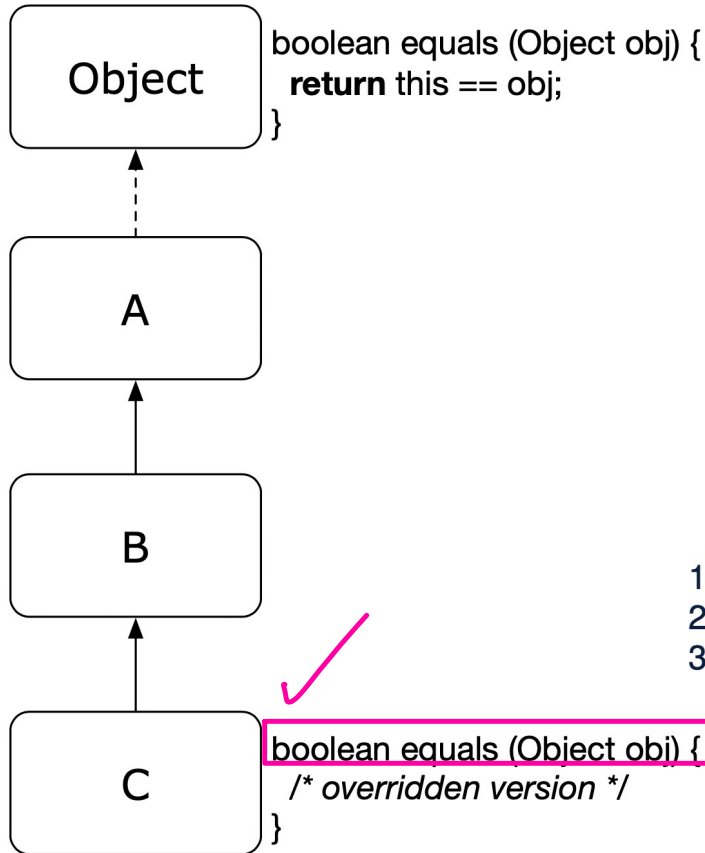
```
1 Object c1 = new C();  
2 Object c2 = new C();  
3 println(c1.equals(c2));
```

L3 calls which version of
equals?

Object

version in DT (C)
called.

Overridden Methods and Dynamic Binding (2)



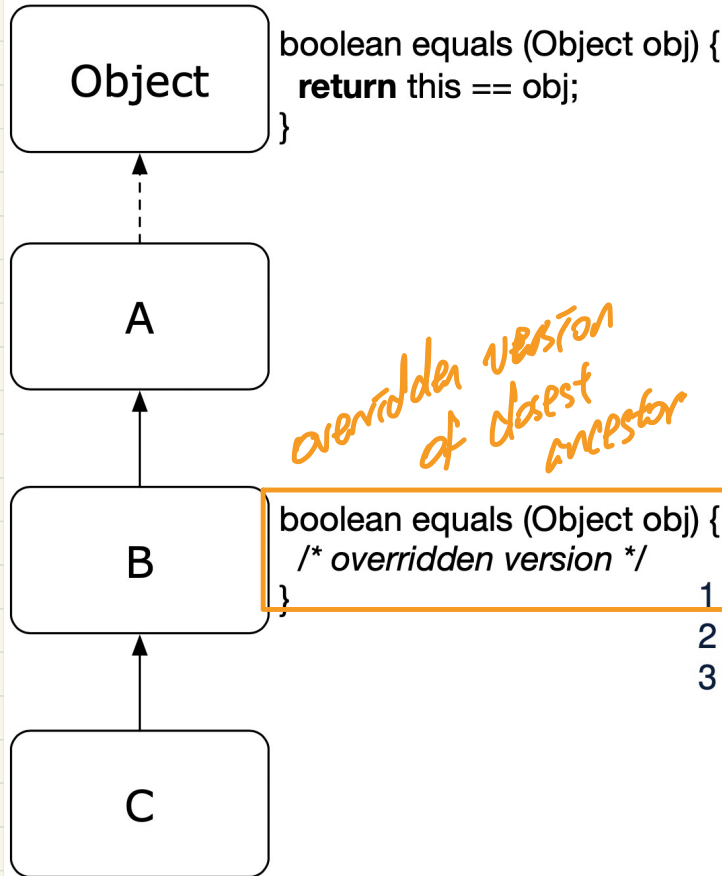
```
class A {  
    /*equals not overridden*/  
}  
class B extends A {  
    /*equals not overridden*/  
}  
class C extends B {  
    boolean equals (Object obj) {  
        /* overridden version */  
    }  
}
```

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

L3 calls which version of equals?



Overridden Methods and Dynamic Binding (3)



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

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

L3 calls which version of equals?

B

Compilation

Summary: Type Checking Rules

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

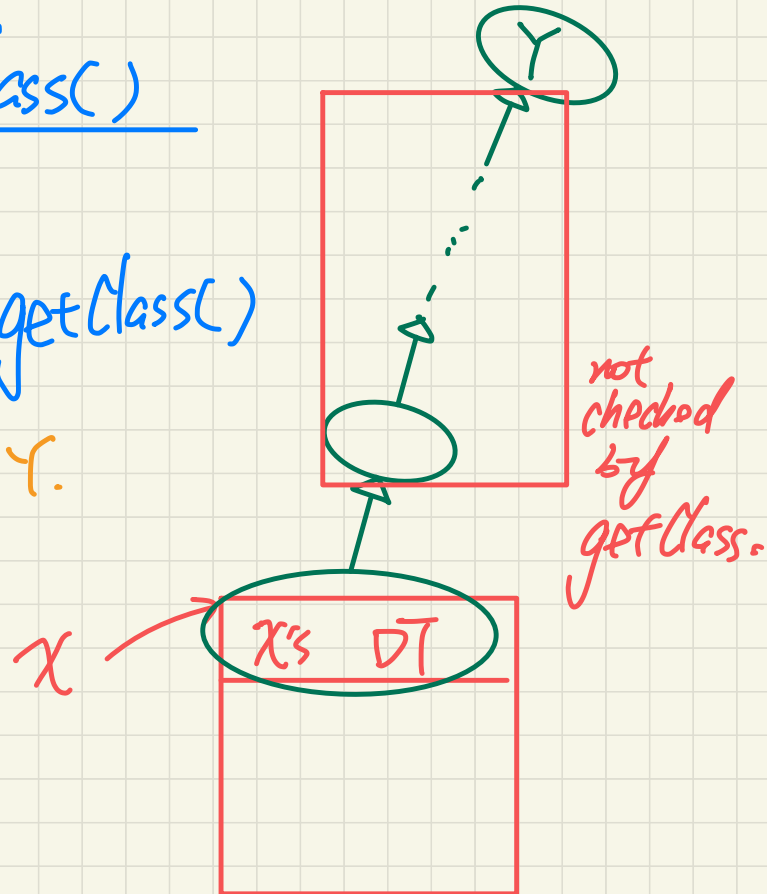
examples

given the compiler for ACE?

instance of vs. getClass()

$x.\text{getClass}() == y.\text{getClass}()$

↓
true means
x's DT is y.



Lecture 22 - Nov 24

Recursion

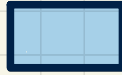
Recursion: Base vs. Recursive Cases
Tracing Recursion: Stack vs. Tree
Examples: fac, fib, isPalindrome

Announcements/Reminders

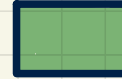
- Today's class: [notes template](#) posted
- **ProgTest2** results and feedback released
- **Lab5** released (deadline: Dec 2; **grace period** until Dec 9)
 - + Required background study: **abstract classes** & **interfaces**

Solving a Problem Recursively

Given a **small** problem:



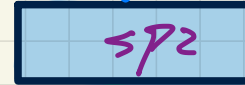
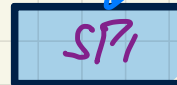
Solve it **directly**:



Given a **big** problem:



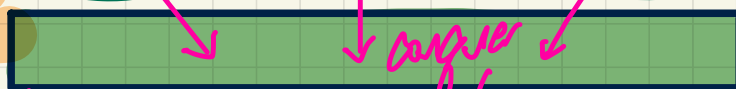
Divide it into **smaller** problems:



Assume solutions to **smaller** problems:



Combine solutions to **smaller** problems:



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

input correct outputs returned.

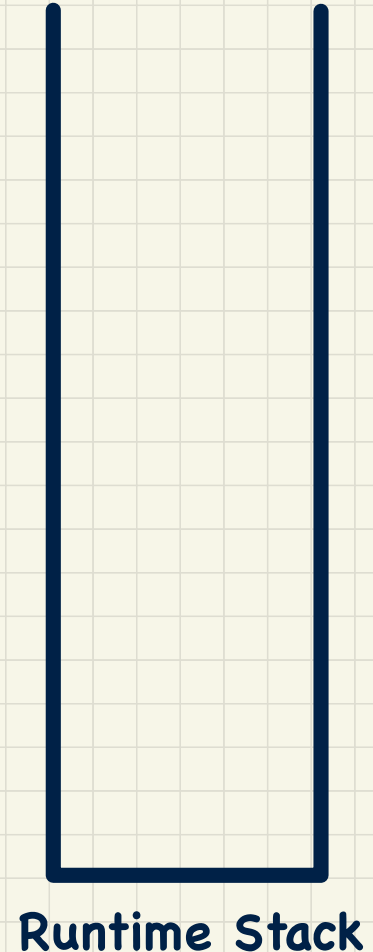
strictly smaller input: $j < i$.

recursive call

combine

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**.



Recursive Solution: factorial

$$4! = 4 \times 3 \times 2 \times 1 = 24.$$

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

Assump:
 $n \geq 0$.

not a recursive solution.
($\because$! is not re-used in the solution)

no recursive call to !.

Recursive Solution

$$n! = \begin{cases} 1 \\ 1 \times (n-1)! \end{cases}$$

strictly smaller input for recursive call.
base case

$$\text{if } n = 0$$

$$\text{if } n > 0$$

recursive call

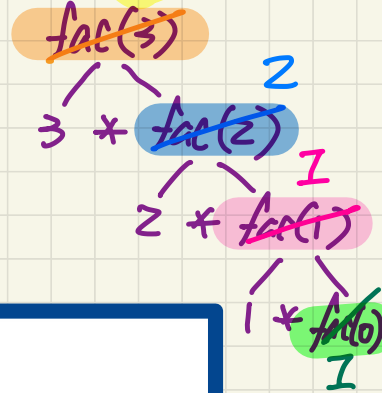
recursive case

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)



Runtime Stack

Tracing Recursions

(1) stack

(good in understanding how
chain of method calls work)



good →

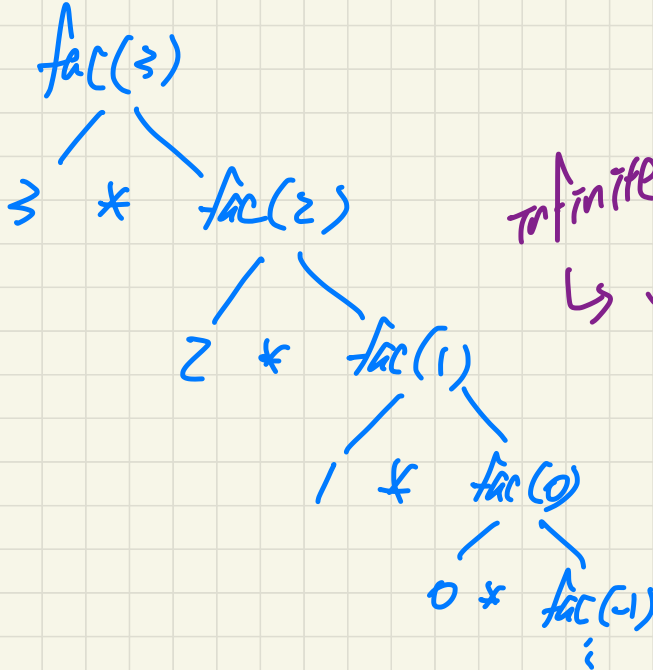
not as good building intuition about
the recursive solution being correct.

(2) "recursion tree"

Common Errors of Recursion (1)

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

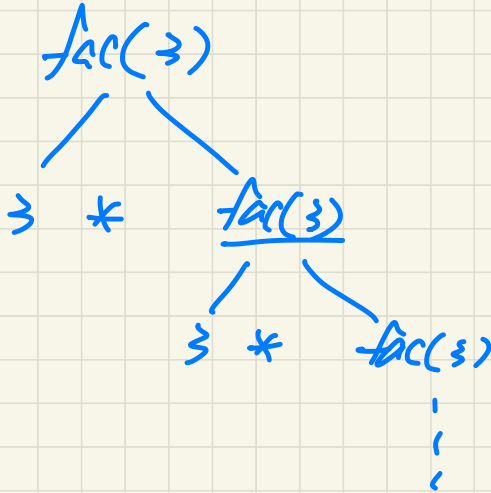
no base
case!



infinite recursive
↳ StackOverflowException

Common **Errors** of Recursion (2)

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



input to
recursive call
is not
strictly smaller!!

Infinite loop

vs.

Infinite recursion

↳
test case calling
the method with infinite loop
will hang indefinitely.
(until you force
quitting the IDE)

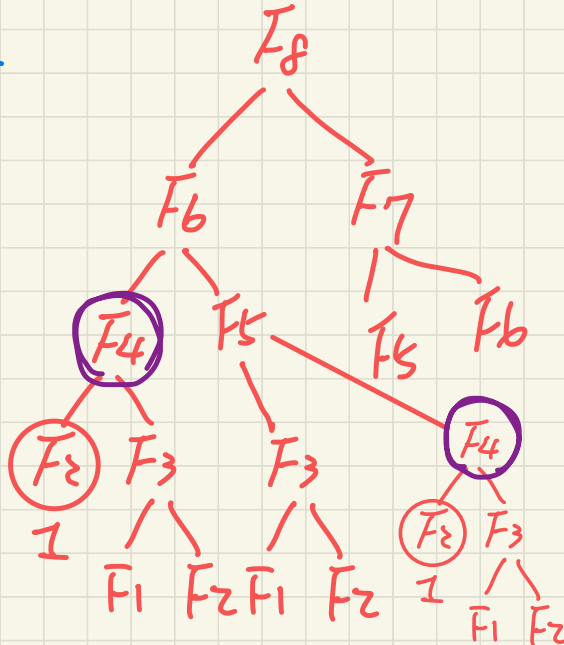
↳
stack overflow
exception.

Recursive Solution: Fibonacci Numbers

3 4 5 6 7 8 9 10 11 -

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

F_1 F_2



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

base cases

$$\begin{cases} \text{if } n=1 \\ \text{if } n=2 \end{cases}$$

$$\text{if } n \geq 3$$

recursive case

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)

Exercise: trace using stack

Runtime Stack

Use of String

$s \leadsto \text{"abcd"}$ $\underline{s} \leadsto$

0	1	2	3
a	b	c	d

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

no character.

$[0, 0] \rightarrow \text{stop: } (-1) - 0 + 1 = 0$

$[0, 3] \rightarrow \text{stop: } 2, 3$

$[0, 2] + [2, 4]$

$s.length()$ $\hookrightarrow s[i]$

$[i, j-1]$
 $[i, j)$

$s.substring(i, j)$
 $\hookrightarrow$
return a substring
by including characters
from indices: i to $j-1$

Size of interval:

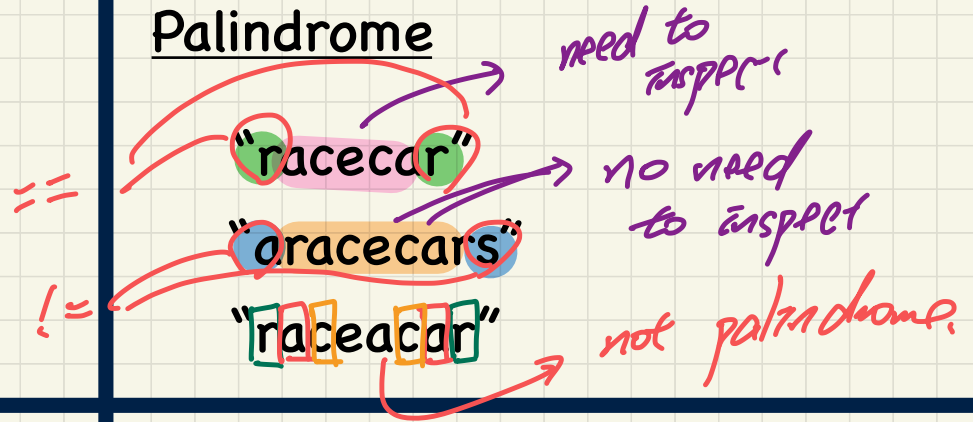
size: $[2, 4]$ 3
2, 3, 4

$[0, 6]$ 7
 $0 \sim 6$

given $a \leq b$
size: $[a, b]$ $b - a + 1$

Recursions on Strings

Palindrome



Reversal

"abcd"

Number of Occurrences

"abca"

'a'

'b'

Lecture 23 - Nov 26

Recursion

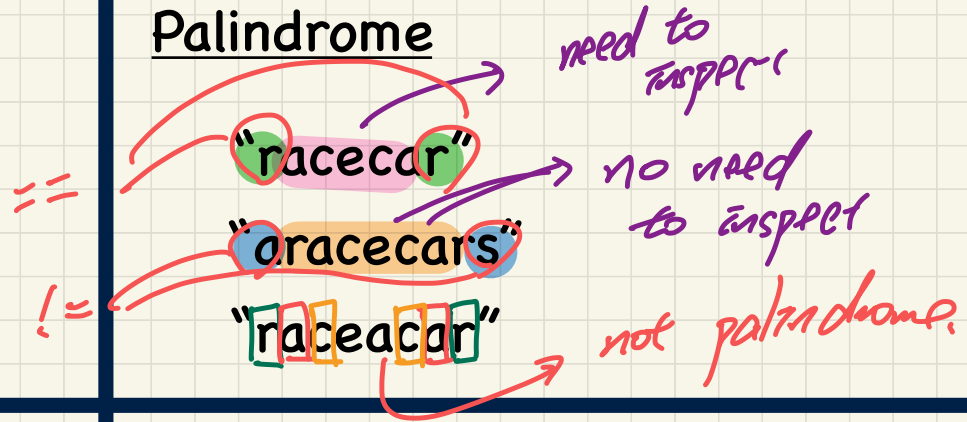
Examples: reverseOf, occurrencesOf
Recursion on Arrays: Creating Sub-Arrays
Recursion on Arrays: Call by Value

Announcements/Reminders

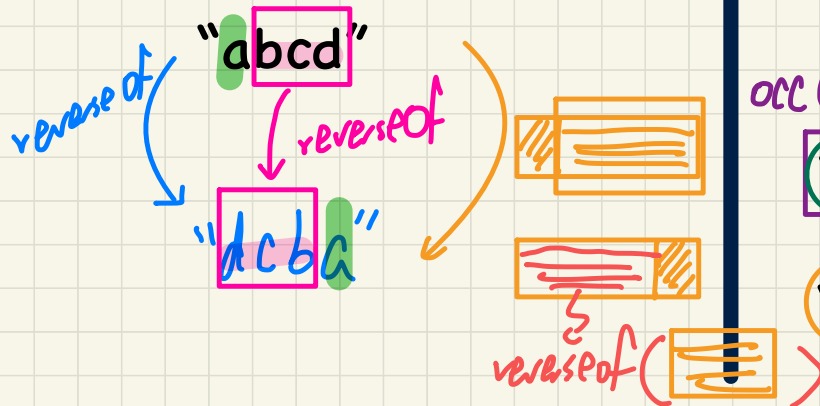
- Today's class: [notes template](#) posted
- **Lab5** released (deadline: Dec 2; **grace period** until Dec 9)
 - + Required background study: **abstract classes** & **interfaces**

Recursions on Strings

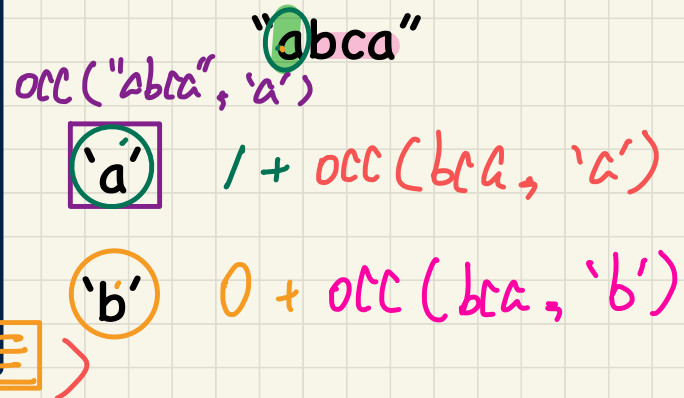
Palindrome



Reversal



Number of Occurrences

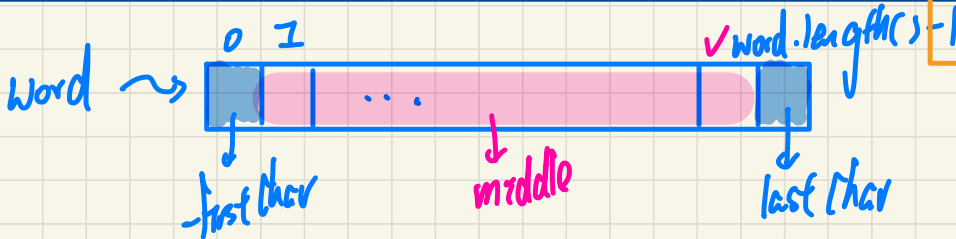


Problem: Palindrome

```
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  
            && isPalindrome (middle);  
        /* See the API of java.lang.String.substring. */  
    }  
}
```

1. LHS is false
↳ skip RHS

2. LHS is true
↳ eval RHS



isP(a**bcb**a)

'a' == 'a'
T

&& isP(bcb)

'b' == 'b'
T

&& isP(c)

True.

T

T

↳ RC. made only if first & last char match.

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;  
    }  
}
```

$r(\boxed{a}\boxed{bc})$

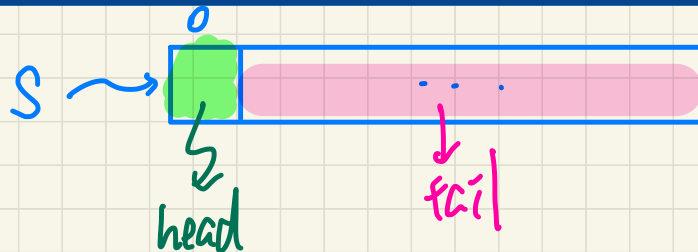
$r(\boxed{b}\boxed{c}) + a$

$r(\boxed{c}) + b$

cb

(cba)

RC.



Exercise

a b c d

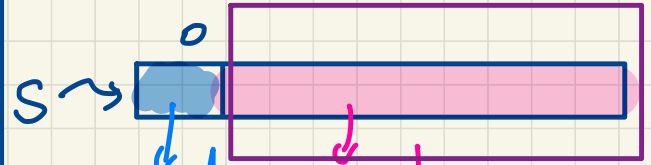
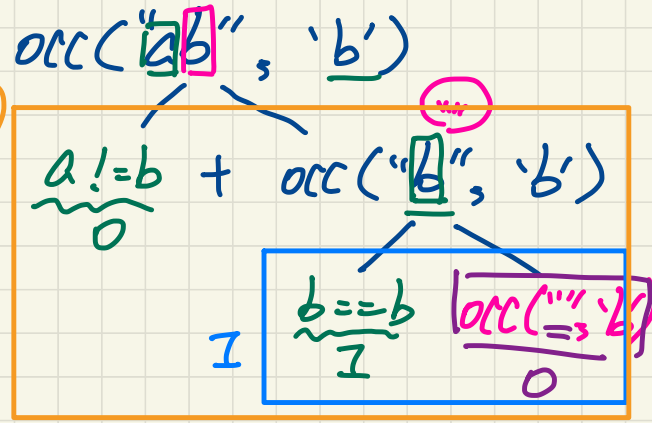
d c b a

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);
        }
    }
}
```

"b"
head
tail: substring(1, 1)
→ ""

①



$\text{head} == c \rightarrow 1$

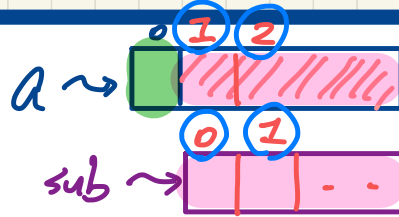
$\text{occ}(\text{tail})$

$\text{head} \neq c \rightarrow 0$

c = 'a'

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) } }  
m(sub)
```

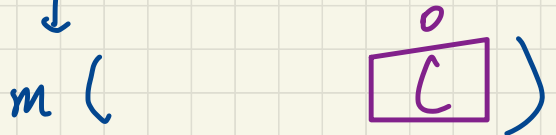
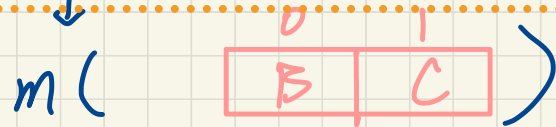
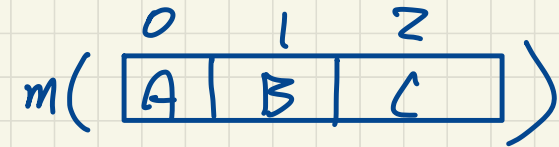


Copy Every thing except the head.

Say a1 = {}, consider m(a1)

↳ base case # 1

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



time & space
wasted in copying
same contents
over & over

Recursion on an Array: Passing Same Array Reference

"sub-arrays"

range of indices

to inspect in array



```
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

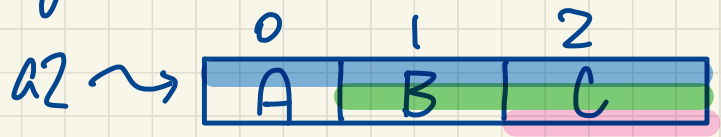
* call by value: passing the ref. of 'a' around.

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

$a1.length == 0$
0 2
1st index last index

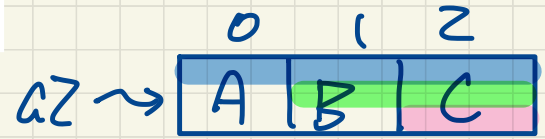
Say $a2 = \{A, B, C\}$, consider $m(a2, 0, a2.length - 1)$

$a2.length == 3$
2



$from > to$
empty range of indices to look at
 $\rightarrow$ empty sub-array

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



$m(a2, \underline{0}, \underline{2})$

↳ input array of length 3

$m(a2, \underline{1}, \underline{2})$

↳ sub-array of length 2

$m(a2, \underline{2}, \underline{2})$

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

As recursive calls are made, narrow down the intervals.

Lecture 24 - Dec 1

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

Problem: Are All Numbers Positive?

universal property

Say $a = \{\}$

↓
true

Say $a = \{4\}$

↓
true.

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

↓
true.

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

↓
false.

$\forall x \cdot P(x) \equiv \text{true.}$) there's no witness to show property violation

$\exists x \cdot P(x) \equiv \text{false}$) there's no witness to show property satisfaction.
e.g. $x > 0$.

allPos ({ ^{from} 4, ¹ 7, ² 3, ³ 9 }, ^{from} 0, ^{to} 3)

$a[0] > 0$ && allPos (a, ^{from+1} to)
4 > 0
↓
true.

Problem: Are All Numbers Positive?

→ public

version for extend use to call.

```
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);  
    }  
}
```

↪ attempt to scan the entire array.

✓ → private. recursive

✓
a



$a[from] > 0 \ \&\& \ \text{allPos}(a, from+1, to)$

a at elements:
from+1 ... to
interval.

Tracing Recursion: **allPositive**

Say a = {}

allPositive(a)

allPH(a,0,-1)

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

Tracing Recursion: allPositive

Say a = {4}

allPositive(a)

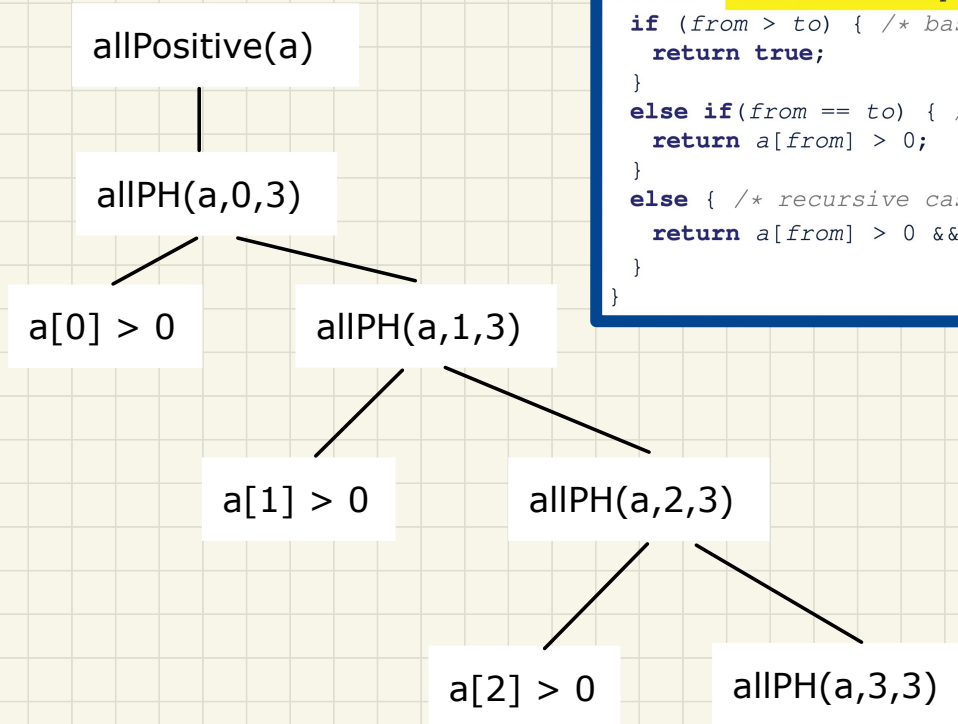
allPH(a,0,0)

a[0] > 0

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

Tracing Recursion: **allPositive**

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



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


Tracing Recursion: allPositive

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

allPositive(a)

allPH(a, 0, 3)

$a[0] > 0$

allPH(a, 1, 3)

$a[1] > 0$

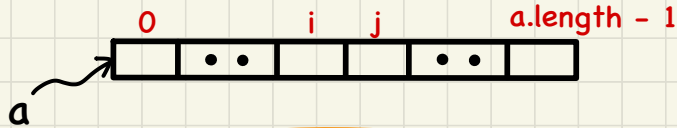
allPH(a, 2, 3)

$a[2] > 0$

allPH(a, 3, 3)

```
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);  
    }  
}
```

Sorting Orders of Arrays



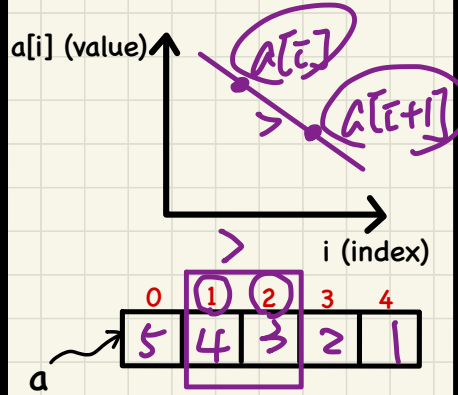
non-descending

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

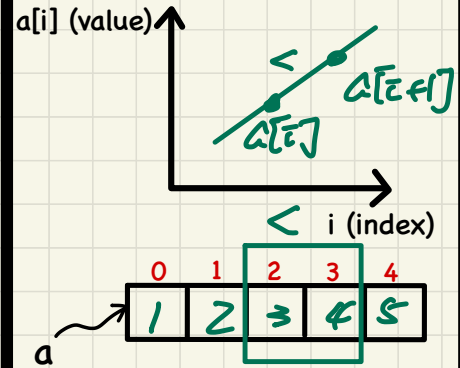
$\equiv \neg(a[i] > a[i+1])$

$\equiv a[i] \leq a[i+1]$
ascending
no change.

decreasing/descending

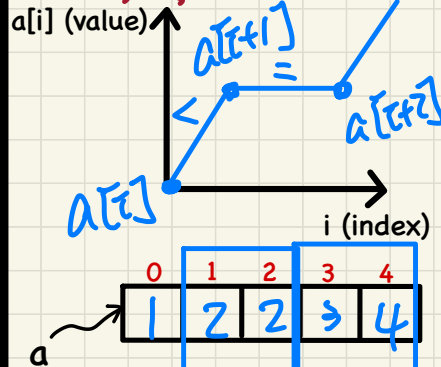


increasing/ascending

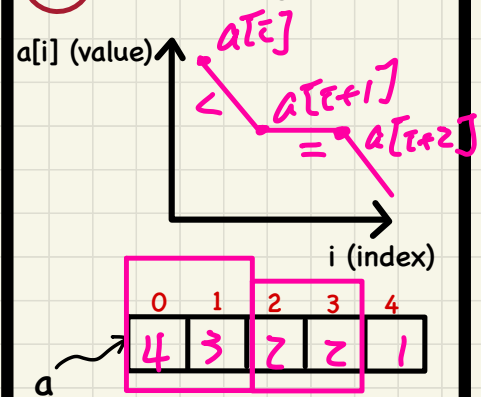


non-descending

not, $\neg$, !



non-ascending



Problem: Are Numbers Sorted?

Say $a = \{\}$

true $\because$ no violation can be found.

Say $a = \{4\}$

true.

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

$0 \leq 1 \leq 3$

$3 \leq 6$

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

should be looked down to:
 $a[0] \leq a[1]$ &&
 $a[1] \leq a[2]$ &&
 $a[2] \leq a[3]$

isSorted(3, 6, 6, 7, 0, 3)

$a[0] \leq a[1]$
 $3 \leq 6$
true.

&& isSorted($a, 1, 3$)

Problem: Are Numbers Sorted?

```
boolean isSorted(int[] a) {  
    return isSortedHelper(a, 0, a.length - 1);  
}  
  
boolean isSortedHelper(int[] a, int from, int to) {  
    if (from > to) { /* base case 1: empty range */  
        return true;  
    }  
    else if (from == to) { /* base case 2: range of one element */  
        return true;  
    }  
    else {  
        return a[from] <= a[from + 1] !.  
            && isSortedHelper(a, from + 1, to);  
    }  
}
```

Tracing Recursion: **isSorted**

Say $a = \{\}$

isSorted(a)

isSH(a,0,-1)

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

Tracing Recursion: **isSorted**

Say $a = \{4\}$

isSorted(a)

isSH(a,0,0)

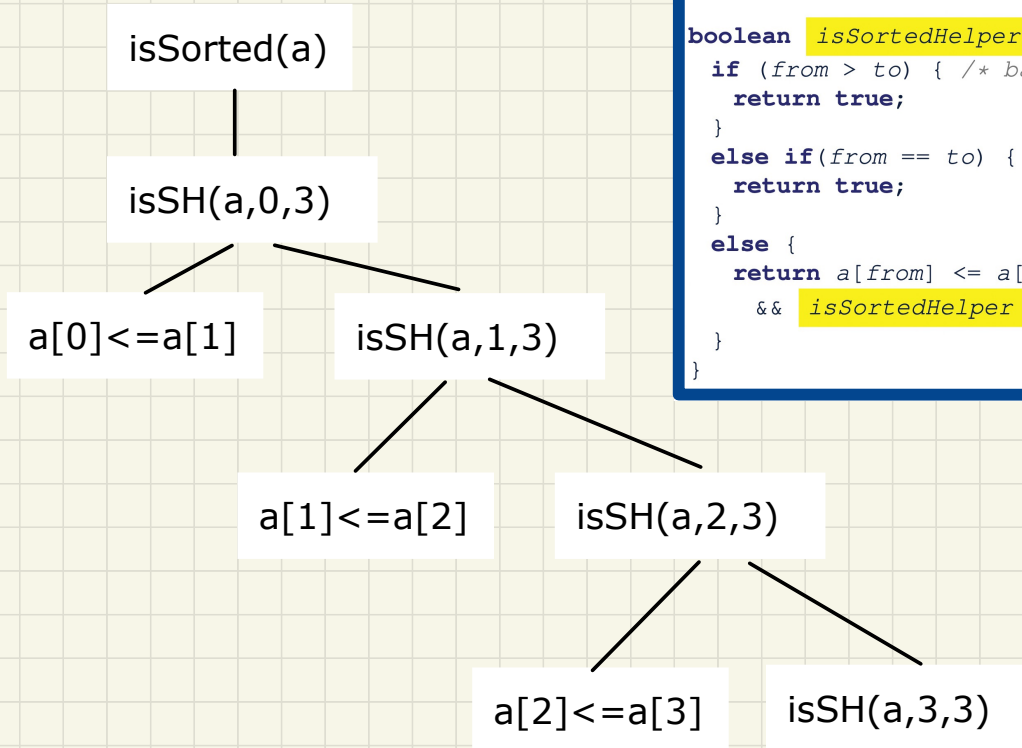
return **true**

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

Tracing Recursion: **isSorted**

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

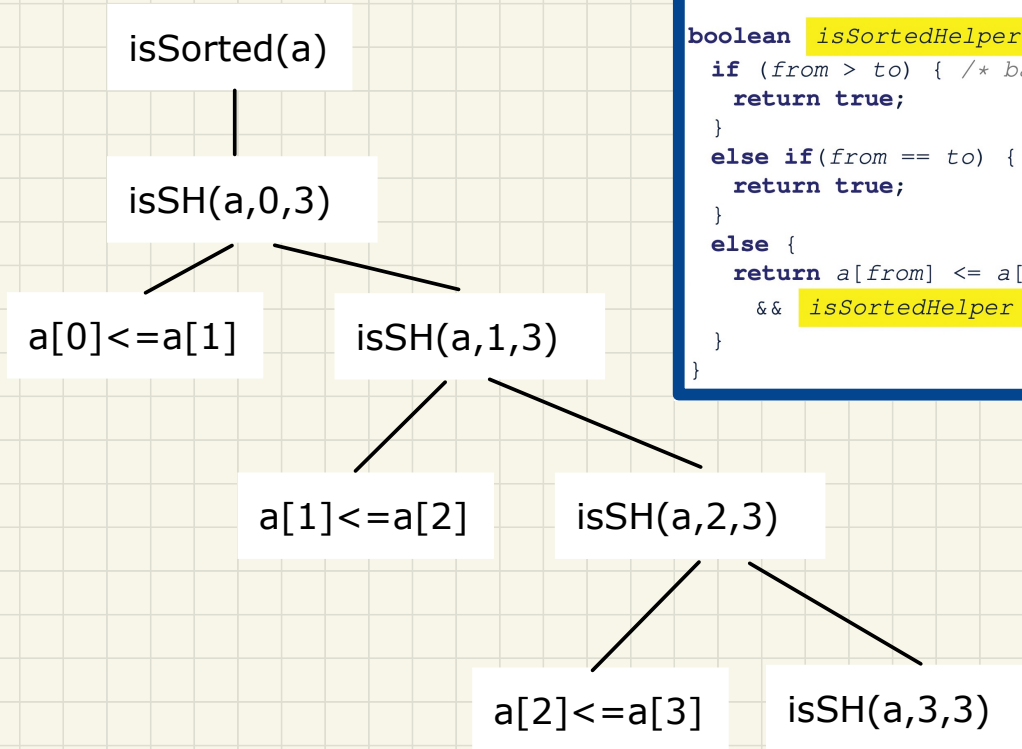
```
boolean isSorted(int[] a) {  
    return isSortedHelper(a, 0, a.length - 1);  
}  
  
boolean isSortedHelper(int[] a, int from, int to) {  
    if (from > to) { /* base case 1: empty range */  
        return true;  
    }  
    else if (from == to) { /* base case 2: range of one element */  
  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



Tracing Recursion: **isSorted**

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

```
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
- Where: TM Tait
- Format: Some Multiple Choice & Mostly Written
- Coverage: **Everything** (lecture materials & labs)
 - + slides, iPad notes, code examples
 - + resources on recursion
- Restrictions:
 - + No data sheet
 - + No sketch paper (Exam booklet includes it)
- What you should bring:
 - + Valid, Physical Photo ID (strict)
 - + Water/Snack

(question booklet
answer booklet.
no sections!

- definitions
- short answers
 - ↳ explanations
 - ↳ tracing
 - ↳ justification
- code

I hope you enjoyed learning with me 



All the best to you ! 