EECS2030 Fall 2021
Advanced Object-Oriented Programming

Lecture Notes

Instructor: Jackie Wang

Lecture 2a

Part A

Exceptions -
Caller vs. Callee in a Method Invocation

Caller vs. Callee

Caller → party calling another method (callee)

Callee → party being called by another method (caller)

- **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 */
  }
}
```

C1.m1 → context of method call (caller)

o → context object

callee: class: (type of c.o.) C2; method: m2

Context of a method call/invocation
1. class
2. method

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

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

o.m1() → callee: C1.m1

Can C2.m2 be a caller as well?
YES: make some method call in C2.m2

Lecture 2a

Part B

Exceptions -
Visualizing a Method Call Chain as a Stack

Visualizing a Call Chain using a Stack

active

m1( ) {
① ______
② m2( );
⑬ ______
⑭ ______
⑮ ______
}

m2( ) {
③ ______
④ ______
⑤ m3( );
⑪ ______
⑫ ______
}

m3( ) {
⑥ ______
⑦ ______
⑧ ______
⑨ ______
⑩ ______
}

return.

return.

suspended until the execution of m2 terminates & returns

suspended until the execution of m3 terminates & returns.

reverse chronological order of method calls.

call stack

m3
m2
m1

Lecture 2a

Part C

Exceptions -
Error Handling via Console Messages

Error Handling via Console Messages: Circles

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

Caller?

Callee?

call stack

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

Console

Invalid radius.
Area 0.0

Circle.setR
CC.main

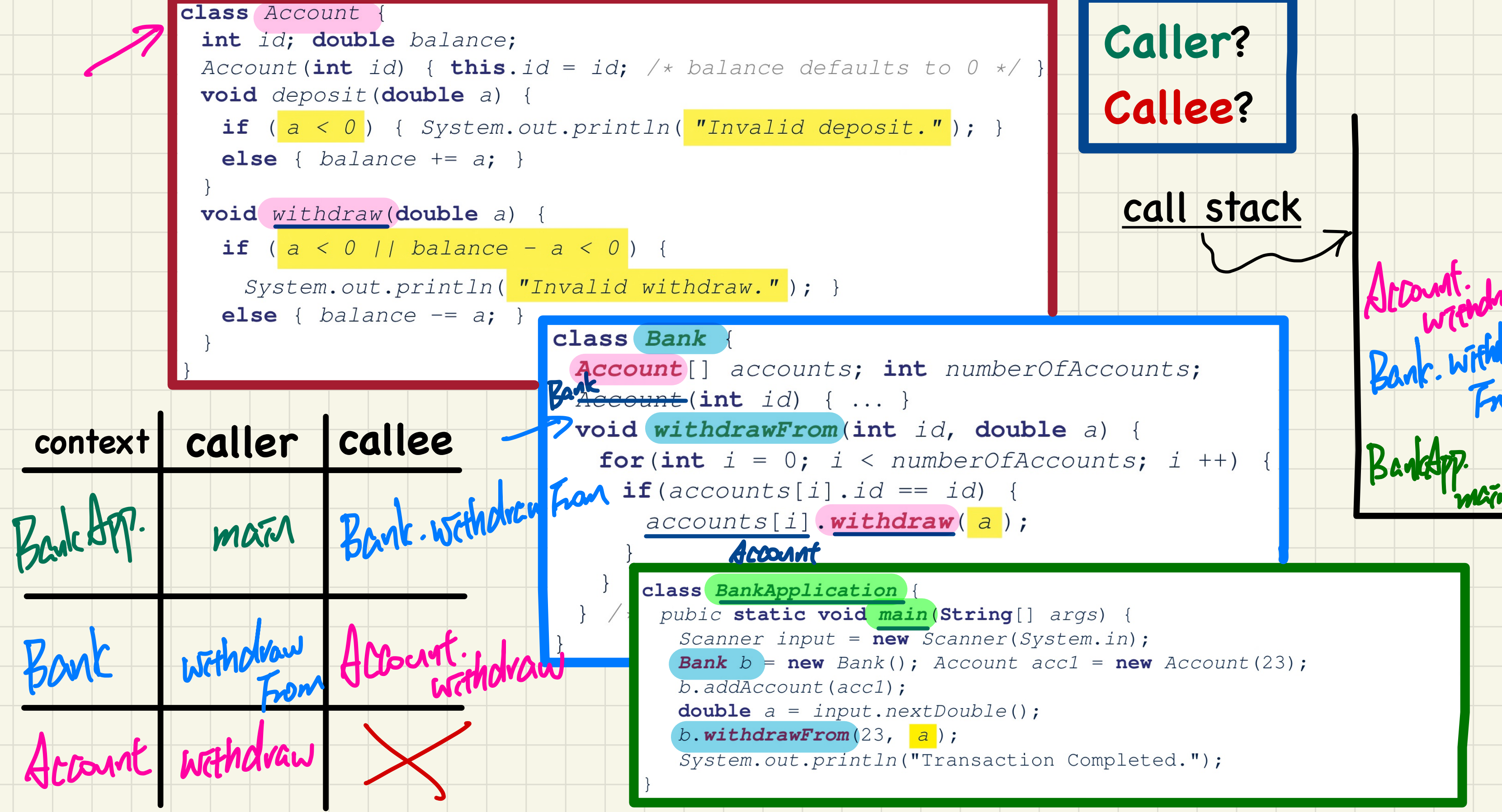
caller

callee

printing an err msg to console does not cause caller to stop.

output to console.

Error Handling via Console Messages: Banks

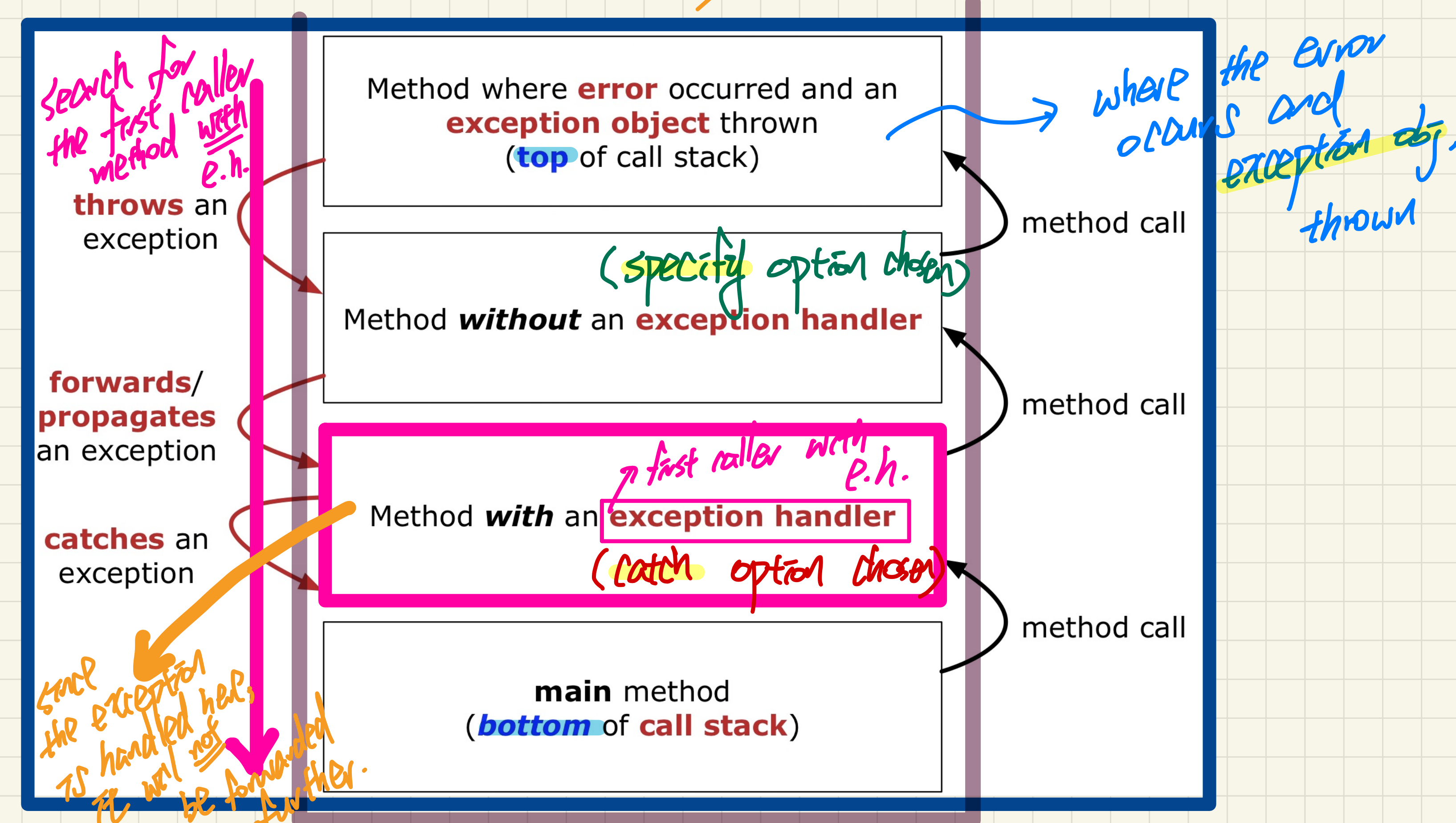


Lecture 2a

Part D

Exceptions -
When an Exception is Thrown,
The Catch-or-Specify Requirement

What to Do When an Exception is Thrown: Call Stack



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(NegativeRaidusException e) {
    ...
  }
}
```

callee throws an exception upon an invalid input value

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);
    ...
  }
}
```

header of method

callee may throw an exception upon an invalid amount

Recap of Exceptions

- Catch-or-Specify Requirement

Normal Flow of Execution

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

no exception was thrown

When the exception does not occur

Abnormal Flow of Execution

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

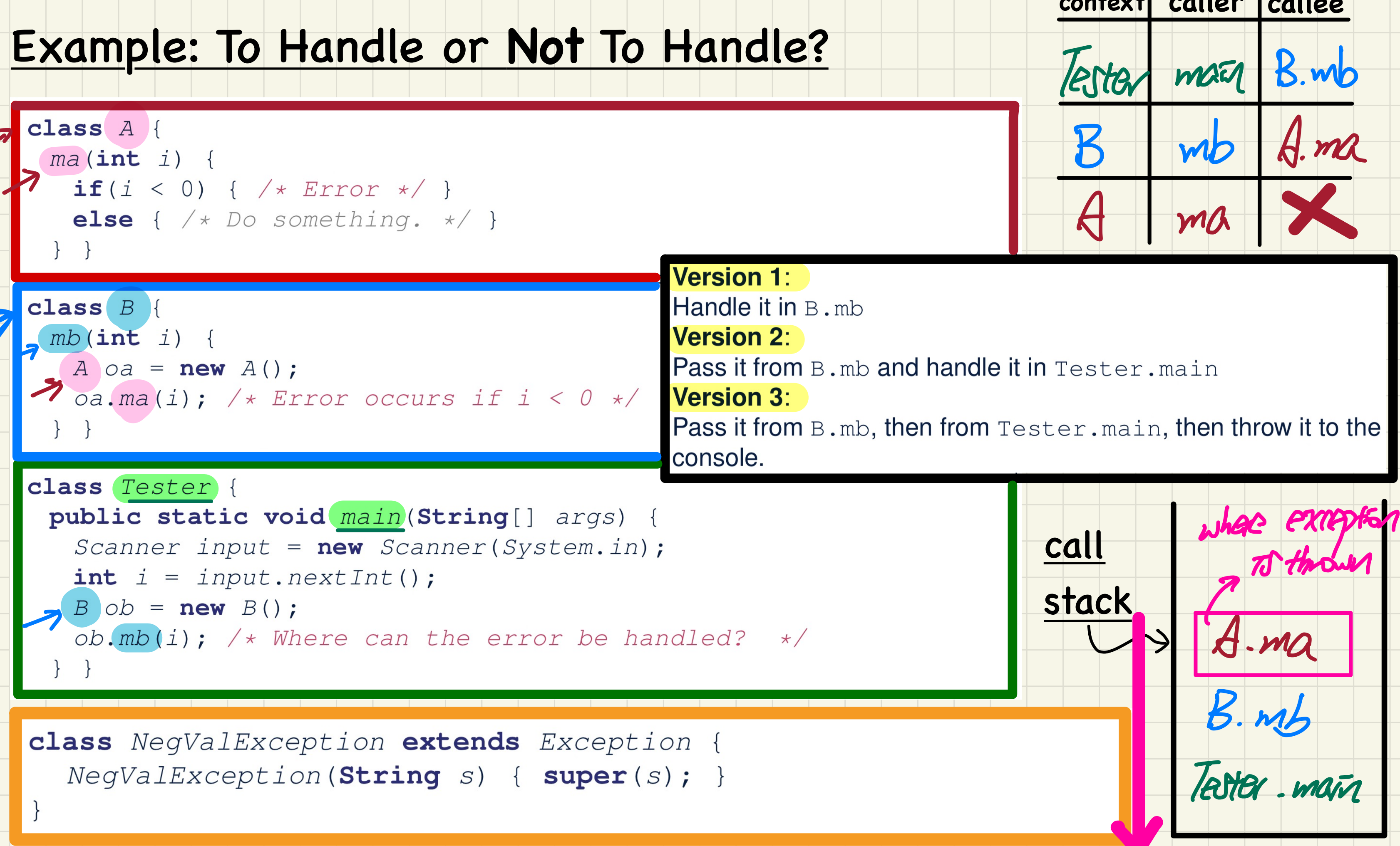
exception was thrown

When the exception occurs

Lecture 2a

Part E

Exceptions -
Example: To Handle or Not to Handle?


Example: To Handle or Not To Handle?
context | caller | callee
Tester | main | B.mb
B | mb | A.ma
A | ma
class A {
ma(int i) {
if(i < 0) { /* Error */ }
else { /* Do something. */ }
} }
class B {
mb(int i) {
A oa = new A();
oa.ma(i); /* Error occurs if i < 0 */
} }
class Tester {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
int i = input.nextInt();
B ob = new B();
ob.mb(i); /* Where can the error be handled? */
} }
class NegValException extends Exception {
NegValException(String s) { super(s); }
}
Version 1:
Handle it in B.mb
Version 2:
Pass it from B.mb and handle it in Tester.main
Version 3:
Pass it from B.mb, then from Tester.main, then throw it to the console.
call stack
where exception is thrown
A.ma
B.mb
Tester.main

Version 1:
Handle the Exception in B.mb

Method **A.ma** causes an **error** and an **NegValException object** is thrown

throws an exception

Method **B.mb** *chooses to handle the error right away using a* ***try-catch*** *block.*

catches an exception

Method **Tester.main** method need not worry about this error.

method call

method call

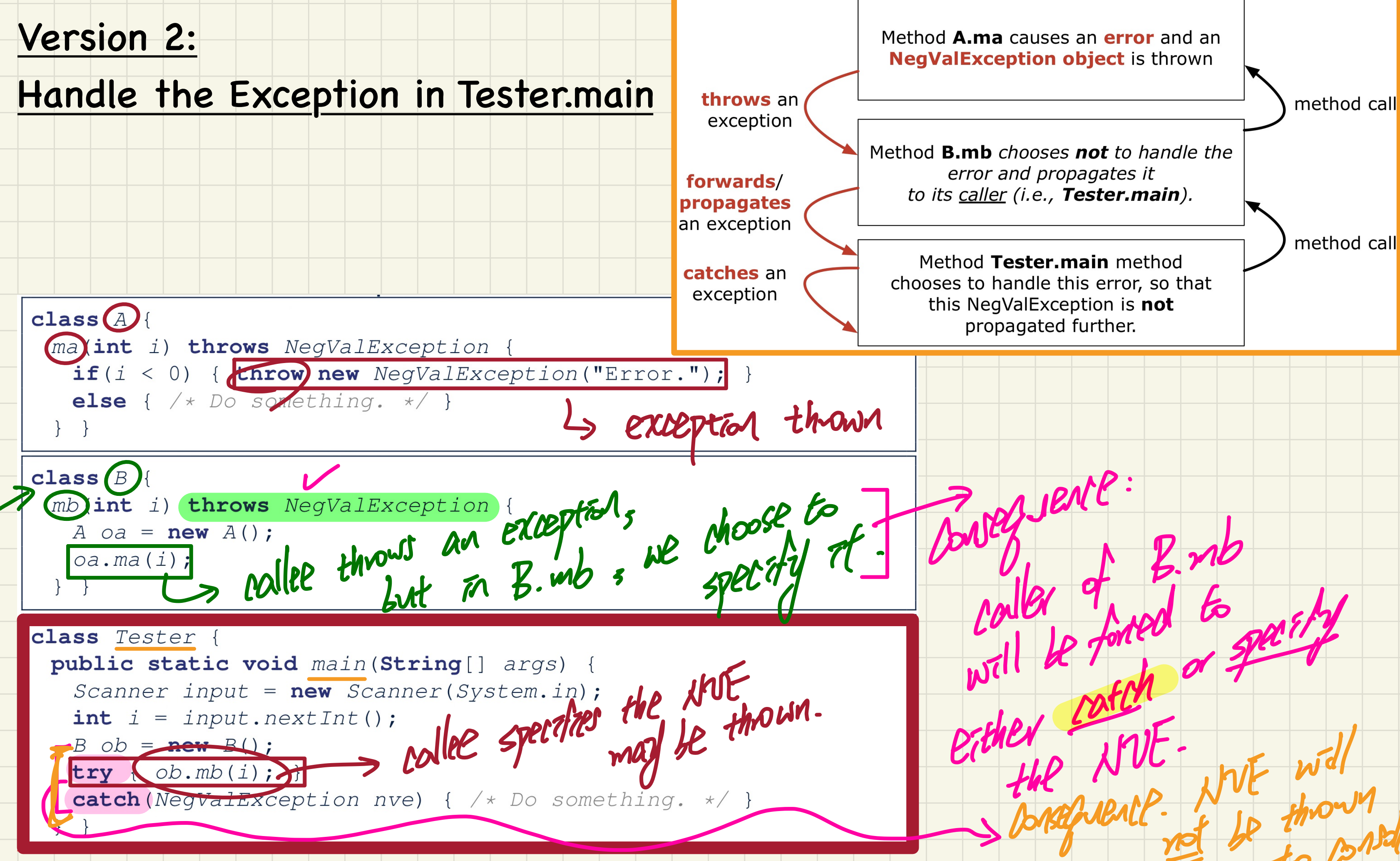
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

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

callee throws NVE

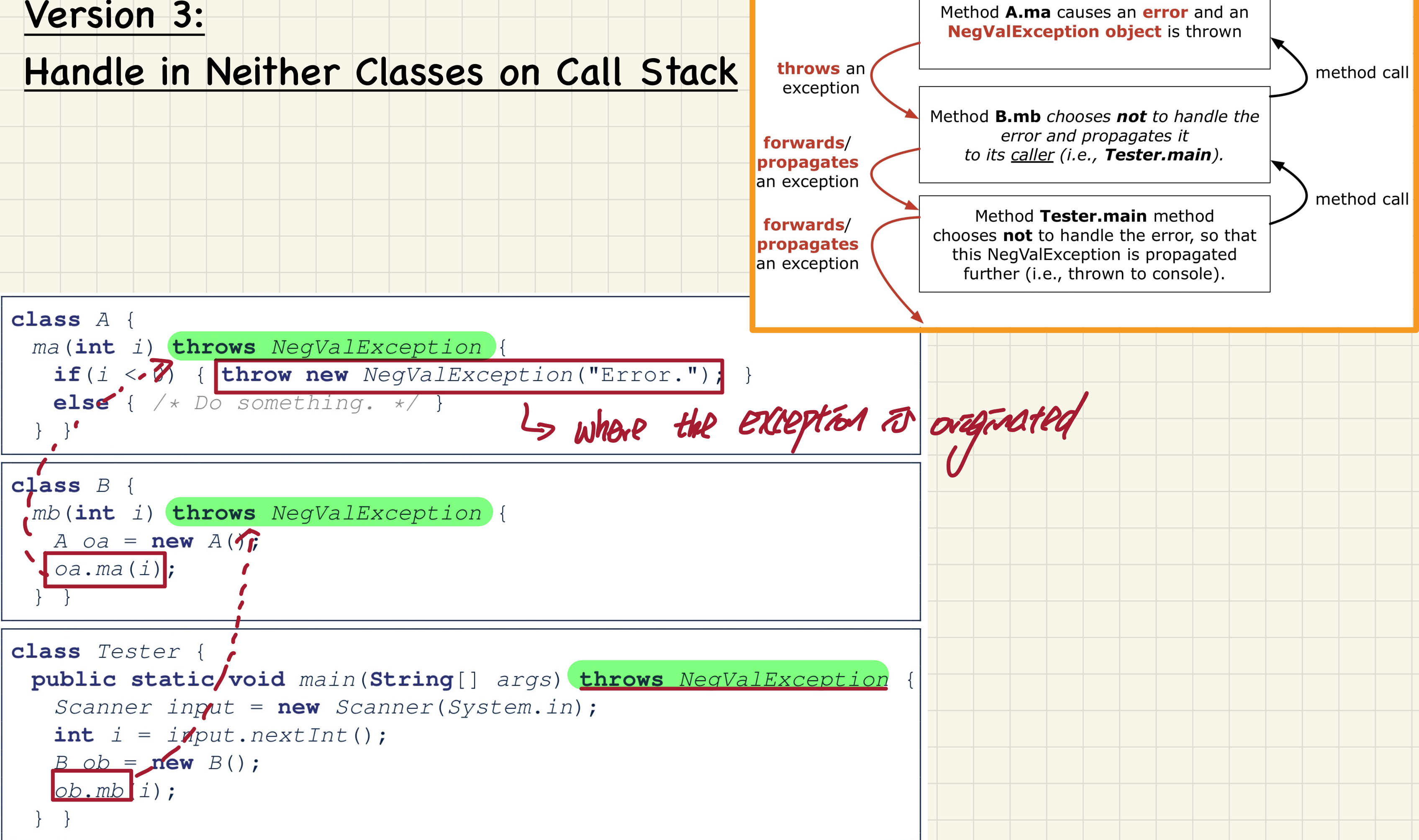
```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Error, if any, would have been handled in B.mb. */
  } }
```

→ B.mb already handles the NVE, so Tester.main does not need to catch or specify it.


Version 2:
Handle the Exception in Tester.main
Method A.ma causes an error and an NegValException object is thrown
throws an exception
method call
Method B.mb chooses not to handle the error and propagates it to its caller (i.e., Tester.main).
forwards/ propagates an exception
method call
Method Tester.main method chooses to handle this error, so that this NegValException is not propagated further.
catches an exception
class A {
ma(int i) throws NegValException {
if(i < 0) { throw new NegValException("Error."); }
else { /* Do something. */ }
} }
exception thrown
class B {
mb(int i) throws NegValException {
A oa = new A();
oa.ma(i);
} }
callee throws an exception, but in B.mb, we choose to specify it.
class Tester {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
int i = input.nextInt();
B ob = new B();
try { ob.mb(i); }
catch(NegValException nve) { /* Do something. */ }
} }
callee specifies the NVE may be thrown.
Consequence: caller of B.mb will be forced to either catch or specify the NVE.
Consequence: NVE will not be thrown to console.

Version 3:

Handle in Neither Classes on Call Stack




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

↳ where the exception is originated

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

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

Lecture 2a

Part F

Exceptions -
Error Handling via Exceptions

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; }
}
```

```
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);
    }
  } }
```

specify

where the IRE is originated

once the exception is handled here, it will not be propagated further.

Error Handling via Exceptions: Circles (Version 2)

```
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; }
}
```

Test Case:

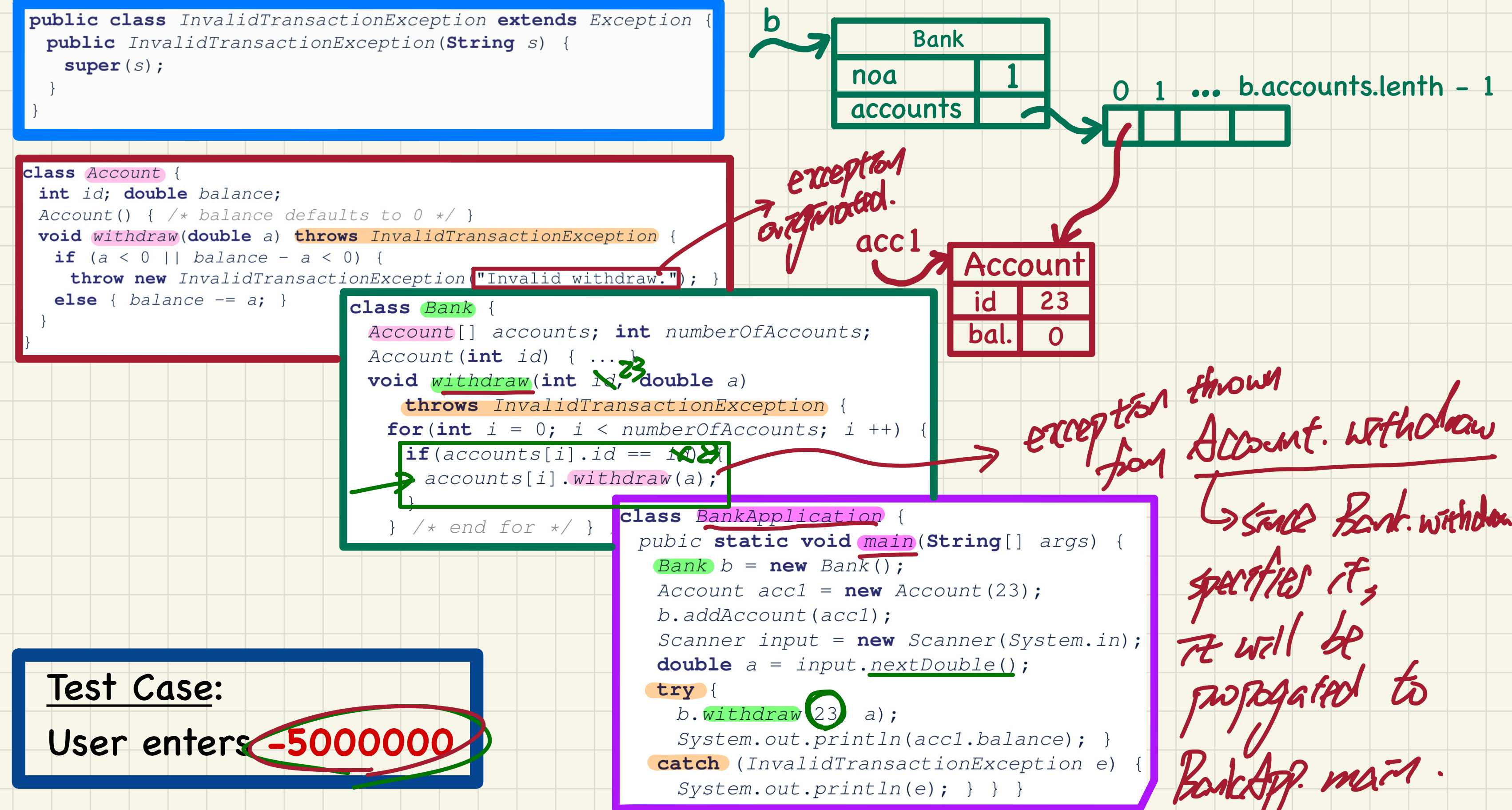
User enters -5

Then user enters 10

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

Enter a radius:
-5
Try again!
Enter a radius: 10
Circle with radius 10 has area 314

Error Handling via Exceptions: Banks



Lecture 2a

Part G

Exceptions - More Examples

More Example: Multiple Catch Blocks

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

Test Case 1:

a: -5000000

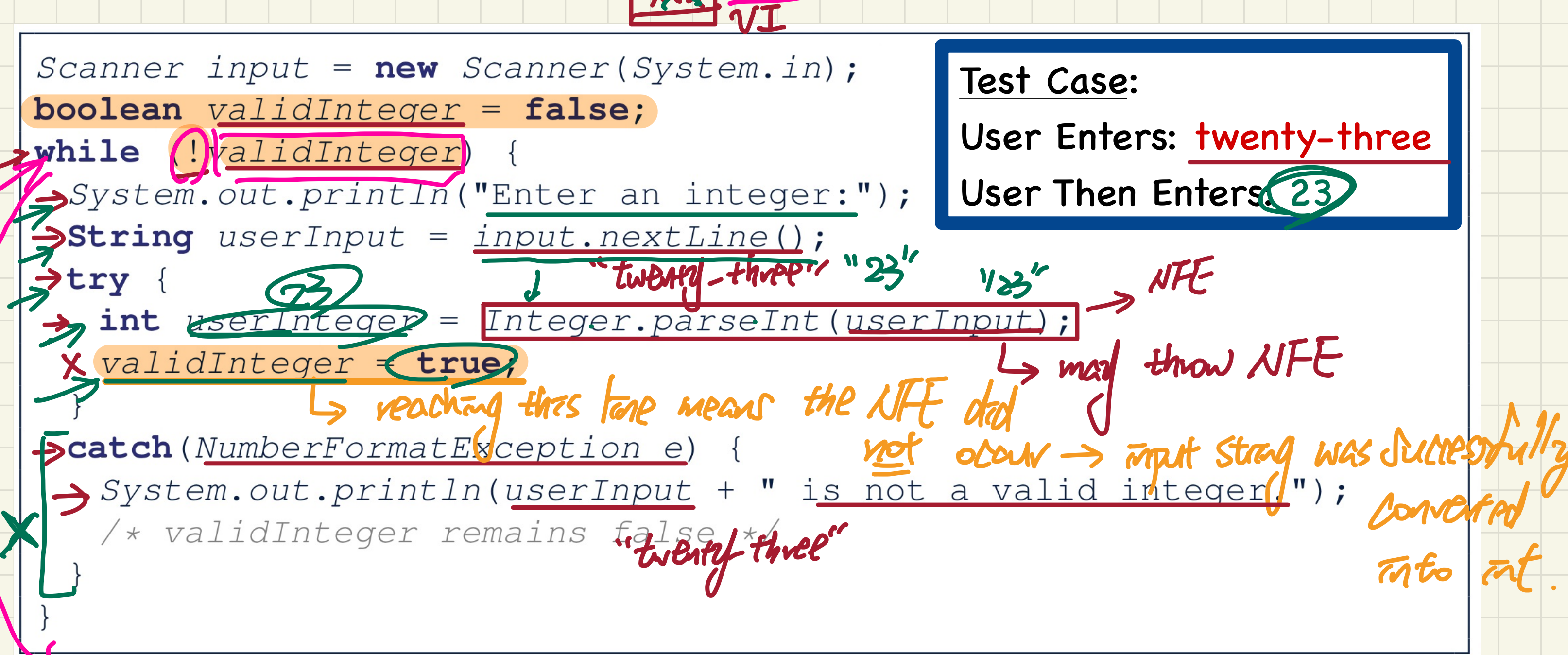
r: 23

Test Case 2:

a: 100

r: -5

More Example: Parsing Strings as Integers



Lecture 2b

Part A

Test-Driven Development (TDD) - Counter Problem, Review on Exceptions

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);
    }
  }
}
```

specify opt.

handle an error

where the exception is originated

Example 2: Method that calls another which throws the exception

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

specify opt.

may throw a VTSE → subject to catch-or-specify req.

C.O.

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) { ... }
  }
}
```

c.o. ↳ may throw VTSE → must either catch or specify.

Exercise: Put VTLE instead

↳ match one of the exceptions that might come from the catch block

↳ VTSE will not be propagated further.

A Class for Bounded Counters

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

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

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

no need to access them using a context object
↳ Counter.MAX_VALUE

MIN → dec() ↳ VTSE
MAX → inc() ↳ VTLE

specify opt.

Exercises:
* ① value < Counter.MAX_VALUE
  ② value > Counter.MAX_VALUE
** ① value < Counter.MIN_VALUE
  ② value > Counter.MIN_VALUE

Lecture 2b

Part B

Test-Driven Development (TDD) - Manual, Console Testers

Manual Tester 1 from the Console

```
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

correct → VTSE thrown as expected

What if decrement is implemented correctly?

```
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

incorrect → VTSE not thrown

Expected Behaviour:

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

What if decrement is implemented incorrectly?

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

Running Console Tester 1 on Correct Implementation

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

thrown as expected

```java
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

Running Console Tester 1 on Incorrect Implementation

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

```java
1   public class CounterTester1 {
2     public static void main(String[] args) {
3       Counter c = new Counter();
4       println("Init val: " + c.getValue());
5       try {
6         c.decrement();
7         println("Error: ValueTooSmallException NOT thrown.");
8       }
9       catch (ValueTooSmallException e) {
10        println("Success: ValueTooSmallException thrown.");
11      }
12    } /* end of main method */
13  } /* end of class CounterTester1 */
```

wrong imp. ⇒ VTSE not thrown

c.getValue == -1

Manual Tester 2 from the Console

Test Case 3

- Nothing unexpected occurs.
- Everything expected occurs.

Test Case 1

VTLE thrown unexpectedly

Test Case 2

VTLE not thrown as expected

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

expected VTLE not thrown

unexpectedly thrown VTLE

Running Console Tester 2 on (Correct) Implementation 1

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

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

As soon as the VTLE is handled, it won't be propagated further.

Running Console Tester 2 on (Incorrect) Implementation 2

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

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

Running Console Tester 2 on (Incorrect) Implementation 3

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

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

Exercise

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

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8      }
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     } /* end of inner catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

throws VTLE (prematurely).

this manipulation of the counter object will still proceed as if there was no error occurring beforehand.

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

A Manual, Iterative Console Tester

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

Lecture 2b

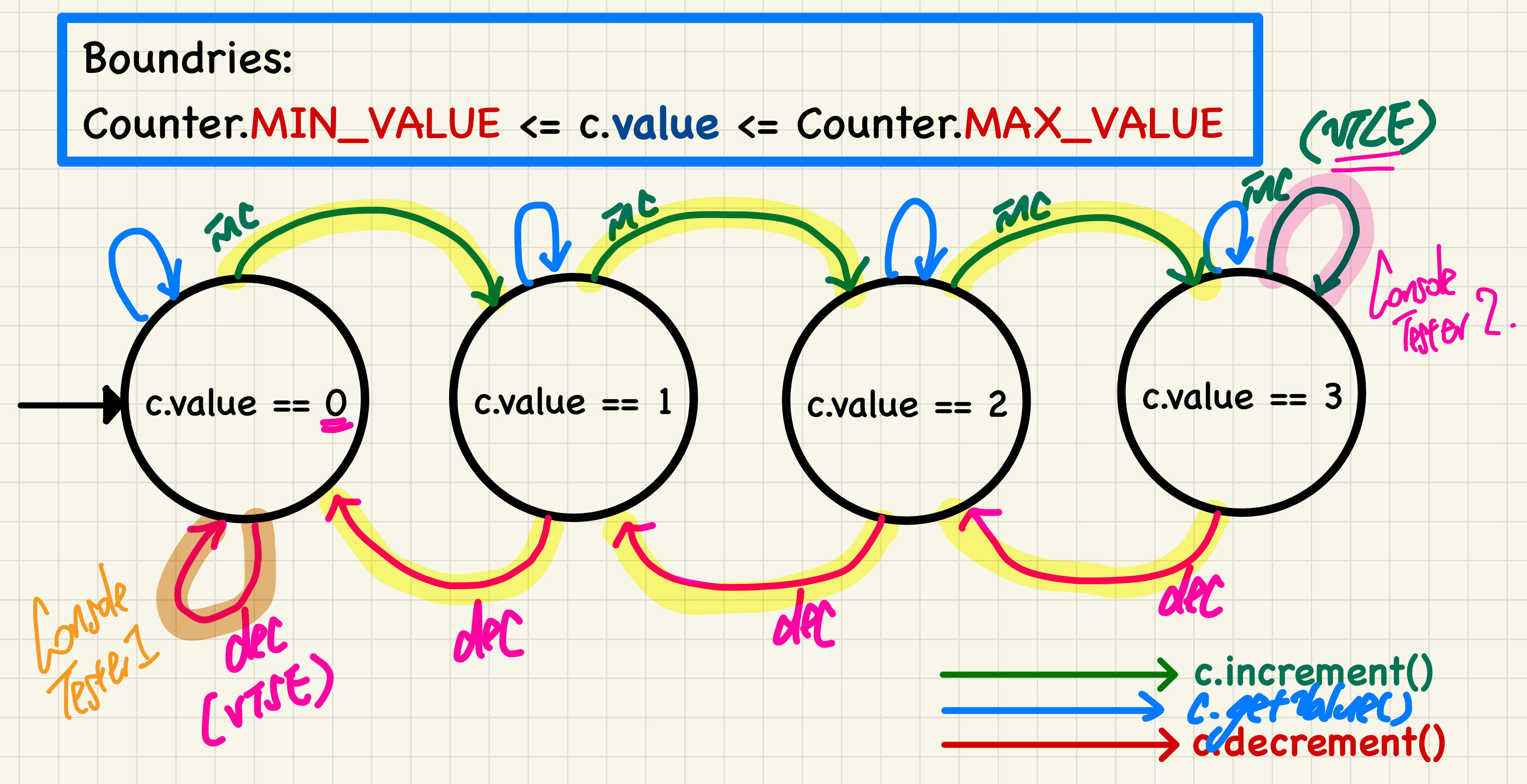
Part C

Test-Driven Development (TDD) -
Test Cases for a Bounded Variable

Coming Up with Test Cases: A Single, Bounded Variable

Boundries:

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



Lecture 2b

Part D

Test-Driven Development (TDD) - JUnit Testing via Assertions

A Default Test Case that Fails

The result of running a test is considered:

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

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
```

② No assertions run to check expected vs. actual values.

useless

∵ ① no meaningful manipulation of objects instantiated from classes in the model package.

do nothing.

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

Examples: JUnit Assertions (1)

Consider the following class:

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

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

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

Examples: JUnit Assertions (2)

Consider the following class:

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

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

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

Lecture 2b

Part E

Test-Driven Development (TDD) - Automated, JUnit Test Cases

JUnit: Where an Exception is 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(ValueTooBigException e) {
10      /* Exception is not expected to be thrown. */
11      fail ("ValueTooBigException is not expected.");
12    }
13  }
```

automated.

VTLE not expected

counterpart in Console Tester: println(c.getValue());

examine manually that 0 is printed to console

reaching this line means VTLE thrown unexpectedly.

What if increment is implemented correctly?

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

What if decrement is implemented incorrectly?

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

JUnit: Where an Exception is Expected (1)

JUnit Test

```
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 }
```

doing nothing is the most trivial way for passing a test

Console Tester

```
1  public class CounterTester1 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Init val: " + c.getValue());
5      try {
6        c.decrement();
7        println("Error: ValueTooSmallException NOT thrown.");
8      }
9      catch (ValueTooSmallException e) {
10       println("Success: ValueTooSmallException thrown.");
11     }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

JUnit: where an Exception is Expected (2)

Console Tester

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

JUnit Test

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

VTLE → unexpected

→ if this assertion is executed, the entire test method will terminate and fail.

→ fail → VTLE not thrown as expected.

→ test passes

Exercise

Why is the JUnit test **logically correct** but the **Console Tester** is 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 }
```

↳ reaching this line will cause the rest of the test method to be bypassed and fail.

→ logically correct

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8      }
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     } /* end of inner catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

↳ executing this line will not prevent the rest of the method from being executed → logically incorrect

→ inappropriate if there was already an error.

Exercise

Q: Can we rewrite `testIncFromMaxValue` to:

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

Hint: Say Line 12 is executed,

is it clear if that ValueTooLargeException was thrown as expected?

Testing Many Values in a Single Test

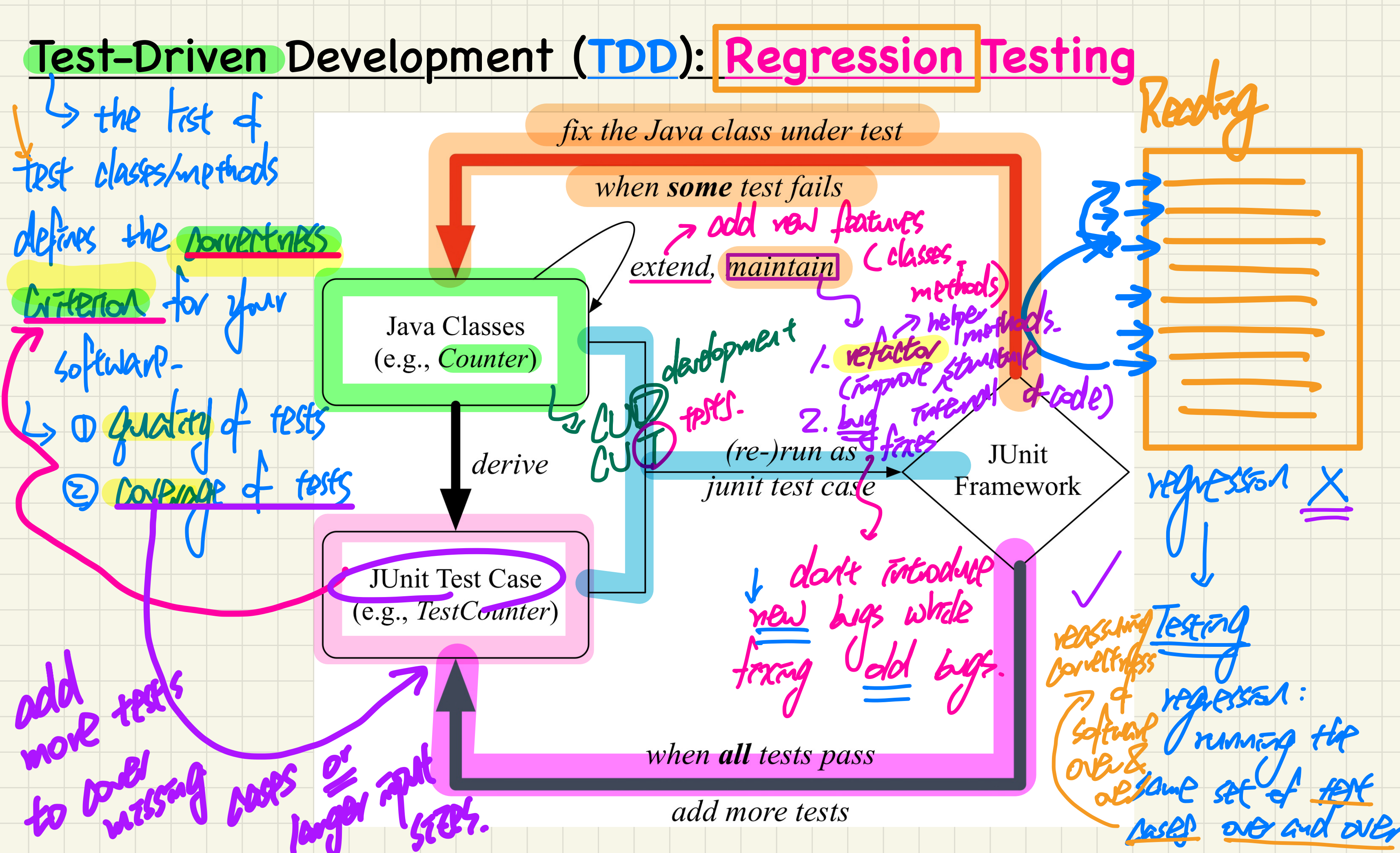
Loops can make it effective on generating test cases:

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

Lecture 2b

Part F

Test-Driven Development (TDD) - Regression Testing


Test-Driven Development (TDD): Regression Testing
fix the Java class under test
when some test fails
extend, maintain
Java Classes
(e.g., Counter)
derive
JUnit Test Case
(e.g., TestCounter)
(re-)run as
junit test case
JUnit
Framework
when all tests pass
add more tests
the list of test classes/methods defines the correctness criterion for your software.
① quality of tests
② coverage of tests
add more tests to cover missing cases or longer input seqs.
add new features (classes, methods)
1. refactor (improve structure of code)
helper methods
internal
2. bug fixes
development
CUT
tests
don't introduce new bugs while fixing old bugs.
Reading
regression X
Testing
regression: running the same set of test cases over and over
reassuring correctness of software over & over

Lecture 3

Part A

Object Equality -
To Override or Not to Override

The equals Method: To Override or Not?

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

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

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

extends

extends

Inherited to each sub/child class unless its overridden (redefined).

Compare references/addresses of this and obj

this: context object of method call

obj: argument

parent

super

child

sub

explicitly override the equals method.

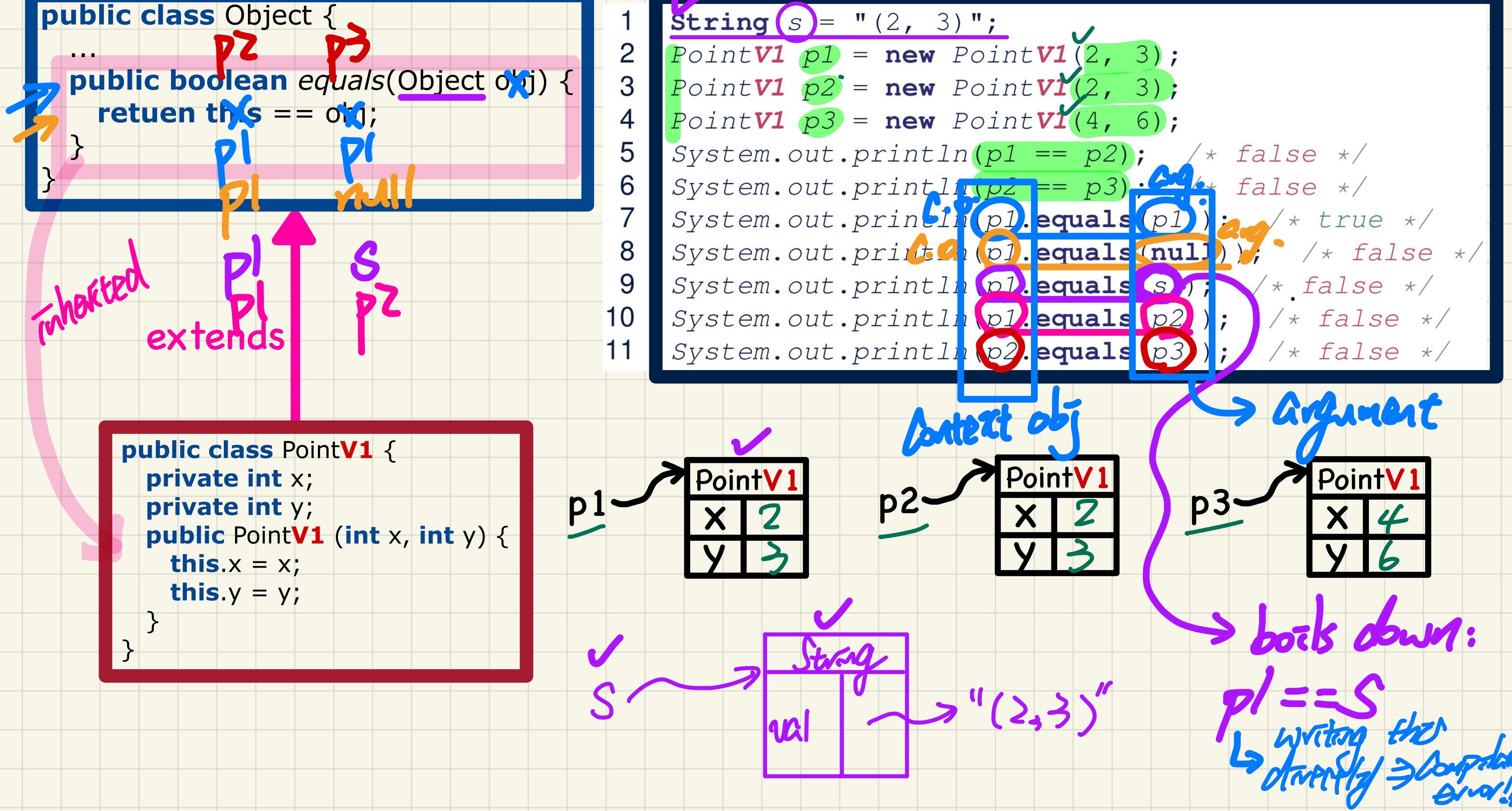
Implicitly, equals from Object inherited

Lecture 3

Part B

Object Equality -
Version 1: Default equals method

The equals Method: Default Version



Lecture 3

Part C

Object Equality -
Version 2: Overridden equals method

The equals Method: Overridden Version

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

extends

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

PointV2	
x	
y	

PointV2	
x	
y	

The equals Method: Overridden Version Example 1

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

extends

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

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

PointV2 (p1)	
x	2
y	3

PointV2 (p2)	
x	2
y	3

PointV2 (p3)	
x	4
y	6

The equals Method:

To Override or Not?

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

p1 p2

extends

inherited

extends

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

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

PointV2

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

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

p1 → PointV1 (x: 2, y: 3)

p2 → PointV1 (x: 2, y: 3)

p1 → PointV2 (x: 2, y: 3)

p2 → PointV2 (x: 2, y: 3)

other

The equals Method: Overridden Version

Example 2

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

extends

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

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

obj1, obj2 → ?

p1 → PointV2: x = 3, y = 4

p2 → PointV2: x = 3, y = 4

p3 → PointV2: x = 4, y = 5

(A) Two objects are reference-equal.

(B) Two objects are contents-equal.

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

not necessarily true ⇒ p1 vs. p2.

Lecture 3

Part D

Object Equality - assertSame vs. assertEquals in JUnit

assertEquals: Reference Comparison or Not

assertEquals(exp1, exp2)

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

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

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2); × /* ∵ different PointV1 objects */
assertEquals(p2, p3); × /* ∵ different types of objects */
```

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

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2); × /* ≈ p1.equals(p2) ≈ p1 == p2 */
assertEquals(p2, p3); × /* ≈ p2.equals(p3) ≈ p2 == p3 */
assertEquals(p3, p2); × /* ≈ p3.equals(p2) ≈ p3.getClass()==p2.getClass() */
```

Lecture 3

Part E

Object Equality -
Asserting Reference vs. Object Equality

Testing Default Equality of Points in JUnit

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

p1 == p2 ⇒ False (default from object)

p2 == p1 ⇓ False

p1 → PointV1: x = 3, y = 4

p2 → PointV1: x = 3, y = 4

```java
public class Object {

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

extends

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

Testing Overridden Equality of Points in JUnit

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

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

extends

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

d.t.

d.t.

p3.equals(p4)

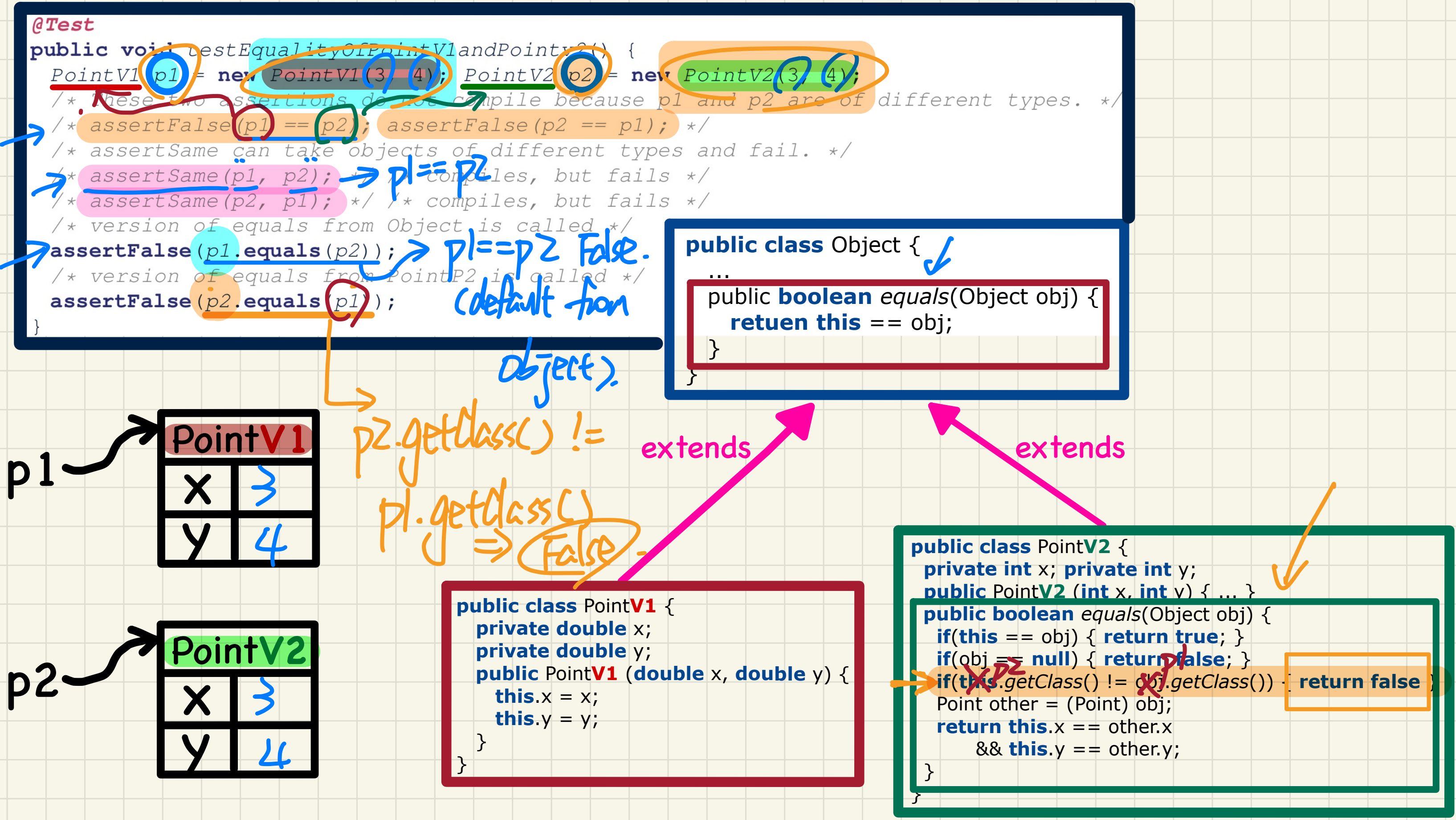
overridden version from PointV2

p4.equals(p3)

p3 → PointV2	
x	3
y	4

p4 → PointV2	
x	3
y	4

Testing Equality of Points in JUnit: Default vs. Overridden



Lecture 3

Part F

Object Equality -
Short-Circuit Effect of && and ||

Short-Circuit Evaluation: &&

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

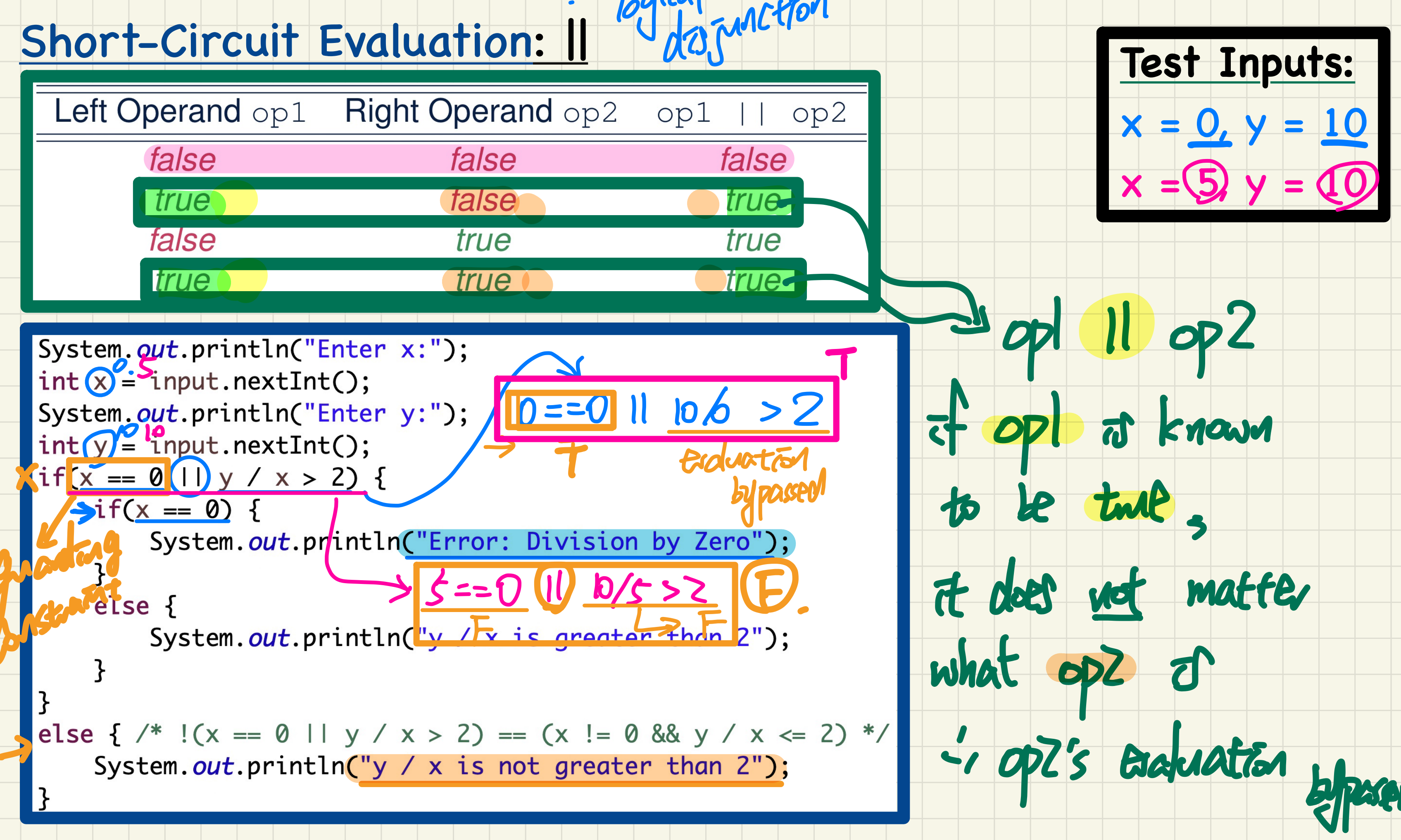
Test Inputs:

x = 0, y = 10

x = 5, y = 10

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

op1 && op2: if op1 is known to be false, it does not matter what op2 evaluates to ∴ evaluation of op2 can be bypassed.


Short-Circuit Evaluation: ||
logical disjunction
Left Operand op1
Right Operand op2
op1 || op2
false
false
false
true
false
true
false
true
true
true
true
true
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x == 0 || y / x > 2) {
if(x == 0) {
System.out.println("Error: Division by Zero");
}
else {
System.out.println("y / x is greater than 2");
}
}
else { /* !(x == 0 || y / x > 2) == (x != 0 && y / x <= 2) */
System.out.println("y / x is not greater than 2");
}
0 == 0 || 10/0 > 2
evaluation bypassed
5 == 0 || 10/5 > 2
guarding constraint
Test Inputs:
x = 0, y = 10
x = 5, y = 10
op1 || op2
if op1 is known to be true, it does not matter what op2 is
∴ op2's evaluation bypassed

Short-Circuit Evaluation: Common Errors

Test Inputs:

x = 0, y = 10

op1 && op2
op1 || op2] evaluation at runtime is from left to right

order of G.C. is crucial

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

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

10/0 > 2 && 0 != 0

division by zero error

G.C. useless

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

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

10/0 <= 2 || 0 == 0

division by zero error

G.C. useless

Lecture 3

Part G

***Object Equality -
Exercises on the equals method***

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

```java
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 }
```

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

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

```java
if(this.getClass() != obj.getClass() || obj == null)
```

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

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

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

v3

```
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 }
```

D.T. Person.

Context object: dynamic type? Person

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

v2

v1: equals from Object class

```
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 }
```

v2

Lecture 3

Part H

Object Equality -
Equality of Array-Typed Attribute

Testing Equality of Person/PersonCollector in JUnit (1)

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

Exercise.
How are these two assertions passed differently?

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

p1

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

p2 (other)

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

p3, p4

p1.getClass()

Recall:
Being Reference equal implies Contents equal.

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

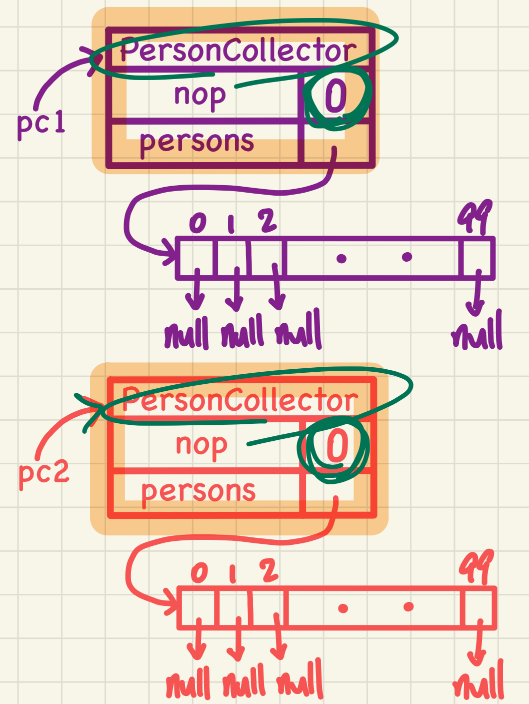
p3 == p4

String

Testing Equality of Person/PersonCollector in JUnit (2)

(continued from testPersonCollector)

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



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

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

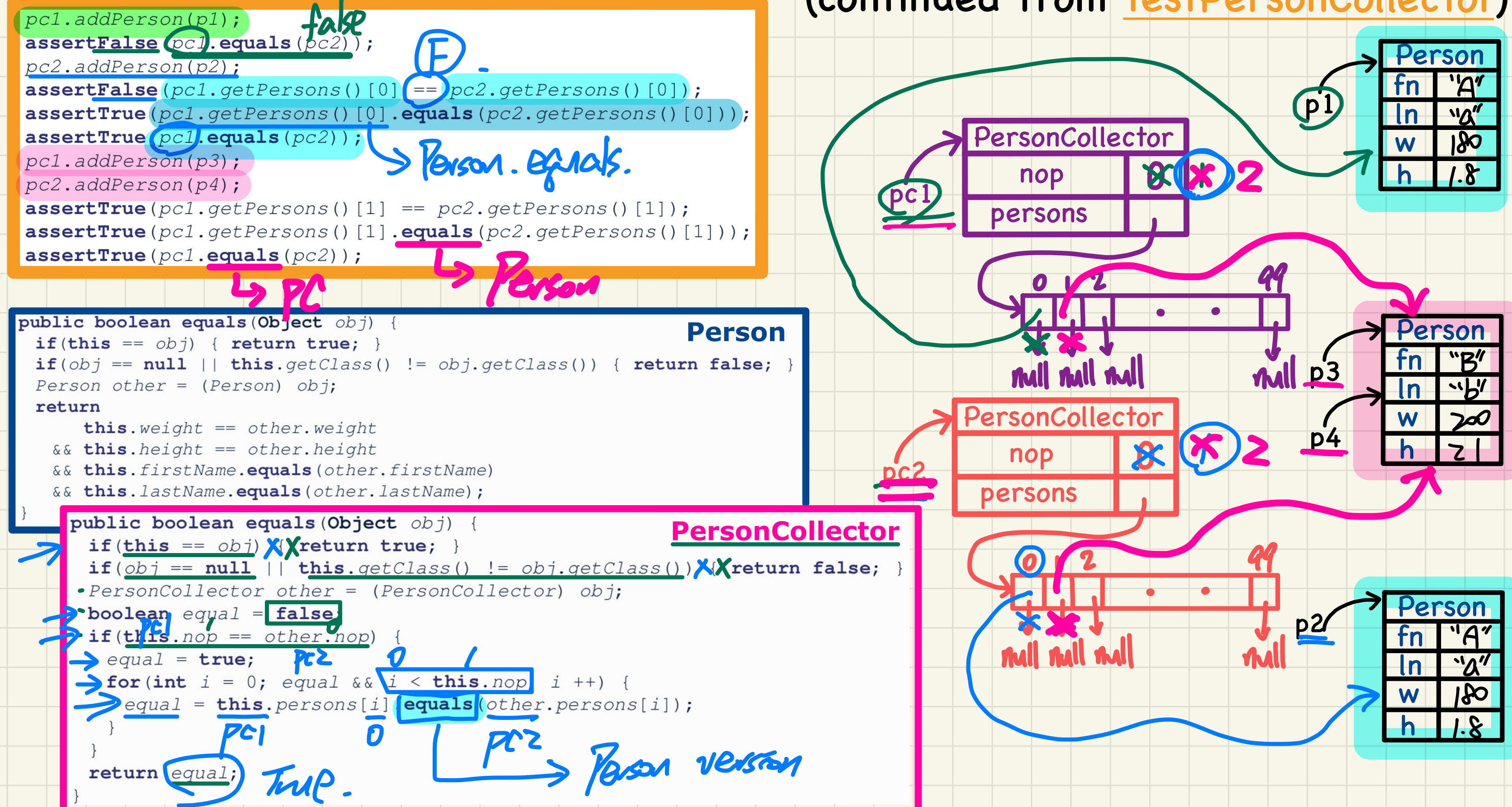
```
public boolean equals(Object obj) {
  if(this == obj) { return true; }
  if(obj == null || this.getClass() != obj.getClass()) { return false;
  PersonCollector other = (PersonCollector) obj;
  boolean equal = false;
  if(this.nop == other.nop) {
    equal = true;
    for(int i = 0; equal && i < this.nop;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

0 < 0 ⇒ F ↳ not Entering the loop.

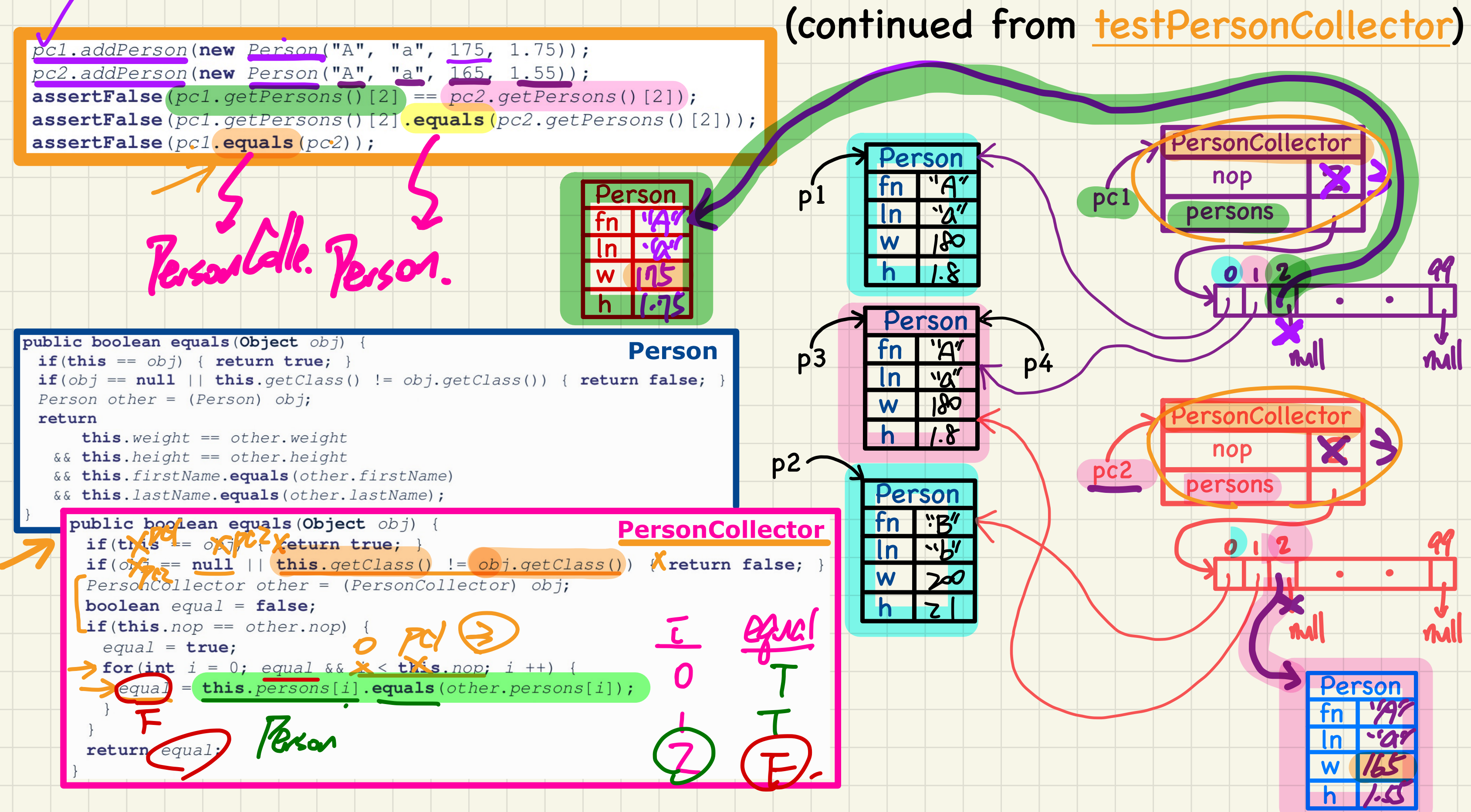
T → empty PCs are equal.

Testing Equality of Person/PersonCollector in JUnit (3)

(continued from testPersonCollector)



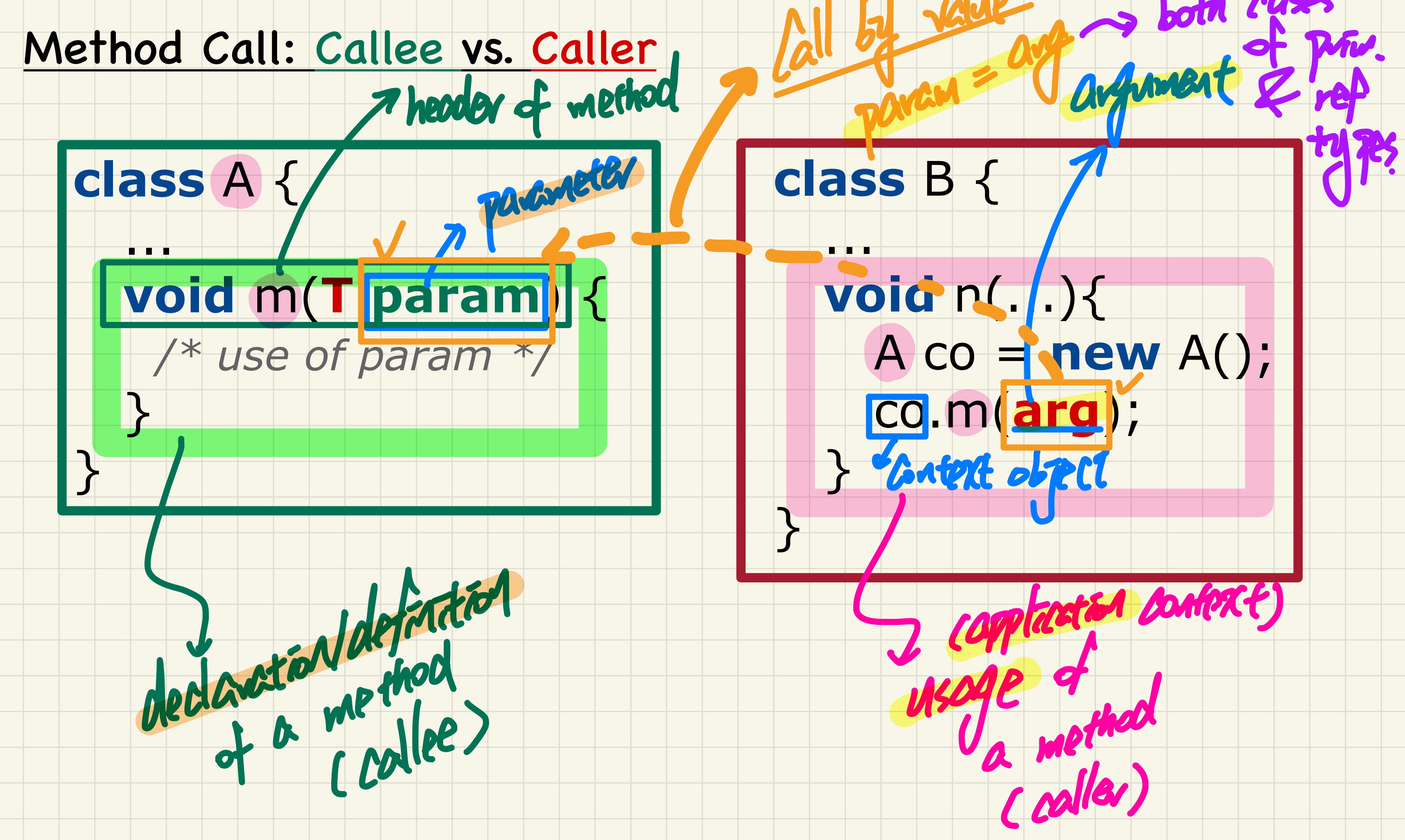
Testing Equality of Person/PersonCollector in JUnit (4)



Lecture 4

Part A

Call by Value -
Primitive vs. Reference Arguments



Method Call: Callee vs. Caller
class A {
...
void m(T param) {
/* use of param */
}
}
class B {
...
void n(...){
A co = new A();
co.m(arg);
}
}
header of method
parameter
Call by value
param = arg
both cases of prim. & ref types
argument
context object
declaration/definition of a method (callee)
(application context)
usage of a method (caller)

Call by Value: Primitive Argument

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

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

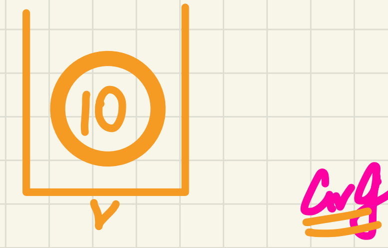
callee/def

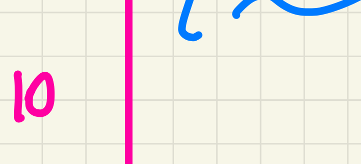
r = arg

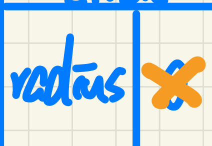
param

caller/application

argument







Call by Value: Reference Argument

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

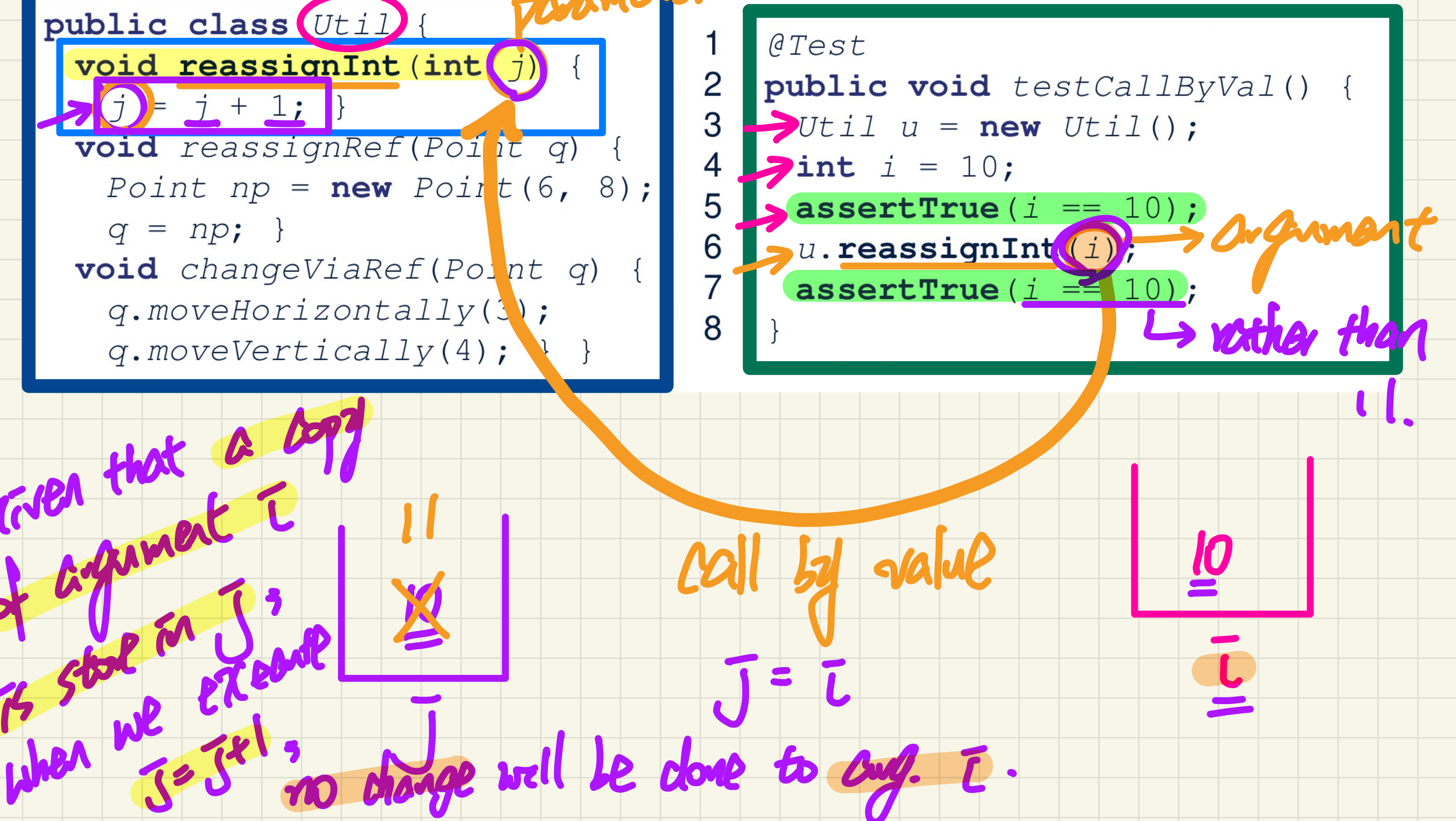
```
class CircleUser {
  ...
  Circle c = new Circle();
  Circle arg = new Circle(10);
  c.setRadius(arg);
  }
}
```

Lecture 4

Part B

Call by Value -
Asserting Call by Value in JUnit

Call by Value: Re-Assigning Primitive Parameter



Call by Value: Re-Assigning Reference Parameter

Will executing this line redirect arg. p?

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

```java
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 }
```

call by value

q = p

p has not been re-assigned by the method

only var being re-assigned

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

Call by Value: Calling Mutator on Reference Parameter

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

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

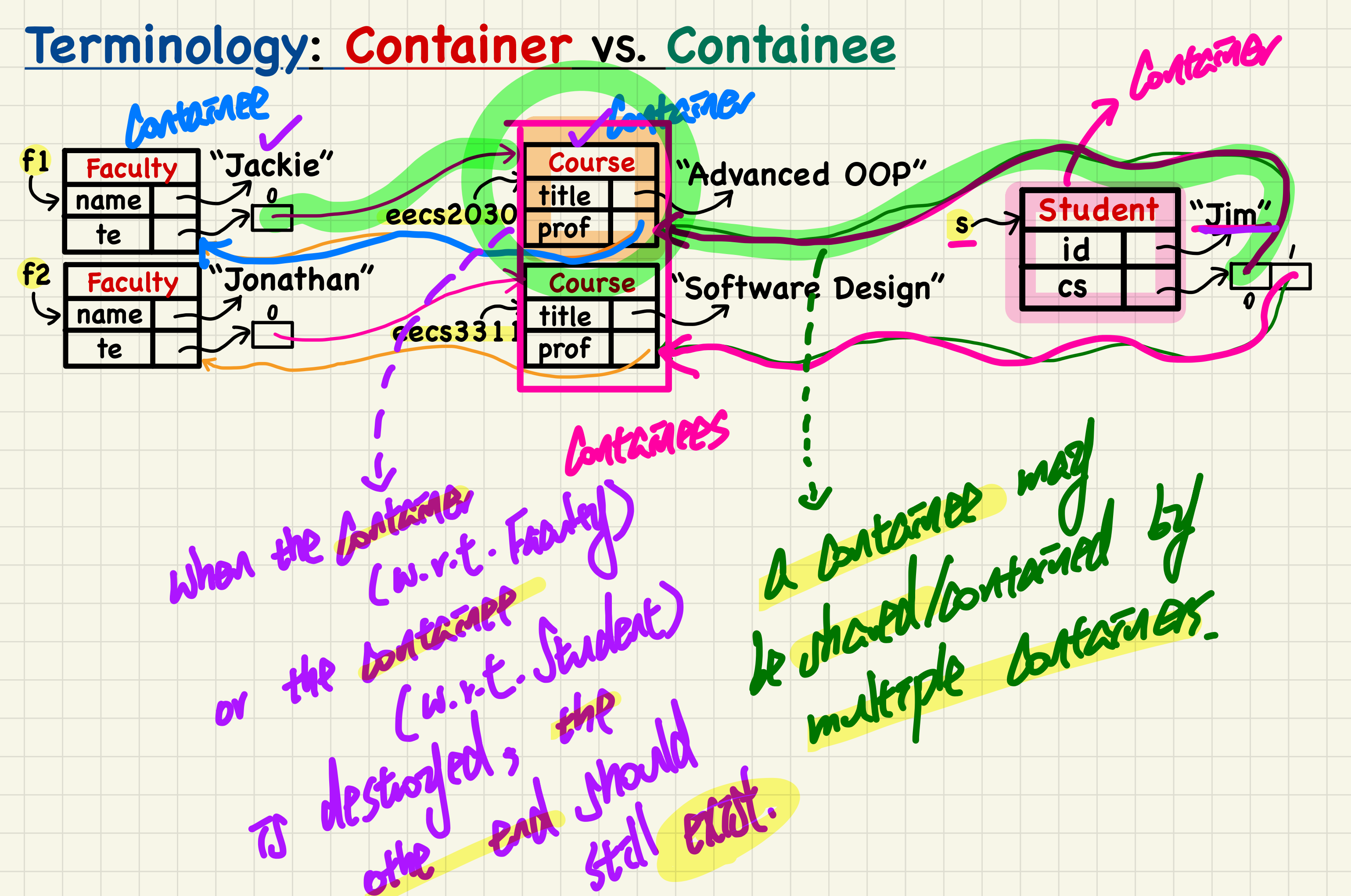
```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y) { this.y += y; }
  public void moveHorizontally(int x) {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

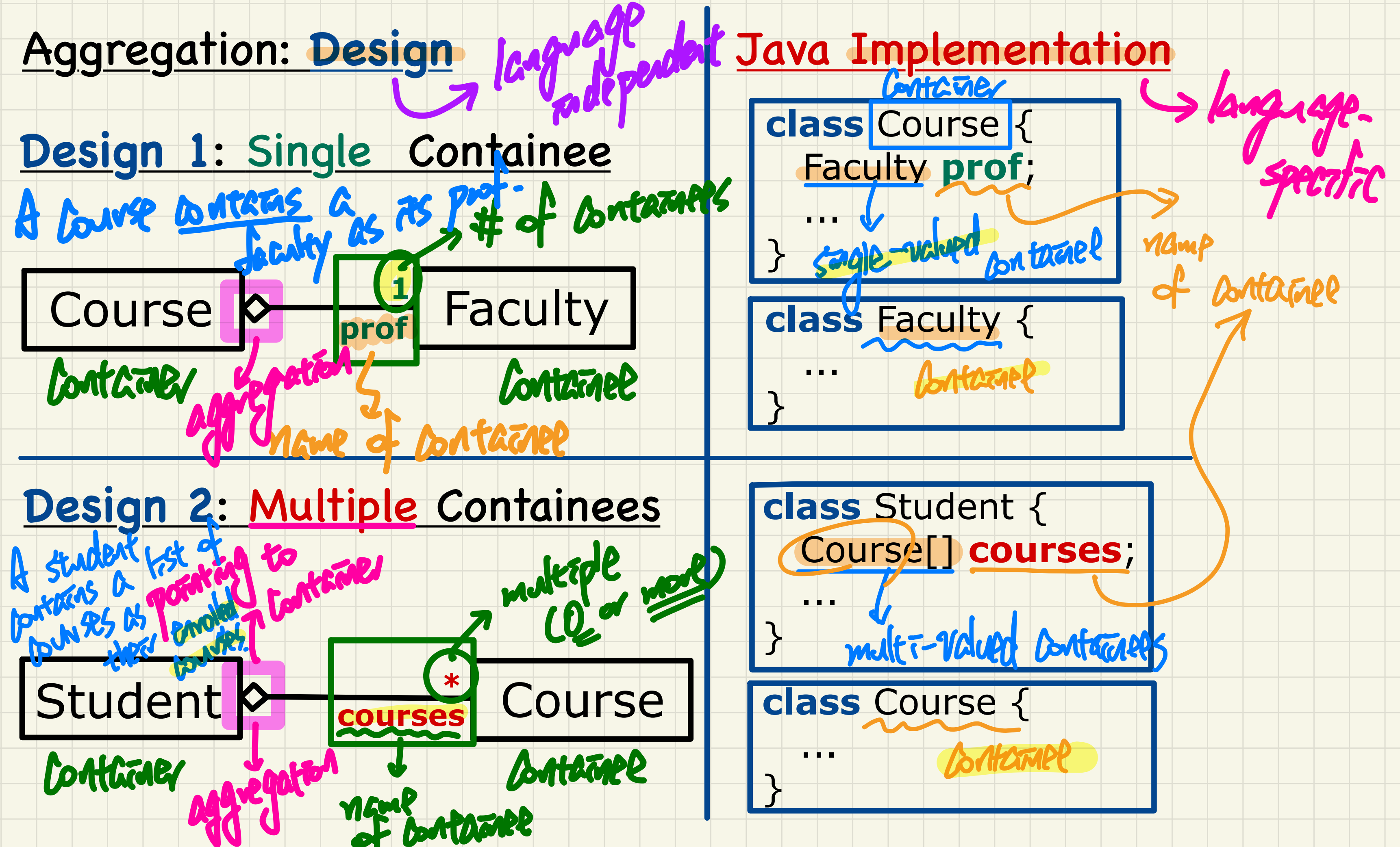
Lecture 4

Part C

Aggregation and Composition - Terminology, Modelling, and Implementation

Terminology: Container vs. Containee



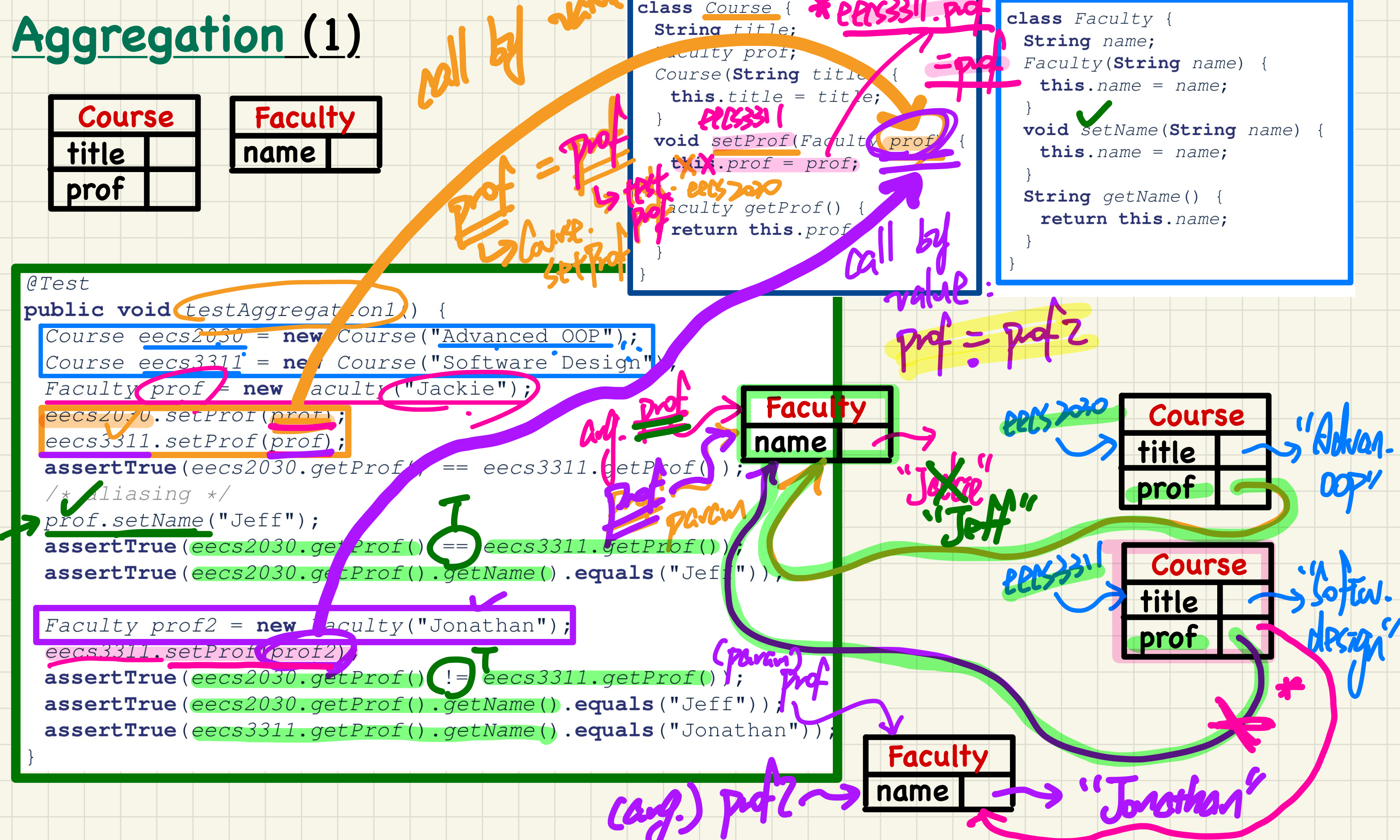


Aggregation: Design
Java Implementation
Design 1: Single Containee
Course
1
prof
Faculty
class Course {
Faculty prof;
...
}
class Faculty {
...
}
Design 2: Multiple Containees
Student
*
courses
Course
class Student {
Course[] courses;
...
}
class Course {
...
}

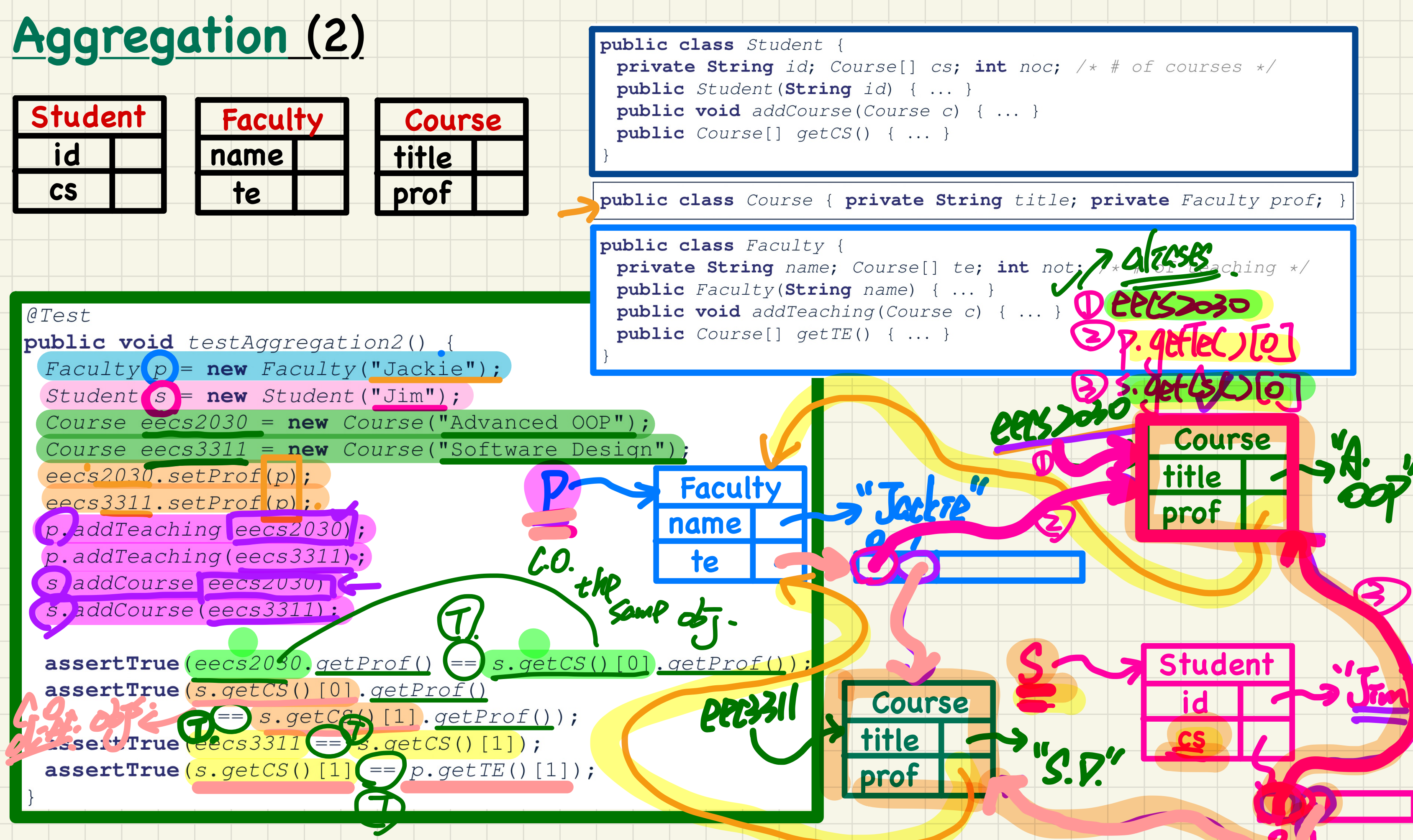
Lecture 4

Part D

Aggregation and Composition - Building Aggregated Object Structure



Aggregation (1)
Course
title
prof
Faculty
name
class Course {
String title;
Faculty prof;
Course(String title) {
this.title = title;
}
void setProf(Faculty prof) {
this.prof = prof;
}
Faculty getProf() {
return this.prof;
}
}
class Faculty {
String name;
Faculty(String name) {
this.name = name;
}
void setName(String name) {
this.name = name;
}
String getName() {
return this.name;
}
}
@Test
public void testAggregation1() {
Course eecs2030 = new Course("Advanced OOP");
Course eecs3311 = new Course("Software Design");
Faculty prof = new Faculty("Jackie");
eecs2030.setProf(prof);
eecs3311.setProf(prof);
assertTrue(eecs2030.getProf() == eecs3311.getProf());
/* aliasing */
prof.setName("Jeff");
assertTrue(eecs2030.getProf() == eecs3311.getProf());
assertTrue(eecs2030.getProf().getName().equals("Jeff"));
Faculty prof2 = new Faculty("Jonathan");
eecs3311.setProf(prof2);
assertTrue(eecs2030.getProf() != eecs3311.getProf());
assertTrue(eecs2030.getProf().getName().equals("Jeff"));
assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
call by value:
call by value:
prof = prof2
"Jackie"
"Jeff"
"Jonathan"
"Advan. OOP"
"Softw. design"



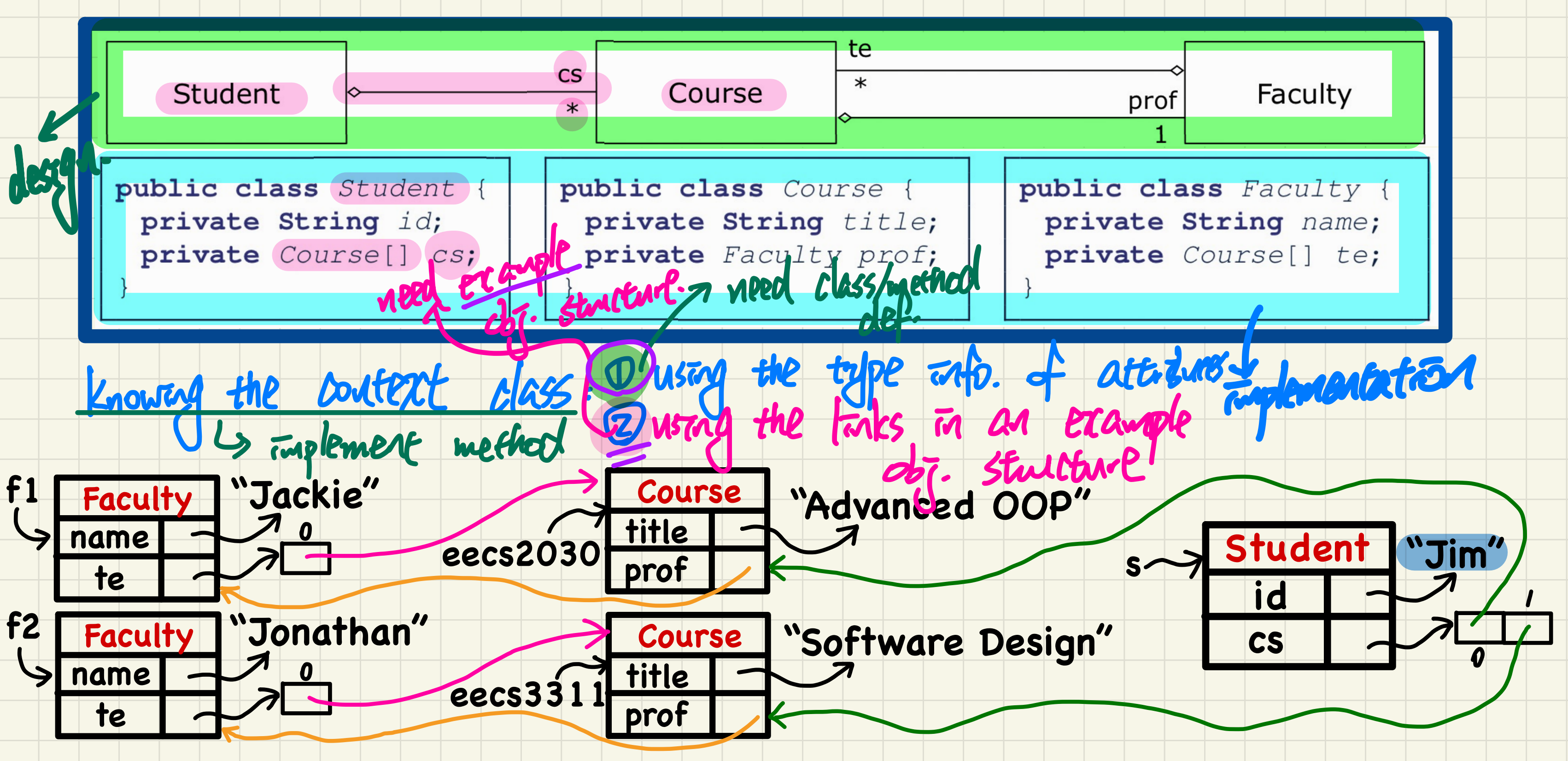
Aggregation (2)
Student
id
cs
Faculty
name
te
Course
title
prof
public class Student {
private String id; Course[] cs; int noc; /* # of courses */
public Student(String id) { ... }
public void addCourse(Course c) { ... }
public Course[] getCS() { ... }
}
public class Course { private String title; private Faculty prof; }
public class Faculty {
private String name; Course[] te; int not; /* # of teaching */
public Faculty(String name) { ... }
public void addTeaching(Course c) { ... }
public Course[] getTE() { ... }
}
@Test
public void testAggregation2() {
Faculty p = new Faculty("Jackie");
Student s = new Student("Jim");
Course eecs2030 = new Course("Advanced OOP");
Course eecs3311 = new Course("Software Design");
eecs2030.setProf(p);
eecs3311.setProf(p);
p.addTeaching(eecs2030);
p.addTeaching(eecs3311);
s.addCourse(eecs2030);
s.addCourse(eecs3311);
assertTrue(eecs2030.getProf() == s.getCS()[0].getProf());
assertTrue(s.getCS()[0].getProf() == s.getCS()[1].getProf());
assertTrue(eecs3311 == s.getCS()[1]);
assertTrue(s.getCS()[1] == p.getTE()[1]);
}

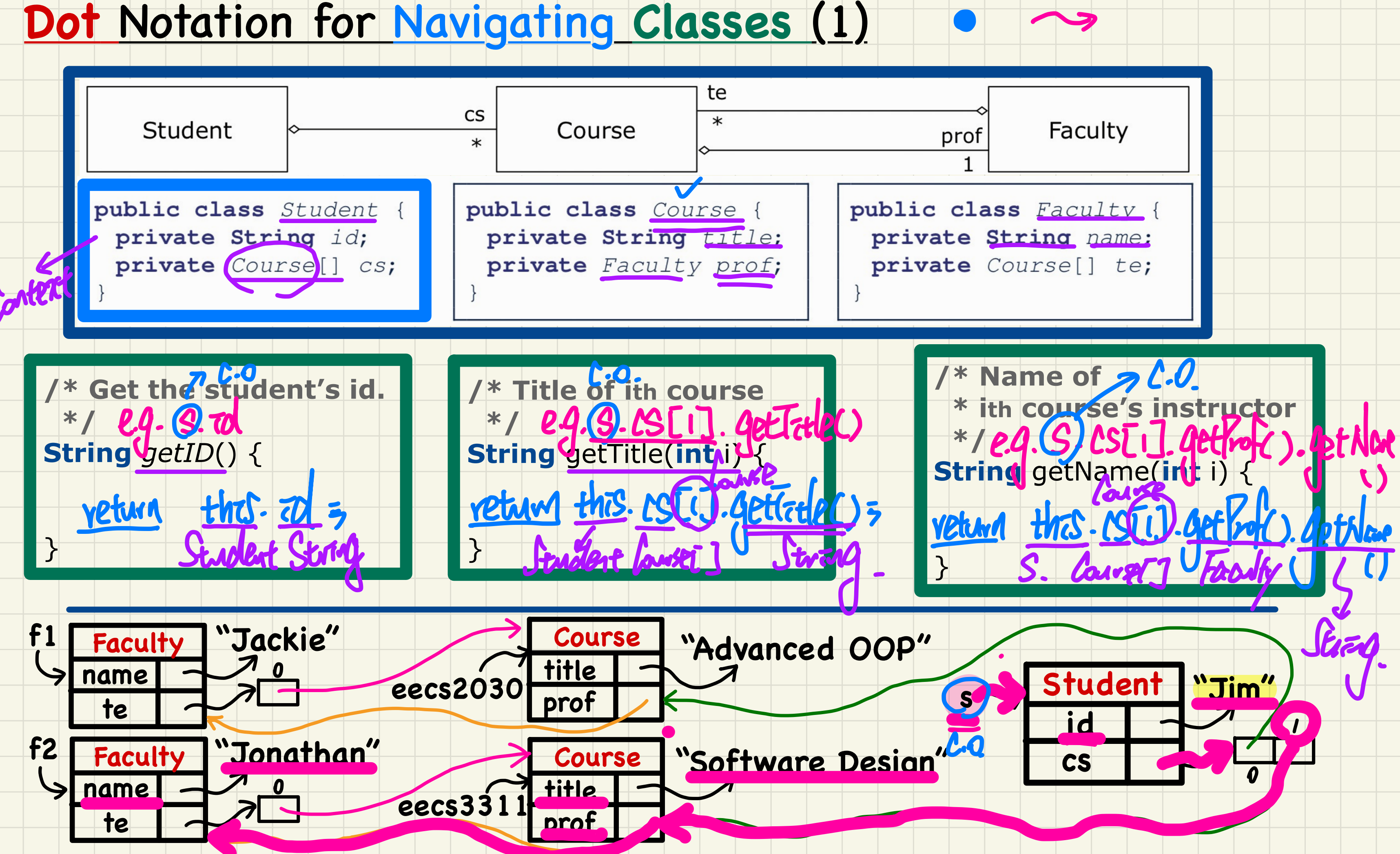
Lecture 4

Part E

Aggregation and Composition -
Navigating Objects via Aggregation Links

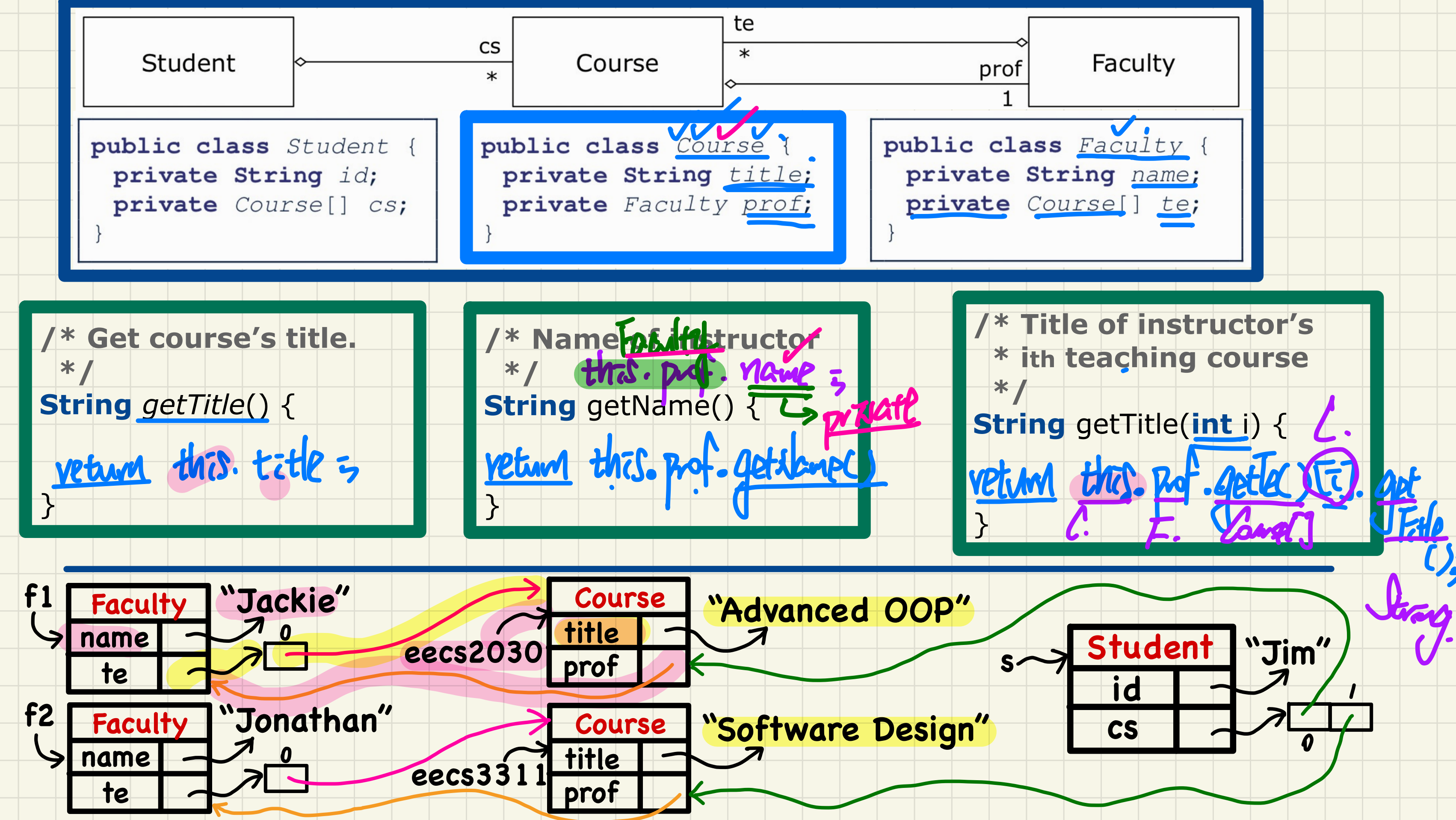
Runtime Object Structure: Student, Course, Faculty



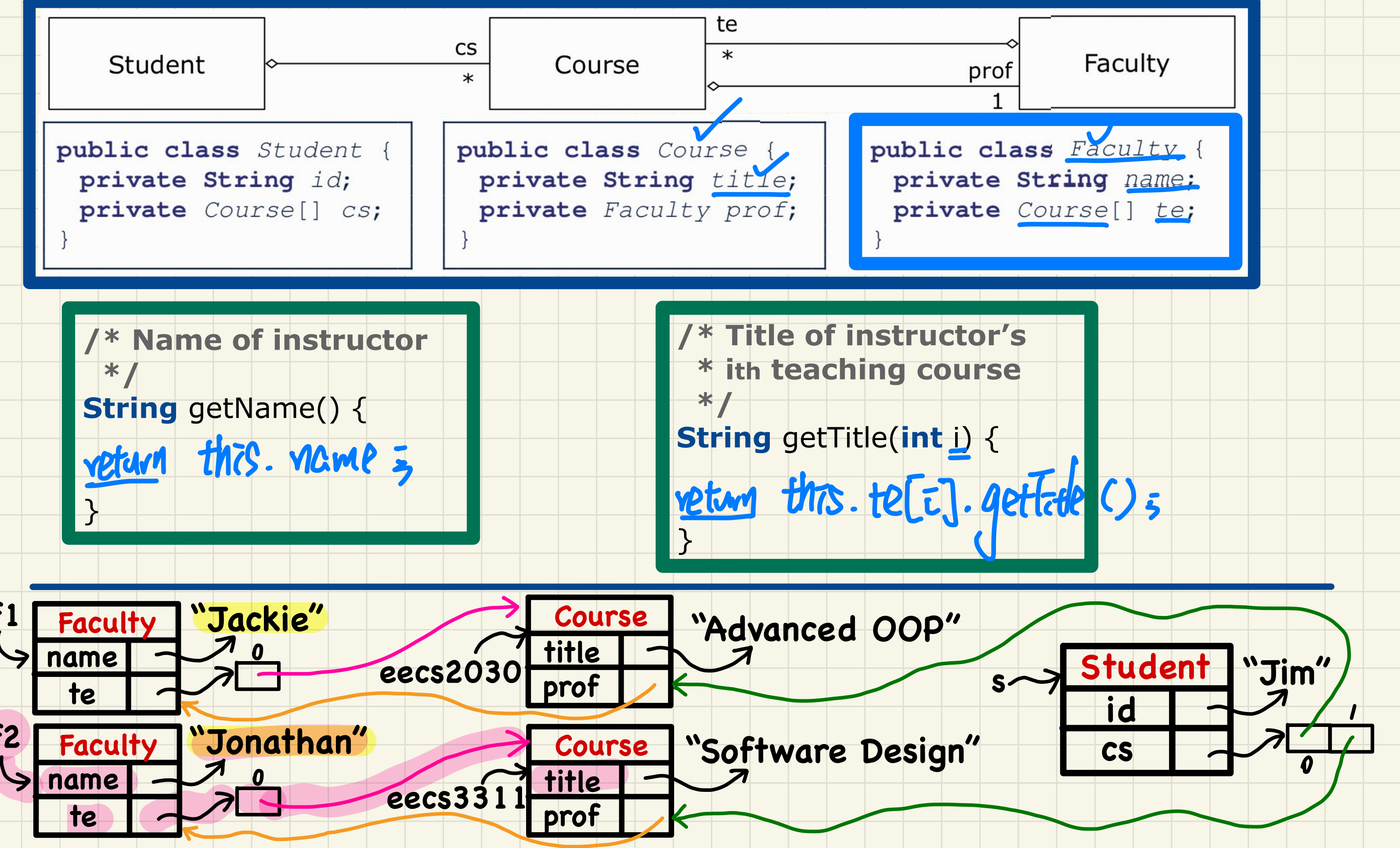


Dot Notation for Navigating Classes (1)
Student
cs
*
Course
te
*
prof
1
Faculty
public class Student {
private String id;
private Course[] cs;
}
public class Course {
private String title;
private Faculty prof;
}
public class Faculty {
private String name;
private Course[] te;
}
/* Get the student's id. */
String getID() {
}
/* Title of ith course */
String getTitle(int i) {
}
/* Name of ith course's instructor */
String getName(int i) {
}
f1
Faculty
name
te
"Jackie"
0
f2
Faculty
name
te
"Jonathan"
0
Course
title
prof
eecs2030
"Advanced OOP"
Course
title
prof
eecs3311
"Software Design"
s
Student
id
cs
"Jim"
0

Dot Notation for Navigating Classes (2)



Dot Notation for Navigating Classes (3)

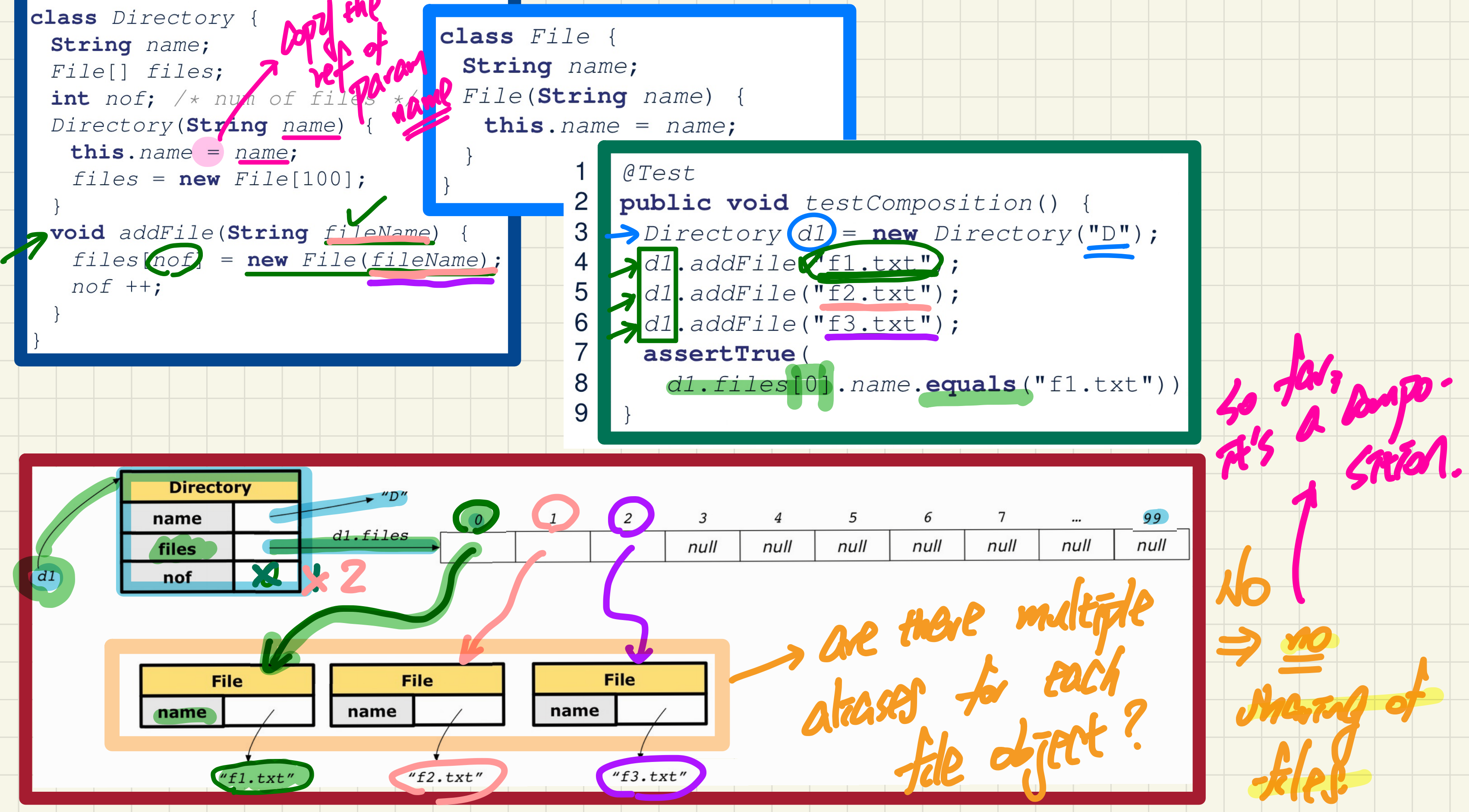


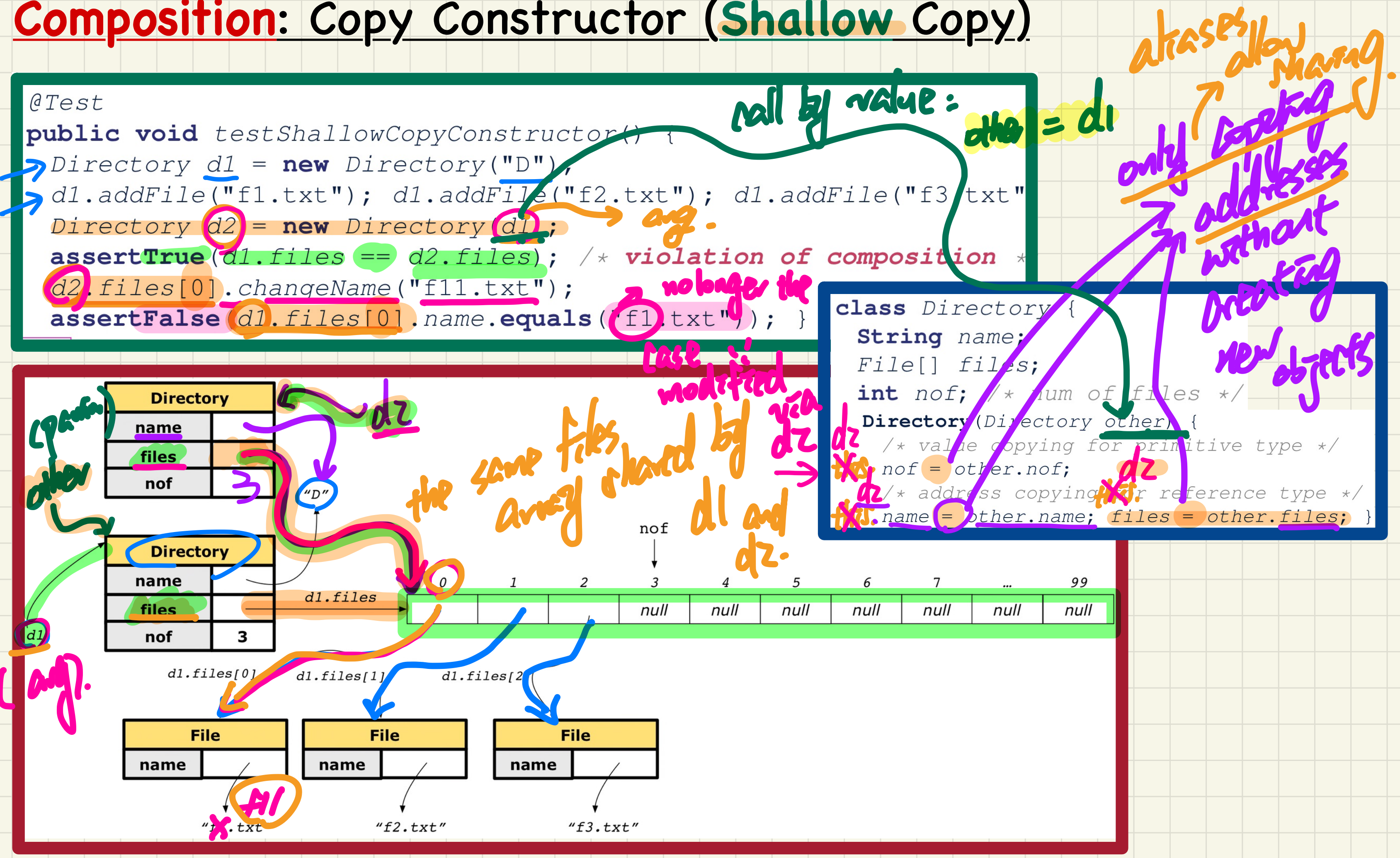
Lecture 4

Part F

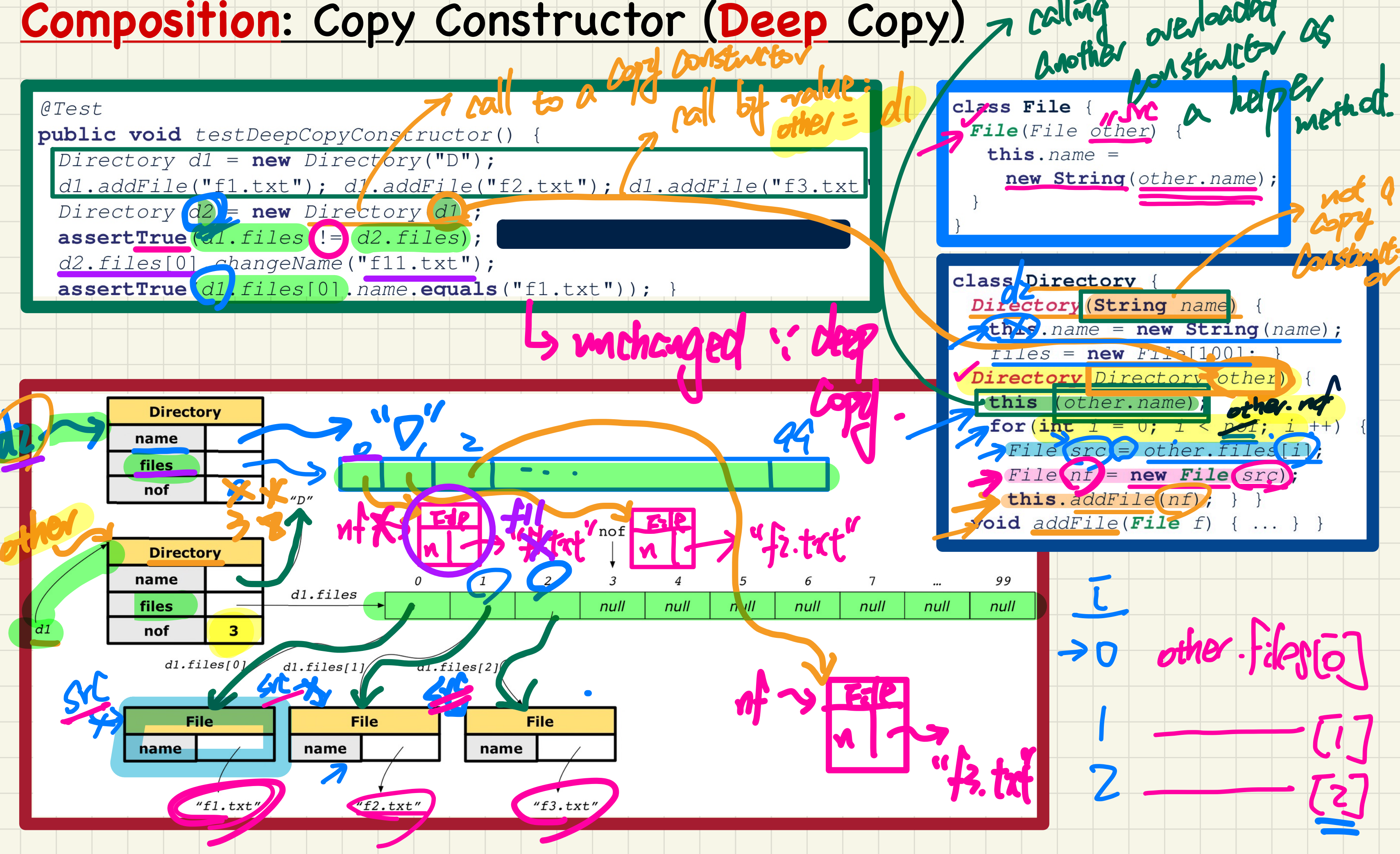
Aggregation and Composition - Implementing Composition via Copy Constructors

Composition: No Sharing



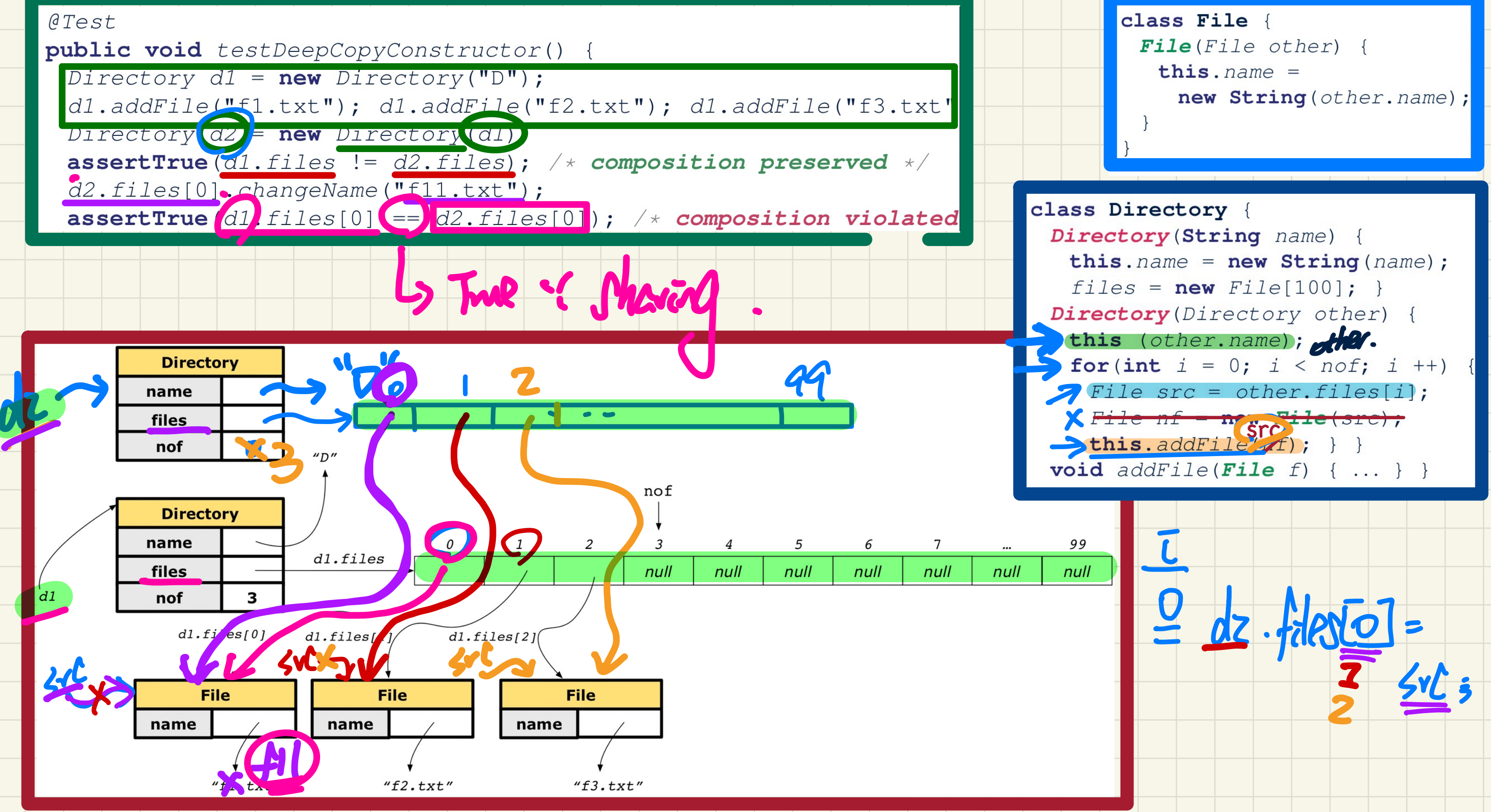


Composition: Copy Constructor (Shallow Copy)
@Test
public void testShallowCopyConstructor() {
Directory d1 = new Directory("D");
d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt"
Directory d2 = new Directory(d1);
assertTrue(d1.files == d2.files); /* violation of composition *
d2.files[0].changeName("f11.txt");
assertFalse(d1.files[0].name.equals("f1.txt")); }
class Directory {
String name;
File[] files;
int nof; /* num of files */
Directory(Directory other) {
/* value copying for primitive type */
nof = other.nof;
/* address copying for reference type */
name = other.name; files = other.files; }
Directory
name
files
nof
3
"D"
Directory
name
files
nof
3
d1
d1.files
nof
0
1
2
3
4
5
6
7
...
99
null
null
null
null
null
null
null
d1.files[0]
d1.files[1]
d1.files[2]
File
name
File
name
File
name
".txt"
"f2.txt"
"f3.txt"
call by value:
other = d1
arg.
the same files array shared by d1 and d2.



Composition: Copy Constructor (Deep Copy)
@Test
public void testDeepCopyConstructor() {
Directory d1 = new Directory("D");
d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt
Directory d2 = new Directory(d1);
assertTrue(d1.files != d2.files);
d2.files[0].changeName("f11.txt");
assertTrue(d1.files[0].name.equals("f1.txt")); }
class File {
File(File other) {
this.name =
new String(other.name);
}
}
class Directory {
Directory(String name) {
this.name = new String(name);
files = new File[100]; }
Directory(Directory other) {
this(other.name);
for(int i = 0; i < nof; i ++) {
File src = other.files[i];
File nf = new File(src);
this.addFile(nf); } }
void addFile(File f) { ... } }
Directory
name
files
nof
"D"
Directory
name
files
nof
3
d1
d1.files
0
1
2
3
4
5
6
7
...
99
null
null
null
null
null
null
null
nof
d1.files[0]
d1.files[1]
d1.files[2]
File
name
"f1.txt"
"f2.txt"
"f3.txt"
call to a copy constructor
call by value: other = d1
calling another overloaded constructor as a helper method
not a copy constructor
unchanged ∵ deep copy
other.files[0]

Exercise: Copy Constructor (Composition?)

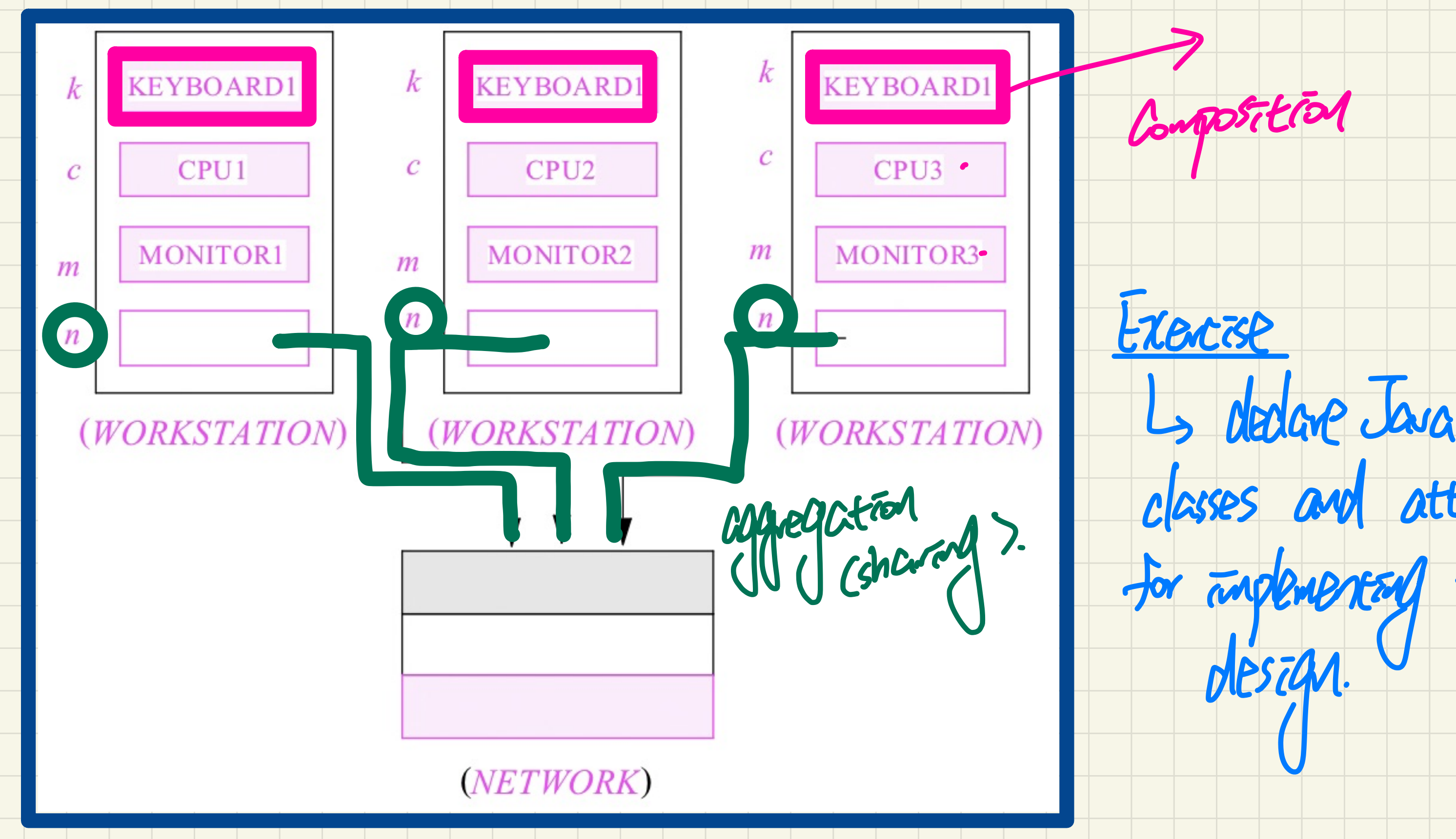


Lecture 4

Part G

Aggregation and Composition - Example and Exercise

Modelling: Aggregation vs. Composition



Exercise

↳ declare Java classes and attributes for implementing this design.

Implementation: Aggregation or Composition

What if the author field gets modified?

author as an *aggregation*	author as a *composition*
Hyperlinked author page	Physical printed copies

Aggregation:

"The Red and the Black"
1830
341
reference

"Life of Rossini"
1823
307
reference

Shared author object (URL):

"Stendhall"
"Henri Beyle"
1783
~~1842~~ 1865

change is visible to all containers (books) sharing this author page

Composition:

"The Red and the Black"
1830
341
"Stendhall" / "Henri Beyle" / 1783 / ~~1842~~ 1865

"Life of Rossini"
1823
307
"Stendhall" / "Henri Beyle" / 1783 / 1842

↳ unchanged.

change only applied to its owning container.

Lecture 5

Part A

Inherited - Student Management System: First-Design (without inheritance)

Inheritance: Motivating Problem

relevant : ① experience ② trials & errors

Nouns -> classes, attributes, accessors

Verbs -> mutators

common attributes

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

NRS

RS

applicable to only one kind of students

First Design Attempt

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

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

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

encoding — 1: RS, 2: NRS

RS: Student rs = new Student(1);

NRS: Student nrs = new Student(2);

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

```java
public double 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 double 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 ++;
  }
}
```

only applicable to RS.

only applicable to NRS.

not applicable ⇒ wasted space

RS: Student rs = new Student(1);

Good design?

Judge by **Cohesion**

↳ In a single class, all attributes and methods are related to each other under a common theme.

First Design Attempt

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

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

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

kind == 3 : international

```java
public double getTuition(){
  double tuition = 0;
  for(int i = 0; i < this.noc; i++){
    tuition += this.courses[i].fee;
  }
  if (this.kind == 1) {
    return tuition * this.premiumRate;
  }
  else if (this.kind == 2) {
    return tuition * this.discountRate;
  }
} else if (this.kind == 3) { ... }
```

```java
public double 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 (this.kind == 3){ ... }

multiple places to repeat WRK

Good design?

Judge by Single Choice Principle

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

When a change is needed, there's only a single (or min # of) place to make such change.

Lecture 5

Part B

***Inheritance -
Student Management System:
Second-Design (without inheritance)***

Testing Student Classes (without inheritance)

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

1000 * 1.25

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

1000 * 0.75

Cohesion ✓

duplicates ⇒ violating single choice principle

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

Res.S.
rcs
pr 1.25
jim

0 1 9

Course
title "EECS2030"
fee 500
c1

Course
title "EECS3311"
fee 500
c2

NonRes.S.
rcs
dr 0.75
jeremy

0 1 9

Student Classes (without inheritance): Maintenance (1)

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

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

if(noc >= MAX){ ... }

if(noc >= MAX){ ... }

Maintenance e.g., a new registration constraint:

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

Student Classes (without inheritance): Maintenance (2)

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

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

* IV

* IV

Maintenance e.g., a new tuition formula:

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

A Collection of Students (without inheritance)

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

Idea!

Student[] to store both kinds of student, while satisfying cohesion & SCP.

⇒ Inheritance.

rss → 0 1 2 . . . 99

Res.S.	
rcs	
pr	

nrss → 0 1 2 . . . 99

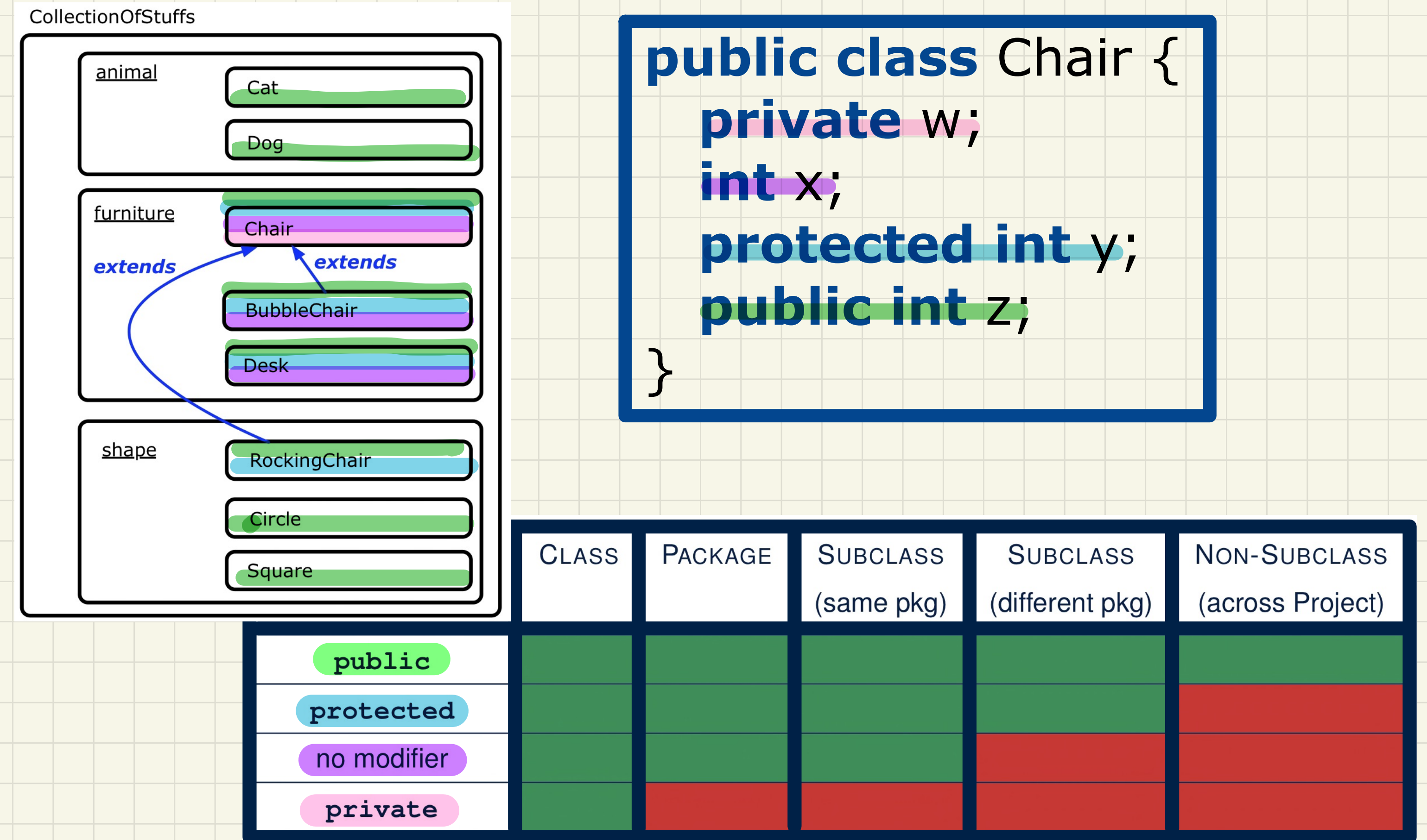
NonRes.S.	
rcs	
dr	

Lecture 5

Part C

Inheritance -
Visibility: Project, Package, (Sub-)Classes

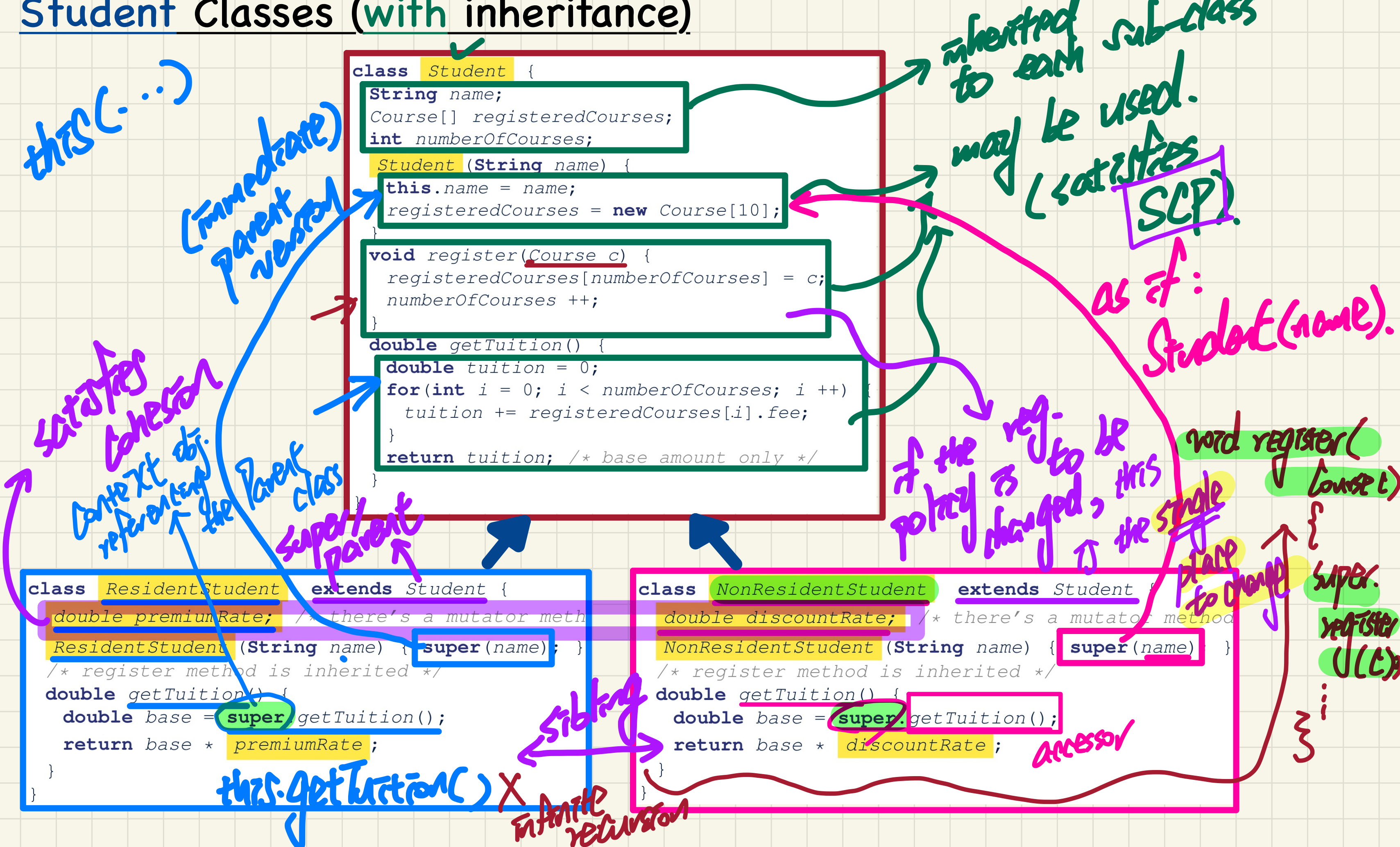
Visibility: Attributes and Methods



Lecture 5

Part D

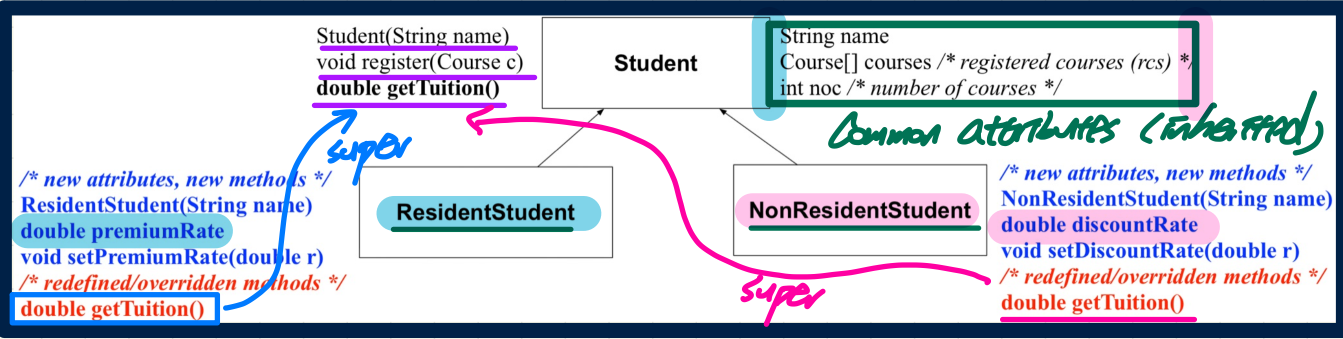
Inheritance -
Student Management System:
Third-Design (with inheritance)


Student Classes (with inheritance)
class Student {
String name;
Course[] registeredCourses;
int numberOfCourses;
Student (String name) {
this.name = name;
registeredCourses = new Course[10];
}
void register(Course c) {
registeredCourses[numberOfCourses] = c;
numberOfCourses ++;
}
double getTuition() {
double tuition = 0;
for(int i = 0; i < numberOfCourses; i ++) {
tuition += registeredCourses[i].fee;
}
return tuition; /* base amount only */
}
}
class ResidentStudent extends Student {
double premiumRate; /* there's a mutator meth
ResidentStudent (String name) { super(name); }
/* register method is inherited */
double getTuition() {
double base = super.getTuition();
return base * premiumRate;
}
}
class NonResidentStudent extends Student {
double discountRate; /* there's a mutator method
NonResidentStudent (String name) { super(name); }
/* register method is inherited */
double getTuition() {
double base = super.getTuition();
return base * discountRate;
}
}

Visualizing Parent and Child Objects

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

Testing Student Classes (with inheritance)




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

jim → Res.S. | rcs | pr 1.25 → [0 1 ... 9]

c1 → Course | title 2030 | fee 500

c2 → Course | title 3311 | fee 500

jeremy → NonRes.S. | rcs | dr 0.75 → [0 1 ... 9]

Designs vs. Principles

	Inheritance?	Cohesion?	single-choice principle
D1			
D2			
D3			

Lecture 5

Part E

Inheritance -
Static Types, Code Reuse, Expectations

Recall: Student Classes (with inheritance)

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

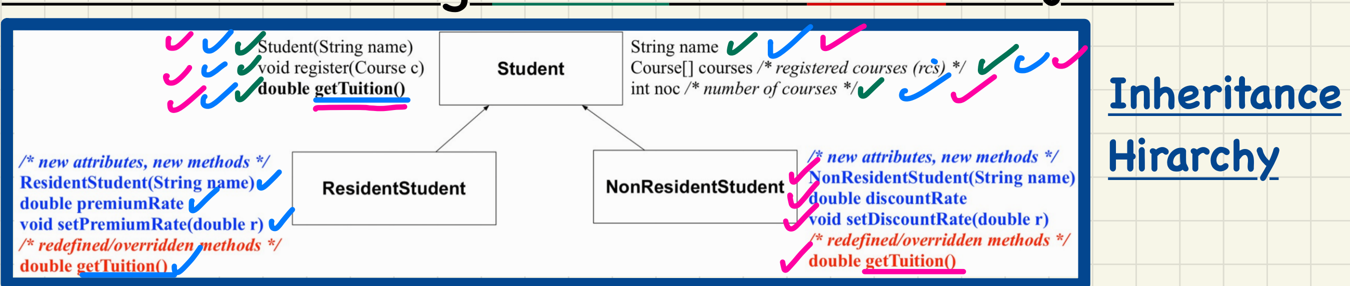
overridden

overridden

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

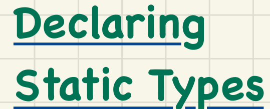
```
class NonResidentStudent extends Student {
  double discountRate; /* there's a mutator method
  NonResidentStudent (String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * discountRate;
  }
}
```

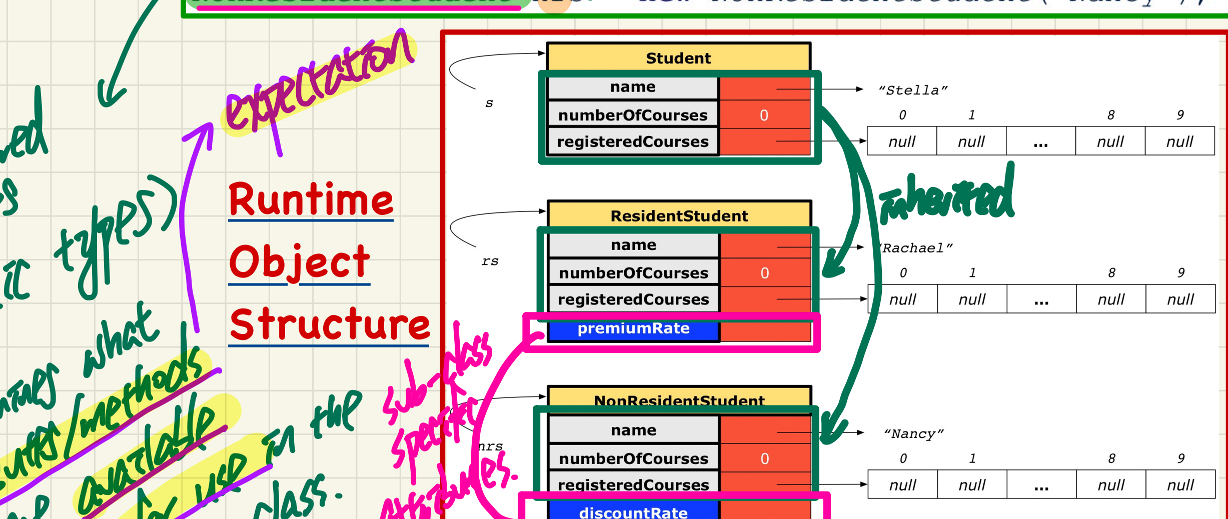
Recall: Visualizing Parent and Child Objects



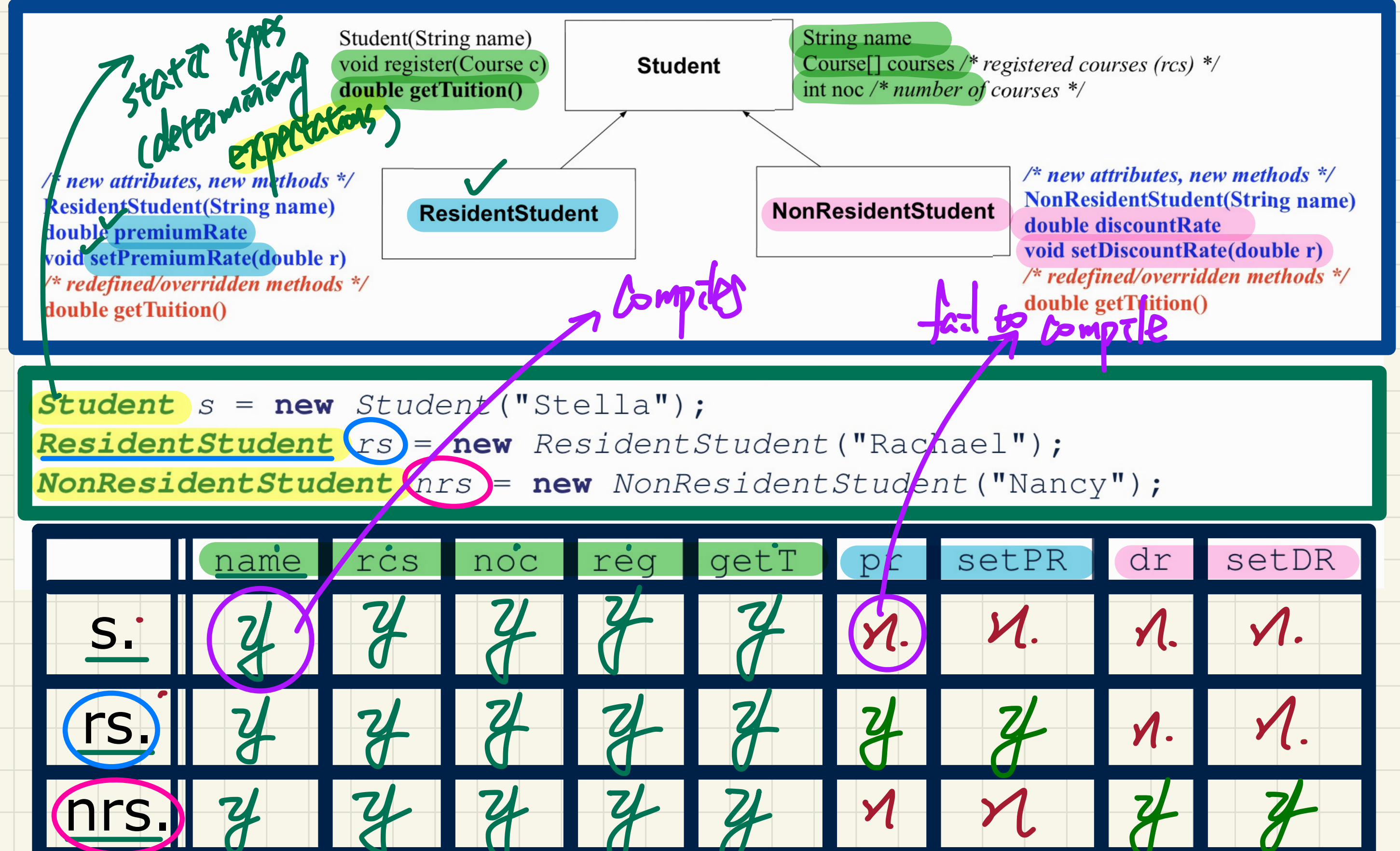
Inheritance Hirarchy

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





Student Classes (with inheritance): Expectations

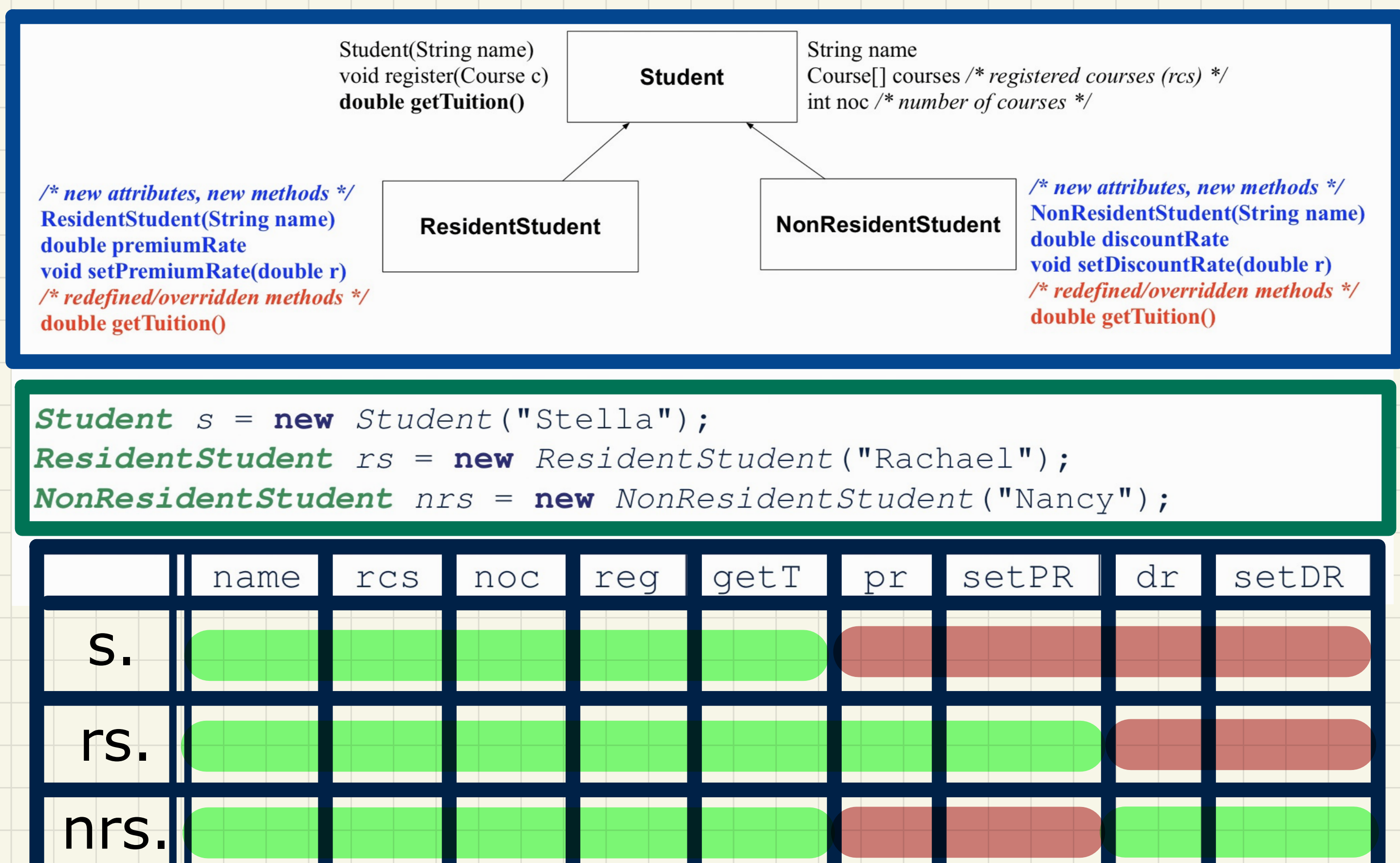


Lecture 5

Part F

Inheritance -
Intuition:
Polymorphism & Dynamic Binding

Recall: Student Classes (with inheritance): Expectations



Intuition: Polymorphism

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

Student

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

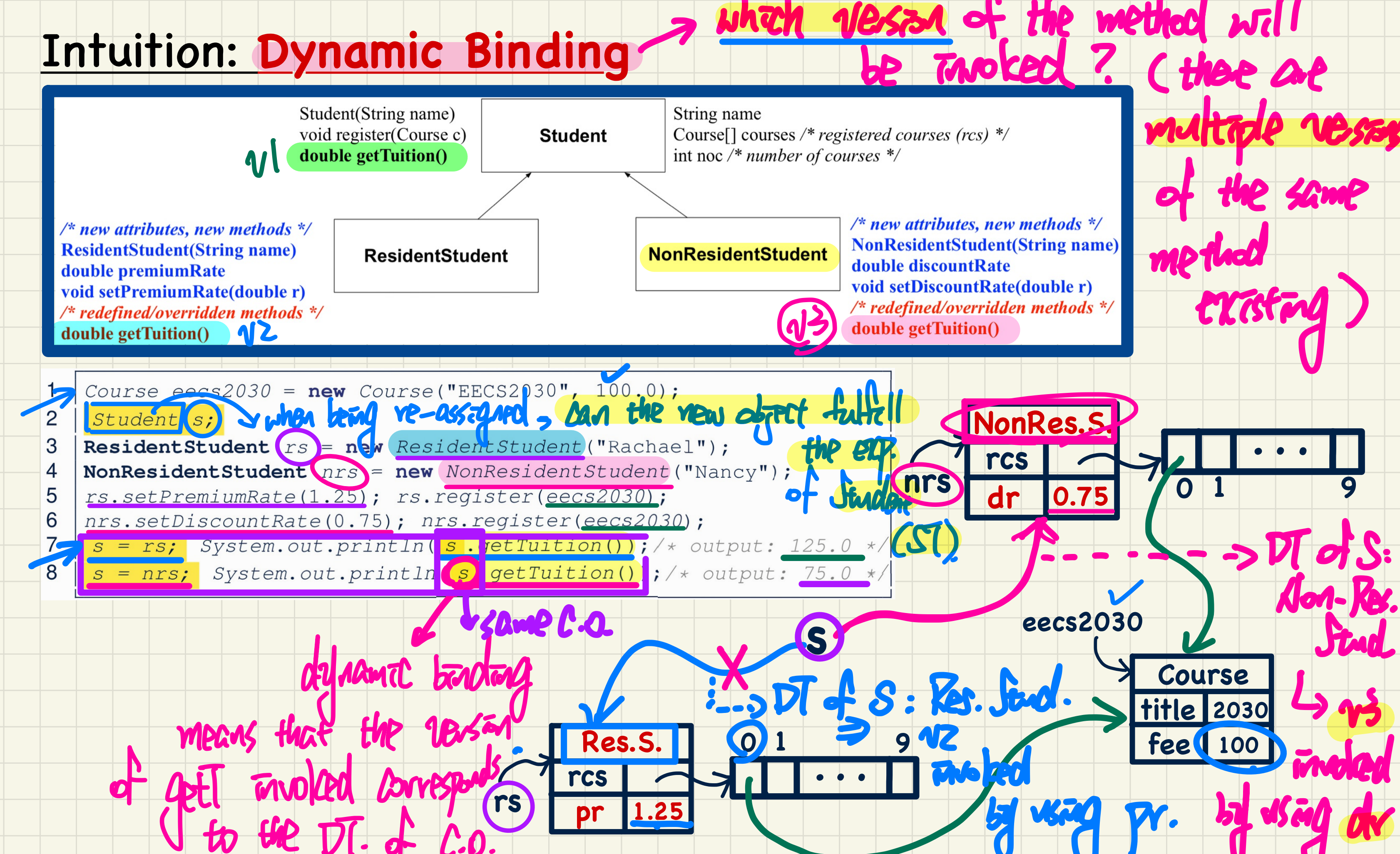
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()

ResidentStudent

NonResidentStudent

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

```
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? */
```



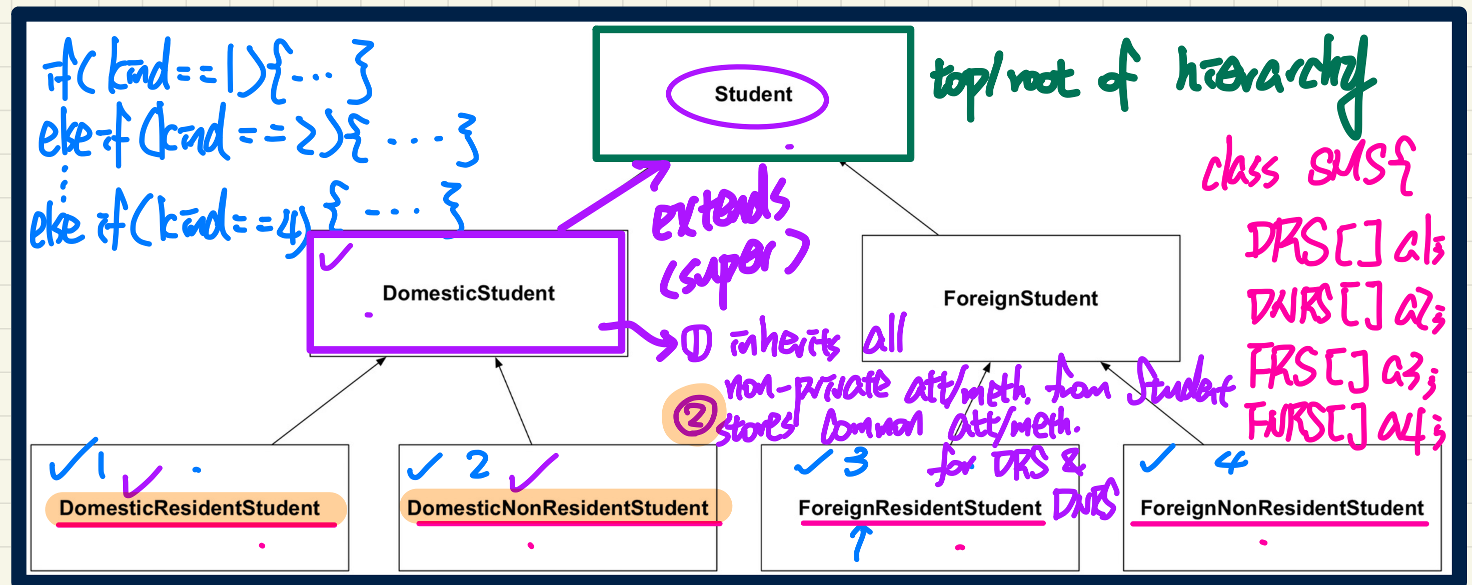
Intuition: Dynamic Binding
which version of the method will be invoked? (there are multiple versions of the same method existing)
Student(String name)
void register(Course c)
double getTuition()
v1
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
v2
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
v3
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()); /* output: 125.0 */
8 s = nrs; System.out.println(s.getTuition()); /* output: 75.0 */
when being re-assigned, can the new object fulfill the exp. of Student (ST)
same C.O.
dynamic binding means that the version of get invoked corresponds to the DT. of C.O.
NonRes.S.
rcs
dr
0.75
nrs
0 1 9
DT of s: Non-Res. Stud.
→ v3 invoked by using dr
eecs2030
Course
title 2030
fee 100
s
DT of s: Res. Stud.
v2 invoked by using pr.
Res.S.
rcs
pr
1.25
rs
0 1 9

Lecture 5

Part G

Inheritance -
Type Hierarchy Formed by Inheritance

Multi-Level Inheritance Hierarchy: Students

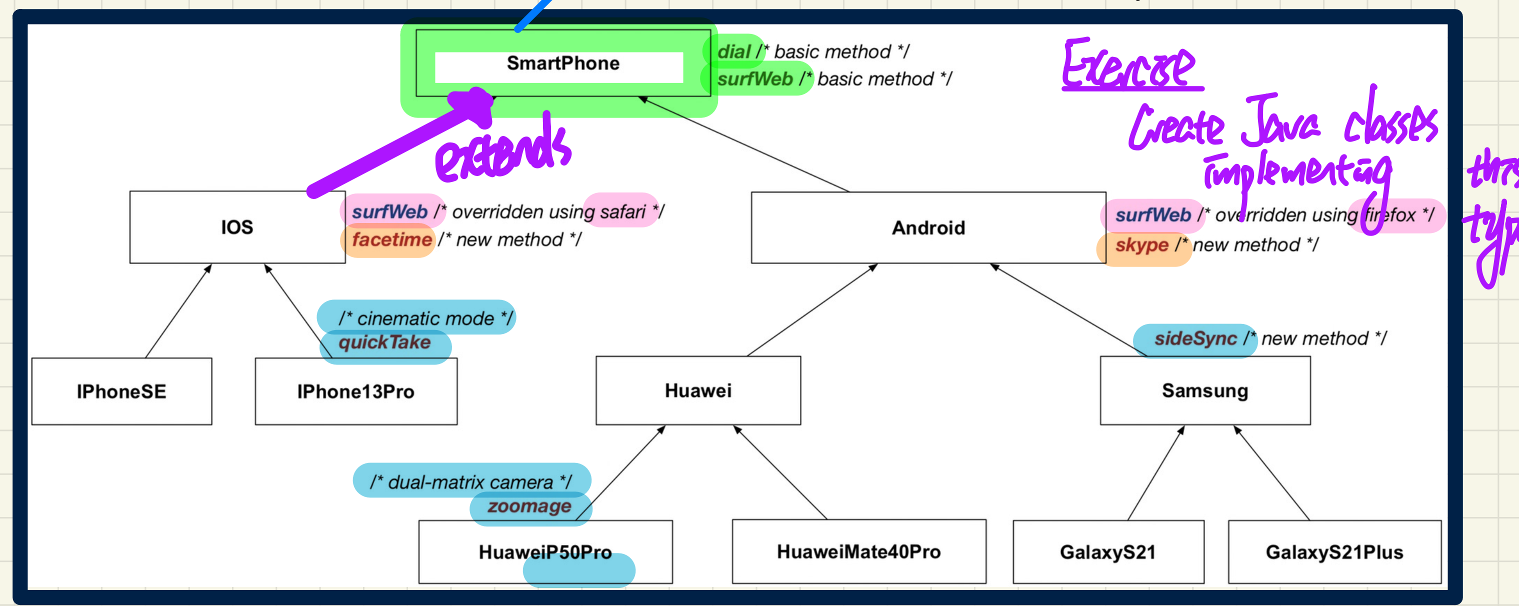


Reflections:

- 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?

$a > b \land b > c \Rightarrow a > c$

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?

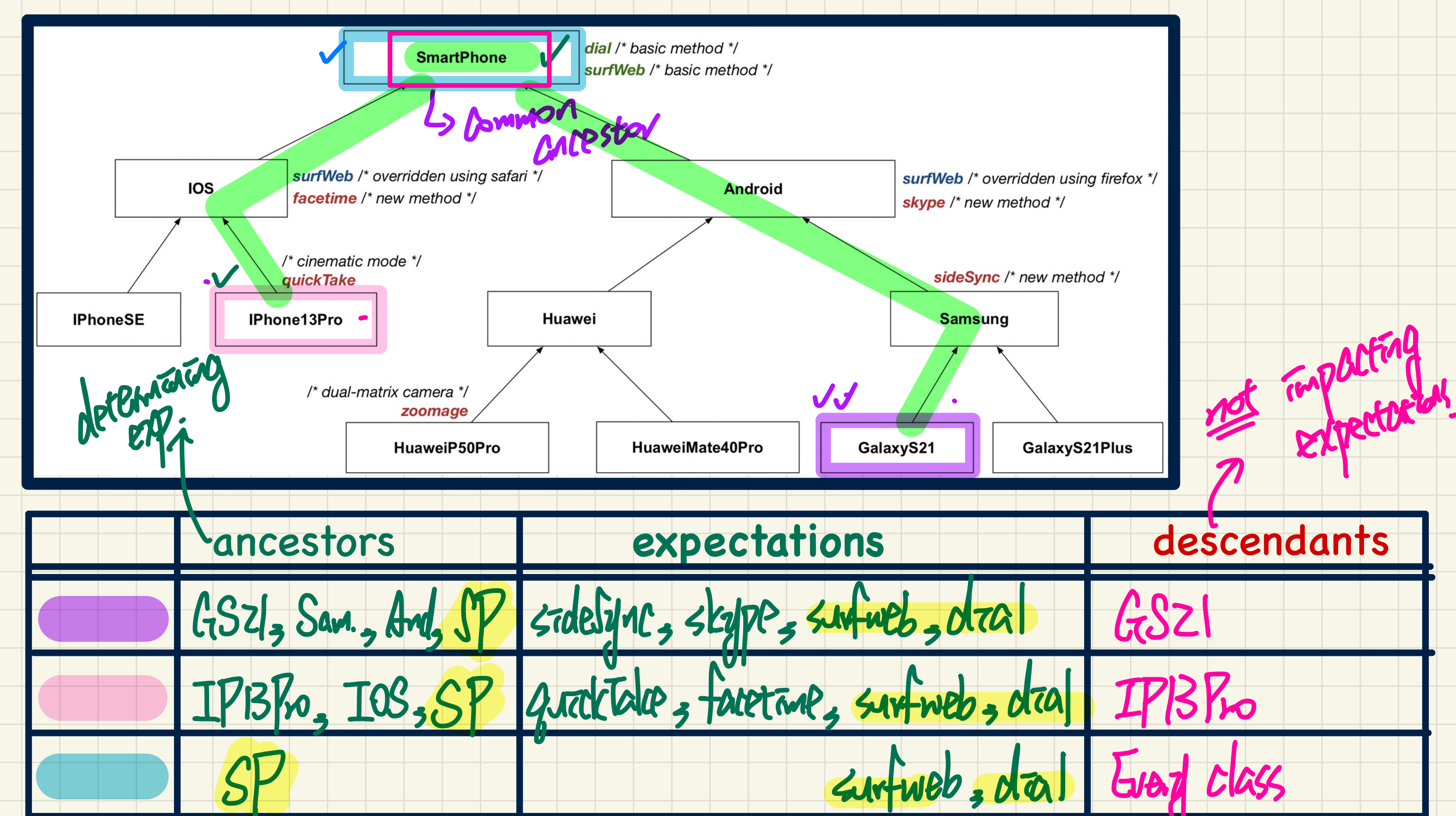
IPhone13Pro myPhone ;

declared type
(static)

↳ at runtime , myPhone may store the address of some IPhone13Pro-"Compatible" object.

Inheritance Forms a Type Hierarchy

Inheritance Accumulates Code for Reuse

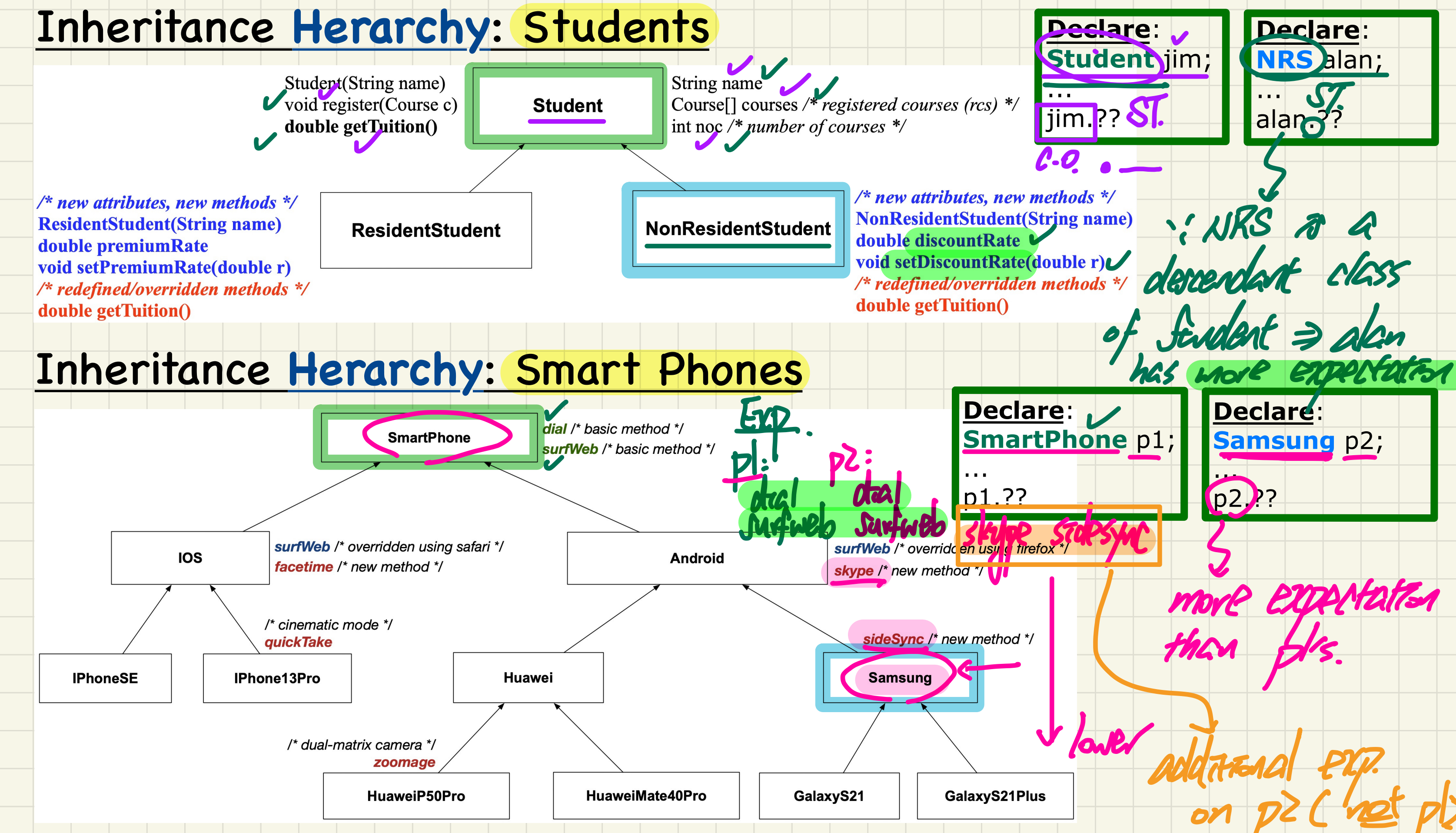


Lecture 5

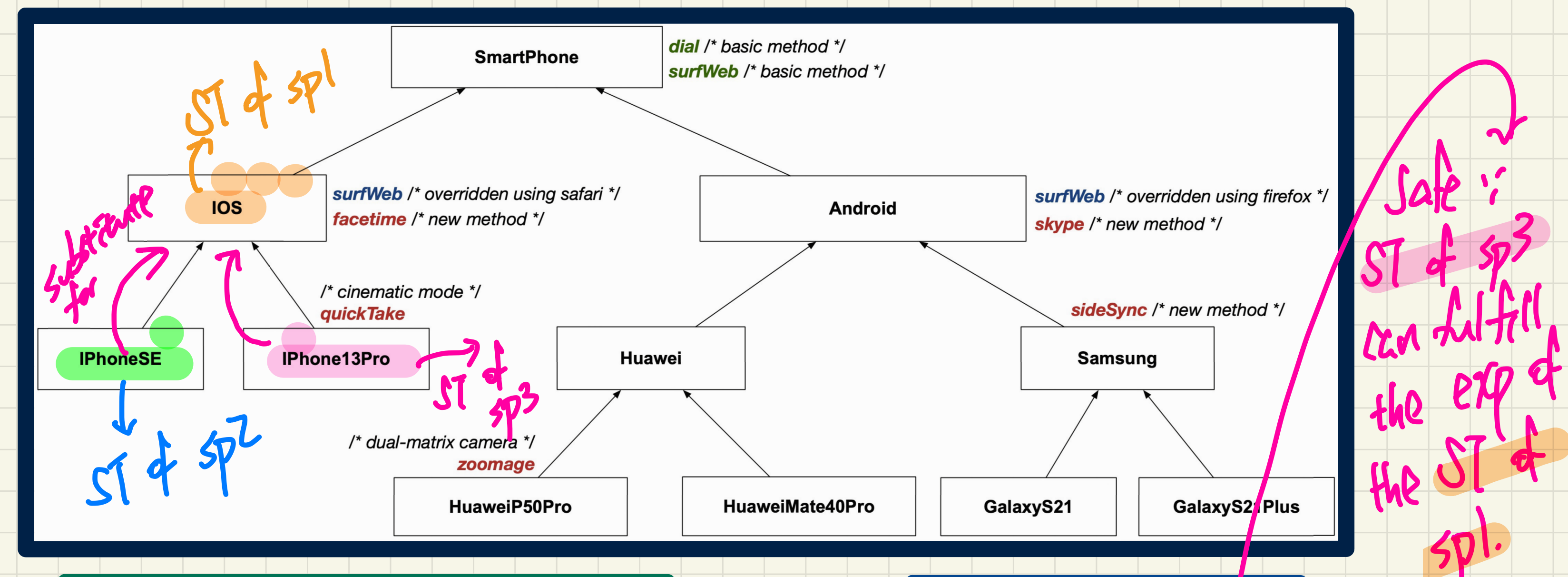
Part H

Inheritance -
Static Types and Rules of Substitutions

Static Types determine Expectations



Rules of Substitutions (1)

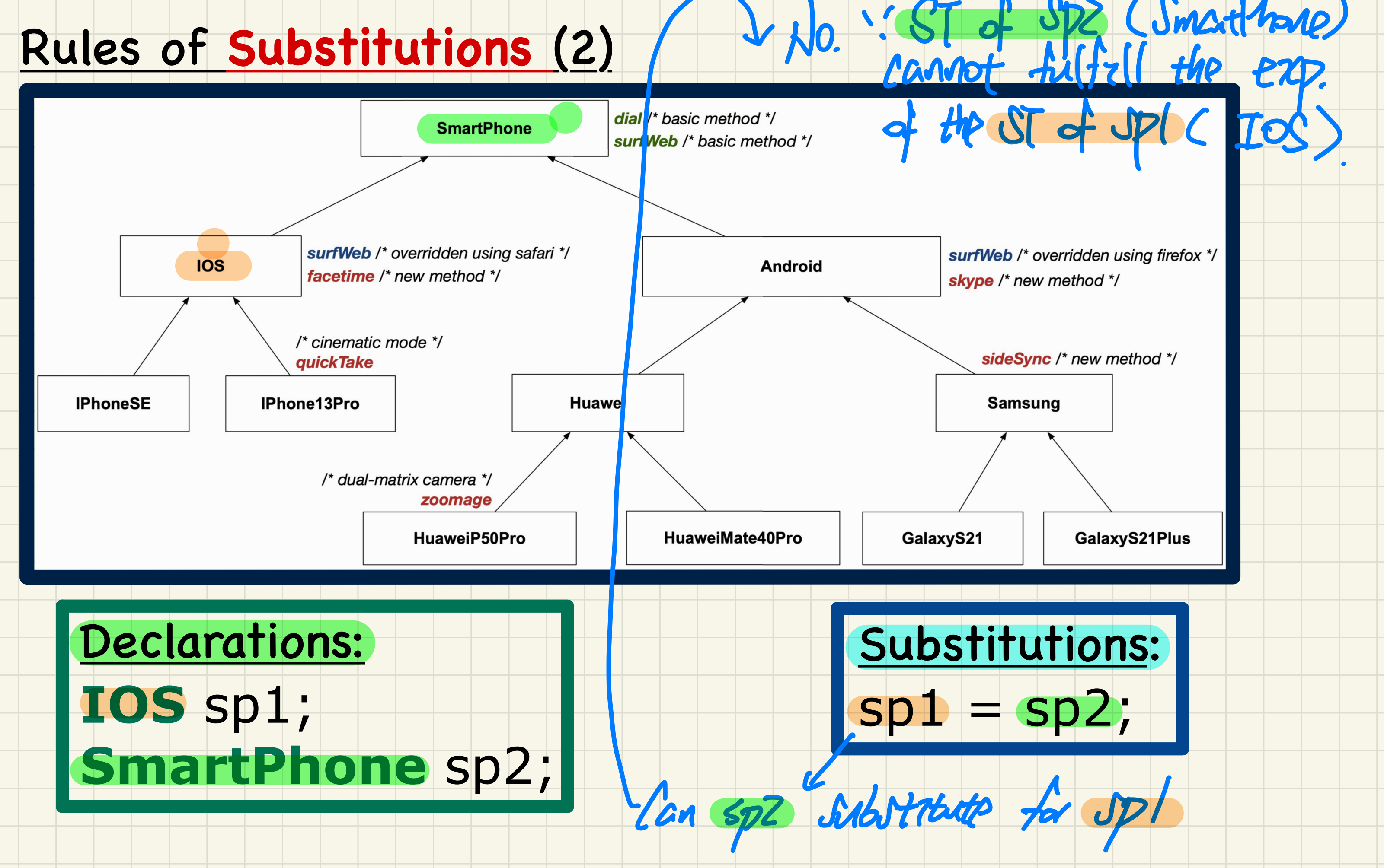


Declarations:

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

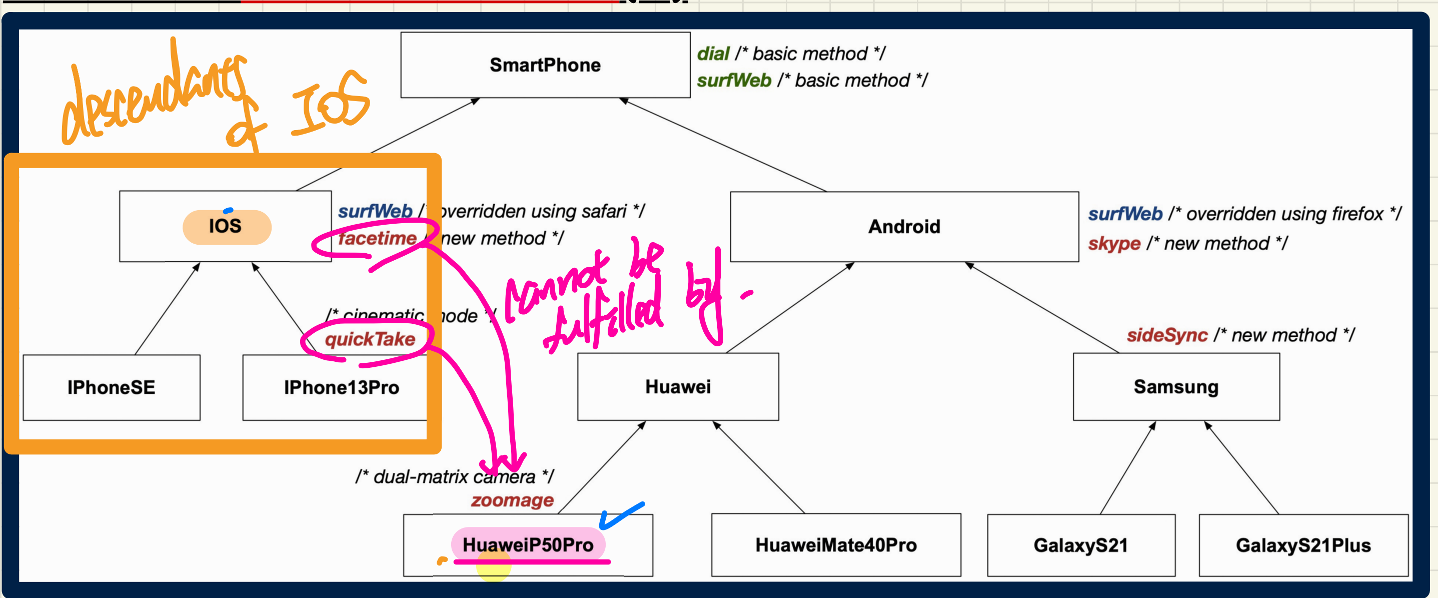
Substitutions:

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



Rules of Substitutions (2)
SmartPhone
dial /* basic method */
surfWeb /* basic method */
IOS
surfWeb /* overridden using safari */
facetime /* new method */
Android
surfWeb /* overridden using firefox */
skype /* new method */
/* cinematic mode */
quickTake
IPhoneSE
IPhone13Pro
Huawei
sideSync /* new method */
Samsung
/* dual-matrix camera */
zoomage
HuaweiP50Pro
HuaweiMate40Pro
GalaxyS21
GalaxyS21Plus
Declarations:
IOS sp1;
SmartPhone sp2;
Substitutions:
sp1 = sp2;
Can sp2 substitute for sp1
No. ∵ ST of sp2 (SmartPhone) cannot fulfill the exp. of the ST of sp1 (IOS).

Rules of Substitutions (3)



Declarations:

IOS sp1;

HuaweiP50Pro sp2;

Substitutions:

sp1 = sp2;

Can HuaweiP50Pro fulfill exp. of IOS?

No. ∵ HP50Pro is not a descendant of IOS.

Substitutions

safe vs. **unsafe**

compiles — fails to compile

v1 = v2

Can v2 substitute v1?

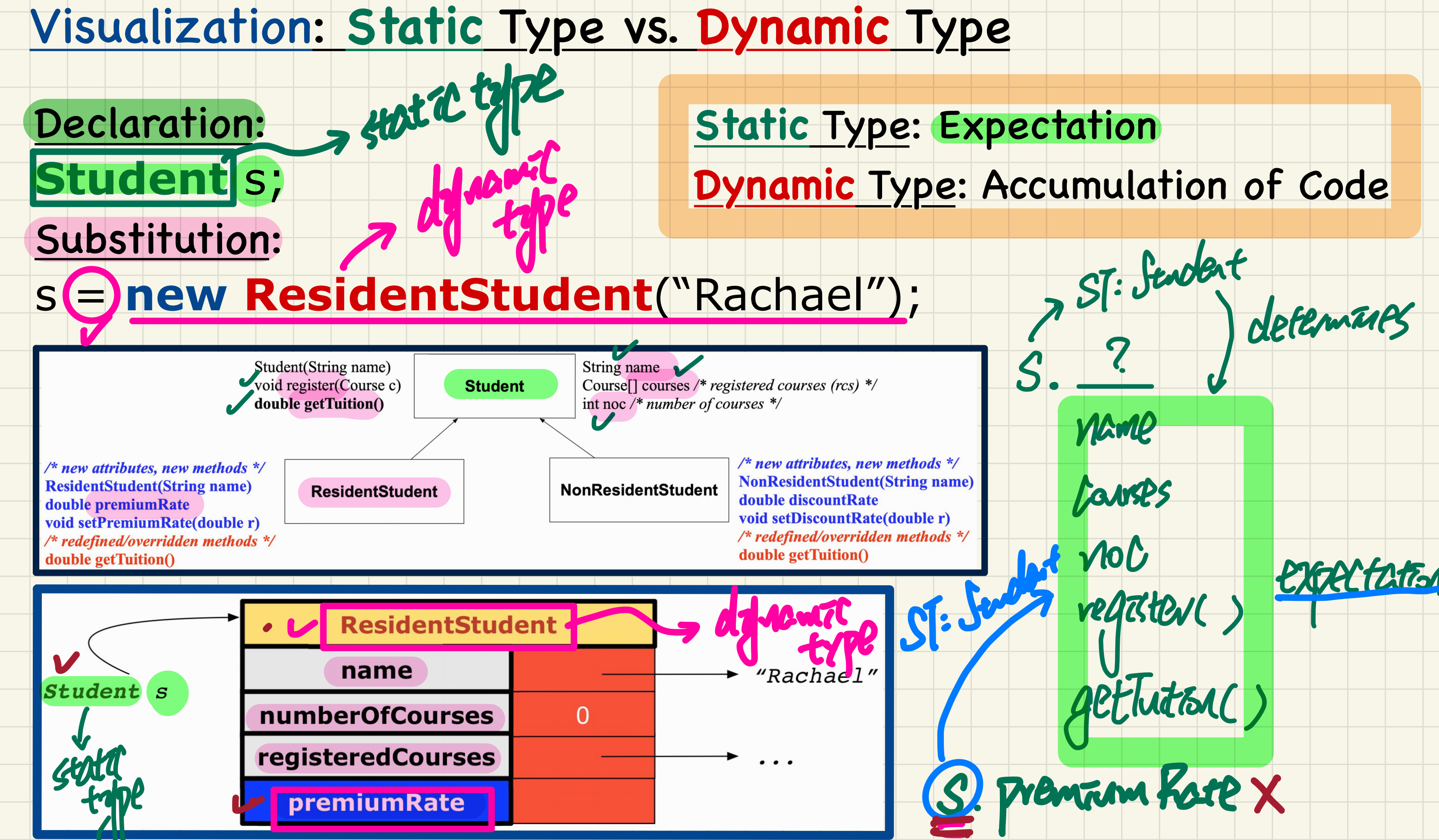
Can the ST of v2 fulfill exp. of v1's ST?

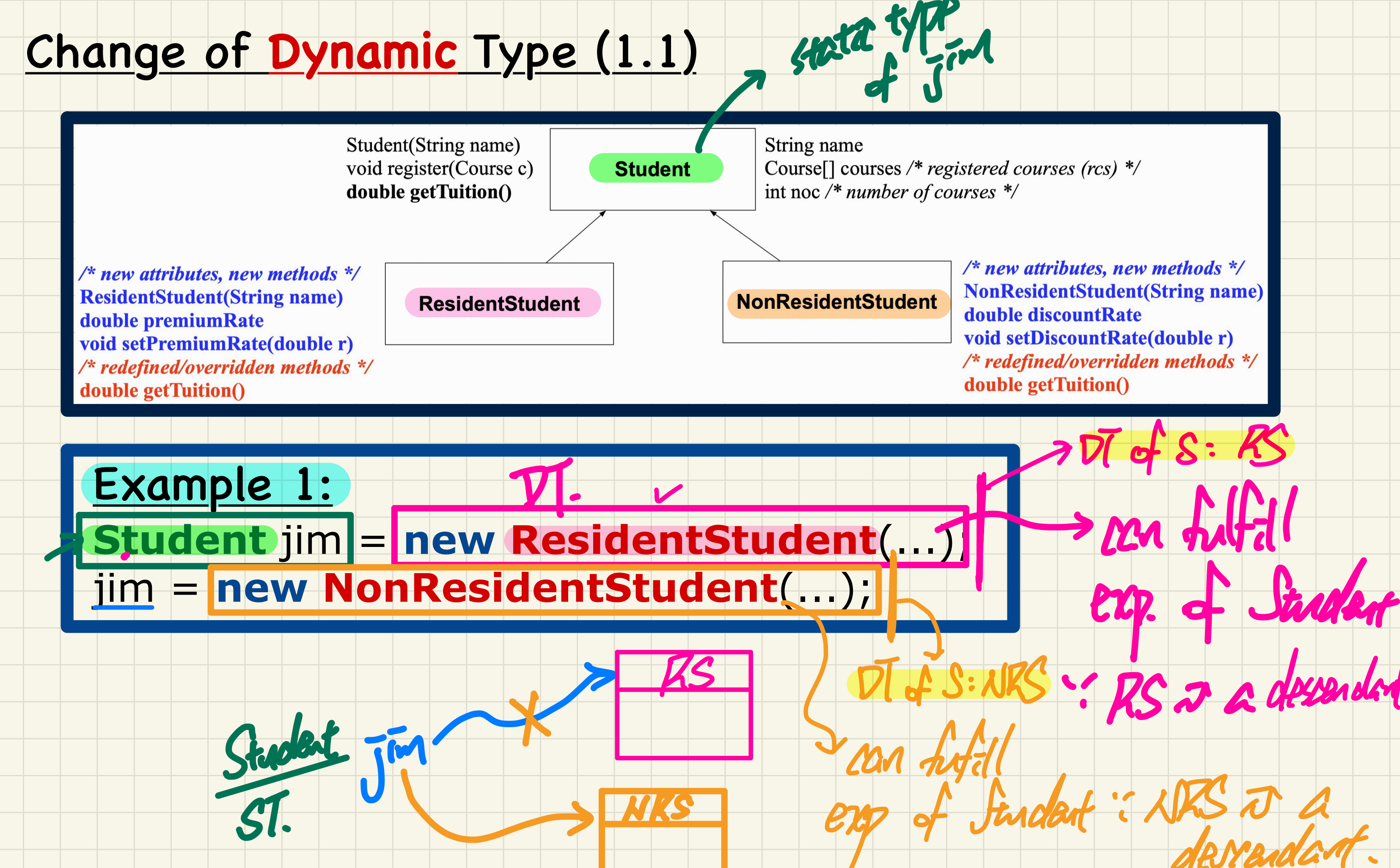
Yes if v2's ST is a descendant of v1's ST.

Lecture 5

Part I

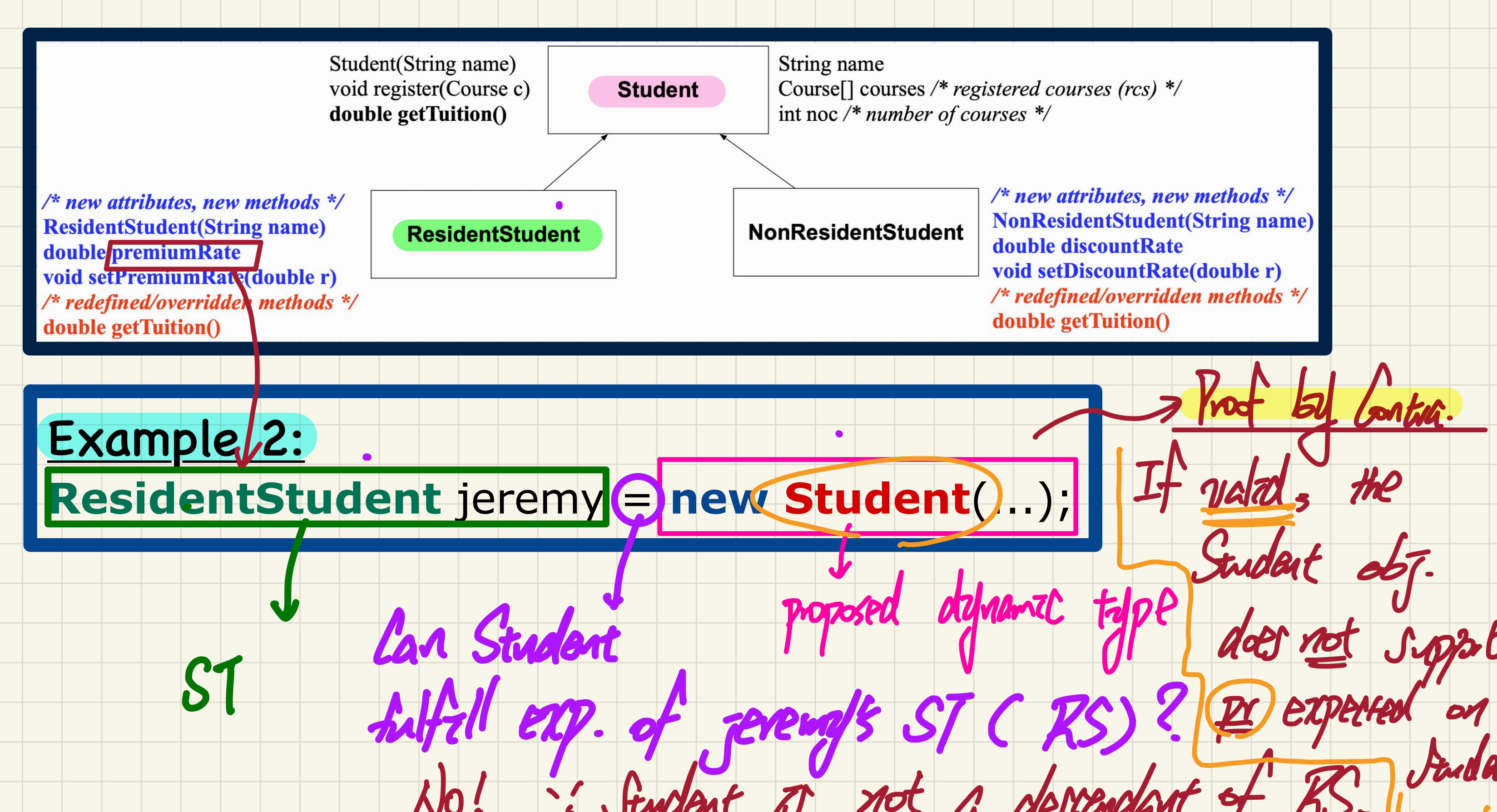
Inheritance -
Dynamic Types,
Polymorphism, Dynamic Binding


Visualization: Static Type vs. Dynamic Type
Declaration:
Student s;
Substitution:
s = new ResidentStudent("Rachael");
Static Type: Expectation
Dynamic Type: Accumulation of Code
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
Student s
ResidentStudent
name
"Rachael"
numberOfCourses
0
registeredCourses
...
premiumRate



Change of Dynamic Type (1.1)
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
Example 1:
Student jim = new ResidentStudent(...);
jim = new NonResidentStudent(...);

Change of Dynamic Type (1.2)



Change of Dynamic Type: Exercise (1)

SmartPhone
dial /* basic method */
surfWeb /* basic method */

IOS
surfWeb /* overridden using safari */
facetime /* new method */

Android
surfWeb /* overridden using firefox */
skype /* new method */

IPhoneSE

IPhone13Pro
/* cinematic mode */
quickTake

Huawei

Samsung
sideSync /* new method */

HuaweiP50Pro
/* dual-matrix camera */
zoomage

HuaweiMate40Pro

GalaxyS21

GalaxyS21Plus

Change of Dynamic Type: Exercise (2)

SmartPhone
dial /* basic method */
surfWeb /* basic method */

IOS
surfWeb /* overridden using safari */
facetime /* new method */

Android
surfWeb /* overridden using firefox */
skype /* new method */

IPhoneSE

IPhone13Pro
/* cinematic mode */
quickTake

Huawei

Samsung
sideSync /* new method */

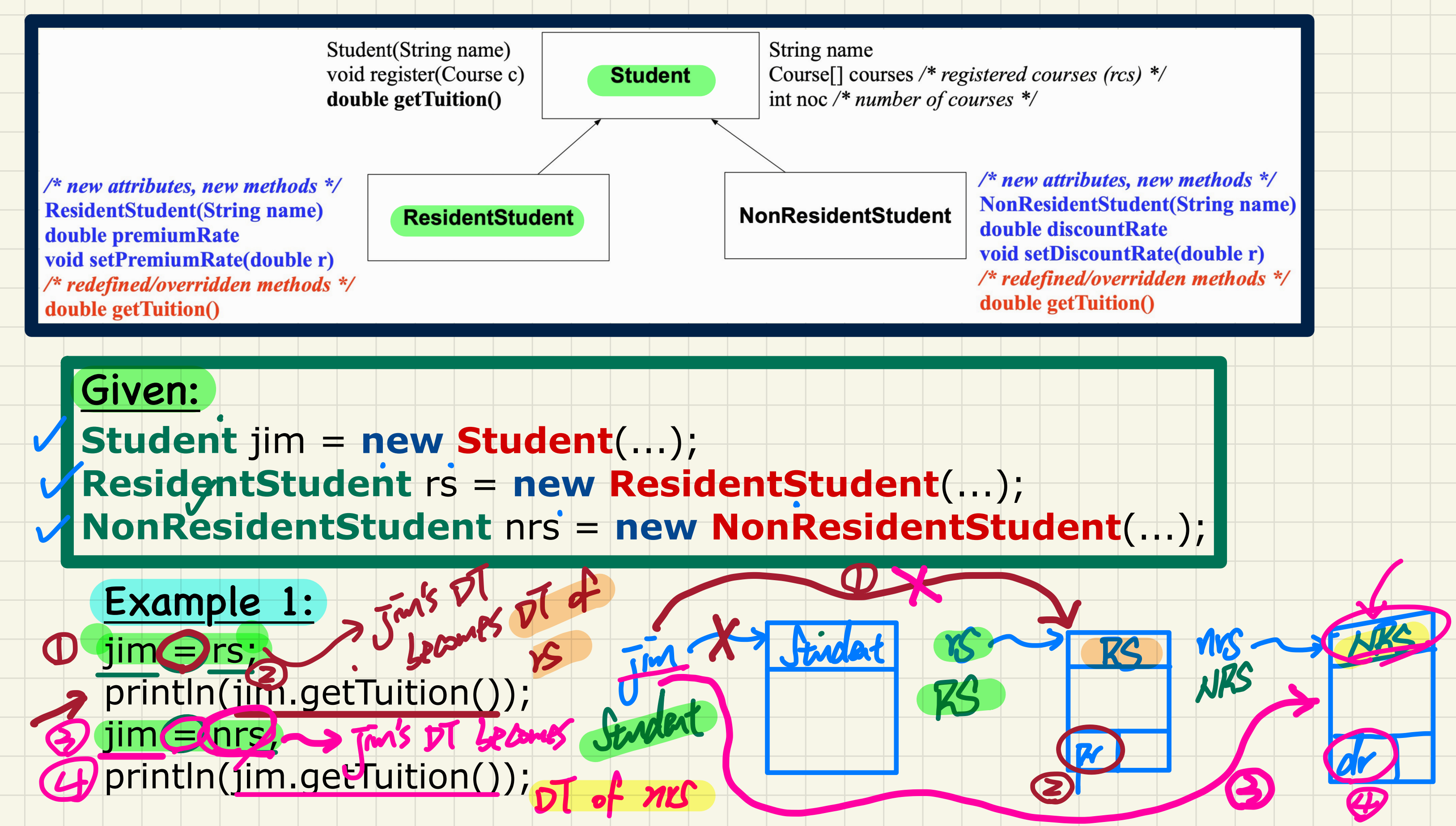
HuaweiP50Pro
/* dual-matrix camera */
zoomage

HuaweiMate40Pro

GalaxyS21

GalaxyS21Plus

Change of Dynamic Type (2.1)



Change of Dynamic Type (2.2)

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

Student

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

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

ResidentStudent

NonResidentStudent

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

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());
```

invalid ⇒ fail to compile!

ST: Student

ST: RS

Can jim's ST fulfill exp. of the ST of rs?

v1 = v2 ✓ to be valid: ST2 must be a descendant of ST1 (so that it can fulfill the exp of ST1)

v1 → ST1

v2 → ST2

Polymorphism: allowable dynamic types (descendants)

→ dynamic binding

---

v1.m(...)

↳ each descendant class of ST1 may have its own version

which version to invoke depends on DT of v1.

Lecture 5

Part J

Inheritance -
Type Casting: Motivation, Syntax, Rules

Type Cast: Motivation

But, given that jim's DT is Res. Stud., why can't the compiler know about this and thus allow rs to point to that object?

Student

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

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

ResidentStudent

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

NonResidentStudent

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

undecidable problem.

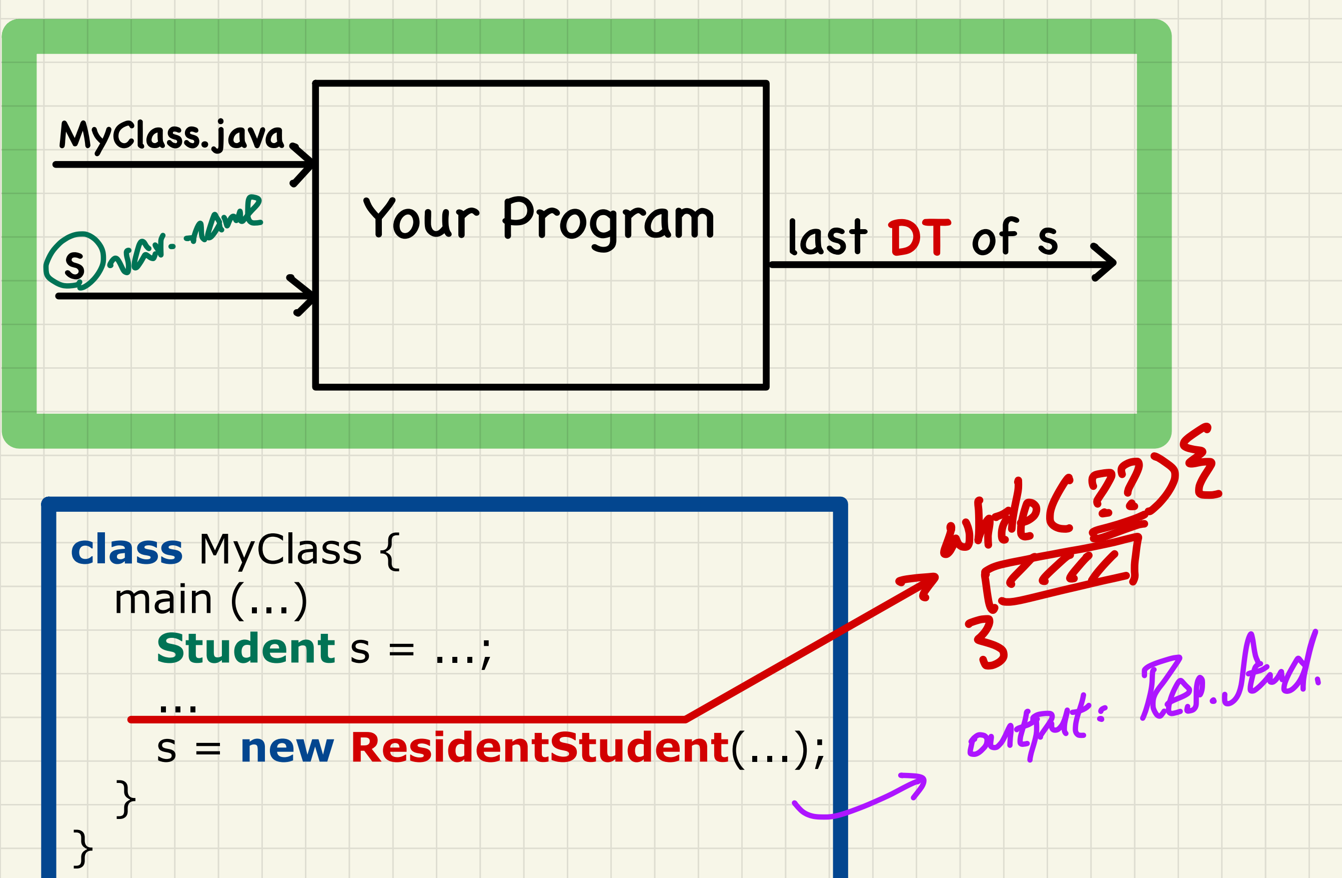
```
1  Student jim = new ResidentStudent("J. Davis");
2  ResidentStudent rs = jim;
3  rs.setPremiumRate(1.5);
```

∵ ST of jim (Student) cannot fulfill exp. of ST of rs (Res. Stud.)

Student jim → Res. Stud.

Res. Stud. rs - - - → Res. Stud.

An A+ Challenge: Inferring the DT of a Variable

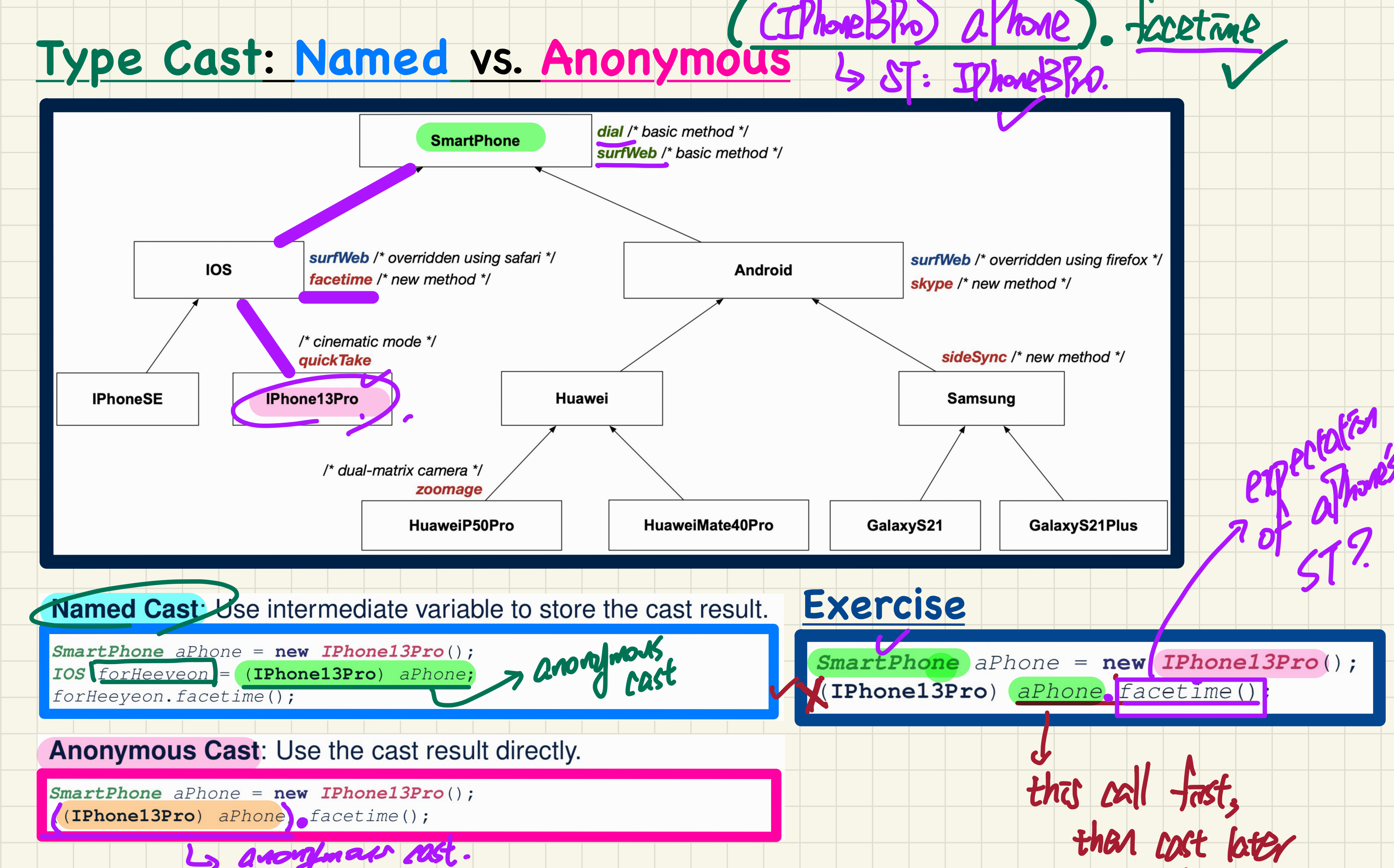


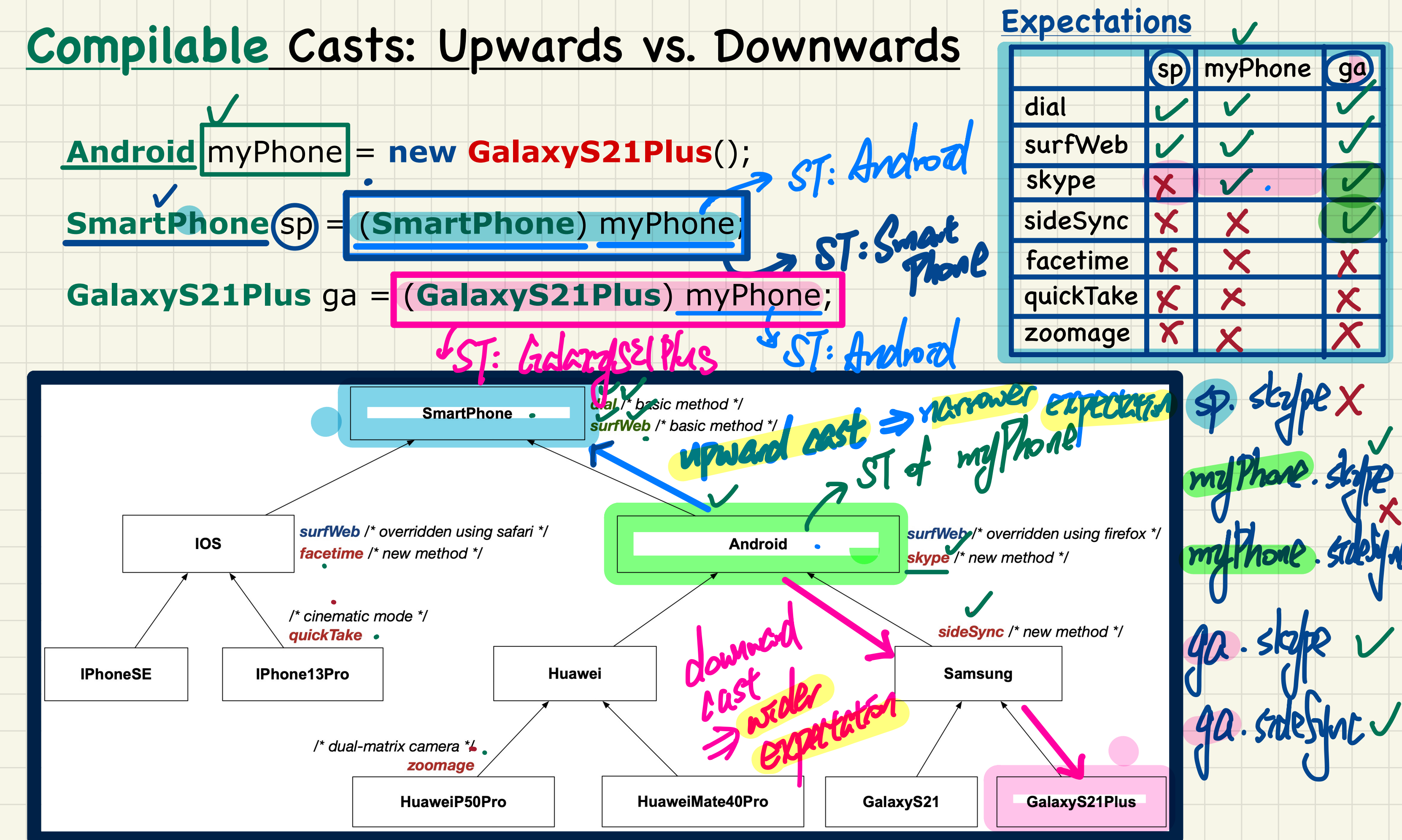
Anatomy of a Type Cast

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

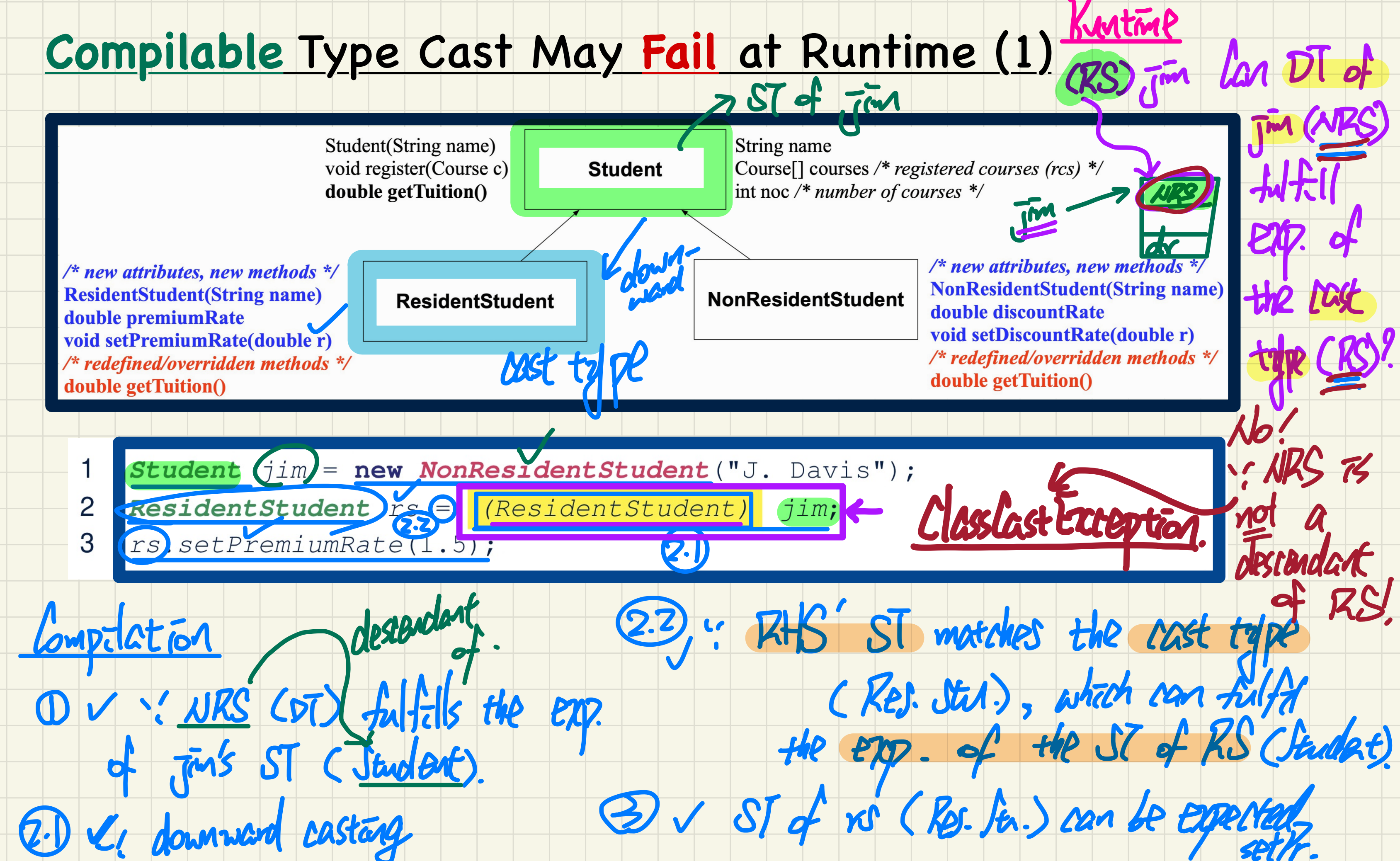
ResidentStudent

pr

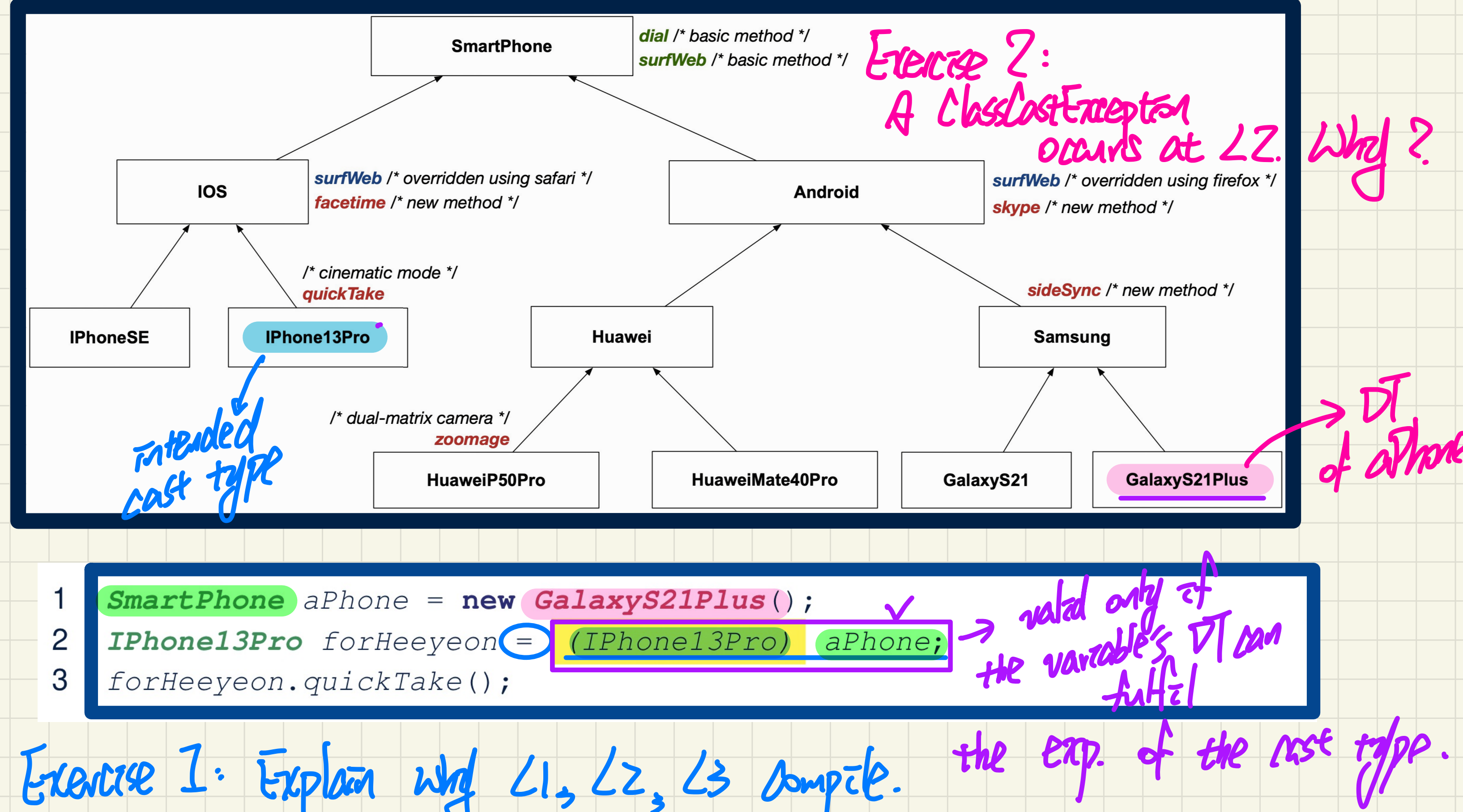

Type Cast: Named vs. Anonymous
SmartPhone
dial /* basic method */
surfWeb /* basic method */
IOS
surfWeb /* overridden using safari */
facetime /* new method */
Android
surfWeb /* overridden using firefox */
skype /* new method */
/* cinematic mode */
quickTake
sideSync /* new method */
IPhoneSE
IPhone13Pro
Huawei
Samsung
/* dual-matrix camera */
zoomage
HuaweiP50Pro
HuaweiMate40Pro
GalaxyS21
GalaxyS21Plus
Named Cast: Use intermediate variable to store the cast result.
SmartPhone aPhone = new IPhone13Pro();
IOS forHeeyeon = (IPhone13Pro) aPhone;
forHeeyeon.facetime();
Anonymous Cast: Use the cast result directly.
SmartPhone aPhone = new IPhone13Pro();
((IPhone13Pro) aPhone).facetime();
Exercise
SmartPhone aPhone = new IPhone13Pro();
(IPhone13Pro) aPhone.facetime();
((IPhone13Pro) aPhone).facetime
ST: IPhone13Pro.
anonymous cast
expectation of aPhone's ST?
this call first, then cast later



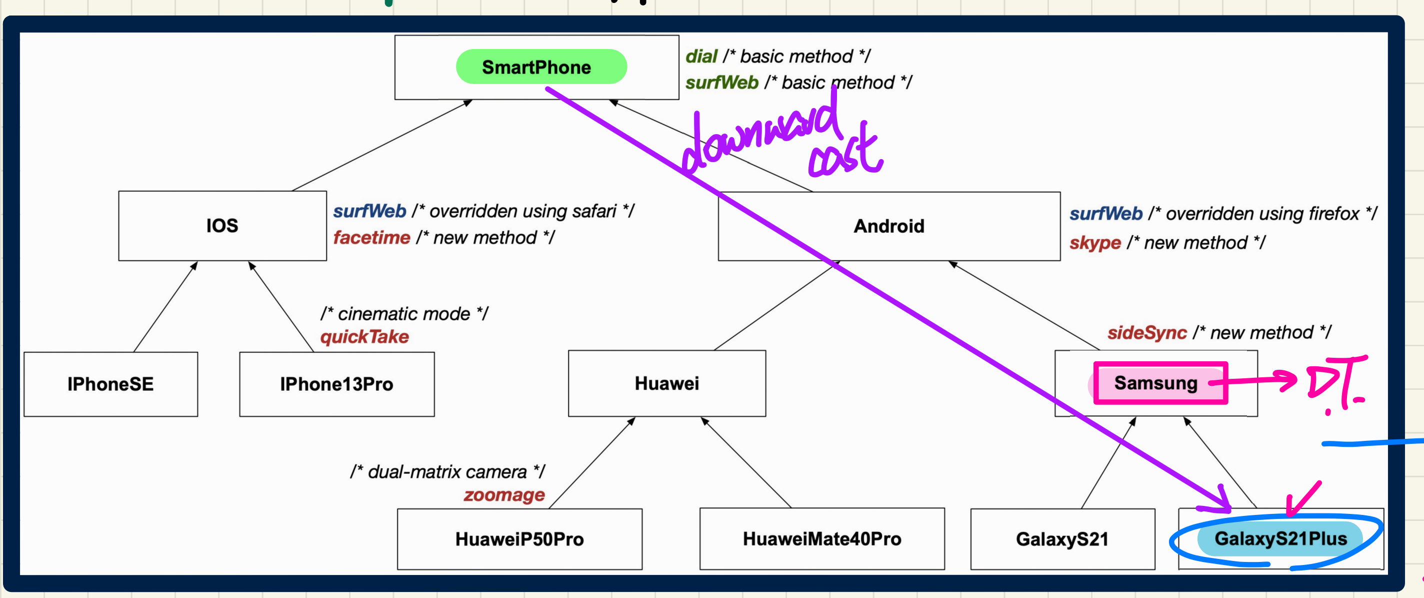
Compilable Casts: Upwards vs. Downwards
Android myPhone = new GalaxyS21Plus();
SmartPhone sp = (SmartPhone) myPhone;
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
ST: Android
ST: SmartPhone
ST: GalaxyS21Plus
ST: Android
Expectations
sp
myPhone
ga
dial
surfWeb
skype
sideSync
facetime
quickTake
zoomage
SmartPhone
dial /* basic method */
surfWeb /* basic method */
upward cast ⇒ narrower expectation
ST of myPhone
IOS
surfWeb /* overridden using safari */
facetime /* new method */
Android
surfWeb /* overridden using firefox */
skype /* new method */
/* cinematic mode */
quickTake
sideSync /* new method */
IPhoneSE
IPhone13Pro
Huawei
Samsung
downward cast ⇒ wider expectation
/* dual-matrix camera */
zoomage
HuaweiP50Pro
HuaweiMate40Pro
GalaxyS21
GalaxyS21Plus
sp. skype
myPhone. skype
myPhone. sideSync
ga. skype
ga. sideSync


Compilable Type Cast May Fail at Runtime (1)
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
1 Student jim = new NonResidentStudent("J. Davis");
2 ResidentStudent rs = (ResidentStudent) jim;
3 rs.setPremiumRate(1.5);
ClassCastException
Compilation
downward casting

Compilable Type Cast May Fail at Runtime (2)



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



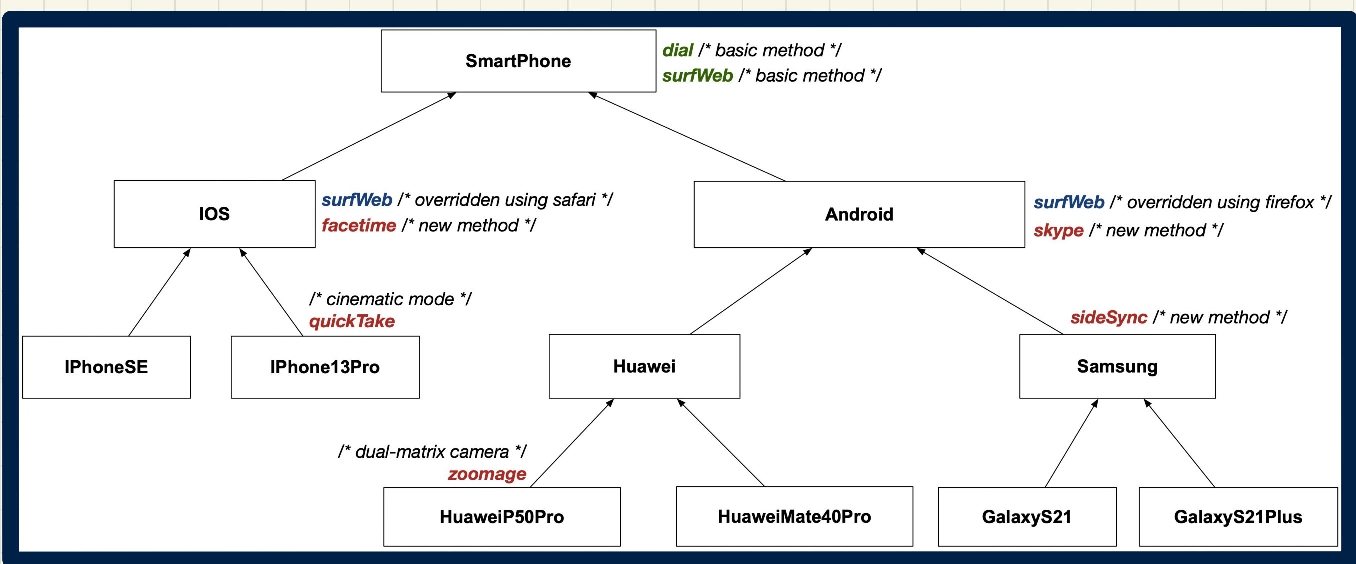

```
SmartPhone myPhone = new Samsung();
/* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
```

↳ downward cast

Compilable? ClassCastException at runtime?

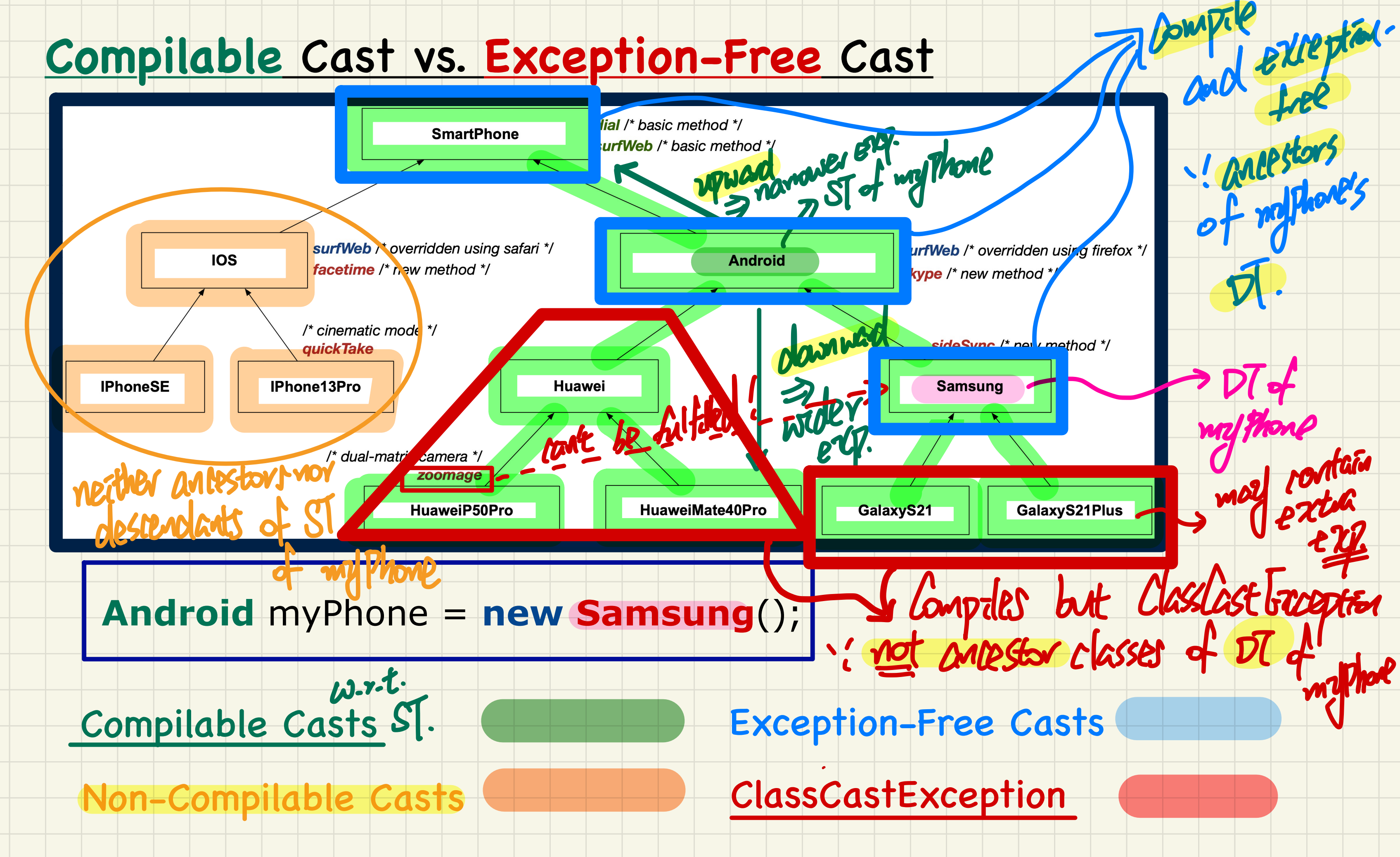
Runtime: Can the DT of myPhone (Samsung) fulfill exp. of cast type (GS21P)

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
SmartPhone
dial /* basic method */
surfWeb /* basic method */
IOS
surfWeb /* overridden using safari */
facetime /* new method */
Android
surfWeb /* overridden using firefox */
skype /* new method */
IPhoneSE
IPhone13Pro
/* cinematic mode */
quickTake
Huawei
Samsung
sideSync /* new method */
/* dual-matrix camera */
zoomage
HuaweiP50Pro
HuaweiMate40Pro
GalaxyS21
GalaxyS21Plus
Android myPhone = new Samsung();
Compilable Casts
w.r.t. ST.
Exception-Free Casts
Non-Compilable Casts
ClassCastException
upward ⇒ narrower exp.
⇒ ST of myPhone
downward ⇒ wider exp.
compile and exception-free
∵ ancestors of myPhone's DT.
DT of myPhone
may contain extra exp.
Compiles but ClassCastException
∵ not ancestor classes of DT of myPhone
can't be fulfilled!
neither ancestors nor descendants of ST of myPhone

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;
```

Part 1

A
B
C
D

Part 2: Compilation (error at L2)

(L1) ✓ DT C can fulfill the exp. of ST B (∵ C is a descendant of B)

(L2) X ∵ cast type D is neither an ancestor nor a descendant of b's ST (B)

valid if it's either upward or downward cast w.r.t. ST B

Part 3: Alternative to L2

D d = (D) ((A) b) ✓ valid (compilable?)

e.g. Floor f = (Floor) ((Object) v)

Part 4: ClassCastException ∵ b's DT (C) cannot fulfill exp. of cast type (D)

cast type is not an ancestor of b's DT.

Lecture 5

Part K

Inheritance -
Checking Dynamic Type via instanceof

Checking Dynamic Types at Runtime (1)

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

Student

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

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

ResidentStudent

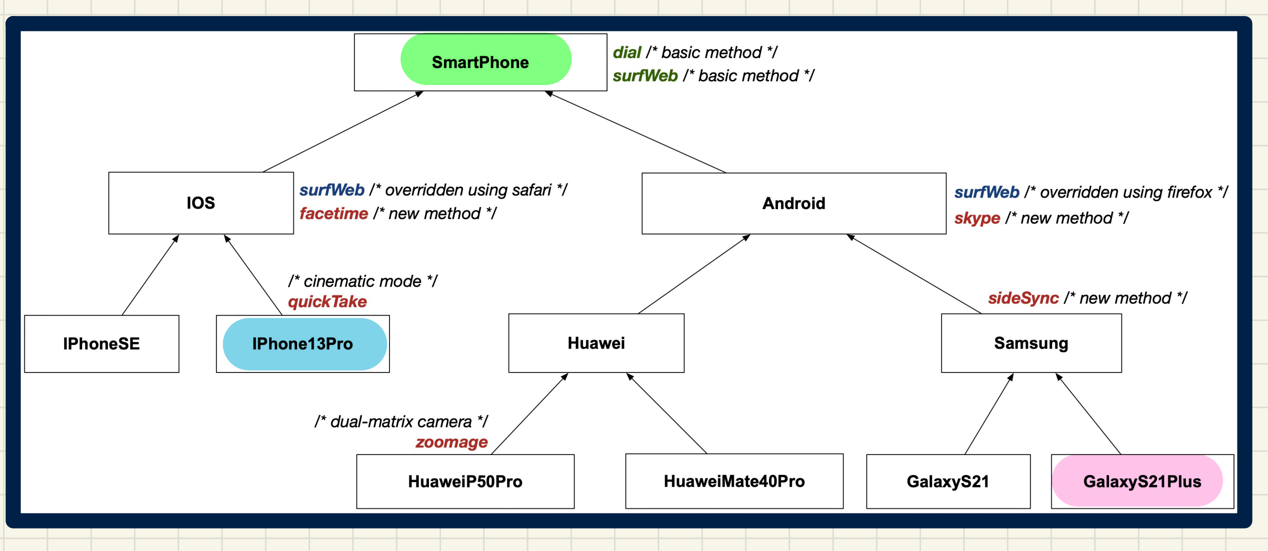
NonResidentStudent

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

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

ST
DT
✓3. Is ResidentStudent* an ancestor of DT of jim?
False ∵ RS is not an ancestor of jim's DT
1. Can DT of jim fulfill the expectation of ResidentStudent*?
2. Is DT of jim a descendant of ResidentStudent*?
expression (dot notation) denoting an object
checked in the if branch
ClassCastException if executed
a class name

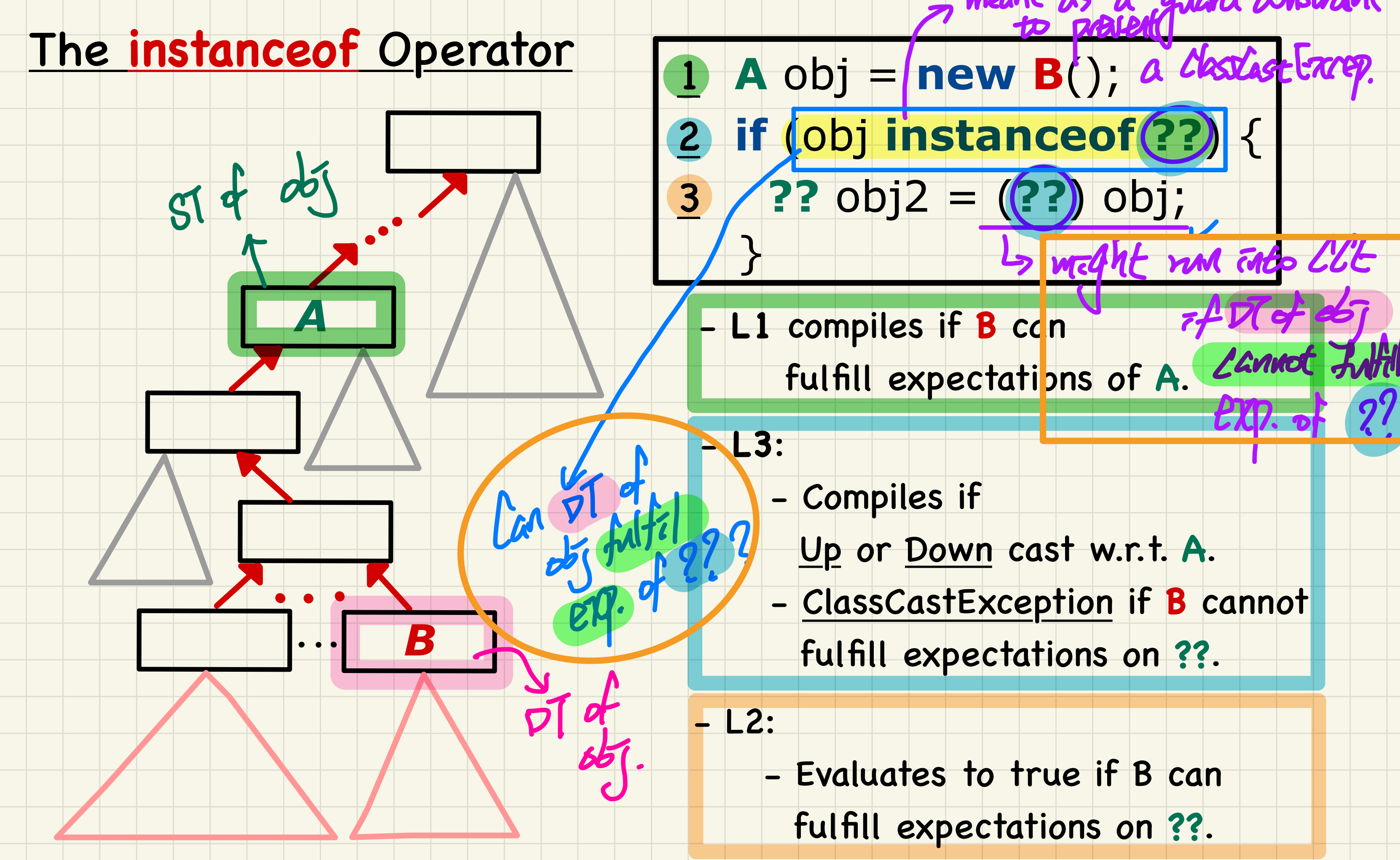
Checking Dynamic Types at Runtime (2)



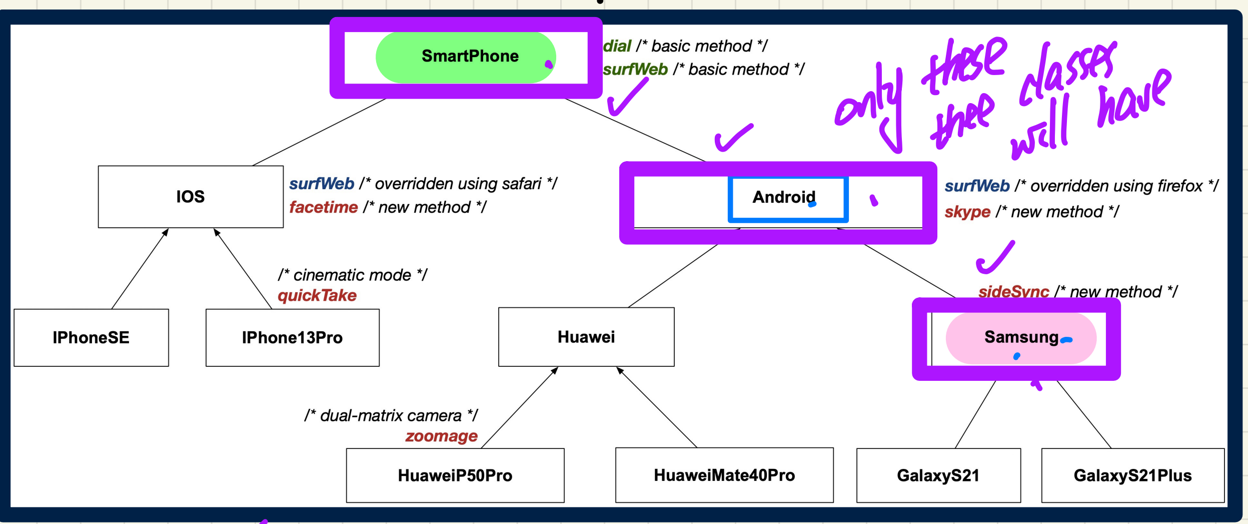

```
1  SmartPhone aPhone = new GalaxyS21Plus();
2  if (aPhone instanceof IPhone13Pro) {
3    IOS forHeeyeon = (IPhone13Pro) aPhone;
4    forHeeyeon.facetime();
5  }
```

Can DT of aPhone fulfill expectations of IPhone13Pro*?

executing this line will cause a ClassCastException.


The instanceof Operator
1 A obj = new B();
2 if (obj instanceof ??) {
3 ?? obj2 = (??) obj;
}
- L1 compiles if B can fulfill expectations of A.
- L3:
- Compiles if Up or Down cast w.r.t. A.
- ClassCastException if B cannot fulfill expectations on ??.
- L2:
- Evaluates to true if B can fulfill expectations on ??.
A
B
ST of obj
DT of obj.
Can DT of obj fulfil exp. of ??
meant as a guard constraint to prevent a ClassCastExcep.
might run into CCE if DT of obj cannot fulfill exp. of ??

Use of the instanceof Operator



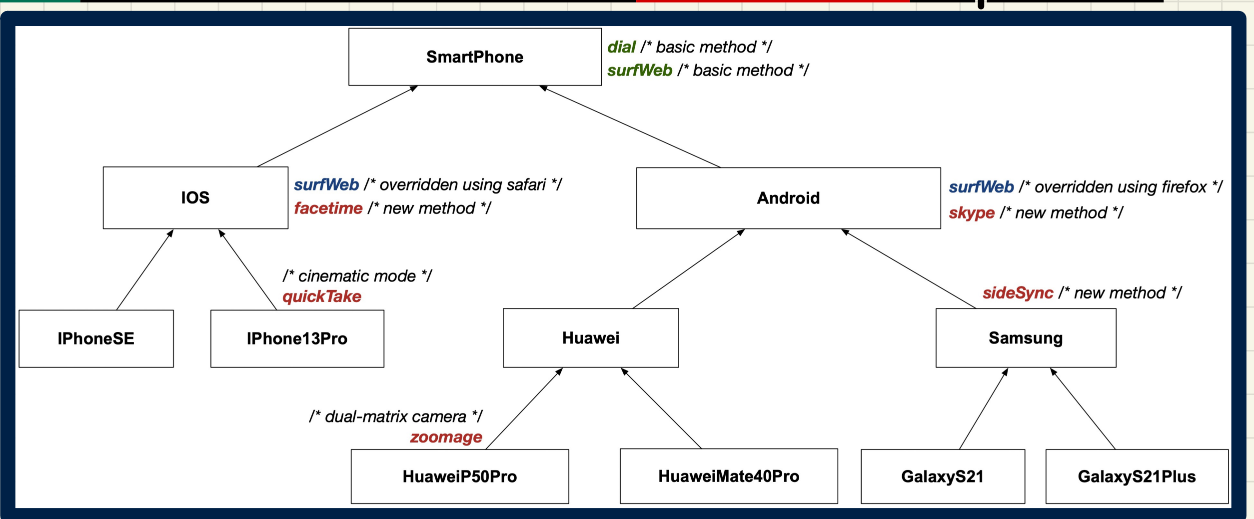
myPhone instanceof Samsung evaluate to true.

```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android);
/* true ∵ Samsung is a descendant of Android */
println(myPhone instanceof Samsung);
/* true ∵ Samsung is a descendant of Samsung */
println(myPhone instanceof GalaxyS21);
/* false ∵ Samsung is not a descendant of GalaxyS21 */
println(myPhone instanceof IOS);
/* false ∵ Samsung is not a descendant of IOS */
println(myPhone instanceof IPhone13Pro);
/* false ∵ Samsung is not a descendant of IPhone13Pro */
```

Can DT of myphone fulfill Android?

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

Safe Cast via Use of the instanceof Operator




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

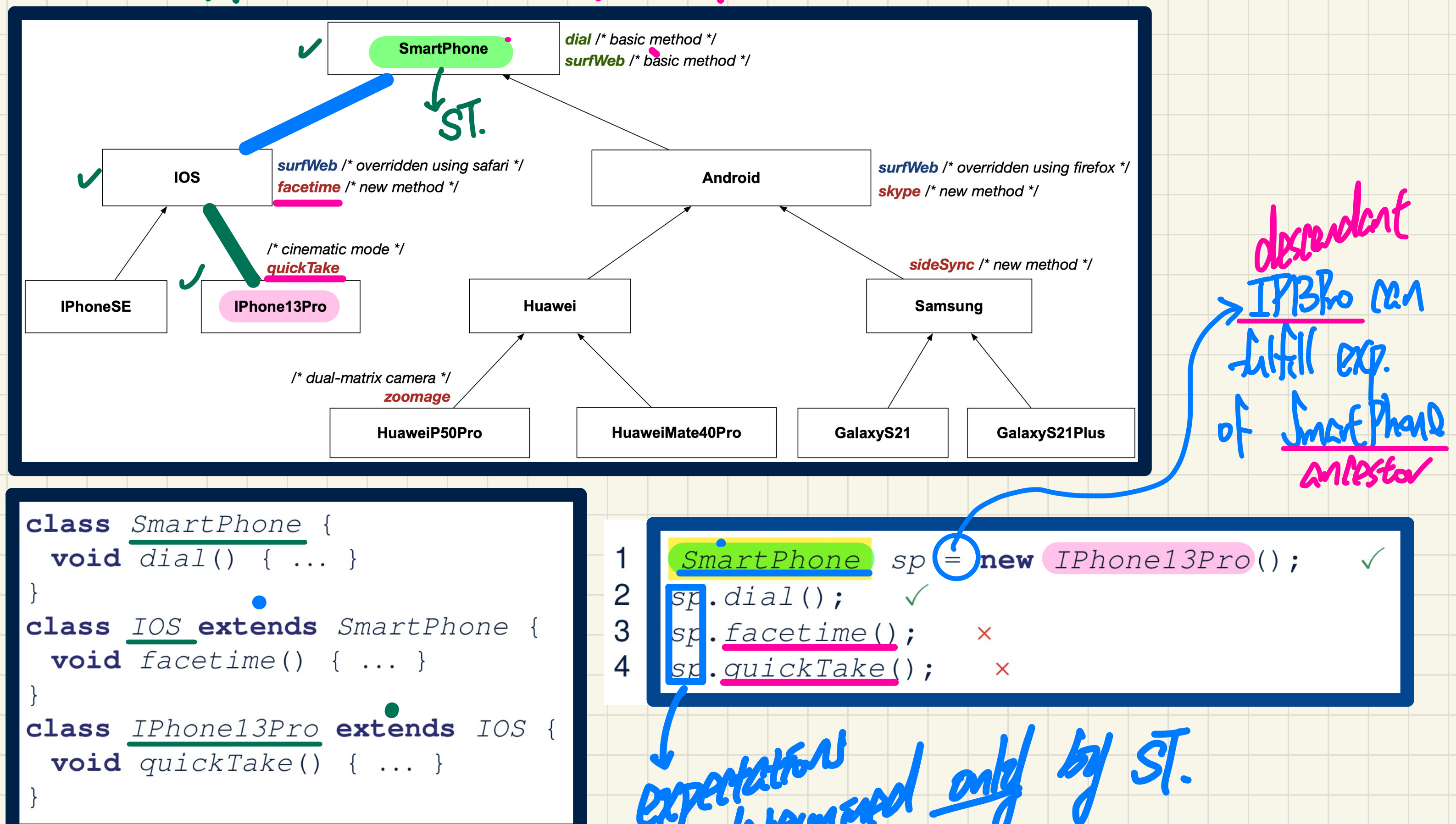
myPhone instanceof ??
evaluates to true if
Samsung can
fulfill expectations on ??.

Lecture 5

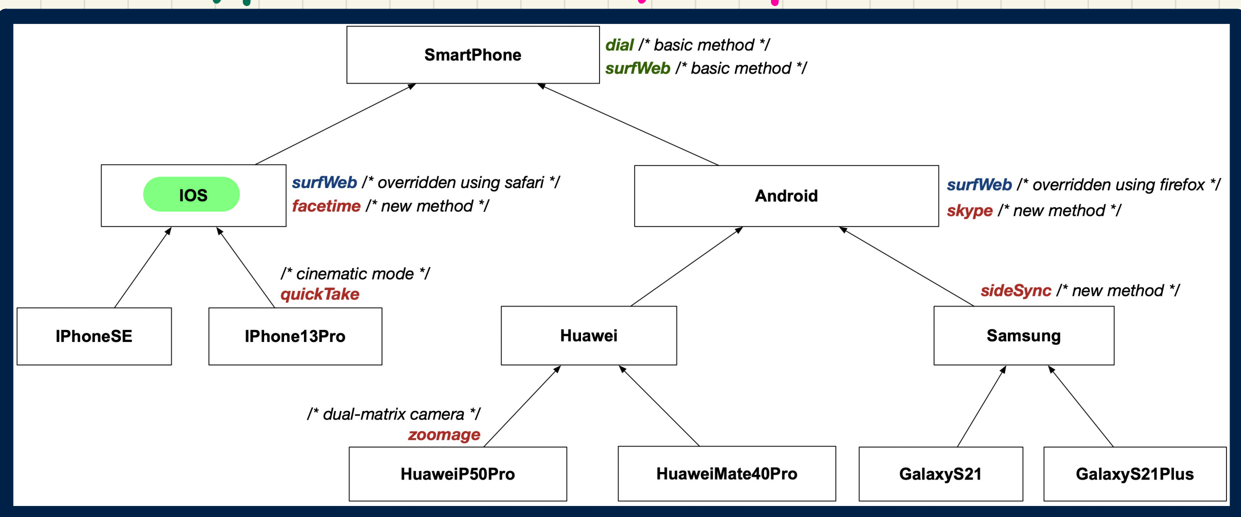
Part L

***Inheritance -
Static Types, Casts, Polymorphism***

Static Types, Casts, Polymorphism (1)



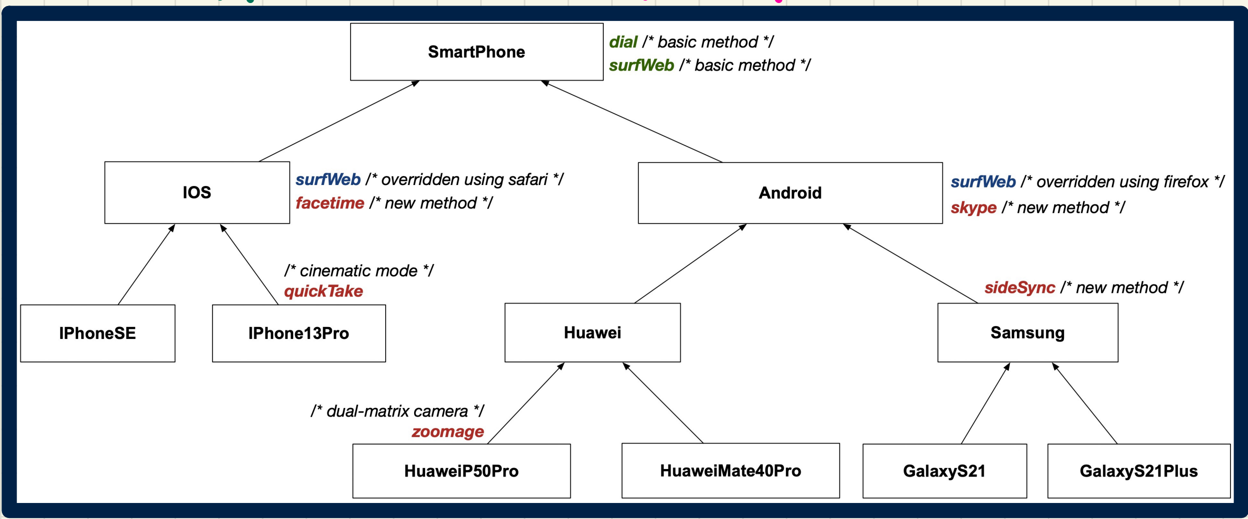
Static Types, Casts, Polymorphism (2)




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

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

Static Types, Casts, Polymorphism (3)




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

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

Static Types, Casts, Polymorphism (4)

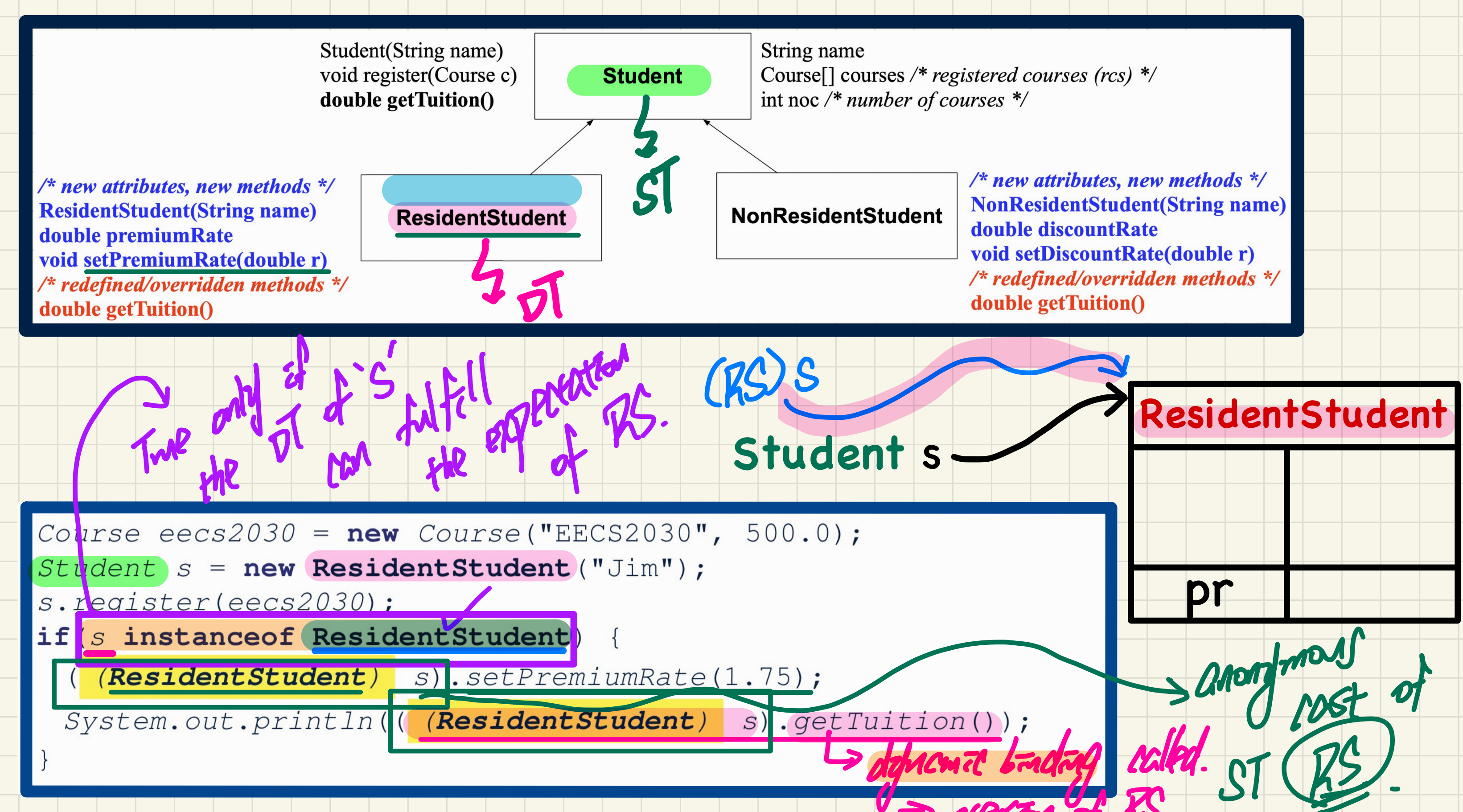

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

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

1. Compilation? w.r.t. ST
   Valid ∵ downward cast
2. ClassCastException?
   Can DT fulfill cast type? NO.

ST, DT, cast type, cast expression has ST matching cast type, ref. var being cast

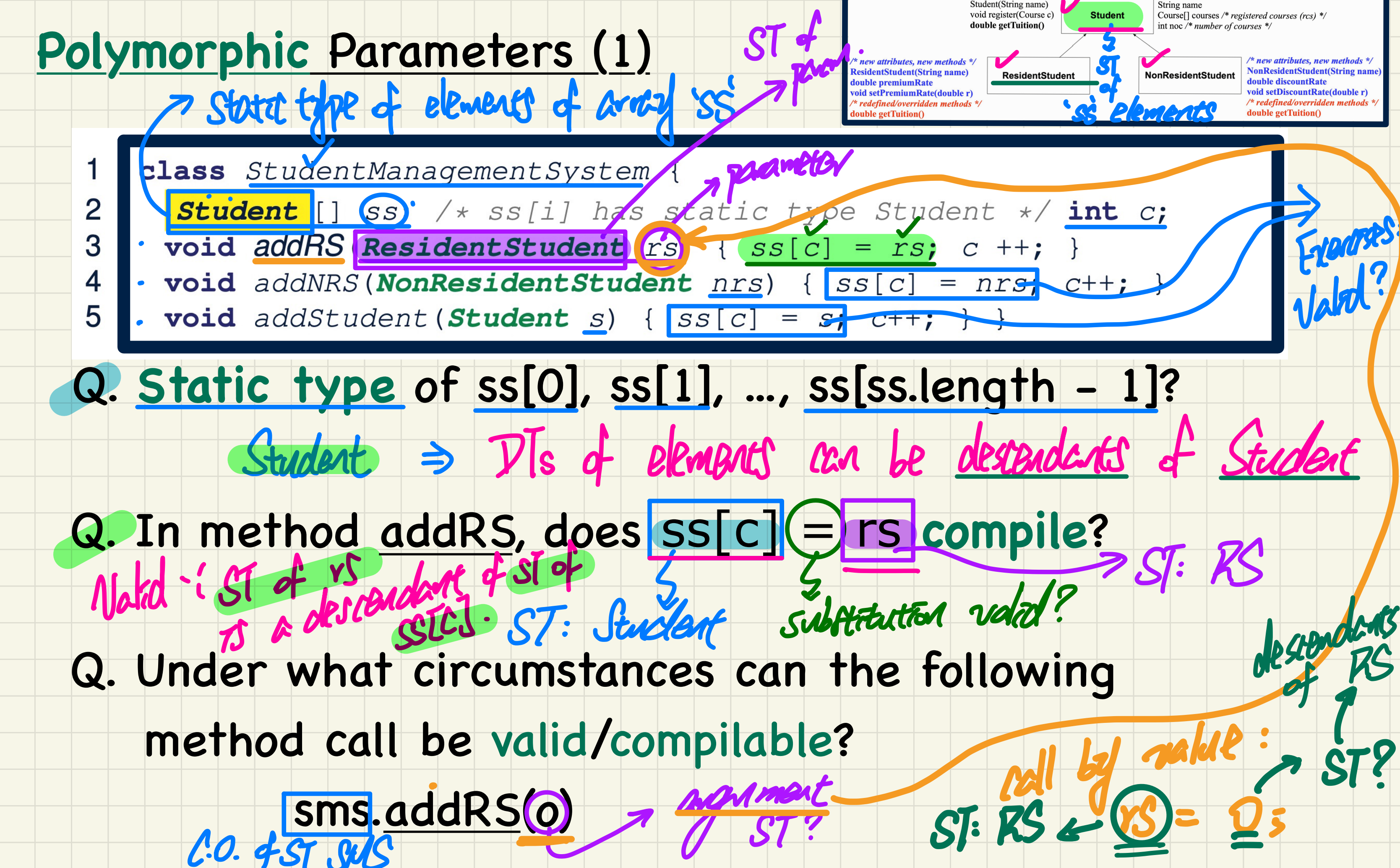
Static Types, Casts, Polymorphism (5)



Lecture 5

Part M

Inheritance -
Polymorphic Parameter Types



Polymorphic Parameters (1)
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
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++; } }
Q. Static type of ss[0], ss[1], ..., ss[ss.length - 1]?
Q. In method addRS, does ss[c] = rs compile?
Q. Under what circumstances can the following
method call be valid/compilable?
sms.addRS(o)

Polymorphic Parameters (2)

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

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

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

Student

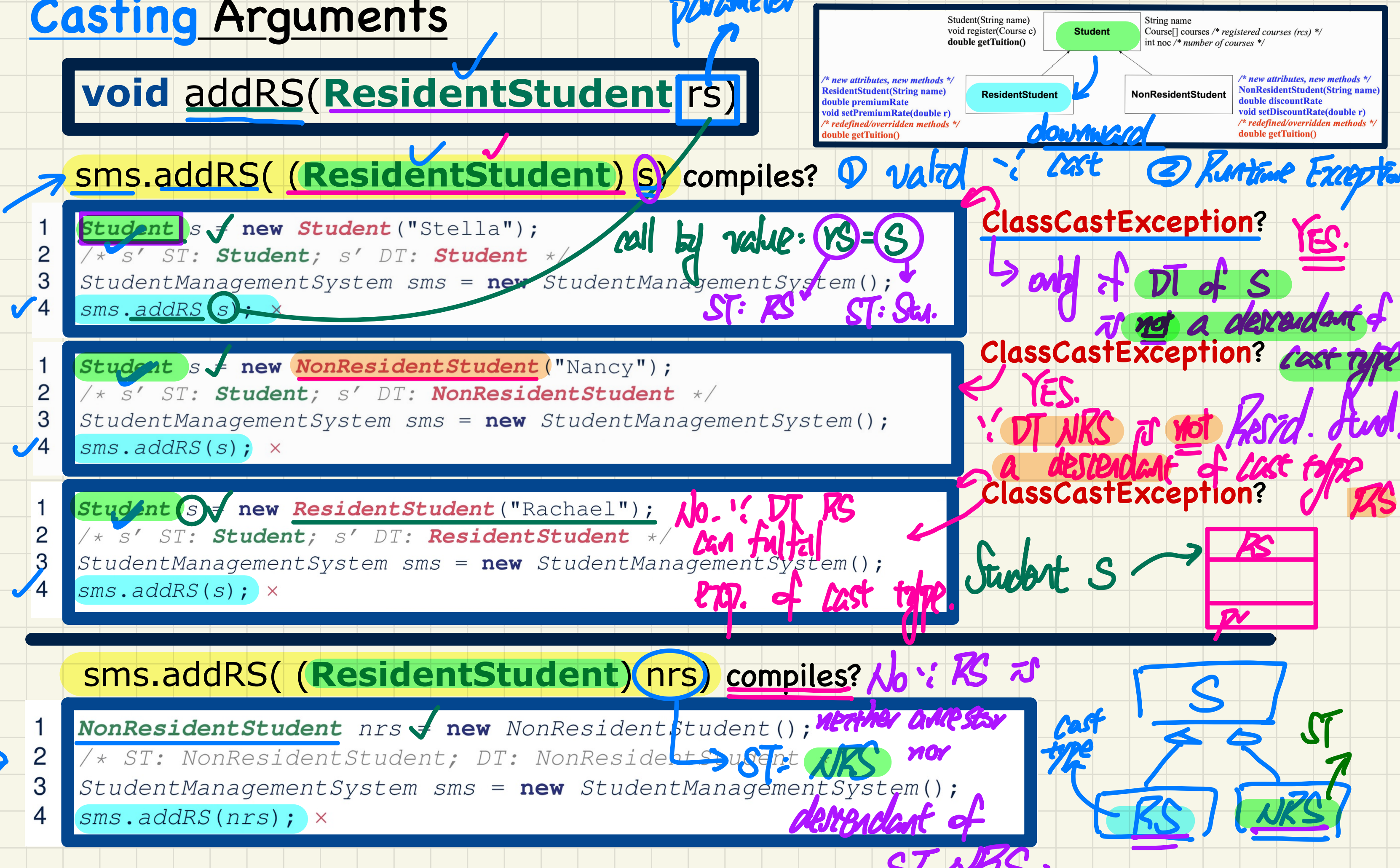
String name
Course[] courses /* *registered courses (rcs)* */
int noc /* *number of courses* */

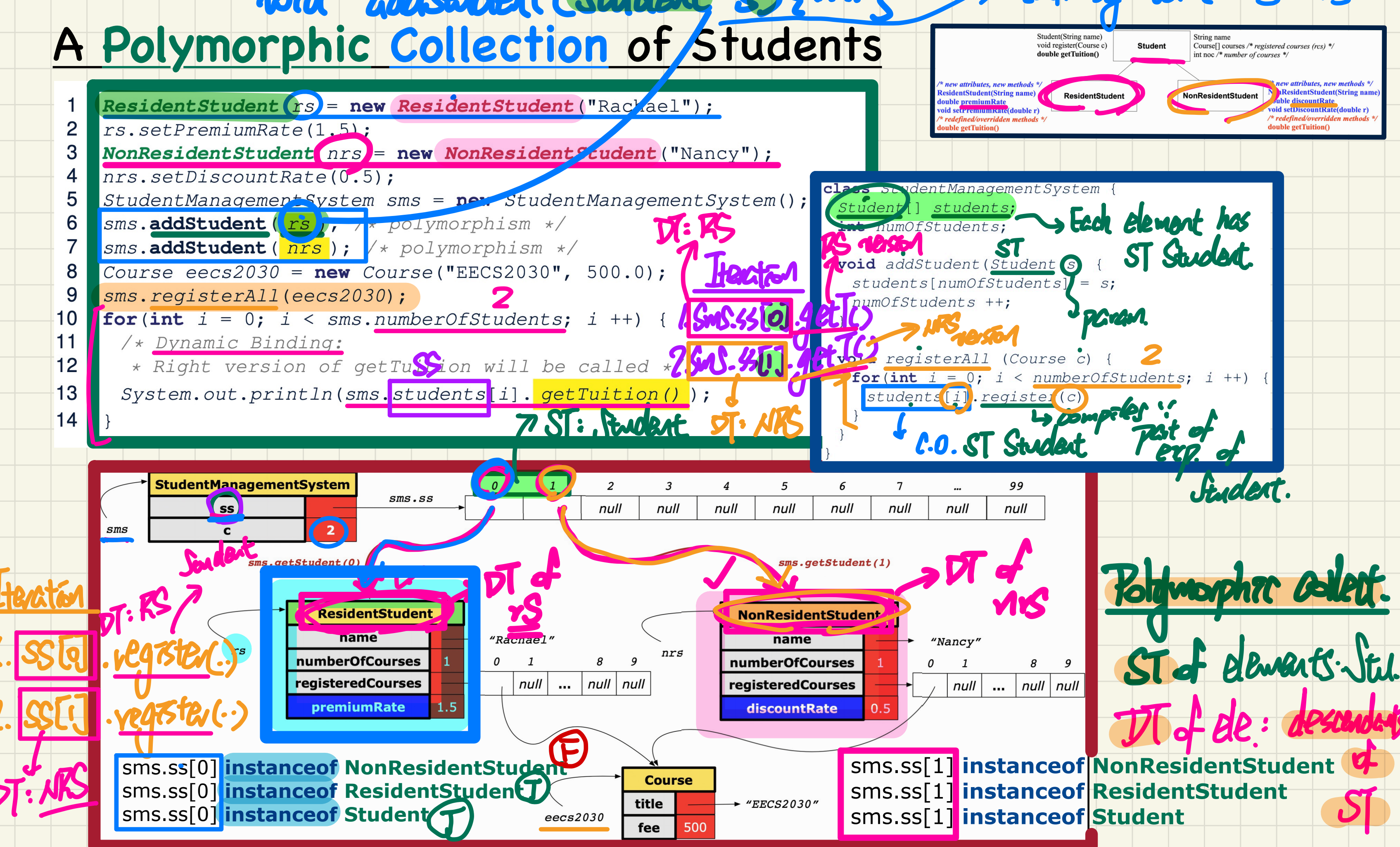
/* *new attributes, new methods* */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* *redefined/overridden methods* */
double getTuition()

ResidentStudent

NonResidentStudent

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


Casting Arguments
void addRS(ResidentStudent rs)
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
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); ×
ClassCastException?
1 Student s = new NonResidentStudent("Nancy");
2 /* s' ST: Student; s' DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
ClassCastException?
1 Student s = new ResidentStudent("Rachael");
2 /* s' ST: Student; s' DT: ResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
ClassCastException?
sms.addRS((ResidentStudent) nrs) compiles?
1 NonResidentStudent nrs = new NonResidentStudent();
2 /* ST: NonResidentStudent; DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(nrs); ×



A Polymorphic Collection of Students
1 ResidentStudent rs = new ResidentStudent("Rachael");
2 rs.setPremiumRate(1.5);
3 NonResidentStudent nrs = new NonResidentStudent("Nancy");
4 nrs.setDiscountRate(0.5);
5 StudentManagementSystem sms = new StudentManagementSystem();
6 sms.addStudent(rs); /* polymorphism */
7 sms.addStudent(nrs); /* polymorphism */
8 Course 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 }
class StudentManagementSystem {
Student[] students;
int numOfStudents;
void addStudent(Student s) {
students[numOfStudents] = s;
numOfStudents ++;
}
void registerAll (Course c) {
for(int i = 0; i < numberOfStudents; i ++) {
students[i].register(c)
}
}
}
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/* redefined/overridden methods */
double getTuition()
ResidentStudent
NonResidentStudent
/* new attributes, new methods */
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/* redefined/overridden methods */
double getTuition()
StudentManagementSystem
ss
c
2
sms
sms.ss
0 1 2 3 4 5 6 7 ... 99
null null null null null null null null
sms.getStudent(0)
sms.getStudent(1)
ResidentStudent
name
numberOfCourses 1
registeredCourses
premiumRate 1.5
"Rachael"
0 1 8 9
null ... null null
rs
NonResidentStudent
name
numberOfCourses 1
registeredCourses
discountRate 0.5
"Nancy"
0 1 8 9
null ... null null
nrs
Course
title
fee 500
"EECS2030"
eecs2030
sms.ss[0] instanceof NonResidentStudent
sms.ss[0] instanceof ResidentStudent
sms.ss[0] instanceof Student
sms.ss[1] instanceof NonResidentStudent
sms.ss[1] instanceof ResidentStudent
sms.ss[1] instanceof Student
void addStudent(Student s){...} → call by value : s = rs
Each element has ST Student
Iteration
Polymorphic collect.
ST of elements: Stu.
DT of ele: descendants of ST

Lecture 5

Part N

Inheritance - Polymorphic Return Types

Polymorphic Return Types

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

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

StudentManagementSystem: ss, c = 2 — sms.ss: indices 0, 1, 2 … 99 (2–99: null)

sms.getStudent(0) → ResidentStudent: name "Rachael", numberOfCourses 1, registeredCourses [0, 1: null, …, 8: null, 9: null], premiumRate 1.5

sms.getStudent(1) → NonResidentStudent: name "Nancy", numberOfCourses 1, registeredCourses [0, 1: null, …, 8: null, 9: null], discountRate 0.5

Course (eecs2030): title "EECS2030", fee 500

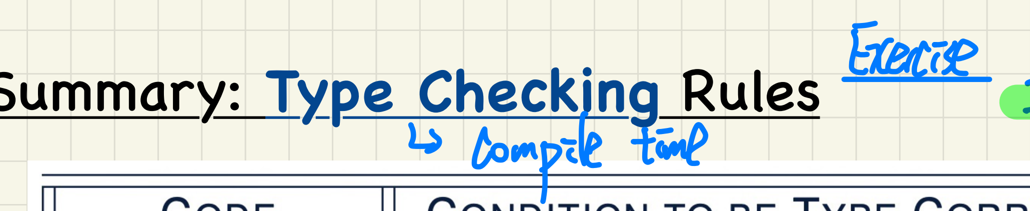
Handwritten notes:
- valid? YES: RT can fulfill the exp. of ST of S
- building polymorphic array / polymorphic collection
- ST: RT of getStudent
- ST: Student (RT)
- DT: RS
- DT of S: NRS
- ST: Student
- valid? ① Can the ST of s fulfill the exp. of the ST of getStudent's RV?
- static type of return value from this accessor method
- ↳ dynamic type of the return value can be any descendant class of ST.

Lecture 5

Part O

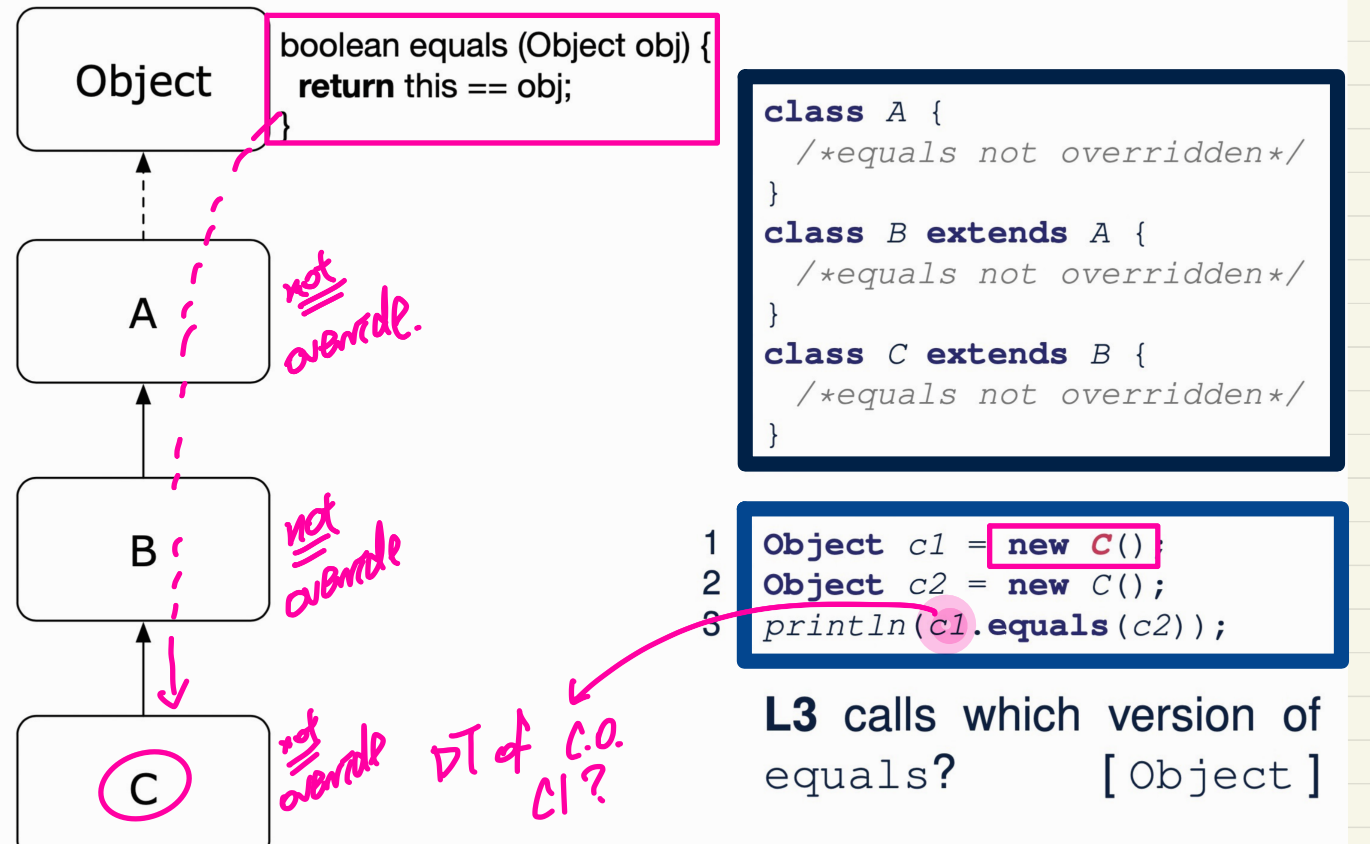
Inheritance -
Overridden Methods and Dynamic Binding

Summary: Type Checking Rules

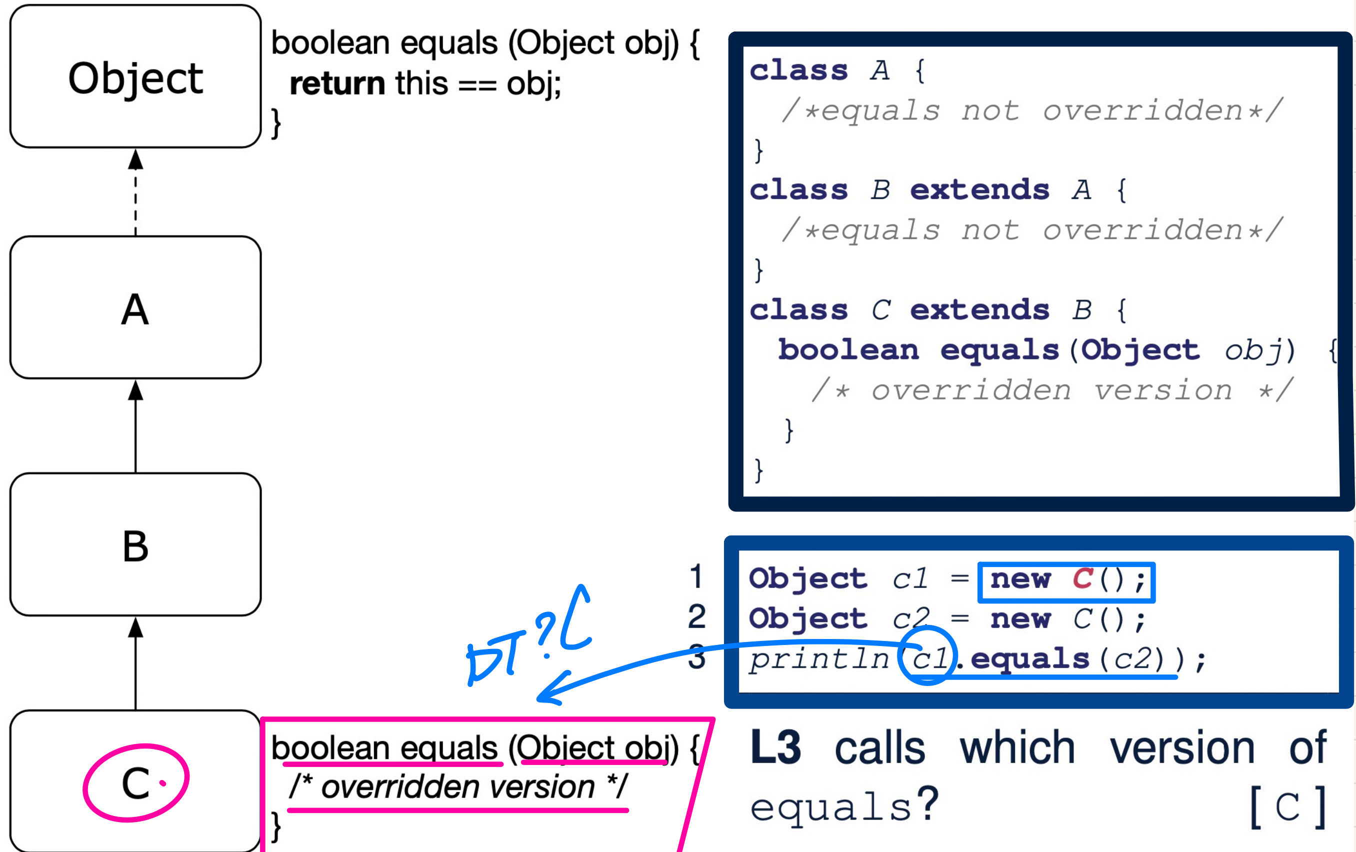


CODE	CONDITION TO BE TYPE CORRECT
x = y	Is y's ***ST*** a **descendant** of x's ***ST***?
x.m(y)	Is method m defined in x's ***ST***? Is y's ***ST*** a **descendant** of m's parameter's ***ST***?
z = x.m(y)	Is method m defined in x's ***ST***? Is y's ***ST*** a **descendant** of m's parameter's ***ST***?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

Overridden Methods and Dynamic Binding (1)



Overridden Methods and Dynamic Binding (2)



Overridden Methods and Dynamic Binding (3)

Object

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

A

B

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

C

closest ancestor overriding the equals method

DT of c.o? C

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

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

L3 calls which version of equals? [B]

Lecture 6

Part A

Abstract Classes

Abstract Implementation vs. Concrete Implementation

Polygon:
```
double getArea() {}
double[] sides;
void grow() { ... }
double getPerimeter() { ... }
```

Rectangle:
```
double getArea() { ... }
```

Triangle:
```
double getArea() { ... }
```

$a * b$

$\sqrt{s(s - a)(s - b)(s - c)}$

Alternative: No getArea in Polygon.

Hint: polymorphic collection of polygons?

ideal: use abstract keyword

delayed/deferred to subclasses

Empty Implementation

∵ at this level, we don't know how to calculate the area

sides.length == 4

sides.length == 3

ST of each element of ps: Polygon

ST: Polygon

not appropriate

```
Polygon[] ps;
double total = 0;
for(int i = 0; i < nop; i++) {
  total += ps[i].getArea();
}
```

not compiling Polygon.

∵ getArea not expected on

Abstract Class vs. Concrete Descendants

```java
public abstract class Polygon {
  double[] sides;
  Polygon(double[] sides) { this.sides = sides; }
  void grow() {
    for(int i = 0; i < sides.length; i ++) { sides[i] ++; }
  }
  double getPerimeter() {
    double perimeter = 0;
    for(int i = 0; i < sides.length; i ++) {
      perimeter += sides[i];
    }
    return perimeter;
  }
  abstract double getArea();
}
```

extends

extends

```java
public class Rectangle extends Polygon {
  Rectangle(double length, double width) {
    super(new double[4]);
    sides[0] = length; sides[1] = width;
    sides[2] = length; sides[3] = width;
  }
  double getArea() { return sides[0] * sides[1]; }
}
```

```java
public class Triangle extends Polygon {
  Triangle(double side1, double side2, double side3) {
    super(new double[3]);
    sides[0] = side1; sides[1] = side2; sides[2] = side3;
  }
  double getArea() {
    /* Heron's formula */
    double s = getPerimeter() * 0.5;
    double area = Math.sqrt(
      s * (s - sides[0]) * (s - sides[1]) * (s - sides[2]));
    return area;
  }
}
```

At least one method is abstract

implementation delayed to the subclasses

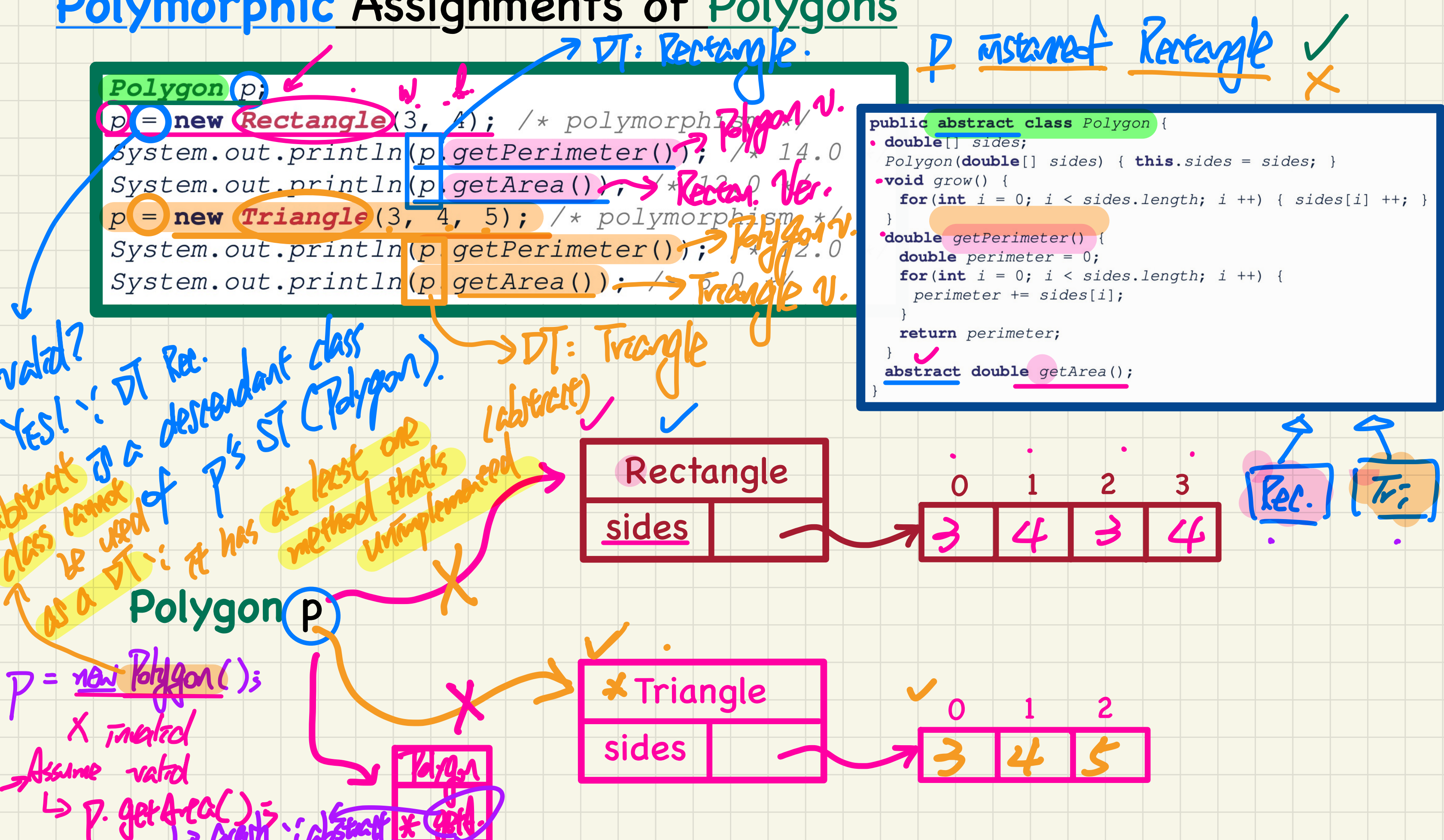
rather than { }

no longer abstract

static method

grow

Polymorphic Assignments of Polygons



Polymorphic Collection of Polygons

```
public abstract class Polygon {
  double[] sides;
  Polygon(double[] sides) { this.sides = sides; }
  void grow() {
    for(int i = 0; i < sides.length; i ++) { sides[i] ++; }
  }
  double getPerimeter() {
    double perimeter = 0;
    for(int i = 0; i < sides.length; i ++) {
      perimeter += sides[i];
    }
    return perimeter;
  }
  abstract double getArea();
}
```

Rectangle   Triangle

```
PolygonCollector col = new PolygonCollector();
col.addPolygon(new Rectangle(3, 4)); /* polymorphism
col.addPolygon(new Triangle(3, 4, 5)); /* polymorphis
System.out.println(col.polygons[0]. getPerimeter ());
System.out.println(col.polygons[1]. getPerimeter ());
col.growAll();
System.out.println(col.polygons[0]. getPerimeter ());
System.out.println(col.polygons[1]. getPerimeter ());
```

```
public class PolygonCollector {
  Polygon[] polygons;
  int numberOfPolygons;
  PolygonCollector() { polygons = new Polygon[10]; }
  void addPolygon(Polygon p) {
    polygons[numberOfPolygons] = p; numberOfPolygons ++;
  }
  void growAll() {
    for(int i = 0; i < numberOfPolygons; i ++) {
      polygons[i].grow();
    }
  }
}
```

PolygonCollector: nop, polygons

Rectangle: sides

0 1 2 3

Triangle: sides

0 1 2

0 1 9

Polymorphic Return Type of Polygons

```java
public abstract class Polygon {
  double[] sides;
  Polygon(double[] sides) { this.sides = sides; }
  void grow() {
    for(int i = 0; i < sides.length; i ++) { sides[i] ++; }
  }
  double getPerimeter() {
    double perimeter = 0;
    for(int i = 0; i < sides.length; i ++) {
      perimeter += sides[i];
    }
    return perimeter;
  }
  abstract double getArea();
}
```

Rectangle

Triangle

```java
public class PolygonConstructor {
  Polygon getPolygon(double[] sides) {
    Polygon p = null;
    if(sides.length == 3) {
      p = new Triangle(sides[0], sides[1], sides[2]);
    }
    else if(sides.length == 4) {
      p = new Rectangle(sides[0], sides[1]);
    }
    return p;
  }
  void grow(Polygon p) { p.grow(); }
}
```

```java
PolygonConstructor con = new PolygonConstructor();
double[] recSides = {3, 4, 3, 4}; p = con.getPolygon(recSides)
System.out.println(p instanceof Polygon);    ✓
System.out.println(p instanceof Rectangle);  ✓
System.out.println(p instanceof Triangle);   ×
System.out.println(p.getPerimeter()); /* 14.0 */
System.out.println(p.getArea()); /* 12.0 */
con.grow(p);
System.out.println(p.getPerimeter()); /* 18.0 */
System.out.println(p.getArea()); /* 20.0 */
double[] triSides = {3, 4, 5}; p = con.getPolygon(triSides);
System.out.println(p instanceof Polygon);    ✓
System.out.println(p instanceof Rectangle);  ×
System.out.println(p instanceof Triangle);   ✓
System.out.println(p.getPerimeter()); /* 12.0 */
System.out.println(p.getArea()); /* 6.0 */
con.grow(p);
System.out.println(p.getPerimeter()); /* 15.0 */
System.out.println(p.getArea()); /* 9.921 */
```

valid ∵ RHS's ST is a descendant of p's ST.

ST: Polygon

DT: Rec.

Polygon v.

Rec. ver.

ST of return value

call by value

valid ∵ expression to return (p) has ST Polygon, which is a descendant of RT Polygon.

Polygon p

Rectangle

sides

0	1	2	3
3 → 4	4 → 5	3 → 4	4 → 5

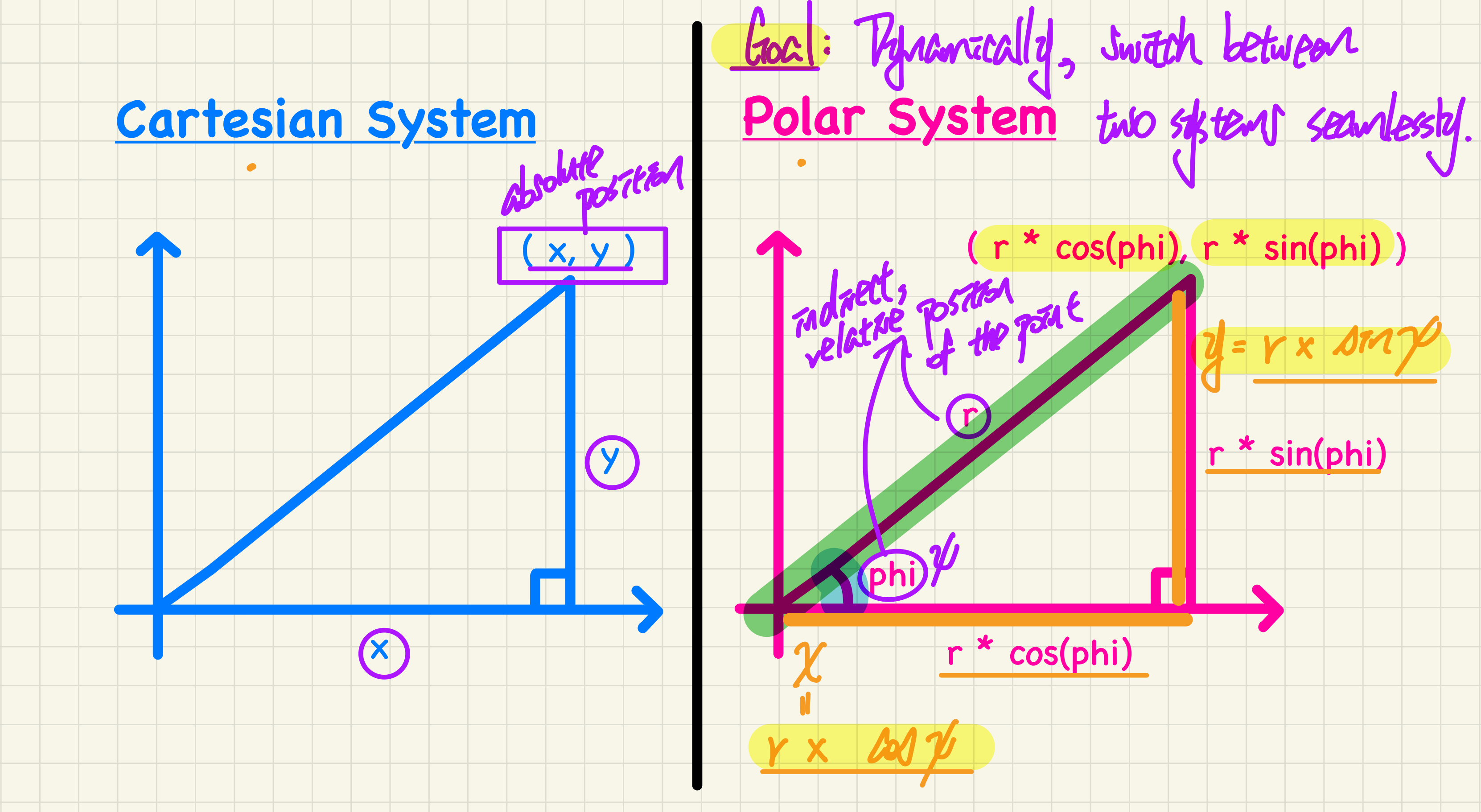
Triangle

sides

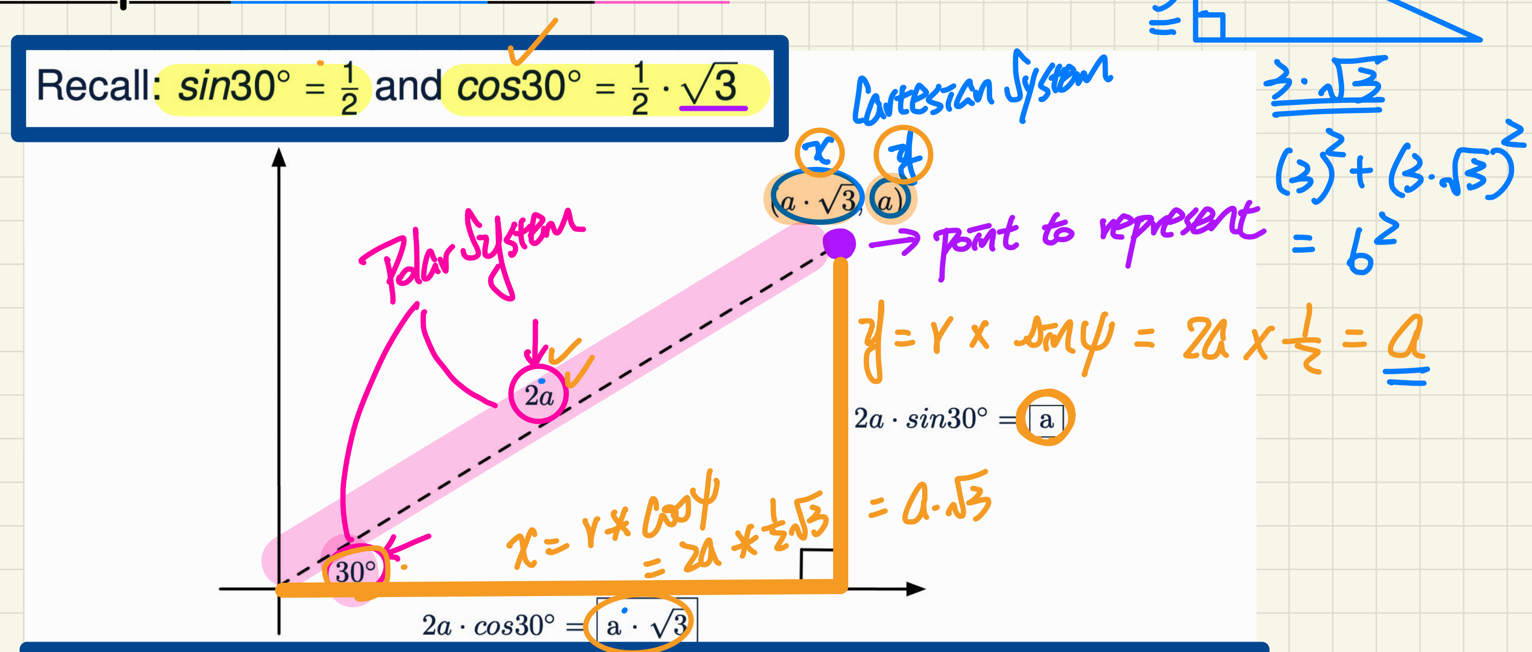
0	1	2

Lecture 6
Part B
Interfaces

Representations of 2-D Points: Cartesian vs. Polar

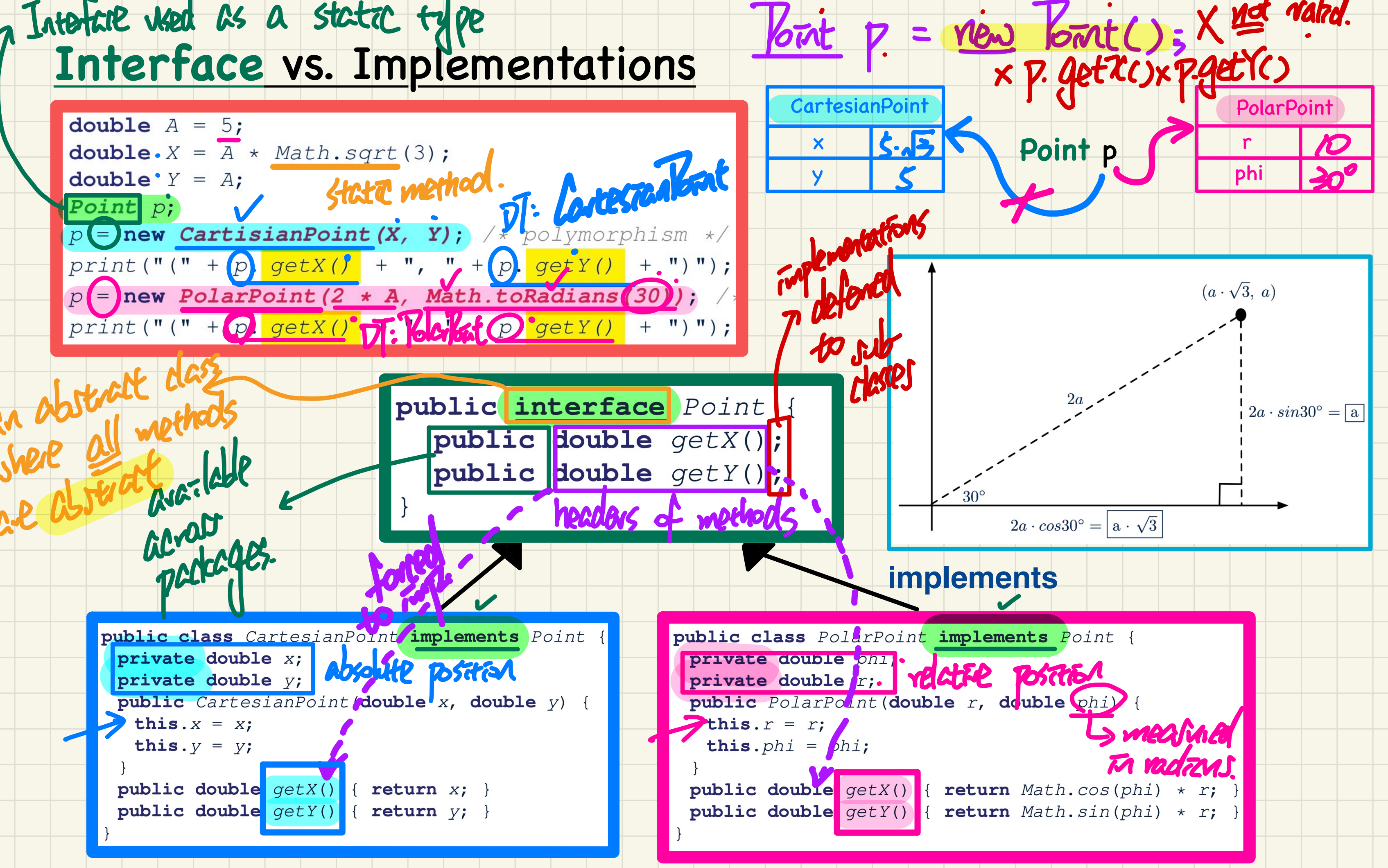


Example: Cartesian vs. Polar



We consider the same point represented differently as:

- $r = 2a$, $\psi = 30°$ [polar system]
- $x = 2a \cdot cos30° = a \cdot \sqrt{3}$, $y = 2a \cdot sin30° = a$ [cartesian system]



Interface vs. Implementations
Interface used as a static type
double A = 5;
double X = A * Math.sqrt(3);
double Y = A;
Point p;
p = new CartisianPoint(X, Y); /* polymorphism */
print("(" + p.getX() + ", " + p.getY() + ")");
p = new PolarPoint(2 * A, Math.toRadians(30)); /*
print("(" + p.getX() + p.getY() + ")");
static method.
DT: CartesianPoint
DT: PolarPoint
Point p = new Point(); X not valid.
x p.getX() x p.getY()
CartesianPoint
x
y
5
Point p
PolarPoint
r
10
phi
30°
implementations deferred to sub classes
public interface Point {
public double getX();
public double getY();
}
headers of methods
an abstract class where all methods are abstract
available across packages.
forced to impl
implements
$(a \cdot \sqrt{3},\ a)$
2a
$2a \cdot sin30° =$ a
30°
$2a \cdot cos30° =$ $a \cdot \sqrt{3}$
public class CartesianPoint implements Point {
private double x;
private double y;
public CartesianPoint(double x, double y) {
this.x = x;
this.y = y;
}
public double getX() { return x; }
public double getY() { return y; }
}
absolute position
public class PolarPoint implements Point {
private double phi;
private double r;
public PolarPoint(double r, double phi) {
this.r = r;
this.phi = phi;
}
public double getX() { return Math.cos(phi) * r; }
public double getY() { return Math.sin(phi) * r; }
}
relative position
measured in radians.

Lecture 7

Part A

Generics in Java -
General Book: Storage vs. Retrieval

General Book

names → [...]

records → [...]

Supplier

```java
public class Book {
  private String[] names;
  private Object[] records;
  /* add a name-record pair to the book */
  public void add (String name, Object record) { ... }
  /* return the record associated with a given name */
  public Object get (String name) { ... } }
```

STORAGE: arg object's ST must be a descendant of Object

RETRIEVAL: return value's DT must be a descendant of Object

Client

```java
1 Date birthday; String phoneNumber;
2 Book b; boolean isWednesday;
3 b = new Book();
4 phoneNumber = "416-67-1010";
5 b.add ("Suyeon", phoneNumber);
6 birthday = new Date(1975, 4, 10);
7 b.add ("Yuna", birthday);
8 isWednesday = b.get("Yuna").getDay() == 4;
```

any objects can be added

Call by value valid ∵ String is a descendant of Object.

Date

✗ (line 8) ST of RV is: Object

available on the DT of RV, not ST.

General Book: Retrieval from a Polymorphic Array

```
1  Date birthday; String phoneNumber;
2  Book b; boolean isWednesday;
3  b = new Book();
4  phoneNumber = "416-67-1010";
5  b.add ("Suyeon", phoneNumber);
6  birthday = new Date(1975, 4, 10);
7  b.add ("Yuna", birthday);
8  isWednesday = b.get("Yuna").getDay() == 4;
```

Client

Object

Date   String

downward casting.

cast type

DT: Date

DT of b.get("Suyeon")

anonymous cast with ST: Date

```
isWednesday = ((Date) b.get("Yuna")).getDay() == 4;
```

```
isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
```

↳ DT: String

Compile (downward cast) but ClassCast Excep.

object expression

```
if (b.get("Suyeon") instanceof Date) {
  isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
}
```

DT: String

evaluates to false

↳ for any retrieval from a general book, it's required to have instanceof checks & type casts.

General Book violates Single Choice Principle

Storage

```
Object rec1 = new C1(); b.add(..., rec1);
Object rec2 = new C2(); b.add(..., rec2);
...
Object rec100 = new C100(); b.add(..., rec100);
```

storage

Retrievals

retrieval

```
Object rec = b.get("Jim");
if (rec instanceof C1) { ((C1) rec).m1; }
...
else if (rec instanceof C100) { ((C100) rec).m100; }
else if (rec instanceof C101) { - - - }
```

prevent ClassCastException

```
Object rec = b.get("Jim");
if (rec instanceof C1) { ((C1) rec).m1; }
...
else if (rec instanceof C100) { ((C100) rec).m100; }
else if (rec instanceof C101) { - - - }
```

the same exhaustive checks on the DT of the retrieved record are repeated

What if a new type C101 is introduced?

What if type C100 becomes obselete?

Lecture 7

Part B

Generics in Java -
Generic Book: Storage vs. Retrieval

Generic Book

Supplier

```java
class Book<E> {
  private String[] names;
  private E[] records;
  /* add a name-record pair to the book */
  public void add (String name, E record) { ... }
  /* return the record associated with a given name */
  public E get (String name) { ... } }
```

Client

```java
1  Date birthday; String phoneNumber;
2  Book<Date> b; boolean isWednesday;
3  b = new Book<Date>();
4  phoneNumber = "416-67-1010";
5  b.add ("Suyeon", phoneNumber);
6  birthday = new Date(1975, 4, 10);
7  b.add ("Yuna", birthday);
8  isWednesday = b.get("Yuna").getDay() == 4;
```

```java
class Book<E> {
  private String[] names;
  private E[] records;
  /* add a name-record pair to the book */
  public void add (String name, E record) { ... }
  /* return the record associated with a given name */
  public E get (String name) { ... } }
```

Lecture 7

Part C

Generics in Java -
Generic Collection Classes

API: ArrayList<E>

int	**size**() Returns the number of elements in this list.
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**get**(int index) Returns the element at the specified position in this list.

declaration of generic type parameter

uses E for type declaration

```
ArrayList<Point> list1;
  ⋮
✓ list1.add(p1);
✗ list1.add("hello");

ArrayList<String> list2;
  ⋮
✗ list2.add(p1);
✓ list2.add("hello");
```

Use of ArrayList<String>

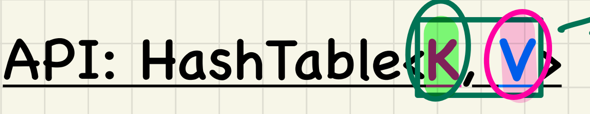
```
1  import java.util.ArrayList;
2  public class ArrayListTester {
3    public static void main(String[] args) {
4      ArrayList<String> list = new ArrayList<String>();
5      println(list.size());
6      println(list.contains("A"));
7      println(list.indexOf("A"));
8      list.add("A");
9      list.add("B");
10     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
11     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
12     list.add(1, "C");
13     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
14     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
15     list.remove("C");
16     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
17     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
18
19     for(int i = 0; i < list.size(); i ++) {
20       println(list.get(i));
21     }
22   }
23 }
```

int	`size()` Returns the number of elements in this list.
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`contains(Object o)` Returns true if this list contains the specified element.
E	`remove(int index)` Removes the element at the specified position in this list.
boolean	`remove(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
int	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	`get(int index)` Returns the element at the specified position in this list.

instantiating E by String.

consequential copy of ArrayList<String>

API: HashTable<K, V>



`int`	`size()` Returns the number of keys in this hashtable.
`boolean`	`containsKey(Object key)` Tests if the specified object is a key in this hashtable.
`boolean`	`containsValue(Object value)` Returns true if this hashtable maps one or more keys to this value.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`V`	`put(K key, V value)` Maps the specified key to the specified `value` in this hashtable.
`V`	`remove(Object key)` Removes the key (and its corresponding value) from this hashtable.

Use of HashTable<String, String>

```java
import java.util.Hashtable;
public class HashTableTester {
  public static void main(String[] args) {
    Hashtable<String, String> grades = new Hashtable<S
     System.out.println("Size of table: " + grades.size
     System.out.println("Key Alan exists: " + grades.co
     System.out.println("Value B+ exists: " + grades.co
     grades.put("Alan", "A");
     grades.put("Mark", "B+");
     grades.put("Tom", "C");
     System.out.println("Size of table: " + grades.size());
     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
     System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
     System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
     System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
     System.out.println("Value A exists: " + grades.containsValue("A"));
     System.out.println("Value B+ exists: " + grades.containsValue("B+"));
     System.out.println("Value C exists: " + grades.containsValue("C"));
     System.out.println("Value A+ exists: " + grades.containsValue("A+"));
     System.out.println("Value of existing key Alan: " + grades.get("Alan"));
     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
     System.out.println("Value of existing key Tom: " + grades.get("Tom"));
     System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
     grades.put("Mark", "F");
     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
     grades.remove("Alan");
     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
     System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
```

int	size()	Returns the number of keys in this hashtable.
boolean	containsKey(Object key)	Tests if the specified object is a key in this hashtable.
boolean	containsValue(Object value)	Returns true if this hashtable maps one or more keys to this value.
	get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
	put(key, value)	Maps the specified key to the specified value in this hashtable.
	remove(Object key)	Removes the key (and its corresponding value) from this hashtable.

Lecture 8

Part A

Recursion -
Basics: Thinking Recursively, Call Stack

Solving a Problem Recursively

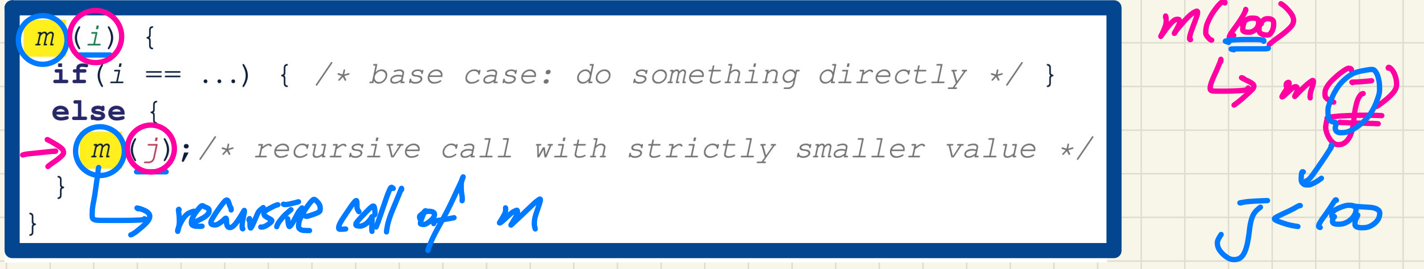
Given a **small** problem: → Solve it **directly**:

Given a **big** problem:

Divide it into **smaller** problems:

Assume solutions to **smaller** problems:

Combine solutions to **smaller** problems:



m(100)
↳ m(j)
j < 100

Recursive Solution: factorial

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

$5! = 5 * 4 * 3 * 2 * 1$

↳ $4!$

$= 5 * 4!$

$= 5 * (5-1)!$

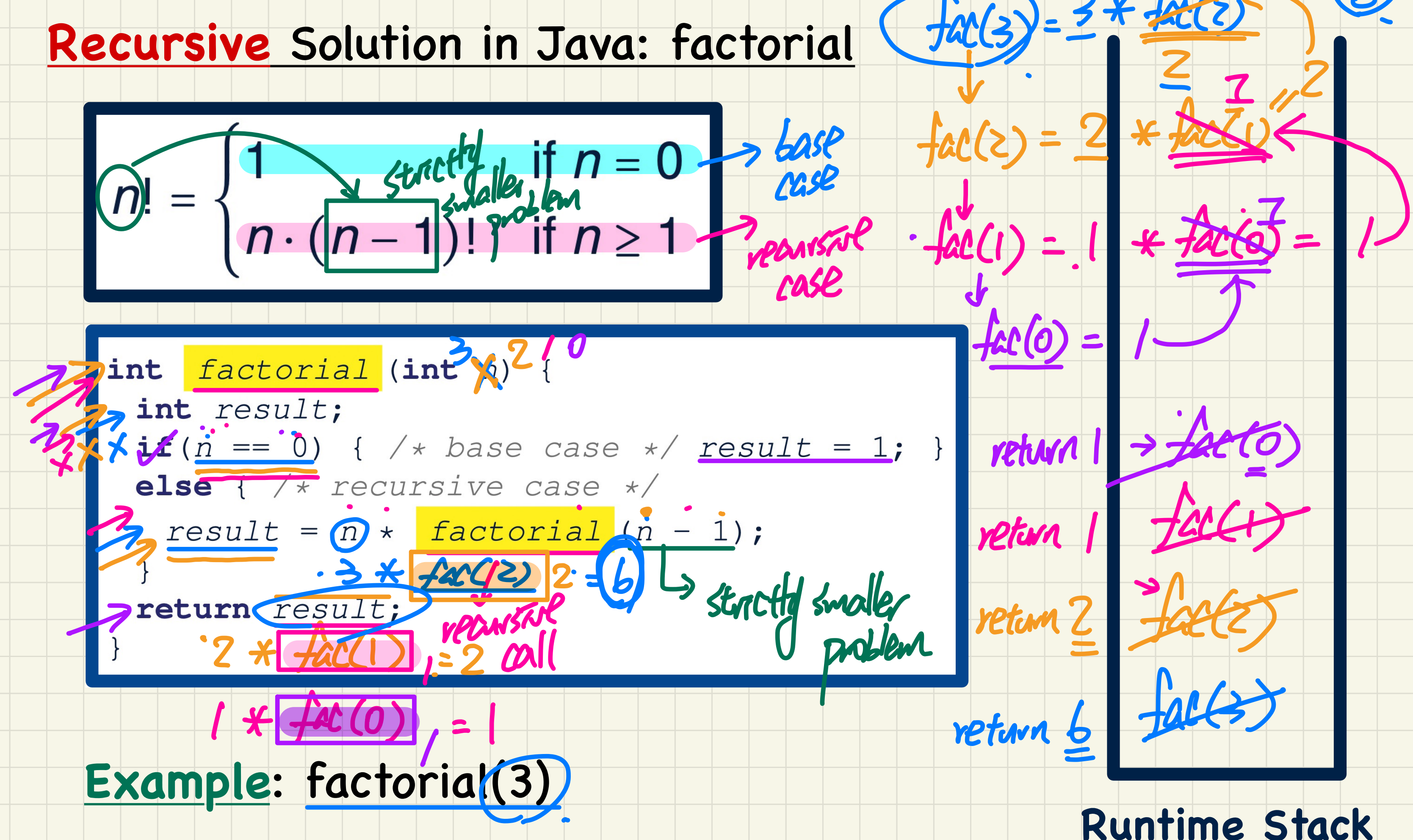
↳ strictly smaller problem size

$n-1 < n$

$n! = n * (n-1) * (n-2) * \cdots * 2 * 1$

↳ $(n-1)!$

$= n * (n-1)!$



Recursive Solution in Java: factorial
n! = 1 if n = 0
n · (n − 1)! if n ≥ 1
base case
recursive case
strictly smaller problem
int factorial (int n) {
int result;
if(n == 0) { /* base case */ result = 1; }
else { /* recursive case */
result = n * factorial (n - 1);
}
return result;
}
recursive call
Example: factorial(3)
return 1
return 1
return 2
return 6
Runtime Stack

Common Errors of Recursion (1)

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

missing base case(s)

Infinite Recursion

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

Common Errors of Recursion (2)

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

Infinite Recursion

fac(3)
↳ fac(3)
↳ fac(3)
↳ fac(3) (never able to reach the base case)
⋮

problem size for recursive call is not strictly smaller.

Recursive Solution: Fibonacci Numbers

F_7 F_8 F_9

F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Base Cases

$F_1 = 1$

$F_2 = 1$

Recursive Cases

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

$n > 2$

$n-1$: strictly smaller than n

$n-2 < n$

solved recursively by two recursive calls

$F_9 = F_7 + F_8$.

Recursive Solution in Java: Fibonacci Numbers

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

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

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

fib(2) + fib(1) = 2

1 1

fib(4) 3

2 fib(3) fib(2) 1

fib(2) fib(1)

1 1

return 1 fib(2)

return 1 fib(1)

return 1 fib(2)

return 2 fib(3)

return 3 fib(4)

Runtime Stack

Example: fib(4)

fib(5)

fib(4) + fib(3)

fib(3) + fib(2) fib(2) + fib(1)

fib(2) + fib(1)

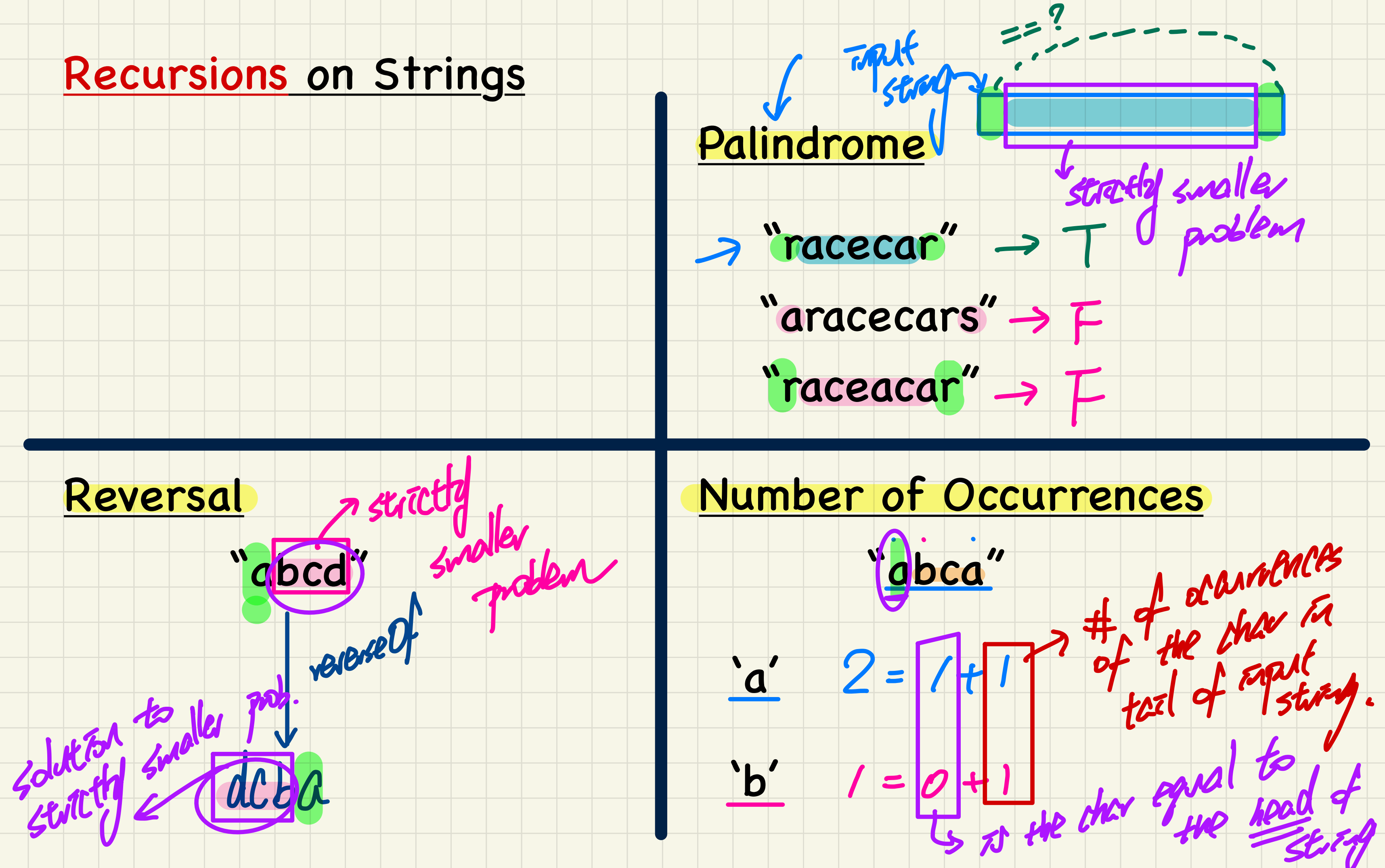
Lecture 7

Part B

Recursion -
Examples: Recursions on Strings

Use of String

```
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();
  }
}
```



Recursions on Strings
Palindrome
input string
==?
strictly smaller problem
"racecar" → T
"aracecars" → F
"raceacar" → F
Reversal
"abcd"
strictly smaller problem
reverseOf
dcba
solution to strictly smaller prob.
Number of Occurrences
"abca"
'a' 2 = 1 + 1
'b' 1 = 0 + 1
# of occurrences of the char in tail of input string.
is the char equal to the head of the string

Problem: Palindrome

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

word → [0 | 1 | middle | word.length() − 1]

word.substring(1, word.length() − 1) → exclusive

base cases

recursive case

recursive call to solve a subproblem with strictly smaller size

isP(racecar) → T

r == r && isP(aceca)

a == a && isP(cec)

c == c && isP(e)

base case → T

Problem: Reverse of a String

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

base cases

recursive call to solve a strictly smaller problem.

recursive case

reverseOf(abcd) → dcba

↳ reverseOf(bcd) + a → dcb

↳ reverseOf(cd) + b → dc

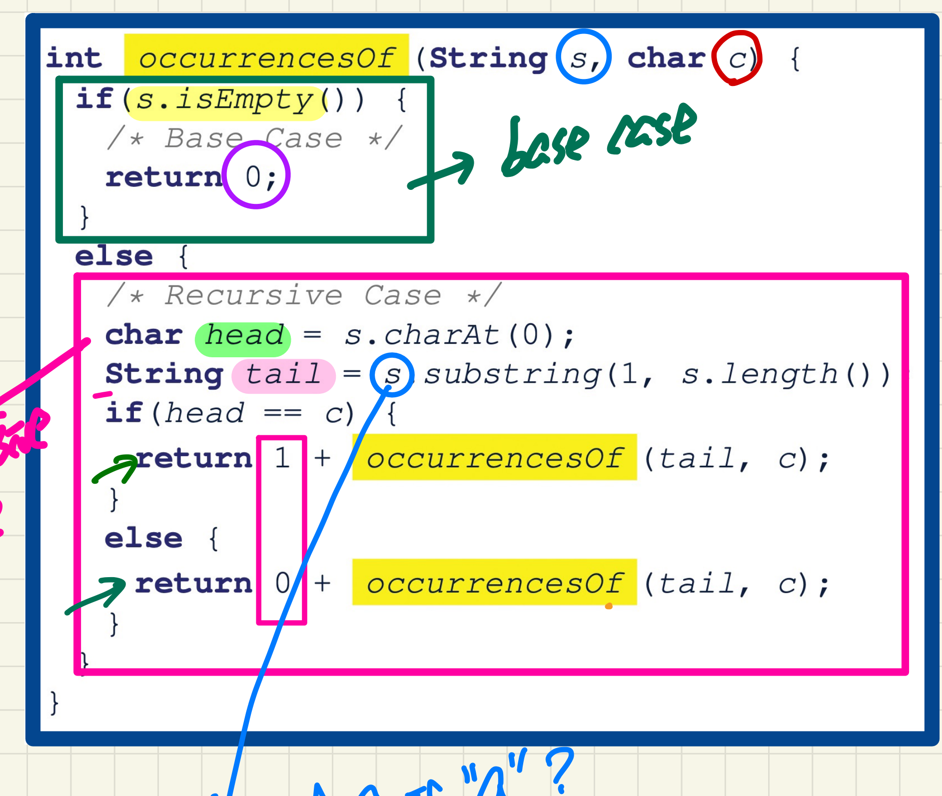
↳ reverseOf(d) + c → d

s, 1, s.length() - 1

tail

output → reverseOf(tail), s.length()-1

Problem: Number of Occurrences



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

Lecture 7

Part C

Recursion -
Examples: Recursions on Arrays

Recursion on an Array: Passing new Sub-Arrays

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

Say a1 = {}, consider m(a1)

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

Recursion on an Array: Passing Same Array Reference

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

base cases

recursive case

array of length 1.

empty array

[0, -1] → empty range.

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

min index

max index

m(a1, 0, -1)

from to

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

0	1	2
A	B	C

a2

m(a2, 0, 2)

m(a2, 1, 2)

strictly smaller problem (elements from indices 1 to 2)

m(2, 2)

strictly smaller problem (last item in array).

Problem: Are All Numbers Positive?

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

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

recursive helper method

min index

max index

base cases

empty array

array of length 1

recursive case

from, from+1, to

0, 1, a.length-1

a

allPositiveHelper(...)

>0

tail of the array

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);
  }
}
```

{4}

a.length - 1

True

True

0

4

from
to

a

Tracing Recursion: allPositive

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

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

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

allPositive(a)

allPH(a,0,3)

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

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

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

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);
  }
}
```

Exercise: Trace!

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);
  }
}
```

recursive helper method.

base cases

recursive case

sorted

a

<=

strictly smaller problem

Tracing Recursion: isSorted

Say a = {}

isSorted(a)

isSH(a,0,-1)

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

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

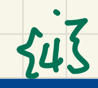
Tracing Recursion: isSorted

Say a = {4}

isSorted(a)

isSH(a,0,0)

return true



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

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

Tracing Recursion: isSorted

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

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

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

isSorted(a) {3,6,6,7}

isSH(a,0,3) T

a[0]<=a[1] T && isSH(a,1,3) T

a[1]<=a[2] T && isSH(a,2,3) T

a[2]<=a[3] T && isSH(a,3,3) T

0	1	2	3
3	6	6	7

Tracing Recursion: isSorted

Say a = {3,6,5,7} → (F)

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

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

```
isSorted(a)
  isSH(a,0,3)
    a[0]<=a[1]
    isSH(a,1,3)
      a[1]<=a[2]
      isSH(a,2,3)
        a[2]<=a[3]
        isSH(a,3,3)
```

Exercise: Trace

I hope you enjoyed the journey with me.

All the Best!