# Lecture 5

## Part A

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

# Inheritance: Motivating Problem

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

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

# First Design Attempt

```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

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

only applicable to RS.

only applicable to NRS.

not applicable => wasted space

RS: Student rs = new Student(1);

```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 ++;
  }
}
```

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 amend NEW

## 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 -<br>Student Management System:<br>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());
  }
}
```

jim → Res.S.: rcs, pr 1.25 → [0 1 ... 9]
jeremy → NonRes.S.: rcs, dr 0.75 → [0 1 ... 9]
c1 → Course: title "EECS2030", fee 500
c2 → Course: title "EECS3311", fee 500

# 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){ ... }

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

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

# Student Classes (without inheritance): Maintenance (2)

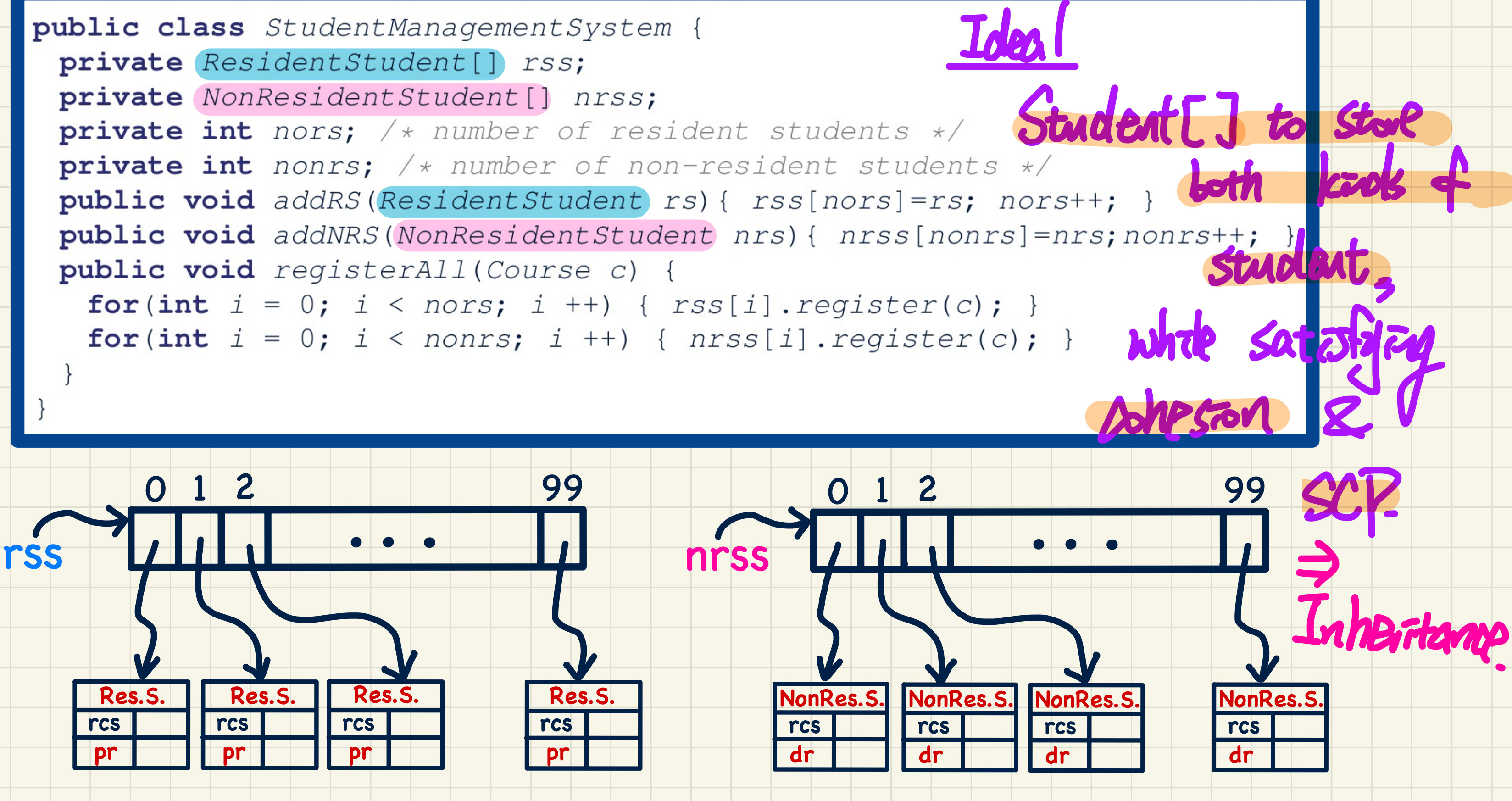
```java
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a m
  public ResidentStudent (String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc]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

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

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

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

# A Collection of Students (without inheritance)

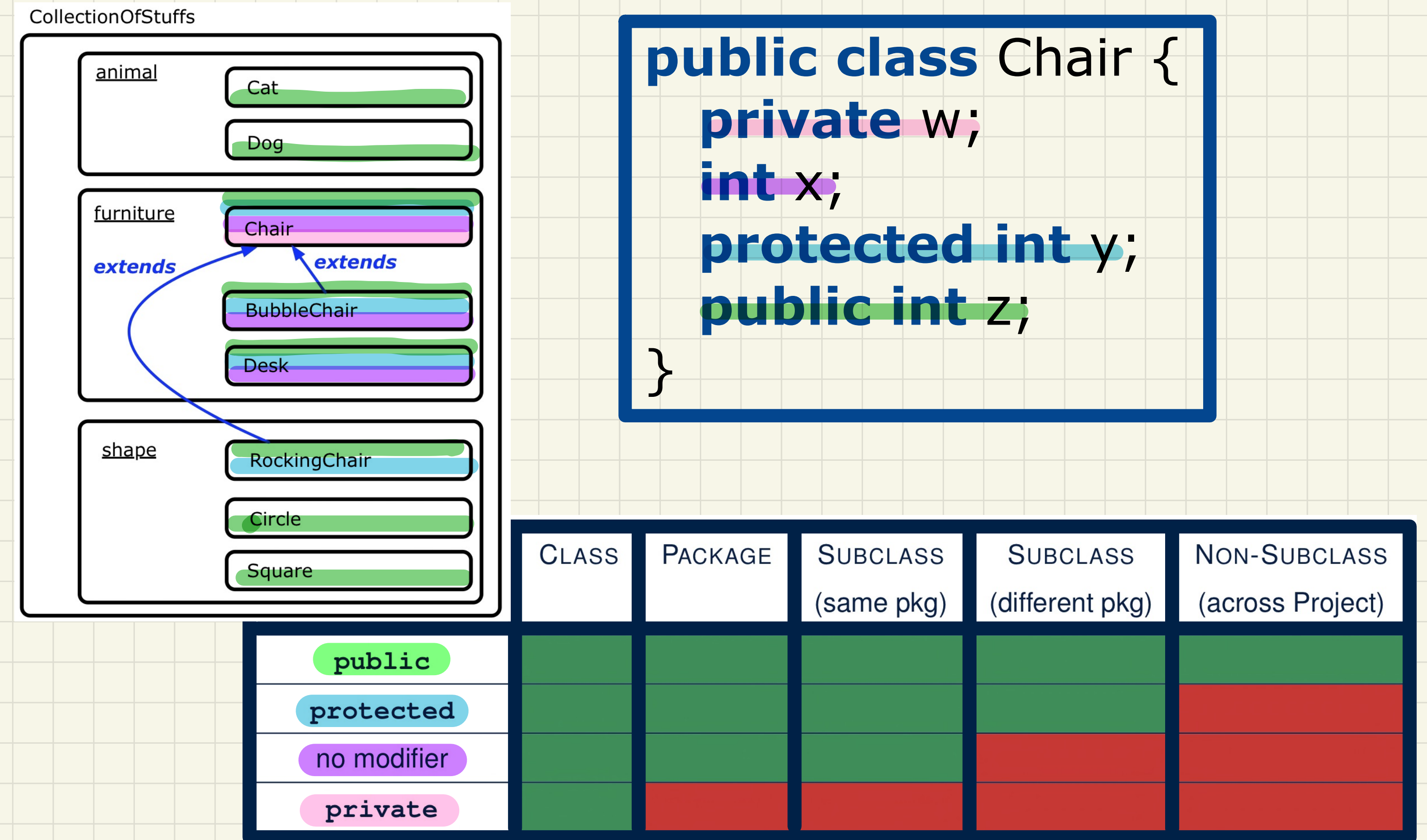


# Lecture 5

## Part C

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

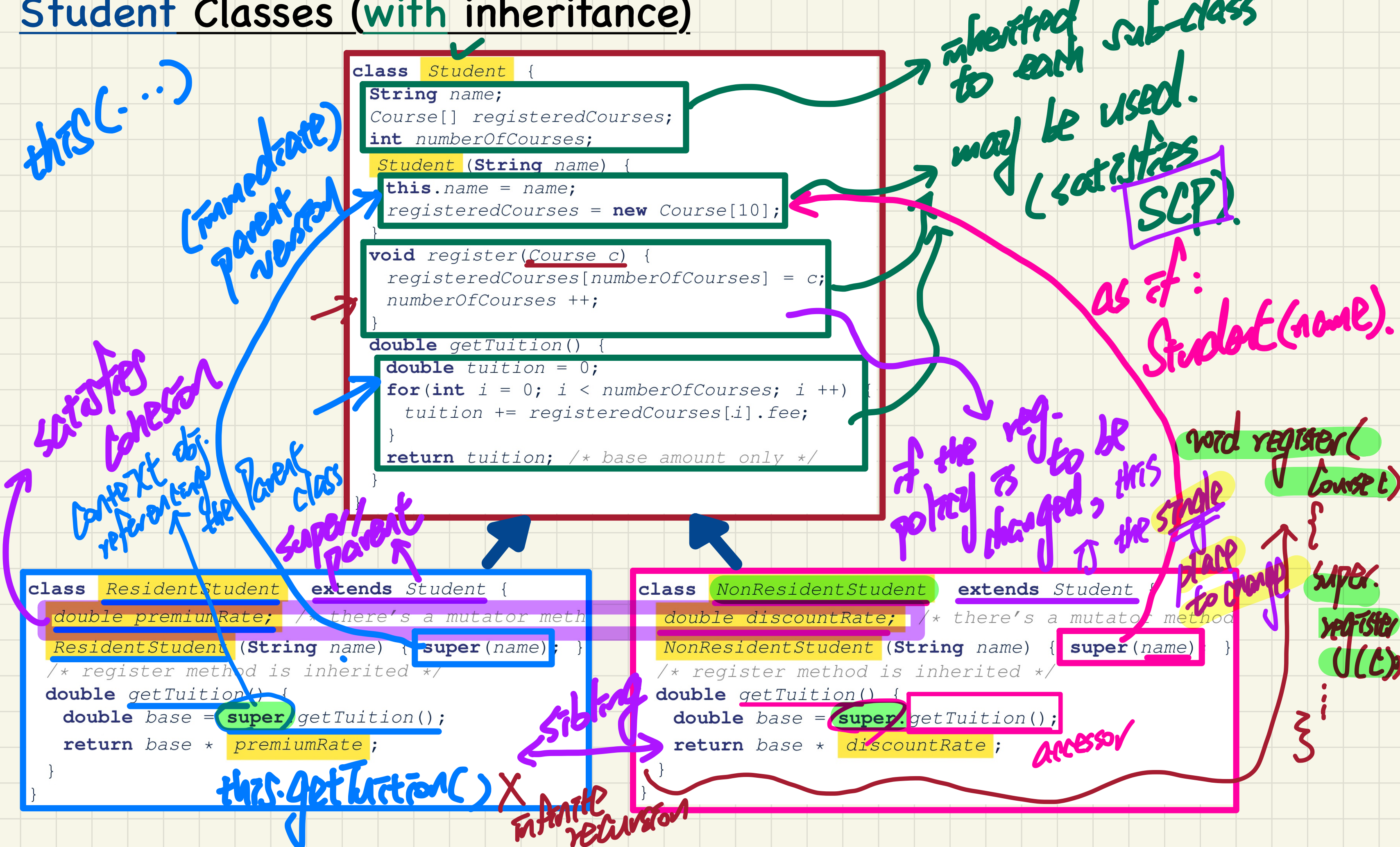
# Visibility: Attributes and Methods



# 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;
}
}
inherited to each sub-class may be used. (satisfies SCP)
as if: Student(name).
this(...)
(immediate) parent version
satisfies cohesion
context obj. referencing the parent class
super/parent
sibling
this.getTuition() X
infinite recursion
accessor
if the policy of reg. is to be changed, this is the single place to change
void register(Course c) { super.register(c); ... }

# Visualizing Parent and Child Objects

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

s → **Student**

| name | → "Stella" |
|---|---|
| numberOfCourses | 0 |
| registeredCourses | → [0: null, 1: null, …, 8: null, 9: null] |

rs → **ResidentStudent**

| name | → "Rachael" |
|---|---|
| numberOfCourses | 0 |
| registeredCourses | → [0: null, 1: null, …, 8: null, 9: null] |
| premiumRate | |

nrs → **NonResidentStudent**

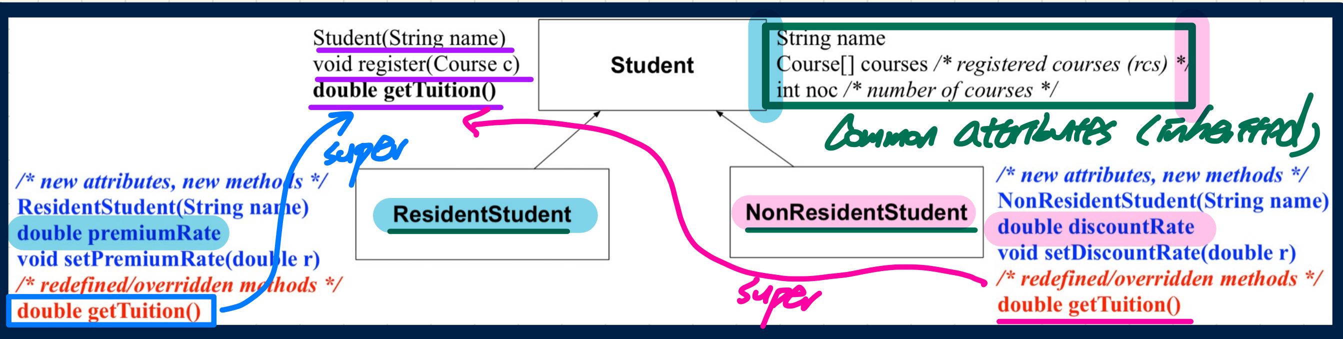
| name | → "Nancy" |
|---|---|
| numberOfCourses | 0 |
| registeredCourses | → [0: null, 1: null, …, 8: null, 9: null] |
| discountRate | |

≥ → child instance has at least as many attributes as those of its parent

→ inherited from the parent Student class counterparts.

specific to sub classes

# 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 → [0 1 ... 9] | pr 1.25

c1 → Course | title 2030 | fee 500

c2 → Course | title 3311 | fee 500

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

# 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 */
  }
}
```

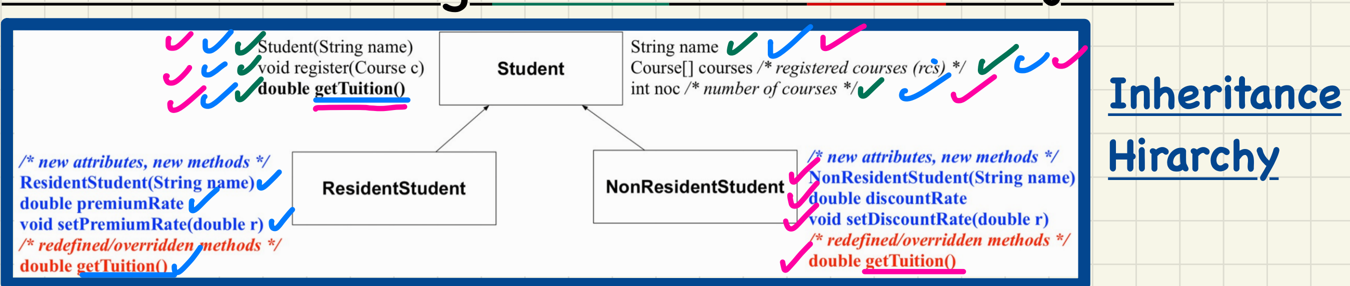
overridden

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;
  }
}
```

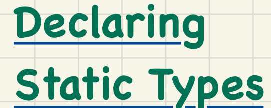
```
class NonResidentStudent extends Student {
  double discountRate; /* there's a mutator method
  NonResidentStudent (String name) { super(name); }
  /* register method is inherited */
  double getTuition() {
    double base = super.getTuition();
    return base * discountRate;
  }
}
```

# Recall: Visualizing Parent and Child Objects




Inheritance Hirarchy

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




declared types (static types) → determines what attributes/methods are available for use in the class.

expectation

Runtime Object Structure

| Student | |
|---|---|
| name | |
| numberOfCourses | 0 |
| registeredCourses | |

s → "Stella"; registeredCourses → | 0: null | 1: null | ... | 8: null | 9: null |

| ResidentStudent | |
|---|---|
| name | |
| numberOfCourses | 0 |
| registeredCourses | |
| premiumRate | |

rs → "Rachael"; registeredCourses → | 0: null | 1: null | ... | 8: null | 9: null |

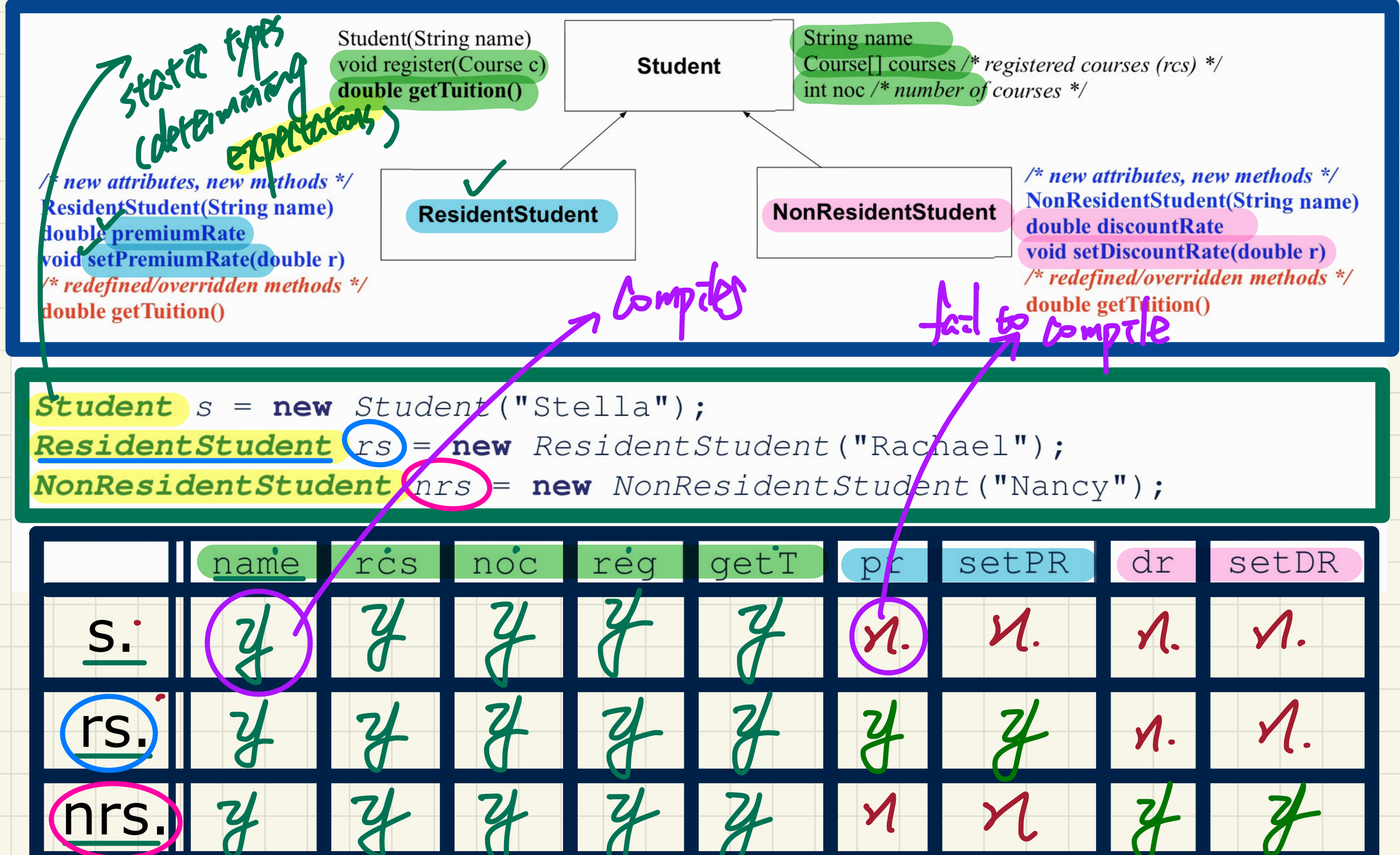
| NonResidentStudent | |
|---|---|
| name | |
| numberOfCourses | 0 |
| registeredCourses | |
| discountRate | |

nrs → "Nancy"; registeredCourses → | 0: null | 1: null | ... | 8: null | 9: null |

inherited

sub-class specific attributes.

# Student Classes (with inheritance): Expectations




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

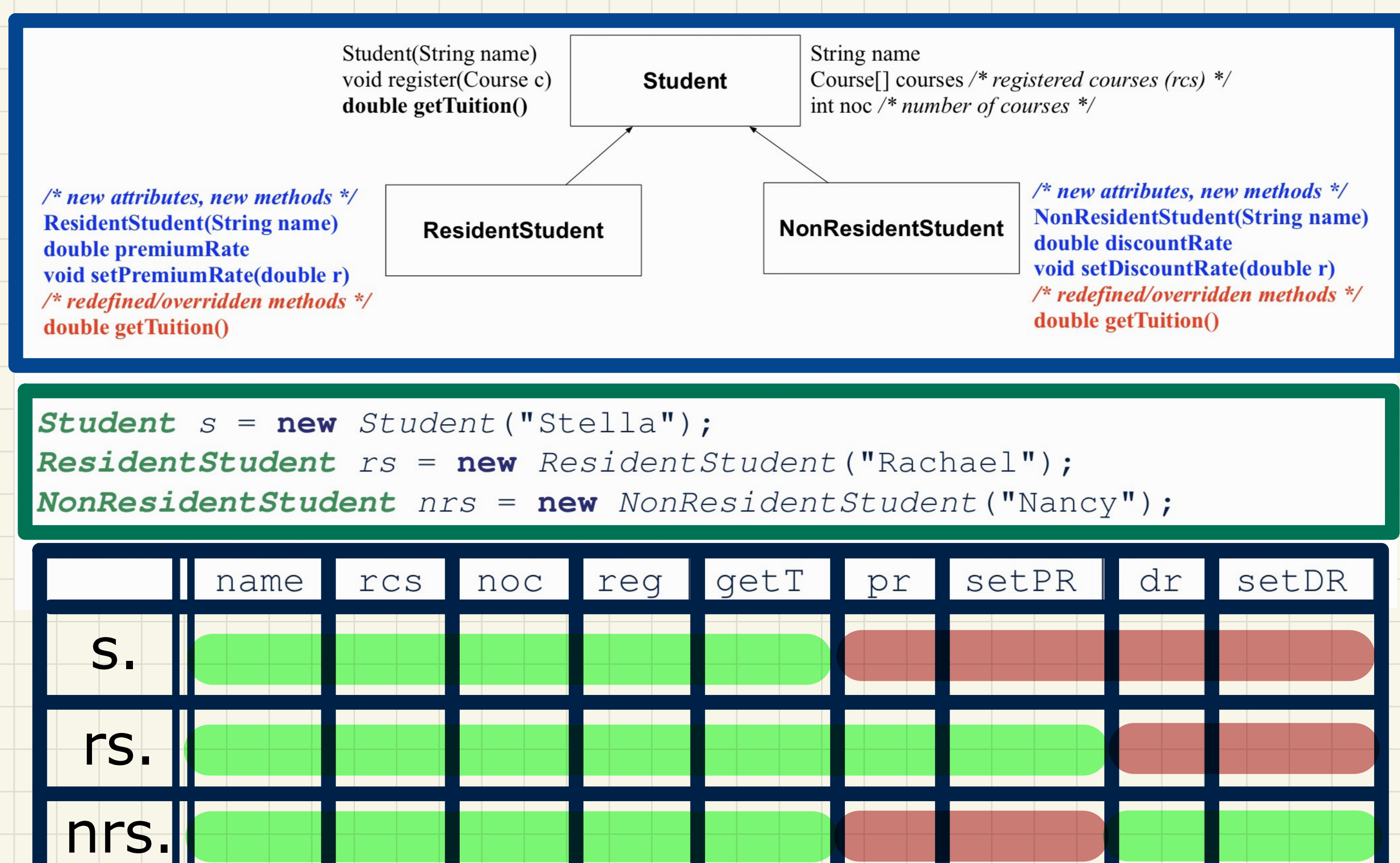
| | name | rcs | noc | reg | getT | pr | setPR | dr | setDR |
|---|---|---|---|---|---|---|---|---|---|
| s. | y | y | y | y | y | n. | n. | n. | n. |
| rs. | y | y | y | y | y | y | y | n. | n. |
| nrs. | y | y | y | y | y | n | n | y | y |

## 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 */

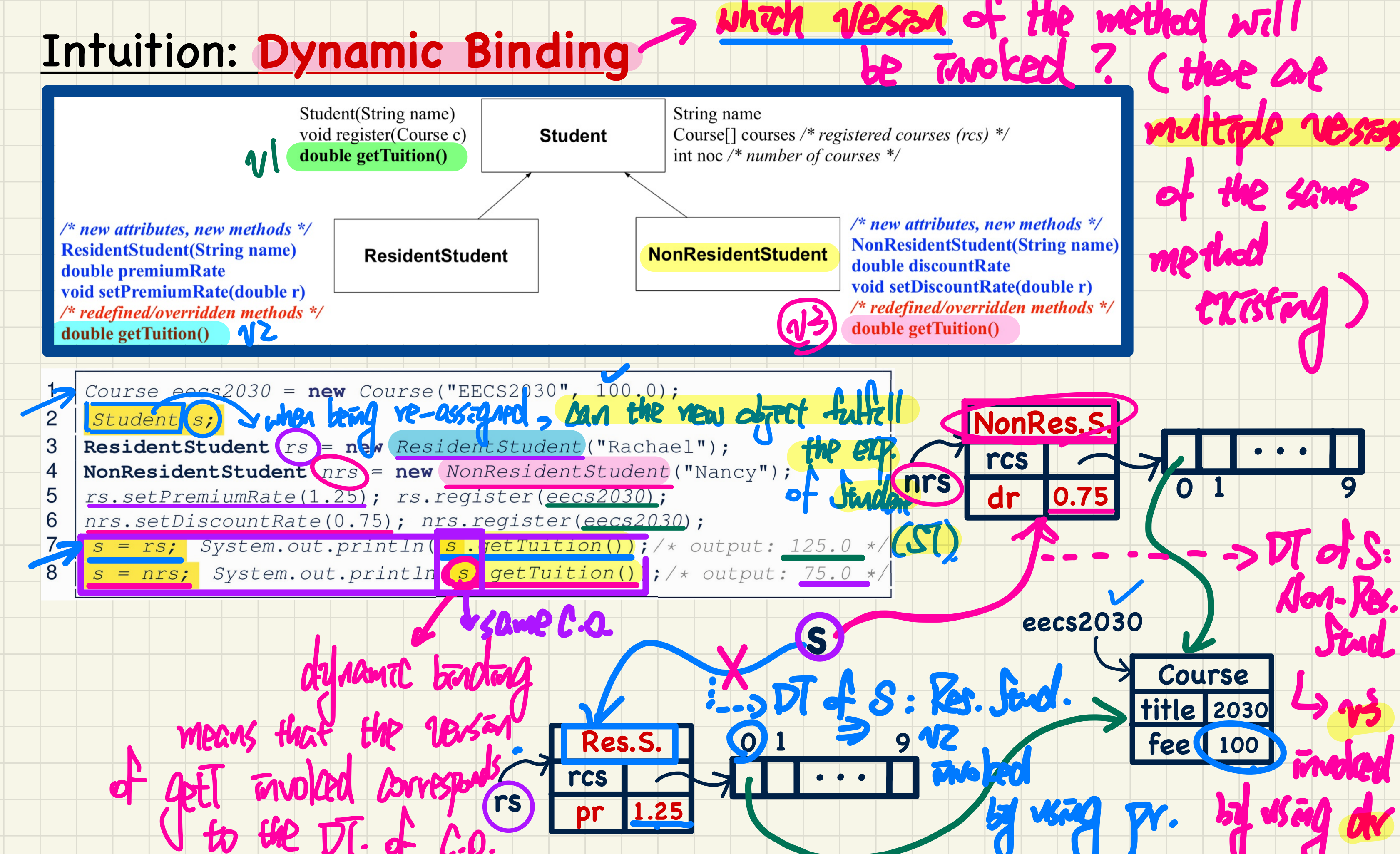
**ResidentStudent**

/* 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()

```
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? */
```



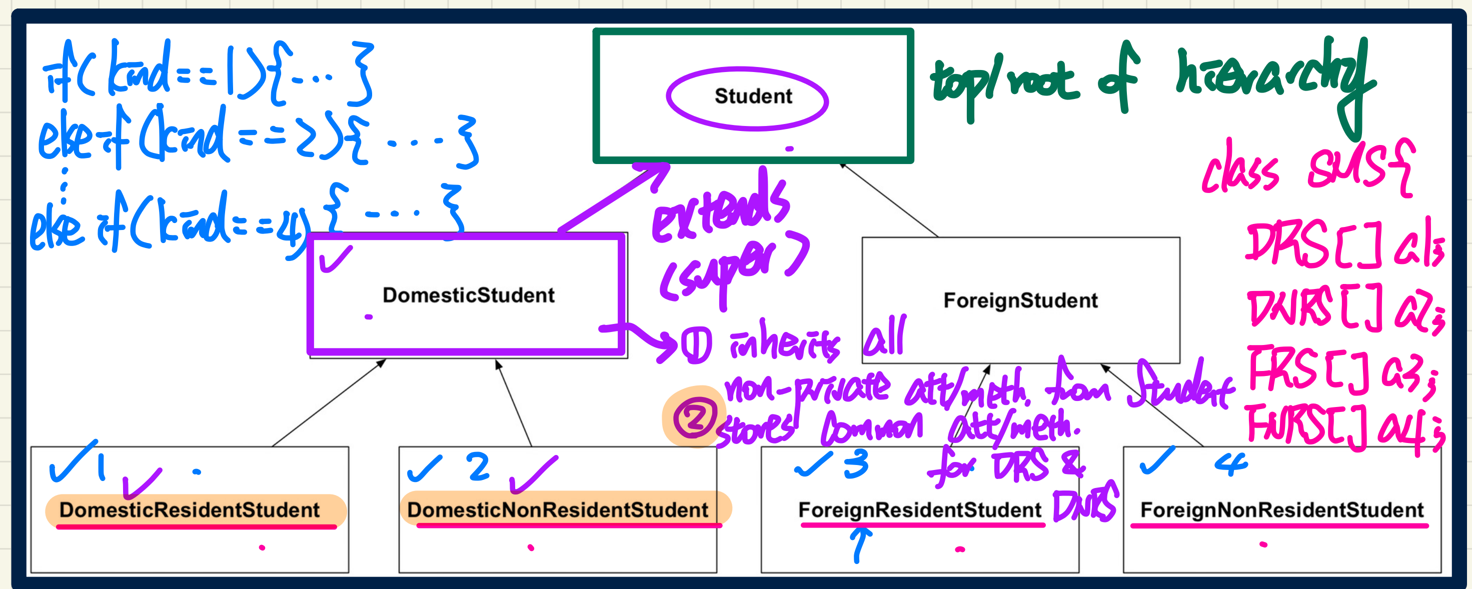
Intuition: Dynamic Binding
Student(String name)
void register(Course c)
double getTuition()
Student
String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */
/* new attributes, new methods */
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double 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 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 */

# Lecture 5

## Part G

### *Inheritance -*
### *Type Hierarchy Formed by Inheritance*

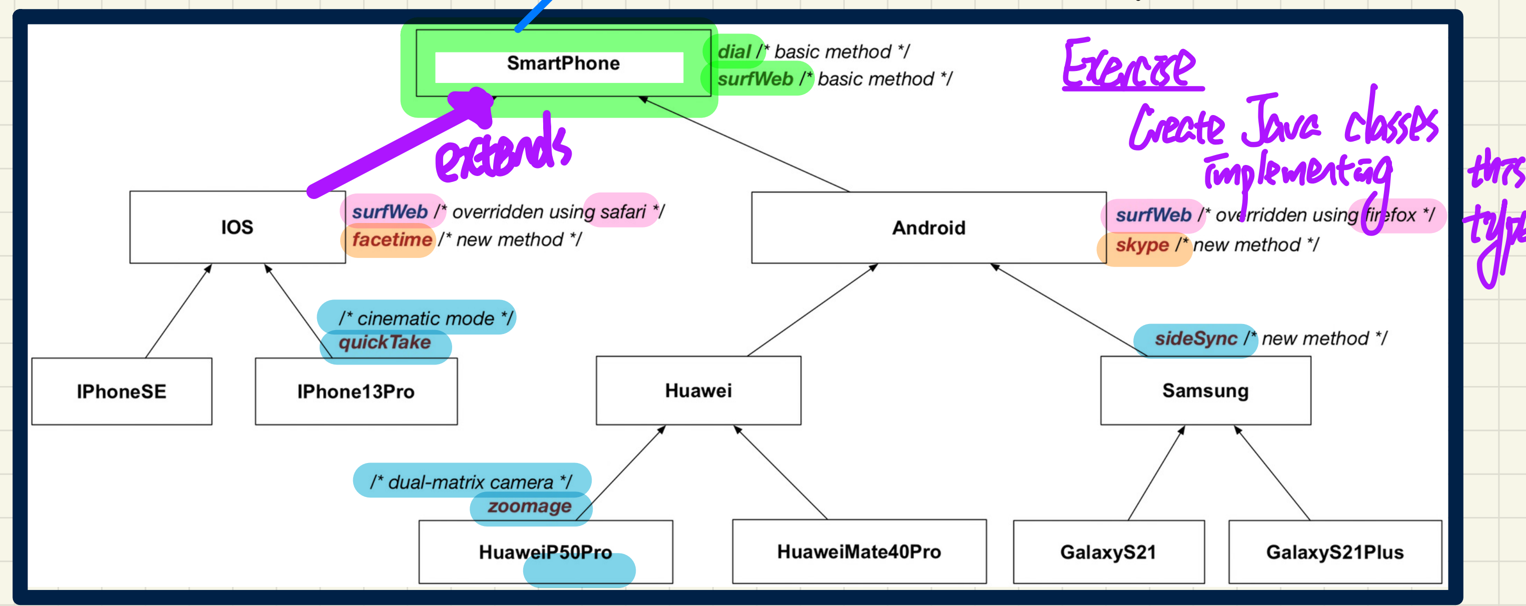
# Multi-Level Inheritance Hierarchy: Students




**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?

# Multi-Level Inheritance Hierarchy: Smartphones




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

**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

higher

lower

more code inherited from ancestors

↳ wider exp.

ancestors of A

descendants of A

root / top of hierarchy.

m1

m2

m3

m4

A

exp: m1

exp.: m1, m2

exp: m1, m2, m3

exp.: m1, m2, m3, m4

# Inheritance Accumulates Code for Reuse