Lecture 21 - Nov. 21

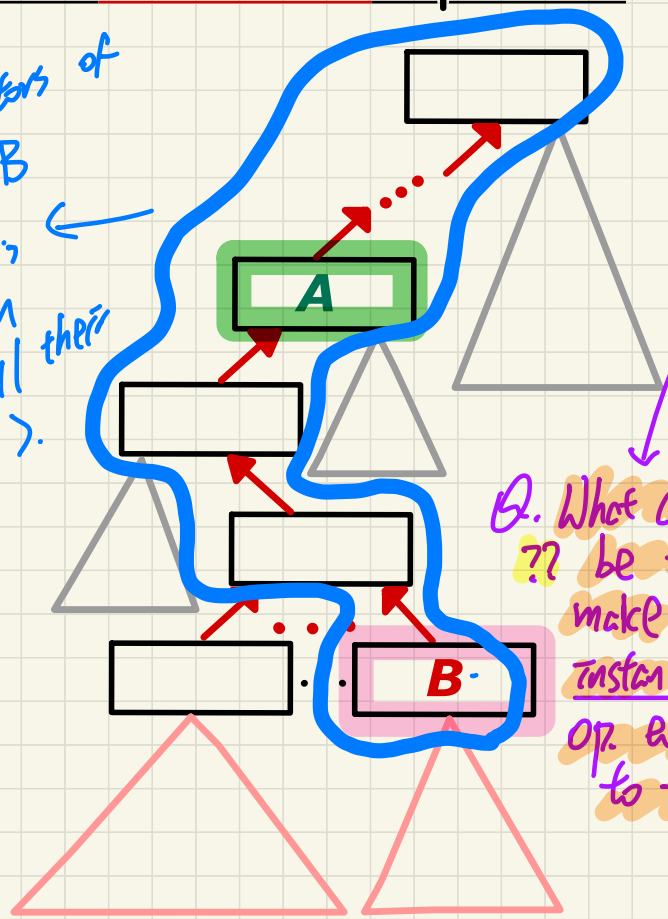
Inheritance

Polymorphic Arrays
Polymorphic Return Values
Type-Checking Rules
Solving Problems Recursively

Announcements/Reminders

- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- **ProgTest3** grading process to start on Monday
- **Exam** Review Sessions eClass Polling
- **Bonus** Opportunity coming: Formal Course Evaluation

The instanceof Operator




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

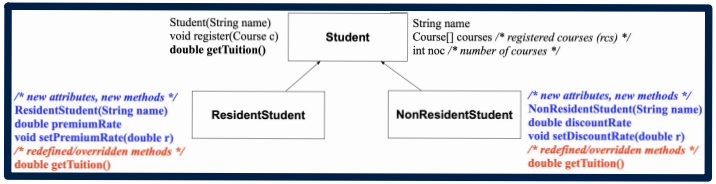
Q. What can ?? be to make instanceof op. eval to true?

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

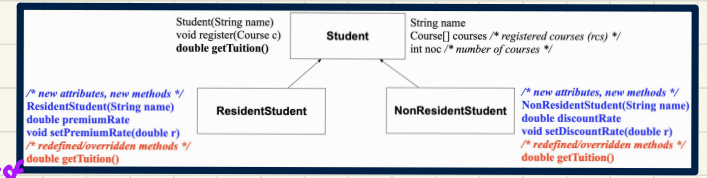
Polymorphic Parameters (2)

```
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);
```



Casting Arguments




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

sms.addRS((ResidentStudent) s) compiles? (downward cast)

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

ClassCastException?

YES

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

ClassCastException?

YES

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

ClassCastException?

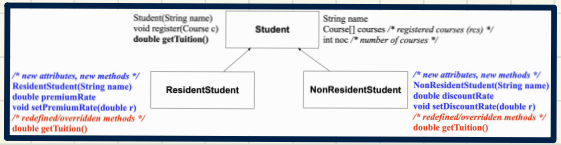
No CCE.

sms.addRS((ResidentStudent) nrs) compiles? No! ✓ neither upward nor downward.

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

ST: 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 }
```

0 1

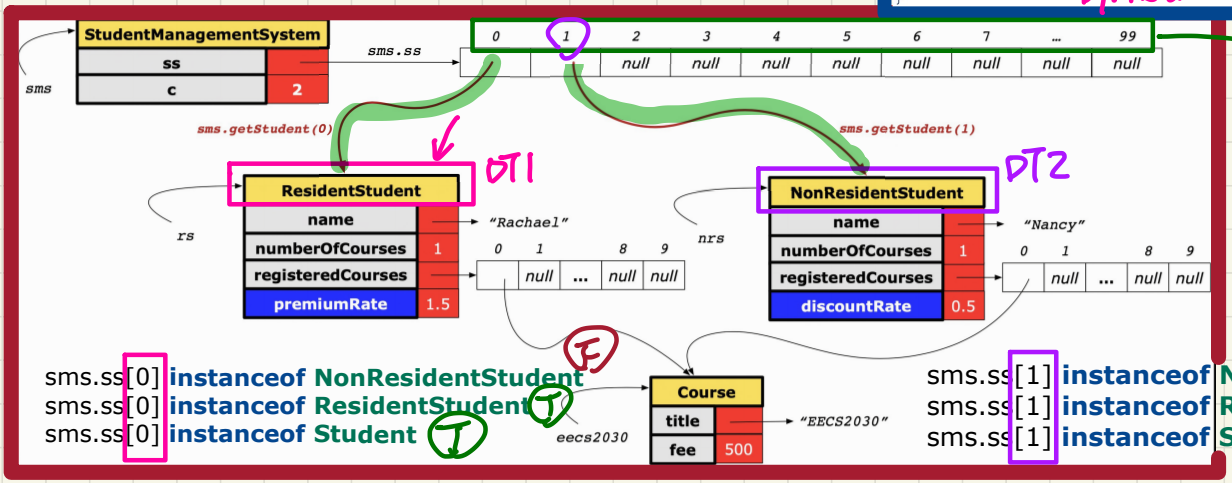
```
class StudentManagementSystem {
  Student[] students;
  int numOfStudents;

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

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

ST of ss[0], ss[1], ...

any arg. of ST that's descendant of Student can be passed (polymorphism).

ST: Student

DT: RS or NRS (descendants of ST)



ss[1] has ST: Student

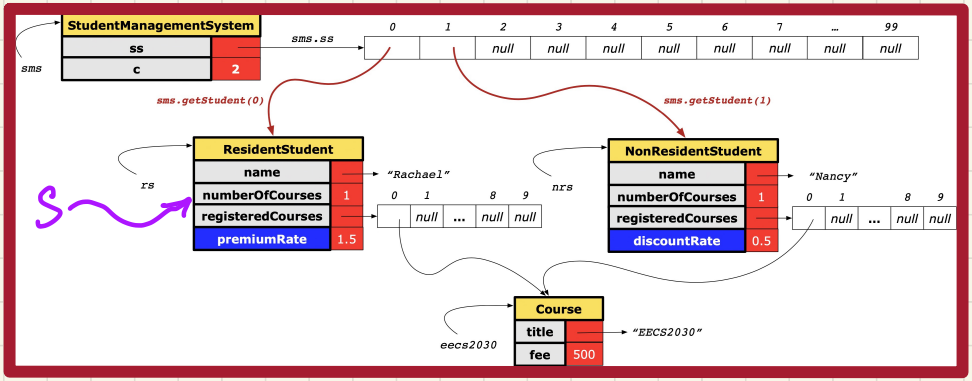
Polymorphic Return Types

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

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

polymorphic query
⇒ s may be assigned an object whose DT is a descend. of Student
s = ss[0];

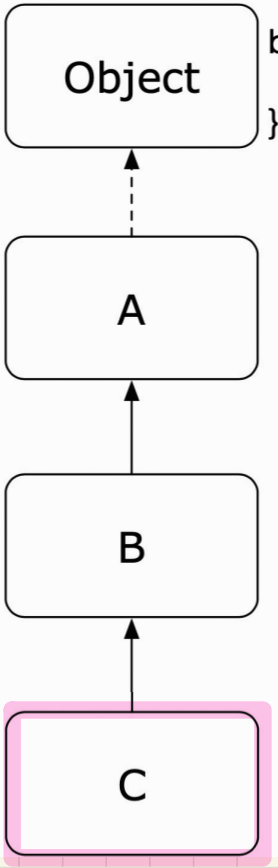
static type of return value
↳ polymorphism: DT of RV can be any of the descendants of Student



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)



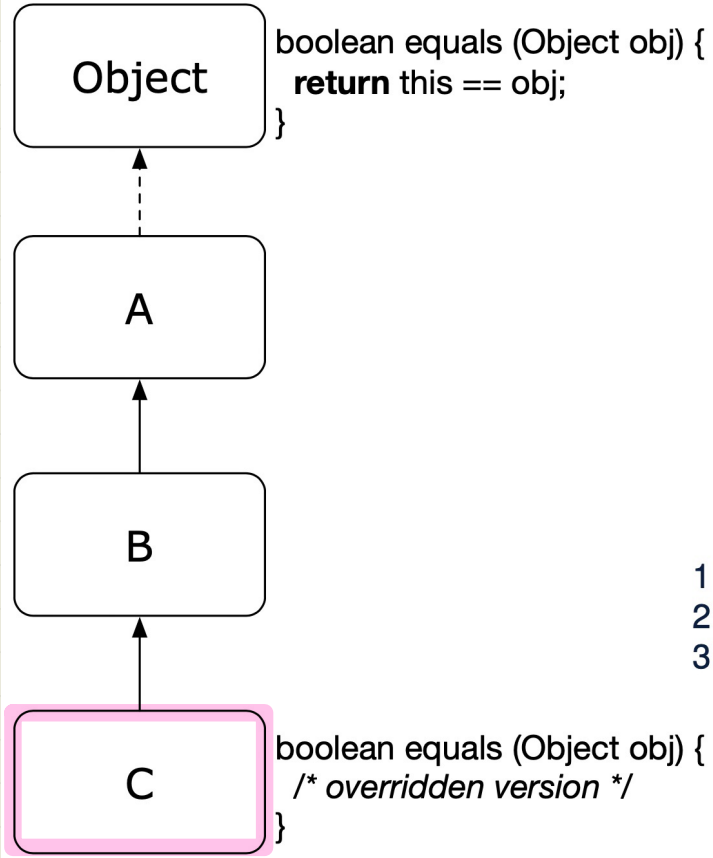

```
boolean equals (Object obj) {
  return this == obj;
}
```

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

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

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

Overridden Methods and Dynamic Binding (2)

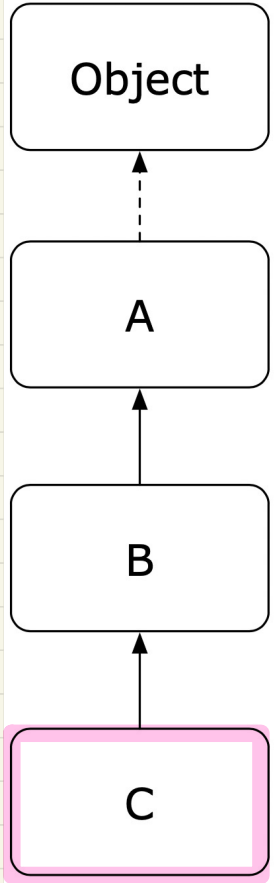



```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
```

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

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

Overridden Methods and Dynamic Binding (3)




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

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

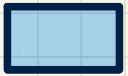
```
class A {
  /*equals not overridden*/
}
class B extends A {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
class C extends B {
  /*equals not overridden*/
}
```

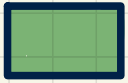
```
1 Object c1 = new C();
2 Object c2 = new C();
3 println(c1.equals(c2));
```

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

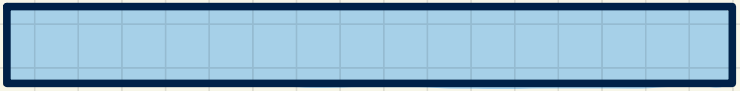
Solving a Problem Recursively

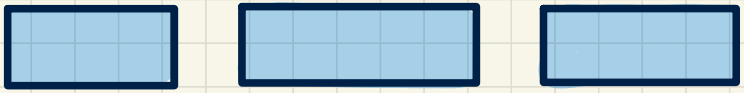
Base Case

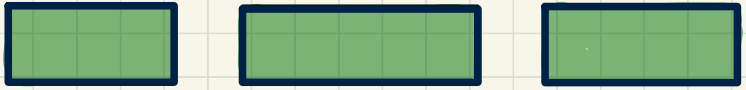
Given a small problem: 

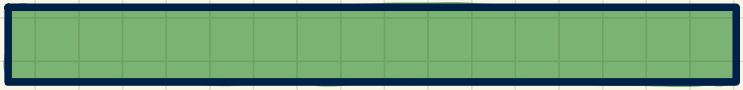
Solve it directly: 

Recursive Cases

Given a big problem: 

Divide it into smaller problems: 

Assume solutions to smaller problems: 

Combine solutions to smaller problems: 

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