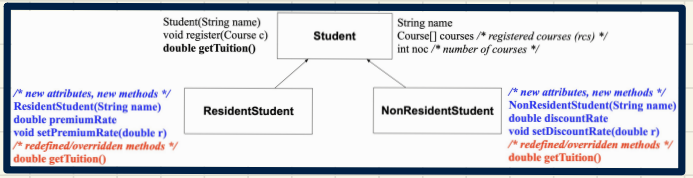
Casting Arguments



void addRS(ResidentStudent rs)

sms.addRS((ResidentStudent) s) compiles?

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

ClassCastException?

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

ClassCastException?

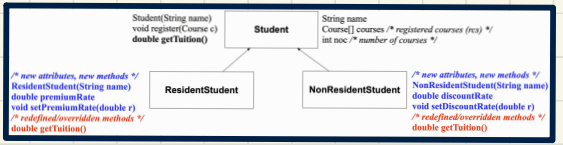
```
1 Student s = new ResidentStudent("Rachael");
2 /* s' ST: Student; s' DT: ResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
```

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

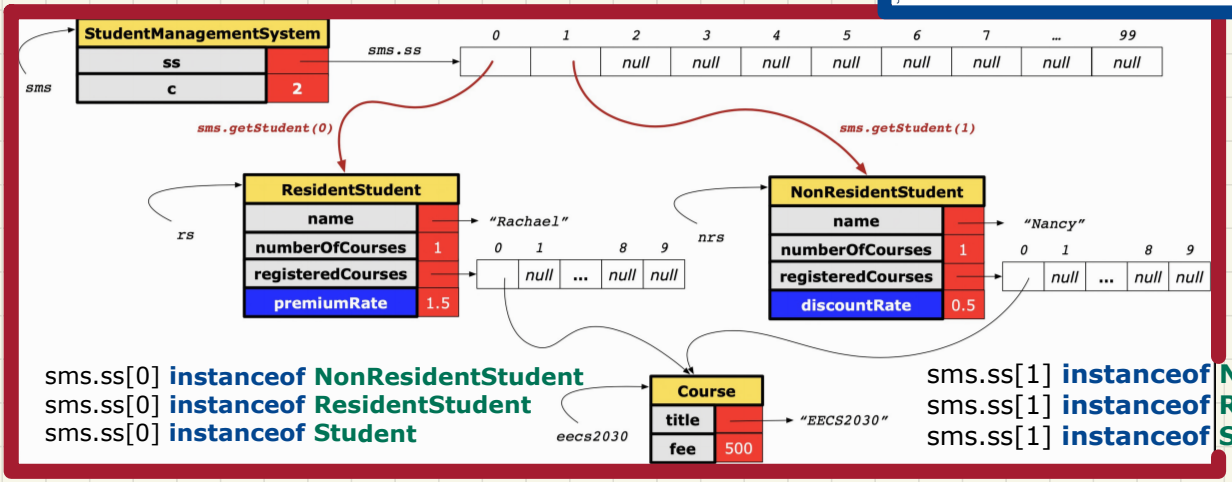



```
1  ResidentStudent rs = new ResidentStudent("Rachael");
2  rs.setPremiumRate(1.5);
3  NonResidentStudent nrs = new NonResidentStudent("Nancy");
4  nrs.setDiscountRate(0.5);
5  StudentManagementSystem sms = new StudentManagementSystem();
6  sms.addStudent( rs ); /* polymorphism */
7  sms.addStudent( 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)
    }
  }
}
```



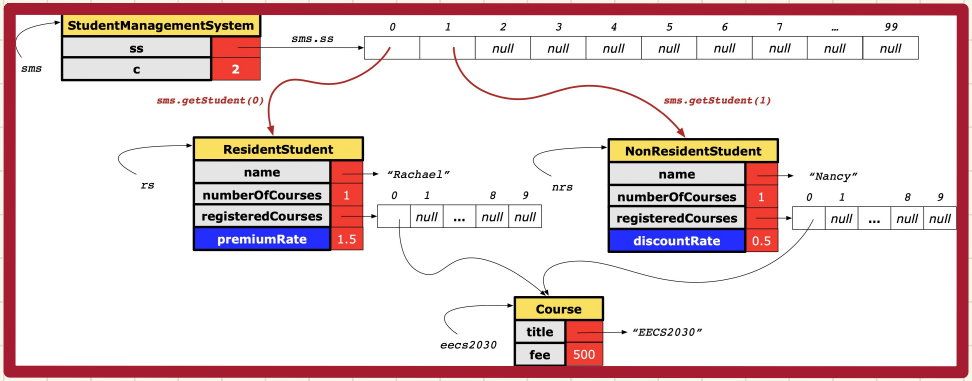
sms.ss[0] **instanceof** **NonResidentStudent**
sms.ss[0] **instanceof** **ResidentStudent**
sms.ss[0] **instanceof** **Student**

sms.ss[1] **instanceof** **NonResidentStudent**
sms.ss[1] **instanceof** **ResidentStudent**
sms.ss[1] **instanceof** **Student**

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); ×
```

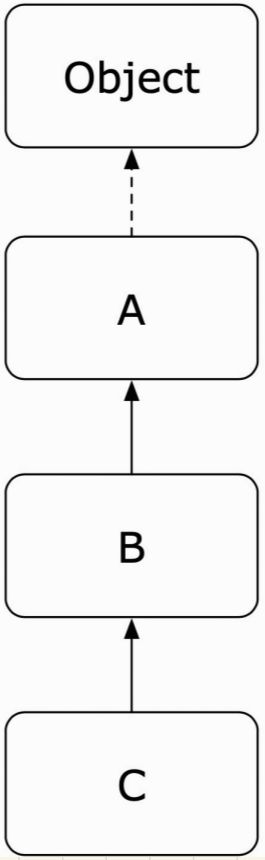
```
class StudentManagementSystem {
  Student[] ss; int c;
  void addStudent(Student s) { ss[c] = s; c++; }
  Student getStudent(int i) {
    Student s = null;
    if(i < 0 || i >= c) {
      throw new IllegalArgumentException("Invalid
    }
    else {
      s = ss[i];
    }
    return s;
  } }
```



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)



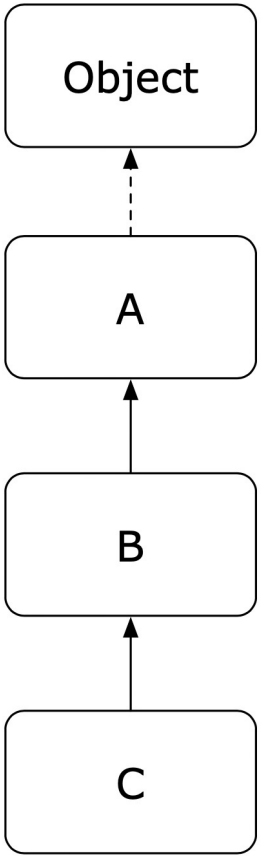

```
boolean equals (Object obj) {
  return this == obj;
}
```

```
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`?

Overridden Methods and Dynamic Binding (2)




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

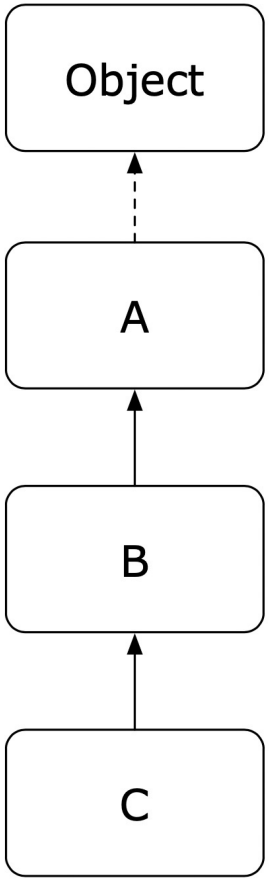
```
boolean equals (Object obj) {
  /* overridden version */
}
```

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

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

L3 calls which version of `equals`?

Overridden Methods and Dynamic Binding (3)




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

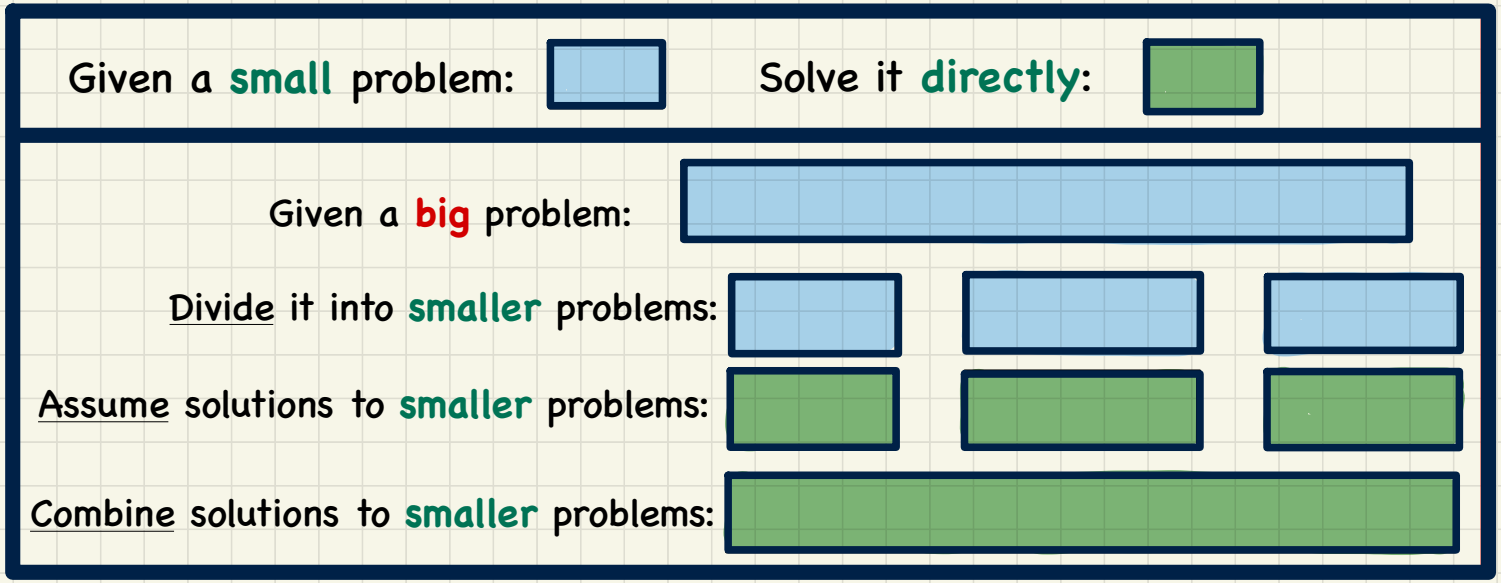
```
boolean equals (Object obj) {
  /* overridden version */
}
```

```
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`?

Solving a Problem Recursively



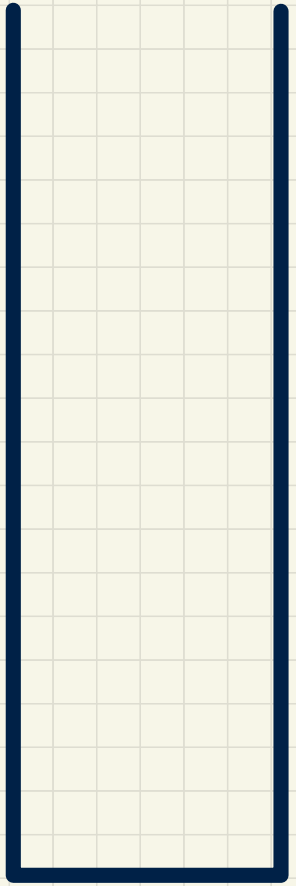

```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

Tracing Recursion via a Stack

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

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

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



Runtime Stack

Recursive Solution: factorial

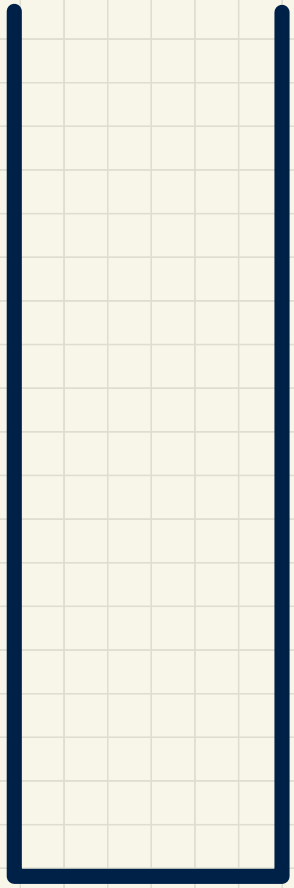
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

Recursive Solution in Java: factorial

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

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

Example: factorial(3)



Runtime Stack

Common Errors of Recursion (1)

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

Common Errors of Recursion (2)

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