Monday Nov. 12

Lecture 18

- Lab Test 3: Nov. 19
- Guide & Exercises

Anatomy of a Type Cast

Student s = new ResidentStudent ("Rachael");

→ ST : Student

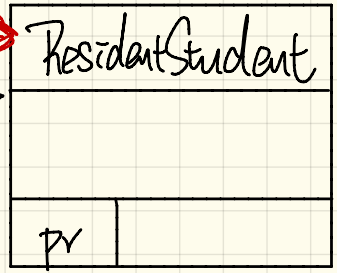
DT : RS descendant

RS

s

Student

s



ResidentStudent

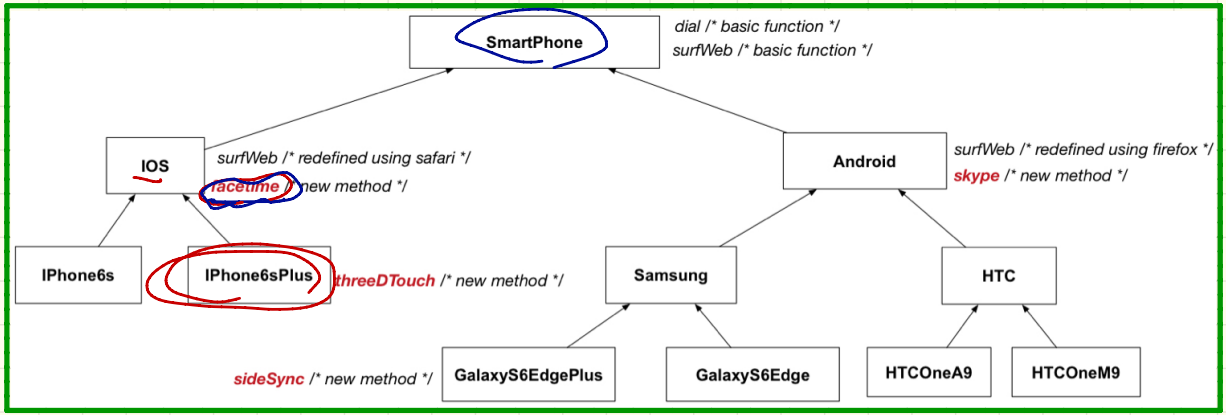
ST

rs = (ResidentStudent) s;

ST: Student

a new ref of ST RS

Type Casts



Named Cast

```
SmartPhone aPhone = new IPhone6sPlus();
IOS forHeeyeon = (IPhone6sPlus) aPhone;
forHeeyeon.facetime();
```

aPhone.facetime() ? ✗

Anonymous Cast

```
SmartPhone aPhone = new IPhone6sPlus();
((IPhone6sPlus) aPhone).facetime();
```

SP

Problem?

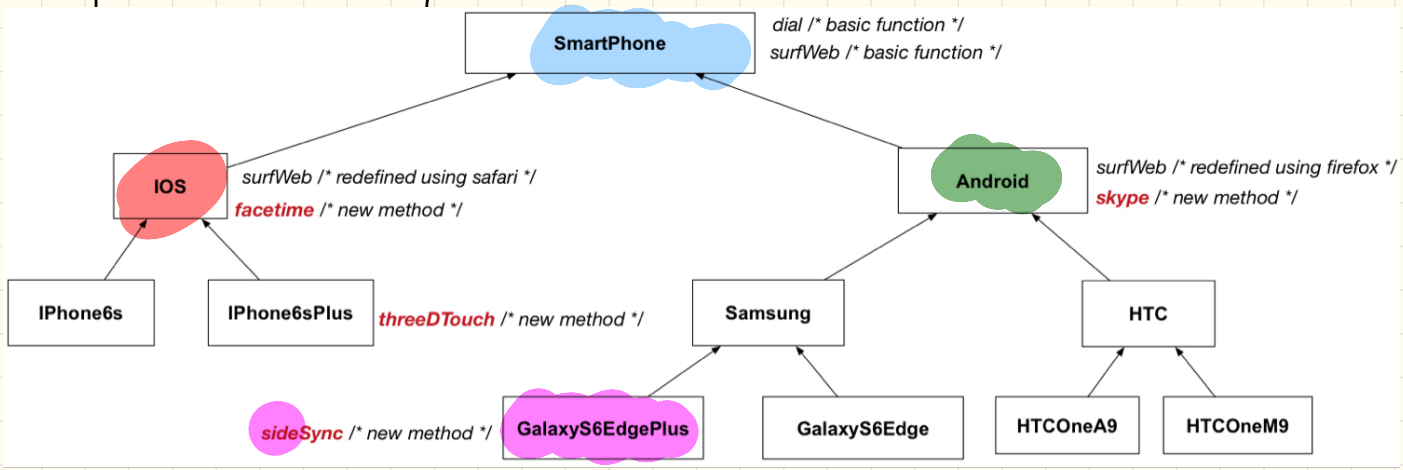
```
1 SmartPhone aPhone = new IPhone6sPlus();
2 ((IPhone6sPlus) aPhone).facetime();
```

(2) (1)

→ (IP6sPlus) (aPhone.facetime()) ✗

(IP6sPlus) aPhone.facetime() ✓

Compilable Cast : Upward vs. Downward

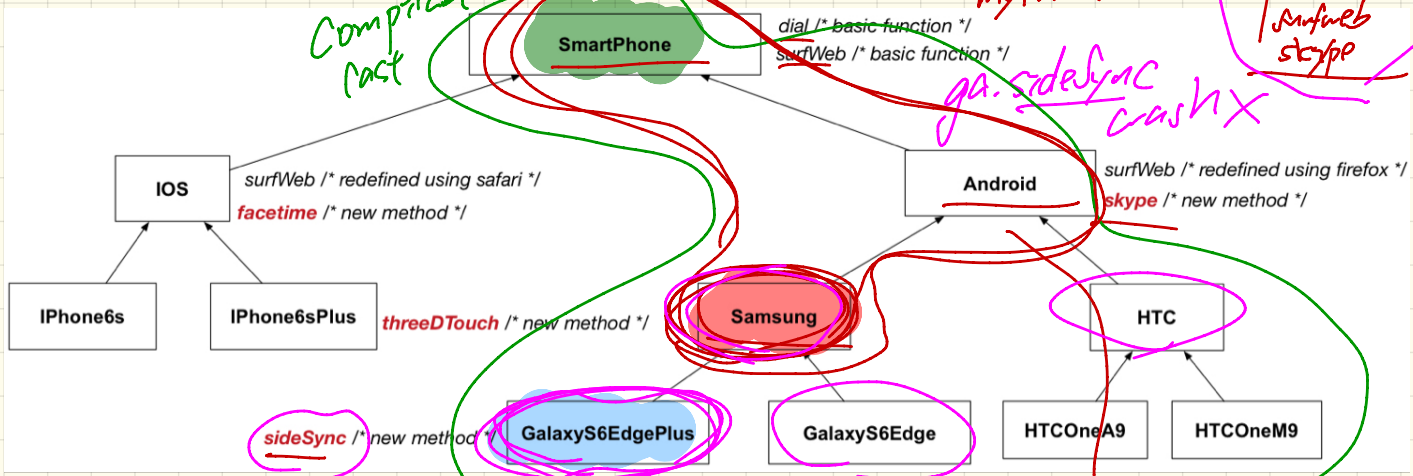



```
Android myPhone = new GalaxyS6EdgePlus();
SmartPhone sp = (SmartPhone) myPhone;
GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
```

EXPECTATIONS

	myPhone	sp	ga
dial			
surfWeb			
facetime			
threeDTouch			
skype			
sideSync			

Compilable Cast May Fail at Runtime (2)



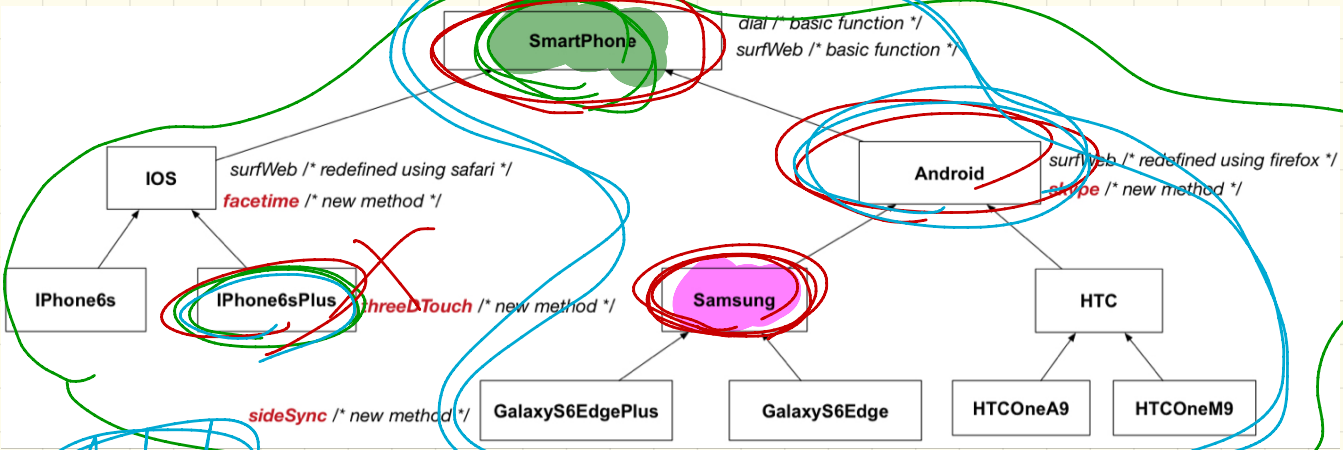

```
SmartPhone myPhone = new Samsung();
GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
```

assume no ClassCastException → invalid :(there's a CCE.

ST: SP

compilable and no CCE

Compilable Cast May Fail at Runtime (3)



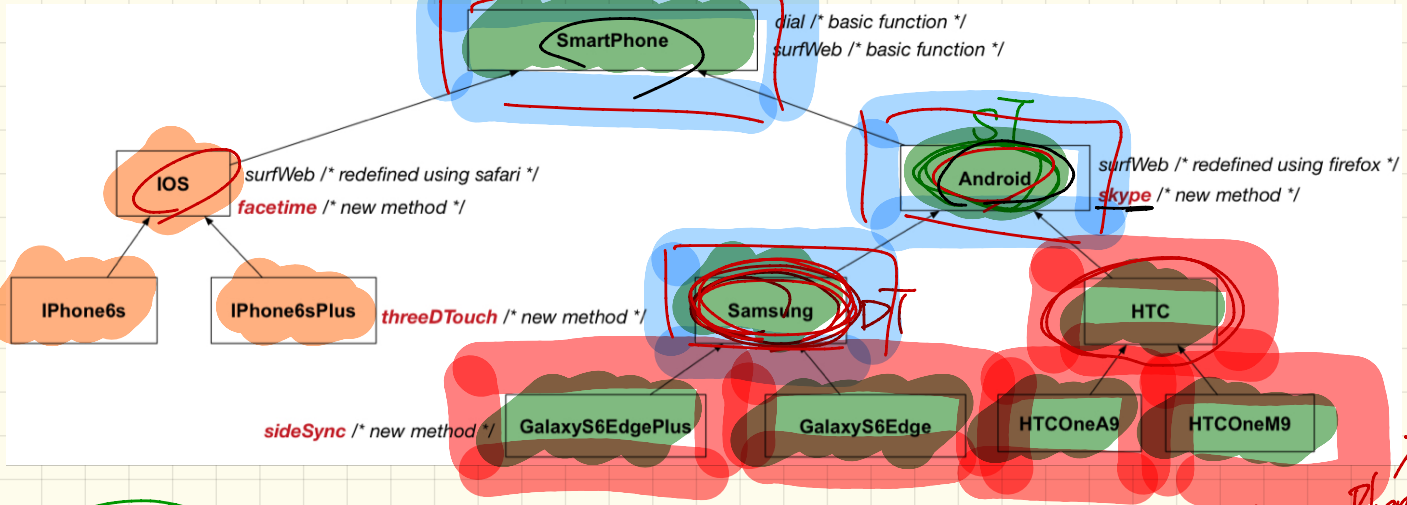
Android

SmartPhone myPhone = new Samsung();

IPhone6sPlus ip = (IPhone6sPlus) myPhone;

compiles but CCE

Compilable Cast vs. Exception-Free Cast




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

IOS iOS = (IOS) myPhone;

X

Compilable Cast

Non-Compilable Cast

Exception-Free Cast

ClassCastException Cast

Type Casts

C oc = new E();

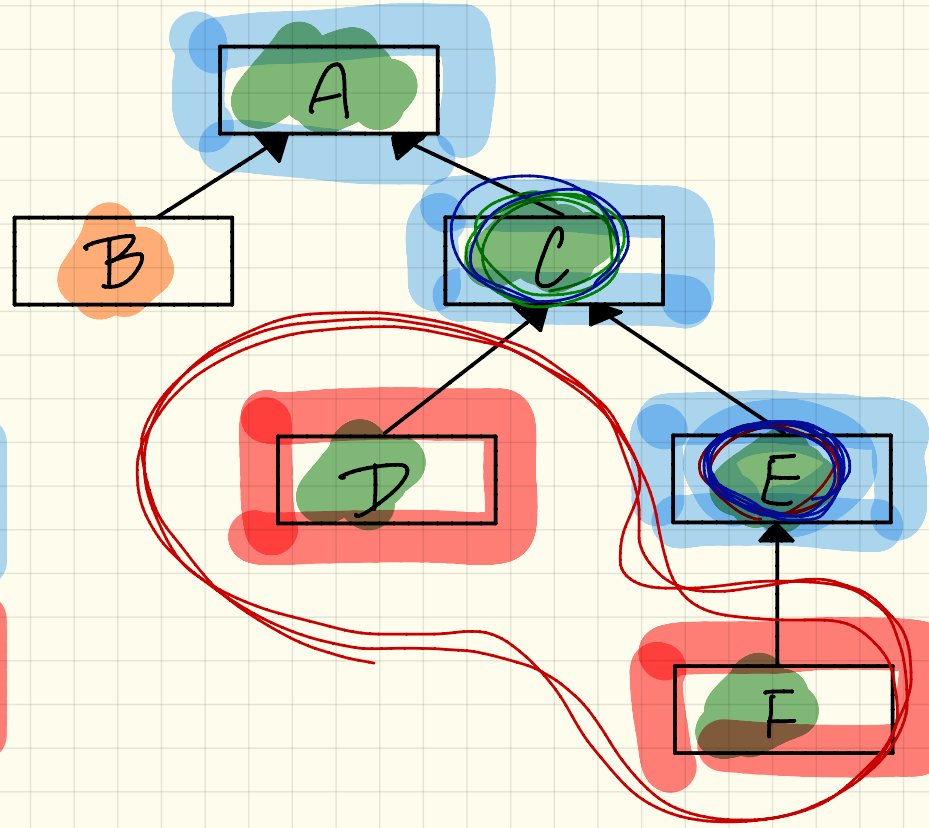
A oa = (A) oc;

E oe = (E) oc;

F of = (F) oc;

D od = (D) oc;

B ob = (B) oc;



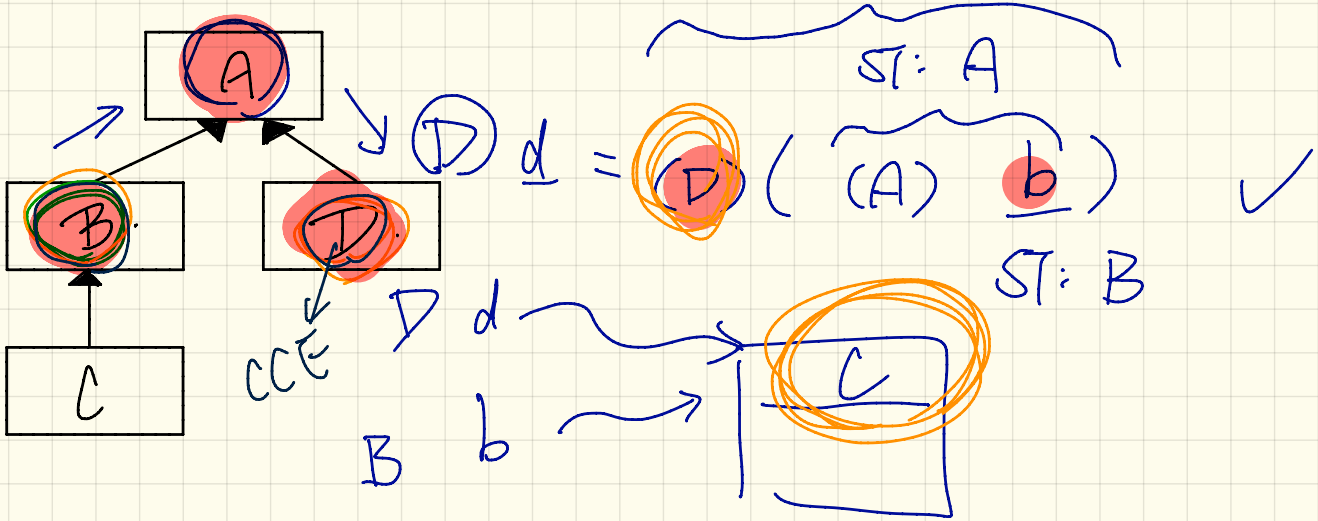
Compilable Cast vs. Exception-Free Cast : Exercise

```
class A { }
class B extends A { }
class C extends B { }
class D extends A { }
```

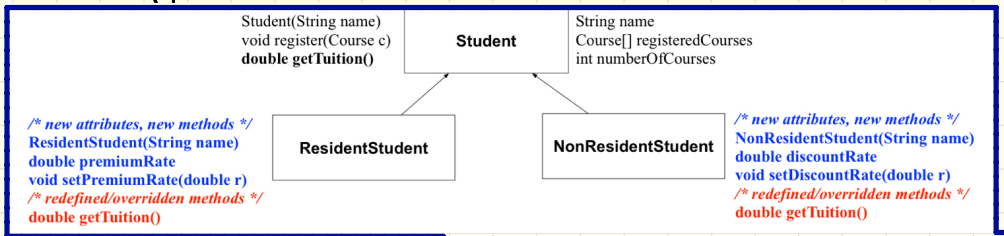
```
1) B b = new C();
2) D d = (D) b;
```

ST: B ✗

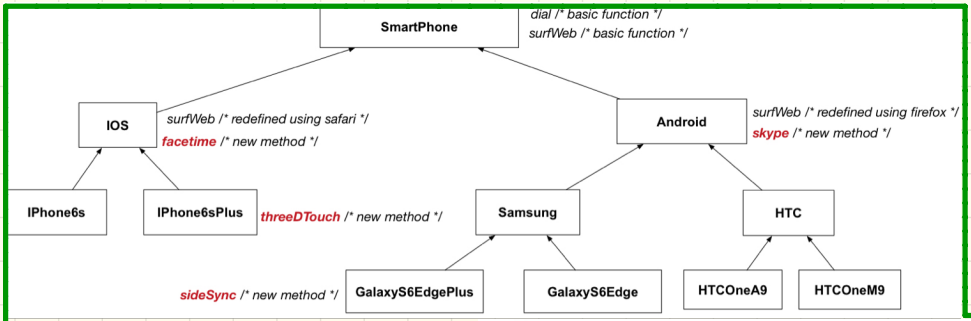
ST: D



Checking Dynamic Types at Runtime

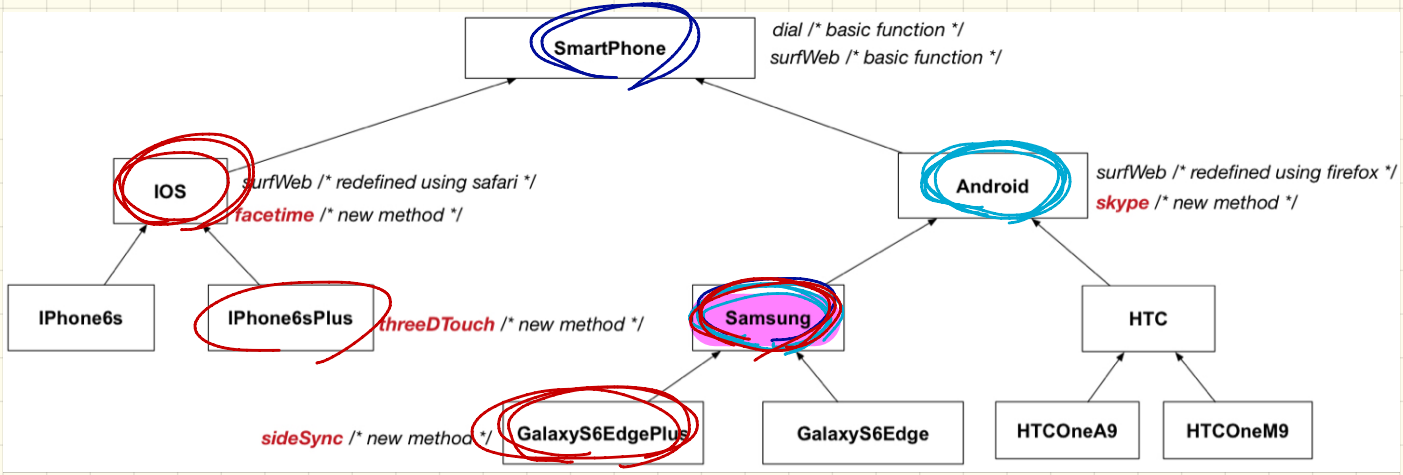



```
1  Student jim = new NonResidentStudent("J. Davis");
2  if (jim instanceof ResidentStudent) {
3    ResidentStudent rs = (ResidentStudent) jim;
4    rs.setPremiumRate(1.5);
5  }
```




```
1  SmartPhone aPhone = new GalaxyS6EdgePlus();
2  if (aPhone instanceof IPhone6sPlus) {
3    IOS forHeeyeon = (IPhone6sPlus) aPhone;
4    forHeeyeon.facetime();
5  }
```

Use of the instanceof Operator

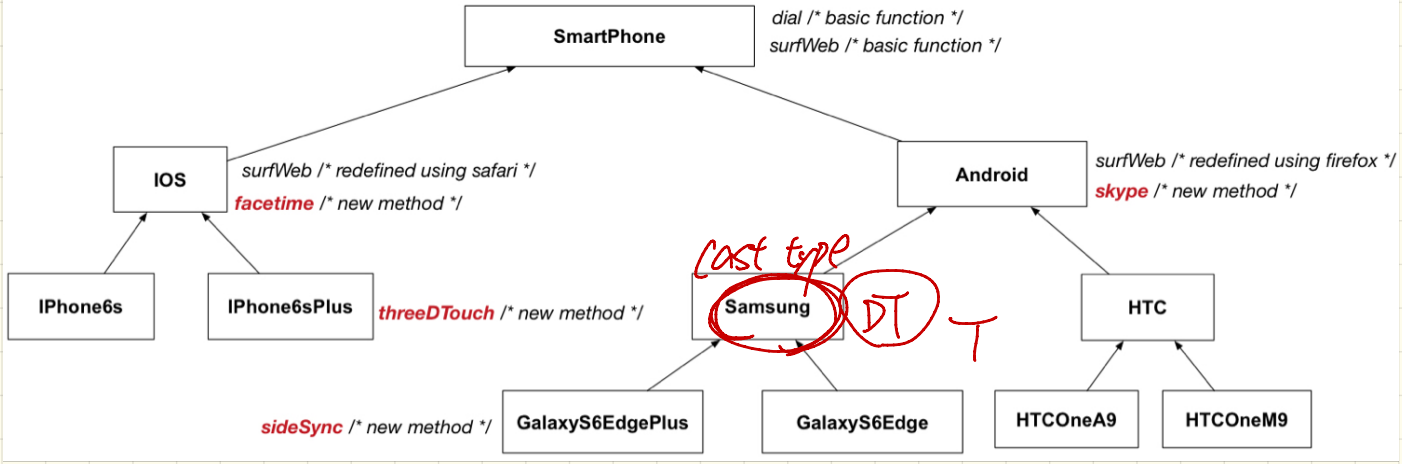



```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android);
/* true ∵ Samsung is a descendant of Android */}
println(myPhone instanceof Samsung);
/* true ∵ Samsung is a descendant of Samsung */}
println(myPhone instanceof GalaxyS6Edge);
/* false ∵ Samsung is not a descendant of GalaxyS6Edge */
println(myPhone instanceof IOS);
/* false ∵ Samsung is not a descendant of IOS */
println(myPhone instanceof IPhone6sPlus);
/* false ∵ Samsung is not a descendant of IPhone6sPlus */
```

myPhone instanceof SmartPhone
True

⇒ GSbEdge o =
(GSbEdge) myPhone;
CCE

Safe Cast via Use of instanceof




```
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 GalaxyS6EdgePlus) {
7    GalaxyS6EdgePlus galaxy = (GalaxyS6EdgePlus) myPhone;
8  }
9  if(myphone instanceof HTC) {
10   HTC htc = (HTC) myPhone;
11 }
```

Polymorphic Arguments (1)

```
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++; } }
```

ST: Student

ST: RS

s.pr

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

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

addRS(RS rs){
  rs.pr
  rs.set_pr
}

ss[0].name

ss[0].pr ✗

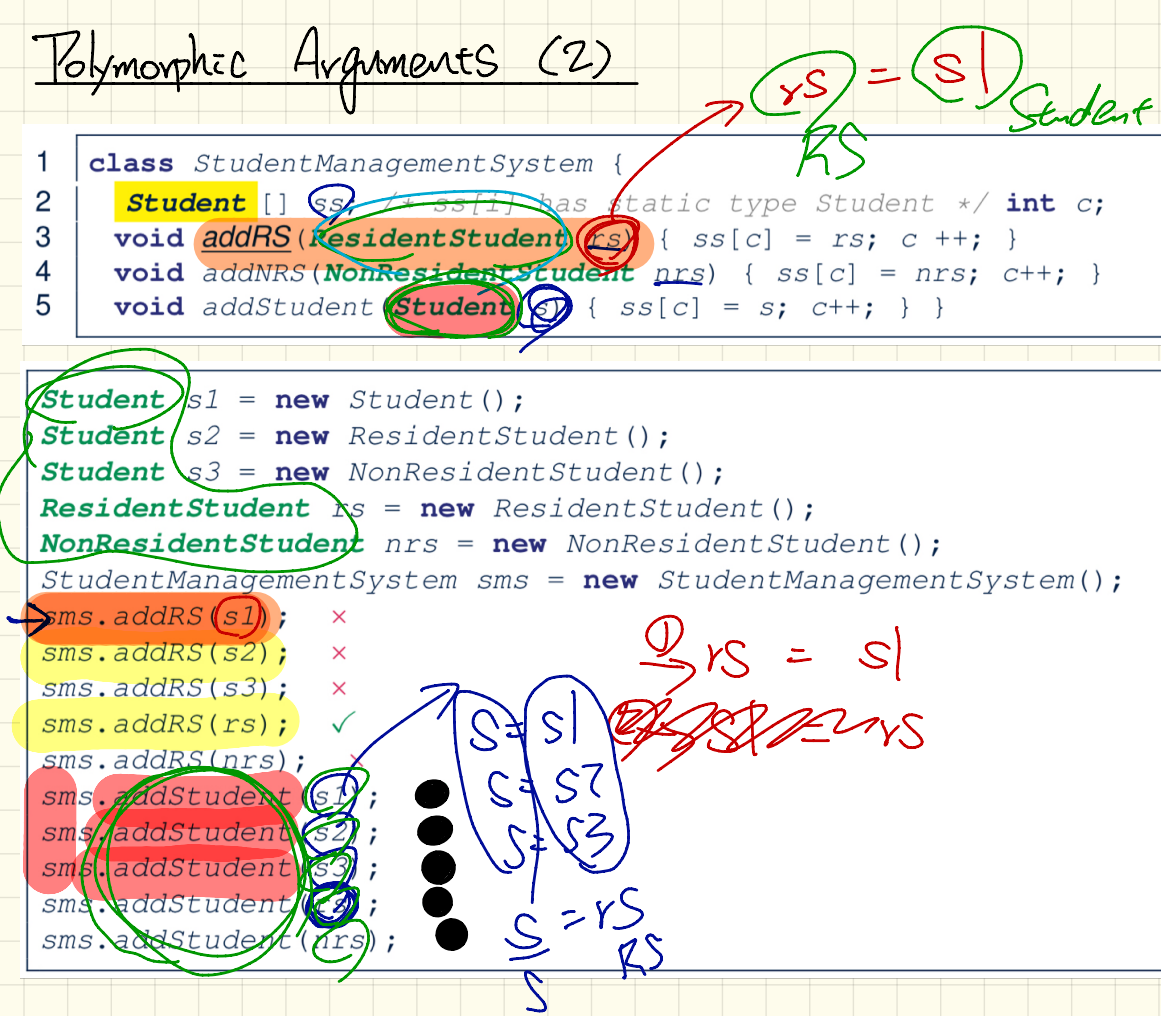
ss[0]: Student

Compile :-(The ST of rs (RS) is a descendant of the ST of ss[c]

Polymorphic Arguments (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);
```



A Polymorphic Collection of Students

ST:S
DT:RS
sms.ss[0].getTuition()

DT:NRS ST:S
sms.ss[1].getTuition()

```
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)
  }
 }
}
```