Inheritance

EECS2030 B & G: Advanced Object Oriented Programming
Fall 2025

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- Alternative designs to **inheritance**
- Using **inheritance** for code reuse
- ***Static Types***, Expectations, ***Dynamic Types***
- *Polymorphism*
  (variable assignments, method arguments & return values)
- *Dynamic Binding*
- ***Type Casting***

Why Inheritance: A Motivating Example

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.

Tasks: Write Java classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

Why Inheritance: A Motivating Example

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.

Tasks: Write Java classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

No Inheritance: `ResidentStudent` Class

```
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a mutator for this */
  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;
  }
}
```

No Inheritance: `NonResidentStudent` Class

```java
public class NonResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double discountRate; /* assume a mutator for this */
  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 ;
  }
}
```

No Inheritance: Testing Student Classes

```java
public class Course {
  private String title; private double fee;
  public Course(String title, double fee) {
    this.title = title; this.fee = fee;
  }
}
```

```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());
  }
}
```

No Inheritance: Issues with the Student Classes

- Implementations for the two student classes seem to work.

  But can you see any potential problems with it?

  Hint. Maintenance of code
- The code of the two student classes share a lot in common.
  - *Duplicates of code make it hard to maintain your software!*
  - This means that when there is a change of policy on the common part, we need modify ***more than one places***.
  - This violates the so-called ***single-choice design principle***.

No Inheritance: Maintainability of Code (1)

What if the way for registering a course changes?

e.g.,

```
public void register(Course c) throws TooManyCoursesException {
  if (this.noc >= MAX_ALLOWANCE) {
    throw new TooManyCoursesException("Too many courses");
  }
  else {
    this.courses[this.noc] = c;
    this.noc ++;
  }
}
```

Changes needed for `register` method in ***both*** student classes!

No Inheritance: Maintainability of Code (2)

What if the way for calculating the base tuition changes?

e.g.,

```
public double getTuition() {
  double tuition = 0;
  for(int i = 0; i < this.noc; i ++) {
    tuition += this.courses[i].fee;
  }
  /* ... can be premiumRate or discountRate */
  return tuition * inflationRate * ...;
}
```

Changes needed for `getTuition` method in ***both*** student classes!

No Inheritance: A Collection of Various Kinds of Students

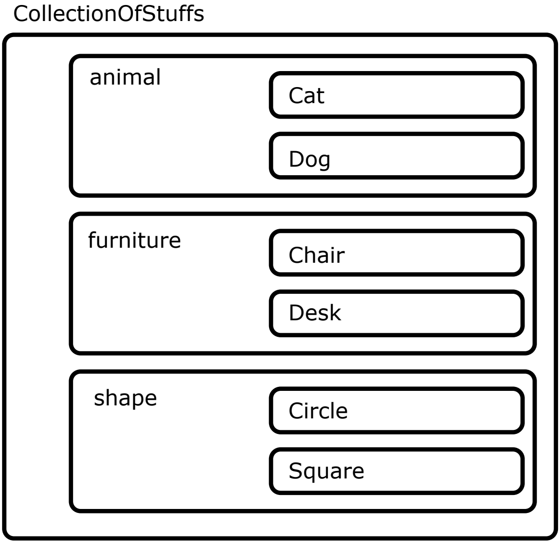
How can we define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
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); }
  }
}
```

But what if we later on introduce ***more kinds of students***?

Very ***inconvenient*** to handle each list of students ***separately***!

Visibility: Project, Packages, Classes



Visibility of Classes

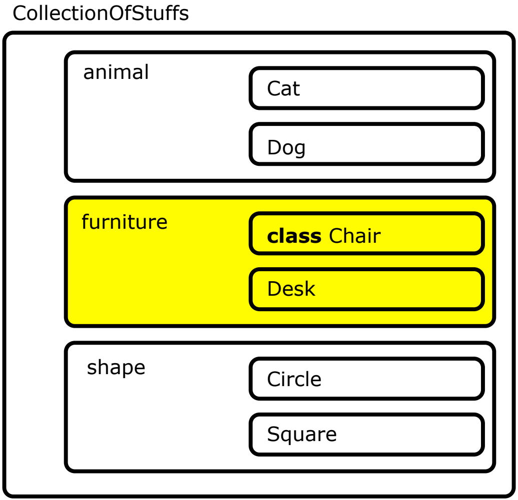
- Only <u>one</u> modifier for declaring visibility of classes: ***public***.
- Use of ***private*** is forbidden for declaring a class.

  e.g., $\boxed{\textit{private}\ \texttt{class}\ \text{Chair}}$ is **not** allowed!!
- **Visibility** of <u>a class</u> may be declared using a <u>modifier</u>, indicating that it is accessible:
  1. Across <u>classes</u> within its residing package [no modifier]

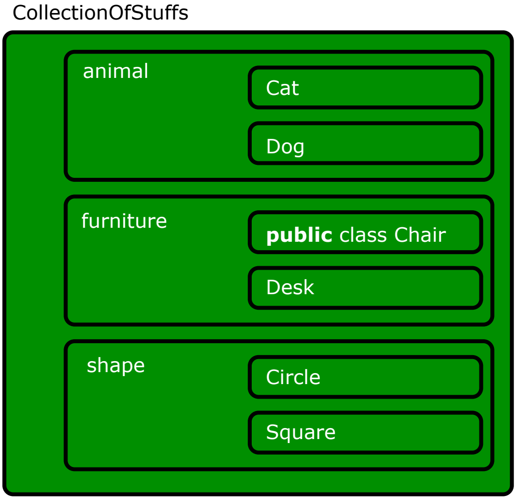
     e.g., Declare $\boxed{\textbf{class}\ \text{Chair}\ \{\ \dots\ \}}$
  2. Across <u>packages</u> [***public***]

     e.g., Declare $\boxed{\textit{public}\ \textbf{class}\ \text{Chair}\ \{\ \dots\ \}}$
- Consider class `Chair` which resides in:
  - package `furniture`
  - project `CollectionOfStuffs`

Visibility of Classes: Across All Classes Within the Resident Package (no modifier)



Visibility of Classes: Across All Classes Within the Resident Package (no modifier)



Visibility of Attributes/Methods: Using Modifiers to Define Scopes

- Two modifiers for declaring visibility of attributes/methods: ***public*** and ***private***
- **Visibility** of an attribute or a method may be declared using a modifier, indicating that it is accessible:
  1. Within its residing class (***most*** restrictive) [***private***]

     e.g., Declare attribute `private int i;`

     e.g., Declare method `private void m(){};`
  2. Across classes within its residing package [no modifier]

     e.g., Declare attribute `int i;`

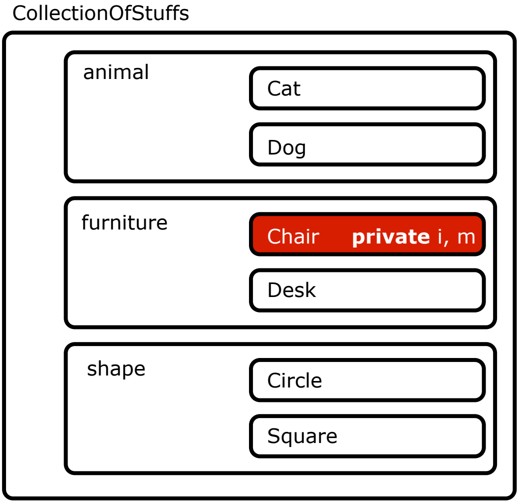
     e.g., Declare method `void m(){};`
  3. Across packages (***least*** restrictive) [***public***]

     e.g., Declare attribute `public int i;`

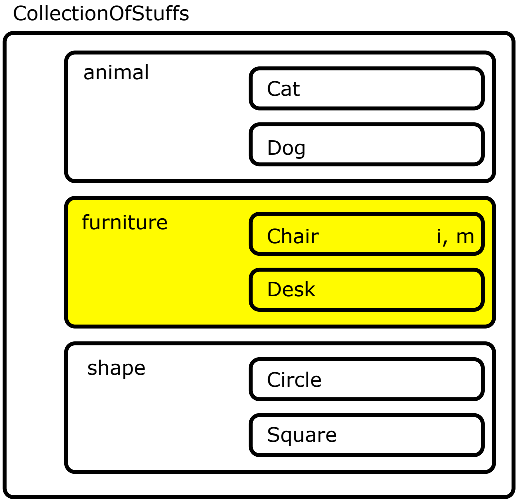
     e.g., Declare method `public void m(){};`
- Consider attributes `i` and `m` residing in:

  Class `Chair`; Package `furniture`; Project `CollectionOfStuffs`.

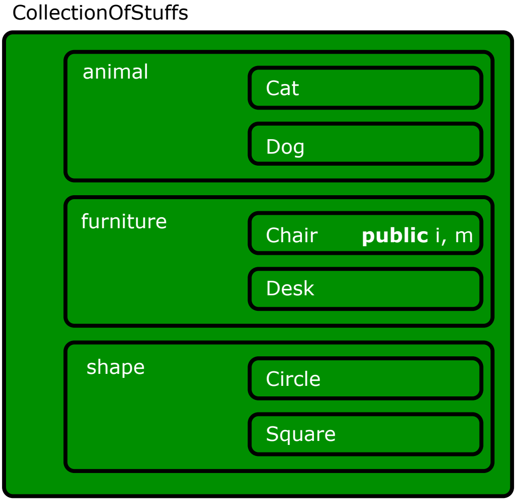
Visibility of Attr./Meth.: Across All Methods Within the Resident Class (`private`)



Visibility of Attr./Meth.: Across All Classes Within the Resident Package (no modifier)



Visibility of Attr./Meth.: Across All Packages Within the Resident Project (`public`)

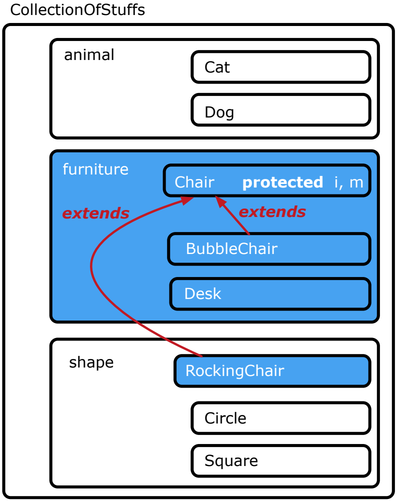


Use of the `protected` Modifier

- ***private*** attributes are not inherited to subclasses.
- package-level attributes (i.e., with **no modifier**) and project-level attributes (i.e., ***public***) are inherited.
- What if we want attributes to be:
  - *visible* to sub-classes outside the current package, but still
  - *invisible* to other non-sub-classes outside the current package?

  Use ***protected***!

Visibility of Attr./Meth.: Across All Methods Same Package and Sub-Classes (`protected`)



Visibility of Attributes/Methods

scope / modifier	CLASS	PACKAGE	SUBCLASS (same pkg)	SUBCLASS (different pkg)	NON-SUBCLASS (across Project)
`public`					
`protected`					
no modifier					
`private`					

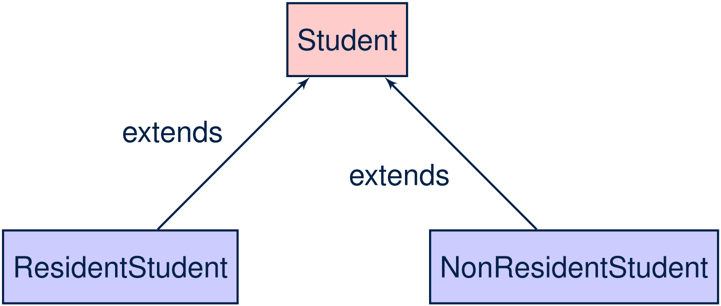
For the rest of this lecture, for simplicity, we assume that:

All relevant parent/child classes are in the same package .

⇒ Attributes with **no modifiers** (***package***-level visibility) suffice.

⇒ Methods with **no modifiers** (***package***-level visibility) suffice.

Inheritance Architecture



Inheritance: The `Student` Parent/Super Class

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

Inheritance: The `ResidentStudent` Child/Sub Class

```
1 class ResidentStudent extends Student {
2   double premiumRate; /* there's a mutator method for this */
3   ResidentStudent (String name) { super(name); }
4   /* register method is inherited */
5   double getTuition() {
6     double base = super.getTuition();
7     return base * premiumRate;
8   }
9 }
```

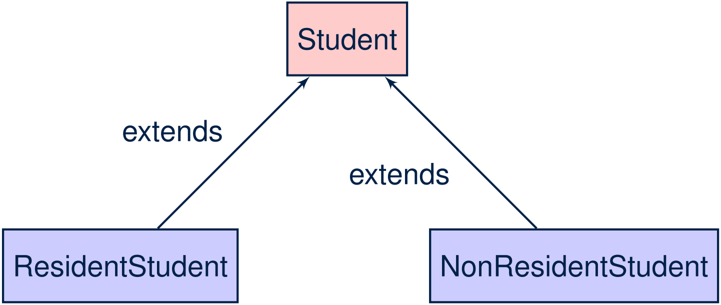
- **L1** declares that `ResidentStudent` inherits all attributes and methods (except constructors) from `Student`.
- There is no need to repeat the `register` method
- Use of *super* in **L3** is as if calling `Student(name)`
- Use of *super* in **L6** returns what `getTuition()` in `Student` returns.
- Use *super* to refer to attributes/methods defined in the super class: `super.name`, `super.register(c)`.

Inheritance: The `NonResidentStudent` Child/Sub Class

```
1 class NonResidentStudent extends Student {
2   double discountRate; /* there's a mutator method for this */
3   NonResidentStudent (String name) { super(name); }
4   /* register method is inherited */
5   double getTuition() {
6     double base = super.getTuition();
7     return base * discountRate;
8   }
9 }
```

- **L1** declares that `NonResidentStudent` inherits all attributes and methods (except constructors) from `Student`.
- There is no need to repeat the `register` method
- Use of *super* in **L3** is as if calling `Student(name)`
- Use of *super* in **L6** returns what `getTuition()` in `Student` returns.
- Use *super* to refer to attributes/methods defined in the super class: `super.name`, `super.register(c)`.

Inheritance Architecture Revisited



- The class that defines the common attributes and methods is called the *parent* or *super* class.
- Each “extended” class is called a *child* or *sub* class.

Using Inheritance for Code Reuse

Inheritance in Java allows you to:

- Define ***common attributes and methods*** in a separate class.
  e.g., the `Student` class
- Define an "extended" version of the class which:
  - ***inherits*** definitions of all attributes and methods
    e.g., `name`, `courses`, `noc`
    e.g., `register`
    e.g., base amount calculation in `getTuition`
    This means code reuse and elimination of code duplicates!
  - ***defines*** **new** attributes and methods if necessary
    e.g., `setPremiumRate` for `ResidentStudent`
    e.g., `setDiscountRate` for `NonResidentStudent`
  - ***redefines***/***overrides*** methods if necessary
    e.g., compounded tuition for `ResidentStudent`
    e.g., discounted tuition for `NonResidentStudent`

Visualizing Parent/Child Objects (1)

- A child class inherits ***all*** non-private attributes from its parent class.

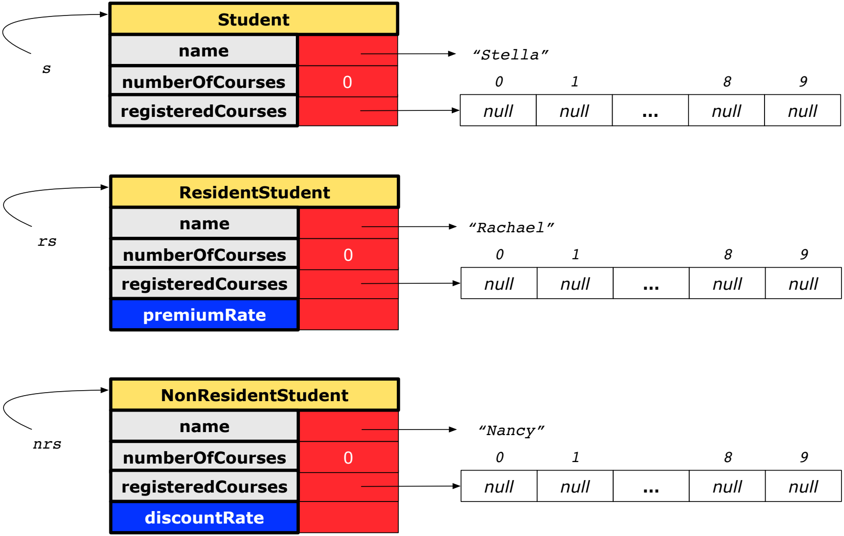
  ⇒ A child instance has ***at least as many*** attributes as an instance of its parent class.

  Consider the following instantiations:

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

- How will these initial objects look like?

Visualizing Parent/Child Objects (2)



Testing the Two Student Sub-Classes

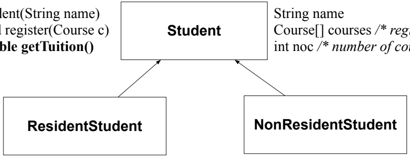
```
public class StudentTester {
  public static void main(String[] args) {
    Course c1 = new Course("EECS2030", 500.00); /* title and fee */
    Course c2 = new Course("EECS3311", 500.00); /* title and fee */
    ResidentStudent jim = new ResidentStudent("J. Davis");
    jim.setPremiumRate(1.25);
    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());
  }
}
```

- The software can be used in the exact same way as before (because we did not modify ***method headers***).
- But now the internal structure of code has been made ***maintainable*** using *inheritance*.

Inheritance Architecture: Static Types & Expectations

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

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()

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

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

<table>
<tr><th></th><th>name</th><th>rcs</th><th>noc</th><th>reg</th><th>getT</th><th>pr</th><th>setPR</th><th>dr</th><th>setDR</th></tr>
<tr><td>s.</td><td colspan="5">✓</td><td colspan="4">×</td></tr>
<tr><td>rs.</td><td colspan="5">✓</td><td colspan="2">✓</td><td colspan="2">×</td></tr>
<tr><td>nrs.</td><td colspan="5">✓</td><td colspan="2">×</td><td colspan="2">✓</td></tr>
</table>

Polymorphism: Intuition (1)

```
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? */
```

- Which one of **L4** and **L5** is *valid*? Which one is *invalid*?
- **Hints**:
  - **L1**: What *kind* of address can `s` store? [`Student`]

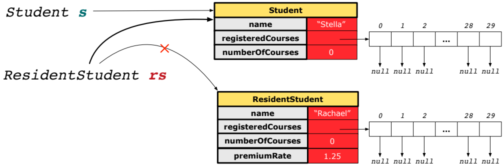
    ∴ The context object `s` is *expected* to be used as:
    - `s.register(eecs2030)` and `s.getTuition()`
  - **L2**: What *kind* of address can `rs` store? [`ResidentStudent`]

    ∴ The context object `rs` is *expected* to be used as:
    - `rs.register(eecs2030)` and `rs.getTuition()`
    - `rs.setPremiumRate(1.50)` [increase premium rate]

Polymorphism: Intuition (2)

```
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? */
```

- *rs* = *s* (**L5**) should be *invalid*:



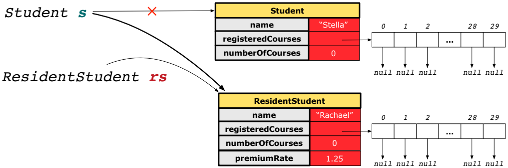
- Since `rs` is declared of type `ResidentStudent`, a subsequent call `rs.setPremiumRate(1.50)` can be expected.
- `rs` is now pointing to a `Student` object.
- Then, what would happen to `rs.setPremiumRate(1.50)`?

CRASH $\because$ `rs.premiumRate` is *undefined*!!

Polymorphism: Intuition (3)

```
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? */
```

- **s** = **rs** (**L4**) should be *valid*:



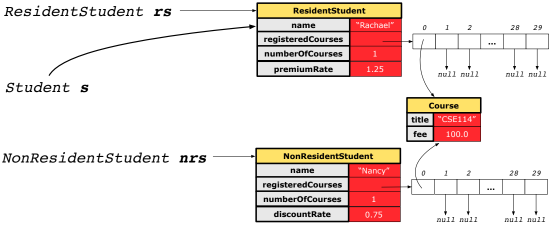
- Since `s` is declared of type `Student`, a subsequent call `s.setPremiumRate(1.50)` is *never* expected.
- `s` is now pointing to a `ResidentStudent` object.
- Then, what would happen to `s.getTuition()`?

OK ∵ `s.premiumRate` is *never directly used*!!

Dynamic Binding: Intuition (1)

```
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()); /* 125.0 */
8  s = nrs; System.out.println( s .getTuition()); /* 75.0 */
```

After `s = rs` (**L7**), `s` points to a `ResidentStudent` object.
⇒ Calling `s .getTuition()` applies the `premiumRate`.

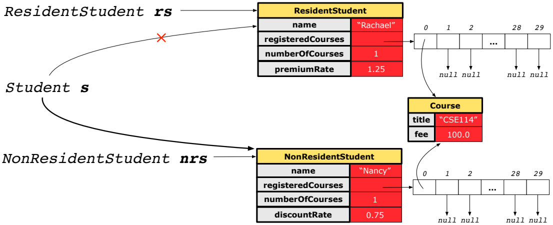


Dynamic Binding: Intuition (2)

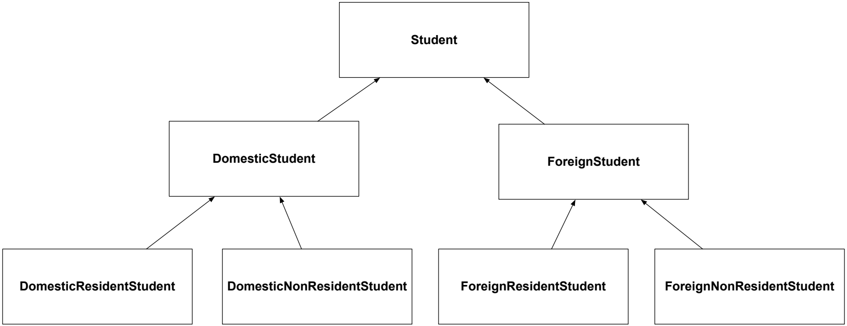
```
1 Course eecs2030 = new Course("EECS2030", 100.0);
2 Student s;
3 ResidentStudent rs = new ResidentStudent("Rachael");
4 NonResidentStudent nrs = new NonResidentStudent("Nancy");
5 rs.setPremiumRate(1.25); rs.register(eecs2030);
6 nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7 s = rs; System.out.println( s .getTuition()); /* 125.0 */
8 s = nrs; System.out.println( s .getTuition()); /* 75.0 */
```

After `s = nrs` (**L8**), `s` points to a `NonResidentStudent` object.

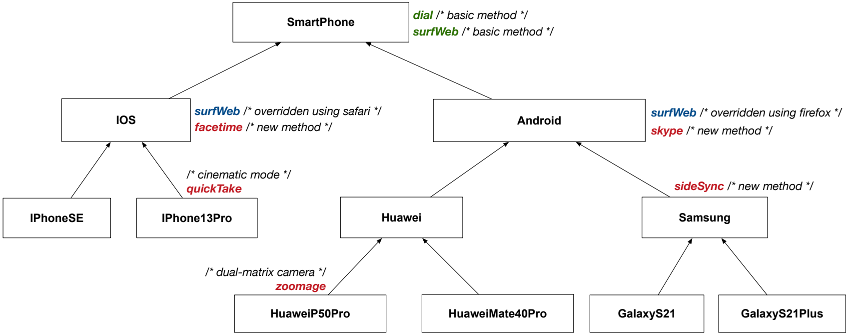
⇒ Calling `s .getTuition()` applies the `discountRate`.



Multi-Level Inheritance Architecture



Multi-Level Inheritance Hierarchy: Smart Phones



Inheritance Forms a Type Hierarchy

- A (data) *type* denotes a set of related ***runtime values***.
  - Every ***class*** can be used as a type: the set of runtime ***objects***.
- Use of ***inheritance*** creates a *hierarchy* of classes:
  - (Implicit) Root of the hierarchy is `Object`.
  - Each `extends` declaration corresponds to an upward arrow.
  - The `extends` relationship is ***transitive***: when `A` extends `B` and `B` extends C, we say `A` *indirectly* extends `C`.
    e.g., Every class implicitly `extends` the `Object` class.
- ***Ancestor*** vs. ***Descendant*** classes:
  - The *ancestor classes* of a class `A` are: `A` itself and all classes that `A` directly, or indirectly, extends.
    - `A` inherits all code (attributes and methods) from its ***ancestor classes***.
      ∴ `A`'s instances have a ***wider range of expected usages*** (i.e., attributes and methods) than instances of its ***ancestor*** classes.
  - The *descendant classes* of a class `A` are: `A` itself and all classes that directly, or indirectly, extends `A`.
    - Code defined in `A` is inherited to all its ***descendant classes***.

Inheritance Accumulates Code for Reuse

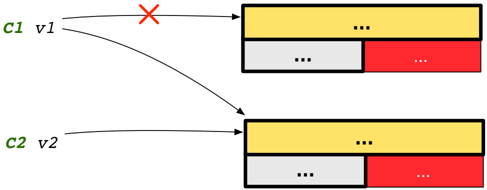
- The ***lower*** a class is in the type hierarchy, the ***more code*** it accumulates from its ***ancestor classes***:
  - A ***descendant class*** inherits all code from its ***ancestor classes***.
  - A ***descendant class*** may also:
    - Declare new attributes
    - Define new methods
    - *Redefine* / *Override* inherited methods
- Consequently:
  - When being used as *context objects*, instances of a class' ***descendant classes*** have a ***wider range of expected usages*** (i.e., attributes and methods).
  - Given a ***reference variable***, expected to store the address of an object of a particular class, we may *substitute* it with (*re-assign* it to) an object of any of its ***descendant classes***.
  - e.g., When expecting a `SmartPhone` object, we may substitute it with either a `IPhone13Pro` or a `Samsung` object.
  - **Justification**: A ***descendant class*** contains *at least as many* methods as defined in its ***ancestor classes*** (but not vice versa!).

Static Types Determine Expectations

- A reference variable's *static type* is what we declare it to be.
  - `Student jim` declares `jim`'s ST as `Student`.
  - `SmartPhone myPhone` declares `myPhone`'s ST as `SmartPhone`.
  - The *static type* of a reference variable *never changes*.
- For a reference variable *v*, its *static type* $\boxed{C}$ defines the *expected usages of v as a context object*.
- A method call `v.m(...)` is ***compilable*** if ***m*** is defined in $\boxed{C}$.
  - e.g., After declaring `Student jim`, we
    - **may** call `register` and `getTuition` on `jim`
    - **may *not*** call `setPremiumRate` (specific to a resident student) or `setDiscountRate` (specific to a non-resident student) on `jim`
  - e.g., After declaring `SmartPhone myPhone`, we
    - **may** call `dial` and `surfWeb` on `myPhone`
    - **may *not*** call `facetime` (specific to an IOS phone) or `skype` (specific to an Android phone) on `myPhone`

Substitutions via Assignments

- By declaring ***C1*** `v1`, ***reference variable*** `v1` will store the ***address*** of an object "of class `C1`" at runtime.
- By declaring ***C2*** `v2`, ***reference variable*** `v2` will store the ***address*** of an object "of class `C2`" at runtime.
- Assignment `v1 = v2` ***copies address*** stored in `v2` into `v1`.
  - `v1` will instead point to wherever `v2` is pointing to. [*object alias*]



- In such assignment `v1 = v2`, we say that we *substitute* an object of (***static***) type `C1` by an object of (***static***) type `C2`.
- *Substitutions* are subject to ***rules***!

Rules of Substitution

When expecting an object of ***static type*** `A`:

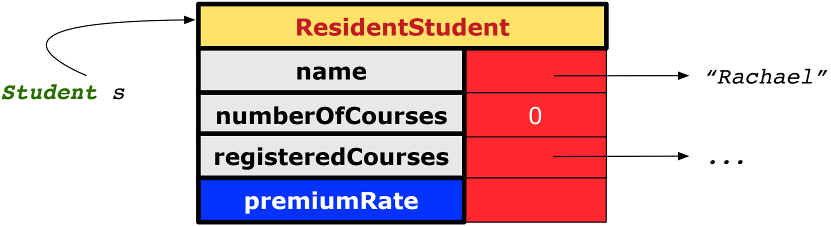
- It is ***safe*** to *substitute* it with an object whose ***static type*** is any of the *descendant class* of `A` (including `A`).
  - ∵ Each *descendant class* of `A`, being the new substitute, is guaranteed to contain all (non-private) attributes/methods defined in `A`.
  - e.g., When expecting an `IOS` phone, you ***can*** substitute it with either an `IPhoneSE` or `IPhone13Pro`.
- It is ***unsafe*** to *substitute* it with an object whose ***static type*** is any of the *ancestor classes of `A`'s parent* (excluding `A`).
  - ∵ Class `A` may have defined new methods that do not exist in any of its *parent's ancestor classes*.
  - e.g., When expecting `IOS` phone, ***unsafe*** to substitute it with a `SmartPhone` ∵ `facetime` not supported in `Android` phone.
- It is also ***unsafe*** to *substitute* it with an object whose ***static type*** is neither an ancestor nor a descendant of `A`.
  - e.g., When expecting `IOS` phone, ***unsafe*** to substitute it with a `HuaweiP50Pro` ∵ `facetime` not supported in `Android` phone.

Reference Variable: Dynamic Type

A ***reference variable***'s *dynamic type* is the type of object that it is currently pointing to at runtime.

- The ***dynamic type*** of a reference variable ***may change*** whenever we *re-assign* that variable to a different object.
- There are two ways to re-assigning a reference variable.

Visualizing Static Type vs. Dynamic Type



- Each segmented box denotes a *runtime* object.
- Arrow denotes a variable (e.g., `s`) storing the object's address.

  Usually, when the context is clear, we leave the variable's ***static type*** implicit (***Student***).
- Title of box indicates type of runtime object, which denotes the ***dynamic type*** of the variable (***ResidentStudent***).

Reference Variable: Changing Dynamic Type (1)

Re-assigning a reference variable to a newly-created object:

- **Substitution Principle**: the new object's class must be a *descendant class* of the reference variable's ***static type***.
- e.g., `Student jim = new ResidentStudent(...)` changes the ***dynamic type*** of `jim` to `ResidentStudent`.
- e.g., `jim = new NonResidentStudent(...)` changes the ***dynamic type*** of `jim` to `NonResidentStudent`.
- e.g., `ResidentStudent jeremy = new Student(...)` is illegal because `Studnet` is **not** a *descendant class* of the ***static type*** of `jeremy` (i.e., `ResidentStudent`).

Reference Variable: Changing Dynamic Type (2)

Re-assigning a reference variable `v` to an existing object that is referenced by another variable `other` (i.e., `v = other`):

- **Substitution Principle**: the static type of `other` must be a *descendant class* of `v`'s ***static type***.
- e.g., Say we declare

```
Student jim = new Student(...);
ResidentStudent rs = new ResidentStudnet(...);
NonResidentStudnet nrs = new NonResidentStudent(...);
```

- `jim = rs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `rs`
- `jim = nrs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `nrs`
- `rs = jim` ×
- `nrs = jim` ×

Polymorphism and Dynamic Binding (1)

- *Polymorphism* : An object variable may have ***"multiple possible shapes"*** (i.e., allowable ***dynamic types***).
  - Consequently, there are ***multiple possible versions*** of each method that may be called.
    - e.g., A ***Student*** variable may have the ***dynamic type*** of **Student**, **ResidentStudent**, or **NonResidentStudent**,
    - This means that there are three possible versions of the `getTuition()` that may be called.
- *Dynamic binding* : When a method `m` is called on an object variable, the version of `m` corresponding to its ***"current shape"*** (i.e., one defined in the ***dynamic type*** of *m*) will be called.

```
Student jim = new ResidentStudent(...);
jim.getTuition(); /* version in ResidentStudent */
jim = new NonResidentStudent(...);
jim.getTuition(); /* version in NonResidentStudent */
```

Polymorphism and Dynamic Binding (2.1)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

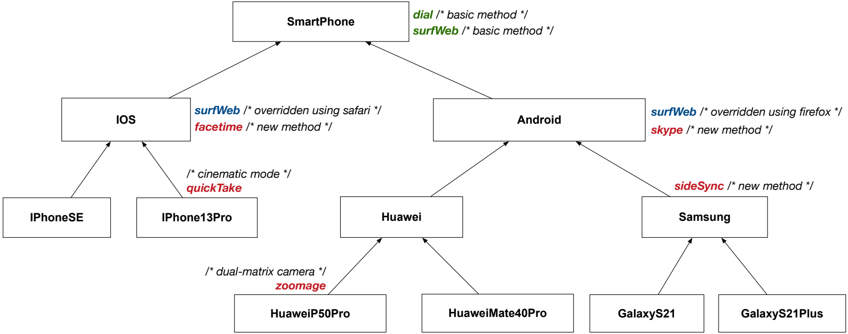
```
class StudentTester1 {
 public static void main(String[] args) {
   Student jim = new Student("J. Davis");
   ResidentStudent rs = new ResidentStudent("J. Davis");
   jim = rs;  /* legal */
   rs = jim;  /* illegal */

   NonResidentStudnet nrs = new NonResidentStudent("J. Davis");
   jim = nrs;  /* legal */
   nrs = jim;  /* illegal */
 }
}
```

Polymorphism and Dynamic Binding (2.2)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

```
class StudentTester2 {
 public static void main(String[] args) {
  Course eecs2030 = new Course("EECS2030", 500.0);
  Student jim = new Student("J. Davis");
  ResidentStudent rs = new ResidentStudent("J. Davis");
  rs.setPremiumRate(1.5);
  jim = rs ;
  System.out.println( jim.getTuition() );  /* 750.0 */
  NonResidentStudnet nrs = new NonResidentStudent("J. Davis");
  nrs.setDiscountRate(0.5);
  jim = nrs ;
  System.out.println( jim.getTuition() );  /* 250.0 */
 }
}
```

Polymorphism and Dynamic Binding (3.1)



Polymorphism and Dynamic Binding (3.2)

```
class SmartPhoneTest1 {
  public static void main(String[] args) {
    SmartPhone myPhone;
    IOS ip = new IPhoneSE();
    Samsung ss = new GalaxyS21Plus();
    myPhone = ip;  /* legal */
    myPhone = ss;  /* legal */

    IOS presentForHeeyeon;
    presentForHeeyeon = ip;  /* legal */
    presentForHeeyeon = ss;  /* illegal */
  }
}
```

Polymorphism and Dynamic Binding (3.3)

```
class SmartPhoneTest2 {
  public static void main(String[] args) {
    SmartPhone myPhone;
    IOS ip = new IPhone13Pro();
    myPhone = ip;
    myPhone. surfWeb (); /* version of surfWeb in IPhone13Pro */

    Samsung ss = new GalaxyS21();
    myPhone = ss;
    myPhone. surfWeb (); /* version of surfWeb in GalaxyS21 */
  }
}
```

Reference Type Casting: Motivation (1.1)

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

- **L1** is *legal*: `ResidentStudent` is a **descendant class** of the ***static type*** of `jim` (i.e., `Student`).
- **L2** is *illegal*: `jim`'s ***ST*** (i.e., `Student`) is ***not*** a **descendant class** of `rs`'s ***ST*** (i.e., `ResidentStudent`).

  Java compiler is *unable to infer* that `jim`'s ***dynamic type*** in **L2** is `ResidentStudent`!
- Force the Java compiler to believe so via a cast in **L2**:

```
ResidentStudent rs = (ResidentStudent) jim;
```

  - The cast (***ResidentStudent***) jim creates for `jim` ***a temporary alias*** whose ***ST*** corresponds to the ***cast type*** (***ResidentStudent***).
  - Alias `rs` of ***ST*** **`ResidentStudent`** is then created via an assignment. **Note**. `jim`'s ***ST*** always remains **`Student`**.
- *dynamic binding*: After the *cast*, **L3** will execute the correct version of `setPremiumRate` (∵ ***DT*** of `rs` is ***`ResidentStudent`***).

Reference Type Casting: Motivation (1.2)

$$\overbrace{\textit{ResidentStudent}\ \ rs}^{\texttt{ST: ResidentStudent}} \overbrace{=}^{\texttt{valid substitution}} \underbrace{(\texttt{ResidentStudent})\ \overbrace{jim}^{\texttt{ST: Student}}}_{\texttt{an alias whose ST is ResidentStudent}} ;$$

- Variable `rs` is declared of ***static type*** (***ST***) `ResidentStudent`.
- Variable `jim` is declared of ***ST*** `Student`.
- The cast $\boxed{(\textbf{\textit{ResidentStudent}})\ \text{jim}}$ creates for `jim` a **temporary alias**, whose ***ST*** corresponds to the ***cast type*** (`ResidentStudent`).

  ⇒ Such a cast makes the assignment valid.

  ∵ RHS's ***ST*** (`ResidentStudent`) is a descendant of LHS's ***ST*** (`ResidentStudent`).

  ⇒ The assignment creates an alias `rs` with ***ST*** `ResidentStudent`.
- **No** new object is created.

  Only an ***alias*** `rs` with a different ***ST*** (`ResidentStudent`) is created.
- After the assignment, `jim`'s ***ST*** **remains** `Student`.

Reference Type Casting: Motivation (2.1)

```
1 SmartPhone aPhone = new IPhone13Pro();
2 IPhone13Pro forHeeyeon = aPhone;
3 forHeeyeon.facetime(1.5);
```

- **L1** is *legal*: `IPhone13Pro` is a **descendant class** of the ***static type*** of `aPhone` (i.e., `SmartPhone`).
- **L2** is *illegal*: `aPhone`'s ***ST*** (i.e., `SmartPhone`) is ***not*** a **descendant class** of `forHeeyeon`'s ***ST*** (i.e., `IPhone13Pro`).

  Java compiler is *unable to infer* that `aPhone`'s ***dynamic type*** in **L2** is `IPhone13Pro`!
- Force the Java compiler to believe so via a cast in **L2**:

  ```
  IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
  ```

  - The cast (***IPhone13Pro***) aPhone creates for `aPhone` ***a temporary alias*** whose ***ST*** corresponds to the ***cast type*** (***IPhone13Pro***).
  - Alias `forHeeyeon` of ***ST*** **`IPhone13Pro`** is then created via an assignment.

    Note. `aPhone`'s ***ST*** always remains **`SmartPhone`**.
- *dynamic binding* : After the *cast* , **L3** will execute the correct version of `facetime` (∵ ***DT*** of `forHeeyeon` is ***`IPhone13Pro`***).

Reference Type Casting: Motivation (2.2)

$$\overbrace{\text{IPhone13Pro forHeeyeon}}^{\texttt{ST: IPhone13Pro}} \overbrace{=}^{\texttt{valid substitution}} \underbrace{(\texttt{IPhone13Pro})\ \overbrace{\text{aPhone}}^{\texttt{ST: SmartPhone}}}_{\texttt{an alias whose ST is IPhone13Pro}} ;$$

- Variable `forHeeyeon` is declared of ***static type*** (***ST***) `IPhone13Pro`.
- Variable `aPhone` is declared of ***ST*** `SmartPhone`.
- The cast $\boxed{\text{(\textit{IPhone13Pro}) aPhone}}$ creates for `aPhone` a **temporary alias**, whose ***ST*** corresponds to the ***cast type*** (`IPhone13Pro`).

  ⇒ Such a cast makes the assignment valid.

  ∵ RHS's ***ST*** (`IPhone13Pro`) is a descendant of LHS's ***ST*** (`IPhone13Pro`).

  ⇒ The assignment creates an alias `forHeeyeon` with ***ST*** `IPhone13Pro`.
- **No** new object is created.

  Only an ***alias*** `forHeeyeon` with a different ***ST*** (`IPhone13Pro`) is created.
- After the assignment, `aPhone`'s ***ST*** **remains** `SmartPhone`.

Type Cast: Named or Anonymous

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();
```

Common Mistake:

```
1 SmartPhone aPhone = new IPhone13Pro();
2 (IPhone13Pro) aPhone.facetime();
```

L2 ≡ `(IPhone13Pro) (aPhone.facetime())`: Call, then cast.

⇒ This does **not** compile ∵ `facetime()` is **not** declared in the *static type* of `aPhone` (`SmartPhone`).

Notes on Type Cast (1)

- Given variable **v** of ***static type*** ST_v, it is *compilable* to cast **v** to C, as long as C is an **ancestor** or **descendant** of ST_v.
- Without cast, we can **only** call methods defined in ST_v on v.
- Casting v to C creates for v an alias with ***ST*** C.
  $\Rightarrow$ All methods that are defined in C can be called.

```
Android myPhone = new GalaxyS21Plus();
/* can call methods declared in Android on myPhone
 * dial, surfweb, skype  ✓  sideSync  × */
SmartPhone sp = (SmartPhone) myPhone;
/* Compiles OK ∵ SmartPhone is an ancestor class of Android
 * expectations on sp narrowed to methods in SmartPhone
 * sp.dial, sp.surfweb  ✓  sp.skype, sp.sideSync  × */
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
/* Compiles OK ∵ GalaxyS21Plus is a descendant class of Android
 * expectations on ga widened to methods in GalaxyS21Plus
 * ga.dial, ga.surfweb, ga.skype, ga.sideSync  ✓ */
```

Reference Type Casting: Danger (1)

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

- **L1** is *legal*: `NonResidentStudent` is a **descendant** of the static type of `jim` (`Student`).
- **L2** is *legal* (where the cast type is `ResidentStudent`):
  - cast type is **descendant** of `jim`'s ST (`Student`).
  - cast type is **descendant** of `rs`'s ST (`ResidentStudent`).
- **L3** is *legal* ∵ `setPremiumRate` is in `rs`' ***ST*** `ResidentStudent`.
- Java compiler is *unable to infer* that `jim`'s *dynamic type* in **L2** is actually `NonResidentStudent`.
- Executing **L2** will result in a *`ClassCastException`*.
  ∵ Attribute `premiumRate` (expected from a ***ResidentStudent***) is *undefined* on the ***NonResidentStudent*** object being cast.

Reference Type Casting: Danger (2)

```
1 SmartPhone aPhone = new GalaxyS21Plus();
2 IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
3 forHeeyeon.quickTake();
```

- **L1** is *legal*: `GalaxyS21Plus` is a **descendant** of the static type of `aPhone` (`SmartPhone`).
- **L2** is *legal* (where the cast type is `Iphone6sPlus`):
  - cast type is **descendant** of `aPhone`'s ST (`SmartPhone`).
  - cast type is **descendant** of `forHeeyeon`'s ST (`IPhone13Pro`).
- **L3** is *legal* ∵ `quickTake` is in `forHeeyeon`' ***ST*** `IPhone13Pro`.
- Java compiler is *unable to infer* that `aPhone`'s *dynamic type* in **L2** is actually `GalaxyS21Plus`.
- Executing **L2** will result in a *`ClassCastException`*.
  ∵ Methods `facetime`, `quickTake` (expected from an ***IPhone13Pro***) is *undefined* on the ***GalaxyS21Plus*** object being cast.

Notes on Type Cast (2.1)

Given a variable *v* of static type ST_v and dynamic type DT_v:

- `(C) v` is ***compilable*** if `C` is ST_v's **ancestor** or **descendant**.
- Casting `v` to `C`'s ***ancestor***/***descendant*** ***narrows***/***widens*** expectations.
- However, being ***compilable*** does not guarantee ***runtime-error-free***!

```
1 SmartPhone myPhone = new Samsung();
2 /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3 GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
4 /* Compiles OK ∵ GalaxyS21Plus is a descendant class of SmartPhone
5  * can now call methods declared in GalaxyS21Plus on ga
6  * ga.dial, ga.surfweb, ga.skype, ga.sideSync  ✓ */
```

- Type cast in **L3** is ***compilable***.
- Executing **L3** will cause `ClassCastException`.

  L3: `myPhone`'s *DT* `Samsung` cannot meet expectations of the temporary *ST* `GalaxyS21Plus` (e.g., `sideSync`).

Notes on Type Cast (2.2)

Given a variable *v* of static type ST_v and dynamic type DT_v:

- `(C) v` is *compilable* if `C` is ST_v's **ancestor** or **descendant**.
- Casting `v` to `C`'s ***ancestor***/***descendant*** ***narrows***/***widens*** expectations.
- However, being *compilable* does not guarantee ***runtime-error-free***!

```
1 SmartPhone myPhone = new Samsung();
2 /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3 IPhone13Pro ip = (IPhone13Pro) myPhone;
4 /* Compiles OK ∵ IPhone13Pro is a descendant class of SmartPhone
5  * can now call methods declared in IPhone13Pro on ip
6  * ip.dial, ip.surfweb, ip.facetime, ip.quickTake ✓ */
```

- Type cast in **L3** is *compilable*.
- Executing **L3** will cause `ClassCastException`.

  L3: `myPhone`'s *DT* `Samsung` cannot meet expectations of the temporary *ST* `IPhone13Pro` (e.g., `quickTake`).

Notes on Type Cast (2.3)

A cast `(C) v` is ***compilable*** and ***runtime-error-free*** if *C* is located along the **ancestor path** of DT_v.

e.g., Given **_Android_** myPhone = new **_Samsung_**();

- Cast `myPhone` to a class along the **ancestor path** of its ***DT Samsung***.
- Casting `myPhone` to a class with more expectations than its ***DT Samsung*** (e.g., `GalaxyS21Plus`) will cause `ClassCastException`.
- Casting `myPhone` to a class irrelevant to its ***DT Samsung*** (e.g., `HuaweiMate40Pro`) will cause `ClassCastException`.

Required Reading: Static Types, Dynamic Types, Casts

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2024/F/EECS2030/notes/EECS2030_F24_Notes_Static_Types_Cast.pdf

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;
```

- After **L1**:
  - ***ST*** of `b` is `B`
  - ***DT*** of `b` is `C`
- Does **L2** compile? [NO]

  ∵ cast type `D` is neither an ancestor nor a descendant of `b`'s ***ST*** `B`
- Would `D d = (D) ((A) b)` fix **L2**? [YES]

  ∵ cast type `D` is an ancestor of `b`'s cast, temporary ***ST*** `A`
- `ClassCastException` when executing this fixed **L2**? [YES]

  ∵ cast type `D` is not an ancestor of `b`'s ***DT*** `C`

Reference Type Casting: Runtime Check (1)

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

- **L1** is *legal*: `NonResidentStudent` is a **descendant class** of the ***static type*** of `jim` (i.e., `Student`).
- **L2** checks if `jim`'s ***DT*** is a descendant of `ResidentStudent`.

  FALSE ∵ `jim`'s *dynamic type* is `NonResidentStudent`!
- **L3** is *legal*: `jim`'s cast type (i.e., `ResidentStudent`) is a **descendant class** of `rs`'s ***ST*** (i.e., `ResidentStudent`).
- **L3** will not be executed at runtime, hence no `ClassCastException`, thanks to the check in **L2**!

Reference Type Casting: Runtime Check (2)


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

- **L1** is *legal*: `GalaxyS21Plus` is a **descendant class** of the static type of `aPhone` (i.e., `SmartPhone`).
- **L2** checks if `aPhone`'s ***DT*** is a <u>descendant</u> of `IPhone13Pro`.

  FALSE $\because$ `aPhone`'s *dynamic type* is `GalaxyS21Plus`!
- **L3** is *legal*: `aPhone`'s cast type (i.e., `IPhone13Pro`) is a **descendant class** of `forHeeyeon`'s ***static type*** (i.e., `IOS`).
- **L3** will not be executed at runtime, hence no `ClassCastException`, thanks to the check in **L2**!

Notes on the `instanceof` Operator (1)

Given a reference variable `v` and a class `C`, you write

`v instanceof C`

to check if the ***dynamic type*** of `v`, at the moment of being checked, is a **descendant class** of `C` (so that `(C) v` is safe).

```
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 */
```

⇒ ***Samsung*** is the most specific type which `myPhone` can be **safely** cast to.

Notes on the `instanceof` Operator (2)

Given a reference variable `v` and a class `C`,

`v instanceof C` checks if the ***dynamic type*** of `v`, at the moment of being checked, is a `descendant class` of `C`.

```
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 }
```

- **L3** evaluates to *true*. [*safe* to cast]
- **L6** and **L9** evaluate to *false*. [*unsafe* to cast]

  This prevents **L7** and **L10**, causing `ClassCastException` if executed, from being executed.

Static Types, Casts, Polymorphism (1.1)


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

```
SmartPhone sp = new IPhone13Pro();   ✓
sp.dial();   ✓
sp.facetime();   ×
sp.quickTake();   ×
```

Static type of *sp* is `SmartPhone`

⇒ can only call methods defined in `SmartPhone` on *sp*

Static Types, Casts, Polymorphism (1.2)

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

```
IOS ip = new IPhone13Pro();   ✓
ip.dial();   ✓
ip.facetime();   ✓
ip.quickTake();   ×
```

Static type of *ip* is `IOS`

⇒ can only call methods defined in `IOS` on *ip*

Static Types, Casts, Polymorphism (1.3)

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

```
IPhone13Pro ip6sp = new IPhone13Pro();   ✓
ip6sp.dial();   ✓
ip6sp.facetime();   ✓
ip6sp.quickTake();   ✓
```

Static type of *ip6sp* is `IPhone13Pro`

⇒ can call all methods defined in `IPhone13Pro` on *ip6sp*

Static Types, Casts, Polymorphism (1.4)

```
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();     ✓
```

L4 is equivalent to the following two lines:

```
IPhone13Pro ip6sp = (IPhone13Pro) sp;
ip6sp.quickTake();
```

Static Types, Casts, Polymorphism (2)

Given a reference variable declaration

```
C v;
```

- *Static type* of reference variable *v* is class *C*
- A method call `v.m` is valid if *m* is a method **defined** in class ***C***.
- Despite the *dynamic type* of *v*, you are only allowed to call methods that are defined in the *static type* `C` on *v*.
- If you are certain that *v*'s *dynamic type* can be expected **more** than its *static type*, then you may use an `insanceof` check and a cast.

```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if(s instanceof ResidentStudent) {
  ((ResidentStudent) s).setPremiumRate(1.75);
  System.out.println(((ResidentStudent) s).getTuition());
}
```

Polymorphism: Method Parameters (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++; } }
```

- **L3**: `ss[c] = rs` is valid. ∵ RHS's ST `ResidentStudent` is a *descendant class* of LHS's ST `Student`.
- Say we have a `StudentManagementSystem` object `sms`:
  - `sms.addRS(o)` attempts the following assignment (recall call by value), which replaces parameter `rs` by a copy of argument `o`:

    ```
    rs = o;
    ```

  - Whether this argument passing is valid depends on `o`'s *static type*.
- In the signature of a method `m`, if the type of a parameter is class `C`, then we may call method `m` by passing objects whose *static types* are `C`'s *descendants*.

Polymorphism: Method Parameters (2.1)

In the `StudentManagementSystemTester`:

```
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);  ✓
```

Polymorphism: Method Parameters (2.2)

In the `StudentManagementSystemTester`:

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

- L4 ***compiles*** with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ `(ResidentStudent)` is a descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- But, there will be a *`ClassCastException`* at runtime!
  ∵ `s`' ***DT*** (`Student`) is **not** a descendant of `ResidentStudent`.
- We should have written:

```
if(s instanceof ResidentStudent) {
  sms.addRS((ResidentStudent) s);
}
```

The **`instanceof`** expression will evaluate to *false*, meaning it is *unsafe* to cast, thus preventing `ClassCastException`.

Polymorphism: Method Parameters (2.3)

In the `StudentManagementSystemTester`:

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

- **L4** ***compiles*** with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ `(ResidentStudent)` is a descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- But, there will be a *`ClassCastException`* at runtime!
  ∵ `s`' ***DT*** (`NonResidentStudent`) **not** descendant of `ResidentStudent`.
- We should have written:

```
if(s instanceof ResidentStudent) {
  sms.addRS((ResidentStudent) s);
}
```

The **`instanceof`** expression will evaluate to *false*, meaning it is *unsafe* to cast, thus preventing `ClassCastException`.

Polymorphism: Method Parameters (2.4)

In the `StudentManagementSystemTester`:

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

- **L4** ***compiles*** with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ `(ResidentStudent)` is a descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- And, there will be **no** *`ClassCastException`* at runtime! ∵ `s`' ***DT*** (`ResidentStudent`) is descendant of `ResidentStudent`.
- We should have written:

```
if(s instanceof ResidentStudent) {
  sms.addRS((ResidentStudent) s);
}
```

The **`instanceof`** expression will evaluate to *true*, meaning it is *safe* to cast.

Polymorphism: Method Parameters (2.5)

In the `StudentManagementSystemTester`:

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

Will **L4** with a cast compile?

```
sms.addRS( (ResidentStudent) nrs)
```

NO ∵ `(ResidentStudent)` is ***not*** a descendant of `nrs`'s ***ST*** (`NonResidentStudent`).

Why Inheritance: A Polymorphic Collection of Students

How do you define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
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)
    }
  }
}
```

a collection of students without inheritance

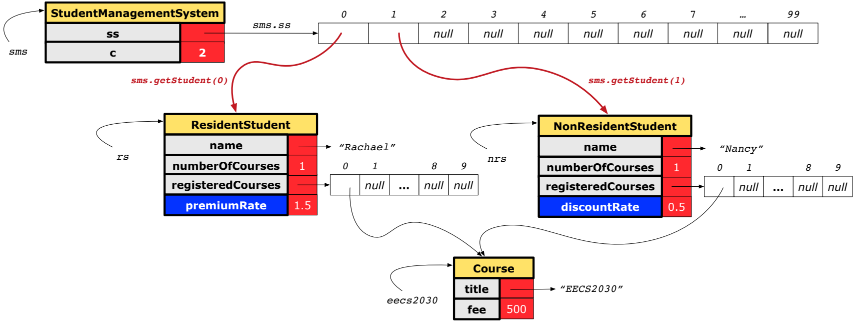
Polymorphism and Dynamic Binding: A Polymorphic Collection of Students (1)

```
ResidentStudent rs = new ResidentStudent("Rachael");
rs.setPremiumRate(1.5);
NonResidentStudent nrs = new NonResidentStudent("Nancy");
nrs.setDiscountRate(0.5);
StudentManagementSystem sms = new StudentManagementSystem();
sms.addStudent( rs ); /* polymorphism */
sms.addStudent( nrs ); /* polymorphism */
Course eecs2030 = new Course("EECS2030", 500.0);
sms.registerAll(eecs2030);
for(int i = 0; i < sms.numberOfStudents; i ++) {
  /* Dynamic Binding:
   * Right version of getTuition will be called */
  System.out.println(sms.students[i]. getTuition() );
}
```

Polymorphism and Dynamic Binding: A Polymorphic Collection of Students (2)

At runtime, attribute `sms.ss` is a *polymorphic* array:

- ***Static type*** of each item is as declared: ***Student***
- ***Dynamic type*** of each item is a **descendant** of ***Student***: ***ResidentStudent***, ***NonResidentStudent***



Polymorphism: Return Types (1)

```
1  class StudentManagementSystem {
2    Student[] ss; int c;
3    void addStudent(Student s) { ss[c] = s; c++; }
4    Student getStudent(int i) {
5      Student s = null;
6      if(i < 0 || i >= c) {
7        throw new InvalidStudentIndexException("Invalid index.");
8      }
9      else {
10       s = ss[i];
11     }
12     return s;
13   } }
```

L4: `Student` is ***static type*** of `getStudent`'s return value.

L10: `ss[i]`'s ST (`Student`) is **descendant** of `s`' ST (`Student`).

Question: What can be the ***dynamic type*** of `s` after **L10**?

Answer: All descendant classes of `Student`.

Polymorphism: Return Types (2)

```java
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

```java
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? */
```

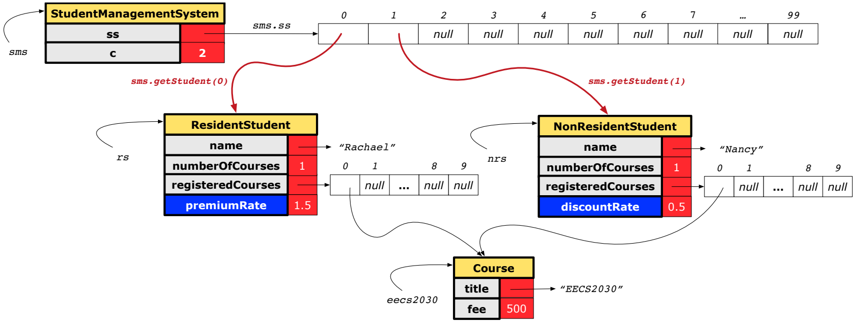
static return type: Student

```java
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); ×
```

Polymorphism: Return Types (3)

At runtime, attribute `sms.ss` is a *polymorphic* array:

- ***Static type*** of each item is as declared: ***Student***
- ***Dynamic type*** of each item is a **descendant** of ***Student***: ***ResidentStudent***, ***NonResidentStudent***



Static Type vs. Dynamic Type: When to consider which?

- *Whether or not Java code compiles* depends only on the ***static types*** of relevant variables.

  $\because$ Inferring the ***dynamic type*** statically is an *undecidable* problem that is inherently impossible to solve.

- *The behaviour of Java code being executed at runtime* (e.g., which version of method is called due to dynamic binding, whether or not a `ClassCastException` will occur, *etc.*) depends on the ***dynamic types*** of relevant variables.

  $\Rightarrow$ Best practice is to visualize how objects are created (by drawing boxes) and variables are re-assigned (by drawing arrows).

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***?

Even if $\boxed{\texttt{(C) y}}$ compiles OK, there will be a runtime `ClassCastException` if `C` is not an **ancestor** of `y`'s ***DT***!

Root of the Java Class Hierarchy

- Implicitly:
  - Every class is a *child/sub* class of the `Object` class.
  - The `Object` class is the *parent/super* class of every class.
- There are two useful *accessor methods* that every class *inherits* from the `Object` class:
  - `boolean equals(Object other)`
    Indicates whether some other object is “equal to” this one.
    - The default definition inherited from `Object`:

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

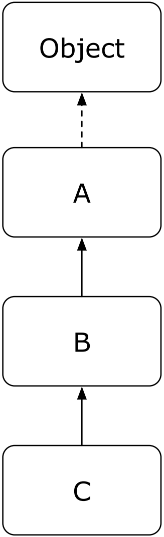
  - `String toString()`
    Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

Overriding and Dynamic Binding (1)

`Object` is the common parent/super class of every class.

- Every class inherits the *default version* of `equals`
- Say a reference variable *v* has ***dynamic type*** *D*:
  - **Case 1** *D* *overrides* `equals`
    ⇒ *v.equals(...)* invokes the *overridden version* in *D*
  - **Case 2** *D* does *not override* `equals`
    Case 2.1 At least one ancestor classes of *D* *override* `equals`
    ⇒ *v.equals(...)* invokes the *overridden version* in the ***closest ancestor class***
    Case 2.2 No ancestor classes of *D* *override* `equals`
    ⇒ *v.equals(...)* invokes *default version* inherited from `Object`.
- Same principle applies to the `toString` method, and all overridden methods in general.

Overriding and Dynamic Binding (2.1)



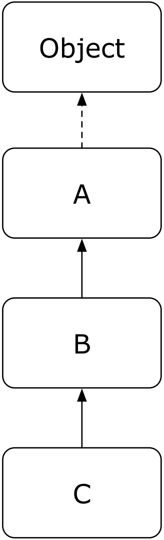
boolean equals (Object obj) {
  return this == obj;
}

```
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`]

Overriding and Dynamic Binding (2.2)




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

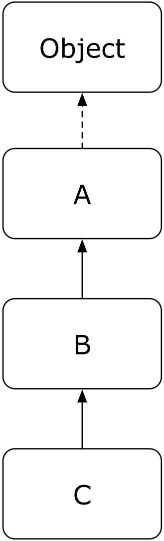
```
boolean equals (Object obj) {
  /* overridden version */
}
```

```
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`? [C]

Overriding and Dynamic Binding (2.3)




```
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]

Behaviour of Inherited `toString` Method (1)

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
Point@677327b6
```

- Implicitly, the `toString` method is called inside the `println` method.
- By default, the address stored in `p1` gets printed.
- We need to *redefine* / *override* the `toString` method, inherited from the `Object` class, in the `Point` class.

Behaviour of Inherited `toString` Method (2)

```
class Point {
  double x;
  double y;
  public String toString() {
    return "(" + this.x + ", " + this.y + ")";
  }
}
```

After redefining/overriding the `toString` method:

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
(2, 4)
```

Behaviour of Inherited `toString` Method (3)

Exercise: Override the `equals` and `toString` methods for the `ResidentStudent` and `NonResidentStudent` classes.

Beyond this lecture. . .

- Implement the ***inheritance hierarchy*** of **Students** and reproduce all lecture examples.
- Implement the ***inheritance hierarchy*** of **Smart Phones** and reproduce all lecture examples.

  Hints. Pay attention to:
  - ***Valid***? ***Compiles***?
  - ***ClassCastException***?
- Study the `ExampleTypeCasts` example: draw the ***inheritance hierarchy*** and experiment with the various substitutions and casts.

Index (1)

Learning Outcomes

Why Inheritance: A Motivating Example

Why Inheritance: A Motivating Example

No Inheritance: `ResidentStudent` Class

No Inheritance: `NonResidentClass`

No Inheritance: Testing Student Classes

No Inheritance:
Issues with the Student Classes

No Inheritance: Maintainability of Code (1)

No Inheritance: Maintainability of Code (2)

Index (2)

No Inheritance:
A Collection of Various Kinds of Students

Visibility: Project, Packages, Classes

Visibility of Classes

Visibility of Classes: Across All Classes
Within the Resident Package (no modifier)

Visibility of Classes: Across All Classes
Within the Resident Package (no modifier)

Visibility of Attributes/Methods:
Using Modifiers to Define Scopes

Visibility of Attr./Meth.: Across All Methods
Within the Resident Class (`private`)

Index (3)

Visibility of Attr./Meth.: Across All Classes
Within the Resident Package (no modifier)

Visibility of Attr./Meth.: Across All Packages
Within the Resident Project (`public`)

Use of the `protected` Modifier

Visibility of Attr./Meth.: Across All Methods
Within the Resident Package and Sub-Classes (`protected`)

Visibility of Attr./Meth.

Inheritance Architecture

Inheritance: The `Student` Parent/Super Class

Inheritance:
The `ResidentStudent` Child/Sub Class

Index (4)

Inheritance:
The `NonResidentStudent` Child/Sub Class

Inheritance Architecture Revisited

Using Inheritance for Code Reuse

Visualizing Parent/Child Objects (1)

Visualizing Parent/Child Objects (2)

Testing the Two Student Sub-Classes

Inheritance Architecture:
Static Types & Expectations

Polymorphism: Intuition (1)

Polymorphism: Intuition (2)

Index (5)

Polymorphism: Intuition (3)

Dynamic Binding: Intuition (1)

Dynamic Binding: Intuition (2)

Multi-Level Inheritance Architecture

Multi-Level Inheritance Hierarchy:
Smart Phones

Inheritance Forms a Type Hierarchy

Inheritance Accumulates Code for Reuse

Static Types Determine Expectations

Substitutions via Assignments

Rules of Substitution

Index (6)

Reference Variable: Dynamic Type

Visualizing Static Type vs. Dynamic Type

Reference Variable:
Changing Dynamic Type (1)

Reference Variable:
Changing Dynamic Type (2)

Polymorphism and Dynamic Binding (1)

Polymorphism and Dynamic Binding (2.1)

Polymorphism and Dynamic Binding (2.2)

Polymorphism and Dynamic Binding (3.1)

Polymorphism and Dynamic Binding (3.2)

Index (7)

Polymorphism and Dynamic Binding (3.3)

Reference Type Casting: Motivation (1.1)

Reference Type Casting: Motivation (1.2)

Reference Type Casting: Motivation (2.1)

Reference Type Casting: Motivation (2.2)

Type Cast: Named or Anonymous

Notes on Type Cast (1)

Reference Type Casting: Danger (1)

Reference Type Casting: Danger (2)

Notes on Type Cast (2.1)

Notes on Type Cast (2.2)

Index (8)

Notes on Type Cast (2.3)

Required Reading:
Static Types, Dynamic Types, Casts

Compilable Cast vs. Exception-Free Cast

Reference Type Casting: Runtime Check (1)

Reference Type Casting: Runtime Check (2)

Notes on the `instanceof` Operator (1)

Notes on the `instanceof` Operator (2)

Static Types, Casts, Polymorphism (1.1)

Static Types, Casts, Polymorphism (1.2)

Static Types, Casts, Polymorphism (1.3)

Index (9)

Static Types, Casts, Polymorphism (1.4)

Static Types, Casts, Polymorphism (2)

Polymorphism: Method Parameters (1)

Polymorphism: Method Parameters (2.1)

Polymorphism: Method Parameters (2.2)

Polymorphism: Method Parameters (2.3)

Polymorphism: Method Parameters (2.4)

Polymorphism: Method Parameters (2.5)

Why Inheritance:
A Polymorphic Collection of Students

Polymorphism and Dynamic Binding:
A Polymorphic Collection of Students (1)

Index (10)

Polymorphism and Dynamic Binding:
A Polymorphic Collection of Students (2)

Polymorphism: Return Types (1)

Polymorphism: Return Types (2)

Polymorphism: Return Types (3)

Static Type vs. Dynamic Type:
When to consider which?

Summary: Type Checking Rules

Root of the Java Class Hierarchy

Overriding and Dynamic Binding (1)

Overriding and Dynamic Binding (2.1)

Index (11)

Overriding and Dynamic Binding (2.2)

Overriding and Dynamic Binding (2.3)

Behaviour of Inherited `toString` Method (1)

Behaviour of Inherited `toString` Method (2)

Behaviour of Inherited `toString` Method (3)

Beyond this lecture...