# Some Final Object Concepts

Object Oriented Programming
2016375 - 5

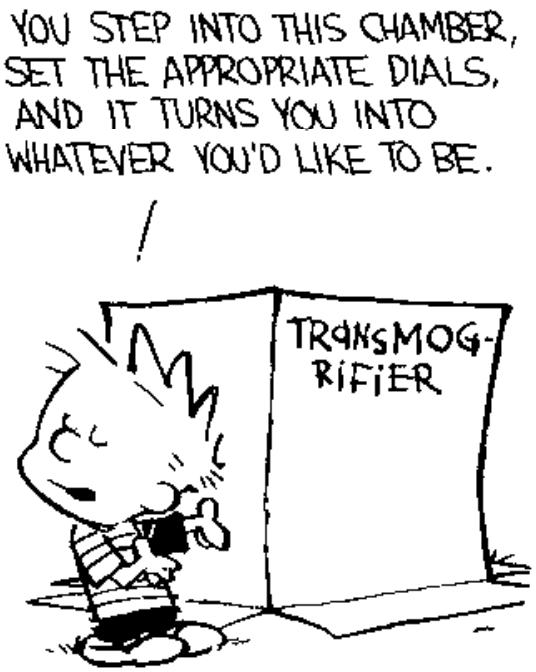
Camilo López

# Outline

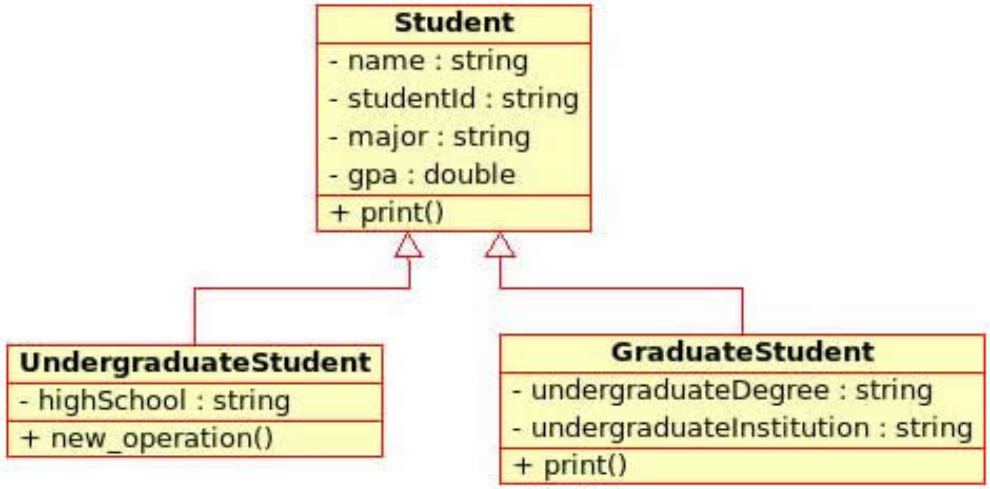
- Polymorphism
- Three Distinguishing Features of an OOPL
- Abstract Classes
- Interfaces
- Static Features

# Polymorphism

- The term ***polymorphism*** is defined in Merriam-Webster's dictionary as "the quality or state of being able to assume different forms."



# Polymorphism




```
ArrayList<Student> students = new ArrayList<Student>();
for (Student s : students) {
    s.print());
}
```

**This won't work unless all objects in the collection understand the message being sent**

**We must guarantee that every object in the collection at run time will have such a method**

# Polymorphism

*Polymorphism Simplifies Code Maintenance*

```
    for (Student s : studentBody) {
        // Process the next student.
        // Pseudocode.
        if (s is an undergraduate student)
            s.printAsUndergraduateStudent();
        else if (s is a graduate student)
            s.printAsGraduateStudent();
        else if ...
}
```

**What if the number of cases grows?**

# Distinguishing Features of an OOPL

- Recap
  We've now defined all three of the features required to make a language truly object oriented:
  - (Programmer creation of) user-defined types
  - Inheritance
  - Polymorphism

# Distinguishing Features of an OOPL

*Benefits of User-Defined Types*

- User-defined types provide an intuitive way to represent real-world objects, resulting in ***easier-to-verify requirements***.
- Classes are convenient units of reusable code, which means ***less code to write from scratch when building an application.***
- Through encapsulation, we ***minimize data redundancy*** —each item of data is stored once, in the object to which it belongs—thereby ***lessening the likelihood of data integrity errors*** across an application.

# Distinguishing Features of an OOPL

*Benefits of User-Defined Types*

- Through information hiding, we ***insulate our application against ripple effects*** if private details of a class must change after deployment, thereby ***dramatically reducing maintenance costs.***

- Objects are responsible for ensuring the integrity of their own data, making it ***easier to isolate errors in an application's (business) logic***; we know to inspect the method(s) of the class to which a corrupted object belongs.

# Distinguishing Features of an OOPL

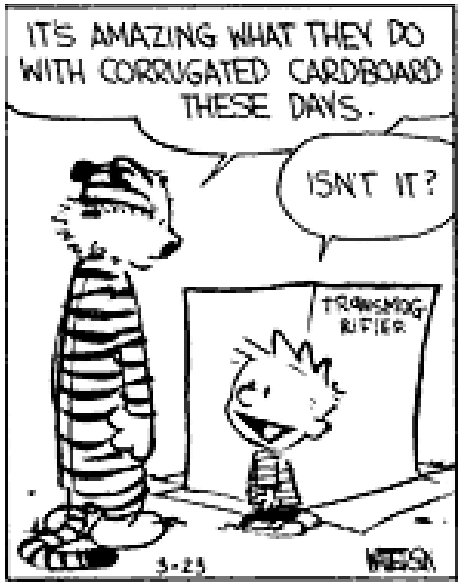
*Benefits of Inheritance*

- We can extend already deployed code without having to change and then retest it, resulting in ***dramatically reduced maintenance costs.***
- Subclasses are much more succinct, which means ***less code overall to write/maintain.***

# Distinguishing Features of an OOPL

*Benefits of Polymorphism*

- It ***minimizes "ripple effects"*** on client code when new subclasses are added to the class hierarchy of an existing application, resulting in ***dramatically reduced maintenance costs.***



# Abstract Classes

- We might determine up front that all Courses, regardless of type, are going to need the following common attributes:
  - String courseName
  - String courseNumber
  - int creditValue
  - *CollectionType enrolledStudents*
  - Professor instructor
- as well as the following common behaviors:
  - enrollStudent
  - assignInstructor
  - establishCourseSchedule

# Abstract Classes

```
import java.util.ArrayList;

public class Course {
    private String courseName;
    private String courseNumber;
    private int creditValue;
    private ArrayList enrolledStudents;
    private Professor instructor;

    // Accessor methods...

    public void enrollStudent(Student s) {
        enrolledStudents.add(s);
    }
    public void assignInstructor(Professor p) {
        setInstructor(p);
    }
    // What about establishCourseSchedule?
}
```

# Abstract Classes

```
import java.util.ArrayList;

public class Course {
    private String courseName;
    private String courseNumber;
    private int creditValue;
    private ArrayList enrolledStudents;
    private Professor instructor;

    // Accessor methods...

    public void enrollStudent(Student s) {
        enrolledStudents.add(s);
    }
    public void assignInstructor(Professor p) {
        setInstructor(p);
    }
}
```

**a generic, "one-size-fits-all" version?**

**omit the method from the Course class?**

# Abstract Classes

```
import java.util.ArrayList;

public abstract class Course {
    private String courseName;
    private String courseNumber;
    private int creditValue;
    private ArrayList enrolledStudents;
    private Professor instructor;

    // Accessor methods...

    public void enrollStudent(Student s) {
        enrolledStudents.add(s);
    }
    public void assignInstructor(Professor p) {
        setInstructor(p);
    }
    public abstract void establishCourseSchedule(String startDate,
                                                 String endDate);
}
```

# Abstract Classes

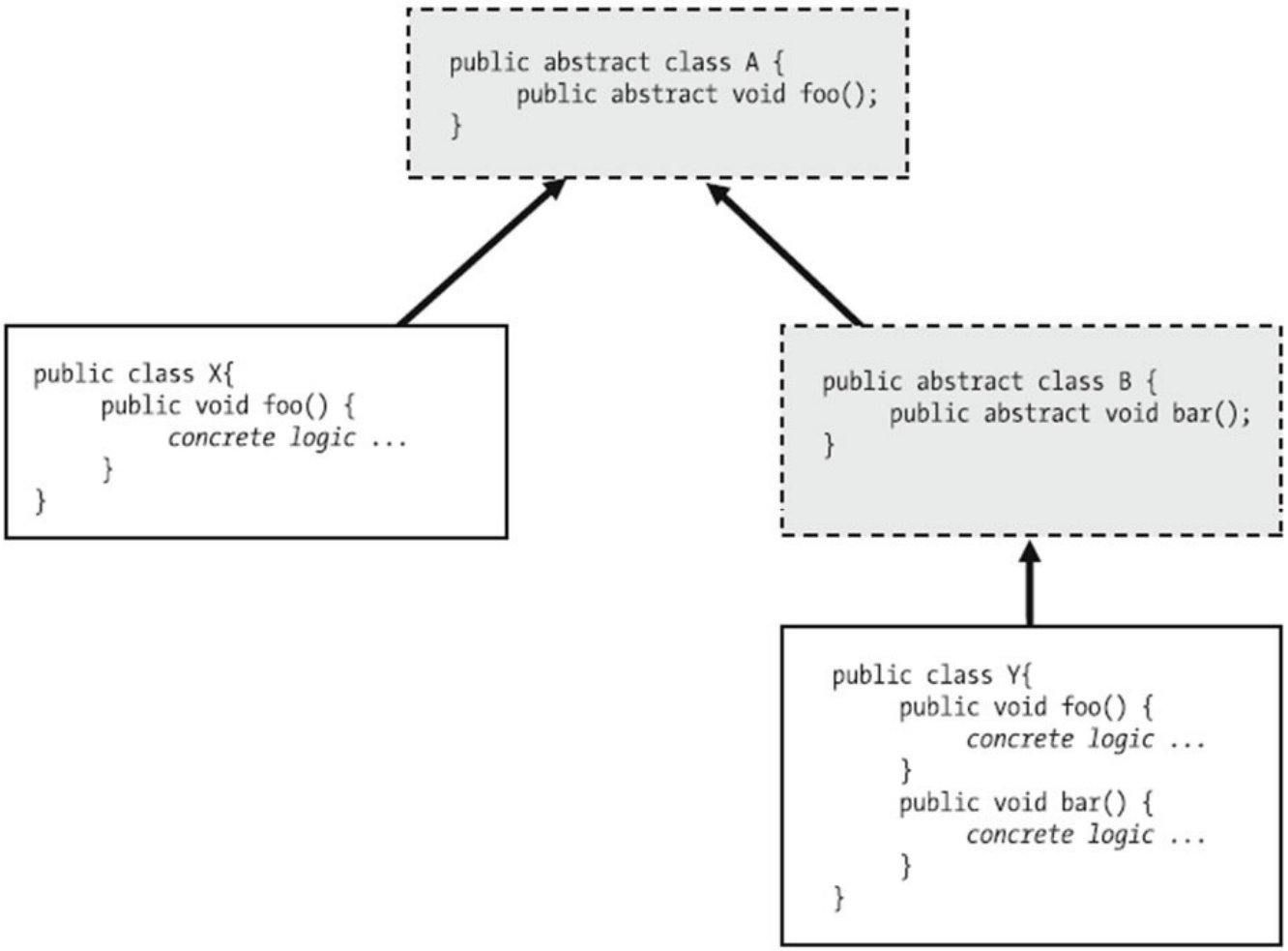
- By providing an abstract method in a superclass A, we've specified a service that all types of A objects must be able to perform, but without pinning down the private details of ***how*** the service should be performed by a given subclass.

- The subclasses are required to ***override*** the ***abstract method*** with an ***implemented version***

- Abstract classes ***cannot be instantiated.*** → **Compilation Error**
  - Abstract methods serve to enforce implementation requirements!

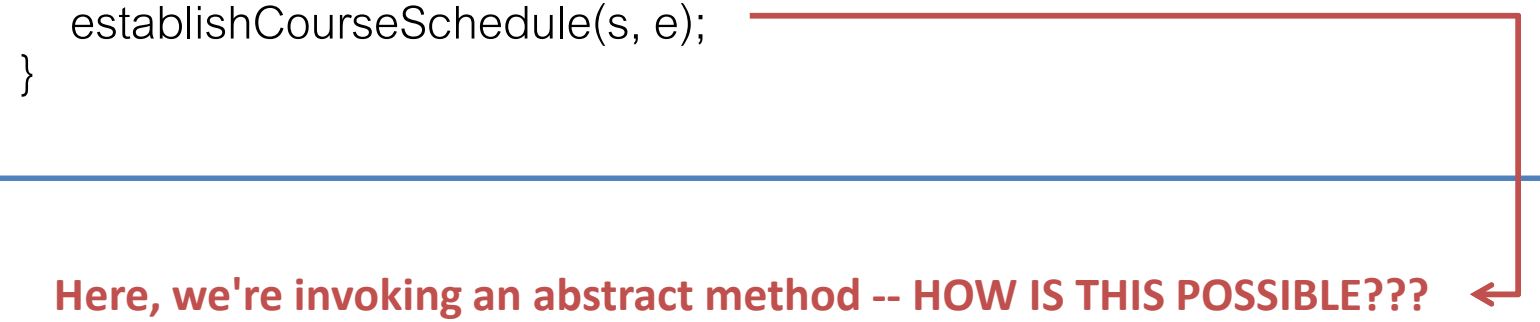
# Abstract Classes

*"Breaking the spell of abstractness"*



# Abstract Classes

```
public abstract class Course {
    // Details omitted.
    public void assignInstructor(Professor p){…}
    public void reserveClassroom(){…}

    public abstract void establishCourseSchedule(String startDate,
                                                 String endDate);

    public void initializeCourse(Professor p, String s, String e) {
        this.assignInstructor(p);
        this.reserveClassroom();

        establishCourseSchedule(s, e);
    }
}
```



**Here, we're invoking an abstract method -- HOW IS THIS POSSIBLE???**

# Interfaces

- let's take the notion of ***abstractness*** one step further.
  - Abstract class → we don't need to program the bodies of methods that are declared to be abstract.
  - But ***what about the data structure*** of such a class?
- Suppose we only wanted to specify common ***behaviors.***

# Interfaces

- let's take the notion of ***abstractness*** one step further.
  - Abstract class → we don't need to program the bodies of methods that are declared to be abstract.
  - But ***what about the data structure*** of such a class?
- Suppose we only wanted to specify common ***behaviors.***

```
public abstract class Teacher {
    // We omit attribute declarations entirely, allowing subclasses to establish
    // their own class-specific data structures.

    public abstract boolean agreeToTeach(Course c);
    public abstract void designateTextbook(TextBook b, Course c);
    public abstract Syllabus defineSyllabus(Course c);
    public abstract boolean approveEnrollment(Student s, Course c);
}
```

# Interfaces

- A Professor ***is capable of Teaching***

```
public class Professor extendsTeacher {
    // Declare relevant attributes.

    public boolean agreeToTeach(Course c) {…}
    public void designateTextbook(TextBook b, Course c) {…}
    public Syllabus defineSyllabus(Course c) {…}
    public boolean approveEnrollment(Student s, Course c) {…}

    //Additional Methods can also be declared
}
```

**Concrete Implementations of ALL inherited methods**

# Interfaces

- However, if our intention is to declare a set of abstract **method headers to define what it means to assume a certain *role*** within an application (i.e. teaching) **without imposing either data structure or concrete behavior on the subclasses**, then the preferred way to do so in Java is with an **interface**.

```
public interface Teacher {

    boolean agreeToTeach(Course c);
    void designateTextbook(TextBook b, Course c);
    Syllabus defineSyllabus(Course c);
    boolean approveEnrollment(Student s, Course c);
}
```

**All methods are implicitly public and abstract**
**We don't need to specify either of those two keywords**

# Interfaces

*Implementing Interfaces*

public class *ClassName* implements *PredefinedInterfaceName*

```
public class Professor implementsTeacher {
    // Declare relevant attributes.

    public boolean agreeToTeach(Course c) {…}
    public void designateTextbook(TextBook b, Course c) {…}
    public Syllabus defineSyllabus(Course c) {…}
    public boolean approveEnrollment(Student s, Course c) {…}

    //Additional Methods can also be declared
}
```

**Concrete Implementations of ALL inherited methods**

**What if a method is not implemented?**

# Interfaces

*Declaring Abstract Classes vs. Interfaces*

```
public abstract class Teacher {

    private String name;
    private String employeeId;
    // etc.

    public abstract void agreeToTeach(
        Course c);
    //more abstract methods…

    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Teacher {




    void agreeToTeach(Course c);
    //more abstract methods




}
```

# Interfaces

*Declaring Abstract Classes vs. Interfaces*

```
public abstract class Teacher {

    private String name;
    private String employeeId;
    // etc.

    public abstract void agreeToTeach(
        Course c);
    //more abstract methods…

    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Teacher {

    void agreeToTeach(Course c);
    //more abstract methods

}
```

**Declaration**

# Interfaces

*Declaring Abstract Classes vs. Interfaces*

```
public abstract class Teacher {

    private String name;
    private String employeeId;
    // etc.

    public abstract void agreeToTeach(
        Course c);
    //more abstract methods…

    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Teacher {

    void agreeToTeach(Course c);
    //more abstract methods

}
```

**Declaration – Data Structure**

# Interfaces

*Declaring Abstract Classes vs. Interfaces*

```
public abstract class Teacher {

    private String name;
    private String employeeId;
    // etc.

    public abstract void agreeToTeach(
        Course c);
    //more abstract methods…

    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Teacher {

    void agreeToTeach(Course c);
    //more abstract methods

}
```

**Declaration – Data Structure – Abstract methods**

# Interfaces

*Declaring Abstract Classes vs. Interfaces*

```
public abstract class Teacher {

    private String name;
    private String employeeId;
    // etc.

    public abstract void agreeToTeach(
        Course c);
    //more abstract methods…

    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Teacher {

    void agreeToTeach(Course c);
    //more abstract methods

}
```

**Declaration – Data Structure – Abstract methods – Concrete methods**

# Interfaces

*Extending Abstract Classes vs. Implementing Interfaces*

```
public class Professor extends Teacher {



    private Department worksFor;

    public void agreeToTeach(Course c){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

```
public class Professor implements
    Teacher {

    private String name;
    private String employeeId;
    private Department worksFor;

    public void agreeToTeach(Coursec){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

# Interfaces

*Extending Abstract Classes vs. Implementing Interfaces*

```
public class Professor extends Teacher {



    private Department worksFor;

    public void agreeToTeach(Course c){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

```
public class Professor implements
    Teacher {

    private String name;
    private String employeeId;
    private Department worksFor;

    public void agreeToTeach(Coursec){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

**Keywords**

# Interfaces

*Extending Abstract Classes vs. Implementing Interfaces*

```
public class Professor extends Teacher {



    private Department worksFor;

    public void agreeToTeach(Course c){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

```
public class Professor implements
    Teacher {

    private String name;
    private String employeeId;
    private Department worksFor;

    public void agreeToTeach(Coursec){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

**Keywords – Data Structure**

# Interfaces

*Extending Abstract Classes vs. Implementing Interfaces*

```
public class Professor extends Teacher {



    private Department worksFor;

    public void agreeToTeach(Course c){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

```
public class Professor implements
    Teacher {

    private String name;
    private String employeeId;
    private Department worksFor;

    public void agreeToTeach(Coursec){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

**Keywords** – **Data Structure** – **Abstract methods**

# Interfaces

*Extending Abstract Classes vs. Implementing Interfaces*

```
public class Professor extends Teacher {



    private Department worksFor;

    public void agreeToTeach(Course c){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

```
public class Professor implements
    Teacher {

    private String name;
    private String employeeId;
    private Department worksFor;

    public void agreeToTeach(Coursec){
        //…
    }

    public void print() {
        System.out.println(name);
    }
}
```

**Keywords – Data Structure – Abstract methods – Concrete methods**

# Interfaces

*Another Form of the "Is A" Relationship*

- If the Professor class ***extends*** the Person class, then a Professor ***is a*** Person.
- If the Professor class ***implements*** the Teacher interface, then a Professor ***is a*** Teacher.

- When a class A implements an interface X, all of the classes that are subsequently derived from A may also be said to implement that same interface X.

```
public class AdjuntProfessor extends Teacher {…}
```

# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
public class Student implements Teacher {…}
```

```
Professor p1 = new Professor();
Student s = new Student();
Teacher t;

s = t;
```

**compiler ERROR!!!**

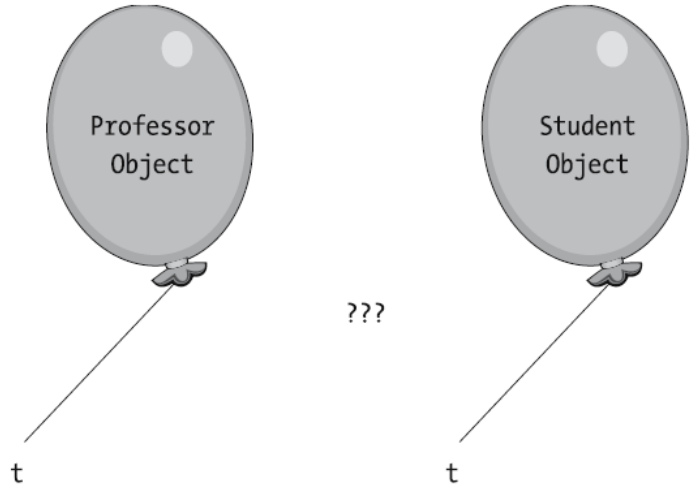
# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
public class Student implements Teacher {…}
```

```
Professor p1 = new Professor();
Student s = new Student();
Teacher t;

t = s;
t = p1;
```



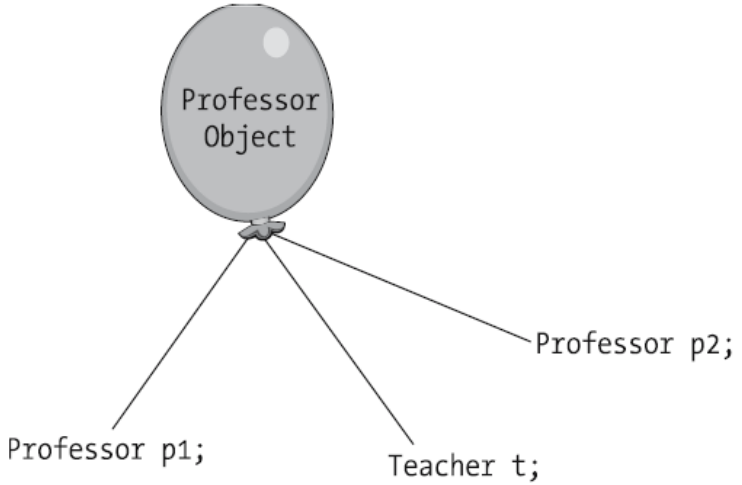
# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
public class Student implements Teacher {…}
```

```
Professor p1 = new Professor();
Student s = new Student();
Teacher t;

t = s;
t = p1;

Professor p2 = (Professor) t
```



# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
public class Student implements Teacher {…}
```

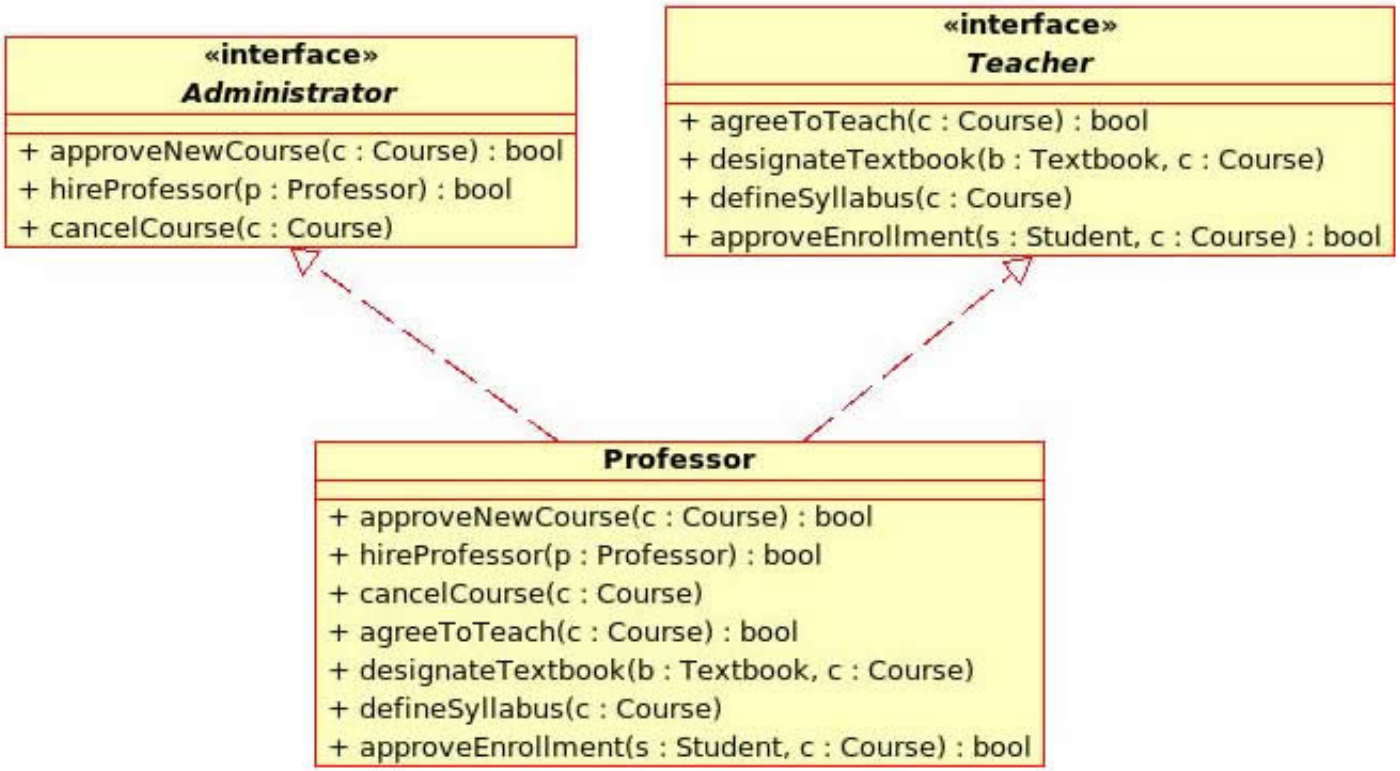
```
Professor p1 = new Professor();
Student s = new Student();
Teacher t;

t = s;
t = p1;

Professor p2 = (Professor) t

t = s;
Professor p2 = (Professor) t
```

**run time ERROR!!!**

# Interfaces

*Implementing Multiple Interfaces*

public class *ClassName* implements *Interface1, Interface2,…, InterfaceN*



# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
Professor p1 = new Professor();
Teacher t;

t = p1;

t.setDepartment("Computer Science");
```

**compiler ERROR!!!** 

**We know we can do this → Every Professor is a Teacher**

# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
Professor p1 = new Professor();
Teacher t;

t = p1;

((Professor) t).setDepartment("Computer Science");
```

**Casting to the rescue**

# Interfaces

*Interfaces and Casting*

```
public class Professor implements Teacher {…}
```

```
Professor p1 = new Professor();
Teacher t;

t = p1;

((Professor) t).setDepartment("Computer Science");
```

```
(Professor) t.setDepartment("Computer Science");
```

**compiler ERROR!!!**

**Cast the result of** t.setDepartment("Computer Science"); **as a Professor**

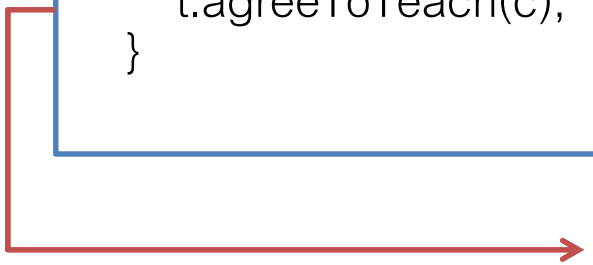
# Interfaces

*Interfaces and Polymorphism*

```
public class Professor implements Teacher {…}
```

```
public class Student implements Teacher {…}
```

```
ArrayList<Teacher> teachers = new ArrayList<Teacher>();
teachers.add(new Student("Becky Elkins"));
teachers.add(new Professor("Bobby Cranston"));
// etc.

for (Teacher t : teachers) {
    t.agreeToTeach(c);
}
```



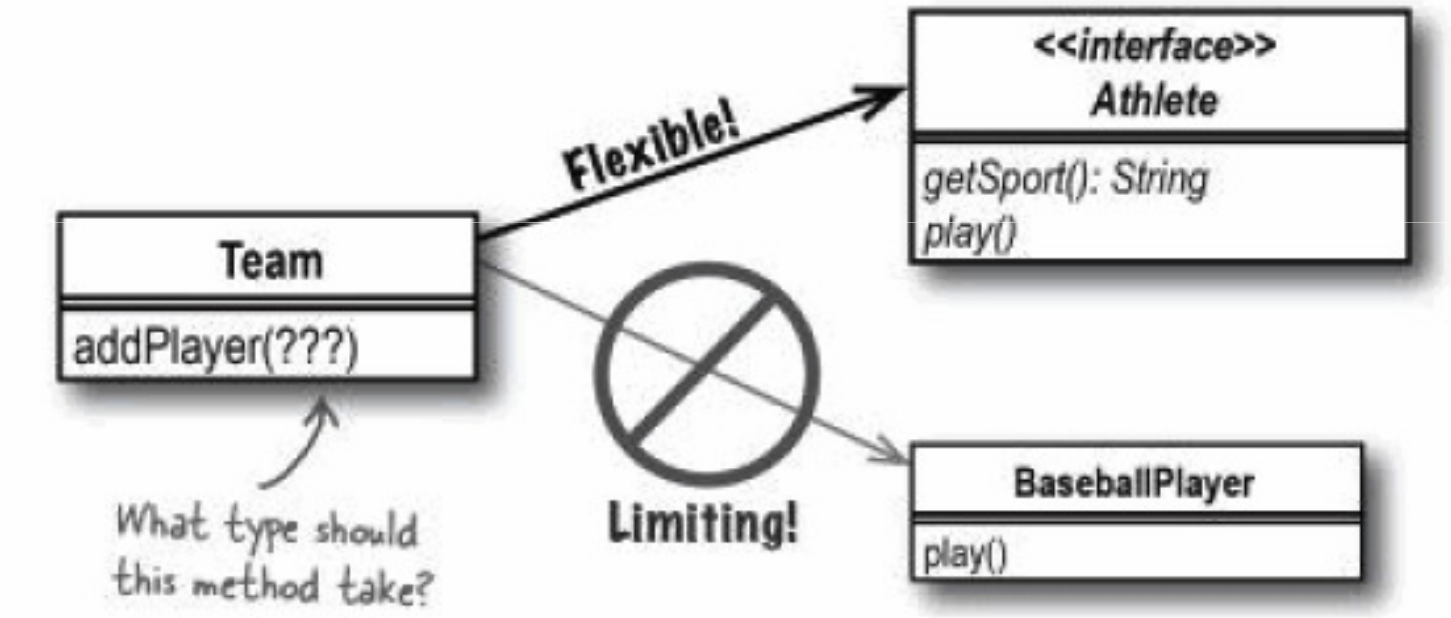
**This line is polymorphic**

# Interfaces

*The importance of Interfaces*

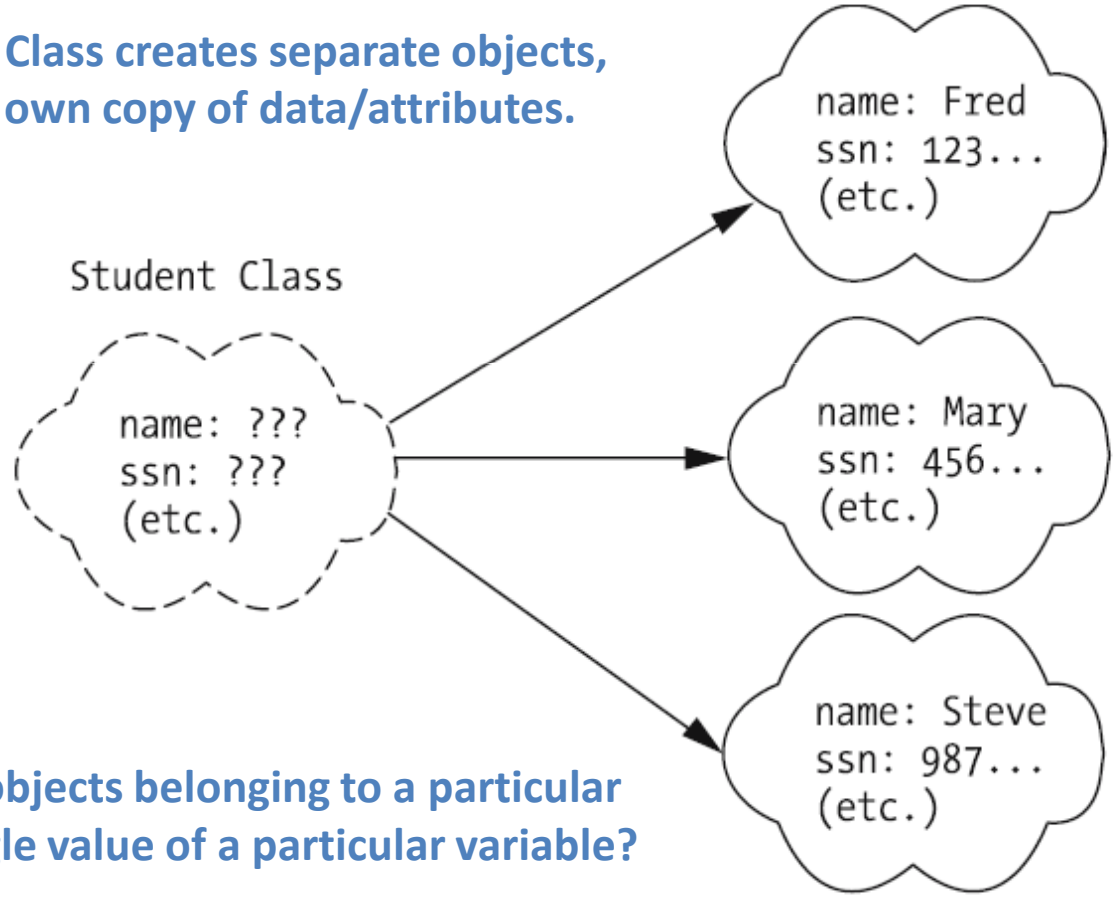
# Interfaces

*The importance of Interfaces*



# Static Features

**Instantiating a Class creates separate objects, each with its own copy of data/attributes.**




**but, what if all objects belonging to a particular class share a single value of a particular variable?**

# Static Features

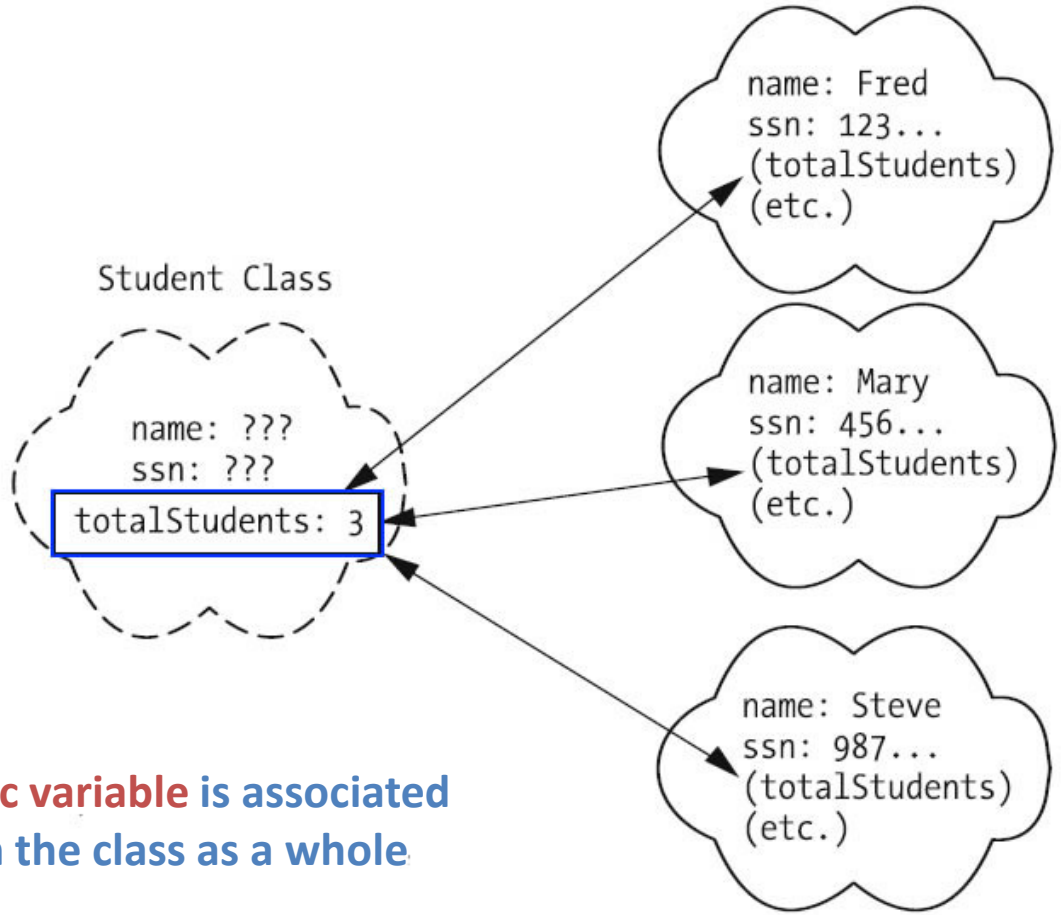
```
public class Student {
    private int totalStudents;
    // Other attribute details omitted.

    public int getTotalStudents() {
        return totalStudents;
    }
    public void setTotalStudents(int x) {
        totalStudents = x;
    }

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment: " + getTotalStudents());
    }
    public void incrementEnrollment() {
        setTotalStudents(getTotalStudents() + 1);
    }
}
```

# Static Features

```
Student s1 = new Student();

s1.incrementEnrollment();
//...

Student s2 = new Student();

// ...and have to remember to increment the enrollment count for BOTH.
s1.incrementEnrollment();
s2.incrementEnrollment();


Student s3 = new Student();
//…
```

# Static Features

*Static Variables*




**A static variable is associated with the class as a whole**

# Static Features

*Static Variables*

```
public class Student {
    private static int totalStudents;
    // Other attribute details omitted.

    public int getTotalStudents() {
        return totalStudents;
    }
    public void setTotalStudents(int x) {
        totalStudents = x;
    }

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment: " + getTotalStudents());
    }
    public void incrementEnrollment() {
        setTotalStudents(getTotalStudents() + 1);
    }
}
```

# Static Features

*Static Variables*

```
Student s1 = new Student();

s1.incrementEnrollment();
//...

Student s2 = new Student();
s2.incrementEnrollment();

Student s3 = new Student();
s2.incrementEnrollment();

s1.reportTotalEnrollment();
s2.reportTotalEnrollment();
s3.reportTotalEnrollment();
```

# Static Features

*Static Variables*

```
Student s1 = new Student();

s1.incrementEnrollment();
//...

Student s2 = new Student();
s2.incrementEnrollment();

Student s3 = new Student();
s2.incrementEnrollment();

s1.reportTotalEnrollment();
s2.reportTotalEnrollment();
s3.reportTotalEnrollment();
```

**Try this code:**

**System.out.println("Total Enrollment: " + Student.totalStudents);**

# Static Features

*Burying Implementation Details*

```
public class Student {
    private static int totalStudents;
    // Other attribute details omitted.

    public int getTotalStudents() {
        return totalStudents;
    }
    public void setTotalStudents(int x) {
        totalStudents = x;
    }

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment: " + getTotalStudents());
    }
    public void incrementEnrollment() {
        setTotalStudents(getTotalStudents() + 1);
    }
}
```

# Static Features

*Burying Implementation Details*

```
public class Student {
    private static int totalStudents;

    public Student(…) {
        //…
        totalStudents++;
    }

    public int getTotalStudents() {…}
    public void setTotalStudents(int x) {…}

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment: " + getTotalStudents());
    }
}
```

**whenever possible, it's desirable to bury such implementation details inside of a class, to lessen the burden on client code, and hence, the likelihood for logic errors.**

# Static Features

*Static Methods*

```
public class Student {
    private static int totalStudents;

    public Student(…) {
        //…
        totalStudents++;
    }

    public static int getTotalStudents() {…}
    public static void setTotalStudents(int x) {…}
    public static int reportTotalEnrollment() {…}
}
```

```
    Student.reportTotalEnrollment();
    s.reportTotalEnrollment();
```

# Static Features

*Restrictions on Static Methods*

```
public class Student {
    public abstract static void incrementEnrollment();
}
```

**compiler ERROR!!!**

```
public class Student {
    private String name;
    private static int totalStudents;

    public static int getTotalStudents() { ... }
    public static void setTotalStudents(int x) { ... }
    public String getName() { ... }
    public void setName(String n) { ... }
    public static void print() {
        System.out.println(getName() + " is one of " + getTotalStudents() +
            "students.");
    }
}
```

**compiler ERROR!!!**

**This is a NONstatic feature**

# Static Features

*The **Final** Keyword*

- The Java final keyword can be applied to variables, methods, and classes as a whole.
- A **final variable** is a variable that can be assigned a value only once in a program; after that first assignment, the variable's value cannot be changed.

```
public class Example {
    public static final int x;
    private final int y;

    public void someMethod() {
        final int z;
        z = 3;
        x = 1;
        y = 2;
    }
}
```

**compiler ERROR!!!**
**compiler ERROR!!!**

# Static Features

*Public Static Final Variables and Interfaces*

- As stated earlier, **interfaces are not permitted to declare variables**, but for **one exception**: as it turns out, interfaces are allowed to declare **public static final variables** to serve as ***global constants*—**that is, constant values that are in scope and hence **accessible throughout an entire application.**

```
public interface Administrator {
    public static final int FULL_TIME = 1;
    public static final int PART_TIME = 2;

    // Valid values for workStatus are FULL_TIME (1) or PART_TIME (2).

    public boolean hireProfessor(Professor p, int workStatus);
    //…
}
```

# Static Features

*Public Static Final Variables and Interfaces*

- As stated earlier, **interfaces are not permitted to declare variables**, but for **one exception**: as it turns out, interfaces are allowed to declare **public static final variables** to serve as ***global constants*—**that is, constant values that are in scope and hence **accessible throughout an entire application.**

```
Administrator pAdmin = new Professor();
Professor p = new Professor();

// Hire p as a full-time faculty member.
pAdmin.hireProfessor(p, Administrator.FULL_TIME);
```

# References


- J. Barker, *Beginning Java Objects: From Concepts To Code, Second Edition*, Apress, 2005.
- Java™ Platform, Standard Edition 6 – The Collection Framework. Available online at: http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html