

# Lecture 04: Composition and Inheritance

## CS 0445: Data Structures

**Constantinos Costa**

<http://db.cs.pitt.edu/courses/cs0445/current.term/>

Sep 10, 2019, 8:00-9:15  
University of Pittsburgh, Pittsburgh, PA



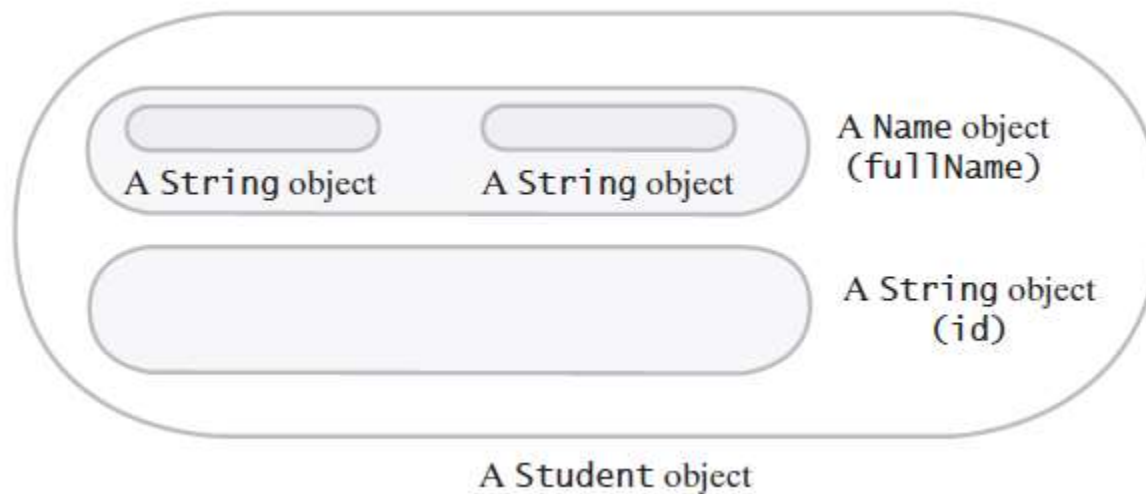
# Composition

- A class uses composition when it has a data field that is an instance of another class
- Composition is a “has a” relationship
- Consider a class of students, each has
  - A name, an identification number.
- Thus, class Student contains two objects as data fields:
  - An instance of the class Name
  - An instance of the class String:



# Composition

A Student object is composed of other objects



# Composition

## • LISTING 2.4 The Student Class

```
1 public class Student
2 {
3     private Name    fullName;
4     private String id;    // Identification number
5
6     public Student()
7     {
8         fullName = new Name();
9         id = "";
10    } // end default constructor
11
12    public Student(Name studentName, String studentId)
13    {
14        fullName = studentName;
15        id = studentId;
16    } // end constructor
17
18    public void setStudent(Name studentName, String studentId)
19    {
20        setName(studentName); // Or fullName = studentName;
21        setId(studentId);     // Or id = studentId;
22    } // end setStudent
23
```



# Composition

## •LISTING D-1 The class Student

```
23
24     public void setName(Name studentName)
25     {
26         fullName = studentName;
27     } // end setName
28
29     public Name getName()
30     {
31         return fullName;
32     } // end getName
33
34     public void setId(String studentId)
35     {
36         id = studentId;
37     } // end setId
38
39     public String getId()
40     {
41         return id;
42     } // end getId
43
44     public String toString()
45     {
46         return id + " " + fullName.toString();
47     } // end toString
48 } // end Student
```



# Adapters

- Consider reuse of a class where ...
  - Names of its methods do not suit your application
  - You want to simplify some methods
  - Or eliminate others
- An adapter class
  - Uses composition to write a new class that has an instance of your existing class as a data field
  - Defines the methods that you want



# Adapters

- The class NickName

```
1 public class NickName
2 {
3     private Name nick;
4
5     public NickName()
6     {
7         nick = new Name();
8     } // end default constructor
9
10    public void setNickName(String nickName)
11    {
12        nick.setFirst(nickName);
13    } // end setNickName
14
15    public String getNickName()
16    {
17        return nick.getFirst();
18    } // end getNickName
19 } // end NickName
```



# Inheritance

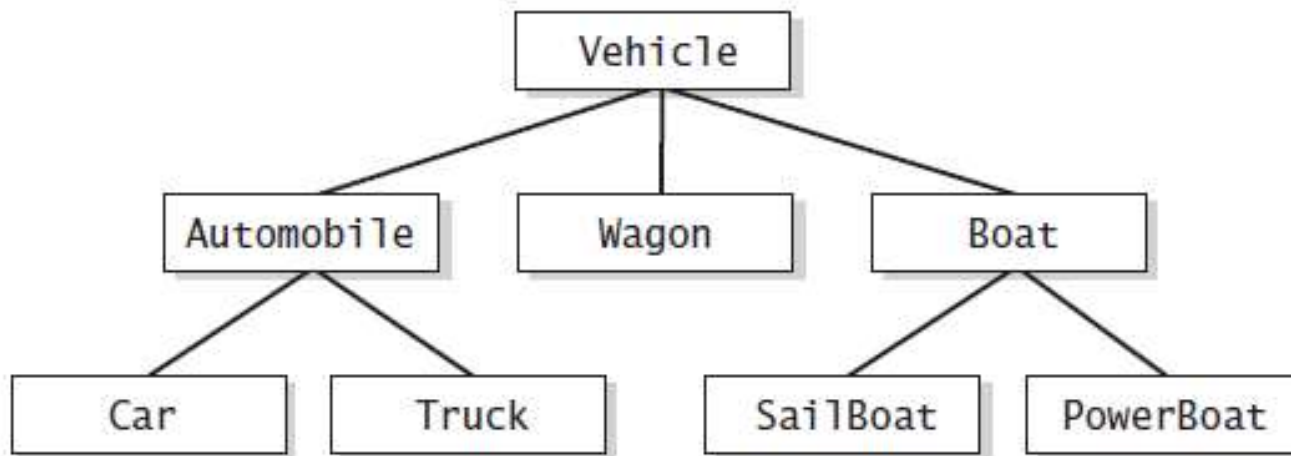
- Allows you to define general class
  - Then later to define more specialized classes
  - Add to or revise the details of the older, more general class definition
- Inheritance is an “is a” relationship
- Example: general class of vehicles
  - Subclasses automobile, wagon, and boat, etc.





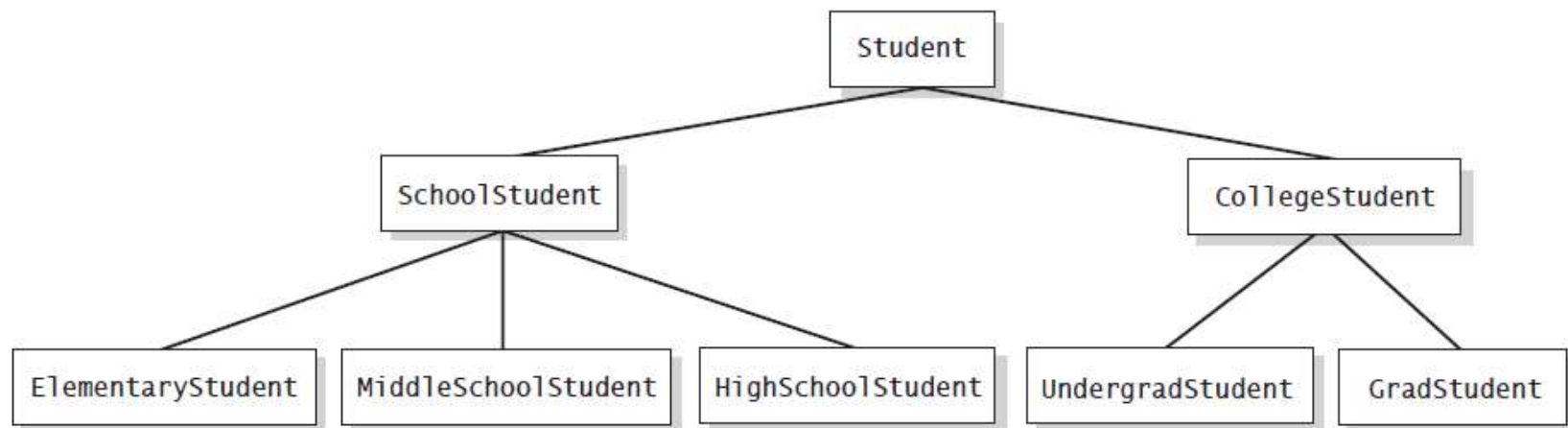
# Inheritance

A hierarchy of vehicle classes



# Inheritance

A hierarchy of student classes



# Inheritance

- The class CollegeStudent

```
1 public class CollegeStudent extends Student
2 {
3     private int    year;    // Year of graduation
4     private String degree; // Degree sought
5
6     public CollegeStudent()
7     {
8         super();    // Must be first statement in constructor
9         year = 0;
10        degree = "";
11    } // end default constructor
12
13    public CollegeStudent(Name studentName, String studentId,
14                          int graduationYear, String degreeSought)
15    {
16        super(studentName, studentId); // Must be first
17        year = graduationYear;
18        degree = degreeSought;
19    } // end constructor
20
21    public void setStudent(Name studentName, String studentId,
```



# Inheritance

- The class CollegeStudent

```
19     } // end constructor
20
21     public void setStudent(Name studentName, String studentId,
22                           int graduationYear, String degreeSought)
23     {
24         setName(studentName); // NOT fullName = studentName;
25         setId(studentId);      // NOT id = studentId;
26         // Or setStudent(studentName, studentId); (see Segment D.16)
27
28         year = graduationYear;
29         degree = degreeSought;
30     } // end setStudent
31     < The methods setYear, getYear, setDegree, and getDegree go here. >
32     . . .
33     public String toString()
34     {
35         return super.toString() + ", " + degree + ", " + year;
36     } // end toString
37 } // end CollegeStudent
```



## Invoking Constructors from Within Constructors

- Constructors typically initialize a class's data fields
- To call constructor of superclass explicitly.
  - Use **super( )** within definition of a constructor of a subclass
- If you omit **super( )**
  - Constructor of subclass automatically calls default constructor of superclass.



# Invoking Constructors from Within Constructors

- Also possible to use **this** to invoke constructor of superclass

```
public CollegeStudent(Name studentName, String studentId)
{
    this(studentName, studentId, 0, "");
} // end constructor
```



# Private Fields and Methods of the Superclass

- Only a method in the class **Student** can access **fullName** and **id** directly by name from within its definition.
- Although the class **CollegeStudent** inherits these data fields,
  - None of its methods can access them by name
- Instead it must use some public mutator method such as **setId**.



# Overriding and Overloading Methods

- Possible to new method invoke the inherited method
  - Need to distinguish between the method for subclass and method from superclass

```
public String toString()  
{  
    return super.toString() + ", " + degree + ", " + year;  
} // end toString
```





# Overriding and Overloading Methods

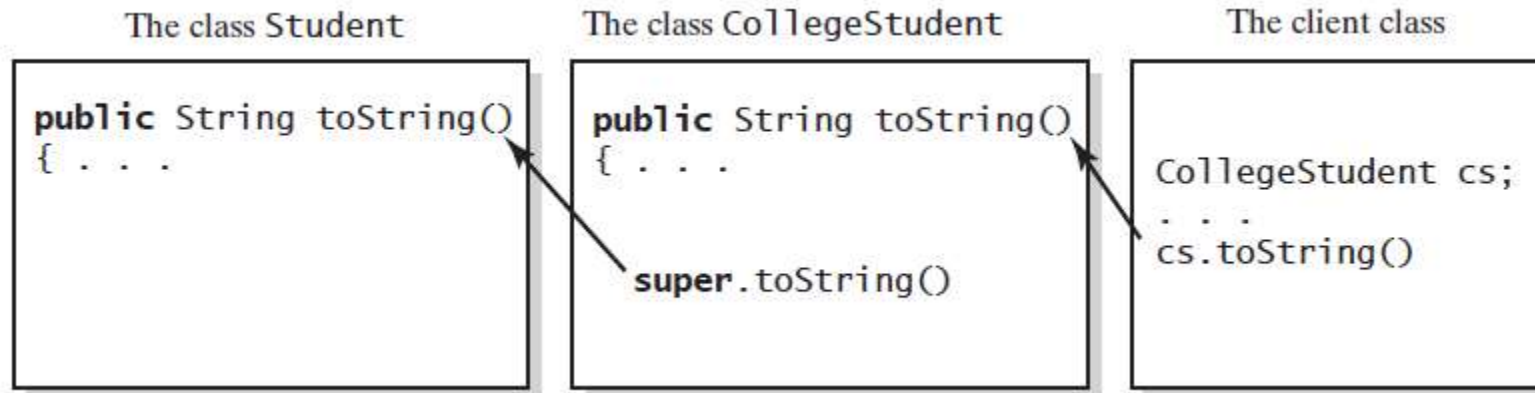
- When a subclass defines a method with
  - the same name
  - the same number and types of parameters
  - and the same return type as a method in the superclass

...
- Then definition in the subclass is said to *override* the definition in the superclass.
- You can use **super** in a subclass to call an overridden method of the superclass.



# Overriding and Overloading Methods

The method **toString** in **CollegeStudent** overrides the method **toString** in **Student**



# Overriding and Overloading Methods

- When subclass has a method with same name as a method in its superclass,
  - but the methods' parameters differ in number or data type
  - ...
- Method in subclass overloads method of superclass.
  - Java is able to distinguish between these methods
  - Signatures of the methods are different



# Overriding and Overloading Methods

- Possible to call an overridden method of the superclass by prefacing the method name with **super** and a dot.
- But ... repeated use of **super** is not allowed

```
super.super.toString(); // ILLEGAL!
```




# Overriding and Overloading Methods

- To specify that a method definition cannot be overridden with a new definition in a subclass
  - Make it a final method by adding the **final** modifier to the method header.

```
public class C
{
    . . .
    public C()
    {
        m();
    } // end default constructor
    public void m()
    {
        . . .
    } // end m
    . . .
}
```

public final void m()



# Multiple Inheritance

- Some programming languages allow one class to be derived from two different superclasses
  - This feature not allowed in Java.
- In Java, a subclass can have only one superclass



# Type Compatibility and Superclasses

- An object of a subclass has more than one data type.
- Everything that works for objects of an ancestor class also works for objects of any descendant class.



# Type Compatibility and Superclasses

- Given **CollegeStudent**, subclass of **Student**
- Legal calls

```
Student amy = new CollegeStudent();  
Student brad = new UndergradStudent();  
CollegeStudent jess = new UndergradStudent();
```

- Illegal calls

```
CollegeStudent cs = new Student();           // ILLEGAL!  
UndergradStudent ug = new Student();         // ILLEGAL!  
UndergradStudent ug2 = new CollegeStudent(); // ILLEGAL!
```





# The Class Object

- Java has a class—named **Object**
  - It is at the beginning of every chain of subclasses
  - An ancestor of every other class
- Class **Object** contains certain methods
  - Examples: **toString**, **equals**, **clone**
  - However, in most cases, you must override these methods



# The Class Object

- Inherited version of **toString** returns value based upon invoking object's memory address.
- Need to override the definition of **toString**
  - Cause it to produce an appropriate string for data in the class being defined



# The Class Object

- Object's **equals** method compares the addresses of two objects
  - Overridden method, when added to the class **Name**, detects whether two **Name** objects are equal by comparing their data fields:

```
public boolean equals(Object other)
{
    boolean result = false;

    if (other instanceof Name)
    {
        Name otherName = (Name)other;
        result = first.equals(otherName.first) &&
                  last.equals(otherName.last);
    } // end if

    return result;
} // end equals
```

