

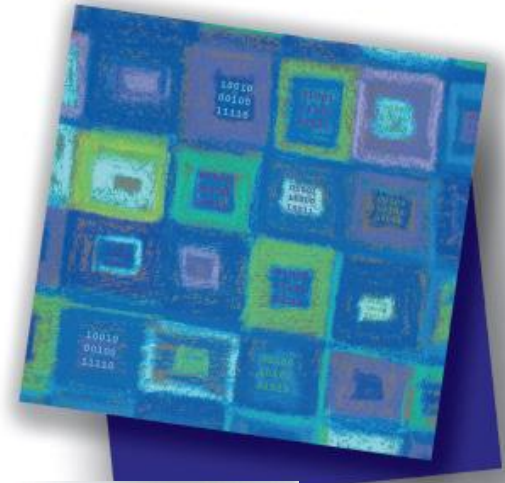
This week

- Tools we will use in making our Data Structure classes:
 - Generic Types
 - Inheritance
 - Abstract Classes and Interfaces
- This is a lot of material but we'll be working with these tools the whole semester

Generic Types

(One last thing from
Appendix B)

**Data Structures and
Abstractions with Java™**
SECOND EDITION



Frank M. Carrano

Slides by Steve Armstrong
LeTourneau University
Longview, TX
© 2007, Prentice Hall

Generic Types

- Consider a class with fields that can be of any class type
 - Use a generic type
 - Follow class name with an identifier enclosed in angle brackets
- ```
public class MyClass <T>
```
- [View the OrderedPair class](#) which takes ordered pairs of any type
  - Note sample code using this generic class

# More than one generic type

- In the previous example, the objects in a pair have **either the same** data type or data types related by inheritance.
- You can define more than one generic type within a class definition by writing their identifiers, separated by commas, within the angle brackets after the class's name, as in the class `Pair` shown in Listing B-6.
- [View the `Pair` class](#)

**Question 23** Can you use the class `OrderedPair`, as defined in **Listing B-5**, to pair two objects having different and unrelated data types? Why or why not?

**Question 24** Can you use the class `Pair`, as defined in the previous segment, to pair two objects having the same data type? Why or why not?

**Question 25** Using the class `Name`, as defined previously in this appendix, write statements that pair two students as lab partners.

**Question 26** Using the class `Name`, as defined previously in this appendix, write statements that pair your name with the random sequence number given in the `int` variable `number`.

**23. No. The class defines only one generic type.**

**24. Yes. You can write the same data type twice to correspond to both S and T.**

**25. Name kristen = new Name("Kristen", "Doe");**

**Name luci = new Name("Luci", "Lei");**

**OrderedPair<Name> labPartners =  
new OrderedPair<Name>(kristen, luci);**

**26. Name kristen = new Name("Kristen", "Doe");**

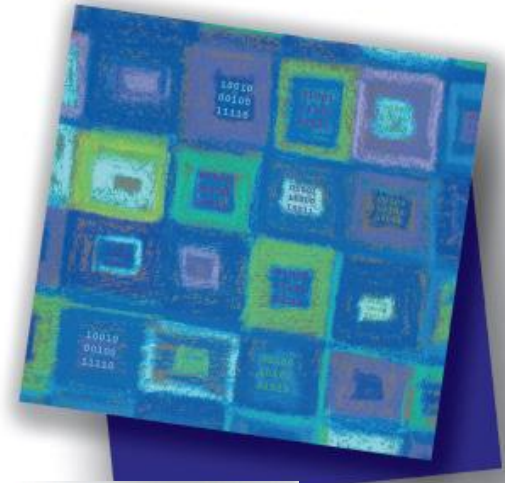
**Integer seqN = number;**

**Pair<Name, Integer> aPair = new Pair<Name, Integer>(kristen,  
seqN);**

# Creating Classes from Other Classes

## Appendix C

**Data Structures and  
Abstractions with Java™**  
SECOND EDITION



**Frank M. Carrano**

Slides by Steve Armstrong  
LeTourneau University  
Longview, TX  
© 2007, Prentice Hall

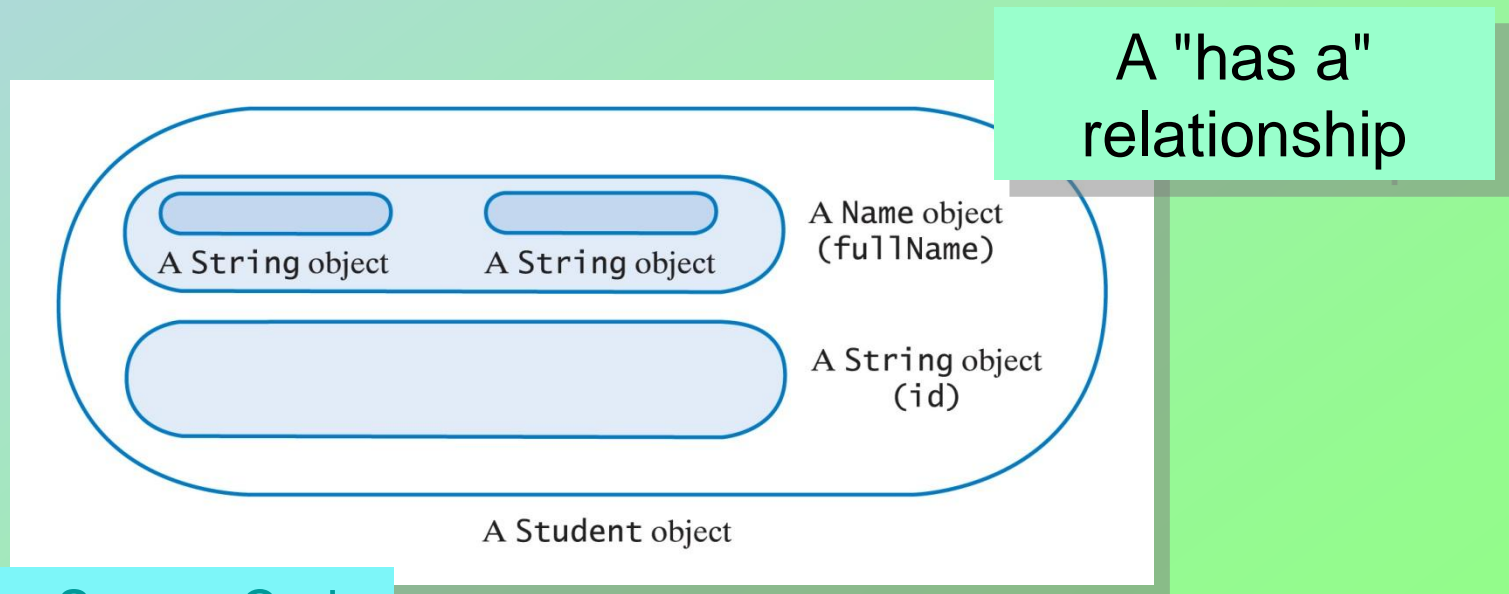
# Chapter Contents

- Composition
  - Generic Types
  - Adapters
- Inheritance
  - Invoking Constructors from Within Constructors
  - Private Fields and Methods of The Base Class
  - Protected Access
  - Overriding, Overloading Methods
  - Multiple Inheritance
- Type Compatibility and Base Classes
  - The Class `Object`
  - Abstract Classes and Methods
- Polymorphism



# Composition

- When a class has a data field that is an instance of another class
- Example – an object of type **Student**.



[Click to View Source Code](#)

Fig. 2-1 A **Student** object composed of other objects

Question 1 What data fields would you use in the definition of a class Address to represent a student's address?

Question 2 Add a data field to the class Student to represent a student's address. What new methods should you define?

Question 3 What existing methods need to be changed in the class Student as a result of the added field that Question 2 described?

Question 4 What is another implementation for the default constructor that uses this, as described in Segment B.25 of Appendix B?

**1.**

**Some possibilities are roomNumber and dorm, or street, city, state, zip.**

**2.**

**private Address residence;**

**Add the methods setAddress and getAddress.**

**3.**

**The constructors, setStudent, and toString.**

**4.**

**public Student()**

**{**

**this(new Name(), "");**

**} // end default constructor**

# Adapters

- Use composition to write a new class
  - Has an instance of an existing class as a data field
  - Defines new methods needed for the new class
- Example – a **NickName** class adapted from class **Name**
- [View source code](#) of class **NickName**

Question 5 Write statements that define bob as an instance of NickName to represent the nickname Bob. Then, using bob, write a statement that displays Bob.

5.

```
NickName bob = new NickName();
```

```
bob.setNickName("Bob");
```

```
System.out.println(bob.getNickName());
```

# Inheritance

- A general or base class is first defined
- Then a more specialized class is defined by ...
  - Adding to details of the base class
  - Revising details of the more general class
- Advantages
  - Saves work
  - Common properties and behaviors are define only once for all classes involved

# Inheritance

An "is a" relationship

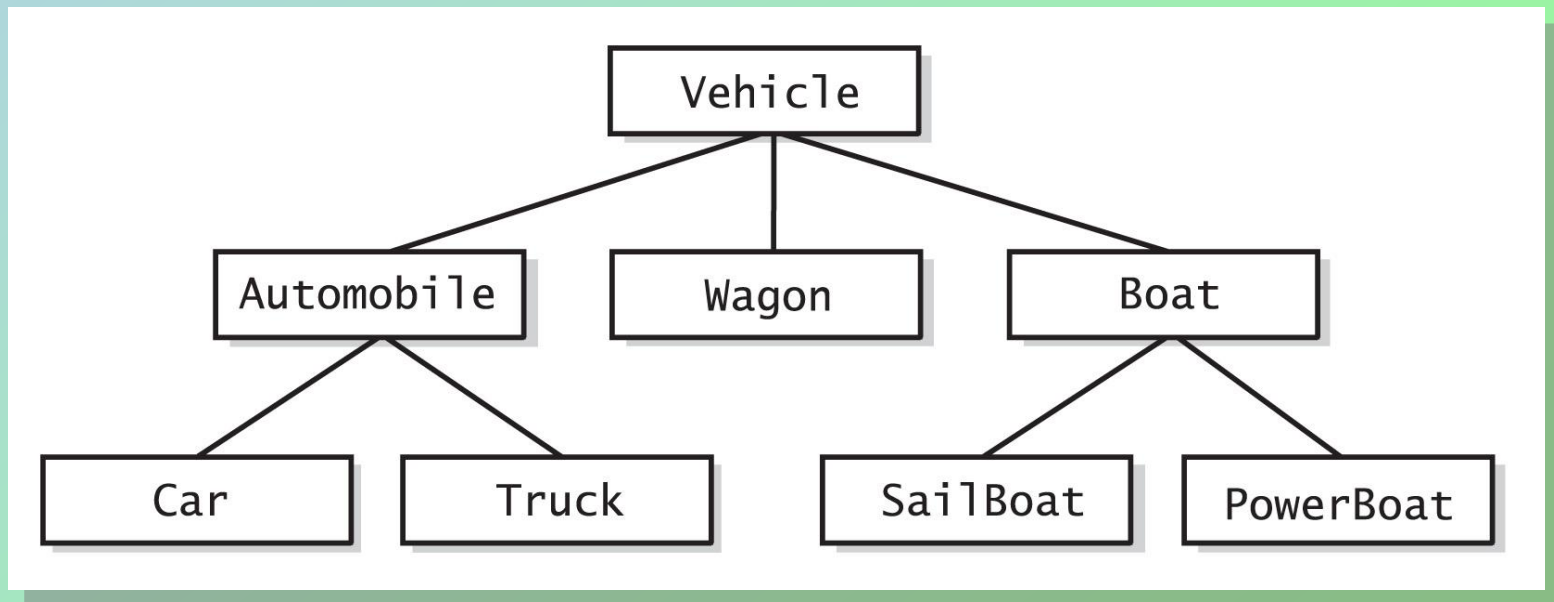


Fig. 2-2 A hierarchy of classes.



Question 6 Some vehicles have wheels and some do not. Revise Figure C-2 to organize vehicles according to whether they have wheels.

**6.**

**The Vehicle class has two subclasses, WheeledVehicle and WheellessVehicle. The subclasses of WheeledVehicle are Automobile and Wagon. Boat is a subclass of WheellessVehicle. The remaining subclasses are the same as given in the figure.**

# Inheritance

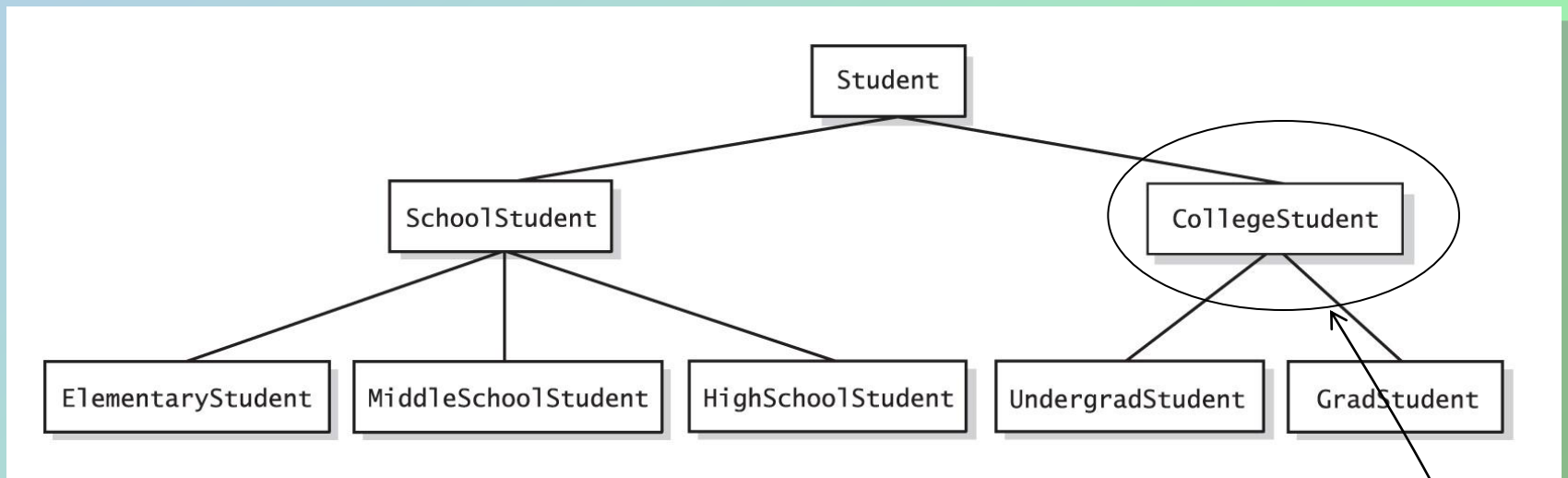


Fig. 2-3 A hierarchy of student classes.

- [View source code](#) of class **CollegeStudent**

# Invoking Constructors from Within Constructors

- Constructors usually initialize data fields
- In a derived class
  - The constructor must call the base class constructor
- Note use of reserved word **super** as a name for the constructor of the base class
  - When **super** is used, it must be the first action in the derived constructor definition
  - Must not use the name of the constructor

# Private Fields, Methods of Base Class

- Accessing inherited data fields
  - Not accessible by name within definition of a method from another class – including a derived class
  - Still they are inherited by the derived class
- Derived classes must use public methods of the base class
- Note that private methods in a base class are also unavailable to derived classes
  - But usually not a problem – private methods are used only for utility duties within their class

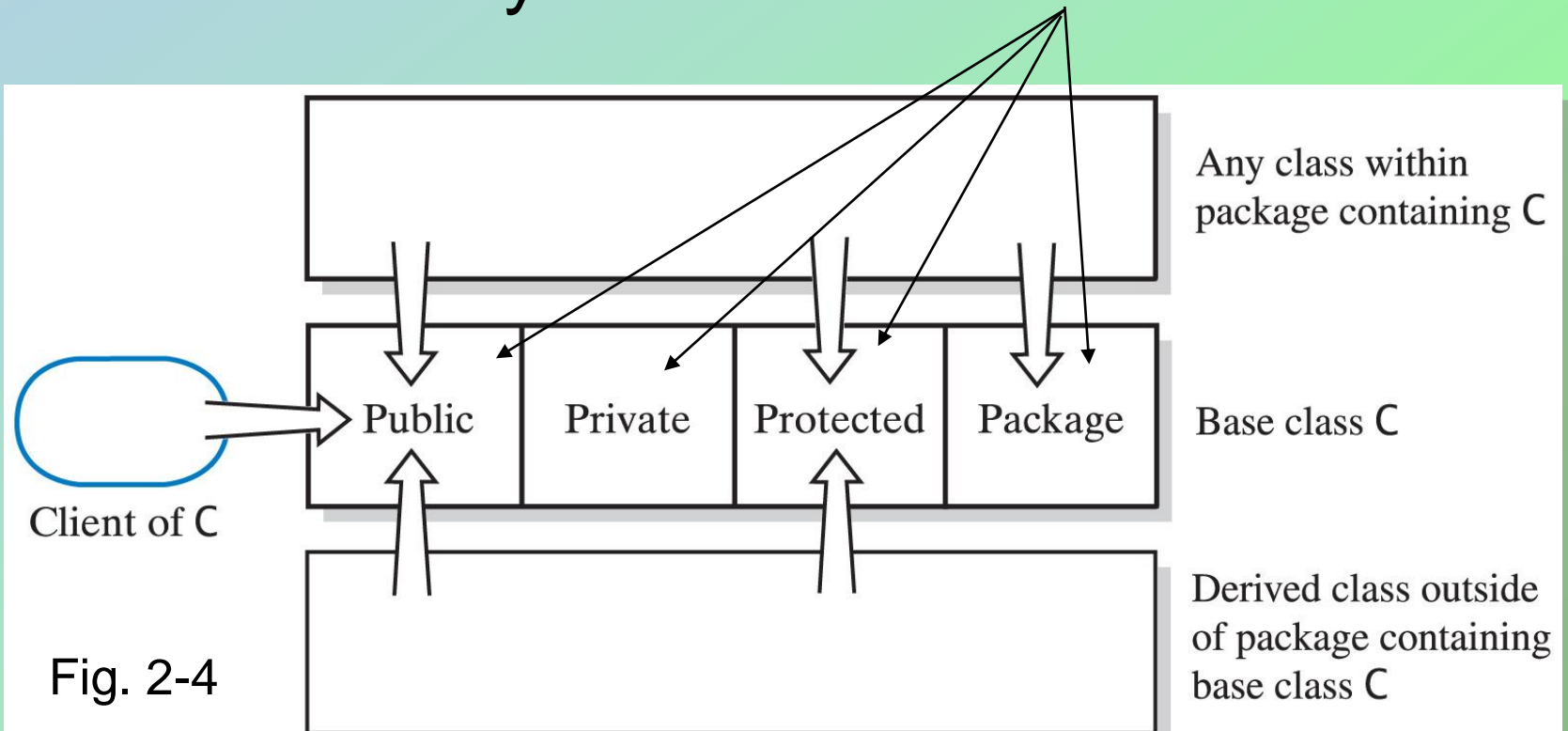
# Protected Access

- A method or data field modified by **protected** can be accessed by name only within
  - Its own class definition
  - Any class derived from that base class
  - Any class within the same package
- A Java package is a collection of classes related to a certain activity, such as graphics:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/package-summary.html>

# Protected Access

- Note accessibility of elements of a class C determined by the access modifiers



# Overriding Methods

- When a derived class defines a method with the same signature as in base class
  - Same name
  - Same return type
  - Same number, types of parameters
- Objects of the derived class that invoke the method will use the definition from the derived class
- It is possible to use **super** in the derived class to call an overridden method of the base class



# Overriding Methods

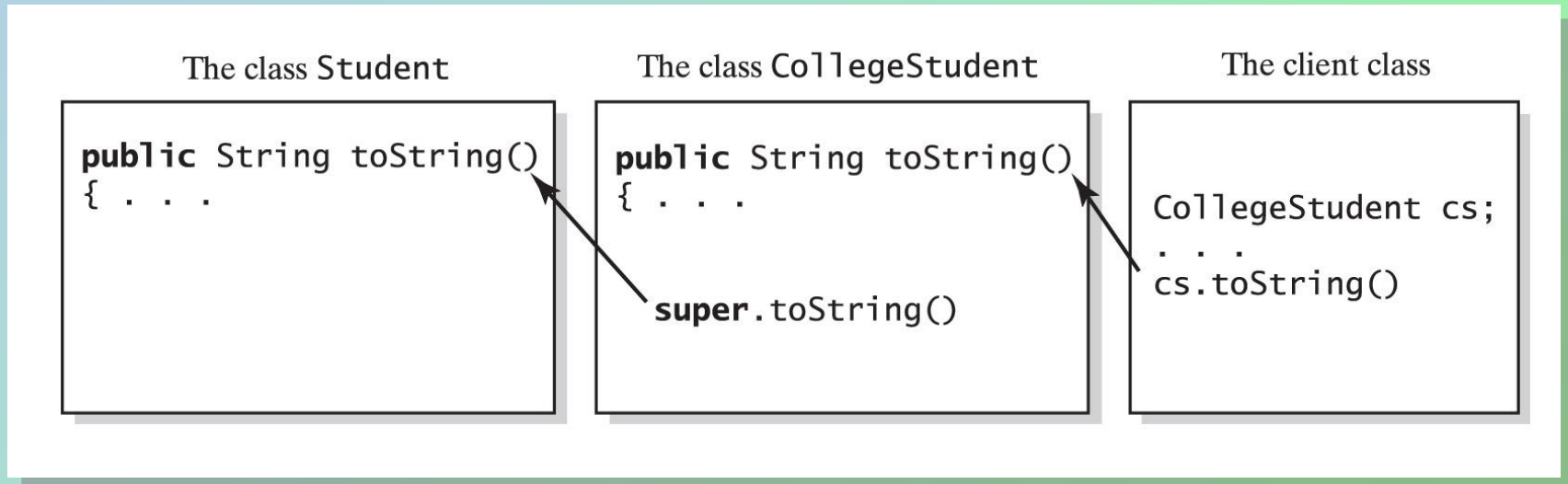


Fig. 2-5 The method `toString` in `CollegeStudent` overrides the method `toString` in `Student`

# Overloading a Method

- When the derived class method has
  - The same name
  - The same return type ... but ...
  - Different number or type of parameters
- Then the derived class has available
  - The derived class method ... and
  - The base class method with the same name
- Java distinguishes between the two methods due to the different parameters

# Multiple use of `super`

- Consider a class derived from a base ... that itself is derived from a base class
  - All three classes have a method with the same signature
- The overriding method in the lowest derived class cannot invoke the method in the base class's base class
  - The construct `super . super` is illegal

# Overloading a Method

- A programmer may wish to specify that a method definition cannot be overridden
  - So that the behavior of the constructor will not be changed
- This is accomplished by use of the modifier **final**

```
public final void whatever()
{
 . . .
}
```

Question 7 Question 5 asked you to create an instance of `NickName` to represent the nickname Bob. If that object is named `bob`, do the following statements produce the same output? Explain.

```
System.out.println(bob.getNickName());
System.out.println(bob);
```

**7.**

**No. Since `getNickName` returns a string, the first statement implicitly calls the method `toString` defined in the class `String`. Thus, `Bob` is displayed. Since the class `NickName` does not define its own version of `toString`, the second statement invokes `Object`'s `toString`. The output involves the memory address of the object referenced by `bob`.**

Question 8 Are the two definitions of the constructors for the class Student ([Segment C.2](#)) an example of overloading or overriding? Why?

Question 9 If you add the method  
`public void setStudent(Name studentName, String studentId)`  
to the class `CollegeStudent` and let it give some default values to the fields `year` and `degree`, are you overloading or overriding `setStudent`? Why?

**8.**

**Overloading.** The constructors have the same name but different signatures.

**9.**

**Overriding.** The revised version of `setStudent` in `CollegeStudent` has the same signature and return type as the version in the superclass `Student`.



# Multiple Inheritance

- Some languages allow programmer to derive class **C** from classes **A** and **B**
- Java does not allow this
  - A derived class can have only one base class
- Multiple inheritance can be approximated
  - A derived class can have multiple interfaces
  - Described in Chapter 3

# Object Types of a Derived Class

- Given :
  - Class **CollegeStudent**,
  - Derived from class **Student**
- Then a **CollegeStudent** object is also a **Student** object
- In general ...  
An object of a derived class is also an object of the base class

Question 10 If `HighSchoolStudent` is a subclass of `Student`, can you assign an object of `HighSchoolStudent` to a variable of type `Student`? Why or why not?

Question 11 Can you assign an object of `Student` to a variable of type `HighSchoolStudent`? Why or why not?

**10.**

**Yes. You can assign an object of a class to a variable of any ancestor type. An object of type `HighSchoolStudent` can do anything that an object of type `Student` can do.**

**11.**

**No. The `Student` object does not have all the behaviors expected of a `HighSchoolStudent` object.**

# The Class `Object`

- Every class is a descendant of the class `Object`
- `Object` is the class that is the beginning of every chain of derived classes
  - It is the ancestor of every other class
  - Even those defined by the programmer
- <http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Object.html>

# Speaking of class Object...

Ancestor of all classes

- defines methods `clone()`, `equals()`, `toString()`, among others
- all classes automatically derive these methods from `Object`
- to be useful, we have to override them so they work with the details of the class in which they are defined

# Overriding the equals method

the parameter is class Object, we use a cast to be able to refer to it as a Name or Student or BankAccount or whater class we are defining it for.

we can refer to the private data members of `that` since it is an object of the same class. Here is equals for Name:

```
public boolean equals (Object other) {
 if (other instanceof Name) {
 Name that = (Name) other;
 return this.first.equals(that.first) &&
 this.last.equals(that.last)
 }
 else return false;
}
```

# Overriding the clone() method

- Makes an exact duplicate object with same data. `joe2 = joe.clone();`
  - We now have two identical objects

```
public Name clone() {
 return new Name(first, last);
}
```




Question 12 If sue and susan are two instances of the class Name, what if statement can decide whether they represent the same name?

**12.**

**if (sue.equals(susan))**

# Abstract Classes and Methods

- Some base classes are not intended to have objects of that type
  - The objects will be of the derived classes
- Declare that base class to be abstract  
`public abstract class Whatever`  
`{ . . . }`
- The designer often specifies methods of the abstract class without a body   
`public abstract void doSomething();`
  - This requires all derived classes to implement this method

# Polymorphism

- When one method name in an instruction can cause different actions
  - Happens according to the kinds of objects that invoke the methods
- Example

```
UndergradStudent ug = new UndergradStudent(. . .);

Student s = ug;
 // s and ug are aliases
s.displayAt(2);
ug.displayAt(4);
```

The object still remembers it is of type UndergradStudent

# Polymorphism

```
public class Student
{
 . . .
 public void display()
 {
 . . .
 } // end display

 public void displayAt(int numberOfLines)
 {
 . . .
 display();
 } // end displayAt
} // end Student
```

```
public class CollegeStudent
 extends Student
{
 . . .
 public void display()
 {
 . . .
 } // end display
 . . .
} // end CollegeStudent
```

```
public class UndergradStudent
 extends CollegeStudent
{
 . . .
 public void display()
 {
 . . .
 } // end display
 . . .
} // end UndergradStudent
```

```
public class Client
{
 public static void main(String[] args)
 {
 . . .
 UndergradStudent ug = new UndergradStudent(. . .);
 ug.displayAt(2);
 . . .
 } // end main
 . . .
} // end Client
```

Figure 2-6 The method `displayAt` calls the correct version of `display`.

# Polymorphism

- Which `displayAt` is called ...
  - Depends on the invoking object's place in the inheritance chain and is not determined by the type of the variable naming the object

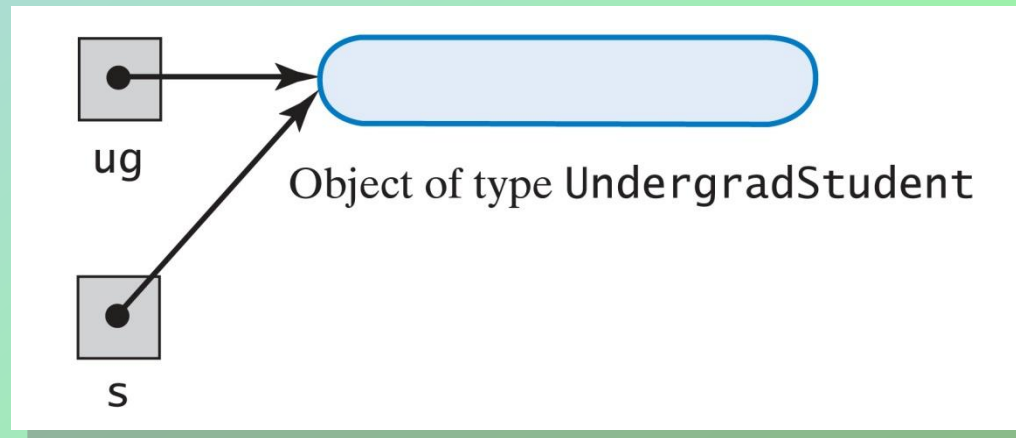


Fig. 2-7 The variable `s` is another name for an undergraduate object.

# Dynamic Binding

- The process that enables different objects to ...
  - Use different method actions
  - For the same method name
- Objects know how they are supposed to act
  - When an overridden method is used ...
  - The action is for the method defined in the class whose constructor created the object

# Dynamic Binding

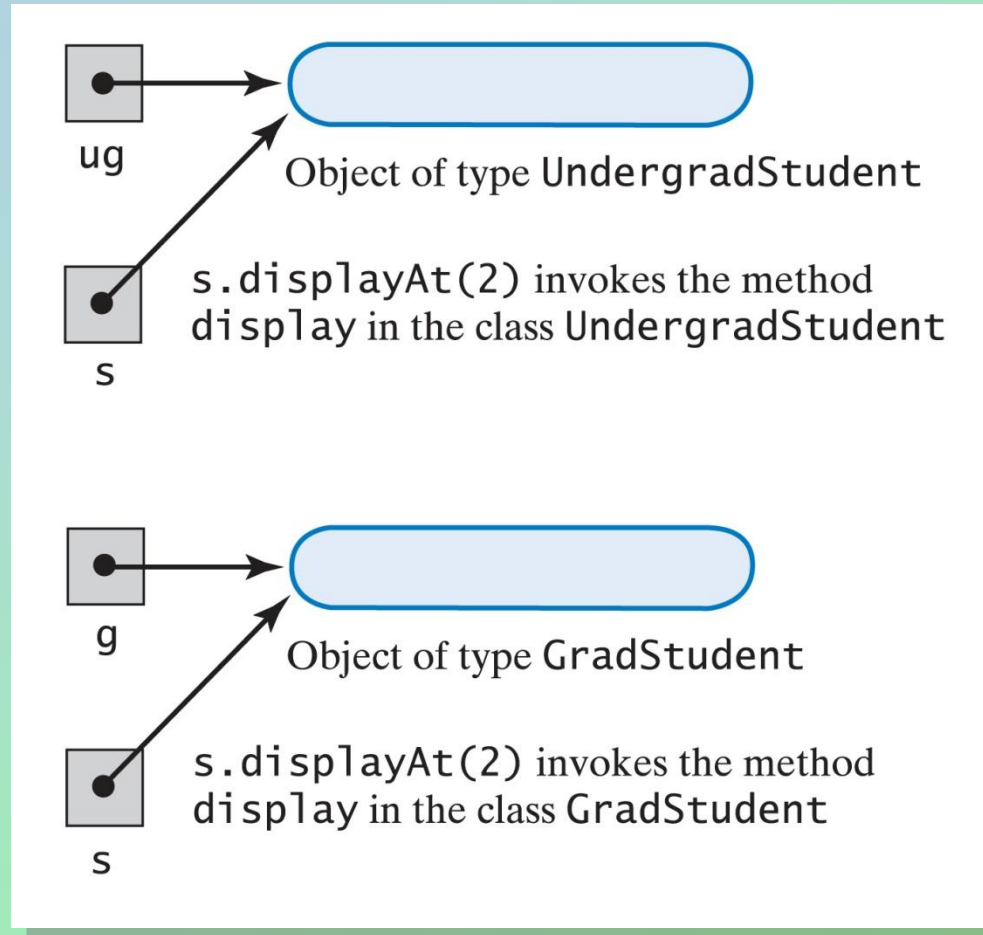


Fig. 2-8 An object, not its name, determines its behavior.



Question 14 Is a method display with no parameters that is defined explicitly in each of the classes Student, CollegeStudent, and UndergradStudent an example of overloading or overriding? Why?

Question 15 Is overloading a method name an example of polymorphism?

14.

Overriding. The methods have the same signatures and return types.

15.

At one time, overloading was an example of polymorphism. Today, polymorphism describes a situation in which an object determines at execution time which action of a method it will use for a method name that is overridden either directly or indirectly.