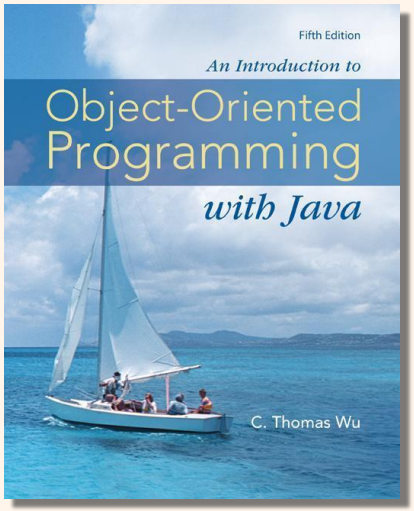


Chapter 13

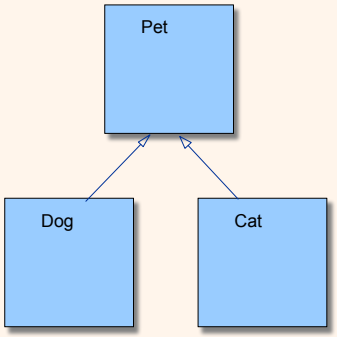
Inheritance and Polymorphism

Chapter 13 Objectives

- After you have read and studied this chapter, you should be able to
  - Write programs that are easily extensible and modifiable by applying polymorphism in program design.
  - Define reusable classes based on inheritance and abstract classes and abstract methods.
  - Differentiate the abstract classes and Java interfaces.
  - Define methods, using the **protected** modifier.
  - Parse strings, using a **String Tokenizer** object.

Simple Example

- We can effectively model similar, but different types of objects using inheritance
- Suppose we want to model dogs and cats. They are different types of pets. We can define the Pet class and the Dog and Cat classes as the subclasses of the Pet class.



The Pet Class

```
class Pet {
    private String name;
    public String getName() {
        return name;
    }

    public void setName(String petName) {
        name = petName;
    }

    public String speak( ) {
        return "I'm your cuddly little pet.";
    }
}
```

Subclasses of The Pet Class

```
class Cat extends Pet {
    public String speak( ) {
        return "Don't give me orders.\n" +
               "I speak only when I want to.";
    }
}
```

The Cat subclass overrides the inherited method speak.

```
class Dog extends Pet {
    public String fetch( ) {
        return "Yes, master. Fetch I will.";
    }
}
```

The Dog subclass adds a new method fetch.

Sample Usage of the Subclasses

```
Dog myDog = new Dog();

System.out.println(myDog.speak());
System.out.println(myDog.fetch());
```

```
I'm your cuddly little pet.
Yes, master. Fetch I will.
```

```
Cat myCat = new Cat();

System.out.println(myCat.speak());
System.out.println(myCat.fetch());
```

```
Don't give me orders.
I speak only when I want to.
```

ERROR

Defining Classes with Inheritance

- Case Study:
  - Suppose we want implement a class roster that contains both undergraduate and graduate students.
  - Each student's record will contain his or her name, three test scores, and the final course grade.
  - The formula for determining the course grade is different for graduate students than for undergraduate students.

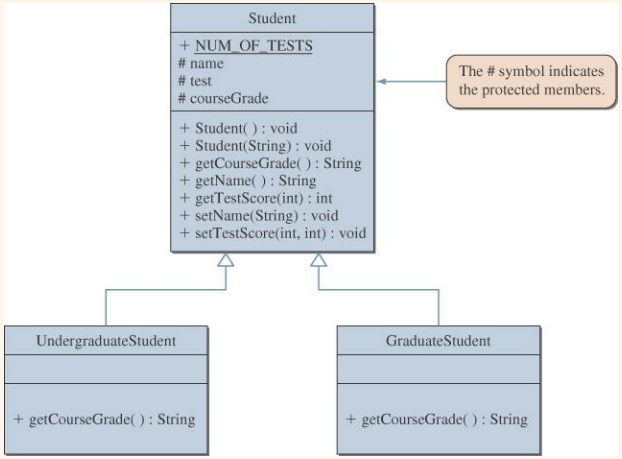
Modeling Two Types of Students

- There are two ways to design the classes to model undergraduate and graduate students.
  - We can define two unrelated classes, one for undergraduates and one for graduates.
  - We can model the two kinds of students by using classes that are related in an inheritance hierarchy.
- Two classes are *unrelated* if they are not connected in an inheritance relationship.

Classes for the Class Roster

- For the Class Roster sample, we design three classes:
  - Student
  - UndergraduateStudent
  - GraduateStudent
- The **Student** class will incorporate behavior and data common to both **UndergraduateStudent** and **GraduateStudent** objects.
- The **UndergraduateStudent** class and the **GraduateStudent** class will each contain behaviors and data specific to their respective objects.

Inheritance Hierarchy



The Protected Modifier

- The modifier **protected** makes a data member or method visible and accessible to the instances of the class and the descendant classes.
- **Public** data members and methods are accessible to everyone.
- **Private** data members and methods are accessible only to instances of the class.

Polymorphism

- Polymorphism allows a single variable to refer to objects from different subclasses in the same inheritance hierarchy
- For example, if Cat and Dog are subclasses of Pet, then the following statements are valid:

```
Pet myPet;

myPet = new Dog();
. . .
myPet = new Cat();
```

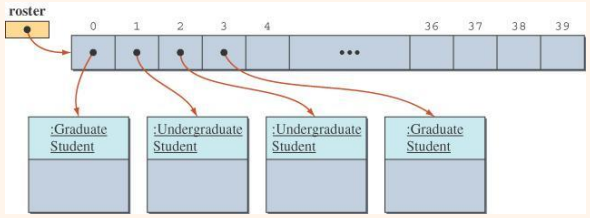
Creating the roster Array

- We can maintain our class roster using an array, combining objects from the **Student**, **UndergraduateStudent**, and **GraduateStudent** classes.

```
Student roster = new Student[40];
. . .
roster[0] = new GraduateStudent();
roster[1] = new UndergraduateStudent();
roster[2] = new UndergraduateStudent();
. . .
```


State of the roster Array

- The roster array with elements referring to instances of **GraduateStudent** or **UndergraduateStudent** classes.



Sample Polymorphic Message

- To compute the course grade using the roster array, we execute

```
for (int i = 0; i < numberOfStudents; i++) {
    roster[i].computeCourseGrade();
}
```

- If roster[i] refers to a GraduateStudent, then the computeCourseGrade method of the GraduateStudent class is executed.
- If roster[i] refers to an UndergraduateStudent, then the computeCourseGrade method of the UndergraduateStudent class is executed.

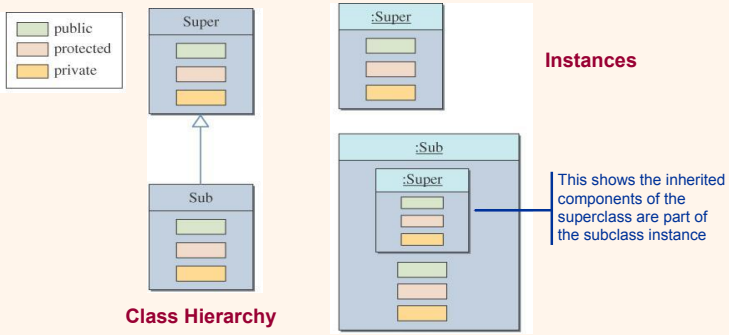
The instanceof Operator

- The instanceof operator can help us learn the class of an object.
- The following code counts the number of undergraduate students.

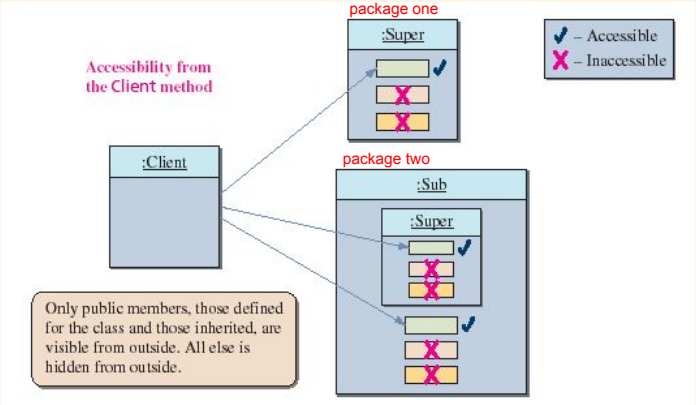
```
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if ( roster[i] instanceof UndergraduateStudent ) {
        undergradCount++;
    }
}
```

Inheritance and Member Accessibility

- We use the following visual representation of inheritance to illustrate data member accessibility.

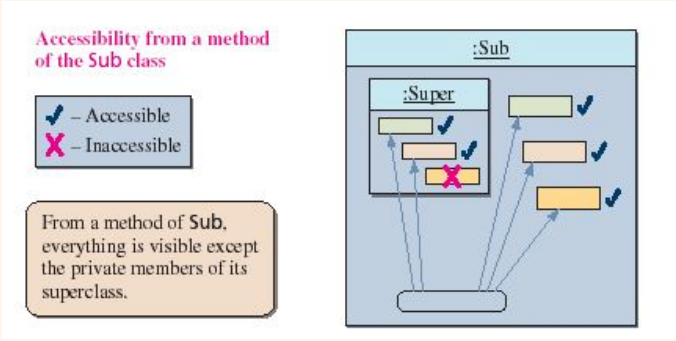


The Effect of Three Visibility Modifiers



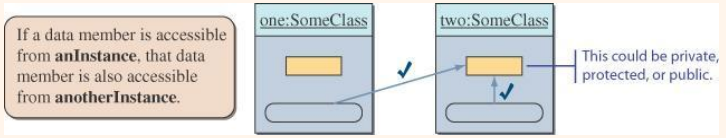
Accessibility of Super from Sub

- Everything except the private members of the Super class is visible from a method of the Sub class.



Accessibility from Another Instance

- Data members accessible from an instance are also accessible from other instances of the same class.



Inheritance and Constructors

- Unlike members of a superclass, constructors of a superclass are *not* inherited by its subclasses.
- You must define a constructor for a class or use the default constructor added by the compiler.
- The statement

```
super();
```

  calls the superclass's constructor.
- If the class declaration does not explicitly designate the superclass with the **extends** clause, then the class's superclass is the **Object** class.

Abstract Superclasses and Abstract Methods

- When we define a superclass, we often do not need to create any instances of the superclass.
- Depending on whether we need to create instances of the superclass, we must define the class differently.
- We will study examples based on the **Student** superclass defined earlier.

Definition: Abstract Class

- An *abstract class* is a class
  - defined with the modifier **abstract** OR
  - that contains an abstract method OR
  - that does not provide an implementation of an inherited abstract method
- An abstract method is a method with the keyword **abstract**, and it ends with a semicolon instead of a method body.
  - Private methods and static methods may not be declared **abstract**.
- No instances can be created from an abstract class.

Case 1

- Student Must Be Undergraduate or Graduate
  - If a student must be either an undergraduate or a graduate student, we only need instances of UndergraduateStudent or GraduateStudent.
  - Therefore, we must define the Student class so that no instances may be created of it.

Case 2

- Student Does Not Have to Be Undergraduate or Graduate.
- In this case, we may design the Student class in one of two ways.
  - We can make the Student class instantiable.
  - We can leave the Student class abstract and add a third subclass, OtherStudent, to handle a student who does not fall into the UndergraduateStudent or GraduateStudent categories.

Which Approach to Use

- The best approach depends on the particular situation.
- When considering design options, we can ask ourselves which approach allows easier modification and extension.

Inheritance versus Interface

- The Java interface is used to share common behavior (only method headers) among the instances of different classes.
- Inheritance is used to share common code (including both data members and methods) among the instances of related classes.
- In your program designs, remember to use the Java interface to share common behavior. Use inheritance to share common code.
- If an entity A is a specialized form of another entity B, then model them by using inheritance. Declare A as a subclass of B.

Problem Statement

Write an application that reads in a text file organized in the manner shown below and displays the final course grades.The course grades are computed differently for the undergraduate and graduate students based on the formulas listed on page 717. The input text file format is as follows:

- *A single line is used for information on one student.*
- *Each line uses the format*

  <Type> <Name> <Test 1> <Test 2> <Test 3>

  where <Type> designates either a ***graduate*** *or an* ***undergraduate***

  student,<Name> designates the student's ***first and last name****, and*

  <Test i> designates the ith test ***score****.*
- *End of input is designated by the word* ***END****. The case of the letters is insignificant.*

Overall Plan

- Tasks
  1. Read an input text file.
  2. Compute the course grades.
  3. Print out the result.
- Input File Format

<Type>	<Name>	<Test 1>	<Test 2>	<Test 3>
U	John Doe	87	78	90
G	Jill Jones	90	95	87
G	Jack Smith	67	77	68
U	Mary Hines	80	85	80
U	Mick Taylor	76	69	79
END				

Development Steps

- We will develop this program in five steps:

1. Start with the program skeleton.Define the skeleton ComputeGrades classes.
2. Implement the printResult method.Define any other methods necessary to implement printResult.
3. Implement the computeGrade method.Define any other methods necessary to implement computeGrade.
4. Implement the readData method.Define any other methods necessary to implement readData.
5. Finalize and look for improvements.

Step 1 Design

- We start with a program skeleton.
- We will define two constructors so the programmer can create a roster of default size or the size of her choice.

Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter13/Step1

Source Files: ComputeGrades.java

Step 1 Test

- We include a temporary output statement inside the (currently stub) method we define.
- We run the test main class and verify that the methods are called correctly.

Step 2 Design

- We design and implement the printResult method
- We use the helper class OutputBox for displaying the result.

```
for each element i in the roster array {
    output the name of roster[i];
    output the test scores of roster[i];
    output the course grade of roster[i];
    skip to the next line;
}
```


Step 2 Code

Directory: Chapter13/Step2

Source Files: ComputeGrades.java

Step 2 Test

- We verify the temporary readData method is working correctly. This confirms that we are using the correct student classes and using their methods correctly.
- We verify the printResult method does indeed display the data in our desired format.

Step 3 Design

- We design and implement the computeGrade method.
- The code for actually determining the course grade is embedded in individual student classes
  - So the code to add to the ComputeGrades class is very simplistic.
  - This is a direct benefit of using polymorphism effectively.

Step 3 Code

Directory: Chapter13/Step3

Source Files: ComputeGrades.java

Step 3 Test

- We will repeat the same test routines from Step 2.
- Instead of seeing four asterisks, we should be seeing the correct grades.
- We test both the passing and not passing test scores.

Step 4 Design

- We design and implement the core functionality of the program—the readData method
- We can express its logic as

```
get the filename from the user;

if (the filename is provided)
    read in data and build the roster array;
else
    output an error message;
```

The buildRoster Method

- The logic of the workhorse private method buildRoster is as follows:

```
set bufReader for input;
while ( !done ) {
    line = get next line;
    if (line is END) {
        done = true;
    } else {
        student = createStudent( line );
        if (student != null) {
            roster[studentCount] = student; //add to roster
            studentCount++;
        }
    }
}
```


The createStudent Method

- We use the StringTokenizer class to break down items in a single line of input

```
StringTokenizer parser = new StringTokenizer( line );
String type;
try {
    type = parser.nextToken();
    if (type.equals(UNDER_GRAD) || type.equals(GRAD)) {
        student = newStudentWithData(type, parser);
    } else { //invalid type is encountered
        student = null;
    }
} catch (NoSuchElementException e) { //no token
    student = null;
}
return student;
```


Step 4 Code

Directory: Chapter13/Step4

Source Files: ComputeGrades.java

Step 4 Test

- We run through a more complete testing routine in this step.We need to run the program for various types of input files. Some of the possible file contents are as follows:

Step 4 Test Data	
Test File File with 5 to 20 entries of student information with all lines in correct format	**Purpose** Test the normal case.
File with 5 to 20 entries of student information with some lines in incorrect format	Test that readData and supporting methods handle the error case properly.
File with no entries	Test that buildRoster method handles the error case properly.
File with more than 25 entries	Test that readData and supporting methods handle the case where the number of entries is larger than the default size for the roster array.

Step 5: Finalize and Improve

- We finalize the program by correcting any remaining errors, inconsistency, or unfinished methods.
- We want to review the methods and improve them as necessarily.
- One problem (which would have been identified in step 4 testing) we need to correct is the missing method for expanding the roster array when the input file includes more student entries than the set default size of 25.
  - We leave this method as Exercise 3.
  - We also leave some of the possible improvements as exercises.