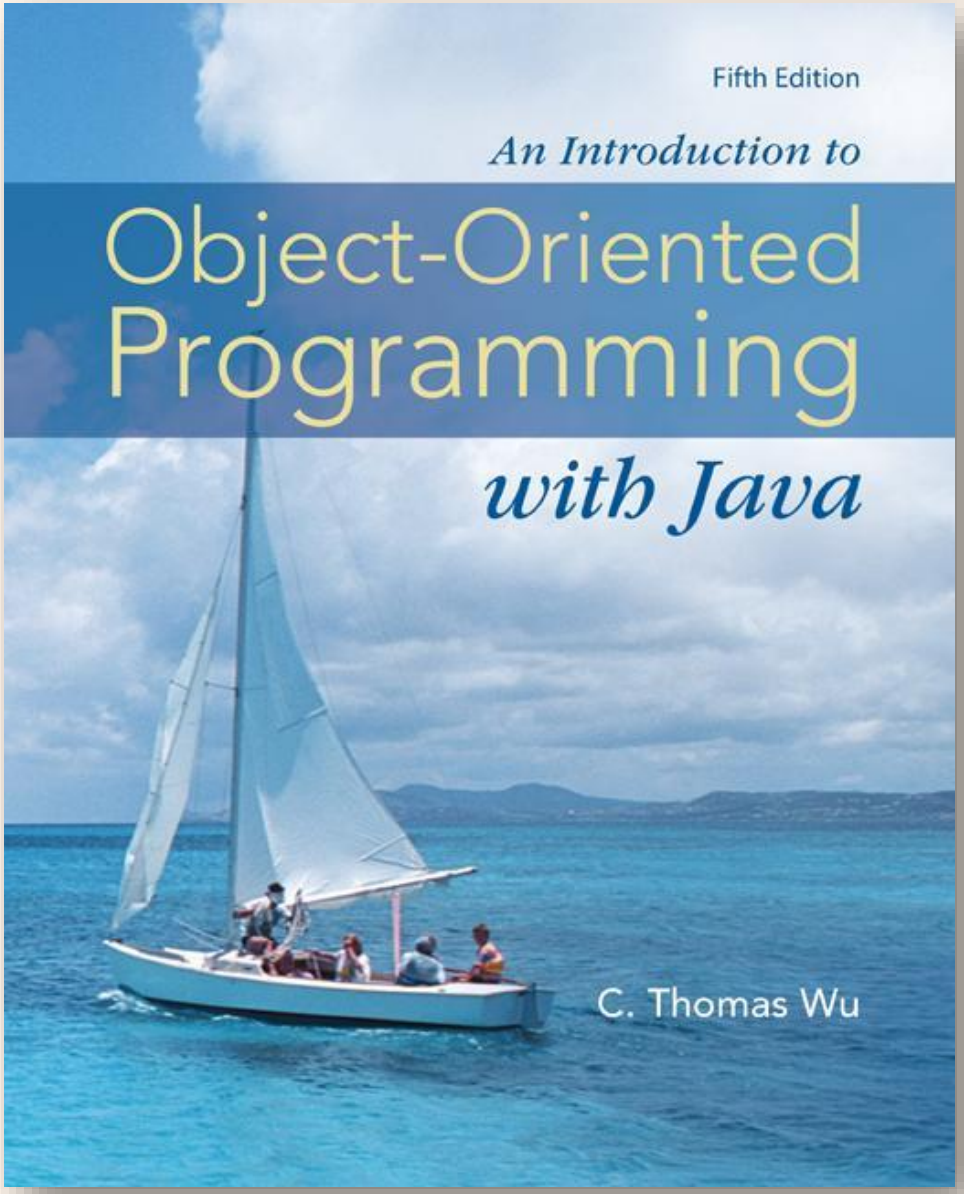


Chapter 13

Inheritance and Polymorphism

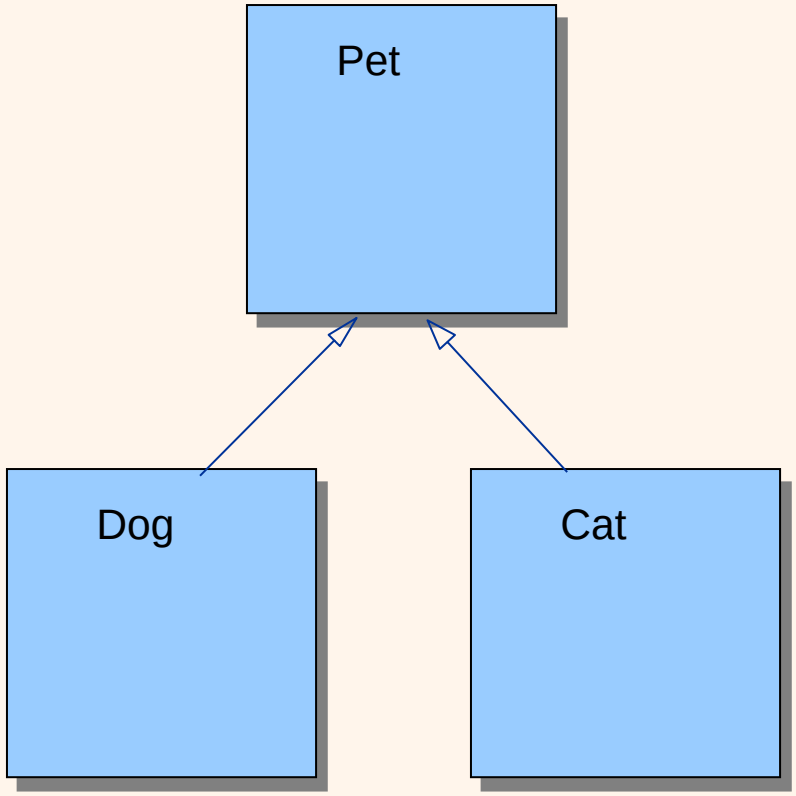
Animated Version

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.

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.

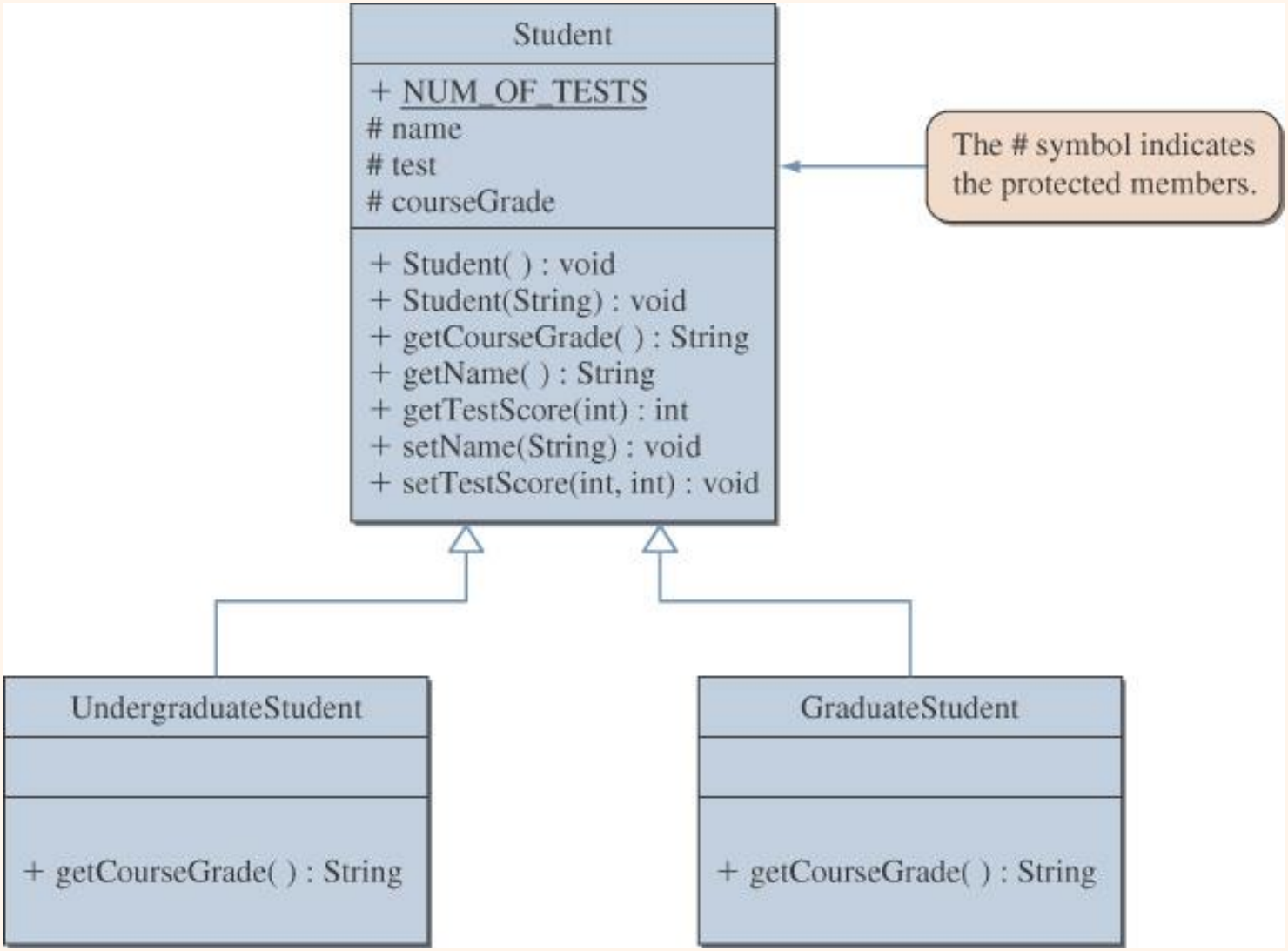
Modeling Two Types of Students

- 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.

Main Point

When one class (the *subclass)* inherits from another class (the *superclass*), all the protected and public data and methods in the superclass are automatically accessible to the subclass. Java supports this notion of inheritance. In Java syntax, a class is declared to be a subclass of another by using the *extends* keyword. Likewise, individual intelligence "inherits from" cosmic intelligence, though each "implementation" is unique.

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();
```

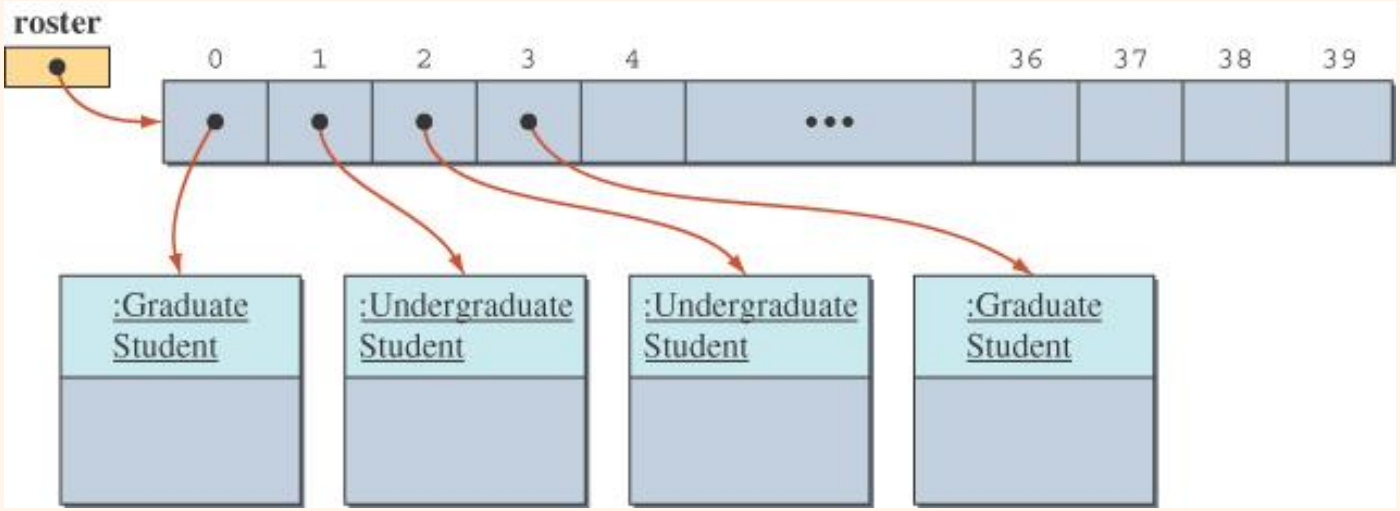
Creating the roster Array

- 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.

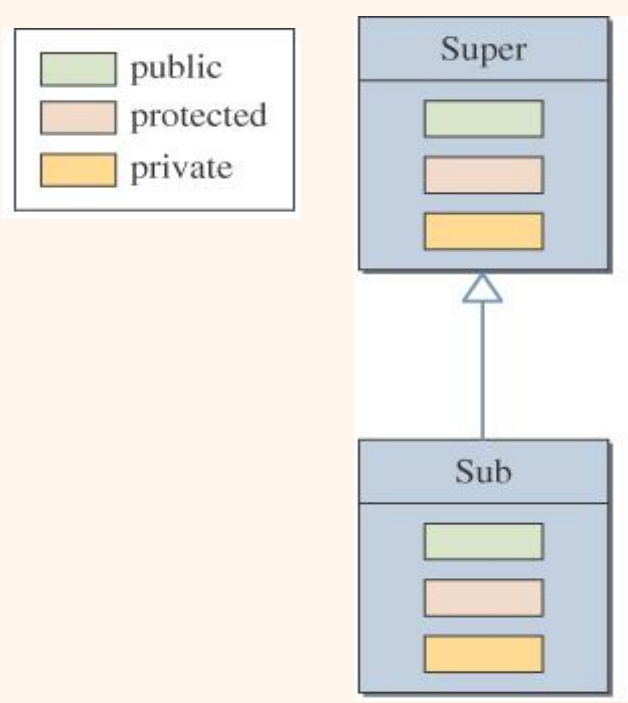
The instanceof Operator

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

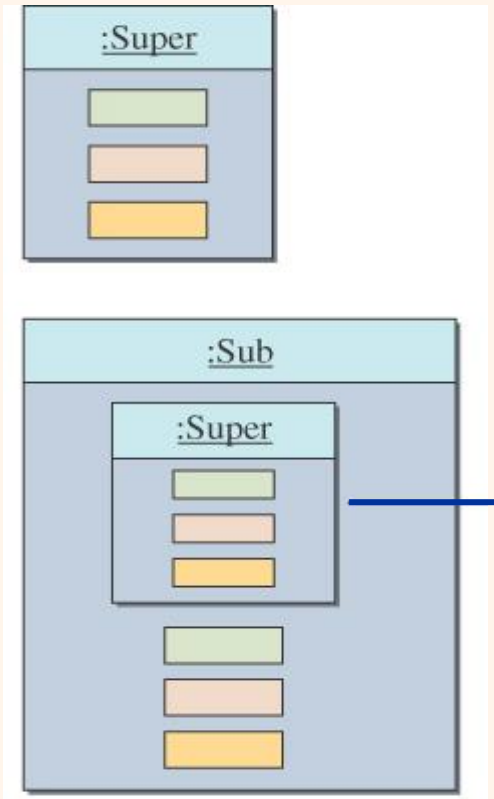
```
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if ( roster[i] instanceof UndergraduateStudent ) {
        undergradCount++;
    }
}
```

Inheritance and Member Accessibility

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



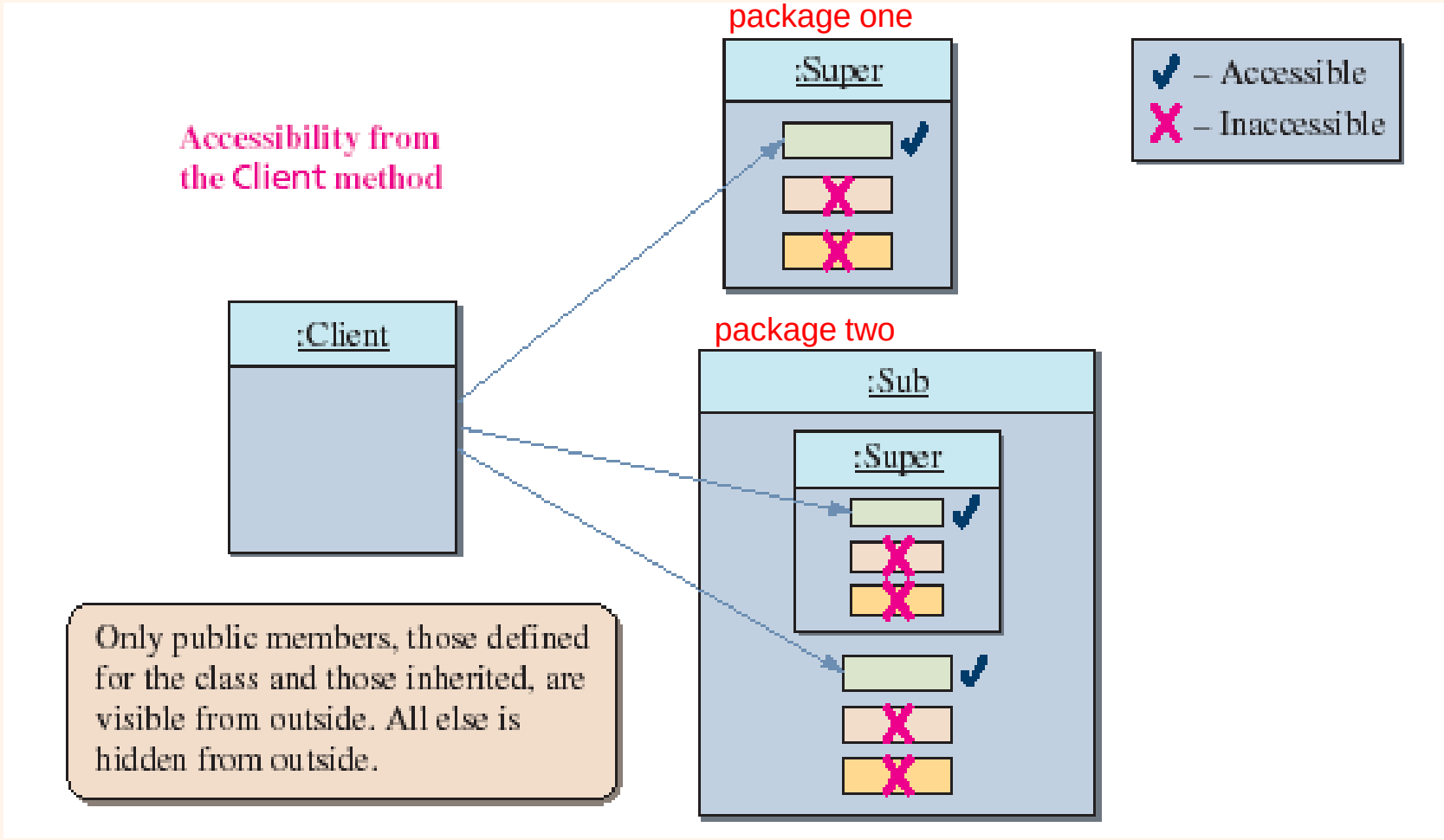
Class Hierarchy



Instances

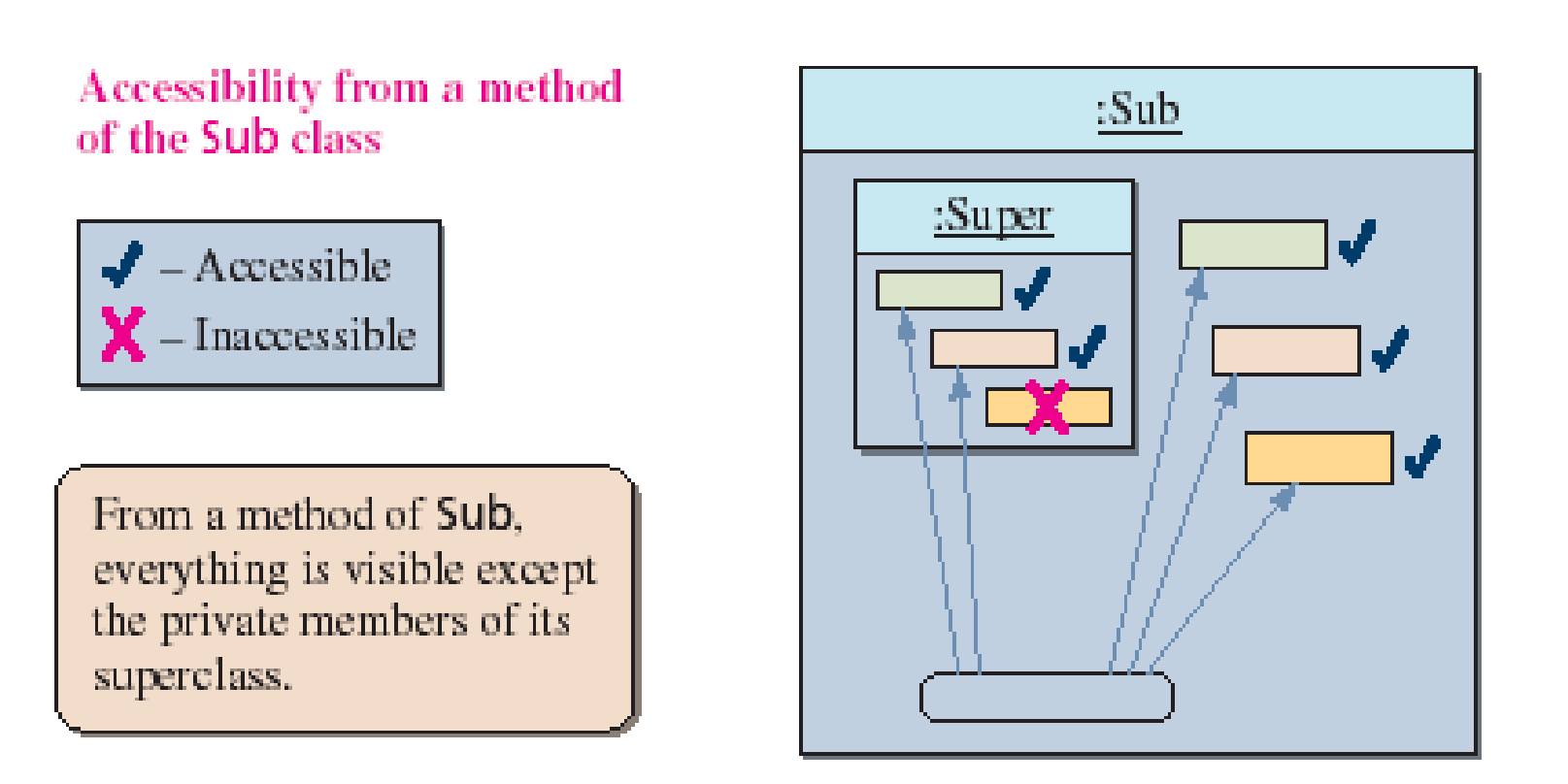
This shows the inherited components of the superclass are part of the subclass instance

The Effect of Three Visibility Modifiers



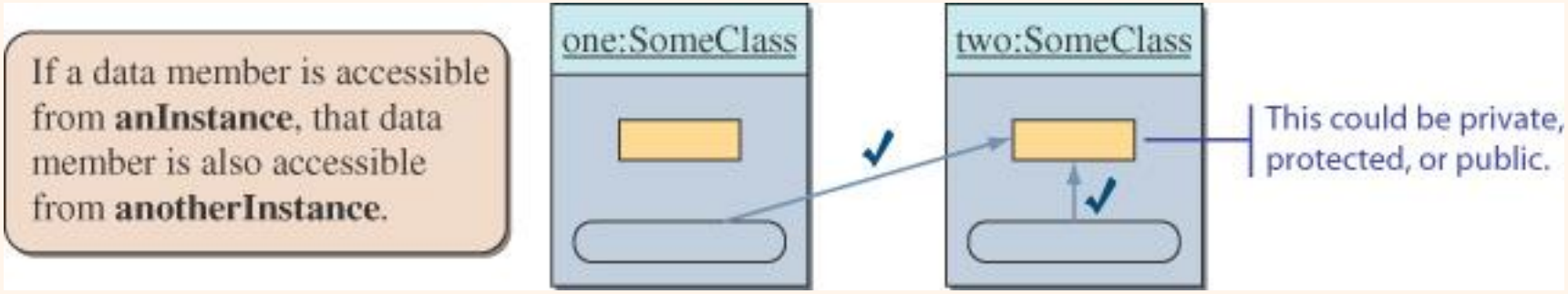
Accessibility of Super from Sub

- 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.

Java Interfaces

- A Java interface is like an abstract class except:
  - No instance variables (other than variables declared final) or implemented methods can occur
  - An interface is declared using the interface keyword, not the class keyword
  - A class that implements an interface uses the implements keyword rather than the extends keyword
  - Java interface is more abstracter than the Java abstract class.
  - **See example lesson3.interface_demo**

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.

Main Point

Interfaces are used in Java to specify publicly available services in the form of method declarations. A class that implements such an interface must make each of the methods operational. Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. The concept of an interface is analogous to the creation itself – the creation may be viewed as an “interface” to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.