Chapter 4

Inheritance

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

Objectives

In this chapter you will learn:

- How inheritance promotes software reusability.
- The notions of superclasses and subclasses.
- To use keyword extends to create a class that inherits attributes and behaviors from another class.
- To use access modifier protected to give subclass methods access to superclass members.
- To access superclass members with super.
- How constructors are used in inheritance hierarchies.
- The methods of class Object, the direct or indirect superclass of all classes in Java.

OUTLINE

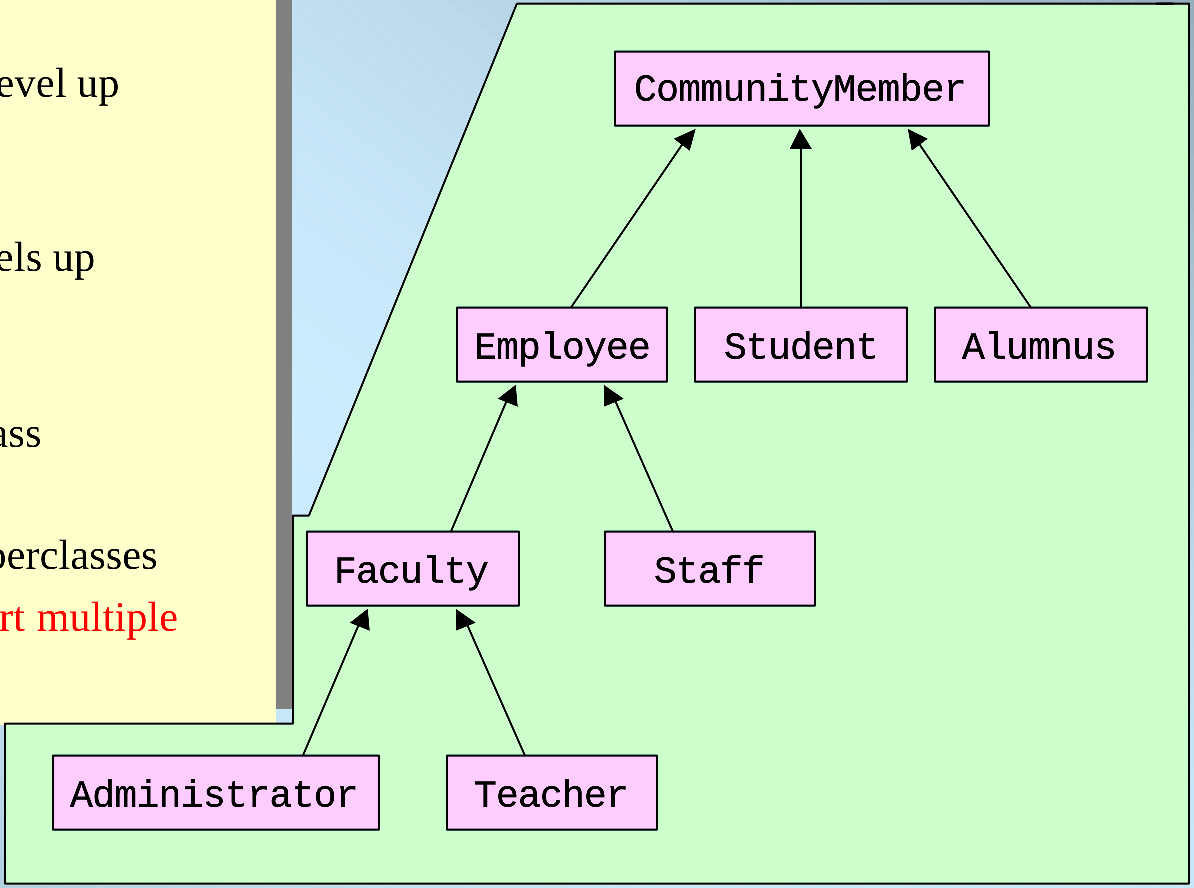
1. Introduction
2. Defining Classes with Inheritance
3. Inheritance and Member Accessibility
4. Inheritance Hierarchy
5. Declaring Subclasses
6. Inheritance and Constructors
7. Examples

1. Introduction

- **Inheritance:** **is the sharing of attributes and methods among classes. We take a class (superclass), and then define other classes based on the first one (subclass). The subclass inherit all the attributes and methods of the superclass, but also have attributes and methods of their own.**
  - **Software reusability**
  - **Create new class from existing class**
    - Absorb existing class’s data and behaviors
    - Enhance with new capabilities
  - **Subclass extends superclass**
    - Subclass
      - More specialized group of objects
      - Behaviors inherited from superclass
        - Can customize
      - Additional behaviors

Introduction

- **Class hierarchy**
  - **Direct superclass**
    - Inherited explicitly (one level up hierarchy)
  - **Indirect superclass**
    - Inherited two or more levels up hierarchy
  - **Single inheritance**
    - Inherits from one superclass
  - **Multiple inheritance**
    - Inherits from multiple superclasses
      - Java does not support multiple inheritance



The important relationship between a subclass and its superclass is the *IS-A* relationship. The IS-A relationship must exist if inheritance is used properly.

2. Defining Classes with Inheritance

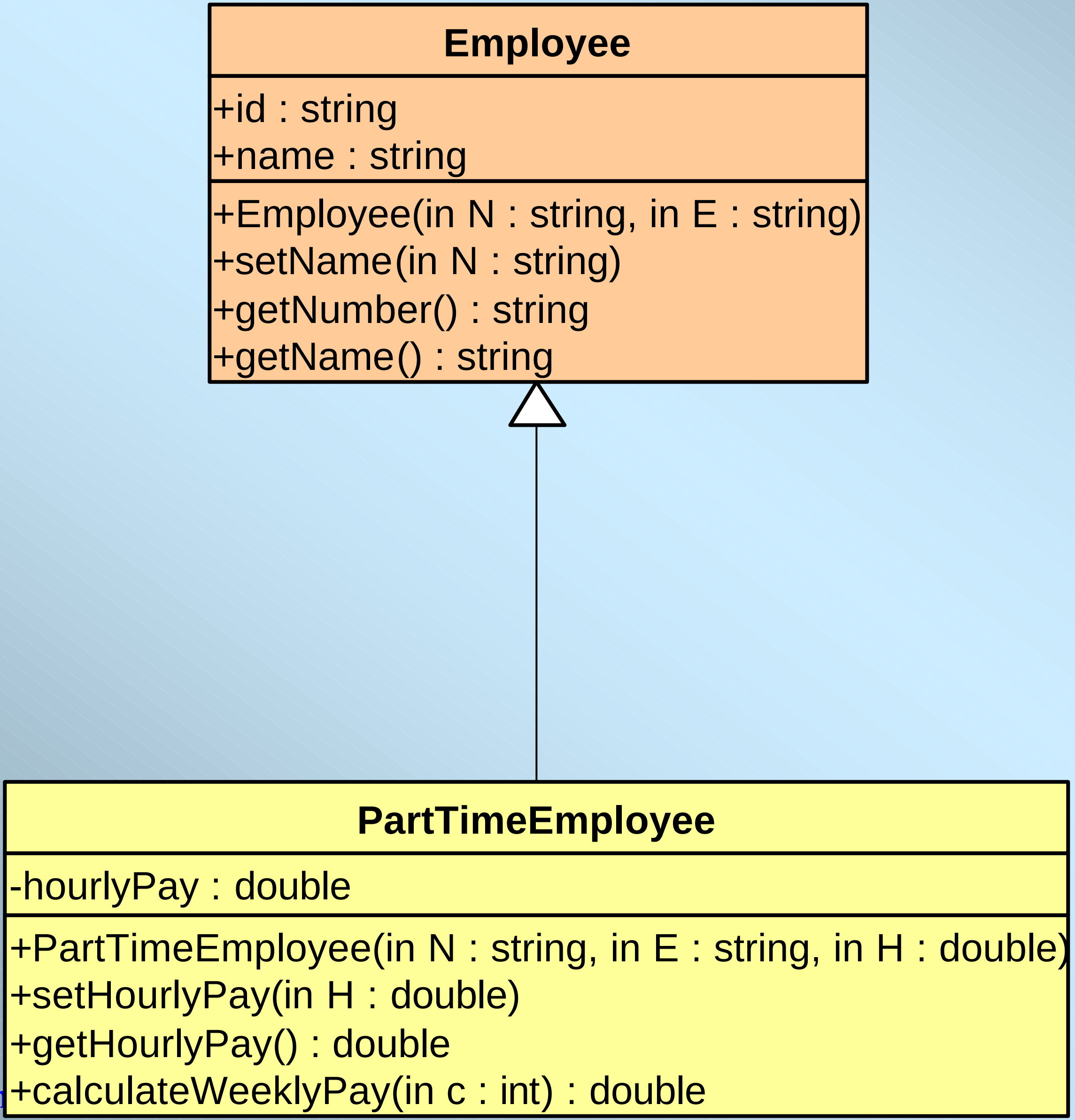
- **Case Study 1:**

- Suppose we want implement a class Employee which has two attributes, id and name, and some basic get- and set- methods for the attributes.

  – We want now define a PartTimeEmployee class; this class will inherit these attributes and methods, but can also have attributes (hourlyPay) and methods of its own (calculateWeeklyPay).

Defining Classes with Inheritance

An inheritance relationship using UML



Employee

+id : string
+name : string

+Employee(in N : string, in E : string)
+setName(in N : string)
+getNumber() : string
+getName() : string

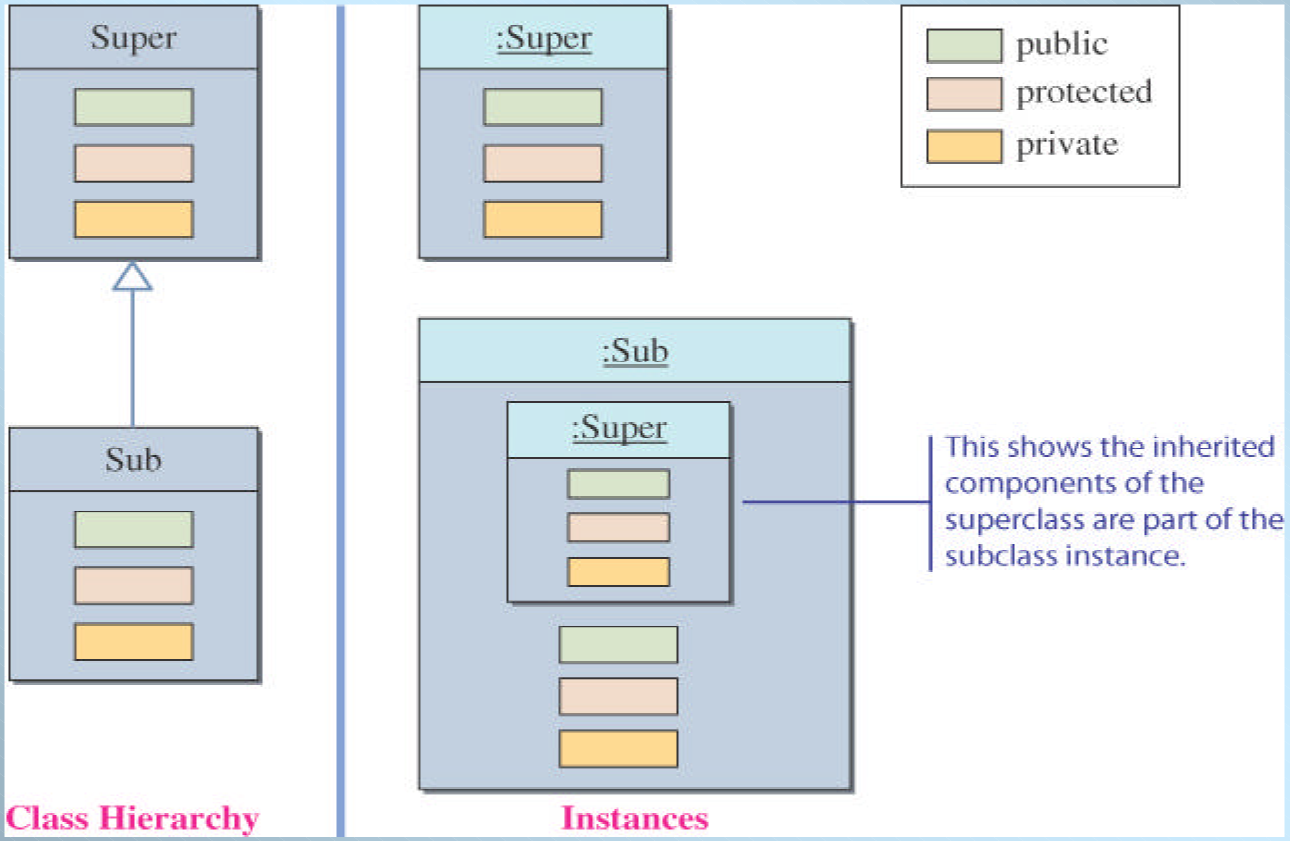
PartTimeEmployee

-hourlyPay : double

+PartTimeEmployee(in N : string, in E : string, in H : double)
+setHourlyPay(in H : double)
+getHourlyPay() : double
+calculateWeeklyPay(in c : int) : double

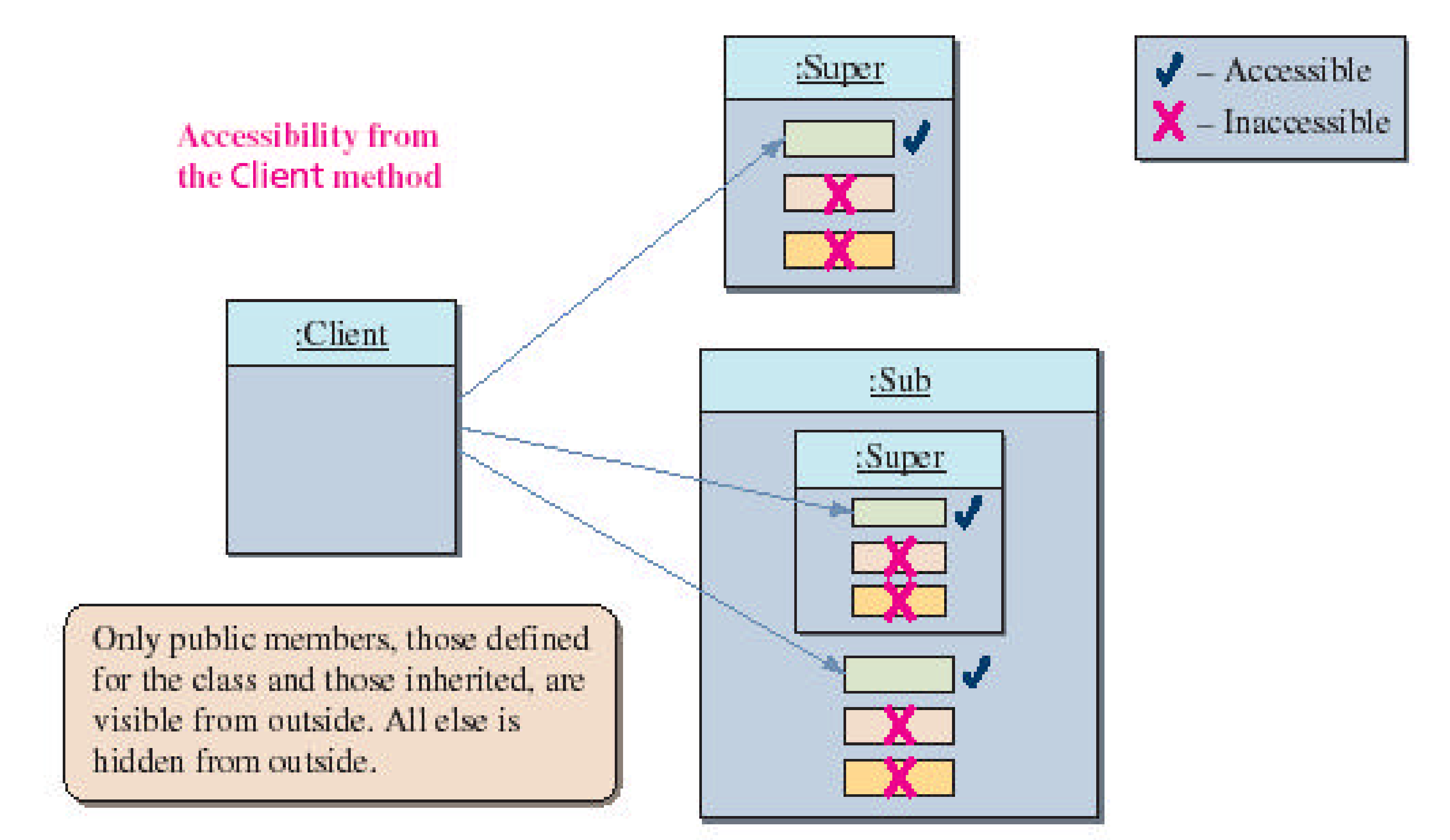
3. Inheritance and Member Accessibility

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



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

The Effect of Three Visibility Modifiers



Accessibility from the Client method

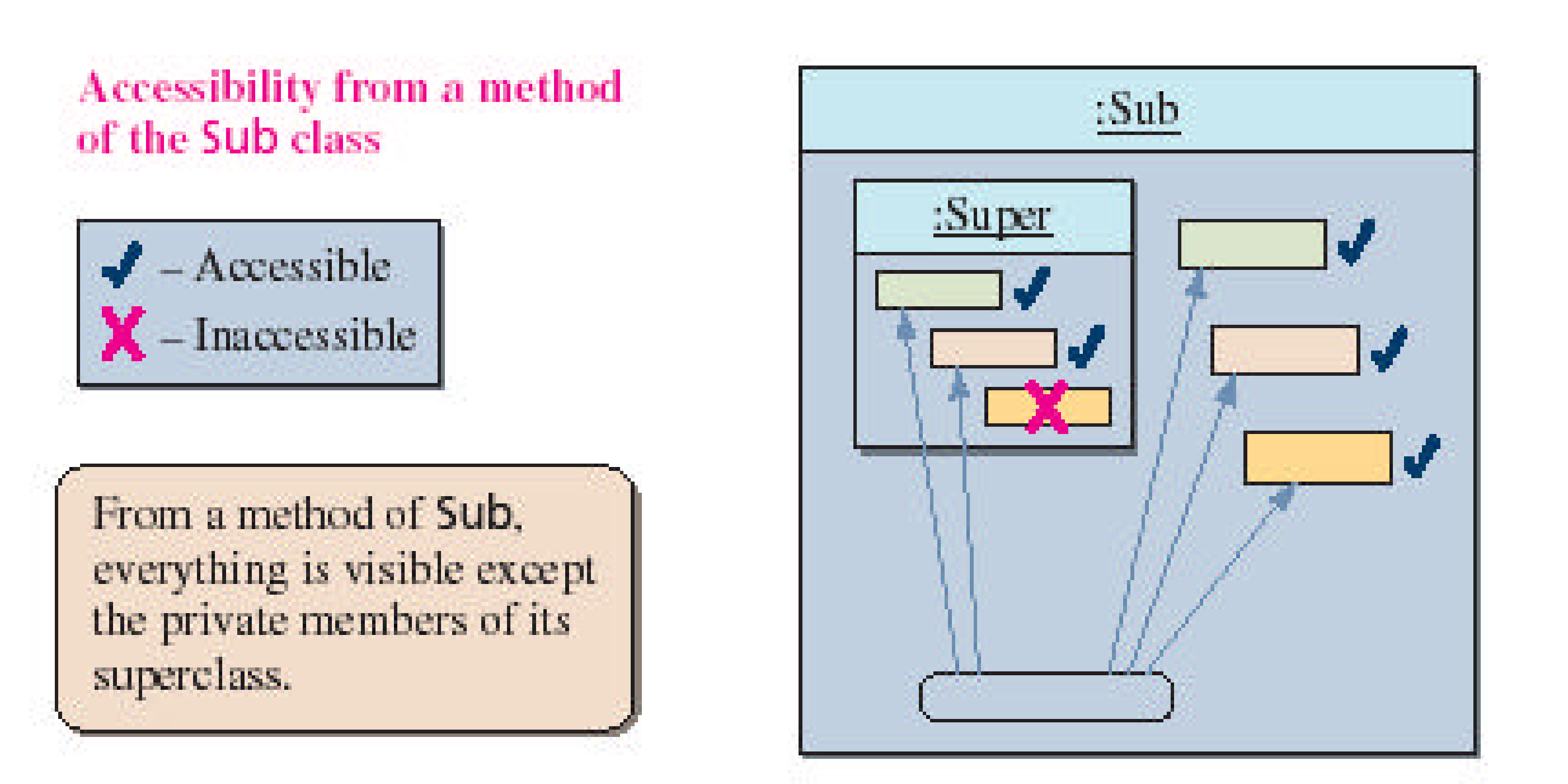
✓ – Accessible

✗ – Inaccessible

Only public members, those defined for the class and those inherited, are visible from outside. All else is hidden from outside.

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 a method of the Sub class

✓ – Accessible

✗ – Inaccessible

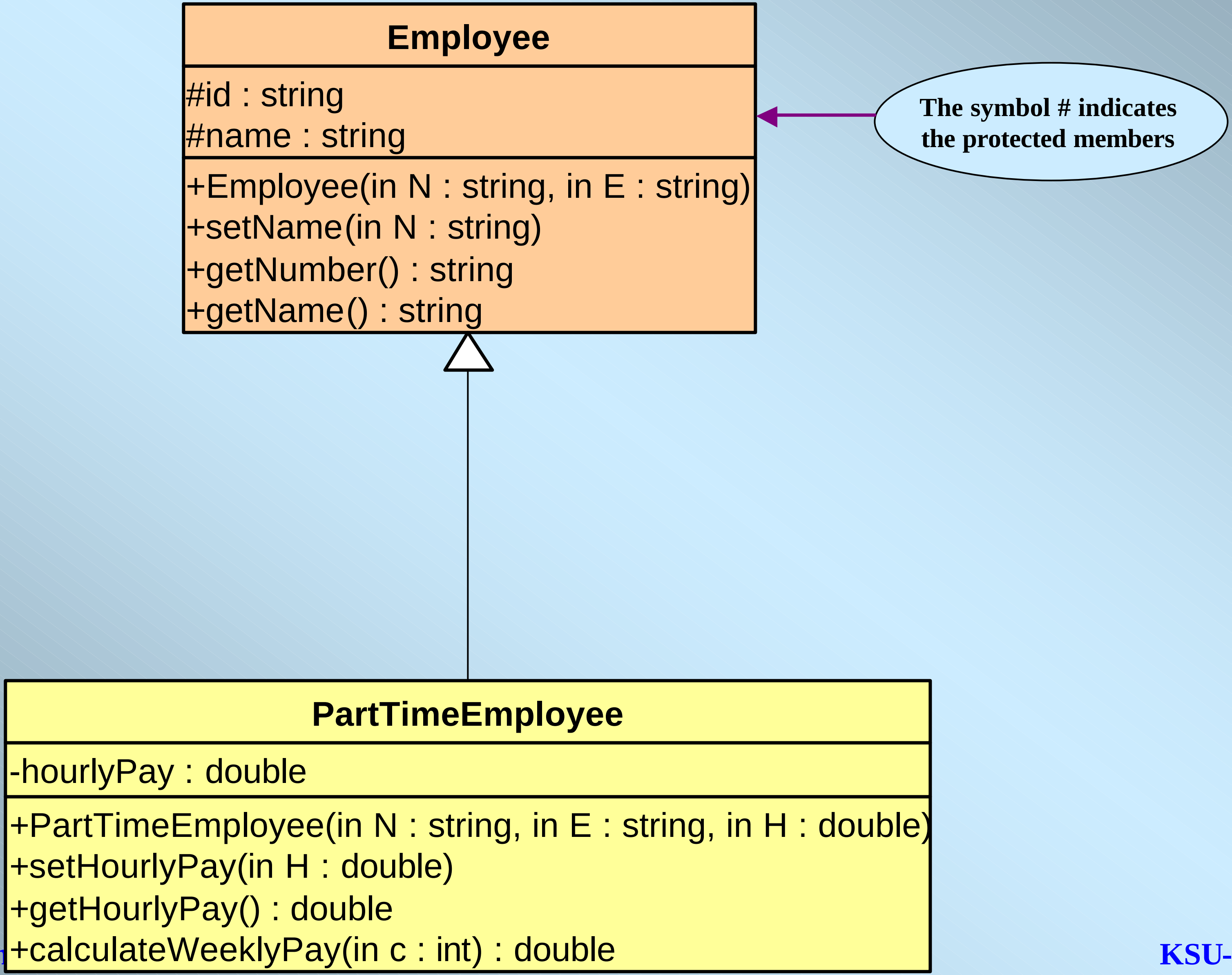
From a method of Sub, everything is visible except the private members of its superclass.

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 (subclasses).
- **Public** data members and methods are accessible to everyone.
- **Private** data members and methods are accessible only to instances of the class.

The Protected Modifier

An inheritance relationship using UML



Employee

#id : string
#name : string

+Employee(in N : string, in E : string)
+setName(in N : string)
+getNumber() : string
+getName() : string

The symbol # indicates the protected members

PartTimeEmployee

-hourlyPay : double

+PartTimeEmployee(in N : string, in E : string, in H : double)
+setHourlyPay(in H : double)
+getHourlyPay() : double
+calculateWeeklyPay(in c : int) : double

Case Study 2: Defining Classes with Inheritance

- 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.

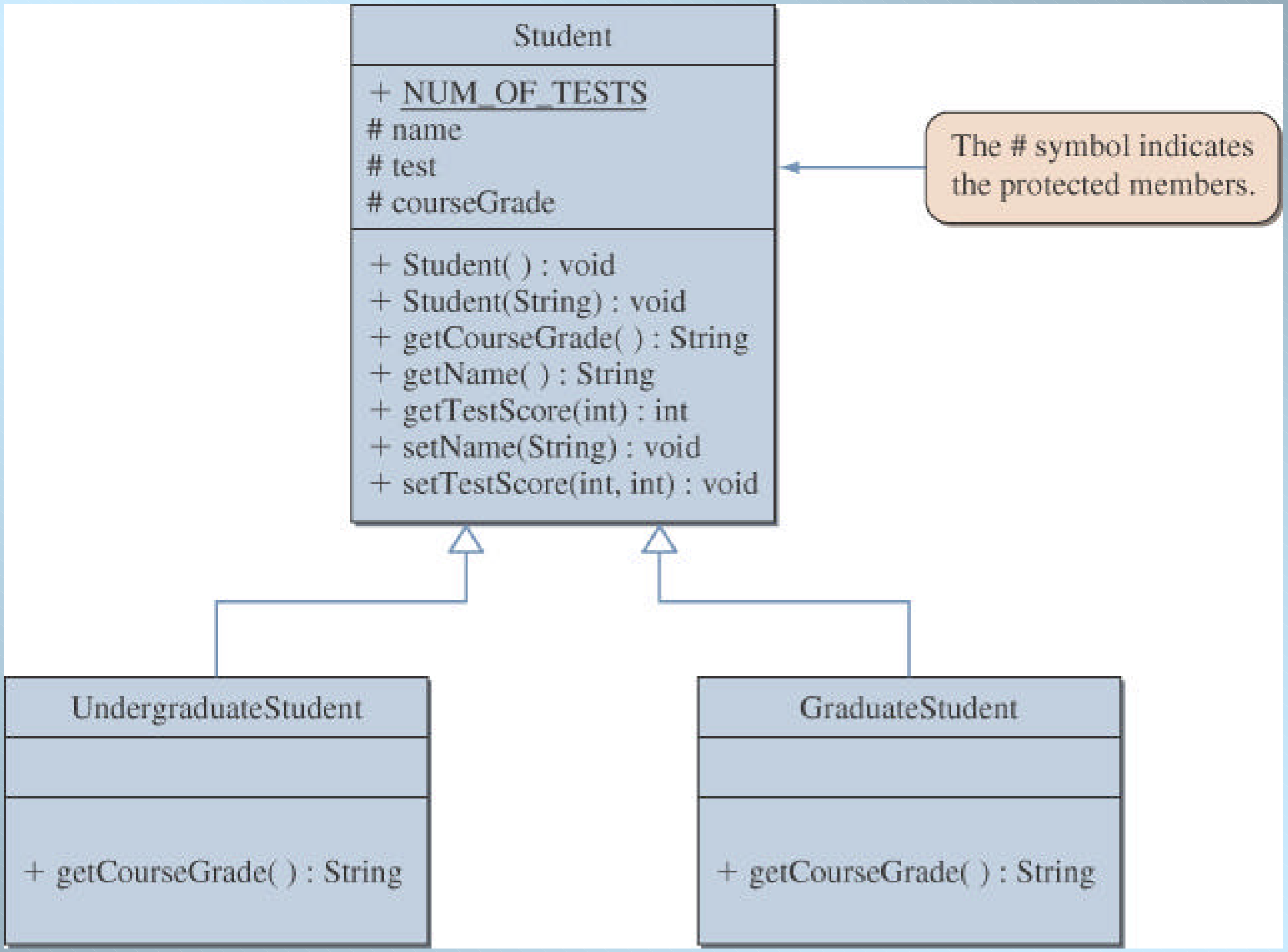
Modeling Two Types of Students

- 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.

4. Inheritance Hierarchy



Student
+ NUM_OF_TESTS
# name
# test
# courseGrade
+ Student( ) : void
+ Student(String) : void
+ getCourseGrade( ) : String
+ getName( ) : String
+ getTestScore(int) : int
+ setName(String) : void
+ setTestScore(int, int) : void
The # symbol indicates the protected members.
UndergraduateStudent
+ getCourseGrade( ) : String
GraduateStudent
+ getCourseGrade( ) : String

5. Declaring Subclasses

```
public class Student
{
   //DATA MEMBERS
   protected String name;
   protected int [ ] test;
   …..
   …..
}
```

Members to be inherited are designated as **protected**

```
public class GraduateStudent extends Student
{
   //DATA MEMBERS
   …..
   …..
}
```

extends allows **GraduateStudent** to inherit **Student**

Implementation of **Case Study 1:**

```
public class Employee
{
   protected String number;
   protected String name;

   public Employee (String N, String E)
   {
      number = N;
      name = E;
    }

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

   public String getNumber()
    {
      return number;
    }

   public String getName()
   {
      return name;
    }
}
```

```
public class PartTimeEmployee extends Employee
{
  private double hourlyPay;

  public PartTimeEmployee(String N, String E, double H)
 {
    number = N;
    name = E;
    hourlyPay = H;
 }

 public void setHourlyPay(double H)
 {
    hourlyPay = H;
 }

  public double getHourlyPay()
 {
    return hourlyPay;
 }

 public double calculateWeeklyPay(int c)
 {
    return hourlyPay * c;
 }

}
```

PartTimeEmployee class test program.

```
import java.util.Scanner;
public class PartTimeEmployeeTest {
   public static void main(String[] args)
  {
     Scanner input = new Scanner(System.in);
     String number, name;
     double pay;
     int hours;
     PartTimeEmployee emp;

     // get the details from the user
     System.out.print (“Employee Number?”);
     number = input.next();
     System.out.print (“Employee Name?”);
     name = input.next();
     System.out.print (“Hourly pay?”);
     pay = input.Double();
     System.out.print (“Hours worked this week?”);
     hours = input.Int();

     // create a new part-time employee
    emp = new PartTimeEmployee (number, name, pay);

    //display employee’s details, including the weekly pay
    System.out.println();
    System.out.println(emp.getName());
    System.out.println(emp.getNumber());
    System.out.println(emp.calculateWeeklyPay(hours));
  }
}
```

Implementation of Case Study 2:

```
class Student {

/** The number of tests this student took */
   protected   final static  int NUM_OF_TESTS = 3;
   protected   String          name;
   protected   int[]           test;
   protected   String          courseGrade;

  public Student( ) { this ("No Name"); }

  public Student(String studentName) {
      name = studentName;
      test = new int[NUM_OF_TESTS];
      courseGrade = "****";
   }
  public void setScore(int s1, int s2, int s3) {
      test[0] = s1; test[1] = s2; test[2] = s3;
   }
  public String getCourseGrade( ) {
     return courseGrade;    }

  public String getName( ) { return name; }

  public int getTestScore(int testNumber) {
     return test[testNumber-1];  }

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

 }
```

```
class GraduateStudent extends Student {
   /**
     * students. Pass if total >= 80; otherwise, No Pass.
   */
   public GraduateStudent(String na)
   { name = na;}
    public void computeCourseGrade() {
       int total = 0;
       for (int i = 0; i < NUM_OF_TESTS; i++) {
         total += test[i]; }
       if (total >= 80) {
         courseGrade = "Pass";
       } else { courseGrade = "No Pass";  }
    }
}
class UndergraduateStudent extends Student {
    public UndergraduateStudent(String na)
    { name = na;}
    public void computeCourseGrade() {
      int total = 0;
      for (int i = 0; i < NUM_OF_TESTS; i++) {
        total += test[i]; }
      if (total / NUM_OF_TESTS >= 70) {
        courseGrade = "Pass";
      } else {   courseGrade = "No Pass";  }
    }
}
```

Student class test program

Since both undergraduate and graduate students are enrolled in a class,
It seems necessary for us to declare two separate arrays, one for graduate students and another for undergraduate students:

```
GraduateStudent gradStudent [20];
UndergraduateStudent undergradStudent [20];
```

```
public class StudentTest {

  public static void main(String[] args) {
    GraduateStudent [] gradStudent= new GraduateStudent[20];
    UndergraduateStudent [] undergradStudent= new UndergraduateStudent[20];

    gradStudent[0] = new GraduateStudent("Ramzi");
    gradStudent[0].setScore (20, 30, 50);
    gradStudent[0].computeCourseGrade();
    System.out.println(gradStudent [0].getCourseGrade( ));

    undergradStudent[0] = new UndergraduateStudent ("Ahmed");
    undergradStudent[0].setScore (10, 17, 13);
    undergradStudent[0].computeCourseGrade();
    System.out.println(undergradStudent[0].getCourseGrade( ));
  }
}
```

6. 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.

- A subclass uses a constructor from the base class to initialize all the data inherited from the base class
    - In order to invoke a constructor from the base class, it uses a special syntax:

```
public class SubClass extends SuperClass
{
   //DATA MEMBERS
    ….
    // Constructors

    super (………);
      …….
}
```

Inheritance and Constructors

- A call to the base class constructor can never use the name of the base class, but uses the keyword **super** instead
- A call to **super** must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to **super**

Inheritance and Constructors

```
public class Employee
{
   protected String number;
   protected String name;

   public Employee (String N, String E)
  {
     number = N;
     name = E;
   }

   …….
}
```

```
public class PartTimeEmployee extends Employee
{
      private double hourlyPay;

              public PartTimeEmployee(String N, String E, double H)
               {
                  number = N;
                  name = E;
                  hourlyPay = H;
               }

 …..
}
```

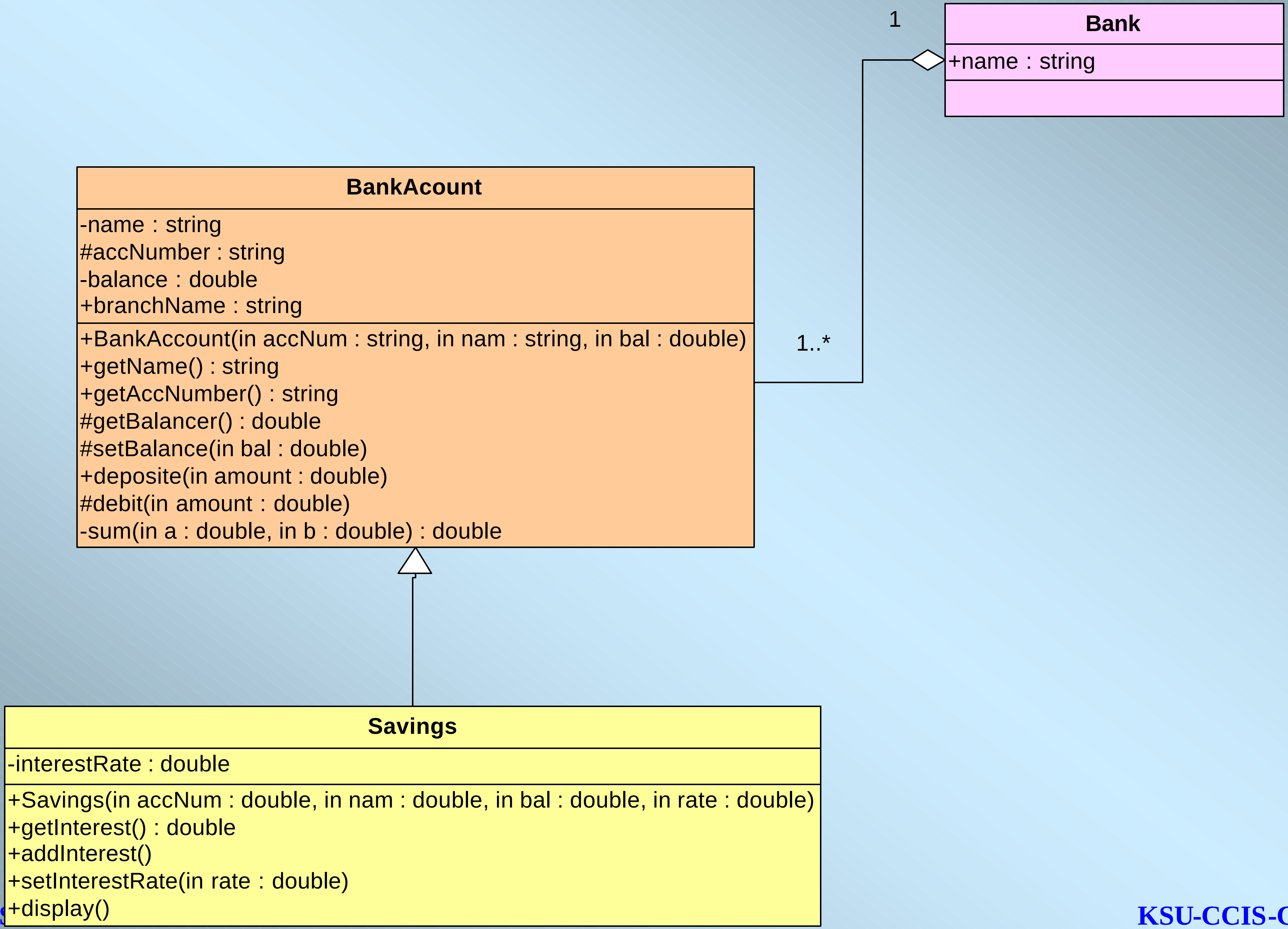
```
public class PartTimeEmployee extends Employee
{
  private double hourlyPay;

  public PartTimeEmployee(String N, String E, double H)
 {
          super (N, E);
          hourlyPay = H;
 }

 …….
}
```

Call to superclass constructor to initialize members inherited from superclass

Case Study 3 : Inheritance Hierarchy of Class BankAccount



Bank

+name : string

BankAcount

-name : string
#accNumber : string
-balance : double
+branchName : string

+BankAccount(in accNum : string, in nam : string, in bal : double)
+getName() : string
+getAccNumber() : string
#getBalancer() : double
#setBalance(in bal : double)
+deposite(in amount : double)
#debit(in amount : double)
-sum(in a : double, in b : double) : double

Savings

-interestRate : double

+Savings(in accNum : double, in nam : double, in bal : double, in rate : double)
+getInterest() : double
+addInterest()
+setInterestRate(in rate : double)
+display()

Implementation of **Case Study 3:**

```
public class BankAccount
{   protected String accNumber;
     private String name;
     private double balance;
     public String branchName;
     public BankAccount(String number, double bal,
                String na , String branNa) {
   accNumber = number;   balance = bal;
    name = na; branchName =branNa;
  }
public String getAccNumber() {return accNumber; }
private double sum( double a, double b) {return a+b;}
public  copy(BankAccount client)
{    accNumber = client.accNumber;
     name = client.name;
   balance=client.balance;
   branchName =client.branchName;
}
protected double getBalance() {return balance; }
protected void setBalance(double bl) { balance = bl;}
public String getName() {return name; }
public void deposite(double amount)  {
                      balance=sum(balance , amount); }
protected void debit(double amount)   {
    if (amount > balance)
 System.out.println("Sorry.. you cannot debit the"+amount);
  else     balance=balance - amount;
  }                }
```

```
public class Savings extends BankAccount
{
private double interestRate;
public Savings(String number, double bal, String na,
String bankNa, double rate) {
                    super(number, bal, na, bankNa);
                    interestRate = rate;
                    }

 public void setInterestRate(double rate) {
                    interestRate = rate;
                    }

public double getInterestRate() {      return
interestRate; }

public void addInterest()  {
  double  interest = (getBalance()* intersetRate )/100;
setBalance(getBalance() + interest);
                      }
public void  display() {
System.out.println(branchName+getName()+accNumber
+getBalance());
}
```

```
public class Bank
{
     private String name;
     private BankAccount [] customer;
     private int nbc;
     public Bank(int size, String  na)
   {
     customer = new BankAccount[size];
      name = na;
      nbc=0;
  }
  public  boolean addCustomers(BankAccount  client)
 {
   if (nbc < customers.length)
       {
         customers[nbc++]= client;
         return true;
     }
  else  return false;
}
```

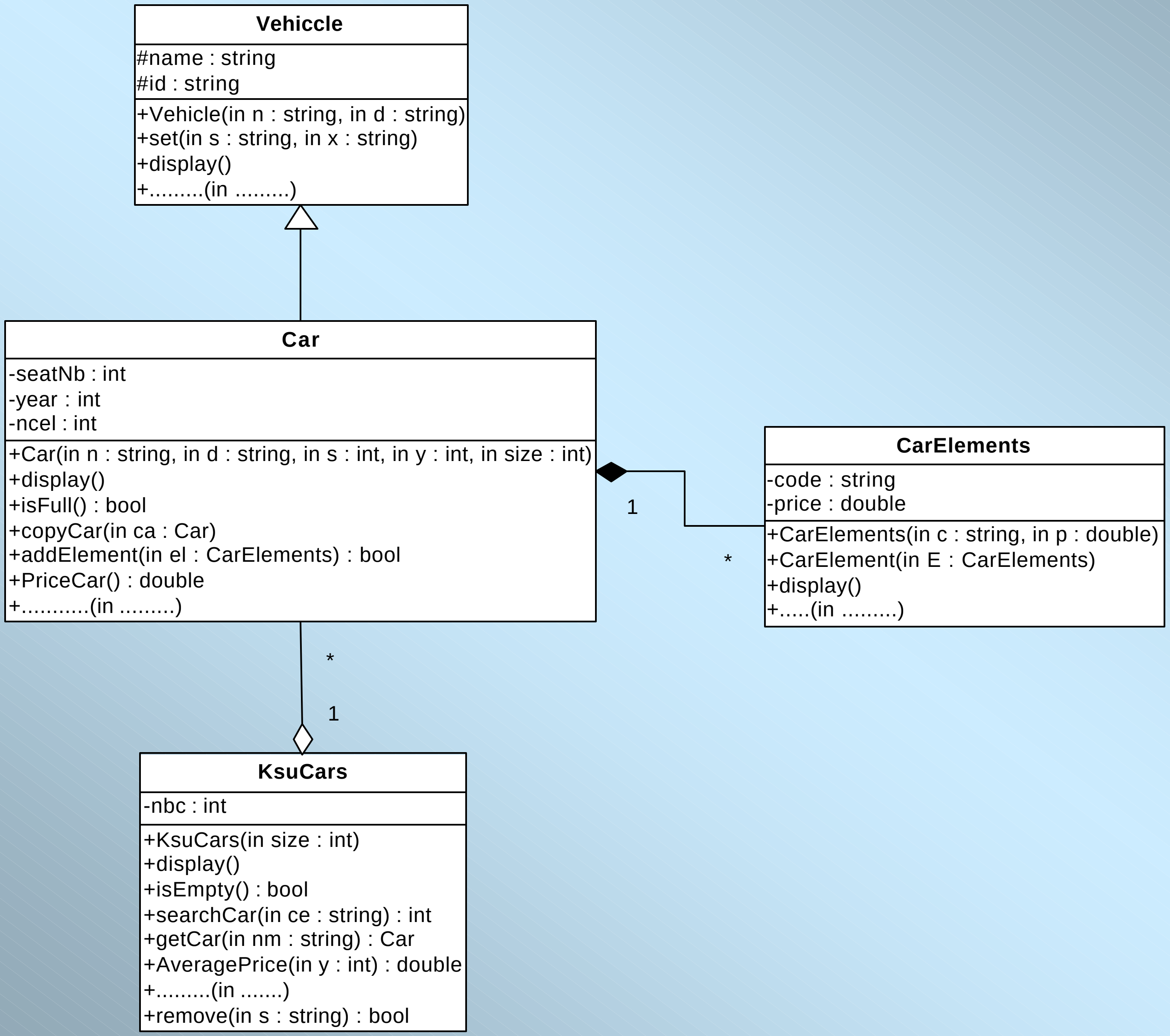
```
public class BankAccountTest {
  public static void main(String[] args)
   {
 Savings savAcc = new Savings("112233", 1000.0, "Ahmed",
"AlMalaz",10.0);

savAcc.display();
savAcc.debit(100.0);  //--- object savAcc inherites method
debit from the superClass BankAccount
 savAcc.display();
savAcc.addInterest(); //--- object savAcc utilizes method
addInterset from  subClass
savAcc.display();
savAcc.deposite(10.5); //--- object savAcc inherites method
deposit from the superClass BankAccount
savAcc.display();
                    }
}
```

```
------------------Execution of the program BankAccountTest-------------------------

Branch Name : AlMalaz  Custemer name : Ahmed  Accunt namber: 112233  Balance : 1000.0
Branch Name : AlMalaz  Custemer name : Ahmed  Accunt namber: 112233  Balance : 900.0
Branch Name : AlMalaz  Custemer name : Ahmed  Accunt namber: 112233  Balance : 990.0
Branch Name : AlMalaz  Custemer name : Ahmed  Accunt namber: 112233  Balance : 1000.5
```

Case Study 4



Vehiccle

#name : string
#id : string

+Vehicle(in n : string, in d : string)
+set(in s : string, in x : string)
+display()
+.........(in ....)

Car

-seatNb : int
-year : int
-ncel : int

+Car(in n : string, in d : string, in s : int, in y : int, in size : int)
+display()
+isFull() : bool
+copyCar(in ca : Car)
+addElement(in el : CarElements) : bool
+PriceCar() : double
+...........(in ....)

CarElements

-code : string
-price : double

+CarElements(in c : string, in p : double)
+CarElement(in E : CarElements)
+display()
+.....(in ....)

KsuCars

-nbc : int

+KsuCars(in size : int)
+display()
+isEmpty() : bool
+searchCar(in ce : string) : int
+getCar(in nm : string) : Car
+AveragePrice(in y : int) : double
+.........(in ....)
+remove(in s : string) : bool

Question: Implement all the classes with all their methods using the following descriptions.

Description of the different classes:

Class Vehicle:

- ✓ *The method* ***display ()*** displays the name and the id.
- ✓ **+ …….. (in ……..) :** if you need an other methods in this class you can add it.

Class CarElements:

- ✓ *The method* **display ()** displays the code and the price.
- ✓ **+ …….. (in ……..)** : if you need an other methods in this class you can add it.

You can't add another constructor.

Class Car:

- seatNb : *Number of seats*
- year : *Production year of car*
- ncel : *number of CarElements object currently in an object of the class Car.*
- ***And other attribute(s) deduced from the UML diagram.***

- ✓ **display ():** Displays all the attributes of an object Car.
- ✓ **addElement (CarElements el)**: This method receives a CarElements object and adds it to the Car object.
- ✓ **priceCar()**: Returns the sum of the CarElements price in an object of the class Car.

+ …….. (in ……..) : if you *need an other methods in this class you can add it.*

Class KsuCars:

- nbc : *number of Car currently in an object of the class KsuCar.*
- ***And other attribute(s) deduced from the UML diagram.***

- ✓ **display ():** Displays all the attributes of an object KsuCars.
- ✓ **search (String ce):** This method receives a String representing the *name* of a Car object and returns the array index of the car object.
- ✓ **getCar (String nm):** This method receives a String representing the *id* of a Car object and returns the Car object if it's exist.
- ✓ **removeCar (String s)**: Removes a Car according to its name. It will return a value *true* if the operation has been completed successfully, or *false* if not.
- ✓ **AveragePrice(int y):** Calculates the average price of all car in an object of class KsuCars that produced after the year **y**.
- ✓ *+ …….. (in ……..) : if* you *need an other methods in this class you can add it.*