Chapter 1

Overview

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

Dr. S. HAMMAMI

Objectives

- After you have read and studied this chapter, you should be able to
  - Define a class with multiple methods and data members
  - Define and use value-returning methods
  - Pass both primitive data and objects to a method
  - Manipulate a collection of data values, using an array.
  - Declare and use an array of objects in writing a program

OUTLINE

1. Passing Objects to a Method
2. Returning an Object From a Method
3. The Use of this in the add Method
4. Overloaded Methods
5. Arrays of Objects
6. Examples

1. Passing Objects to a Method

- As we can pass **int** and **double** values, we can also pass an **object** to a method.

- When we pass an object, we are actually passing the **reference** (name) of an object

  – it means a duplicate of an object is NOT created in the called method

LibraryCard class: A LibraryCard object is owned by a Student, and it records the number of books being checked out.

Student class

```
/*
   File: Student.java
*/

class Student {
   // Data Member
   private String name;
   private String email;

   //Constructor
   public Student() {
      name = "Unassigned";
      email = "Unassigned";
   }
   //Returns the email of this student
   public String getEmail( ) {
      return email;
   }
   //Returns the name of this student
   public String getName( ) {
           return name;
   }
   //Assigns the name of this student
   public void setName(String studentName) {
      name = studentName;
   }
   //Assigns the email of this student
   public void setEmail(String address) {
      email = address;
   }
 }
```

LibraryCard class

```
/*
   File: LibraryCard.java
 */
class LibraryCard {
   //student owner of this card
   private Student owner;
   //number of books borrowed
   private int borrowCnt;
   //numOfBooks are checked out
   public void checkOut(int numOfBooks) {
      borrowCnt = borrowCnt + numOfBooks;
   }
   //Returns the name of the owner of this card
   public String getOwnerName( ) {
      return owner.getName( );
   }
   //Returns the number of books borrowed
   public int getNumberOfBooks( ) {
      return borrowCnt;
   }
   //Sets the owner of this card to student
   public void setOwner(Student student) {
      owner = student;
   }
   //Returns the string representation of this card
   public String display( ) {
      return  "Owner Name:     " + owner.getName( ) + "\n" +
            "      Email:    " + owner.getEmail( ) + "\n" +
            "Books Borrowed: " + borrowCnt;
   }
 }
```

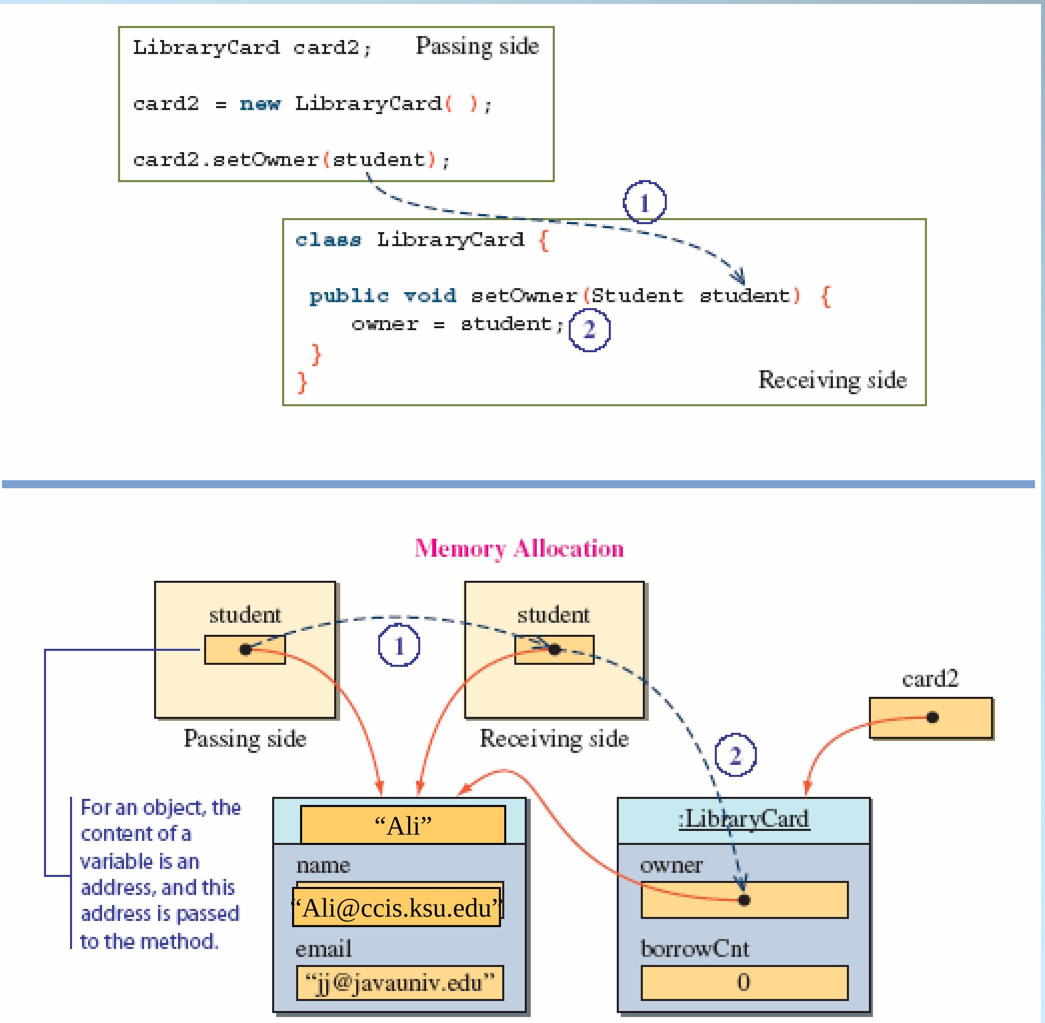
Passing a Student Object

Suppose a single student owns two library cards. Then we can make the data member owner of two LibraryCard Objects to refer to the same Student object. Here’s one such program

```
/*
   File: Librarian.java
*/
class Librarian {
   public static void main( String[] args ) {

      Student    student;
      LibraryCard card1, card2;

      student = new Student( );
      student.setName(“Ali");
      student.setEmail(“ali@ccis.ksu.edu");

      card1 = new LibraryCard( );
      card1.setOwner(student);
      card1.checkOut(3);

      card2 = new LibraryCard( );
      card2.setOwner(student); //the same student is the owner
                               //of the second card, too

      System.out.println("Card1 Info:");
      System.out.println(card1.display() + "\n");

      System.out.println("Card2 Info:");
      System.out.println(card2.display() + "\n");

   }
}
```

Passing a Student Object

When we say pass an object to a method, we are not sending a copy of an object, but rather a reference to the object.
This diagram illustrates how an objects is passed as an arguments to a method



```
LibraryCard card2;

card2 = new LibraryCard( );

card2.setOwner(student);
```

Passing side

```
class LibraryCard {

 public void setOwner(Student student) {
     owner = student;
 }
}
```

Receiving side

Memory Allocation

For an object, the content of a variable is an address, and this address is passed to the method.

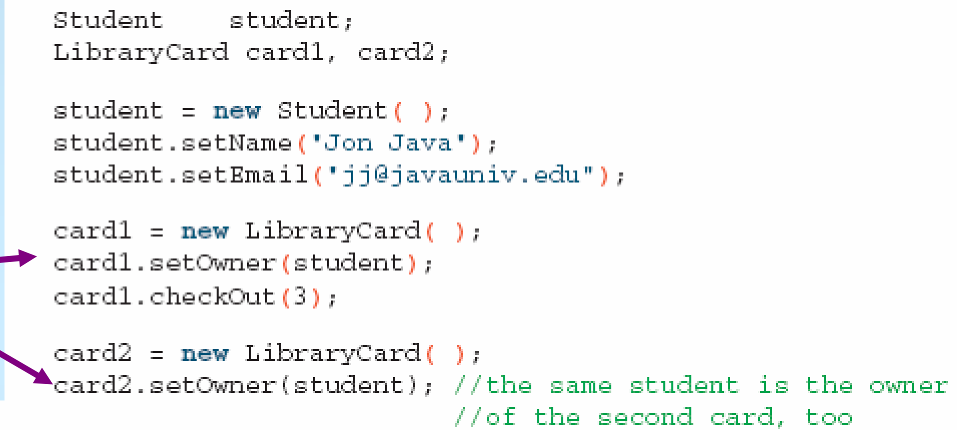
Sharing an Object

In this program, we create one Student object. Then we create two LibraryCard objects. For each of these LibraryCard objects, we pass the same student when calling their setOwner methods:

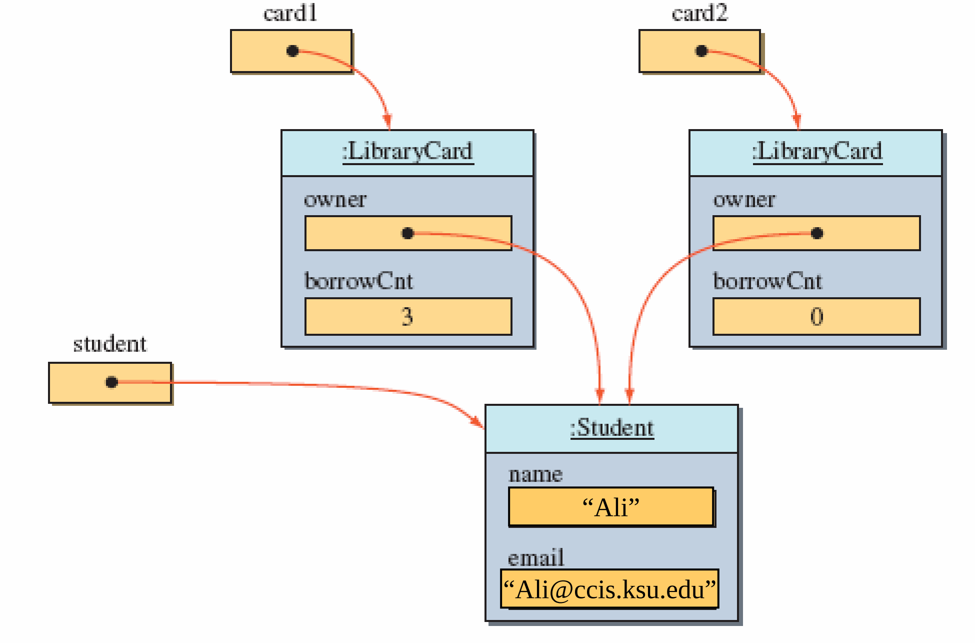
card1.setOwner(student);
….
card2.setOwner(student);

After the setOwner method of card2 is called in the main method, we have the following state of memory.

- We pass the same Student object to card1 and card2



```
Student     student;
LibraryCard card1, card2;

student = new Student( );
student.setName("Jon Java");
student.setEmail("jj@javauniv.edu");

card1 = new LibraryCard( );
card1.setOwner(student);
card1.checkOut(3);

card2 = new LibraryCard( );
card2.setOwner(student); //the same student is the owner
                         //of the second card, too
```



- Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the same Student object

2. Returning an Object From a Method

- As we can return a primitive data value from a method, we can return an object from a method also.
- We return an object from a method, we are actually returning a reference (or an address) of an object.
  - This means we are not returning a copy of an object, but only the reference of this object

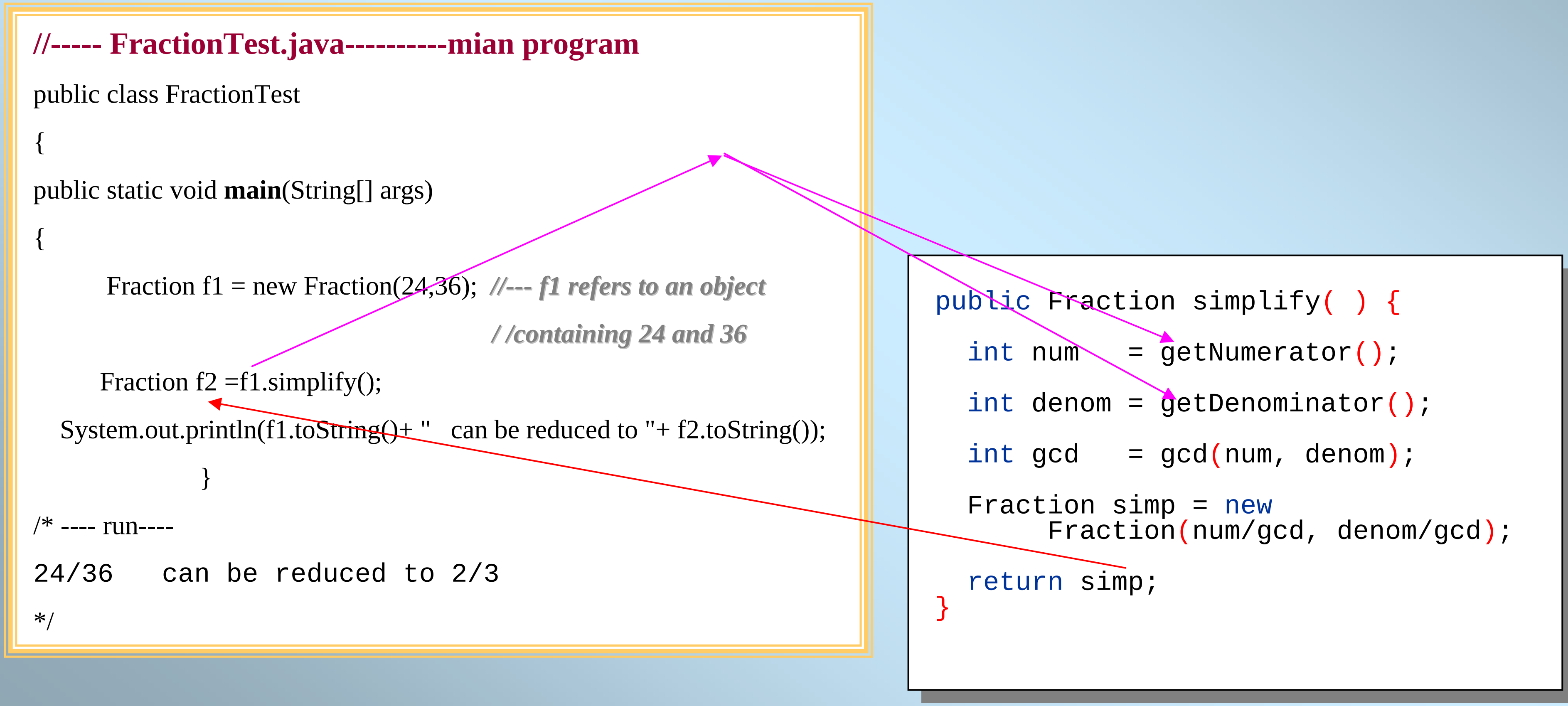
```
//== Class Fraction============
public class Fraction
{  private int numerator;
    private int  denominator;
    //===== Constructors ========//
    public Fraction()     {this(0,1);  }
    public Fraction(int number) {this(number,1); }
   public Fraction(Fraction frac)
   {this(frac.getNumerator(), frac.getDenominator()) }
    public Fraction(int num, int denom)
  {setNumerator(num); setDenominator(denom); }
  //======Public Instance Methods =============
   public int getNumerator()  {return (numerator); }
  public int getDenominator() { return (denominator); }
  public void setNumerator(int num) {numerator=num; }
  public void setDenominator(int denom)
 {   if (denom == 0)
      {  System.out.println("Fatal error, divid by zero");
       System.exit(1);
      }
    denominator=denom;
 }
```

```
//== Class Fraction:  continue ============
//---- sum = this + frac ---------
public Fraction add(Fraction frac)
{   int n1,d1, n2,d2;
    n1=this.getNumerator();  d1=this.getDenominator();
    n2=frac.getNumerator();  d2=frac.getDenominator();
    Fraction sum =new Fraction(n1*d2+d1*n2, d1*d2);
    return(sum);
}
//----- sum = this + number ---------
public Fraction add(int number)
{  Fraction frac = new Fraction(number, 1);
   Fraction sum = this.add(frac);
   return(sum);
}
//----- sub = this - frac ----------
public Fraction subtract(Fraction frac)
{  int n1,d1, n2,d2;
   n1=numerator;      d1=denominator;
   n2=frac.numerator;  d2=frac.denominator;
    Fraction sub =new Fraction(n1*d2-d1*n2, d1*d2);
   return(sub);
}
//----- sub = this - number ---------
public Fraction subtract(int number)
{  Fraction frac = new Fraction(number, 1);
    return(this.subtract(frac));
}
```

```
//== Class Fraction continue ==========
//---- mult = this * frac ---------
public Fraction multiply(Fraction frac)
{
   int n1,d1, n2,d2;
   n1=this.getNumerator(); d1=this.getDenominator();
   n2=frac.getNumerator();  d2=frac.getDenominator();
   Fraction mult =new Fraction(n1*n2, d1*d2);
    return(mult);
 }
  //----- mult = this * number ---------
 public Fraction multiply(int number)
 {
   Fraction frac = new Fraction(number, 1);
   return(this.multiply(frac));
}
//---- div = this / frac ---------
public Fraction divide(Fraction frac)
{
   int n1,d1, n2,d2;   n1=numerator;   d1=denominator;
   n2=frac.getNumerator();    d2=frac.getDenominator();
  Fraction div =new Fraction(n1*d2, d1*n2);return(div);
}
 //----- mult = this / number ---------
 public Fraction devide(int number)
  {
     Fraction frac = new Fraction(number, 1);
     return(this.divide(frac));   }
  public boolean equals(Fraction frac)
   {
 Fraction f1 =this.simplify();
Fraction f2 =frac.simplify();
if ((f1.getDenominator()== f2.getDenominator()) &&
f1.getNumerator()== f2.getNumerator())  return true;
return false;
}
```

```
//== Class Fraction:  continue ============
public Fraction simplify()
{
      int num =getNumerator();     int denom= this.getDenominator();
     int gcd =this.gcd(num,denom );
    Fraction simp =new Fraction(num/gcd, denom/gcd);
     return(simp);
}
public String toString()
{
return (this.getNumerator() + "/" + this.getDenominator());
}
//====Class Methods================
public static int gcd(int m,int n)
{
     int r= n%m;
     while(r !=0) { n=m;    m=r;  r=n%m;}  return (m);
}
public static Fraction minimum(Fraction f1, Fraction f2)
{
     double  dec1 = f1.decimal();
     double  dec2 = f2.decimal();
     if (dec1 < dec2)  return (f1);
        return f2;
}
//======= Private Methods==============
private double decimal()
{
  return(this.getNumerator()/ this.getDenominator());
}

}//---- end of calss Fraction---
```

When we say “return an object from a method”, we are actually returning the address, or the reference, of an object to the caller

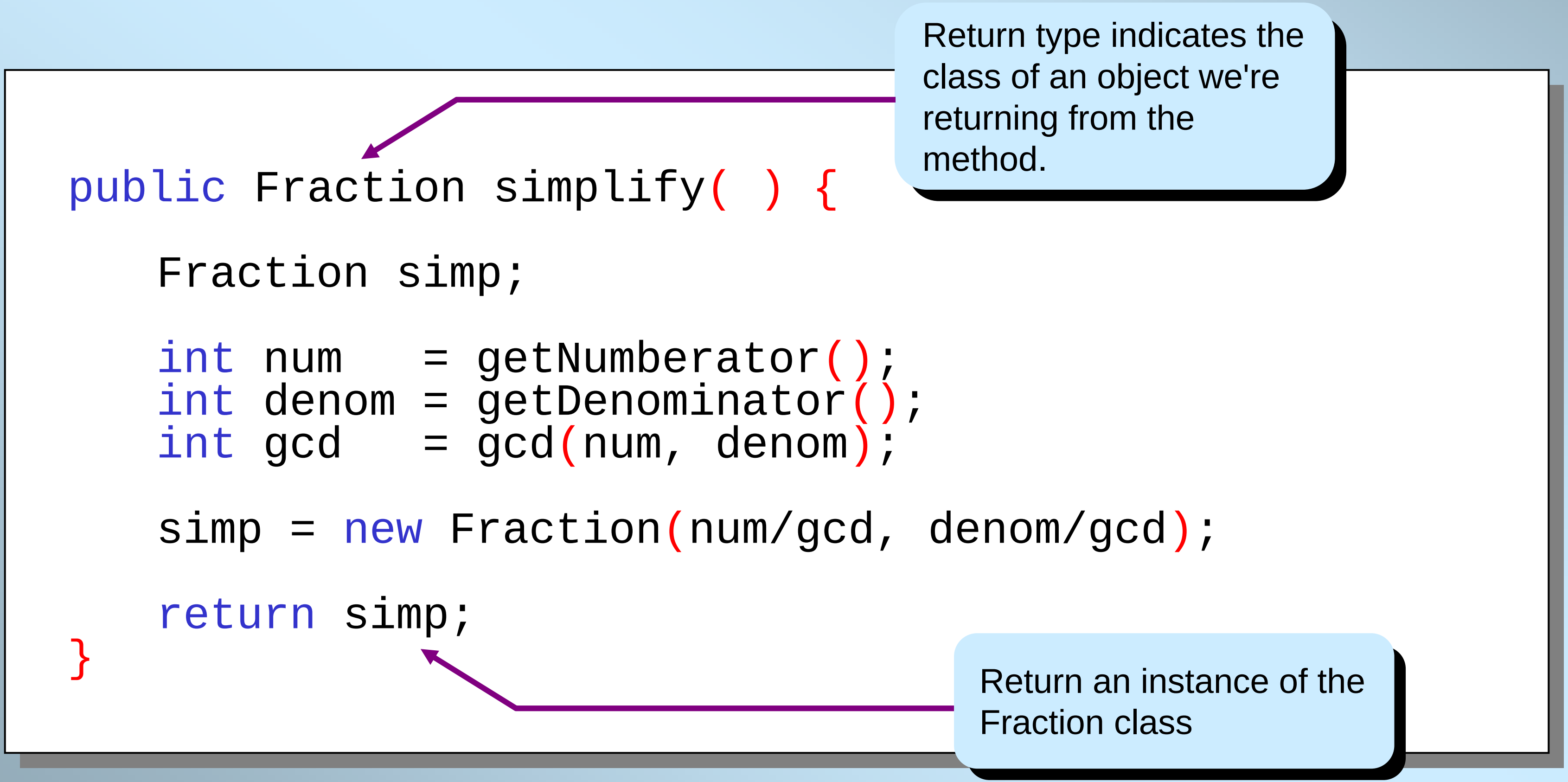


```
//----- FractionTest.java----------mian program
public class FractionTest
{
public static void main(String[] args)
{
          Fraction f1 = new Fraction(24,36);  //--- f1 refers to an object
                                              / /containing 24 and 36
         Fraction f2 =f1.simplify();
   System.out.println(f1.toString()+ "   can be reduced to "+ f2.toString());
                    }
/* ---- run----
24/36   can be reduced to 2/3
*/
```

```
public Fraction simplify( ) {
   int num   = getNumerator();
   int denom = getDenominator();
   int gcd   = gcd(num, denom);
   Fraction simp = new
        Fraction(num/gcd, denom/gcd);
   return simp;
}
```

Sample Object-Returning Method

- Here's a sample method that returns an object:



```
public Fraction simplify( ) {

    Fraction simp;

    int num   = getNumberator();
    int denom = getDenominator();
    int gcd   = gcd(num, denom);

    simp = new Fraction(num/gcd, denom/gcd);

    return simp;
}
```

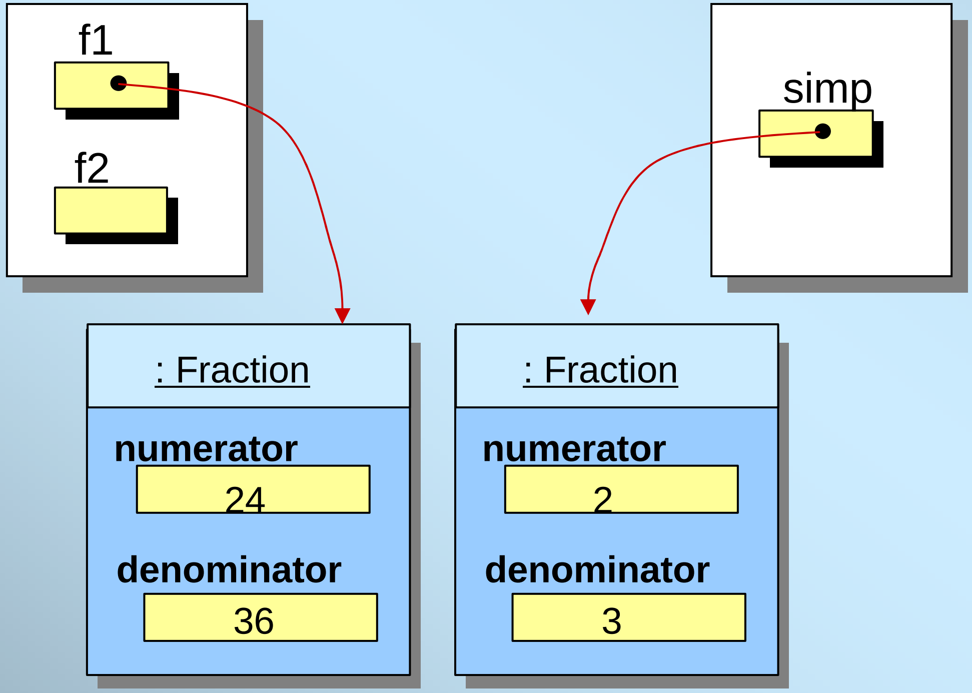
Return type indicates the class of an object we're returning from the method.

Return an instance of the Fraction class

A Sample Call to simplify

```
f1 = new Fraction(24, 26);
```

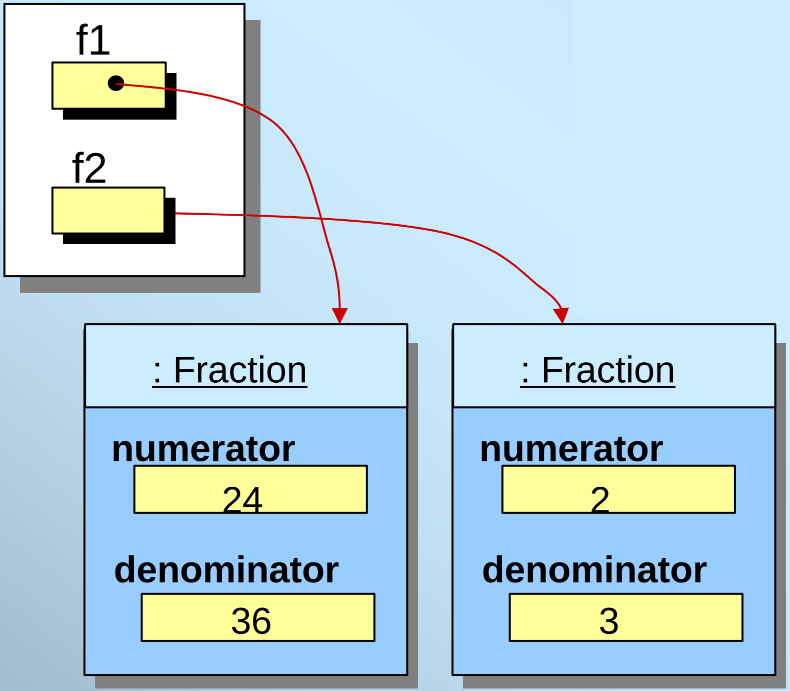
```
public Fraction simplify( ) {

   int num   = getNumerator();
   int denom = getDenominator();
   int gcd   = gcd(num, denom);

   Fraction simp = new
        Fraction(num/gcd, denom/gcd);

   return simp;
}
```



A Sample Call to simplify (cont'd)

```
f1 = new Fraction(24, 26);
```

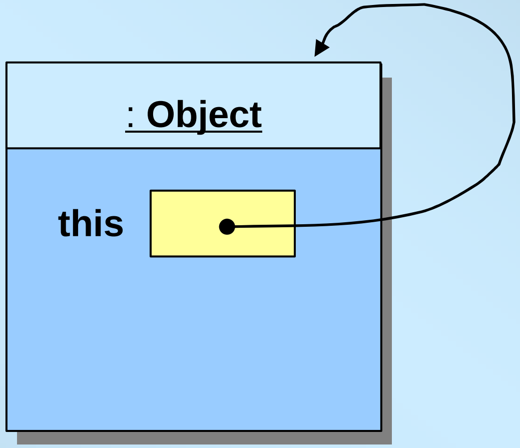
```
public Fraction simplify( ) {

   int num   = getNumerator();
   int denom = getDenominator();
   int gcd   = gcd(num, denom);

   Fraction simp = new
        Fraction(num/gcd, denom/gcd);

   return simp;
}
```



The value of simp, which is a reference, is returned and assigned to f2.

Reserved Word this

- The reserved word this is called a *self-referencing pointer* because *it refers to an object* from the object's method.

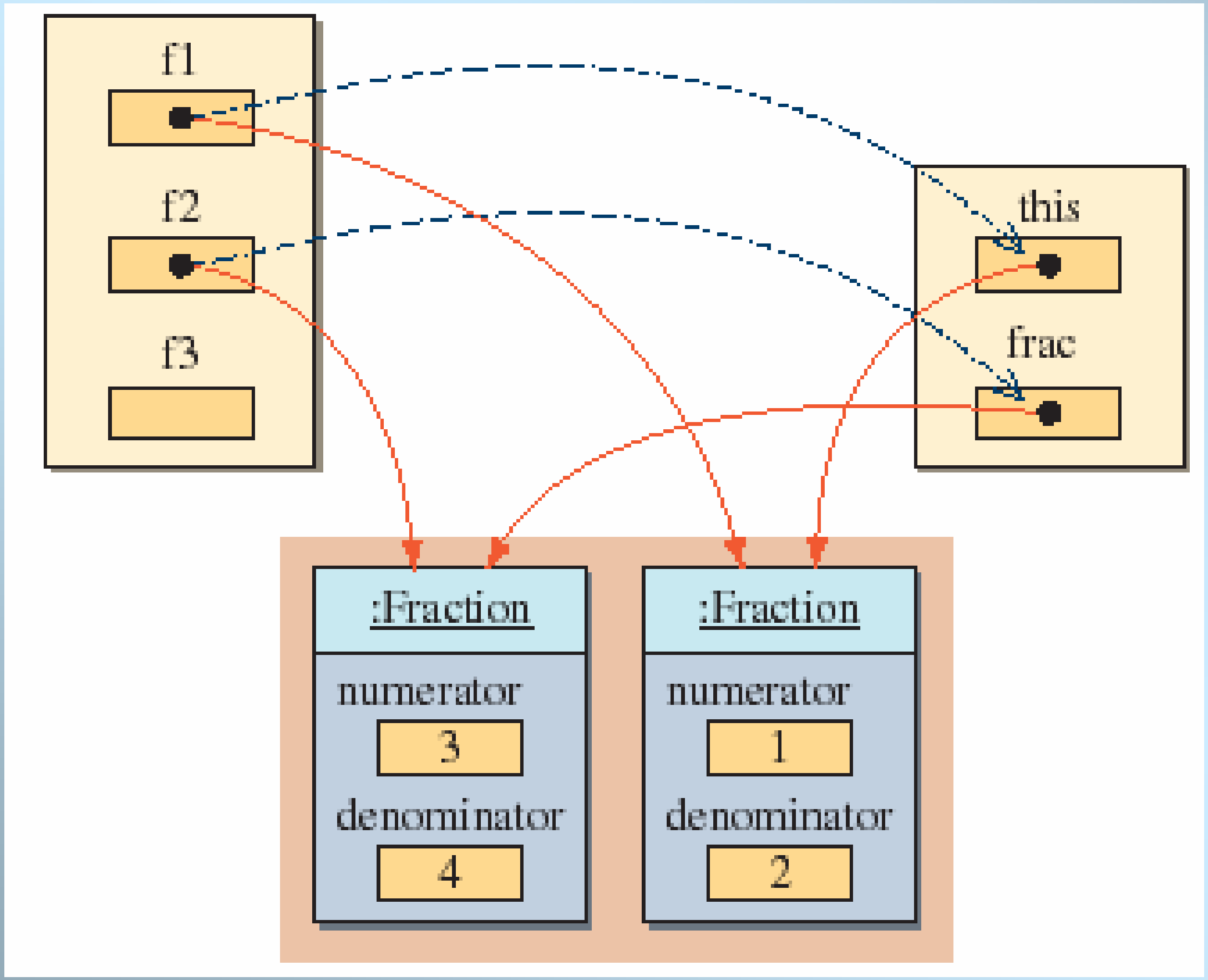


- The reserved word this can be used in different ways. We will see all uses in this chapter.

3. The Use of this in the add Method

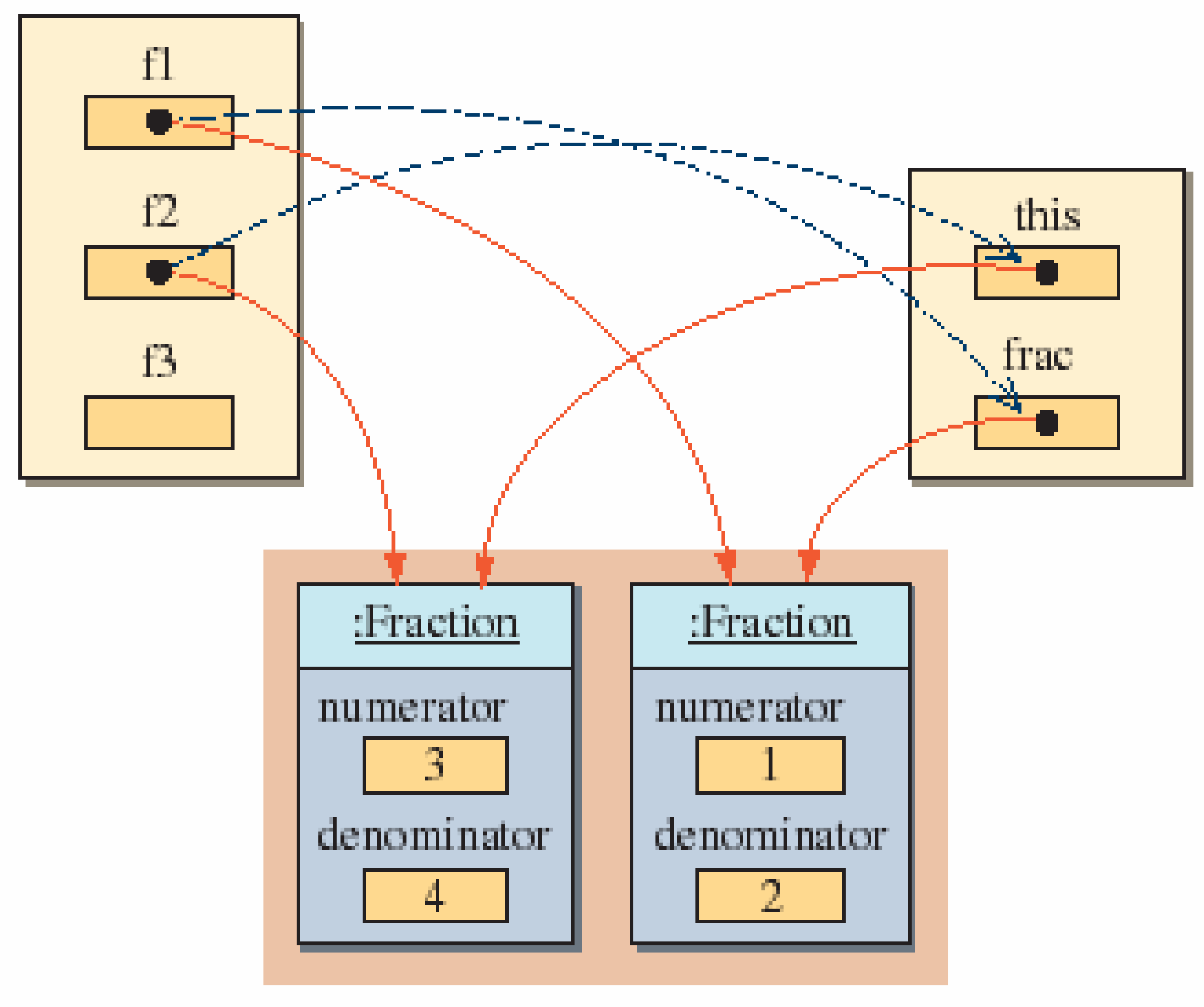
```
public Fraction add(Fraction frac) {

   int      a, b, c, d;
   Fraction sum;

   a = this.getNumerator();   //get the receiving
   b = this.getDenominator(); //object's num and denom

   c = frac.getNumerator();   //get frac's num
   d = frac.getDenominator(); //and denom

   sum = new Fraction(a*d + b*c, b*d);

   return sum;
}
```

f3 = f1.add(f2)



Because f1 is the receiving object (we're calling f1's method), so the reserved word this is referring to f1.

f3 = f2.add(f1)



This time, we're calling f2's method, so the reserved word this is referring to f2.

Using this to Refer to Data Members

- In the previous example, we showed the use of this to call a method of a receiving object.
- It can be used to refer to a data member as well.

```
class Person {

   int  age;

    public void setAge(int val) {
       this.age = val;
   }
   . . .
}
```

4. Overloaded Methods

- Methods can share the same name as long as
  - they have a different number of parameters (Rule 1) or
  - their parameters are of different data types when the number of parameters is the same (Rule 2)

Note: It is not necessary to create an object for f3 and f4

```
//----- FractionTest.java----------mian program

public class FractionTest {

public static void main(String[] args)   {

        Fraction f1, f2, f3,f4;

        f1 = new Fraction(3,4);  //-- create an object for f1

        f2 = new Fraction(2,5);  //--create and object for f2

        f3=f1.multiply(f2);   //--- f3 = f1 x f2  = 6 / 20

         f4=f1.multiply(6);   //--- f4 = f1 x 6   = 18 / 4

        System.out.println(" f3 =  "+ f3.toString()+

                                   "  and f4 =  "+ f4.toString());
}

/* ---- run----

 f3 =  6/20  and f4 =  18/4

*/
```

```
//---- mult = this * frac ---------
public Fraction multiply(Fraction frac)
{
   int n1,d1, n2,d2;
   n1=this.getNumerator(); d1=this.getDenominator();
   n2=frac.getNumerator();  d2=frac.getDenominator();
   Fraction mult =new Fraction(n1*n2, d1*d2);
    return(mult);
 }
  //----- mult = this * number ---------
  public Fraction multiply(int number)
 {
   Fraction frac = new Fraction(number, 1);
   return(this.multiply(frac));
 }
```

5. Arrays of Objects

- In Java, in addition to arrays of primitive data types, we can declare arrays of objects

- An array of primitive data is a powerful tool, but an array of objects is even more powerful.

- The use of an array of objects allows us to model the application more cleanly and logically.

The Person Class

- We will use Student objects to illustrate the use of an array of objects.

```
public class Person
{
            private String name;
            private int age;
            private char gender;
            public Person()    {age=0; name=" "; gender=' ';}
            public Person(String na, int ag, char gen)  {setAge(ag); setName(na); setGender(gen); }
            public Person(Person pr)       { setPerson(pr);}
            public void setPerson(Person p)
            { age=p.age; gender =p.gender;
              name=p.name. substring(0, p.name.length());      }
            public void setAge (int a) {age=a;}
            public void setGender (char g) {gender=g;}
            public void setName(String na)
             {name= new String(na);}
            public int getAge(){return age;}
            public char getGender () {return gender;}
            public String getName () { return name;}
}
```

Creating an Object Array - 1

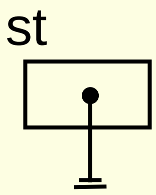
Code

A

```
Student[ ]  st;

st = new Student[20];

st[0] = new Student( );
```

Only the name pr is declared, no array is allocated yet.

State of Memory



After A is executed

Creating an Object Array - 2

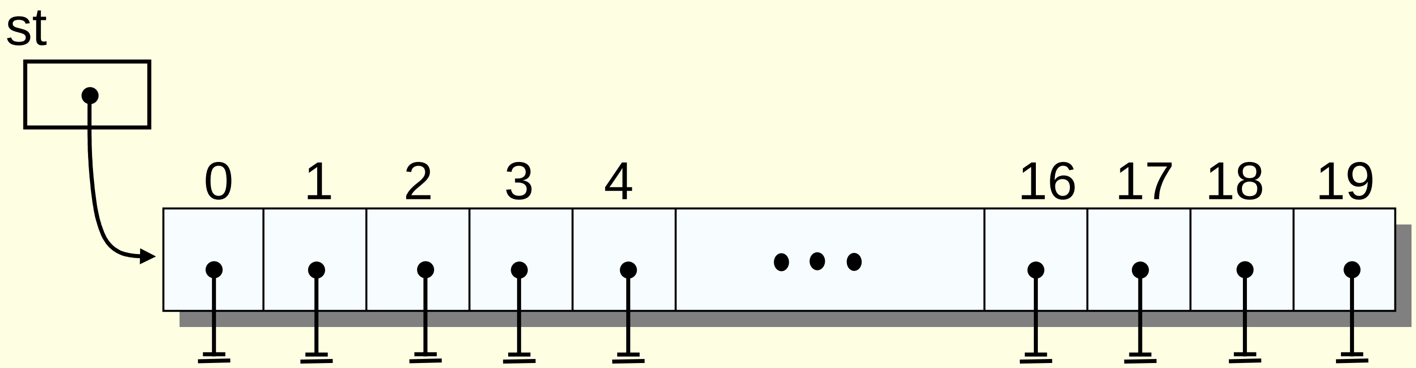
Code

B

```
Student[ ]  st;
st = new Student[20];
st[0] = new Student( );
```

Now the array for storing 20 Student objects is created, but the Student objects themselves are not yet created.

State of Memory



After B is executed

Creating an Object Array - 3

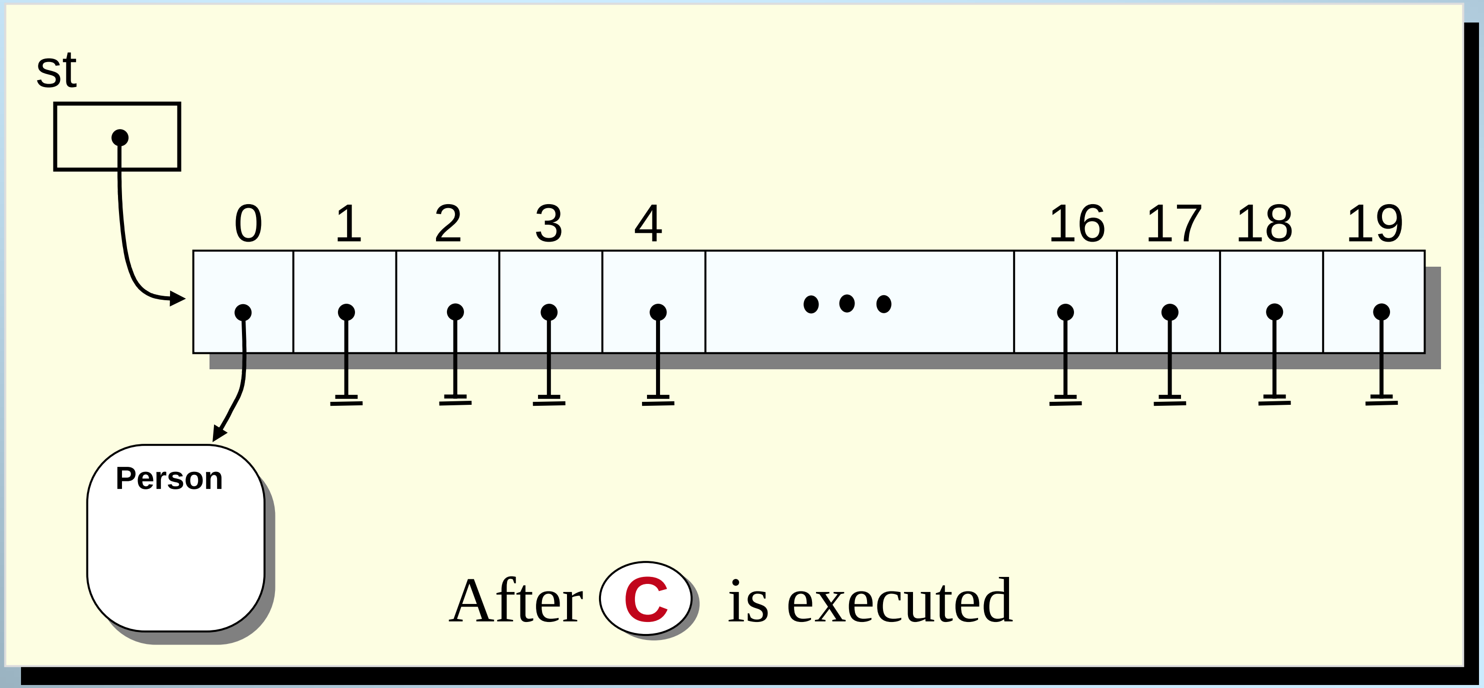
Code

C

```
Student[ ]  st;
st = new Student[20];
st[0] = new Student( );
```

One Student object is created and the reference to this object is placed in position 0.

State of Memory

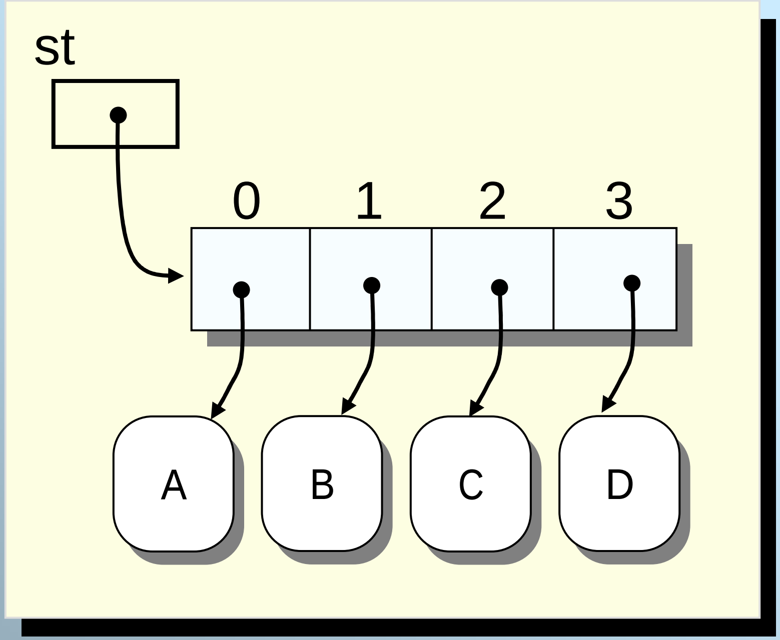


Object Deletion – Approach 1

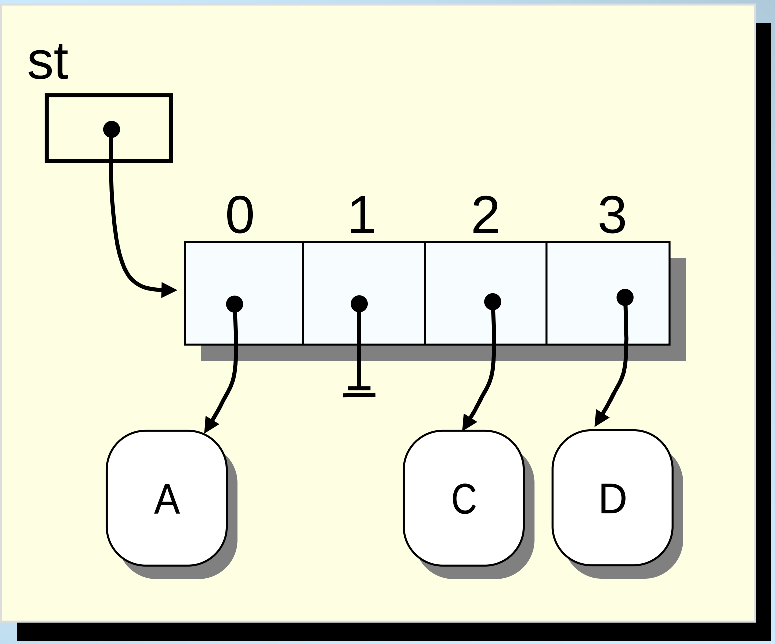
```
int Idx = 1;
st[Idx] = null;
```

A

Delete Student B by setting the reference in position 1 to null.



Before A is executed



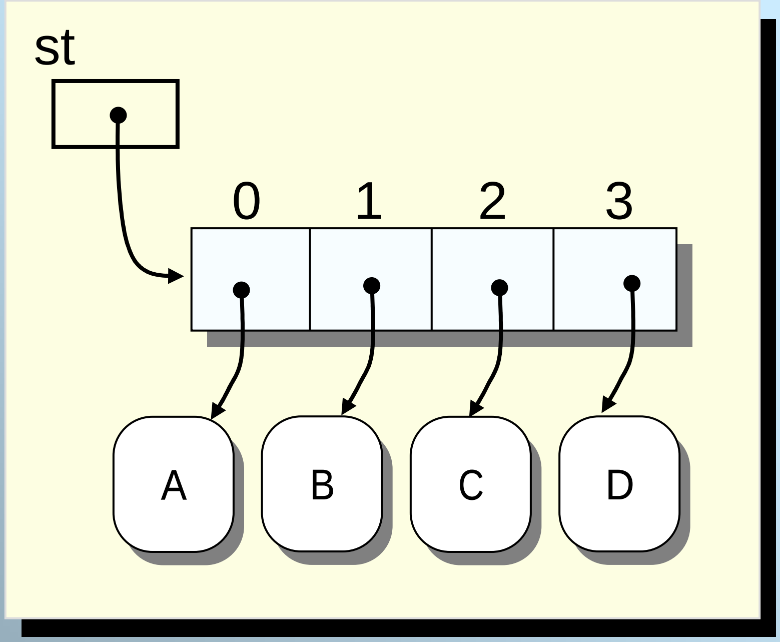
After A is executed

Object Deletion – Approach 2

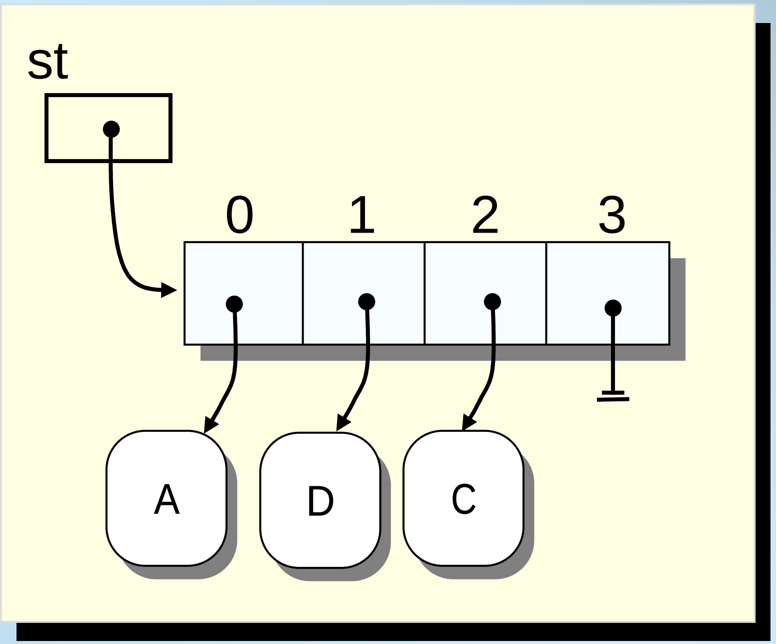
```
int Idx = 1, last = 3;
st[Idx] = st[last];
st[last]     = null;
```

A

Delete Student B by setting the reference in position 1 to the last person.



Before A is executed



After A is executed

Person Array Processing – Sample 1

Create Person objects and set up the p array.

```
import java.util.Scanner;

public class ArrayOfPersons {

private Person p[];

  private int nbp;
  Scanner input = new Scanner(System.in);

  public ArrayOfPersons(int size)

   {
       p = new Person[size];
       nbp=0;
    }

public ArrayOfPersons(Person pr[])
   {
       p = new Person[pr.length];
       for (int i =0; i< p.length; i++)
         p[i]= new Person(pr[i]); // p[i]=pr[i];
       nbp=p.length;
};

public void setArrayOfPersons(Person [] pr)
  {
     for (int i =0; (i< pr.length) && (i<p.length); i++)
       { p[i].setPerson(pr[i]); nbp++; }
  }
```

```
public void setArrayOfPersons()
   {  String s="";
       for (int i =0; i< p.length; i++)
     { p[i].setName(input.next()+input.nextLine());
        p[i].setAge(input.nextInt());
        s=input.next();
        p[i].setGender(s.charAt(0));
     }
     nbp=p.length;
   }
public boolean insertPerson(Person p1)
{ if (nbp = = p.length) return false;
   p[nbp++] = p1; // p[nbp]  = p1; nbp++;
   return true;
}
 //--- Average of all ages -----
   public double averageOfAge( )
   {   double s=0.0;
        for(int i =0; i<=nbp-1; i++)
         s+=p[i].getAge();
      return (s/nbp);
   }
```

Person Array Processing – Sample 1

Create Person objects and set up the person array.

```
//---- Find the oldest  persons
public Person OldestPerson()
{
    Person old = p[0];
    for (int i =1; i<=nbp-1; i++)
       if (old.getAge() < p[i].getAge())
           old =p[i];
    return (old);
}
  //--- search  for a particular person ----
public boolean findPersonByName(String na)
{
     for (int i=0; i<nbp; i++)  {
       if (p[i].getName().equals(na)== true)
             return (true);

     }
   return (false);
}
//---- return index of a person if exist and -1 if not
public int findPerson(Person pr)
{
      for (int i=0; i<nbp; i++)  {
        if (p[i].getName().equals(pr.getName())== true)
           if (p[i].getAge() == pr.getAge())
             if (p[i].getGender()== pr.getGender())
               return (i);

        }
      return (-1);
}
```

```
public  boolean delete1Person(Person pr)
{ int x = findPerson(pr)
   if (x != -1)
   {
      p[x] = p[nbp – 1];
     p[--nbp] = null;
      return true;
    }
    return false;
}


 public  boolean delete2Person(Person pr)
 { int x = findPerson(pr)
   if (x != -1)
   {for (int i = x; i<nbp - 1; i++)
     p[i] = p[i + 1];
     p[--nbp] = null;
     return true;
    }
    return false;
 }
}
```

Chapter 2

Relationships between classes Using UML

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

Dr. S. HAMMAMI

Objectives: What is UML?

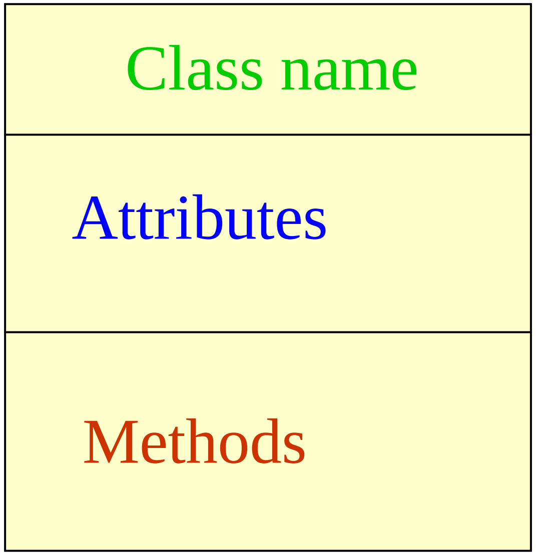
- *``UML is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct and document the artifacts of a software intensive system"*
- Defined semantics for each of the graphical symbols
- Allows for unambiguous specification and for inspection of requirements and designs
- Allows tools to directly generate code from diagrams - but programmers still has to do some work
- Provides documentation of products, so allowing auditing and facilitating management

OUTLINE

1. UML Object Models: Classes
2. Association
3. Composition
4. Aggregation
5. Examples

1. UML Object Models:

Classes



Class name

Attributes

Methods

Class Name

Should be descriptive of the class and capitalized in the first letter

Attributes

The named properties of the class. Can be typed, possibly with default values

Methods

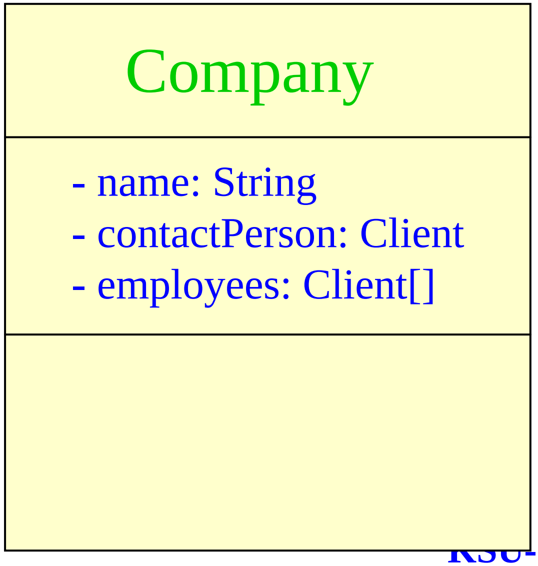
Services offered by the class. Methods can be typed e.g. parameter types and return types specified.

2. UML Object Models:
Association, Aggregation, and Composition

Associations

UML diagrams show a collection of named boxes - indicating classes or types of object. The boxes have lines connecting them called links. Each link is called an *association* and should model some relationship or connection between the classes. Associations also play roles in classes that are often given special names.

Example:
Classes can contain references to each other. The *Company* class has two attributes that reference the Client class.


Company

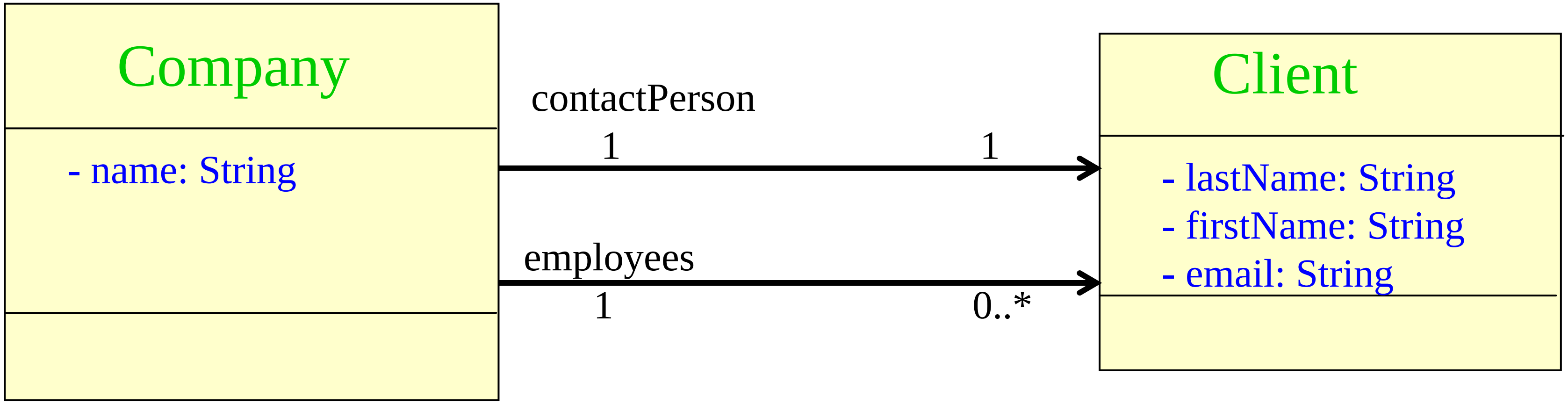
- name: String
- contactPerson: Client
- employees: Client[]

UML Object Models:

Association, Aggregation, and Composition

Associations

Although this is perfectly correct, it is sometimes more expressive to show the attributes as associations.



UML Object Models:

Association, Aggregation, and Composition

Associations

The above two associations have the same meaning as the attributes in the old version of the *Contact* class.

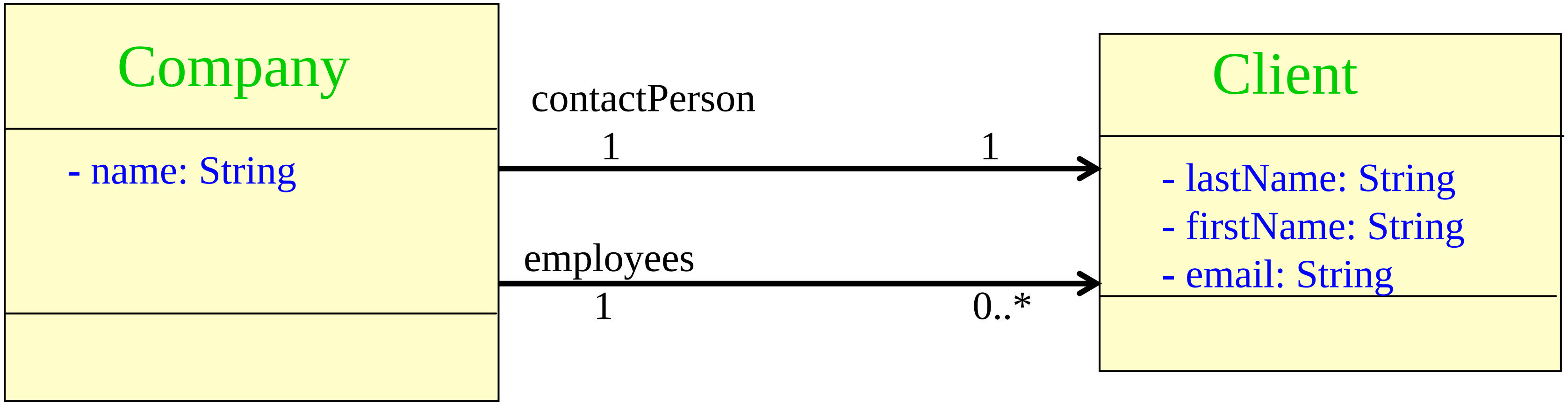
The first association (the top one) represents the old *contactPerson* attribute. There is one contact person in a single Company.

The ***multiplicity*** of the association is one to one meaning that for every *Companythere* is one and only one *contactPerson* and for each *contactPerson* there is one Company.

UML Object Models:

Association, Aggregation, and Composition

Associations



Company

- name: String

contactPerson

1

1

employees

1

0..*

Client

- lastName: String
- firstName: String
- email: String

The first association (the top one) represents the old *contactPerson* attribute. There is one contact person in a single Company.

The ***multiplicity*** of the association is one to one meaning that for every *Companythere* is one and only one *contactPerson* and for each *contactPerson* there is one Company.

In the bottom association there are zero or many employees for each company.

UML Object Models:

Association, Aggregation, and Composition

Associations

Multiplicities can be anything you specify. Some examples are shown:

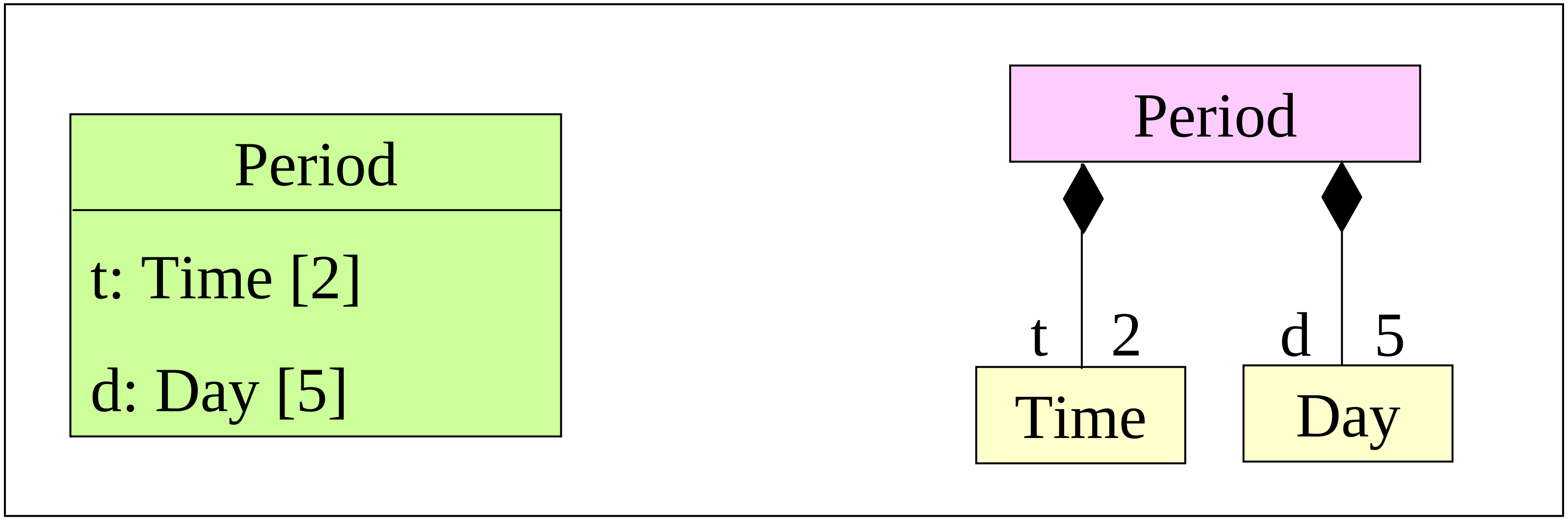
0	zero
1	one
1..*	one or many
1..2, 10..*	one, two or ten and above but **not** three through nine

3. UML Object Models: Aggregation, and Composition

Composition

UML provides several notations that can express the physical construction of a class. The filled in diamond is often used when a class contain other objects within them as parts or components. **The *composition* association is represented by the solid diamond.**

Here are two examples: **Period is *composed* of *Time and Day*.**



Period

t: Time [2]

d: Day [5]

Period

t 2

d 5

Time

Day

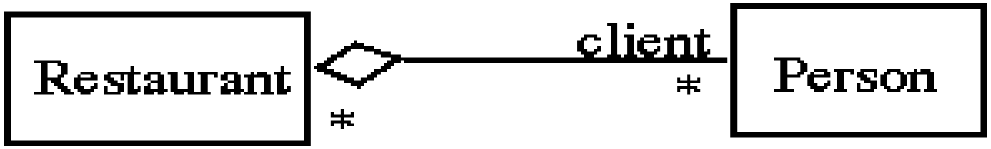
We can use the dark diamond to indicate that the class possesses the components in the sense of controlling whether they exist of not. The filled in diamond indicates that the deletion of an object may delete its components as well.

4. UML Object Models: Aggregation, and Composition

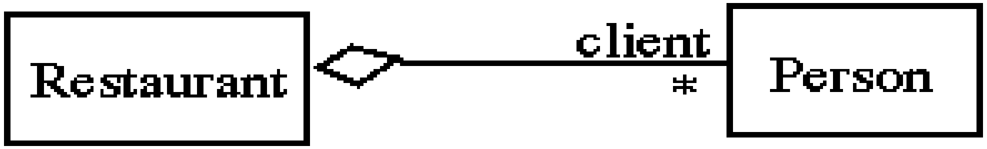
Aggregation

We can also show that a class *has* some parts and yet they have an independent existence. Example: In the computer world a page on the world Wide Web can use a hypertext reference to point to another resource -- deleting the page does not effect the other page. This association is called aggregation. **is represented by the hollow diamond.**

Here is an example showing that a Restaurant will have a number of clients who are People and the the clients exist whether or not they are clients of the Restaurant:



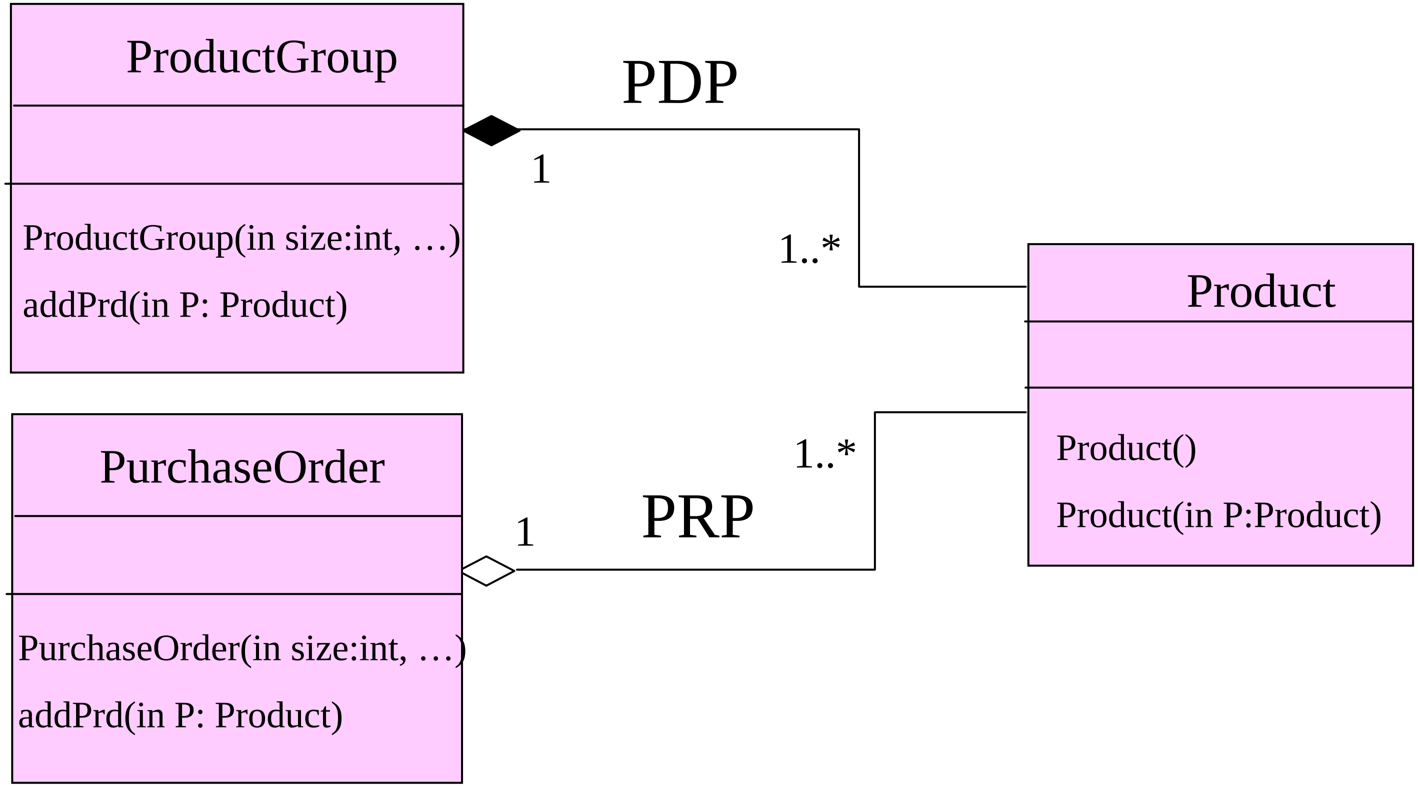
Each Person eats at any number of Restaurents



Each Person eats at one Restaurent.

UML Object Models: Aggregation, and Composition

Example 1:



ProductGroup

ProductGroup(in size:int, …)

addPrd(in P: Product)

PurchaseOrder

PurchaseOrder(in size:int, …)

addPrd(in P: Product)

Product

Product()

Product(in P:Product)

ProductGroup **is *composed* of *Products*. This means that if a *ProductGroup* is destroyed, the *Products* within the group are destroyed as well.**

PurchaseOrder **is an *aggregate* of *Products*. If a *PurchaseOrder* is destroyed, the *Products* still exist.**

If you have trouble remembering the difference between composition and aggregation, just think of the alphabet. Composition means destroy and the letters 'c' and 'd' are next to each other.

UML Object Models: Aggregation, and Composition

How to Implement Aggregation?

```
public class PurchaseOrder
{
  private Product PRP [];
  private int nprp; // number of current
                              product in the array.
  …..

  public PurchaseOrder (int size, …)
  {
     PRP = new Product[size];
     nprp=0;
     ….
   }

   ……
}
```

How to Implement Composition?

```
public class ProductGroup
{
  private Product PDP [];
  private int npdp; // number of current
                              product in the array.
  …..

  public ProductGroup (int size, …)
  {
     PDP = new Product[size];
     npdp=0;
     ….
   }

   ……
}
```

UML Object Models: Aggregation, and Composition

Aggregation: How to add a new product

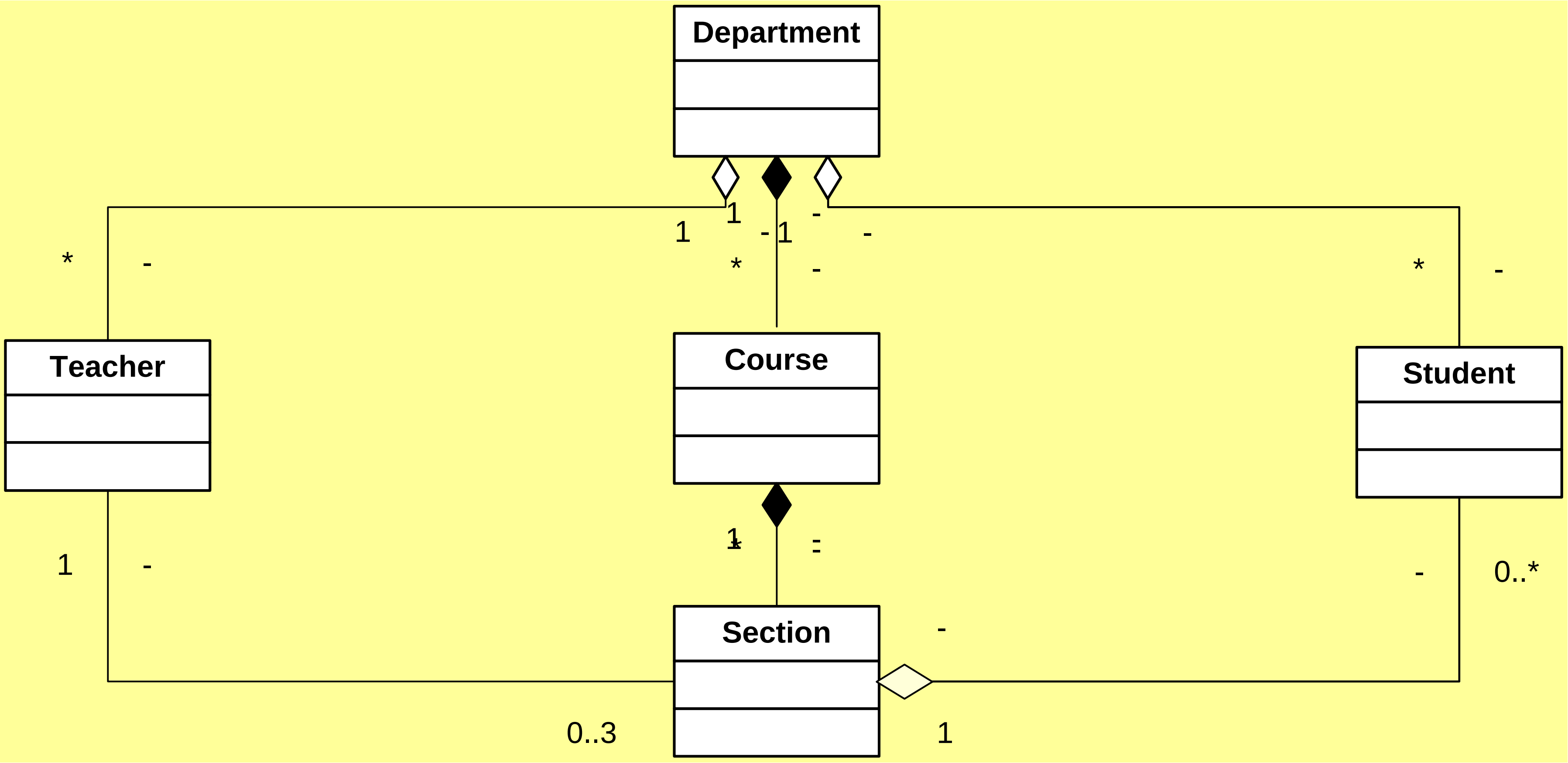
```
public class PurchaseOrder
{
  private Product PRP [];
  private int nprp; // number of current
                              product in the array.
  …..

  public void addPrd (Product P)
  {
     PRP[nprp] = P;
     nprp++;
   }

   ……
}
```

Composition: How to add a new product

```
public class ProductGroup
{
  private Product PDP [];
  private int npdp; // number of current
                              product in the array.
  …..

  public void addPrd (Product P)
  {
     PDP[npdp] = new Product(P);
     nprp++;
   }

   ……
}
```

UML Object Models: Aggregation, and Composition

Example 2:



UML Object Models: Aggregation, and Composition

Example 2:

The following Java code shows just how the links between the different objects can be implemented in Java. Note that this code just shows the links. It does not show constructors, or any other methods what would be required to actually use these objects.

```
/*
* Student.java -
*/
public class Student
{
  private String name;
  private String id;

  public void copyStudent(Student st)
  {
    name= st.name;
    id= st.id ;
  }
// ...
}
```

```
/*
* Section.java -
 */
public class Section
{
  private String sectionName;
  private int capacity;
  private int currentNbStudents;
  private Student[ ] stud;
 ….
  public void addStudent(Student s)
  {
    stud[currentNbStudents]=s;
    currentNbStudents ++;
  }
// ...
}
```

UML Object Models: Aggregation, and Composition

Example 2:

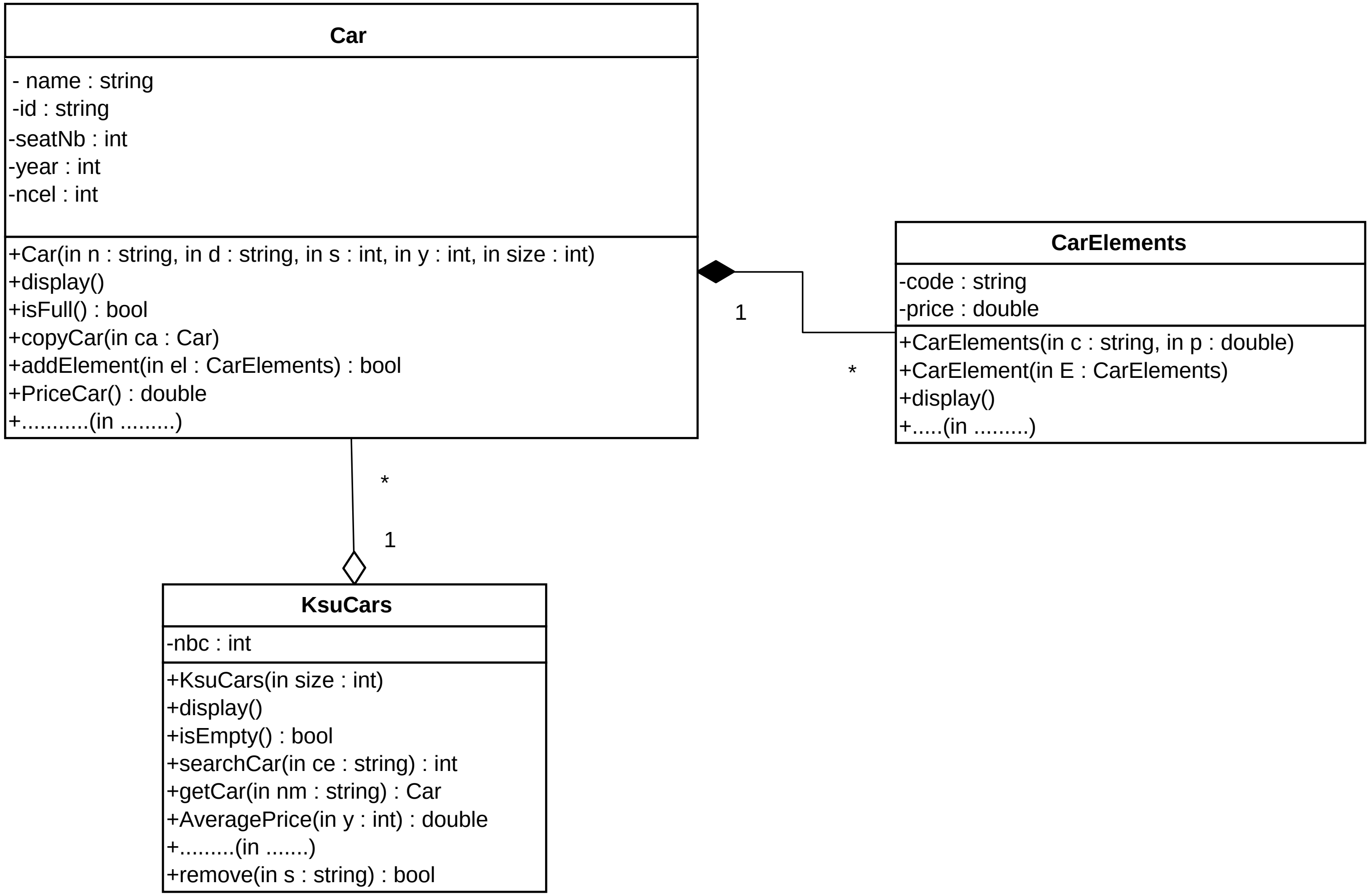
```
/*
* Course.java -

*/
public class Course
{
  private String
  courseName;
  private int nbSection;
  private Section[ ] sect;
// ...

}
```

```
/*
* Teacher.java -

*/
public class Teacher
{
  private String
teacherName;
  private String Id;
  private Section[3] sect;
// ...

}
```

```
/*
* Teacher.java -

*/
public class Department
{
  private String
departName;
  private Student[ ] stud;
  private Course[ ] csc;
  private Teacher[ ] teach;

// ...
}
```

UML Object Models: Aggregation, and Composition

Example 3:



Car

- name : string
-id : string
-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 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:

- name
- id
- 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.*

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

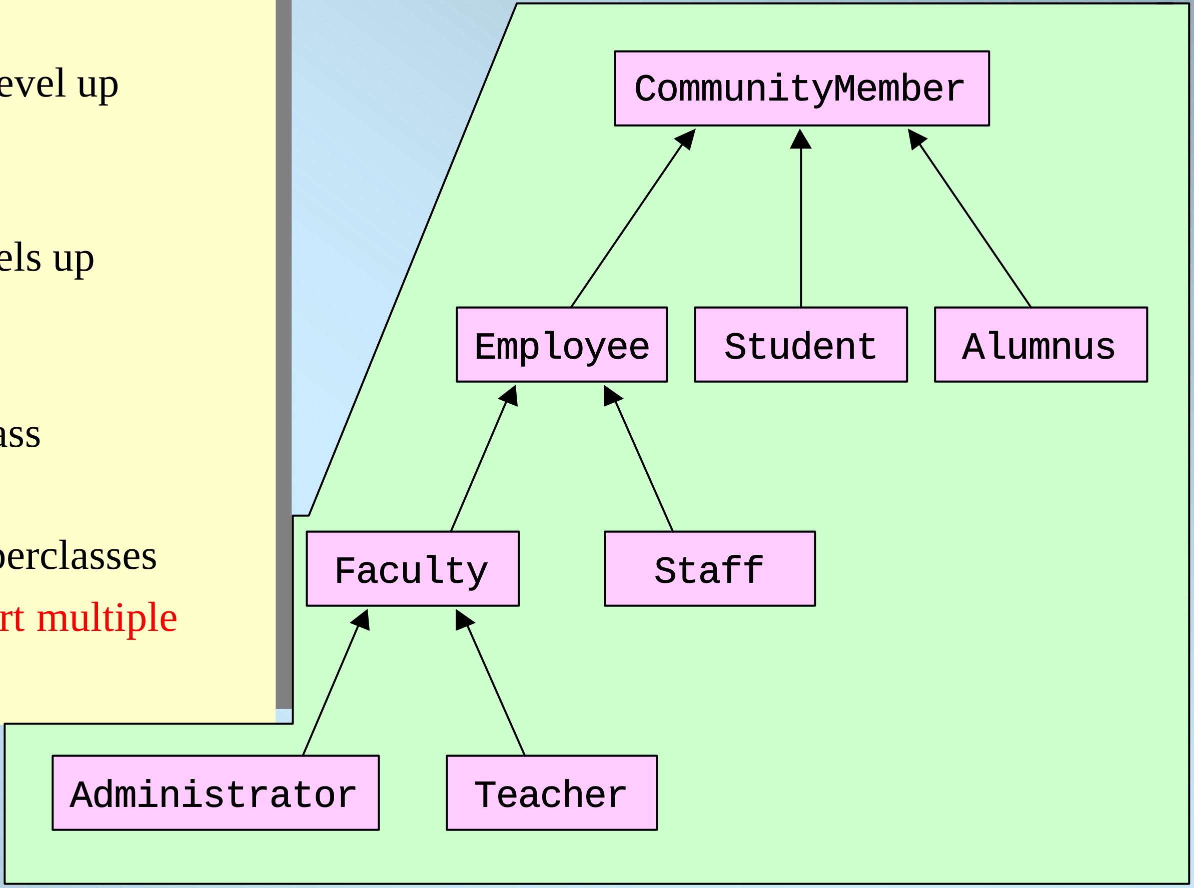
1. Introduction
2. Defining Classes with Inheritance
3. Inheritance and Member Accessibility
4. Inheritance Hierarchy
5. Declaring Subclasses
6. Inheritance and Constructors
7. Examples

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

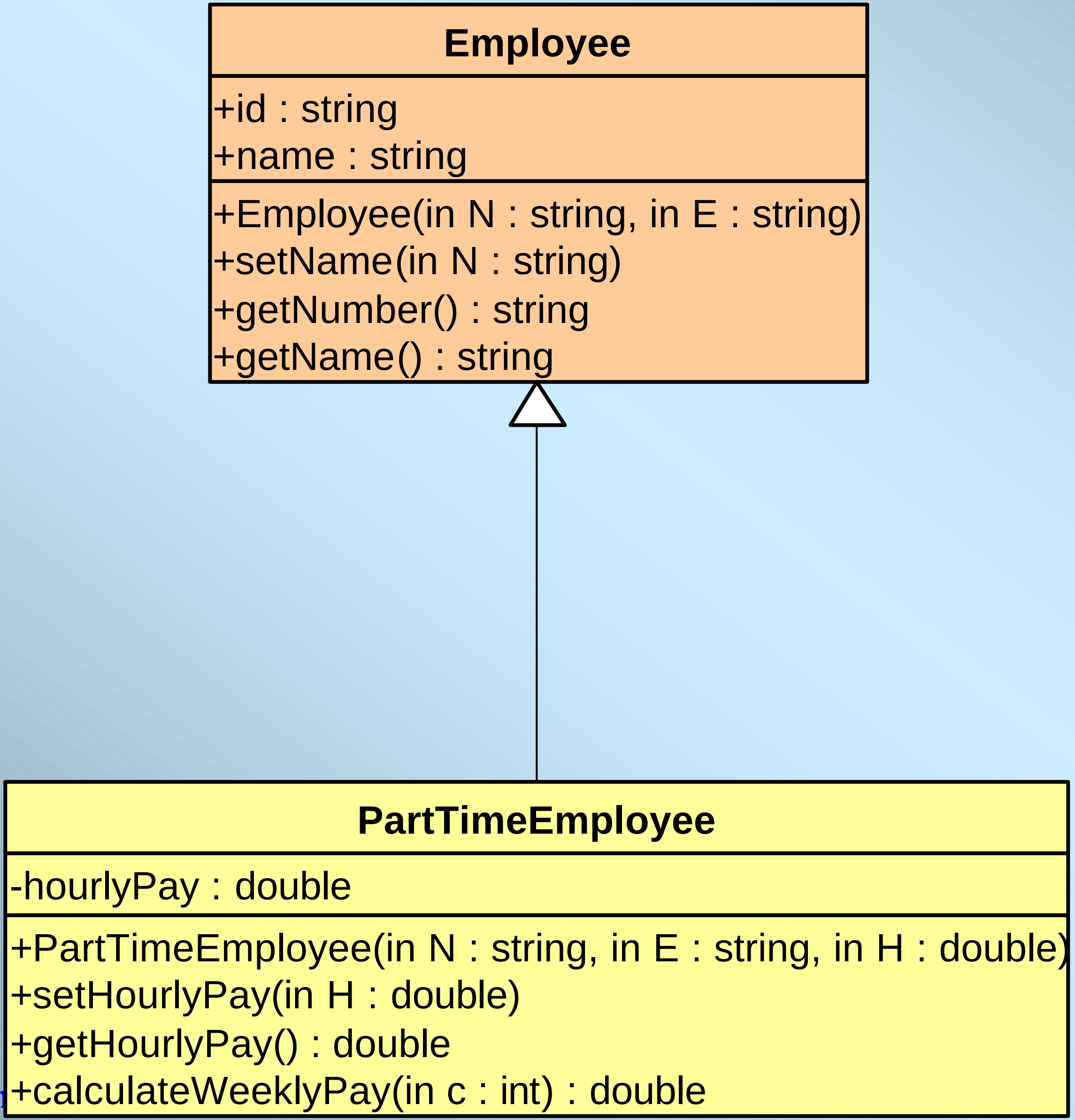
- **Case Study 1:**

- 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

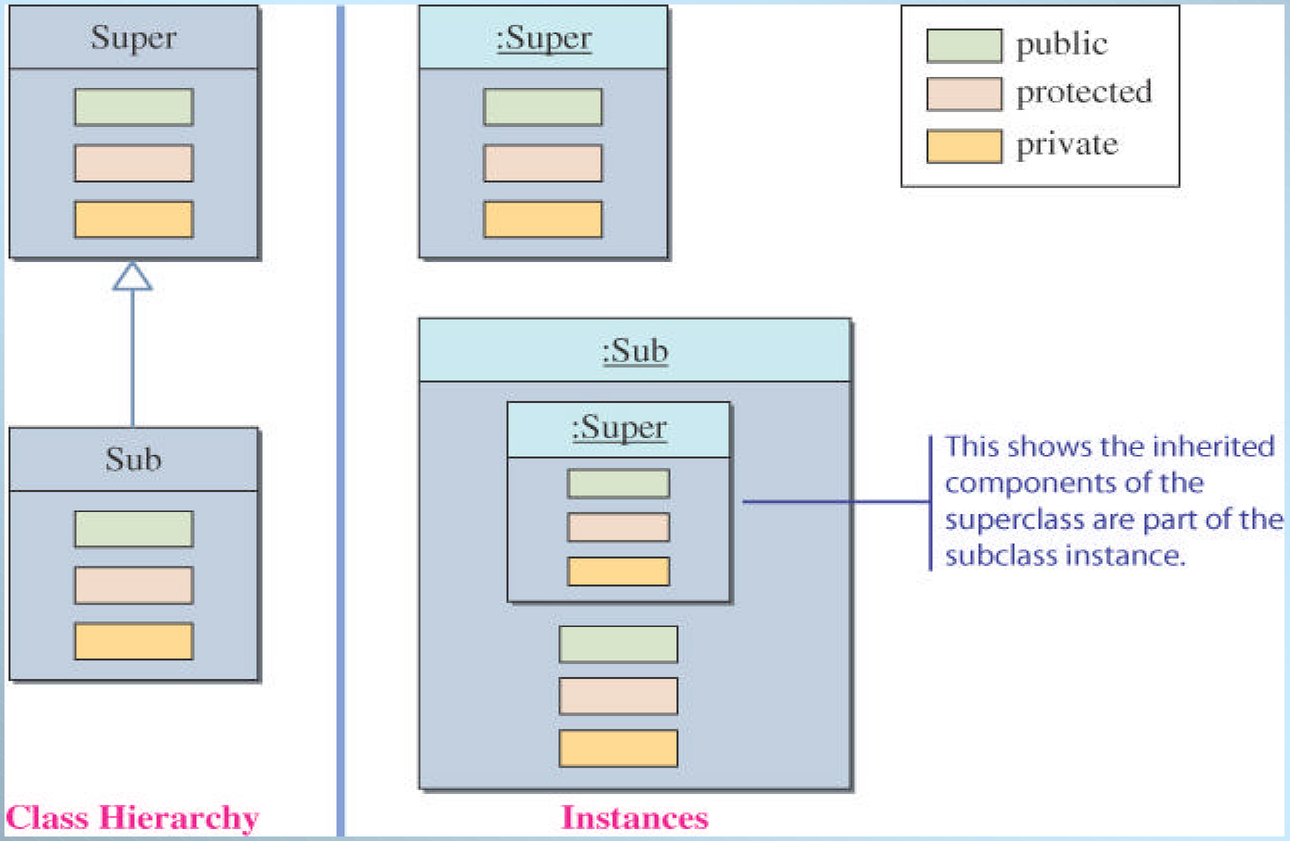
PartTimeEmployee

-hourlyPay : double

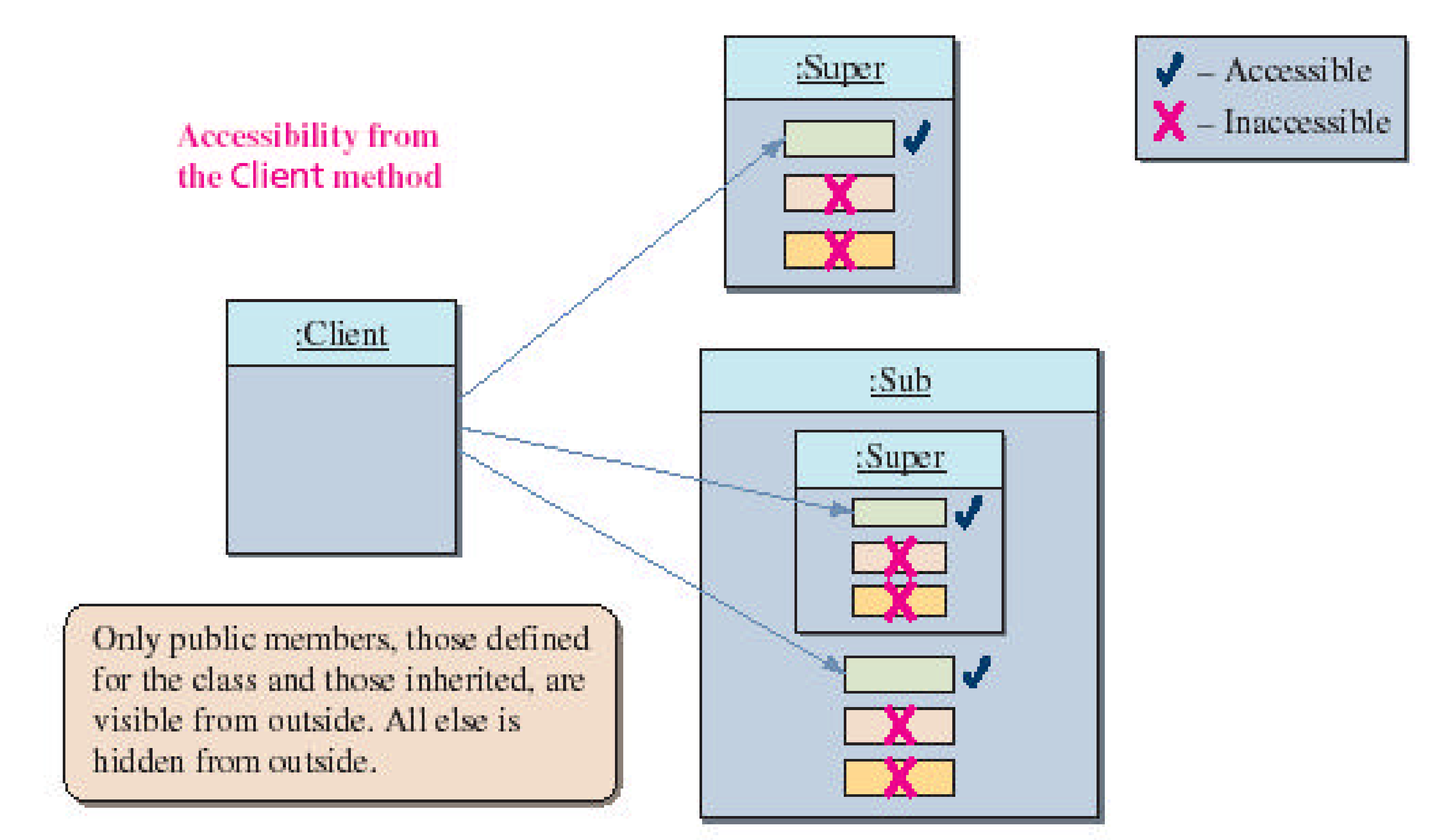
+PartTimeEmployee(in N : string, in E : string, in H : double)
+setHourlyPay(in H : double)
+getHourlyPay() : double
+calculateWeeklyPay(in c : int) : double

3. Inheritance and Member Accessibility

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


Super
Sub
:Super
:Sub
:Super
public
protected
private
This shows the inherited components of the superclass are part of the subclass instance.
Class Hierarchy
Instances

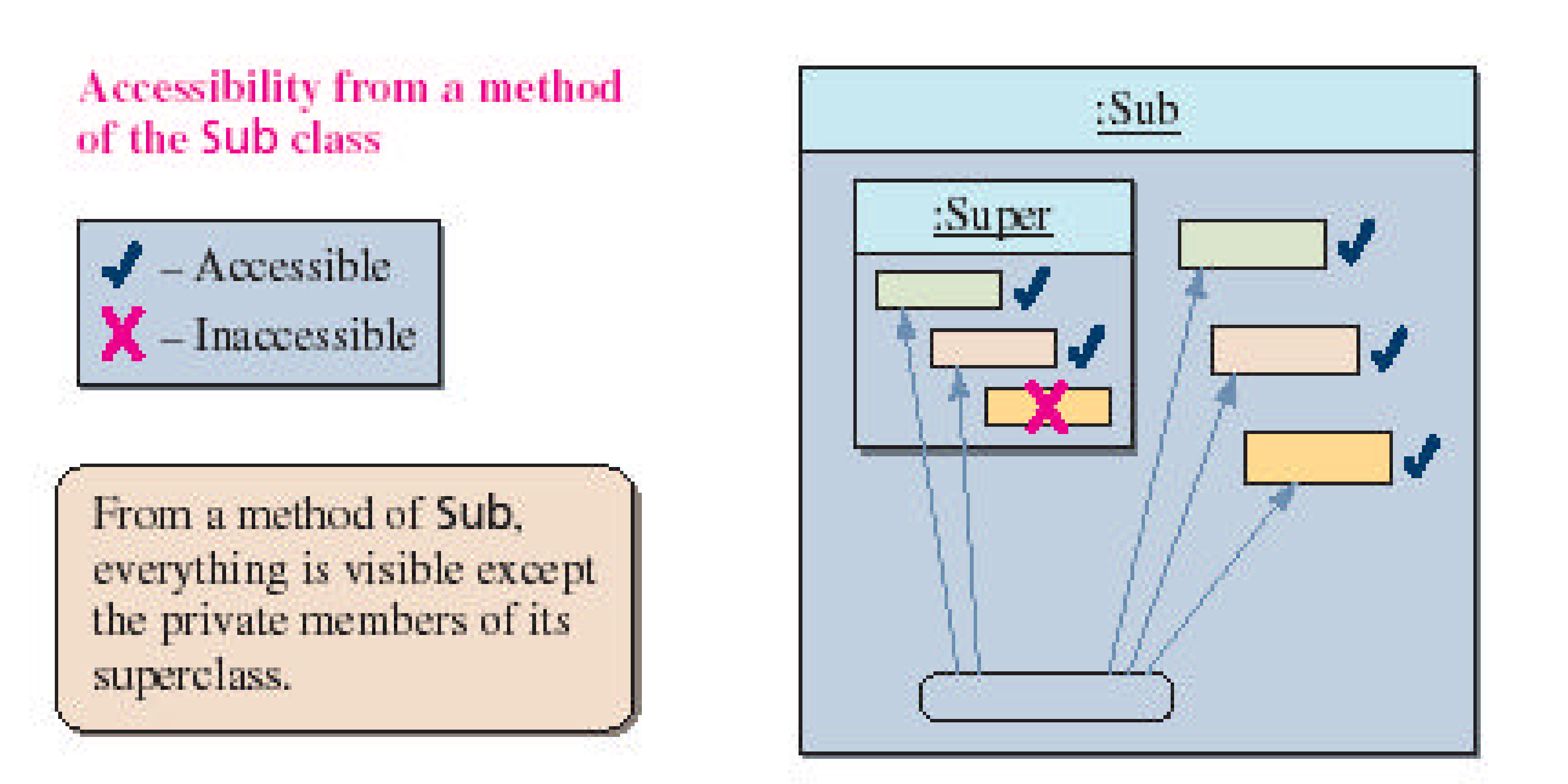
The Effect of Three Visibility Modifiers



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

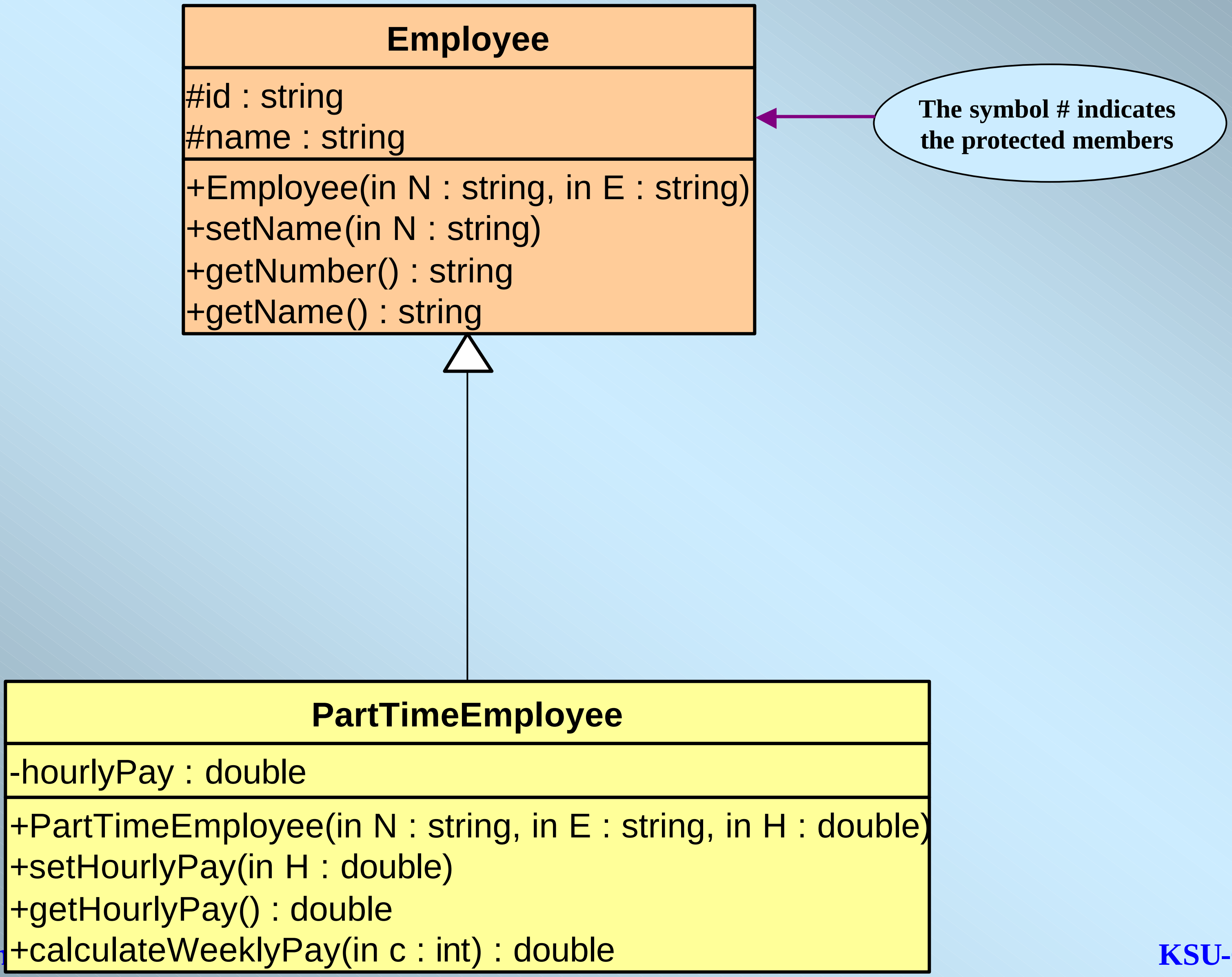
From a method of Sub, everything is visible except the private members of its superclass.

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.

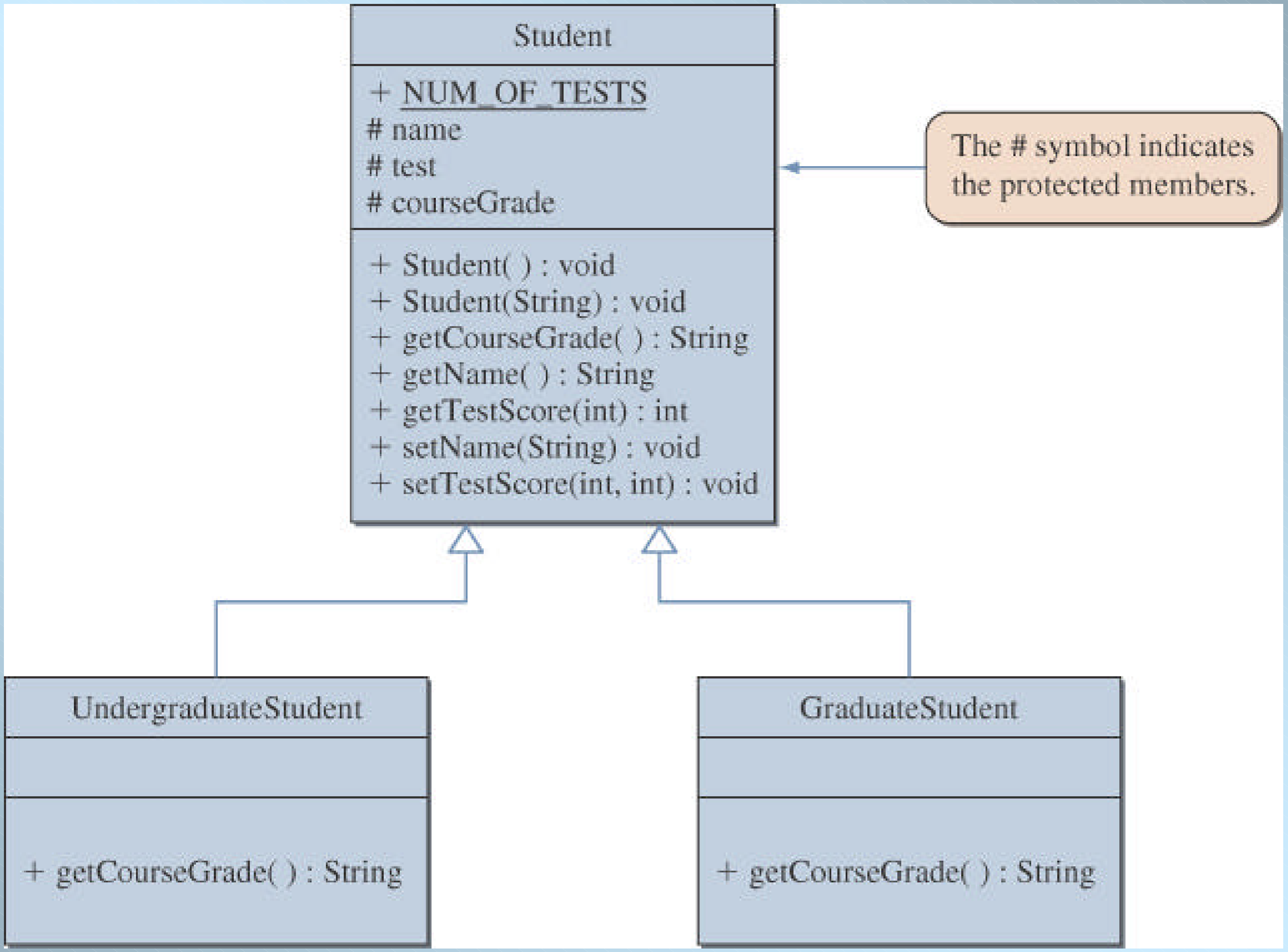
Modeling Two Types of Students

- 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.

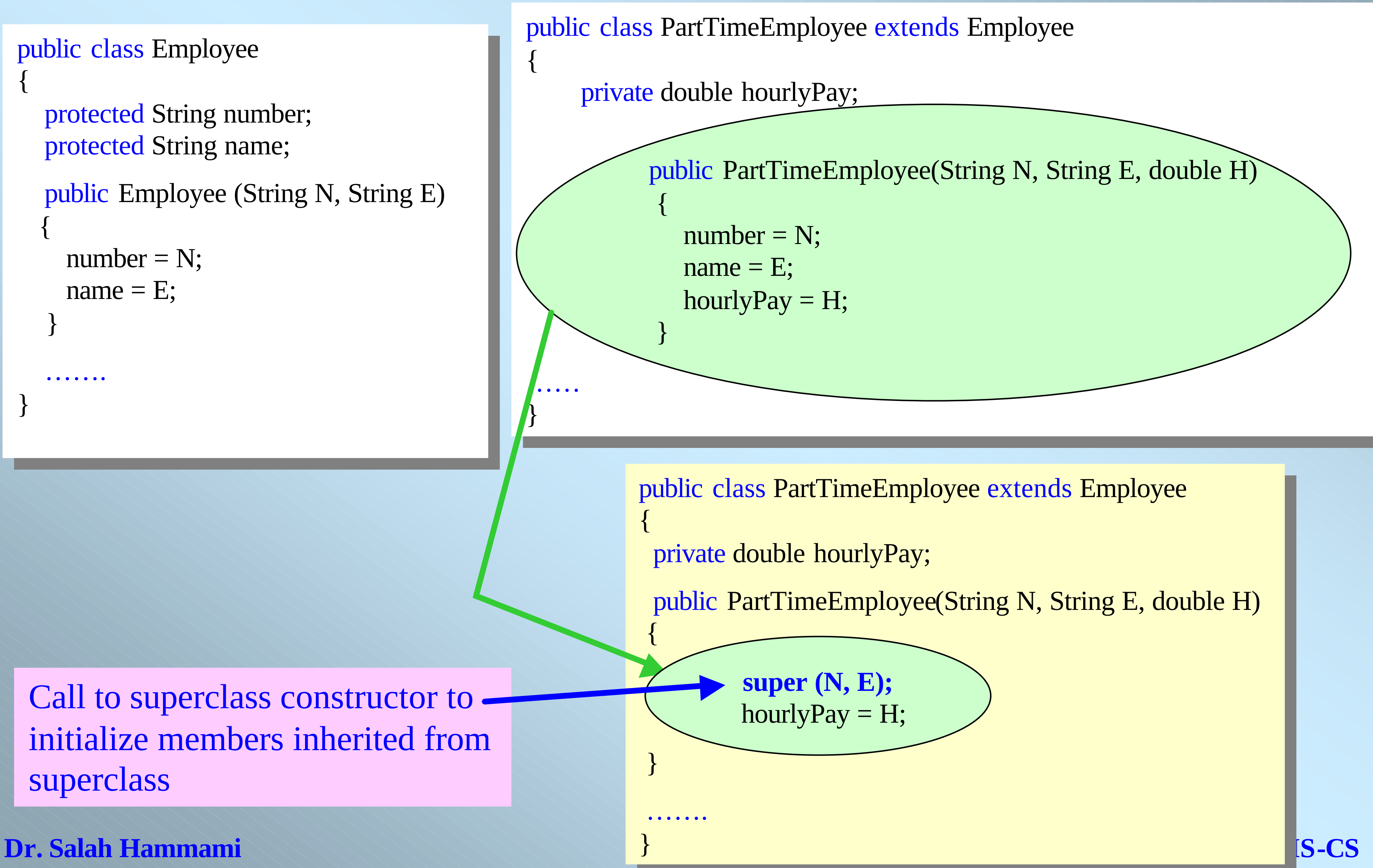
- A subclass uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```
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



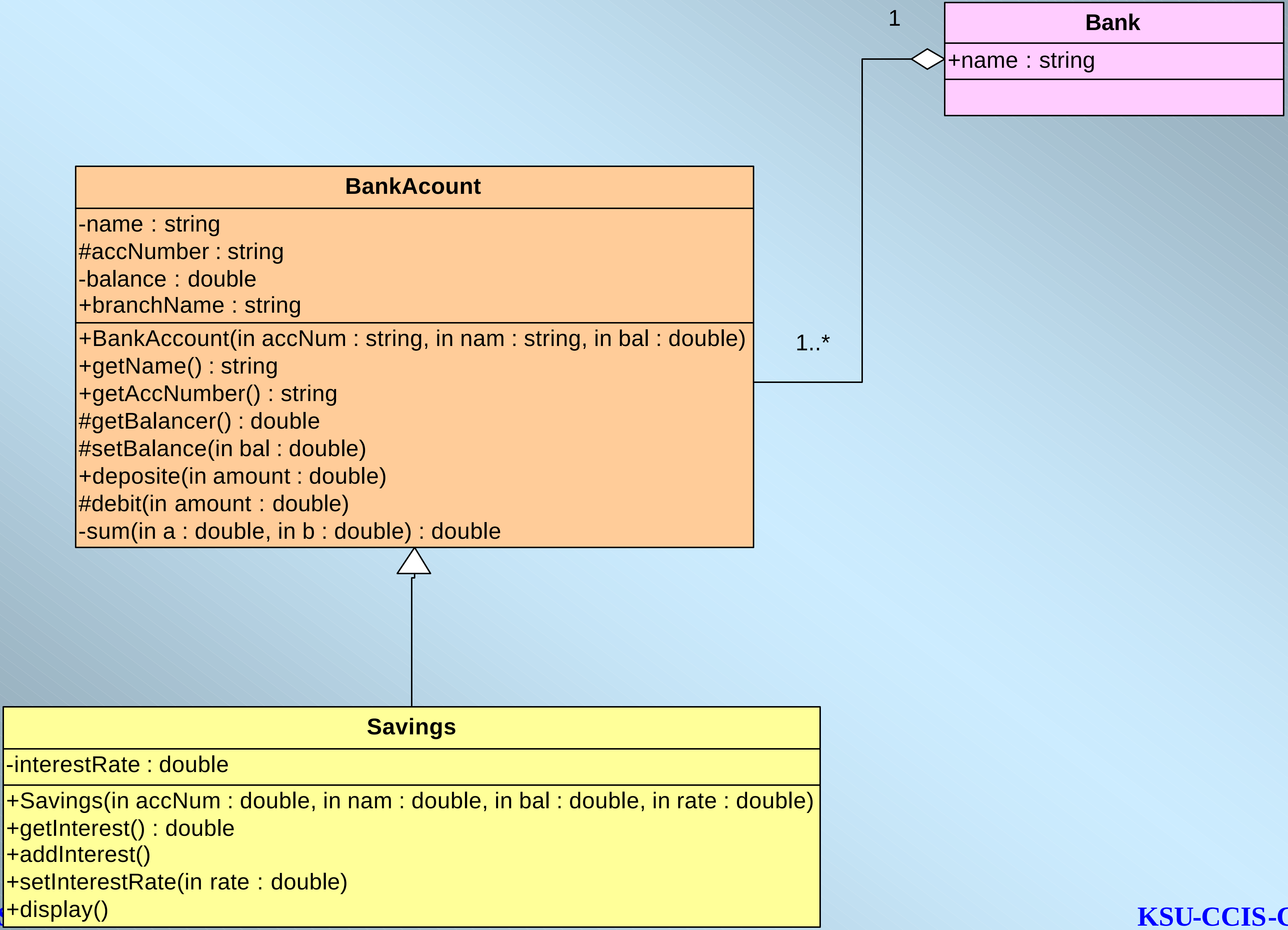
```
public class Employee
{
   protected String number;
   protected String name;

   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



1

Bank

+name : string

1..*

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;
  }                }
```

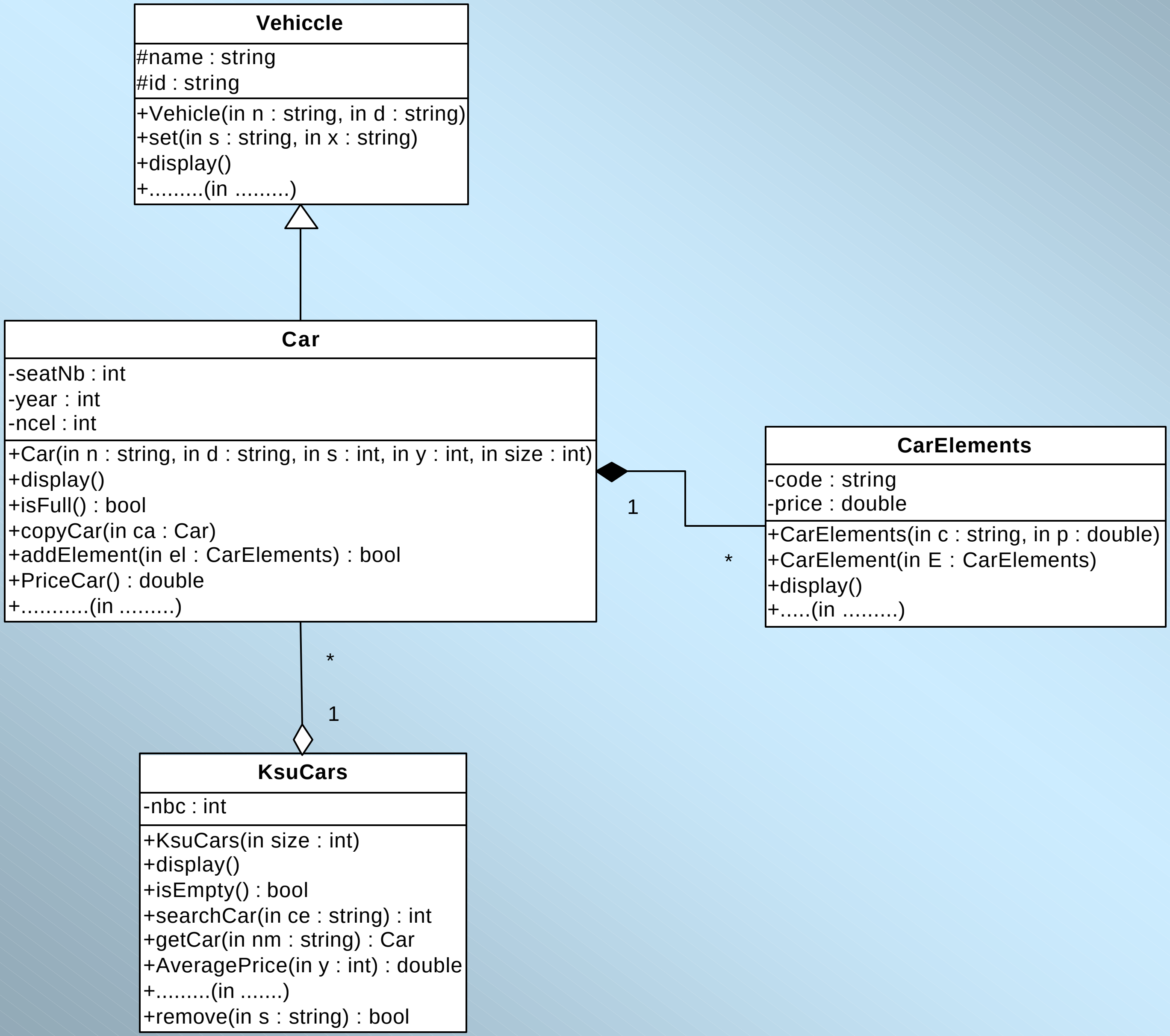
```
public class Savings extends BankAccount
{
private double interestRate;
public Savings(String number, double bal, String na,
String bankNa, double rate) {
                  super(number, bal, na, bankNa);
                  interestRate = rate;
                  }

 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.*

Chapter 4

Polymorphism

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

Dr. S. HAMMAMI

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.

Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism

- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

- A polymorphic method is one that has the same name for different classes of the same family, but has different implementations, or behavior, for the various classes.

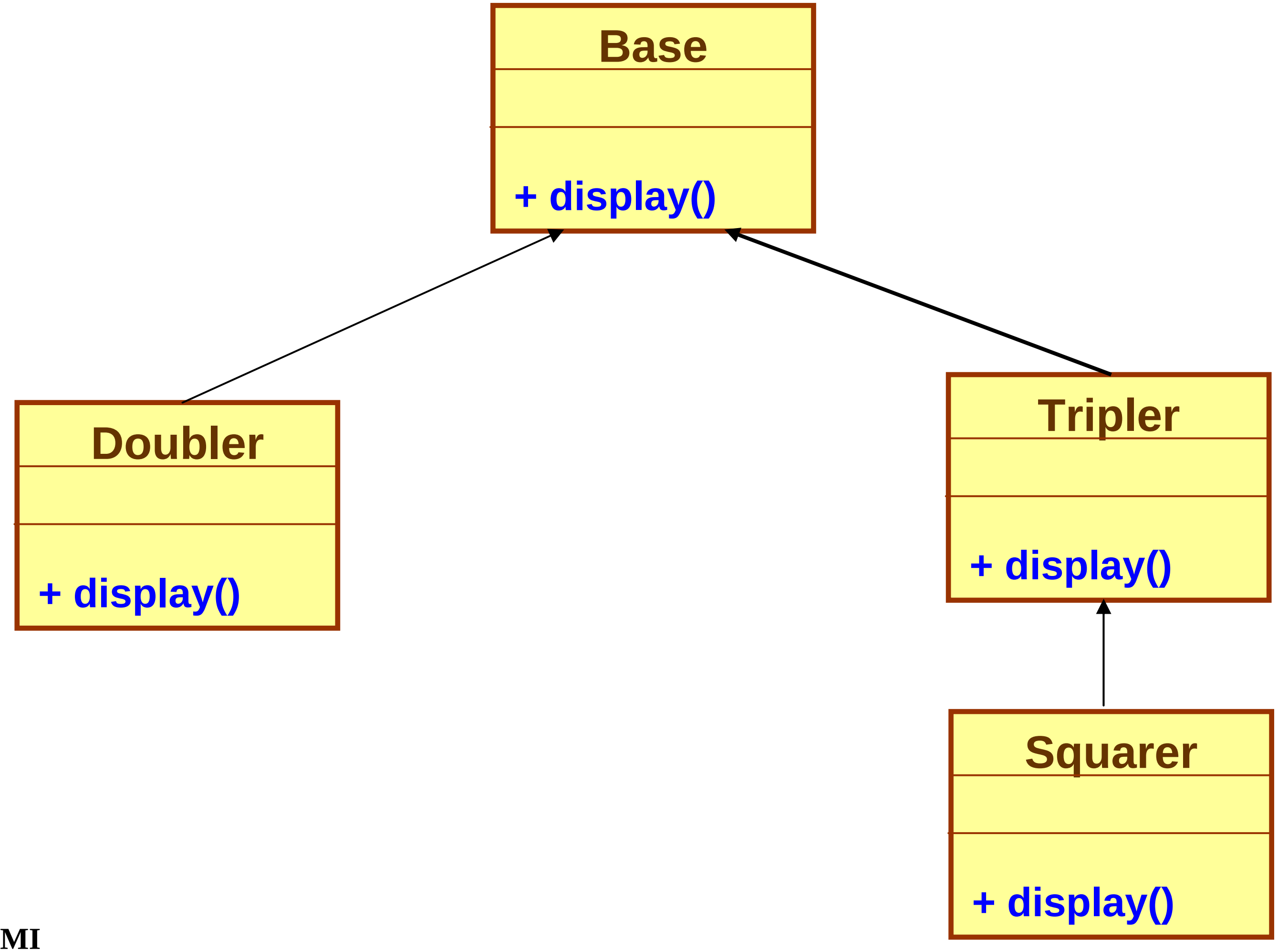
Introduction to Polymorphism

- **Polymorphism**
  - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
  - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked
  - Facilitates adding new classes to a system with minimal modifications to the system’s code

Example: Demonstrating Polymorphic Behavior

A polymorphic method (ex: display())

- A method that has multiple meanings
- Created when a subclass overrides a method of the superclass



Example: Demonstrating Polymorphic Behavior

```
public class Base {
   protected int i = 100;


   ...
   public void display() {   System.out.println( i );
   } }
```

```
public class Doubler extends Base {
   ...
   public void display() {System.out.println( i*2 );
   } }
```

```
public class Tripler extends Base {
   ...
   public void display() {
      System.out.println(i*3 );
   }  }
```

```
public class Squarer extends Tripler {
    ...
    public void display() {System.out.println( i*i );
    }  }
```

Example: Demonstrating Polymorphic Behavior

Case: Static binding

Some main program **output**

```
Base  B = new Base();
B. display();              →    100

Doubler D = new Doubler();
D. display();              →    200

Tripler T = new Tripler();
T. display();              →    300

Squarer S = new Squarer();
S. display();              →    10000
```

Static binding occurs when a method is defined with the same name but with different headers and implementations. The actual code for the method is attached, or bound, at compile time. Static binding is used to support overloaded methods in Java.

Example: Demonstrating Polymorphic Behavior
Case: Dynamic binding

- A superclass reference can be aimed at a subclass object
  - This is possible because a subclass object *is a* superclass object as well
  - When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called

Some main program

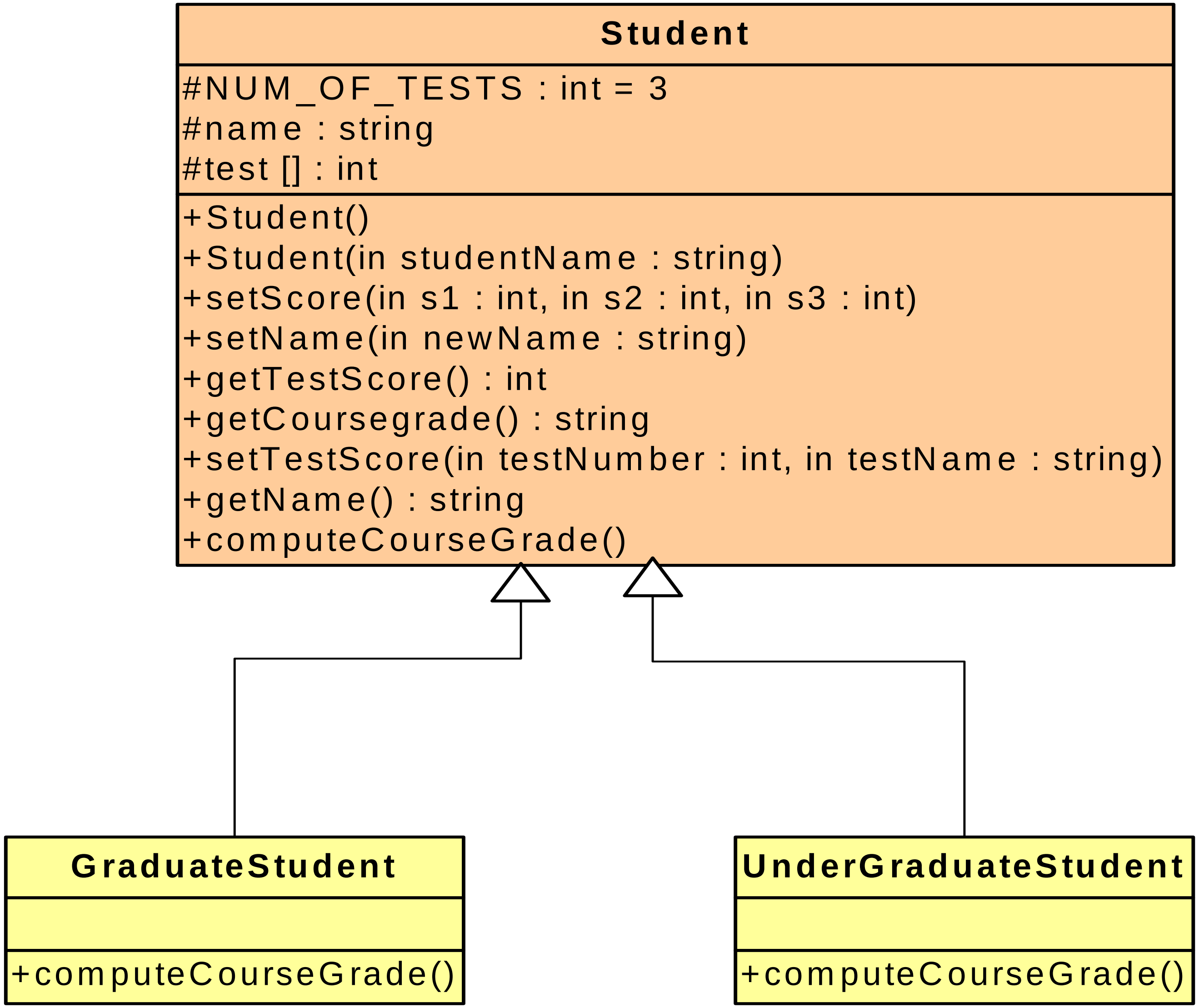
output

```
Base  B = new Base();
B. display();                →   100

Base D;
D = new Doubler();
D. display();                →   200


Base T;
T = new Tripler();
T. display();                →   300


Base S;
S = new Squarer();
S. display();                →   10000
```

Late binding or dynamic binding :

The appropriate version of a polymorphic method is decided at execution time

Example: Inheritance Hierarchy of Class Student : Polymorphism case



Student

#NUM_OF_TESTS : int = 3
#name : string
#test [] : int

+Student()
+Student(in studentName : string)
+setScore(in s1 : int, in s2 : int, in s3 : int)
+setName(in newName : string)
+getTestScore() : int
+getCoursegrade() : string
+setTestScore(in testNumber : int, in testName : string)
+getName() : string
+computeCourseGrade()

GraduateStudent

+computeCourseGrade()

UnderGraduateStudent

+computeCourseGrade()

Example: Inheritance Hierarchy of Class Student : Polymorphism case

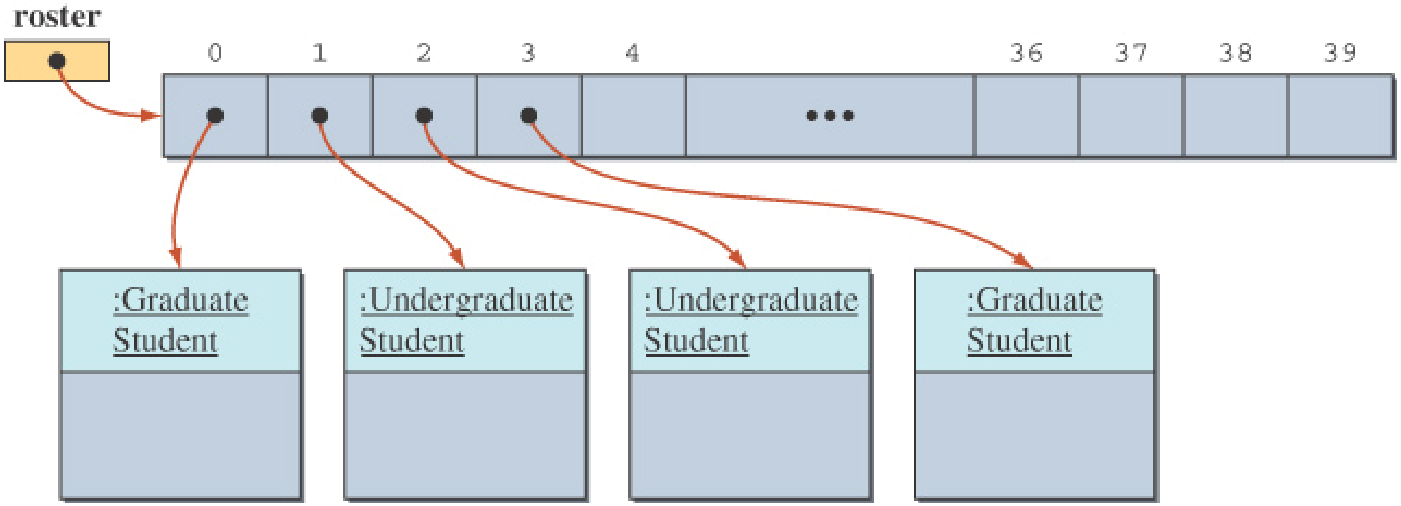
Creating the roster Array

- We mentioned in array definition that an array must contain elements of the same data type. For example, we can’t store integers and real numbers in the same array.
- To follow this rule, it seems necessary for us to declare two separate arrays, one for graduate and another for undergraduate students. This rule, however, does not apply when the array elements are objects using the polymorphism. We only need to declare a single array.
- We can create the roster 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.

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

Implementation *Student* in Java

Case Study :

```
class Student {
   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 void computeCourseGrade() { courseGrade="";}

  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;  }
 public void setTestScore(int testNumber, int testScore)
  {             test[testNumber-1]=testScore;  }
 }
```

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

Implementation *StudentTest* in Java

Case Study :

```
public class StudentTest  {
  public static void main(String[] args)
  {
      Student roster[]= new Student[2];
       roster[0] = new  GraduateStudent();
       roster[1] = new  UnderGraduateStudent();

       roster[0].setScore (20, 30, 50);
        roster[1].setScore (10, 17, 13);

       for (int i=0; i<roster.length; i++)
       {
         System.out.println("The name of the class is : " +  roster[i].getClass().getName());
         roster[i].computeCourseGrade();
          System.out.println(" Pass or Not  : " + roster[i].getCourseGrade());
       }
   }
 }
```

```
------ execution------------------

The name of the class is : GraduateStudent

 Pass or Not  : Pass

The name of the class is : UnderGraduateStudent

 Pass or Not  : No Pass
```

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

If roster[i] refers to a UnderGraduateStudent, then the computeCourseGrade method of the UnderGraduateStudent class is executed.

We call the message computeCourseGrade *polymorphic*

Implementation *StudentTest2* in Java

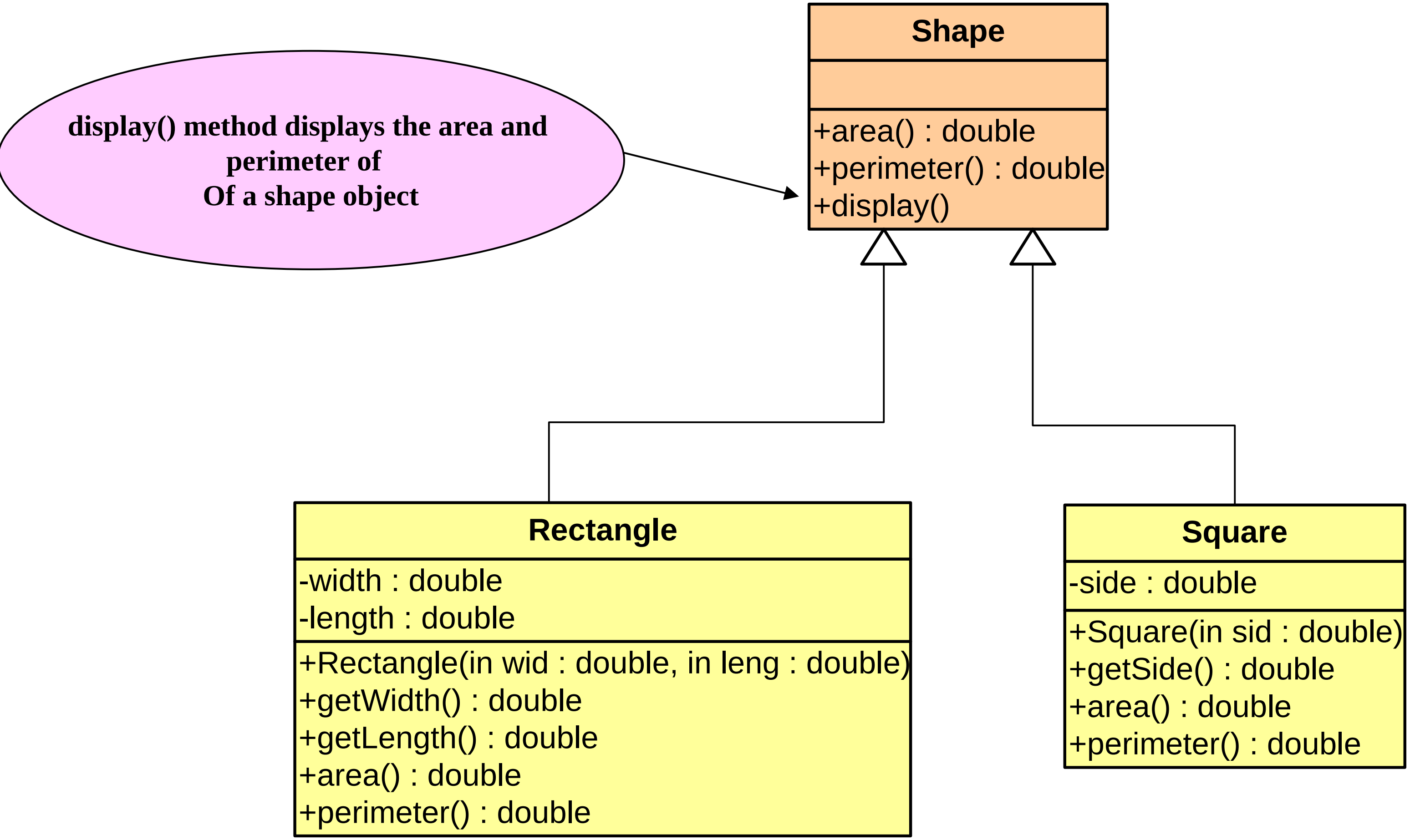
Case Study : **Question: Count the number of under graduate students**

```
public class StudentTest2  {
  public static void main(String[] args)
  {
      Student roster[]= new Student[2];
       roster[0] = new  GraduateStudent();
       roster[1] = new  UnderGraduateStudent();

       roster[0].setScore (20, 30, 50);
        roster[1].setScore (10, 17, 13);
       int nb=0;   //=== count the number of Under Graduate Students
       for (int i=0; i<roster.length; i++)
           if (roster[I] instanceof UnderGraduateStudent )
                  nb++;

          System.out.println("The number of Under Graduate Students : " + nb);
       }       }  }
```

```
------ execution-----------------

The number of Under Graduate Students : 1
```

Rule: To Determine the class of an object , we use the *instanceof* operator.

Example: Inheritance Hierarchy of Class Shape



display() method displays the area and perimeter of Of a shape object

Test: inheritance of Super-Class Shape

```
public class ShapeTest {
     public static void main(String[] args)
    {
            Shape shp =new Shape();  // shp is an object from class Shape
            Rectangle  rect =new Rectangle(4.0, 5.0);  // rect is an object from class Rectangle
            Square sqr = new Square(6.0);  // sqr is an object from class Square

            shp.display();  //--- uses the method display() from the class Shape
            rect.display();  //--- object rect inherits method display() from Superclass Shape
            sqr.display();  //--- object sqr inherits method display() from Superclass Shape
}
}
```

```
---- execution --------
The name of the class is : Shape
The area is :0.0
The perimeter is :0.0


The name of the class is : Rectangle
The area is :20.0
The perimeter is :13.0


The name of the class is : Square
The area is :36.0
The perimeter is :24.0
```

Implementation inheritance in Java

Case Study 3:
Shape

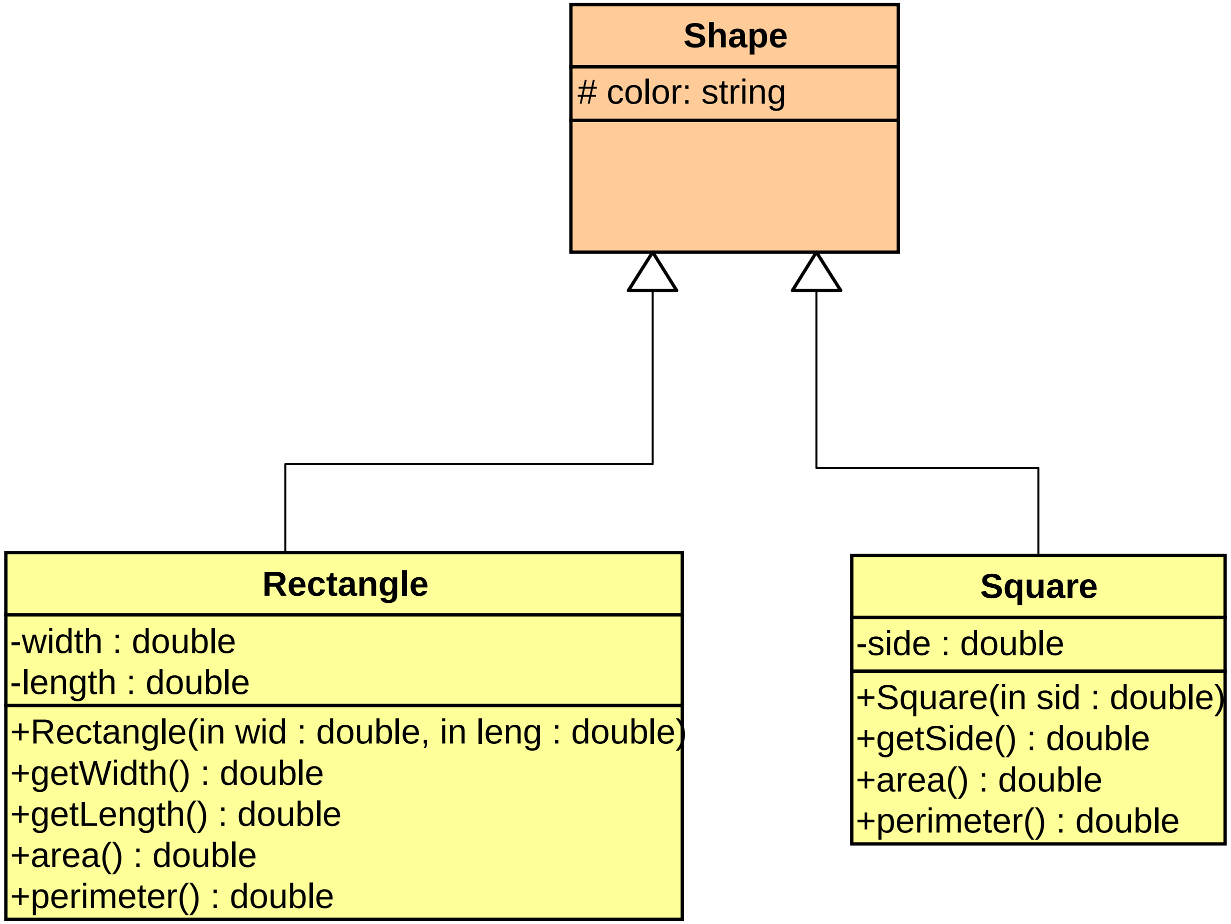
```
public class Shape  {
     public double area()    {  return 0.0;  }
     public double perimeter() { return 0.0;};
     public void display()  {
      System.out.println("The name of the class is : " +  this.getClass().getName());
       //--- getClass() a method inherits from the super class Object.
       //--- getName() a method from the class String.
       System.out.println("The area is :"+ area());
       System.out.println("The perimeter is :"+ perimeter()+"\n\n");}
}
```

```
public class Rectangle extends Shape {
   private double width;
   private double length;
   public Rectangle(double length, double width)
   {   this.length = length;    this.width = width;
    }
public double getLength() {return length; }
public double getWidth()  {return width; }
 public double area(){
   return (this.getLength()*this.getWidth());
  }
 public double perimeter(){
   return (2*this.getLength()+this.getWidth()); }}
```

```
public class Square extends Shape  {
    private double side;
   public Square(double side) {  this.side = side; }
   public double getSide() {         return side;          }
   public double area() {
         return (this.getSide()*this.getSide());
    }
    public double perimeter() {
                 return (4*this.getSide());

    }}
```

Introduction to Abstract Classes

In the following example, we want to add to the Shape class a **display** method that prints the area and perimeter of a shape.



Introduction to Abstract Classes

Abstract Method

- The following method is added to the **Shape** class

```
public void display()
{
  System.out.println (this.area());
  System.out.println (this.perimeter());

}
```

Introduction to Abstract Classes

Abstract Method

- There are several problems with this method:
  - The **`area`** and **`perimeter`** methods are invoked in the **`display`** method
  - There are **`area`** and **`perimeter`** methods in each of the subclasses
  - There is no **`area`** and **`perimeter`** methods in the **`Shape`** class, nor is there any way to define it reasonably without knowing whether the shape is Rectangle or Square.

Introduction to Abstract Classes

Abstract Class

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the subclasses
- **The class that contains an abstract method is called an *abstract class***

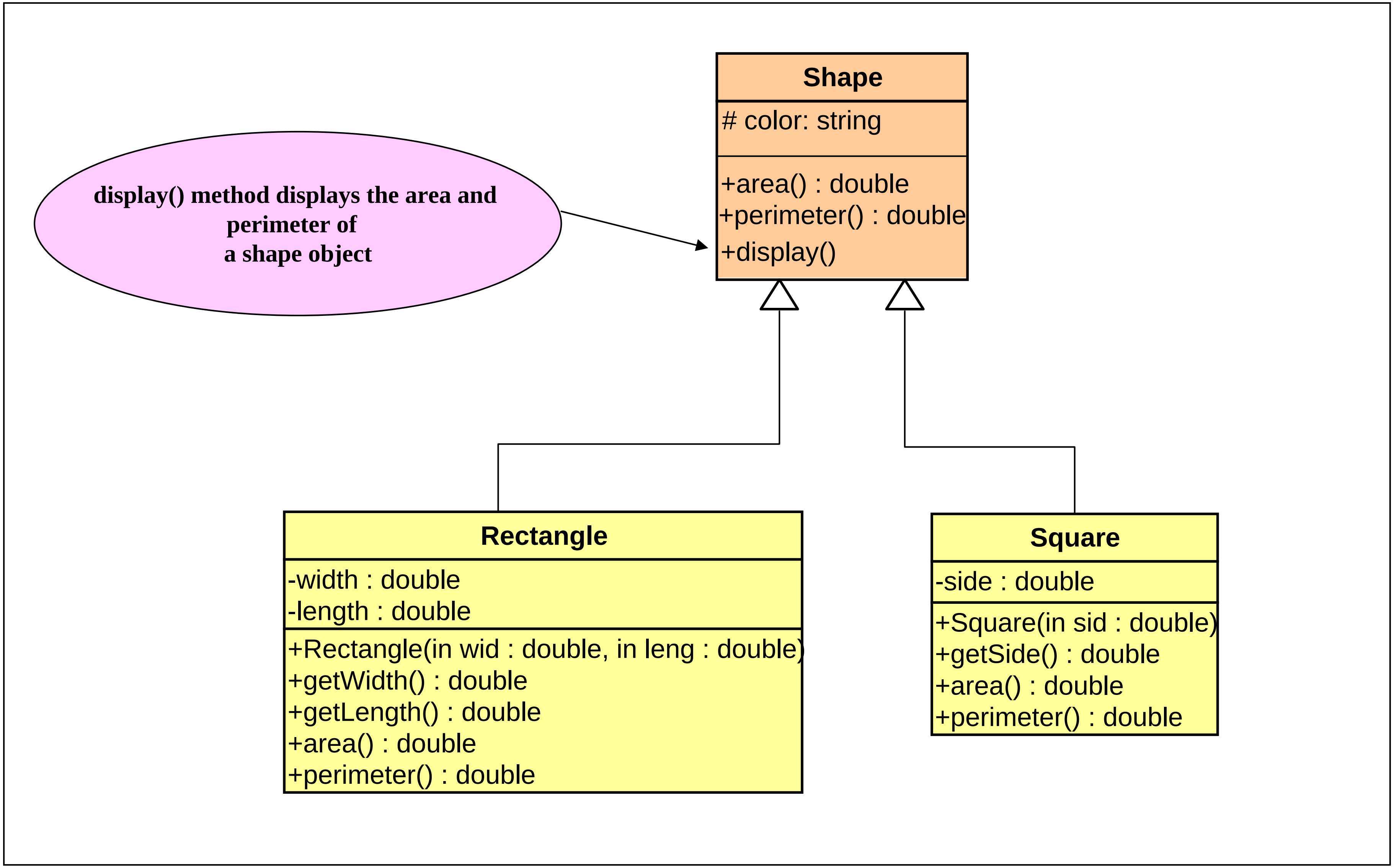
Introduction to Abstract Classes

Abstract Method

- An abstract method is like a **placeholder** for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **abstract**
- **It cannot be private**
- It has **no method body**, and ends with a semicolon in place of its body

```
public abstract double area();
public abstract double perimeter();
```

Introduction to Abstract Classes



display() method displays the area and perimeter of
a shape object

Introduction to Abstract Classes

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **abstract** included in its class heading:

```
public abstract class Shape
{
  protected String color;
  . . .
  public abstract double area();
  public abstract double perimeter();
  public void display()
  {
    System.out.println (this.area());
    System.out.println (this.perimeter());
  }
  . . .
}
```

Introduction to Abstract Classes

  - An abstract class can have any number of abstract and/or fully defined methods

  - If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **`abstract`** to its modifier

- A class that has no abstract methods is called a *concrete class*

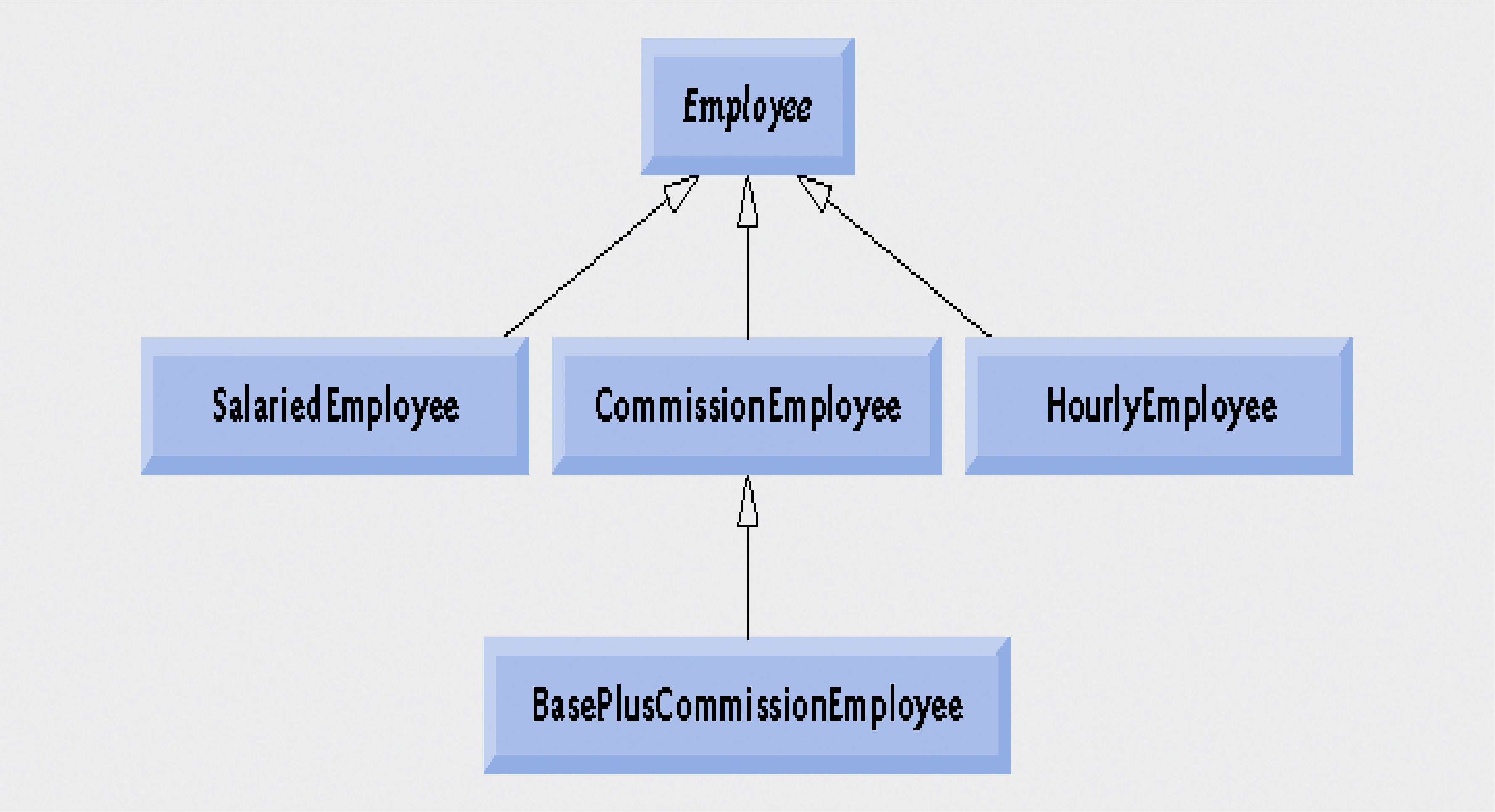
Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss shape in general, in reality a shape must be a rectangle form or a square form
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a subclass constructor will include an invocation of the abstract class constructor in the form of **`super`**

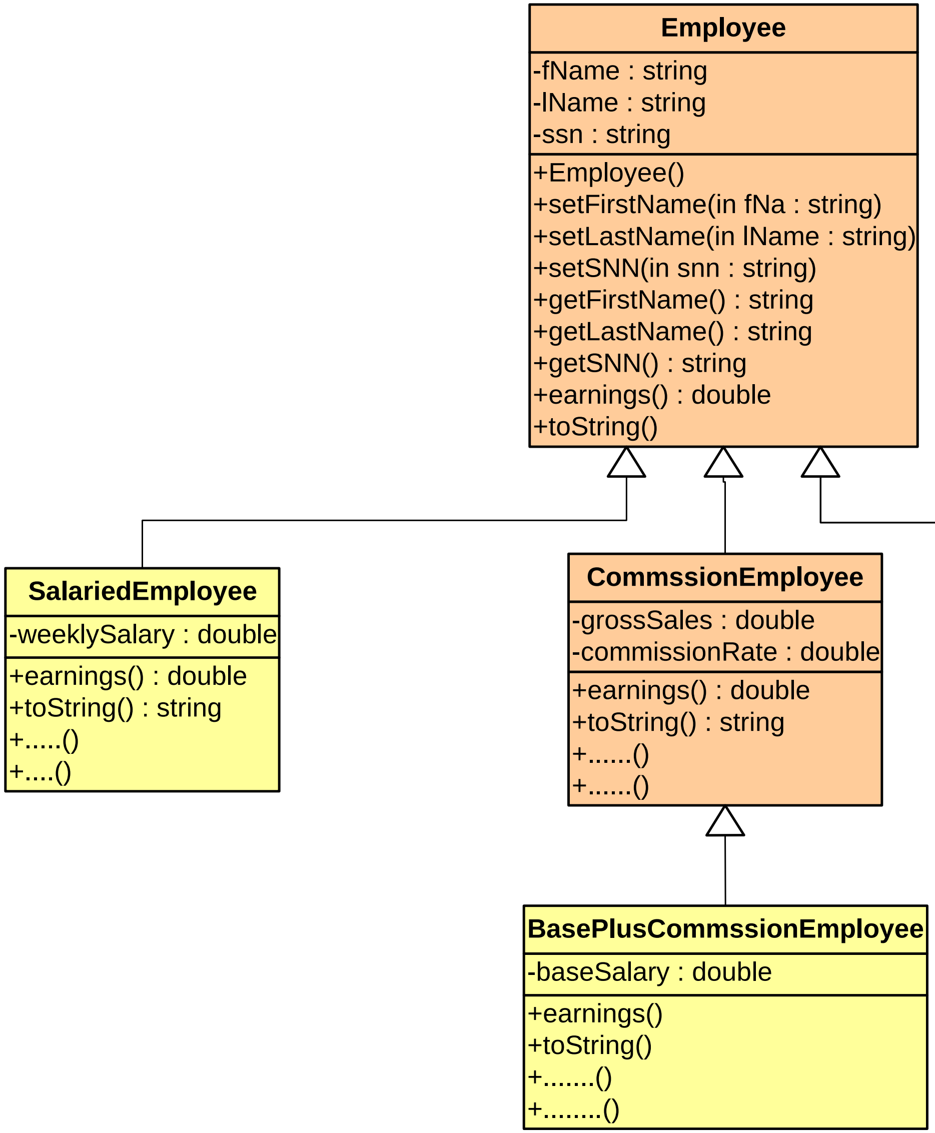
Dynamic Binding and Abstract Classes

- Controlling whether a subclass can override a superclass method
  - Field modifier `final`
    - Prevents a method from being overridden by a subclass
  - Field modifier `abstract`
    - Requires the subclass to override the method
- Early binding or static binding
  - The appropriate version of a method is decided at compilation time
  - Used by methods that are `final` or `static`

`Employee` hierarchy UML class diagram.



Example: Inheritance Hierarchy of Class Employee



Employee

-fName : string
-lName : string
-ssn : string

+Employee()
+setFirstName(in fNa : string)
+setLastName(in lName : string)
+setSNN(in snn : string)
+getFirstName() : string
+getLastName() : string
+getSNN() : string
+earnings() : double
+toString()

SalariedEmployee

-weeklySalary : double

+earnings() : double
+toString() : string
+.....()
+....()

CommssionEmployee

-grossSales : double
-commissionRate : double

+earnings() : double
+toString() : string
+......()
+......()

HourlyEmployee

-wage : double
-hours : double

+earnings() : double
+toString() : string
+......()
+....()

BasePlusCommssionEmployee

-baseSalary : double

+earnings()
+toString()
+.......()
+........()

Implementation *Employee* in Java

Case Study :

```
public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
public Employee( String first, String last, String ssn
)
   {    firstName = first;    lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   { return firstName;
   } // end method getFirstName
   // set last name
   public void setLastName( String last )
   {lastName = last;
   } // end method setLastName
```

```
// return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName
   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {  socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {  return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   public String toString()
   {return ("The name is :"+ getFirstName()+" "+
getLastName() + "\nThe Social Security Number: "+
getSocialSecurityNumber() );
   } // end method toString

   // abstract method overridden by subclasses
   public abstract double earnings(); // no
implementation here
} // end abstract class Employee
```

Implementation *SalariedEmployee* in Java

Case Study :

```
public class SalariedEmployee extends Employee
{
  private double weeklySalary;
  // four-argument constructor
  public SalariedEmployee( String first, String last, String
ssn,   double salary )
  {
     //super( first, last, ssn ) code reuse
    super( first, last, ssn ); // pass to Employee constructor
    setWeeklySalary( salary ); // validate and store salary
  } // end four-argument SalariedEmployee constructor

  // set salary
  public void setWeeklySalary( double salary )
  {
    weeklySalary = salary < 0.0 ? 0.0 : salary;
// this mean that, if salary is <0 then put it 0 else put it salary
  } // end method setWeeklySalary

  // return salary
  public double getWeeklySalary()
  {     return weeklySalary;
  } // end method getWeeklySalary
```

```
// calculate earnings; override abstract method earnings
in Employee
  public double earnings()
  { return getWeeklySalary();
  } // end method earnings

  // return String representation of SalariedEmployee
object
  //  this method override toString() of superclass method
  public String toString()
  {     //***** super.toString() : code reuse (good
example)
      return ( super.toString()+ "\nearnings = " +
getWeeklySalary());
         } // end method toString
} // end class SalariedEmployee
```

Implementation *HourlyEmployee* in Java

Case Study :

```
public class HourlyEmployee extends Employee
{
   private double wage; // wage per hour
   private double hours; // hours worked for week
   // five-argument constructor
   public HourlyEmployee( String first, String last, String
ssn,    double hourlyWage, double hoursWorked )
   {
       //  super( first, last, ssn ) code (constructor) reuse
      super( first, last, ssn );
  setWage( hourlyWage ); // validate and store hourly wage
setHours( hoursWorked ); // validate and store hours
worked
   } // end five-argument HourlyEmployee constructor
   public void setWage( double hourlyWage )
   {    wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
   } // end method setWage
   public double getWage()
   { return wage;
   } // end method getWage
   public void setHours( double hoursWorked )
   {hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <=
168.0 ) ) ?   hoursWorked : 0.0;
   } // end method setHours
   public double getHours()
   {      return hours;
   } // end method getHours
```

```
   // calculate earnings; override abstract method earnings
in Employee
   public double earnings()
   {
      if ( getHours() <= 40 ) // no overtime
         return getWage() * getHours();
      else
         return 40 * getWage() + ( getHours() - 40 ) *
getWage() * 1.5;
   } // end method earnings

   // return String representation of HourlyEmployee
object
   public String toString()  /* here overriding the
toString() superclass method */
   {   /*code reuse using super.   */
       return (super.toString() + "\nHourly wage: " +
getWage() +
       "\nHours worked :"+ getHours()+ "\nSalary is :
"+earnings() );
   } // end method toString
} // end class HourlyEmployee
```

Implementation *ComissionEmployee* in Java

Case Study :

```
public class CommissionEmployee extends Employee
{
  private double grossSales; // gross weekly sales
  private double commissionRate; // commission
percentage

  // five-argument constructor
  public CommissionEmployee( String first, String last,
String ssn,   double sales, double rate )
  {
     super( first, last, ssn );
     setGrossSales( sales );  setCommissionRate( rate );
  } // end five-argument CommissionEmployee constructor

  // set commission rate
  public void setCommissionRate( double rate )
  { commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate :
0.0;
  } // end method setCommissionRate

  // return commission rate
  public double getCommissionRate()
  {return commissionRate;
  } // end method getCommissionRate
```

```
// set gross sales amount
  public void setGrossSales( double sales )
  { grossSales = ( sales < 0.0 ) ? 0.0 : sales;
  } // end method setGrossSales

  // return gross sales amount
  public double getGrossSales()
  { return grossSales;
  } // end method getGrossSales

  // calculate earnings; override abstract method earnings
in Employee
  public double earnings()
  { return getCommissionRate() * getGrossSales();
  } // end method earnings

  // return String representation of
CommissionEmployee object
  public String toString()
  { return (super.toString() + "\nGross sales: " +
getGrossSales() +  "\nCommission rate: " +
getCommissionRate() + "\nearnings = " + earnings() );
  } // end method toString
} // end class CommissionEmployee
```

Implementation *BasePlusComissionEmployee* in Java

Case Study :

```
public class BasePlusCommissionEmployee extends
CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first,
String last,  String ssn, double sales, double rate, double
salary ) {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee
constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-
negative
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary
```

```
// calculate earnings; override method earnings in
CommissionEmployee
   public double earnings()
   {
      return getBaseSalary() + super.earnings(); //code
reuse form CommissionEmployee
   } // end method earnings

   // return String representation of
BasePlusCommissionEmployee object
   public String toString()
   {
      return ( "\nBase-salaried :" + super.toString() +
      "\nBase salary: " + getBaseSalary() + "\nearnings ="
+ earnings() );
   }// end method toString
} // end class BasePlusCommissionEmployee
```

Implementation PayrollSystemTest in Java

```
public class PayrollSystemTest
{
   public static void main( String args[] )
   {
      // create subclass objects
  SalariedEmployee SA=  new SalariedEmployee( “Ali", "Samer", "111-11-1111", 800.00 );
  HourlyEmployee HE =   new HourlyEmployee( “Ramzi", “Ibrahim", "222-22-2222", 16.75, 40 );
  CommissionEmployee CE =  new CommissionEmployee(  “Med", “Ahmed", "333-33-3333", 10000, .06 );
 BasePlusCommissionEmployee BP = new BasePlusCommissionEmployee(“Beji", “Lotfi", "444-44-4444", 5000, .04, 300 );

  System.out.println( "Employees processed individually:\n" );
      /*  salariedEmployee is the same as salariedEmployee.toString()  */
  System.out.println( SA.toString()+ "\nearned: " + SA.earnings()+"\n\n" );
  System.out.println( HE + "\n earned: " + HE.earnings()+"\n\n" );
  System.out.println(CE + "\n earned: " + CE.earnings()+"\n\n" );
  System.out.println(BP + "\n earned: "+ BP.earnings()+"\n\n" );
// create four-element Employee array
  Employee employees[] = new Employee[ 4 ];
  employees[ 0 ] = SA;   employees[ 1 ] = HE;  employees[ 2 ] = CE;   employees[ 3 ] = BP;
  System.out.println( "Employees processed polymorphically:\n" );
// generically process each element in array employees
  for (Employee currentEmployee : employees)        {
    System.out.println( currentEmployee );} // invokes toString  : here is polymorphysim : call toString() of class at the executiontime.
                                             //  called dynamic binding or late binding
                                             // Note :only methods of superclass  can be called via superclass variable
 // get type name of each object in employees array
  for ( int j = 0; j < employees.length; j++ )
    System.out.printf( "Employee %d is a %s\n", j,   employees[ j ].getClass().getName() );  // display the name of the class whos
                                                                                              //object is employee[j]
   } // end main     } // end class PayrollSystemTest
```

Chapter 5

Interface

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

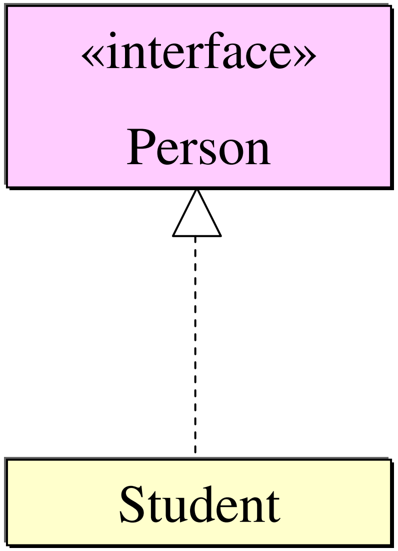
Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that implements the interface*
- The syntax for defining an interface is similar to that of defining a class
  - Except the word **`interface`** is used in place of **`class`**

  - **`public interface Person`**

- An interface specifies a set of methods that any class that implements the interface must have
  - It contains method headings and constant definitions only
  - It contains no instance variables nor any complete method definitions

The Person Interface



```
public interface Person
{
  public double getSalary(); // calculate salary, no implementation
} // end interface Person
```

Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected
  - When a class implements an interface, it must make all the methods in the interface public
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

Interfaces

- To *implement an interface*, a concrete class must do two things:

1. It must include the phrase

   ```
   implements Interface_Name
   ```

   at the start of the class definition

   ```
   public class Student implements Person
   ```

   – If more than one interface is implemented, each is listed, separated by commas

2. The class must implement all the method headings listed in the definition(s) of the interface(s)

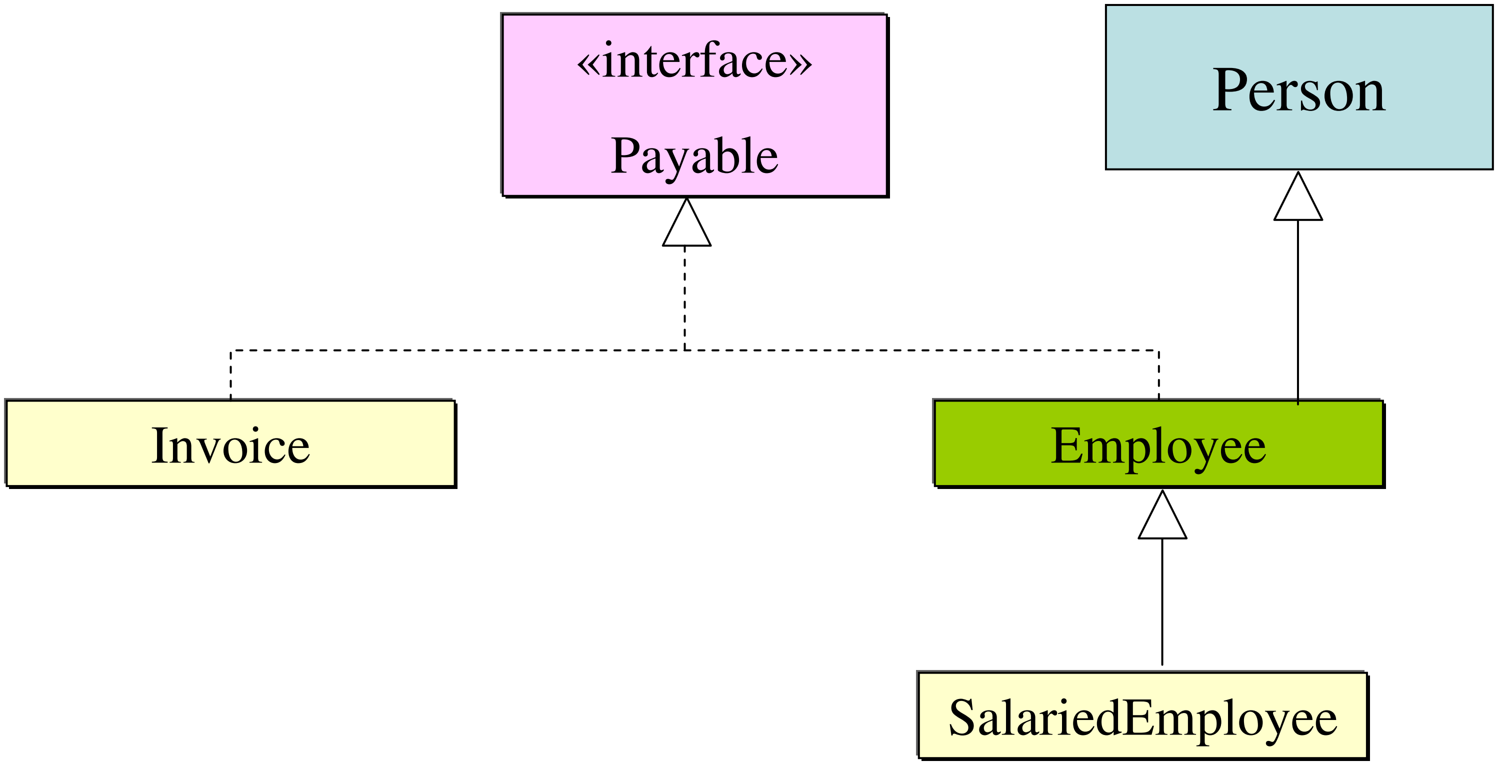
Implementation of an Interface

```
public class Student implements Person
{
   private int gpa;
   ……
   ……

   public double getSalary()
   {
      return (gpa * 200);
    }
}
```

Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

An Abstract Class Implementing an Interface



Payable & Person Class implementation

```
// Payable interface declaration.

public interface Payable

{  double getPaymentAmount(); // calculate payment; no implementation }
```

```
// Person class.

public class Person

{  protected String  address;

    public Person (String ad)

    {

         address = new String (ad);

      }

} // end Person class
```

Invoice class implementation

```
// Invoice class implements Payable.
public class Invoice implements Payable
{ private String partNumber,
  private String partDescription;
  private int quantity;
  private double pricePerItem;
  // constructor
  public Invoice( String part, String description,
                           int count, double price )
  { partNumber = part;
    partDescription = description;
    setQuantity( count );
    setPricePerItem( price );
  }
// set part number
  public void setPartNumber( String part )
  {  partNumber = part;
  }
```

```
// get part number
  public String getPartNumber()
  { return partNumber; }
// set description
  public void setPartDescription( String description )
  {  partDescription = description; }
// get description
  public String getPartDescription()
  { return partDescription; }
// set quantity
  public void setQuantity( int count )
  { quantity = ( count < 0 ) ? 0 : count; }
  // get quantity
  public int getQuantity()
  { return quantity; }
// set price per item
  public void setPricePerItem( double price )
  {  pricePerItem = ( price < 0.0 ) ? 0.0 : price; }
```

Invoice class implementation: Cont

```
// get price per item
   public double getPricePerItem()
   {  return pricePerItem; }
   // return String representation of Invoice object
   public String toString()
   {  return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
         "invoice", "part number", getPartNumber(), getPartDescription(),
         "quantity", getQuantity(), "price per item", getPricePerItem() );
   }
   // method required to carry out contract with interface Payable
   public double getPaymentAmount()
   {  return getQuantity() * getPricePerItem();    }
} // end class Invoice
```

Employee Abstract class implementation

```
// Employee abstract superclass implements Payable.
public abstract class Employee extends Person implements Payable
{ private String firstName;
   private String lastName;
   private String socialSecurityNumber;
  // four-argument constructor
   public Employee( String first, String last, String ssn, String ad )
   { supper (ad);
      firstName = first;  lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor
   // set first name
   public void setFirstName( String first )
   { firstName = first; } // end method setFirstName
    // return first name
   public String getFirstName()
   { return firstName; } // end method getFirstName
```

Employee Abstract class implementation: Cont

```
  public void setLastName( String last )
   { lastName = last; } // end method setLastName
  public String getLastName()
   { return lastName; } // end method getLastName
   public void setSocialSecurityNumber( String ssn )
   { socialSecurityNumber = ssn;} // end method setSocialSecurityNumber
    // return social security number
   public String getSocialSecurityNumber()
   {return socialSecurityNumber; } // end method getSocialSecurityNumber
    // return String representation of Employee object
   public String toString()
   { return String.format( "%s %s\nsocial security number: %s",
      getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString
   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

SalariedEmployee Concrete class implementation

```
// SalariedEmployee class extends Employee, which implements Payable.
public class SalariedEmployee extends Employee
{    private double weeklySalary;
     public SalariedEmployee( String first, String last, String ssn, double salary )
     {  super( first, last, ssn ); // pass to Employee constructor
        setWeeklySalary( salary ); // validate and store salary
     } // end four-argument SalariedEmployee constructor
   public void setWeeklySalary( double salary )
    {weeklySalary = salary < 0.0 ? 0.0 : salary; } // end method setWeeklySalary
   public double getWeeklySalary()
   { return weeklySalary; } // end method getWeeklySalary
   // calculate earnings; implement interface Payable method that was abstract in superclass Employee
   public double getPaymentAmount()
   { return getWeeklySalary(); } // end method getPaymentAmount
   public String toString()
   { return String.format( "salaried employee: %s\n%s: $%,.2f",
      super.toString(), "weekly salary", getWeeklySalary() ); } // end method toString
} // end class SalariedEmployee
```

PayableInterfaceTest

```
// Tests interface Payable.
public class PayableInterfaceTest
{  public static void main( String args[] )
   { // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];
      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] = new SalariedEmployee( “Ali", “Yassin", "111-11-1111", 800.00, “Malaz” );
      payableObjects[ 3 ] = new SalariedEmployee( “Med", “Ahmed", "888-88-8888", 1200.00, “Makka” );
      System.out.println( "Invoices and Employees processed polymorphically:\n" );
      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      { System.out.printf( "%s \n%s: $%,.2f\n\n", currentPayable.toString(), "payment due",
                                    currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

Derived Interfaces (Extending an Interface)

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase
    `extends BaseInterfaceName`
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

public interface X extends Y

Defined Constants in Interfaces

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be public, static, and final
  - Because this is understood, Java allows these modifiers to be omitted
- Any class that implements the interface has access to these defined constants

Chapter 5

Exceptions

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

Dr. S. HAMMAMI

Objectives

- After you have read and studied this chapter, you should be able to
  - Improve the reliability of code by incorporating exception-handling and assertion mechanisms.
  - Write methods that propagate exceptions.
  - Implement the try-catch blocks for catching and handling exceptions.
  - Write programmer-defined exception classes.
  - Distinguish the checked and unchecked, or runtime, exceptions.

Introduction to Exception Handling

- No matter how well designed a program is, there is always the chance that some kind of error will arise during its execution.

- A well-designed program should include code to handle errors and other exceptional conditions when they arise.

- Sometimes the best outcome can be when nothing unusual happens

- However, the case where exceptional things happen must also be prepared for

  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling the exception*

Definition

- An *exception* represents an error condition that can occur during the normal course of program execution.
- When an exception occurs, or is *thrown*, the normal sequence of flow is terminated.
- The exception-handling routine is then executed; we say the thrown exception is *caught*.

Not Catching Exceptions

- The avgFirstN() method expects that *N* > 0.
- If *N* = 0, a *divide-by-zero* error occurs in *avg/N*.

```
/**
  * Precondition:  N > 0
  * Postcondition: avgFirstN() equals the average of (1+2+…+N)
  */
public double avgFirstN(int N) {
    duuble sum = 0;
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;      // What if N is 0 ??
} // avgFirstN()
```

Bad Design: Doesn’t guard against divide-by-0.

Not Catching Exceptions

```
class AgeInputVer1 {
    private int age;
    public void setAge(String s) {
        age = Integer.parseInt(s);
    }
    public int getAge() {
        return age;
    }
}
```

```
public class AgeInputMain1 {
 public static void main( String[] args ) {
   AgeInputVer1 P = new AgeInputVer1( );
   P.setAge("nine");
  System.out.println(P.getAge());
  }
}
```

Error message for invalid input

```
Exception in thread "main"
java.lang.NumberFormatException: For input string: "nine"
at java.lang.NumberFormatException.forInputString(Unknown
Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at AgeInputVer1.setAge(AgeInputVer1.java:5)
at AgeInputMain1.main(AgeInputMain1.java:8)
```

Not Catching Exceptions

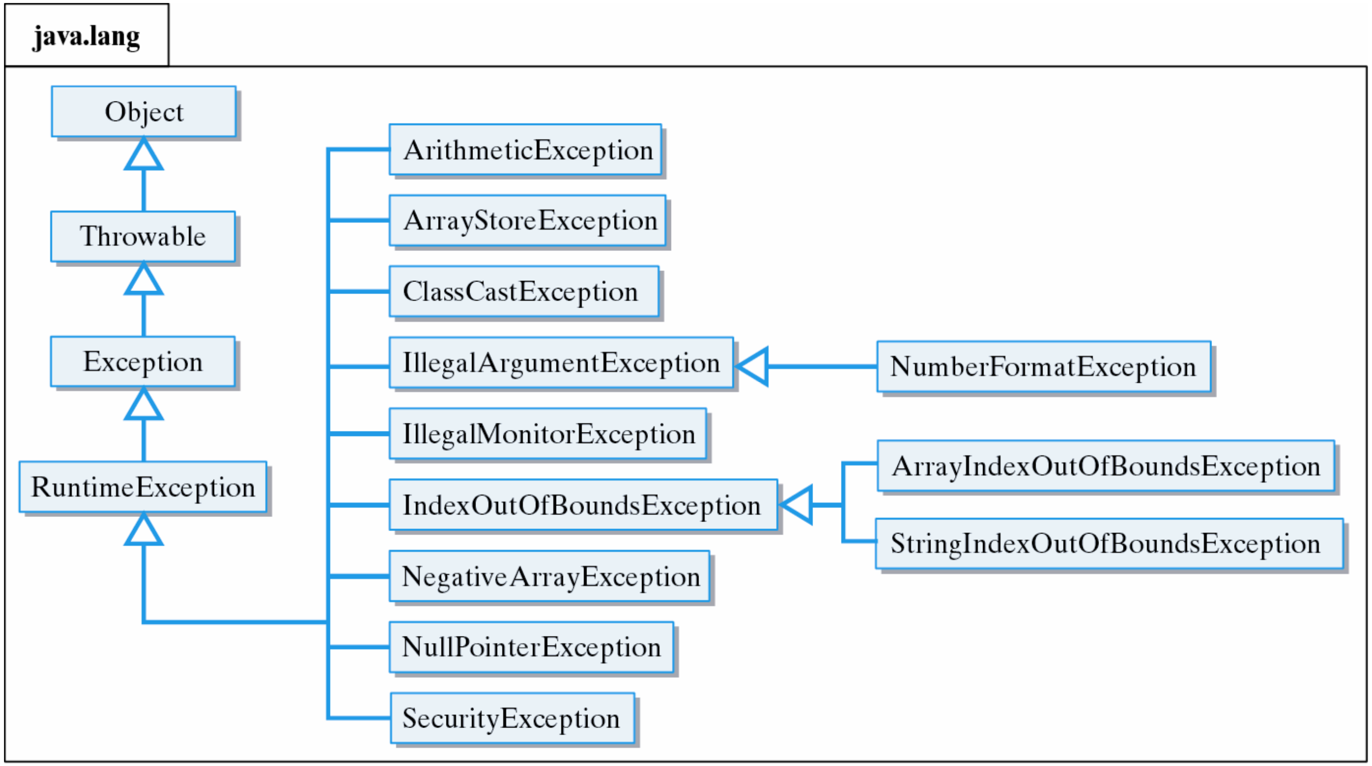
```
class AgeInputVer1 {
    private int age;
    public void setAge(String s) {
        age = Integer.parseInt(s);
    }
    public int getAge() {
        return age;
    }
}
```

```
public class AgeInputMain2 {
    public static void main( String[] args ) {
    AgeInputVer1 P = new AgeInputVer1( );
    P.setAge("9");
    System.out.println(P.getAge());
 }
}
```

```
9
```

Java's Exception Hierarchy

- *Unchecked exceptions*: belong to a subclass of RuntimeException and are not monitored by the compiler.



Some Important Exceptions

Class	Description
ArithmeticException	Division by zero or some other kind of arithmetic problem
ArrayIndexOutOfBounds-	An array index is less than zero or Exception greater than or equal to the array's length
FileNotFoundException	Reference to a unfound file IllegalArgumentException Method call with improper argument
IndexOutOfBoundsException	An array or string index out of bounds
NullPointerException	Reference to an object which has not been instantiated
NumberFormatException	Use of an illegal number format, such as when calling a method
StringIndexOutOfBoundsException	A String index less than zero or greater than or equal to the String's length

Catching an Exception

```
class AgeInputVer2 {
 private int age
 public void setAge(String s)
  {
    try {

        age = Integer.parseInt(s);

   } catch (NumberFormatException e){

          System.out.Println(“age is invalid, Please enter digits only");
   }
  }
  public int getAge() {    return age;     }
}
```

try

catch

We are catching the number format exception, and the parameter e represents an instance of the **NumberFormatException** class

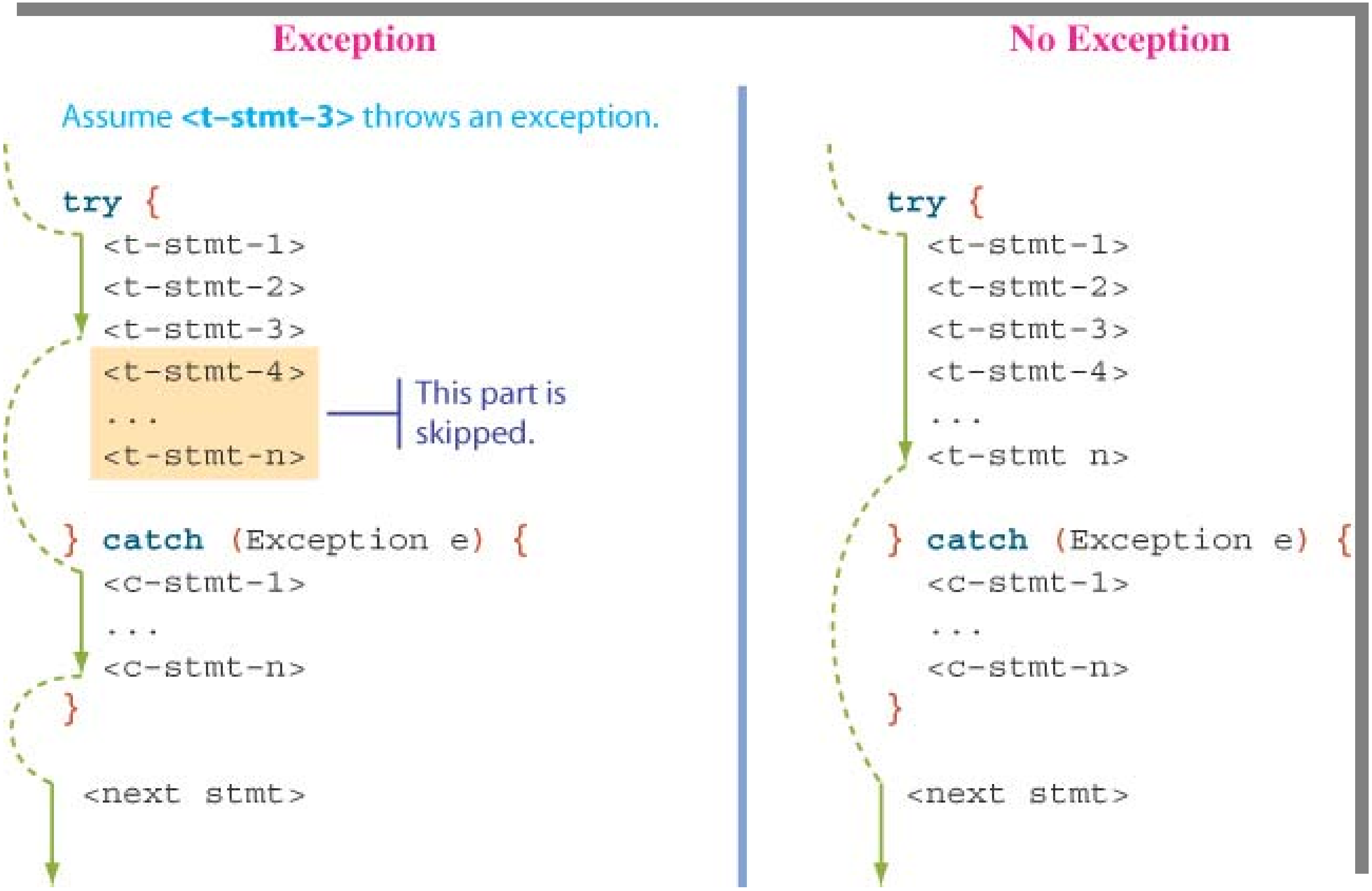
Catching an Exception

To accomplish this repetition, we will put the whole try-catch statement in side a loop:

```
import java.util.Scanner;

class AgeInputVer3 {

  private int age;

  public void setAge(String s) {

    String m =s;

    Scanner input = new Scanner(System.in);

    boolean ok = true;

     while (ok) {

       try {

            age = Integer.parseInt(m);

            ok = false;

       } catch (NumberFormatException e){

                  System.out.println("age is invalid, Please enter digits only");

                  m = input.next();

       }

   }

      public int getAge() {   return age;    }

}
```

This statement is executed only if no exception is thrown by **parseInt**.

try-catch Control Flow



Exception

Assume **<t-stmt-3>** throws an exception.

```
try {
   <t-stmt-1>
   <t-stmt-2>
   <t-stmt-3>
   <t-stmt-4>
   . . .
   <t-stmt-n>

} catch (Exception e) {
   <c-stmt-1>
   . . .
   <c-stmt-n>
}

  <next stmt>
```

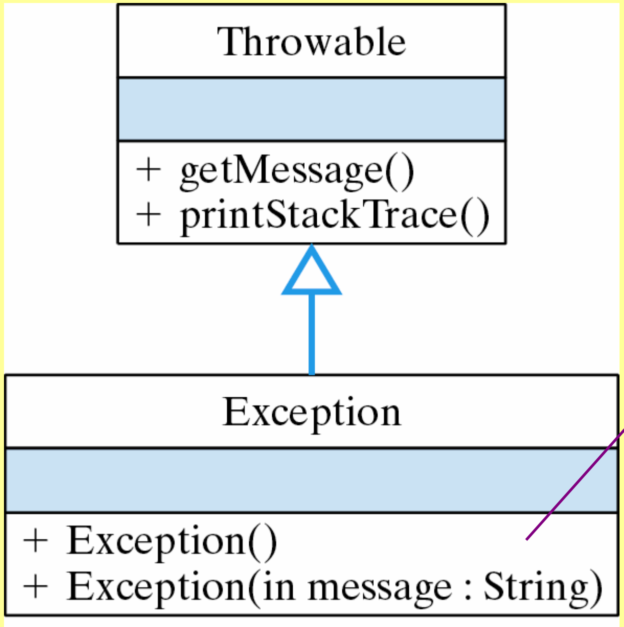
This part is skipped.

No Exception

```
try {
   <t-stmt-1>
   <t-stmt-2>
   <t-stmt-3>
   <t-stmt-4>
   . . .
   <t-stmt n>

} catch (Exception e) {
   <c-stmt-1>
   . . .
   <c-stmt-n>
}

  <next stmt>
```

The Exception Class: Getting Information

- There are two methods we can call to get information about the thrown exception:
  - **getMessage**
  - **printStackTrace**



Simple: only constructor methods.

The Exception Class: Getting Information

We are catching the number format exception, and the parameter e represents an instance of the **NumberFormatException** class

```
try {
    . . .
} catch (NumberFormatException e){

    System.out.println(e.getMessage());
    System.out.println(e.printStackTrace());
}
```

```
For input string: "nine"
```

```
java.lang.NumberFormatException: For input string: "nine"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at AgeInputVer1.setAge(AgeInputVer1.java:11)
at AgeInputMain1.main(AgeInputMain1.java:8)
```

`try-throw-catch` Mechanism

throw new
***ExceptionClassName*(*PossiblySomeArguments*);**

- When an exception is thrown, the execution of the surrounding **try** block is stopped
  - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class
  - The execution of a **throw** statement is called *throwing an exception*

`try-throw-catch` Mechanism

- A throw statement is similar to a method call:
  - throw new ExceptionClassName(SomeString);
  - In the above example, the object of class ExceptionClassName is created using a string as its argument
  - This object, which is an argument to the throw operator, is the exception object thrown
- Instead of calling a method, a throw statement calls a catch block

`try-throw-catch` Mechanism

- When an exception is thrown, the catch block begins execution
  - The catch block has one parameter
  - The exception object thrown is plugged in for the catch block parameter
- The execution of the catch block is called catching the exception, or handling the exception
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some catch block

`try-throw-catch` Mechanism

- When a **`try`** block is executed, two things can happen:
  1. No exception is thrown in the try block
     - The code in the **`try`** block is executed to the end of the block
     - The **`catch`** block is skipped
     - The execution continues with the code placed after the **`catch`** block

- 2. An exception is thrown in the try block and caught in the catch block

  The rest of the code in the try block is skipped

  Control is transferred to a following catch block (in simple cases)

  The thrown object is plugged in for the catch block parameter

  The code in the catch block is executed

  The code that follows that catch block is executed (if any)

```
public class CalcAverage {
  public double avgFirstN(int N ){
    double sum = 0;
    try {
       if (N <=0)
        throw new Exception("ERROR: Can't average 0 elements");
       for (int k = 1; k <= N; k++)
         sum += k;
       return sum/N;
  }
}

public class CalcAverage {
 public double avgFirstN(int N ){
  double sum = 0;
  try {
    if (N <0)
    throw new Exception("ERROR: Can't average negative elements");
    for (int k = 1; k <= N; k++)
      sum += k;
    return sum/N;
  }
  catch(ArithmeticException e)
  {
   System.out.println(e.getmessage());
   System.out.println(“N=Zero is an valid denomiattor, please try again ");
  }
  catch (Exception e) {System.out.println e.getmessage() + “Please enter
                                                    positive integer”);}
  }
}
```

Multiple catch Blocks

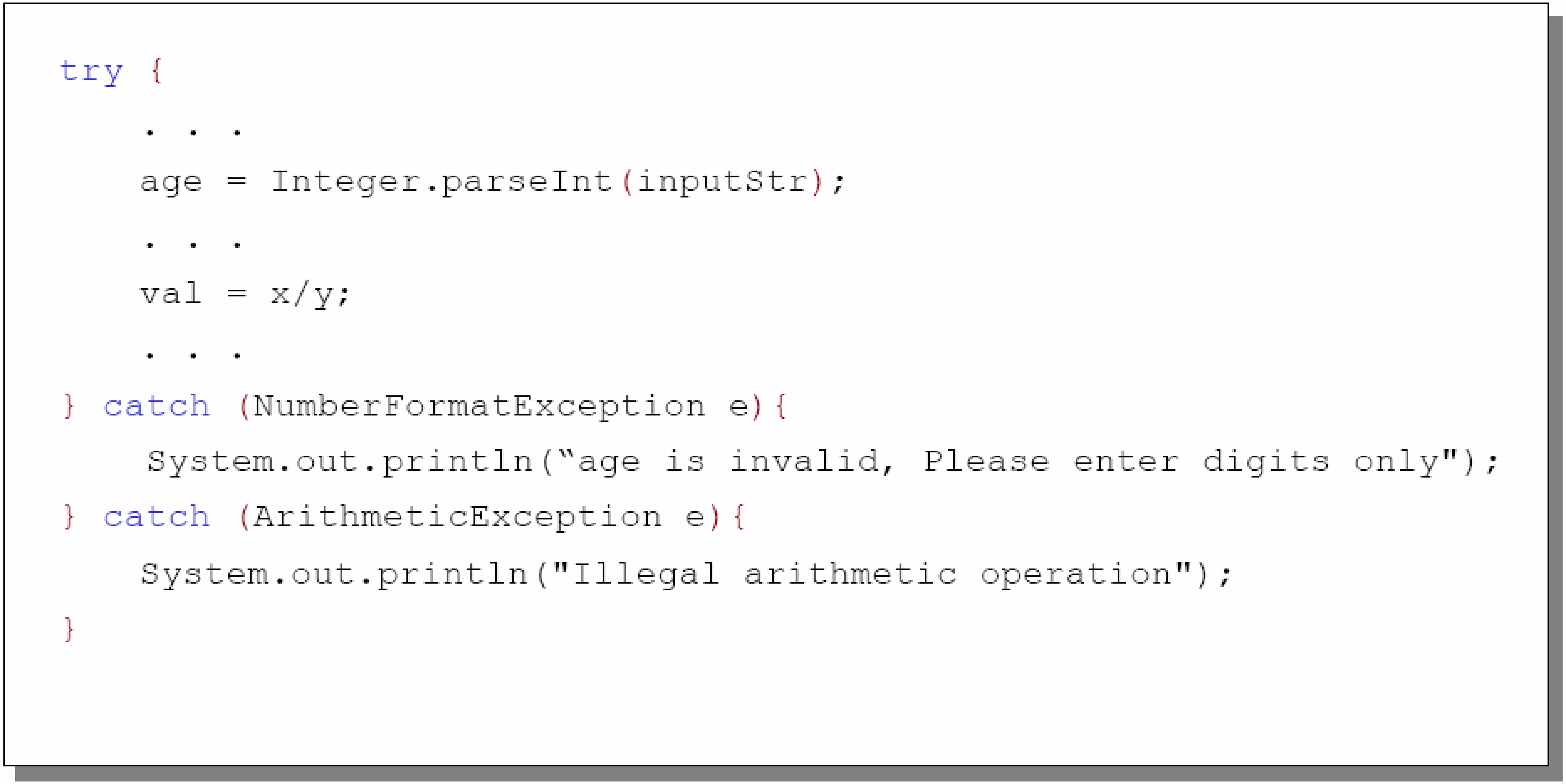
- A **try** block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a **try** block, at most one exception can be thrown (since a throw statement ends the execution of the **try** block)
  - However, different types of exception values can be thrown on different executions of the **try** block

- Each **catch** block can only catch values of the exception class type given in the **catch** block heading

- Different types of exceptions can be caught by placing more than one **catch** block after a **try** block
  - Any number of **catch** blocks can be included, but they must be placed in the correct order

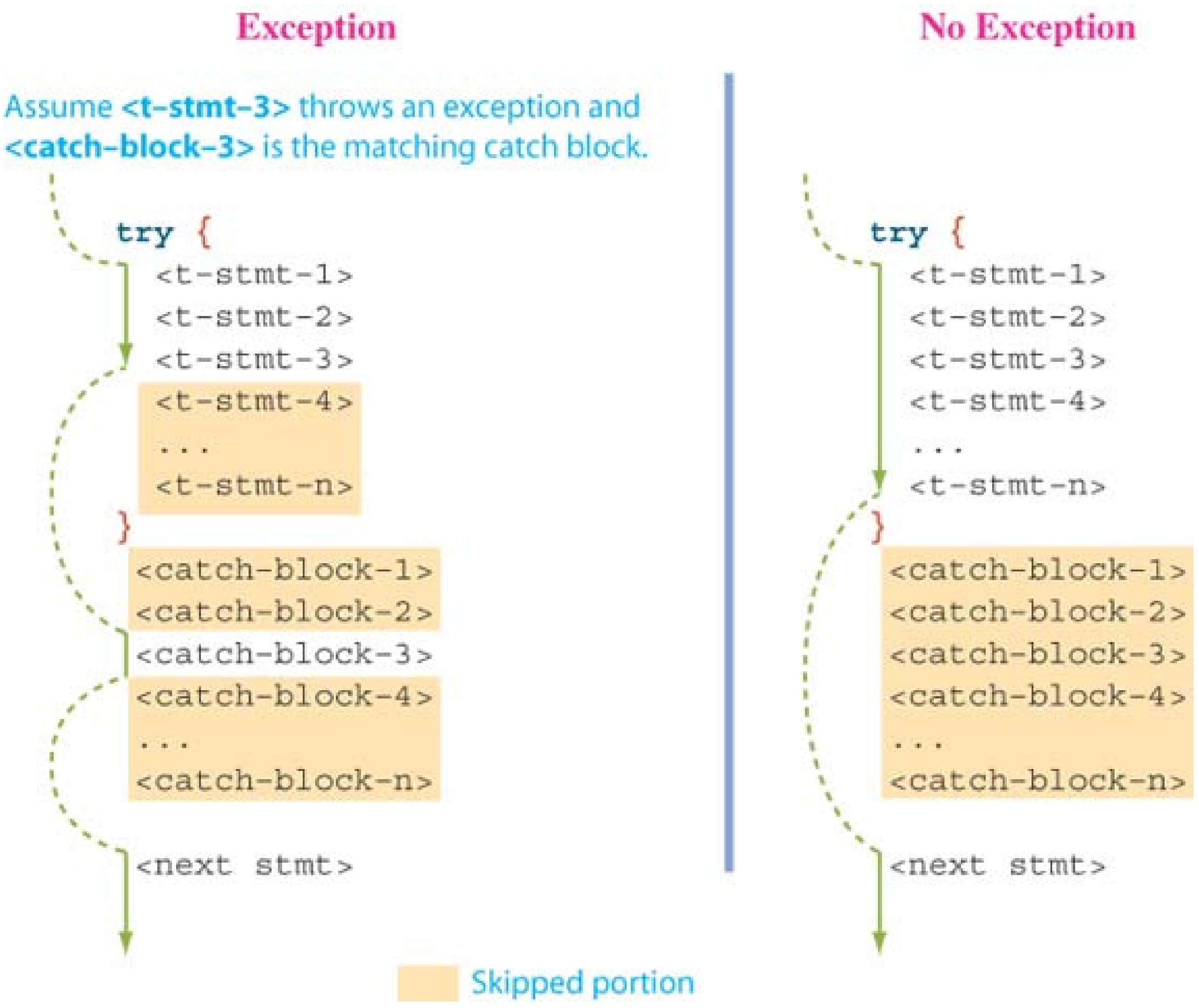
Multiple catch Blocks

- A single try-catch statement can include multiple catch blocks, one for each type of exception.



```
try {
     . . .
     age = Integer.parseInt(inputStr);
     . . .
     val = x/y;
     . . .
} catch (NumberFormatException e){
      System.out.println(“age is invalid, Please enter digits only");
} catch (ArithmeticException e){
     System.out.println("Illegal arithmetic operation");
}
```

Multiple catch Control Flow



Exception

Assume <t-stmt-3> throws an exception and <catch-block-3> is the matching catch block.

```
try {
   <t-stmt-1>
   <t-stmt-2>
   <t-stmt-3>
   <t-stmt-4>
   . . .
   <t-stmt-n>
}
 <catch-block-1>
 <catch-block-2>
 <catch-block-3>
 <catch-block-4>
 . . .
 <catch-block-n>

 <next stmt>
```

No Exception

```
try {
   <t-stmt-1>
   <t-stmt-2>
   <t-stmt-3>
   <t-stmt-4>
   . . .
   <t-stmt-n>
}
 <catch-block-1>
 <catch-block-2>
 <catch-block-3>
 <catch-block-4>
 . . .
 <catch-block-n>

 <next stmt>
```

🟧 Skipped portion

Multiple catch Control Flow: Example

```
import java.util.*; //InputMismatchException;
//import java.util.ArithmeticException;
public class DividbyZero2
{   public static void main (String args[]) //throws ArithmeticException
    {  Scanner input = new Scanner(System.in);
       boolean done=false;
       do {
          try
          {System.out.print("Please enter an integer number :  ");
            int a =input.nextInt();
            System.out.print("Please enter an integer number :  ");
            int b =input.nextInt();
            int c=a/b;
            System.out.println("a ="+ a +"  b= "+b+ "   amd quotient ="+c);
            done=true;
          }
```

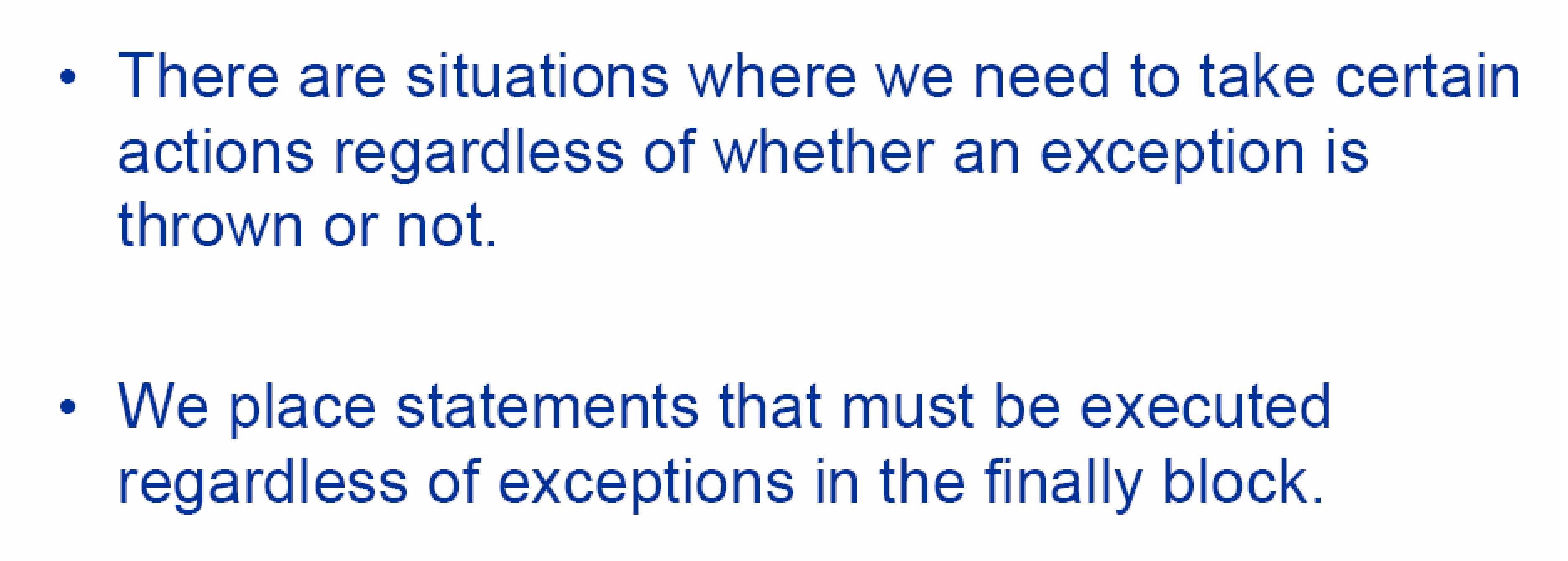
```
    catch (InputMismatchException var1 )
            {  System.out.println("Exception :"+ var1);
               System.out.println("please try again: ");
            }
   catch(ArithmeticException var2)
            {
               System.out.println("\nException :"+ var2);
               System.out.println("Zero is an valid denomiattor");
            }
  }while(!done);
 }
 }
```

```
Please enter an integer number :  car
Exception :java.util.InputMismatchException
You must enter an integer value, please try again:
Please enter an integer number :  14
Please enter an integer number :  0

Exception :java.lang.ArithmeticException: / by zero
Zero is an valid denomiattor, please try again
Please enter an integer number :  7
Please enter an integer number :  3
a =7  b= 3   amd quotient =2
```

Pitfall: Catch the More Specific Exception First

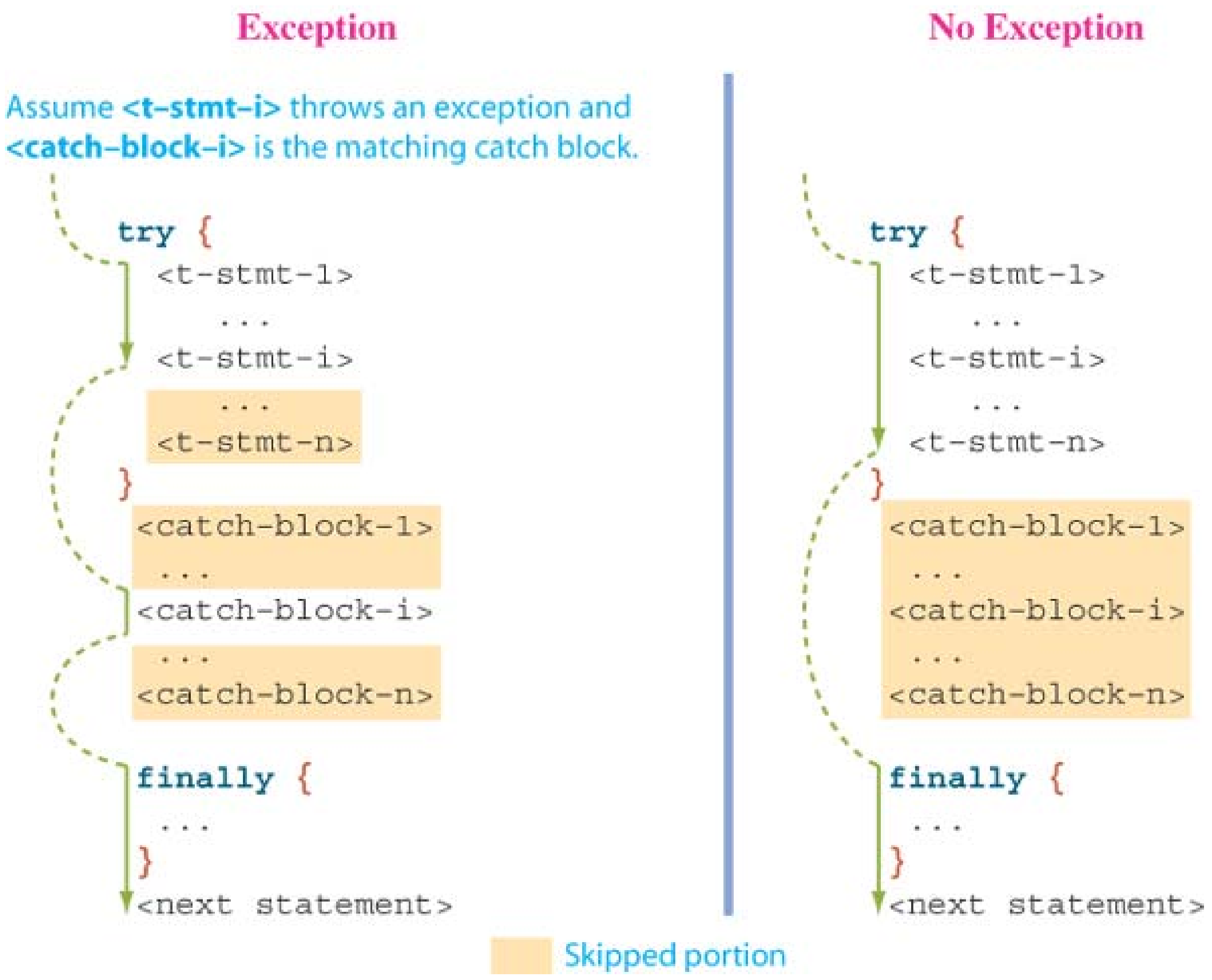
- When catching multiple exceptions, the order of the **`catch`** blocks is important
  - When an exception is thrown in a **`try`** block, the **`catch`** blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

The finally Block



- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions in the finally block.

try-catch-finally Control Flow



Exception

Assume **<t-stmt-i>** throws an exception and **<catch-block-i>** is the matching catch block.

```
try {
  <t-stmt-1>
     ...
  <t-stmt-i>
     ...
  <t-stmt-n>
}
<catch-block-1>
 ...
<catch-block-i>
 ...
<catch-block-n>
finally {
 ...
}
<next statement>
```

No Exception

```
try {
  <t-stmt-1>
     ...
  <t-stmt-i>
     ...
  <t-stmt-n>
}
<catch-block-1>
 ...
<catch-block-i>
 ...
<catch-block-n>
finally {
 ...
}
<next statement>
```

▨ Skipped portion

try-catch-finally Control Flow

- If the **`try-catch-finally`** blocks are inside a method definition, there are three possibilities when the code is run:

1. The **`try`** block runs to the end, no exception is thrown, and the **`finally`** block is executed

2. An exception is thrown in the **`try`** block, caught in one of the **`catch`** blocks, and the **`finally`** block is executed

3. An exception is thrown in the **`try`** block, there is no matching **`catch`** block in the method, the **`finally`** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

Propagating Exceptions

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a try block, and catch the exception in a catch block that follows
- In this case, the method itself would not include try and catch blocks
  - However, it would have to include a throws clause

Declaring Exceptions in a `throws` Clause

- If a method can throw an exception but does not catch it, it must provide a warning
  - This warning is called a throws clause
  - The process of including an exception class in a throws clause is called declaring the exception

    ```
    throws AnException  //throws clause
    ```

  - The following states that an invocation of aMethod could throw AnException

    ```
    public void aMethod() throws AnException
    ```

Declaring Exceptions in a **`throws`** Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws AnException,AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

Exception propagation and throws clause

Method **getDepend()** may throw a ***number format exception*** when converting a string to an integer, but it does not catch this exception.

The call to **getDepend()** occurs in the try block of method **main()**, so **main()** handles the exception in its catch block.

If main() did not have a catch block for number format exceptions, the exception would be handled by the JVM.

```
// postcondition: Returns int value of a numeric data string.
//   Throws an exception if string is not numeric.
public static int getDepend() throws NumberFormatException {
   String numStr = jnputnext();
   return Integer.parseInt(numStr);
}

// postcondition: Calls getDepend() and handles its exceptions.
public static void main(String[] args) {
  int children = 1;     // problem input, default is 1
  try {
        children = getDepend();
  }
  catch (NumberFormatException ex) {
     // Handle number format exception.
      System.out.println("Invalid integer“ + ex);
  }
}
```

Exception Thrower

- When a method may throw an exception, either directly or indirectly, we call the method an *exception thrower*.

- Every exception thrower must be one of two types:

  – catcher.

  – propagator.

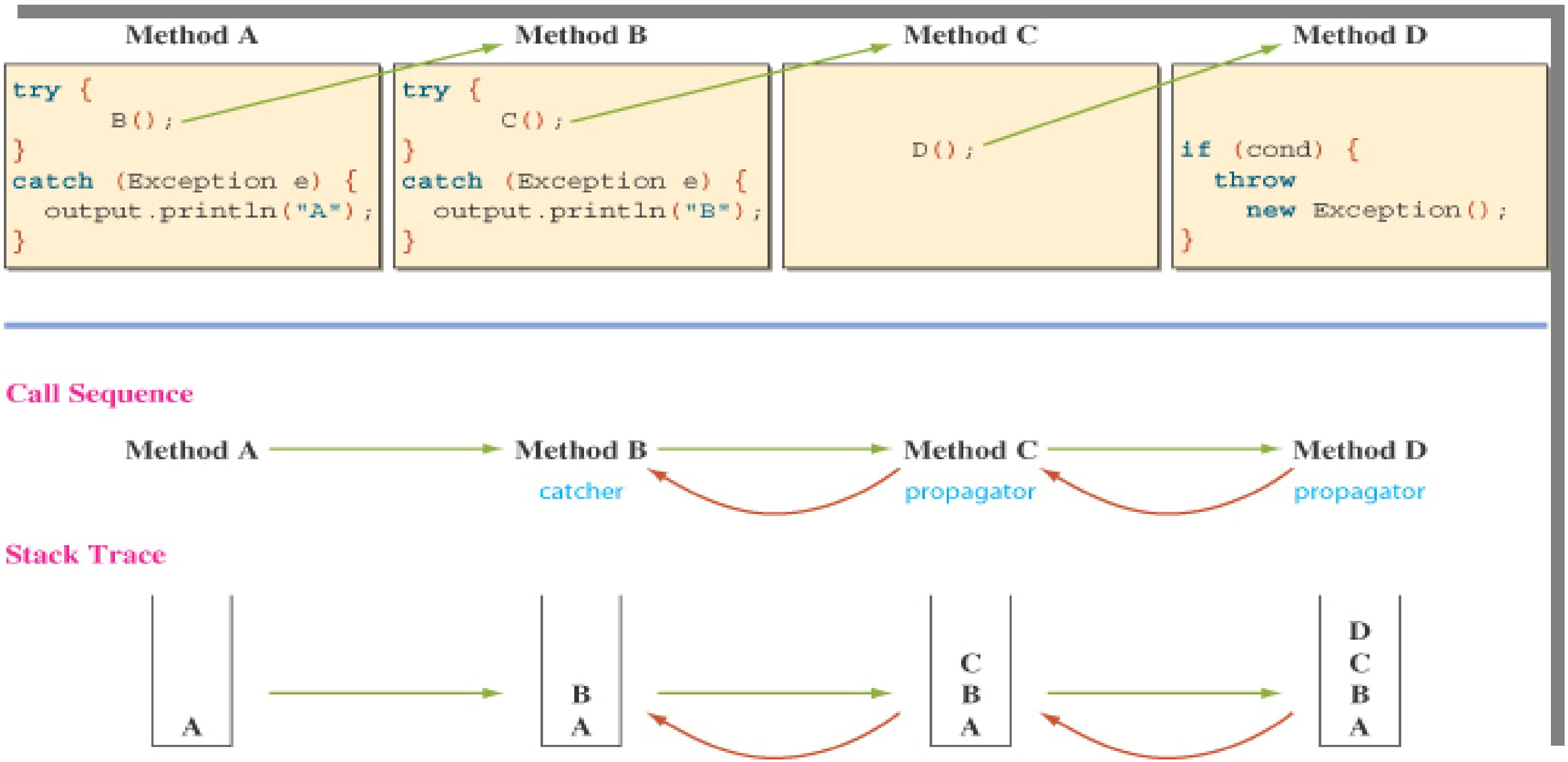
Types of Exception Throwers

- An *exception catcher* is an exception thrower that includes a matching **catch** block for the thrown exception.

- An *exception propagator* does not contain a matching **catch** block.

- A method may be a catcher of one exception and a propagator of another.

Sample Call Sequence



Method A

```
try {
      B();
}
catch (Exception e) {
  output.println("A");
}
```

Method B

```
try {
      C();
}
catch (Exception e) {
  output.println("B");
}
```

Method C

```
D();
```

Method D

```
if (cond) {
  throw
    new Exception();
}
```

Method A calls method B,
Method B calls method C,
Method C calls method D.

Every time a method is executed, the method’s name is placed on top of the stack.

Sample Call Sequence

- When an exception is thrown, the system searches **down** the **stack** from the top, looking for the **first matching exception catcher**.

- Method D throws an exception, but **no** matching **catch block** exits in the method, so method D is an **exception propagator**.

- The system then checks method C. C is also an **exception propagator**.

- Finally, the system locates the matching catch block in method B, and therefore, method B is the catcher for the exception thrown by method D.

- Method A also includes the matching catch block, but it will not be executed because the thrown exception is already caught by method B and method B does not propagate this exception.

```
void C() throws Exception {
   ….
}
```

```
void D() throws Exception {
   ….
}
```

Example

Consider the Fraction class. The setDenominator method of the Fraction class
was defined as follows:

Throwing an exception is a much better approach. Here’s the modified method that throws an IlleglArgumentException when the value of 0 is passed as an argument:

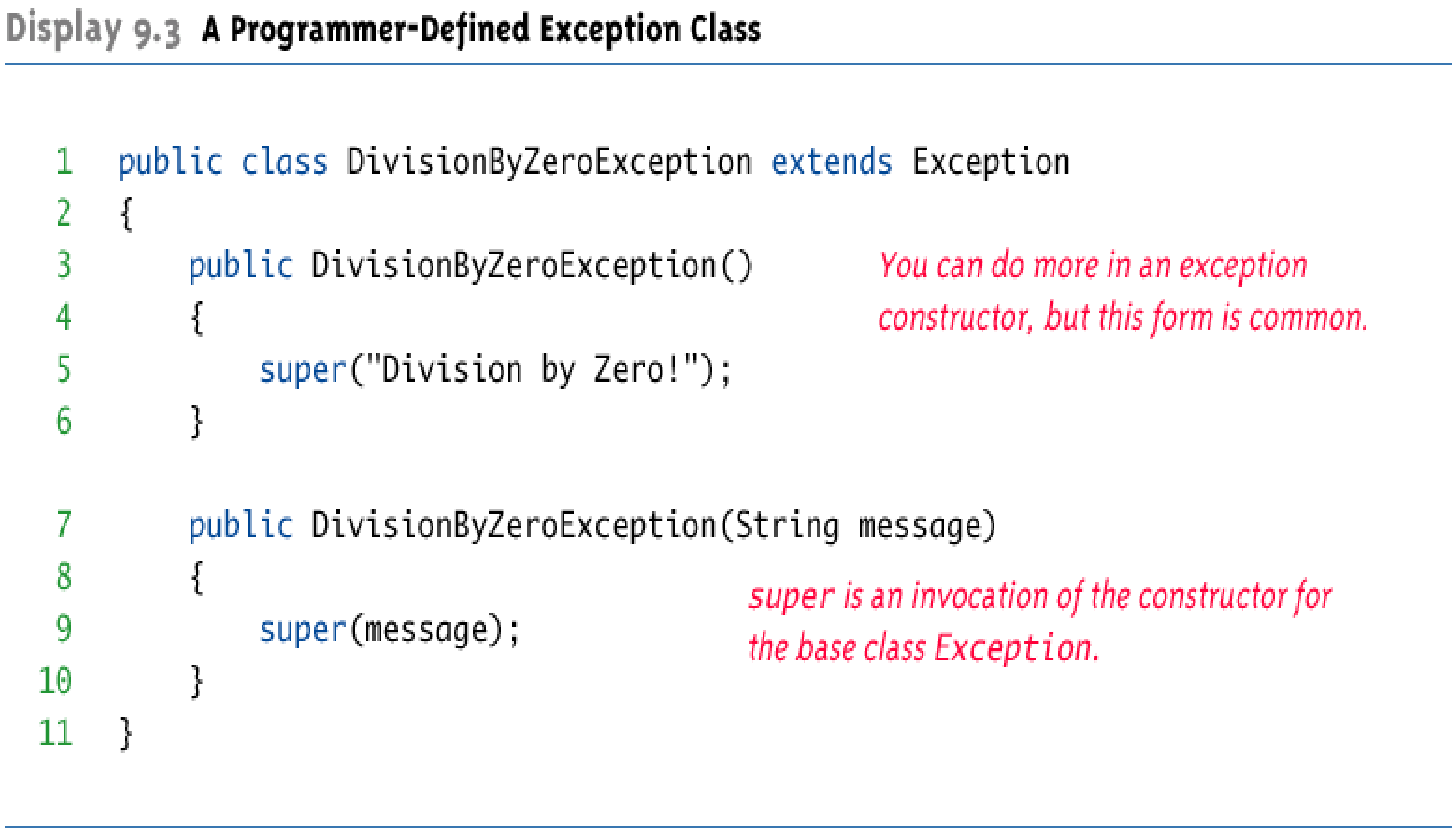
```
public void setDenominator (int d)
{
   if (d = = 0)
   {
      System.out.println(“Fatal Error”);
      System.exit(1);
   }
   denominator = d;
}
```

```
public void setDenominator (int d)
                              throws IlleglArgumentException
{
   if (d = = 0)
   {
      throw new IlleglArgumentException (“Fatal Error”);

   }
   denominator = d;
}
```

Programmer-Defined Exception Classes

- A **`throw`** statement can throw an exception object of any exception class
- Instead of using a predefined class, *exception classes can be programmer-defined*
  - These can be tailored to carry the precise kinds of information needed in the **`catch`** block
  - A different type of exception can be defined to identify each different exceptional situation
- Every exception class to be defined must be a sub-class of some already defined exception class
  - It can be a sub-class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

A Programmer-Defined Exception Class



Display 9.3 A Programmer-Defined Exception Class

```java
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class

- The class **`Exception`** can be used as the base class, unless another class would be more suitable

- At least two constructors should be defined, sometimes more

- The exception class should allow for the fact that the method **`getMessage`** is inherited

Programmer-Defined Exceptions: AgeInputException

```
class AgeInputException extends Exception
{
   private static final String DEFAULT_MESSAGE = "Input out of bounds";
   private int lowerBoun, upperBound, value;
   public AgeInputException(int low, int high, int input)
{ this(DEFAULT_MESSAGE, low, high, input);   }
   public AgeInputException(String msg, int low, int high, int input)
    {     super(msg);
          if (low > high)    throw new IllegalArgumentException();
          lowerBound = low;   upperBound = high;    value      = input;
    }
    public int lowerBound()  {  return lowerBound;}
   public int upperBound()   {return  upperBound;}
    public int value() {  return value;}
}
```

Class AgeInputVer5 Uses AgeInputException

```
import java.util.Scanner;
class AgeInputVer5 {
    private static final String DEFAULT_MESSAGE = "Your age:";
    private static final int DEFAULT_LOWER_BOUND = 0;
    private static final int DEFAULT_UPPER_BOUND = 99;
    private int lowerBound, upperBound;
    public AgeInputVer5( ) throws IllegalArgumentException {
        setBounds(DEFAULT_LOWER_BOUND, DEFAULT_UPPER_BOUND);
    }
    public AgeInputVer5(int low, int high)   throws IllegalArgumentException
    {
        if (low > high)
        {    throw new IllegalArgumentException( "Low (" + low + ") was " +
                 "larger than high(" + high + ")");
        } else setBounds(low, high);
    }
   public int getAge() throws AgeInputException
 {
  return      getAge(DEFAULT_MESSAGE);
    }
public int getAge(String prompt) throws AgeInputException  {
      Scanner T = new Scanner(System.in);
      String inputStr;    int    age;
      while (true)  {
         inputStr  = prompt;
         try
         {
            age = Integer.parseInt(inputStr);
            if (age < lowerBound || age > upperBound) {
                throw new AgeInputException("Input out of bound  ",
                                    lowerBound, upperBound, age);
          }
          return age; //input okay so return the value & exit
         } catch (NumberFormatException e) {
            System.out.println("\n"+ inputStr + " is invalid age.");
            System.out.print("Please enter age as an integer value : ");
            prompt = T.next()+T.nextLine();
         }       } }
   private void setBounds(int low, int high) { lowerBound = low;   upperBound = high;
   }}
```

Main Using throws

```
public class TestAgeInputUsingThrows {
    public static void main( String[] args ) throws AgeInputException  {
        int   entrantAge=0;
        AgeInputVer5 input = new AgeInputVer5(25, 50);
        entrantAge   = input.getAge("Thirty");
         System.out.println("Input Okay ");  }  }
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty


fourty is invalid age.
Please enter age as an integer value : 40
Input Okay
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 55
Exception in thread "main" AgeInputException: Input out of bound
at AgeInputVer5.getAge(AgeInputVersion5.java:42)
at TestAgeInputVer5.main(TestAgeInputVer5.java:7)
```

Main Using try-catch

```
public class Test2AgeInput {
    public static void main( String[] args )  {
       int   entrantAge;
       try   {
            AgeInputVer5 input = new AgeInputVer5(25, 50);
            entrantAge   = input.getAge("Thirty");
             System.out.println("Input Okay ");
        }
        catch (AgeInputException e)  {
        System.out.println("Error: " + e.value() + " is entered. It is " +  "outside the valid range of [" +
           e.lowerBound() +", " + e.upperBound() + "]"); }     }  }
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 40
Input Okay
```

```
Thirty is invalid age.
Please enter age as an integer value : fourty

fourty is invalid age.
Please enter age as an integer value : 55
Error: 55 is entered. It is outside the valid range of [25, 50]
```

Chapter 3

File Input/Output

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

Dr. S. HAMMAMI

Chapter 3: Objectives

- After you have read and studied this chapter, you should be able to
  - Include a JFileChooser object in your program to let the user specify a file.
  - Write bytes to a file and read them back from the file, using FileOutputStream and FileInputStream.
  - Write values of primitive data types to a file and read them back from the file, using DataOutputStream and DataInputStream.
  - Write text data to a file and read them back from the file, using PrintWriter and BufferedReader
  - Read a text file using Scanner
  - Write objects to a file and read them back from the file, using ObjectOutputStream and ObjectInputStream

Files

- Storage of data in variables and arrays is temporary—the data is lost when a local variable goes out of scope or when the program terminates.

- Computers use files for long-term retention of large amounts of data, even after programs that create the data terminate. We refer to data maintained in files as persistent data, because the data exists beyond the duration of program execution.

- Computers store files on secondary storage devices such as magnetic disks, optical disks and magnetic tapes.

Files

There are two general types of files you need to learn about: ***text*** files and ***binary*** files…

- A **text**, or character-based, file stores information using ASCII character representations. Text files can be viewed with a standard editor or word processing program but cannot be manipulated arithmetically without requiring special conversion routines.

- A **binary** file stores numerical values using the internal numeric binary format specified by the language in use. A Java program can read a binary file to get numeric data, manipulate the data arithmetically, and write the data to a binary file without any intermediate conversions.

File Operations

There are three basic operations that you will need to perform when working with disk files:

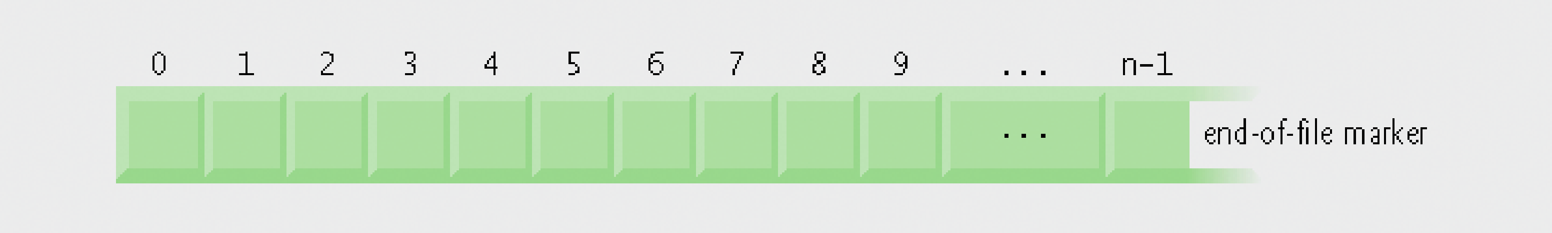
- Open the file for input or output.

- Process the file, by reading from or writing to the file.

- Close the file.

Files and Streams

- Java views each files as a **sequential stream of bytes**
- Operating system provides mechanism to determine end of file
  - End-of-file marker
  - Count of total bytes in file
  - Java program processing a stream of bytes receives an indication from the operating system when program reaches end of stream



Java's view of a file of *n* bytes.

Files and Streams

- File streams
  - Byte-based streams – stores data in binary format
    - **Binary files** – created from byte-based streams, read by a program that converts data to human-readable format
  - Character-based streams – stores data as a sequence of characters
    - **Text files** – created from character-based streams, can be read by text editors
- Java opens file by creating an object and associating a stream with it
- Standard streams – each stream can be redirected
  - `System.in` – standard input stream object, can be redirected with method setIn
  - `System.out` – standard output stream object, can be redirected with method setOut
  - `System.err` – standard error stream object, can be redirected with method setErr

The Class File

- Class `File` useful for retrieving information about files and directories from disk

- Objects of class `File` do not open files or provide any file-processing capabilities

- File objects are used frequently with objects of other `java.io` classes to specify files or directories to manipulate.

Creating File Objects

- To operate on a file, we must first create a File object (from java.io).

Class `File` provides constructors:

1. Takes `String` specifying name and path (location of file on disk)

```
File filename = new File(“sample.dat”);
```

Opens the file sample.dat in the current directory.

```
File filename = new File(“C:/SamplePrograms/test.dat”);
```

Opens the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the full pathname.

2. Takes two Strings, first specifying path and second specifying name of file

```
File filename = new File(String pathToName, String Name);
```

File Methods

Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a string with the absolute path of the file or directory.
`String getName()`	Returns a string with the name of the file or directory.
`String getPath()`	Returns a string with the path of the file or directory.
`String getParent()`	Returns a string with the parent directory of the file or directory (i.e., the directory in which the file or directory can be found).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, `0` is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of strings representing the contents of a directory. Returns `null` if the `File` object does not represent a directory.

Some File Methods

```
if (filename.exists( ) ) {
```

To see if `filename` is associated to a real file correctly.

```
if (filename.isFile() ) {
```

To see if `filename` is associated to a file or not. If false, it is a directory.

```
File directory = new
    File("C:/JavaPrograms/Ch4");

String Arrayfilename[] = directory.list();

for (int i = 0; i < Arrayfilename.length; i++)
{
    System.out.println(Arrayfilename[i]);
}
```

List the name of all files in the directory C:\JavaProjects\Ch4

Demonstrating Class File

```java

// Demonstrating the File class.
import java.io.File;

public class FileDemonstration
{
   // display information about file user specifies
   public void analyzePath( String path )
   {
      // create File object based on user input
      File name = new File( path );

      if ( name.exists() ) // if name exists, output information about it
      {
         // display file (or directory) information
         System.out.printf(
            "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
            name.getName(), " exists",
            ( name.isFile() ? "is a file" : "is not a file" ),
            ( name.isDirectory() ? "is a directory" :
               "is not a directory" ),
            ( name.isAbsolute() ? "is absolute path" :
               "is not absolute path" ), "Last modified: ",
            name.lastModified(), "Length: ", name.length(),
            "Path: ", name.getPath(), "Absolute path: ",
            name.getAbsolutePath(), "Parent: ", name.getParent() );

```

Create new **File** object; user specifies file name and path

Returns **true** if file or directory specified exists

Retrieve name of file or directory

Returns **true** if **name** is a file, not a directory

Returns **true** if **name** is a directory, not a file

Returns **true** if **path** was an absolute path

Retrieve length of file in bytes

Retrieve time file or directory was last modified (system-dependent value)

Retrieve absolute path of file or directory

Retrieve path entered as a string

Retrieve parent directory (path where **File** object's file or directory can be found)

```
         if ( name.isDirectory() ) // output directory listing
         {
            String directory[] = name.list();
            System.out.println( "\n\nDirectory contents:\n" );

            for ( String directoryName : directory )
               System.out.printf( "%s\n", directoryName );
         } // end else
      } // end outer if
      else // not file or directory, output error message
      {
         System.out.printf( "%s %s", path, "does not exist." );
      } // end else
   } // end method analyzePath
} // end class FileDemonstration
```

Returns true if `File` is a directory, not a file

Retrieve and display contents of directory

```

// Testing the FileDemonstration class.
import java.util.Scanner;

public class FileDemonstrationTest
{
   public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );
      FileDemonstration application = new FileDemonstration();

      System.out.print( "Enter file or directory name here: " );
      application.analyzePath( input.nextLine() );
   } // end main
} // end class FileDemonstrationTest
```

```
Enter file or directory name here: C:\Program Files\Java\jdk1.5.0\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1083938776645
Length: 0
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Parent: C:\Program Files\Java\jdk1.5.0\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
TableExample
```

```
Enter file or directory name here:
C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
readme.txt exists
is a file
is not a directory
is absolute path
Last modified: 1083938778347
Length: 7501
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Parent: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D
```

Low-Level File I/O

- To read data from or write data to a file, we must create one of the Java stream objects and attach it to the file.
- A *stream* is a sequence of data items (sequence of characters or bytes) used for program input or output. Java provides many different input and output stream classes in the **java.io** API.
- A ***file stream*** is an *object* that enables the flow of data between a program and some I/O device or file

Low-Level File I/O

- – Java has two types of streams: an *input stream* and an *output stream*.
- – If the data flows into a program, then the stream is called an **input stream**
- – If the data flows out of a program, then the stream is called an **output stream**

Streams for Low-Level File I/O

Binary File Stream Classes

FileInputStream	To open a binary input stream and connect it to a physical disk file
FileOutputStream	To open a binary output stream and connect it to a physical disk file
DataInputStream	To read binary data from a stream
DataOutputStream	To write binary data to a stream

A File Has Two Names

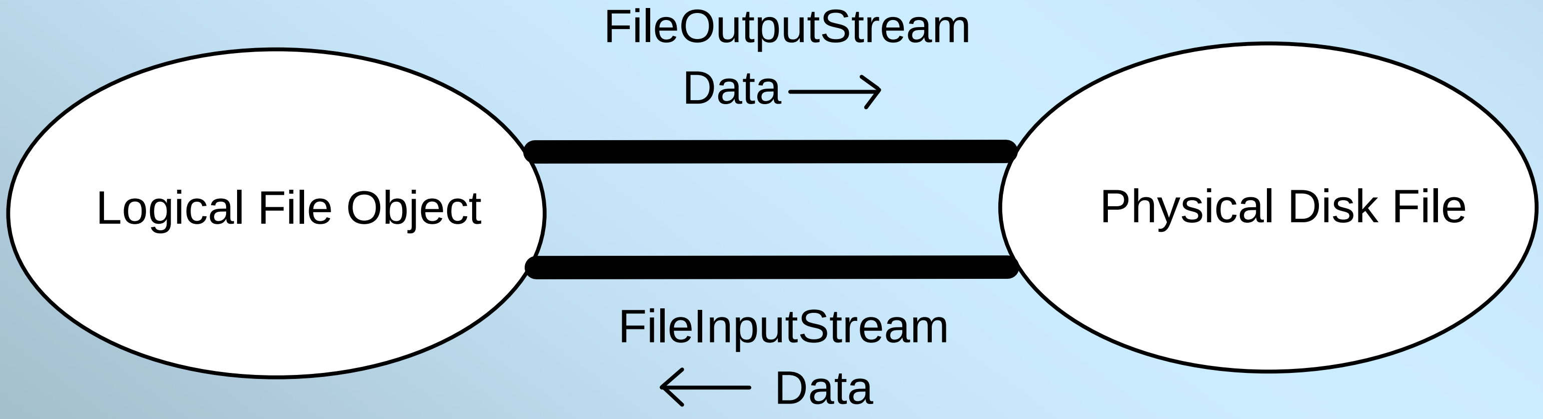
- Every input file and every output file used by a program has two names:

1. The real file name used by the operating system
2. The name of the stream that is connected to the file

- The actual file name is used to connect to the stream
- The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

Opening a File

A ***file stream*** provides a connection between your program and the outside world. Opening a file makes the connection between a logical program object and a physical file via the file stream.



Opening a Binary File for Output

Using the FileOutputStream class, create a file stream and connect it to a physical disk file to open the file. We can output only a sequence of bytes.

```
Import java.io.*
Class TestFileOuputStream {
Public static void main (String [] args) throws IOException
{
  //set up file and stream
  File  F   = new File("sample1.data");

  FileOutputStream OutF = new FileOutputStream( F );

  //data to save
  byte[] A = {10, 20, 30, 40,50, 60, 70, 80};

//write the whole byte array at once to the stream
  OutF.write( A );

  //output done, so close the stream
  OutF.close();
 }
}
```

To ensure that all data are saved to a file, close the file at the end of the file access.

Opening a Binary File for Input

Using the FileInputStream class, create a file stream and connect it to a physical disk file to open the file.

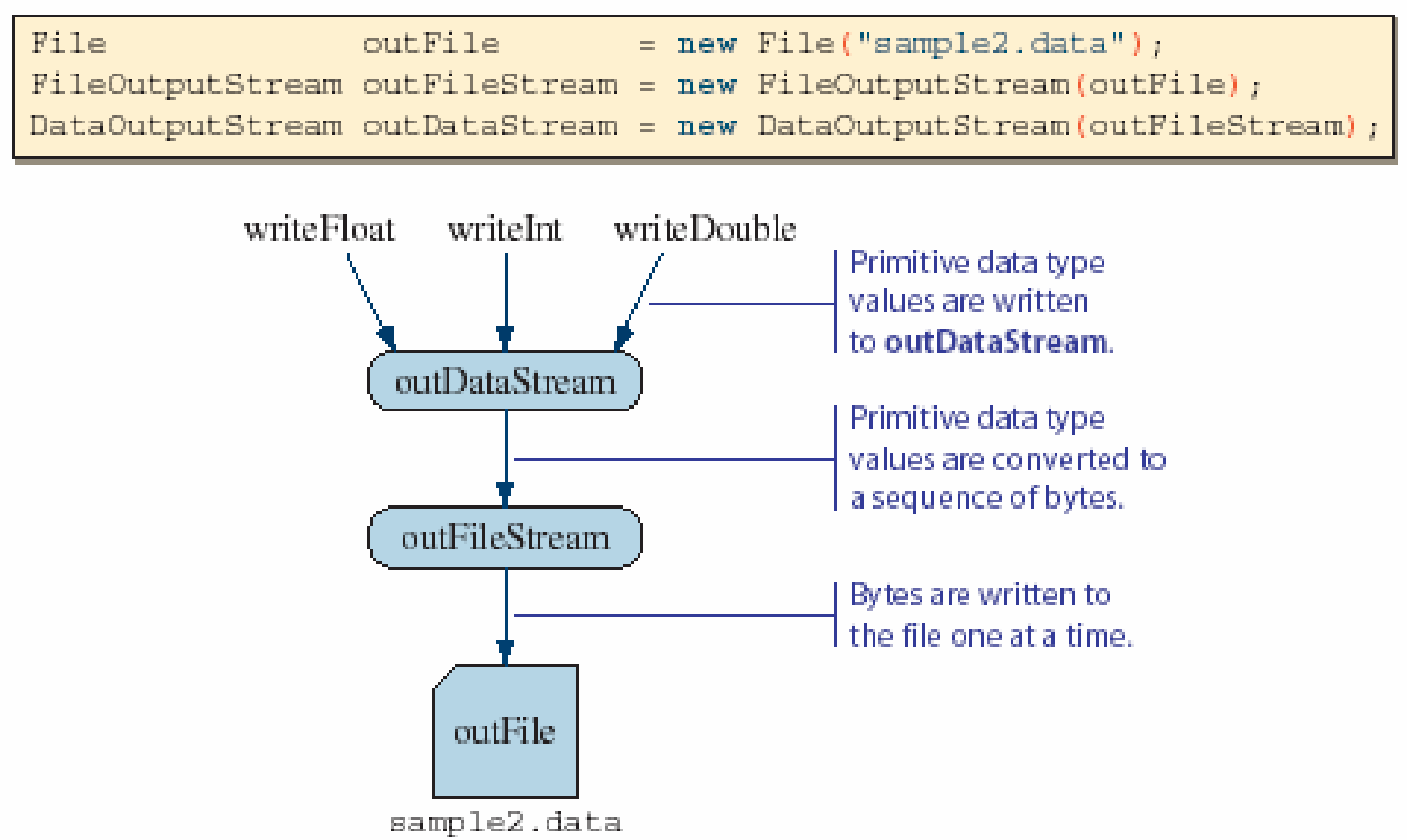
```
Import java.io.*
Class TestFileInputStream {
Public static void main (String [] args) throws IOException
{
 //set up file and stream
  File  G = new File("sample1.data");
  FileInputStream InG = new FileInputStream(G);

  //set up an array to read data in
  int fileSize  = (int)G.length();
  byte[] B = new byte[fileSize];

  //read data in and display them
  InG.read(B);
  for (int i = 0; i < fileSize; i++) {
       System.out.println(B[i]);
    }
  //input done, so close the stream
  InG.close();
 }
}
```

Streams for High-Level File I/O

- FileOutputStream and DataOutputStream are used to output primitive data values
- FileInputStream and DataInputStream are used to input primitive data values
- To read the data back correctly, we must know the order of the data stored and their data types

Setting up DataOutputStream

A standard sequence to set up a DataOutputStream object:



```
File             outFile       = new File("sample2.data");
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```

Primitive data type values are written to **outDataStream**.

Primitive data type values are converted to a sequence of bytes.

Bytes are written to the file one at a time.

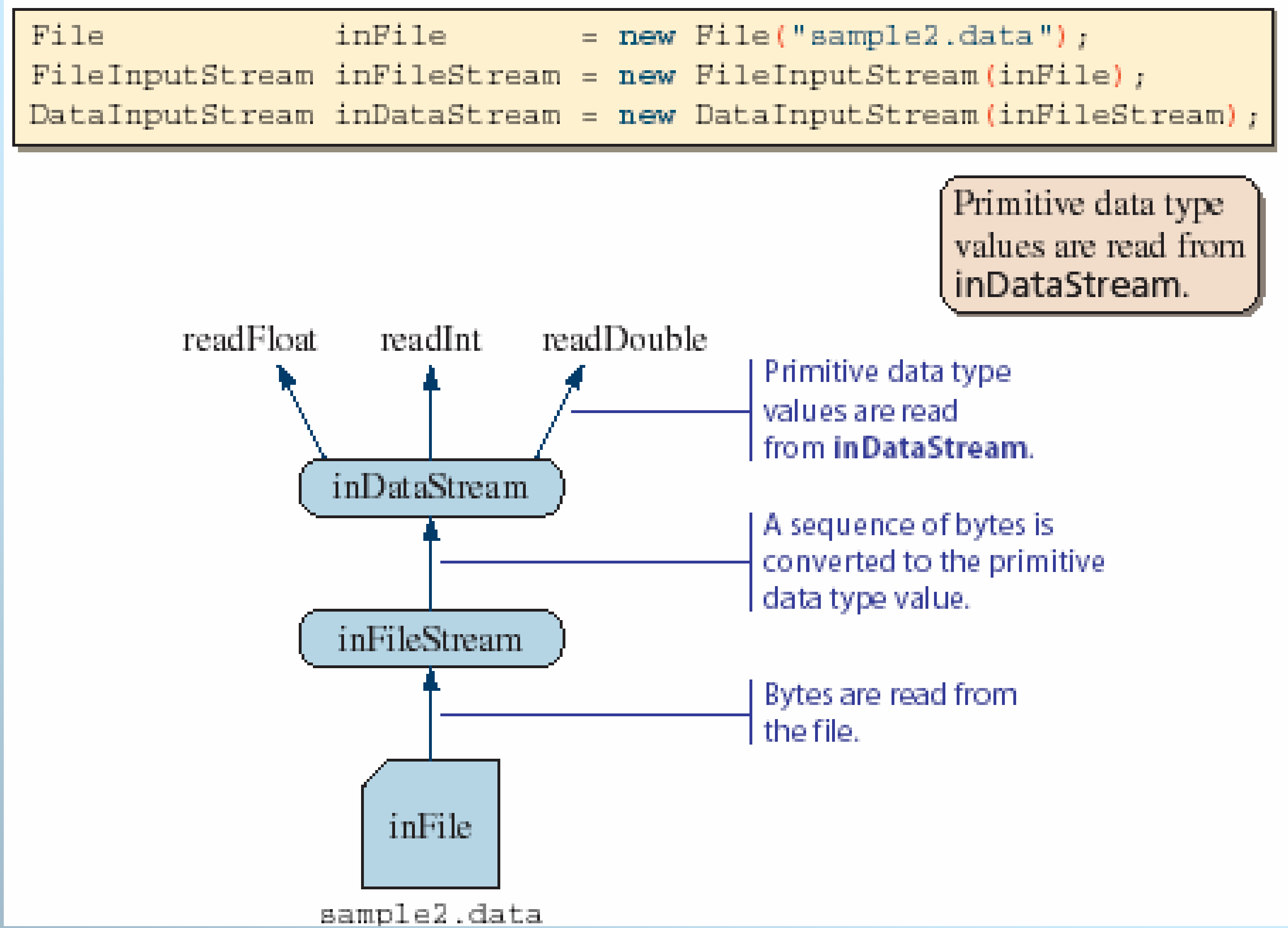
Sample Output

```
import java.io.*;
class TestDataOutputStream {
 public static void main (String[] args) throws IOException {

  //set up file and stream

  File  F   = new File("sample3.data");

  FileOutputStream OutF = new FileOutputStream( F );

  DataOutputStream DF = new   DataOutputStream(OutF);


       //write values of primitive data types to the stream
       DF.writeByte(12);
       DF.writeInt(1234);
       DF.writeLong(9876543);
       DF.writeFloat(1234F);
       DF.writeDouble(1234.4565345);
       DF.writeChar('A');
       DF.writeBoolean(false);

       //output done, so close the stream
       DF.close();
    }
}
```

```
/*========= run============

inside the file "sample3.data" is:

□  □ Ò   –´?Dš@ @“IÓ}Ç«ü A

************************/
```

Setting up DataInputStream

A standard sequence to set up a DataInputStream object:



```
File            inFile       = new File("sample2.data");
FileInputStream inFileStream = new FileInputStream(inFile);
DataInputStream inDataStream = new DataInputStream(inFileStream);
```

Primitive data type values are read from inDataStream.

Primitive data type values are read from **inDataStream**.

A sequence of bytes is converted to the primitive data type value.

Bytes are read from the file.

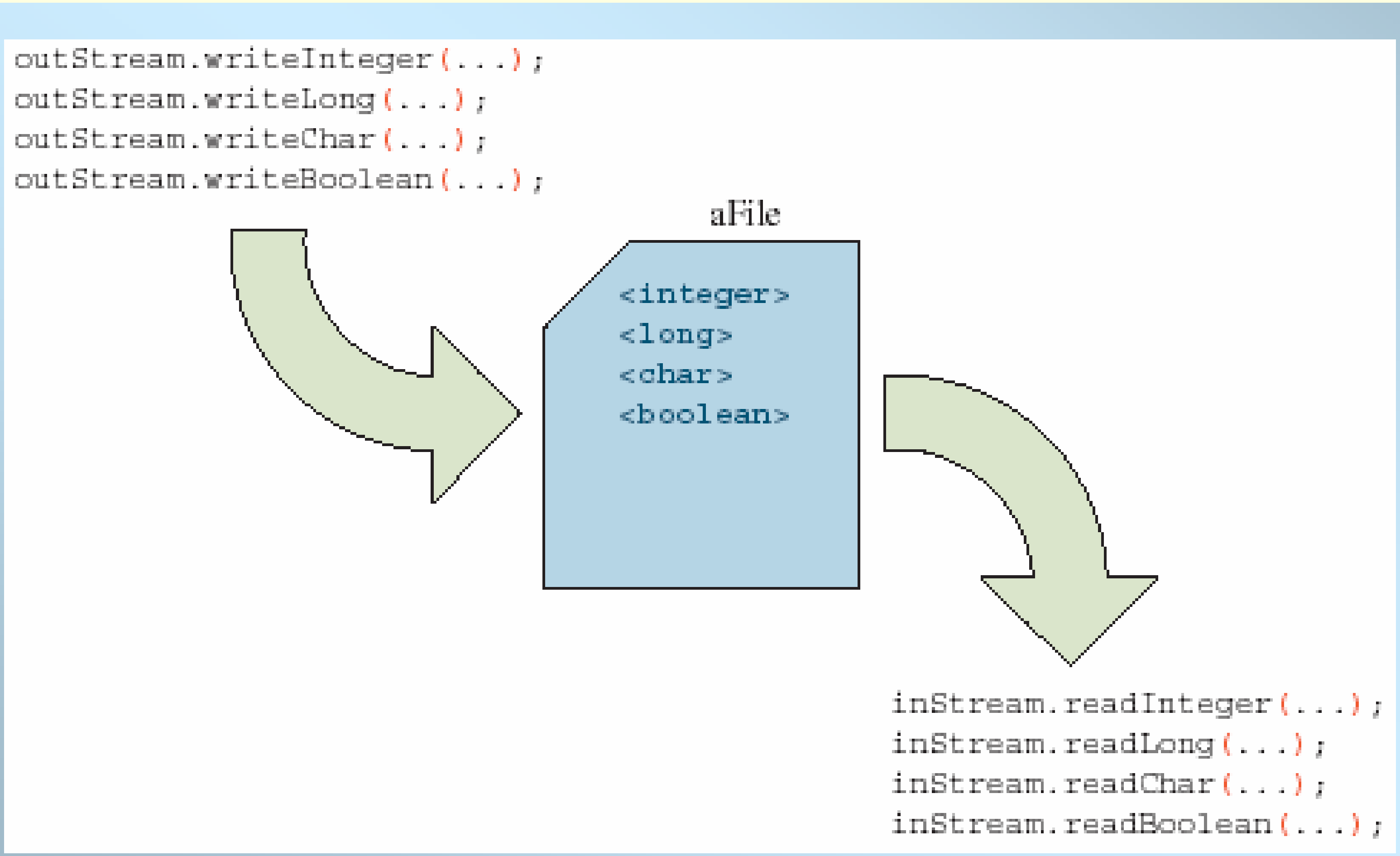
Sample Input

```java
import java.io.*;
class TestDataInputStream {
    public static void main (String[] args) throws IOException {
        //set up inDataStream

        File  G   = new File("sample3.data");

        FileInputStream InF = new FileInputStream( G );

        DataInputStream DF = new   DataInputStream(InF);

        //read values back from the stream and display them
        System.out.println(DF.readByte());
        System.out.println(DF.readInt());
        System.out.println(DF.readLong());
        System.out.println(DF.readFloat());
        System.out.println(DF.readDouble());
        System.out.println(DF.readChar());
        System.out.println(DF.readBoolean());

        //input done, so close the stream
        DF.close();
    }
}
```

```
/*output after reading file sample3.dtat"
12
1234
9876543
1234.0
1234.4565345
A
true
************************
```

Reading Data Back in Right Order

The order of write and read operations must match in order to read the stored primitive data back correctly.



```
outStream.writeInteger(...);
outStream.writeLong(...);
outStream.writeChar(...);
outStream.writeBoolean(...);
```

```
inStream.readInteger(...);
inStream.readLong(...);
inStream.readChar(...);
inStream.readBoolean(...);
```

Textfile Input and Output

- Instead of storing primitive data values as binary data in a file, we can convert and store them as a string data.
  - This allows us to view the file content using any text editor
- To output data as a string to file, we use a **PrintWriter** object.
- To input data from a textfile, we use **FileReader** and **BufferedReader** classes
  - From Java 5.0 (SDK 1.5), we can also use the Scanner class for inputting textfiles

Text File Stream Classes

FileReader	To open a character input stream and connect it to a physical disk file
FileWriter	To open a character output stream and connect it to a physical disk file
BufferedReader	To provide buffering and to read data from an input stream
BufferedWriter	To provide output buffering
PrintWriter	To write character data to an output stream

Sample Textfile Output

A test program to save data to a file using PrintWriter for high-level IO

```
import java.io.*;
class TestPrintWriter {
  public static void main (String[] args) throws IOException {

        //set up file and stream
        File outFile = new File("sample3.data");
        FileOutputStream SF = new FileOutputStream(outFile);
        PrintWriter PF = new PrintWriter(SF);

        //write values of primitive data types to the stream
        PF.println(987654321);
        PF.println("Hello, world.");
        PF.println(true);

        //output done, so close the stream
        PF.close();
  }
}
```

We use println and print with PrintWriter. The print and println methods convert primitive data types to strings before writing to a file.

Sample Textfile Input

To read the data from a text file, we use the FileReader and BufferedReadder objects.

To read back from a text file:

- we need to associate a BufferedReader object to a file,

```
File inF = new File("sample3.data");
FileReader FR = new FileReader(inF);
BufferedReader BFR = new BufferedReader(FR);
```

- read data using the readLine method of BufferedReader,

```
String str;
str = bufReader.readLine();
```

- convert the string to a primitive data type as necessary.

```
int i = Integer.parseInt(str);
```

Sample Textfile Input

```
import java.io.*;
class TestBufferedReader {

public static void main (String[] args) throws IOException
{

   //set up file and stream
   File inF = new File("sample3.data");
   FileReader FR = new FileReader(inF);
   BufferedReader BFR = new BufferedReader(FR);
   String str;
   //get integer
   str = BFR.readLine();
   int i = Integer.parseInt(str);

    //get long
      str = BFR.readLine();
      long l = Long.parseLong(str);

     //get float
      str = BFR.readLine();
      float f = Float.parseFloat(str);
```

```
     //get double
      str = BFR.readLine();
      double d = Double.parseDouble(str);

      //get char
      str = BFR.readLine();
      char c = str.charAt(0);

      //get boolean
      str = BFR.readLine();
      Boolean boolObj = new Boolean(str);
      boolean b = boolObj.booleanValue( );

      System.out.println(i);
      System.out.println(l);
      System.out.println(f);
      System.out.println(d);
      System.out.println(c);
      System.out.println(b);

      //input done, so close the stream
     BFR.close();
    }
}
```

Sample Textfile Input with Scanner

```
import java.util.*;
import java.io.*;
class TestScanner {

    public static void main (String[] args) throws IOException {

        //open the Scanner
     try{
         Scanner input = new Scanner(new File("sample3.data"));
        } catch (FileNotFoundException e) {System.out.println(“Error opening file”);
                                             System. Exit(1);}
        int i = input.nextInt();
        long l = input.nextLong();
        float f = input.nextFloat();
        double d = input.nextDouble();
        char c = input.next().charAt(0);
        boolean b = input.nextBoolean();

        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
        System.out.println(c);
        System.out.println(b);

        input.close();
   }
}
```

We can associate a new Scanner object to a File object. For example:

Scanner scanner = new File (“sample3.data”));

Will associate scanner to the file sample3.data. Once this association is made, we can use scanner methods such as nexInt, next, and others to input data from the file.

The code is the same as TestBufferedReader but uses the Scanner class instead of BufferedReader. Notice that the conversion is not necessary with the Scanner class by using appropriate input methods such as nexInt and nexDouble.

Saving Objects

To save objects to a file, we first create an ObjectOutputStream object. We use the method writeObject to write an object.

```
import java.io.*;
Class TestObjectOutputStream {
 public static void main (String[] args) throws IOException {

    File outFile = new File("objects.data");
    FileOutputStream   outFileStream = new FileOutputStream(outFile);
    ObjectOutputStream outObjectStream = new ObjectOutputStream(outFileStream);
    Person p;
    for (int i =0; i<10; i++) {
       s=input.next();
       p = new Person ();
       p.setName(input.next()+input.nextLine());
       p.setAge(input.nextInt());
       p.setGender(s.charAt(0));


       outObjecttStream.writeObject(p);
     }
      outObjectStream.close();
  }
}
```

Saving Objects

It is possible to save different type of objects to a single file. Assuming the Account and Bank classes are defined properly, we can save both types of objects to a single file:

```
File outFile = new File("objects.data");

FileOutputStream   outFileStream = new FileOutputStream(outFile);

ObjectOutputStream outObjectStream = new ObjectOutputStream(outFileStream);
```

```
Person person = new Person("Mr. Ali", 20, 'M');

outObjectStream.writeObject( person );
```

```
account1    = new Account();
bank1       = new Bank();

outObjectStream.writeObject( account1 );
outObjectStream.writeObject( bank1    );
```

Could save objects from the different classes.

Saving Objects

We can even mix objects and primitive data type values, for example,

```
Account account1, account2;
Bank bank1, bank2;

account1 = new Account();
account2 = new Account();
bank1 = new Bank();
bank2 = new Bank();


outObjectStream.writeInt( 15 );

outObjectStream.writeObject( account1 );

outObjectStream.writeChar( ‘X’ );
```

Reading Objects

To read objects from a file, we use FileInputStream and ObjectInputStream. We use the method readObject to read an object.

```
import java.io.*;
Class TestObjectInputStream {
 public static void main (String[] args) throws IOException {
    File  inFile = new File("objects.data");
    FileInputStream    inFileStream = new FileInputStream(inFile);
    ObjectInputStream inObjectStream = new ObjectInputStream(inFileStream);
    Person p;
    for (int i =0; i<10; i++) {
       p =  (Person) inObjectStream.readObject();
       System.out.println(p.getName() + “    “ + p.getAge() + “  “ +p.getGender());
    }
    inObjectStream.close();
  }
}
```

Reading Objects

If a file contains objects from different classes, we must read them in the correct order and apply the matching typecasting. For example, if the file contains two Account and two Bank objects, then we must read them in the correct order:

```
account1 = (Account) inObjectStream.readObject( );

account2 = (Account) inObjectStream.readObject( );
bank1 = (Bank) inObjectStream.readObject( );

bank2 = (Bank) inObjectStream.readObject( );
```

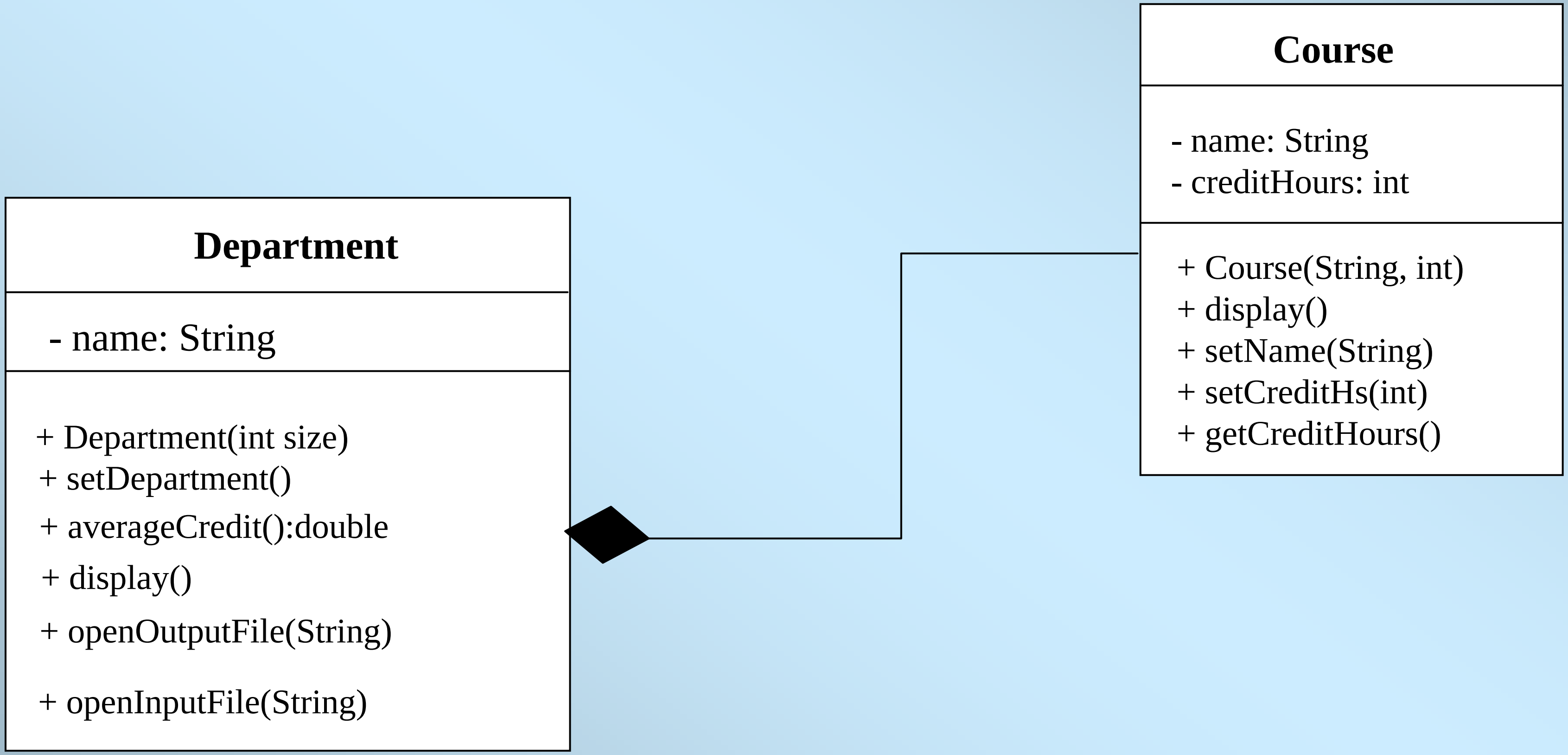
Saving and Loading Arrays

- Instead of processing array elements individually, it is possible to save and load the whole array at once.

```
Person[] p = new Person[ N ];
           //assume N already has a value

//build the people array
. . .
//save the array
outObjectStream.writeObject ( p );
```

```
//read the array

Person[ ] p = (Person[]) inObjectStream.readObject( );
```

Example: Class Department



Department

- name: String

+ Department(int size)
+ setDepartment()
+ averageCredit():double
+ display()
+ openOutputFile(String)
+ openInputFile(String)

Course

- name: String
- creditHours: int

+ Course(String, int)
+ display()
+ setName(String)
+ setCreditHs(int)
+ getCreditHours()

Example: Class Department

Implementation of Class Course

```
import java.io.*;
public class Course implements Serializable
{
  private String name;
  private int creditHours;
  public Course (String na, int h)
  {
    name=na;
    creditHours=h;
  }
  public void display()
  {
    System.out.println("Name : "+name);
    System.out.println("Credit Hours : "+ creditHours);
 }
```

```
  public void setName(String na)
 {
   name=na;
  }
  public void setCreditHs(int h)
 {
    creditHours=h;
  }
  public double getCreditHours()
 {
  return creditHours;
  }
}
```

Example: Class Department

Implementation of Class Department

```
import java.io.*;
import java.util.Scanner;
public class Department
{
  private String name;
  private Course []c;
  public Department(int size)
  {
     name= " ";
     c= new Course[size];
  }
```

```
public void setDepartment()
  {
    Scanner input = new Scanner(System.in);
    System.out.print("Please enter the name of Department :");
    name =input.next()+input.nextLine();
   for (int i=0; i<c.length; i++)
  {
    System.out.print("Please enter the name of the course :");
   c[i]=new course();
    c[i].setName(input.next()+ input.nextLine());
    System.out.print("Please enter the credit hours : ");
    c[i].setCreditHs(input.nextInt());
    }
  }
```

Example: Class Department

Implementation of Class Department

```
 public void openOutputFile(String fileName) throws
IOException
{
  File  f = new File(fileName);
  FileOutputStream  g = new FileOutputStream(f);
  ObjectOutputStream obj = new ObjectOutputStream(g);
  obj.writeBytes(name);
  obj.writeObject(c);
  obj.close();
 }
```

```
public void openInputFile(String fileName) throws
                              ClassNotFoundException, IOException
{
  File  f = new File(fileName);
  FileInputStream  g = new FileInputStream(f);
 ObjectInputStream obj = new ObjectInputStream(g);
 name=obj.readLine();
 c = (Course [])obj.readObject();
 obj.close();
}
```

Example: Class Department

Implementation of Class Department

```
public double averageCredit()
{
  double s=0.0;
  for (int i=0; i<c.length; i++)
  s+=c[i].getCreditHours();
  return (s/c.length);
 }
 public void display()
{
  System.out.println("==========================");
  System.out.println("The name of the department is :" + name);
  for (int i=0; i<c.length; i++)
      c[i].display();
  System.out.println("The average of credit hours is :" + averageCredit());
                    }
}
```

Implementation of DepartmentTest1

```
import java.io.*;
public class DepartmentTest1
{
  public static void main(String[] args) throws IOException
 {
   Department dep = new Department(3);
   dep.setDepartment();
  dep.openOutputFile("computer.data");
   Department dep2 = new Department(2);
   dep2.setDepartment();
   dep2.openOutputFile("engineering.data");
  }
}
```

```
/*    run
Please enter the name of Department :Computer science
Please enter the name of the course :csc107
Please enter the credit hours : 3
Please enter the name of the course :csc112
Please enter the credit hours : 3
Please enter the name of the course :csc113
Please enter the credit hours : 4
Please enter the name of Department :Engineering
Please enter the name of the course :eng123
Please enter the credit hours : 4
Please enter the name of the course :eng125
Please enter the credit hours : 3
*/
```

Implementation of DepartmentTest2

```
import java.io.*;
public class DepartmentTest2
{
  public static void main(String[] args) throws
                          ClassNotFoundException, IOException
 {
   Department d1 = new Department(3);
  d1.openInputFile("computer.data");
  d1.display();
  Department d2 = new Department(2);
  d2.openInputFile("engineering.data");
  d2.display();
  }
}
```

```
/*
=========================================
The name of the department is :Computer science
Name : csc107
Credit Hours : 3
Name : csc112
Credit Hours : 3
Name : csc113
Credit Hours : 4
The average of credit hours is :3.3333333333333335
=========================================
The name of the department is :Engineering
Name : eng123
Credit Hours : 4
Name : eng125
Credit Hours : 3
The average of credit hours is :3.5
*/
```

Chapter 9

Data Structures: Linked Lists

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

Dr. S. HAMMAMI

Objectives

After you have read and studied this chapter, you should be able to:

- Understand the concept of a dynamic data structure.
- Be able to create and use dynamic data structures such as linked lists.
- Understand the stack and queue ADTs.
- Various important applications of linked data structures.
- Know how to use inheritance to define extensible data structures.
- Create reusable data structures with classes, inheritance and composition.

Outline

1. **Introduction**
2. **Self-Referential Classes**
   - 2.1. Definition
   - 2.2. Generic Node Class
   - 2.3. Example
   - 2.4. Implementation of Generic Class Node
   - 2.5. Connecting two nodes
   - 2.6. Examples

3. **Linked Lists**
   - 3.1. Definition
   - 3.2. Graphical representation
   - 3.3. Performance
   - 3.4. Single Linked List
   - 3.5. Basics Methods of Linked List: Implementation
   - 3.6. Examples

1. Introduction

- A *data structure* is organizes information so that it efficient to access and process.
- An *array* is a *static* structure -- it can’t change size once it is created.
- A *vector* is a *dynamic* structure -- it can grow in size after creation.
- In this chapter we study several dynamic data structures -- *lists*, *queues*, and *stacks*.

2. Self-Referential Classes: Definition

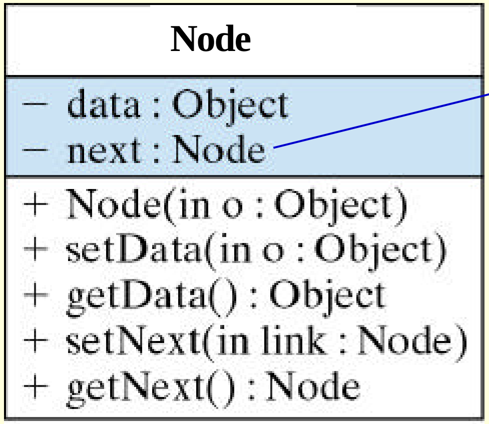
- **Self-referential class**

  Contains an instance variable that refers to another object of the same class type

  That instance variable is called a link

  A `null` reference indicates that the link does not refer to another object

2. Self-Referential Classes (cont)

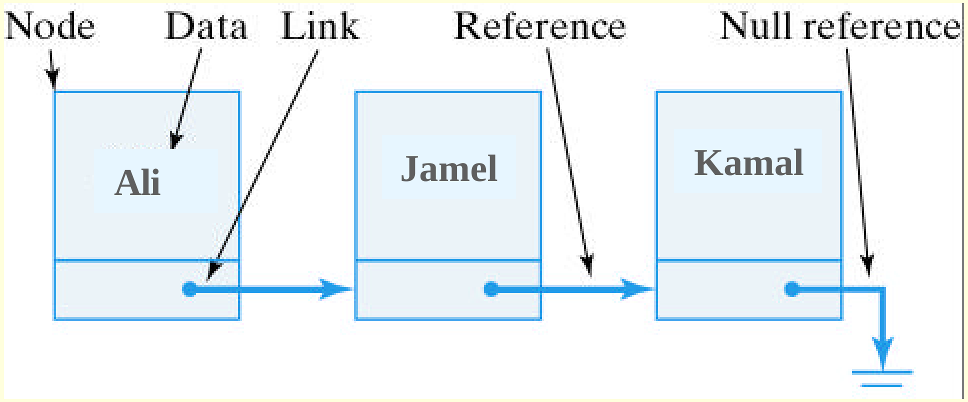


Node

- data : Object
- next : Node

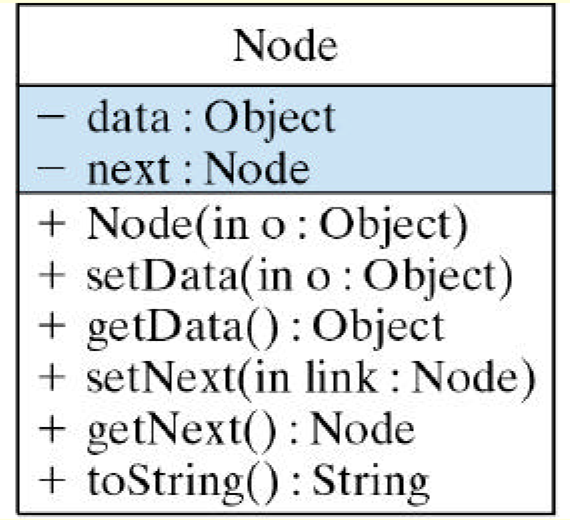
+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node

A *link* to another Node object.



Basic Node: The Generic Node Class

- A node in a linked list contains data elements and link elements.



Node

− data : Object
− next : Node

+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node
+ toString() : String

Generic Node Class: Implementation

```
public class Node {
    private Object data;  // Stores any kind of data
    private Node next;

    public Node(Object obj) {  // Constructor
        data = obj;
        next = null;
    }

  // Link access methods
    public void setNext( Node nextPtr ) {
        next = nextPtr;
    }

    public Node getNext() {
        return next;
    }
} // Node
```

```
// Data access methods
public void setData(Object obj)
{
    data = obj;
}

public Object getData() {
    return data;
}

public String toString() {
    return data.toString();
}
```

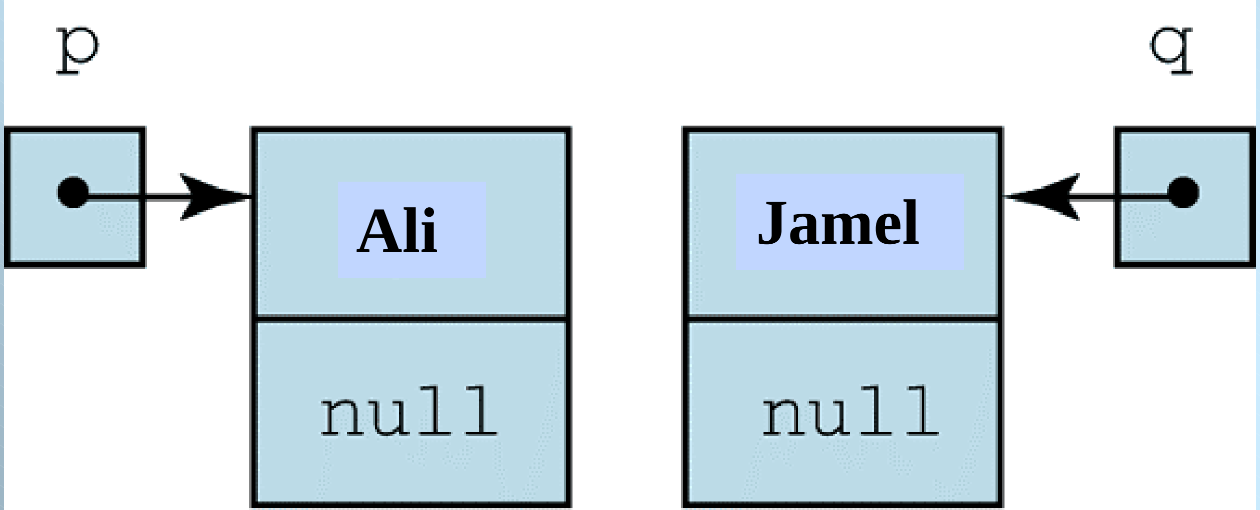
Connecting two nodes

The statements

```
Node p = new Node(“Ali");
Node q = new Node(“Jamel");
```

allocate storage for two objects of type **Node** referenced by **p** and **q**. The node referenced by **p** stores the string **“Ali"**, and the node referenced by **q** stores the string **“Jamel"**. The **next** fields of both nodes are **null**.

Nodes referenced by p and q



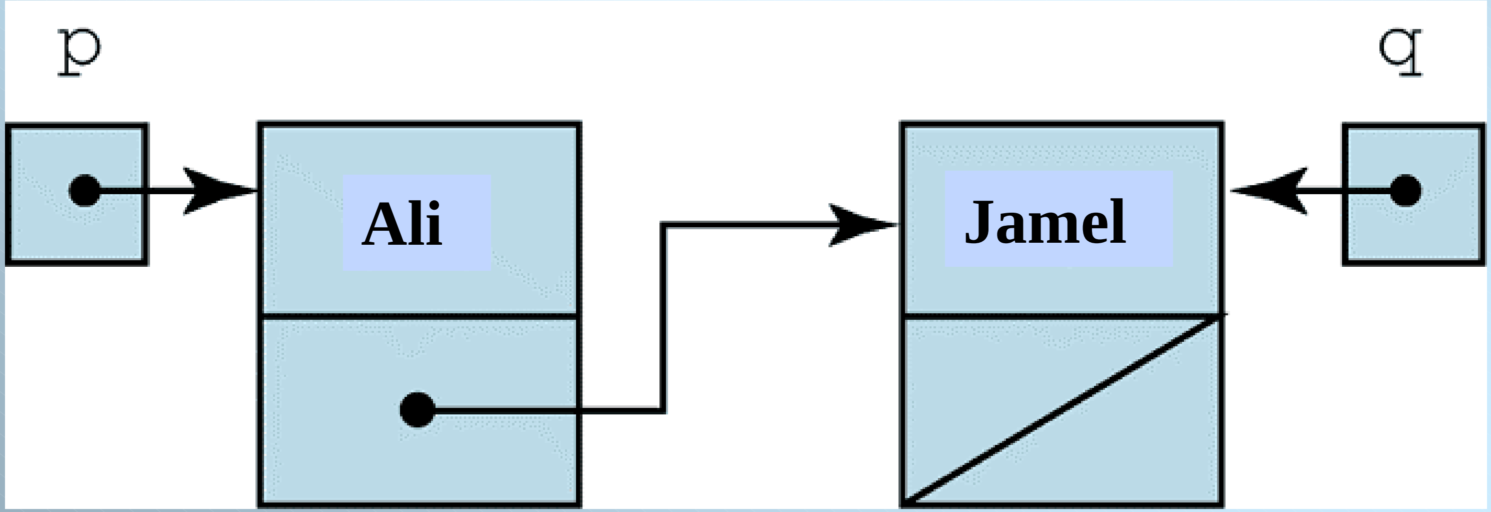
Connecting two nodes (Cont)

The statement

p.next = q;

stores the address of node **q** in the link field of node **p**, thereby connecting node **p** to node **q**, and forming a linked list with 2 nodes. The diagonal line in the **next** field of the second list node indicates the value **null**.

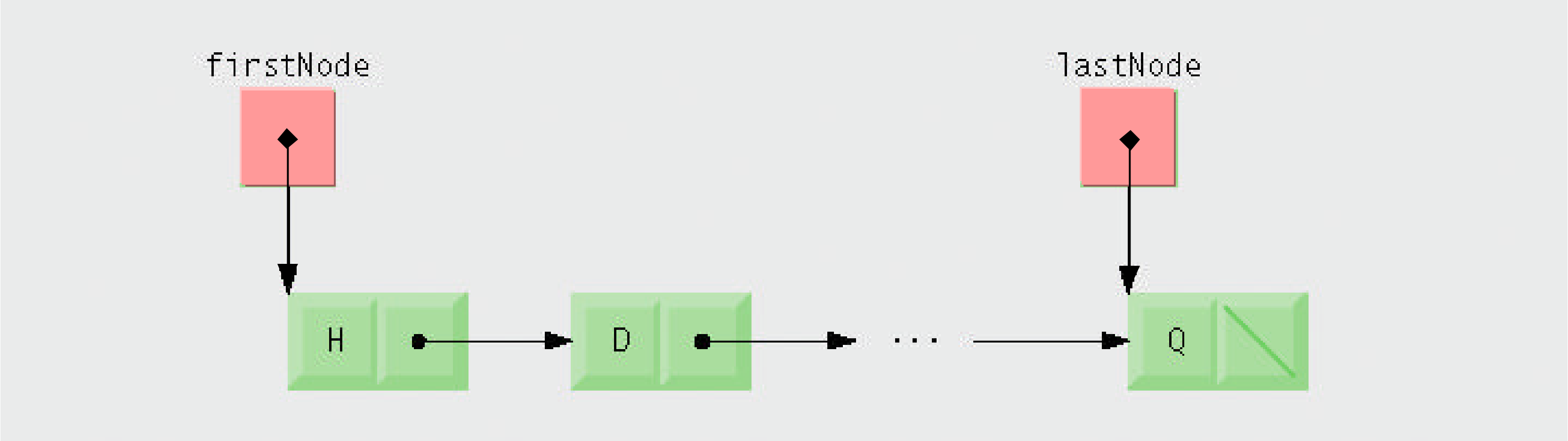
Linked list with two nodes



3. Linked Lists: Definition

- Linked list
  - Linear collection of nodes
    - A *linked list* is based on the concept of a **self-referential object** -- an object that refers to an object of the same class.
  - A program typically accesses a linked list via a reference to the first node in the list
    - A program accesses each subsequent node via the link reference stored in the previous node
  - Are dynamic
    - The length of a list can increase or decrease as necessary
    - Become full only when the system has insufficient memory to satisfy dynamic storage allocation requests

Linked list graphical representation.



Linked Lists: Performance

- An array can be declared to contain more elements than the number of items expected, but this wastes memory. Linked lists provide better memory utilization in these situations. Linked lists allow the program to adapt to storage needs at runtime.

- Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point). All existing node objects remain at their current locations in memory.

- Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

Single Linked List & Doubly Linked List

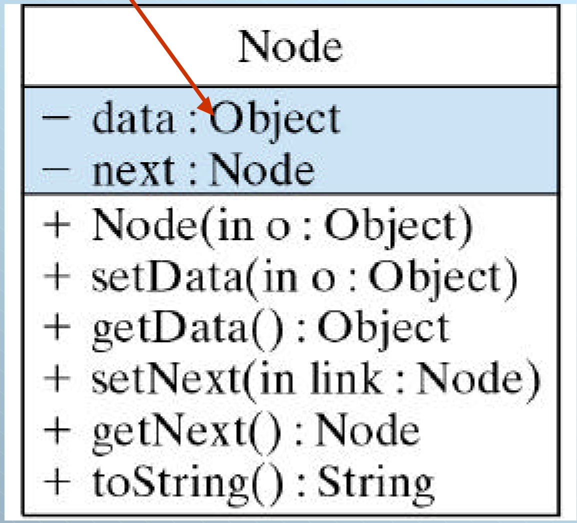
- Singly linked list
  - Each node contains one reference to the next node in the list (Example)

- Doubly linked list
  - Each node contains a reference to the next node in the list and a reference to the previous node in the list (Example)

  - `java.util`'s `LinkedList` class is a doubly linked list implementation

The Generic List Class: Implementation

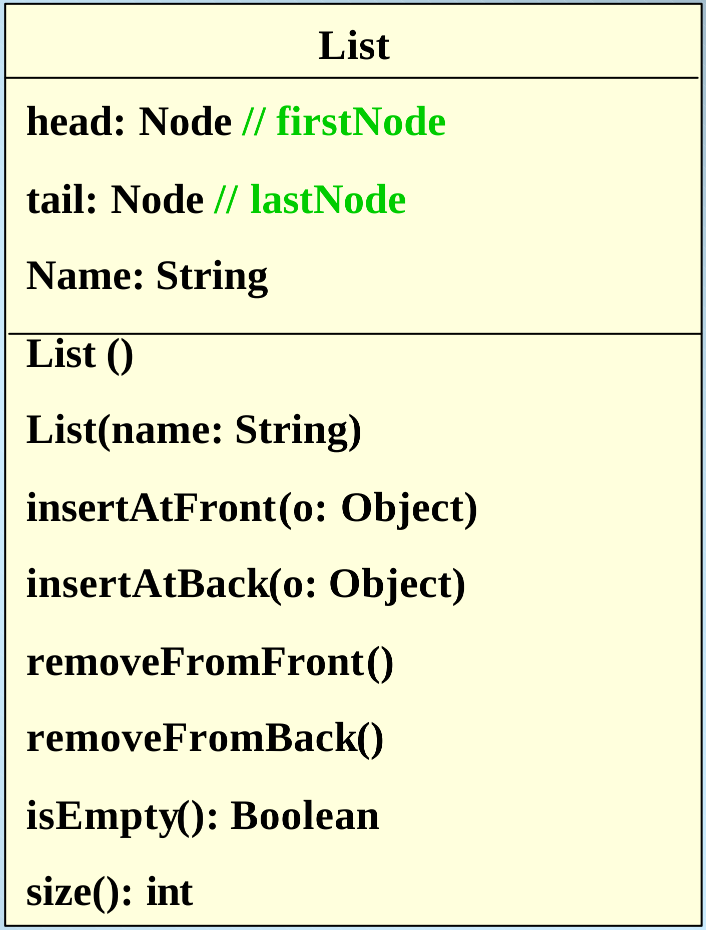
The data field is an Object reference, so it can refer to any object.



Node

− data : Object
− next : Node

+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node
+ toString() : String



List

head: Node // firstNode

tail: Node // lastNode

Name: String

List ()

List(name: String)

insertAtFront(o: Object)

insertAtBack(o: Object)

removeFromFront()

removeFromBack()

isEmpty(): Boolean

size(): int

The Generic List Class: Implementation (Cont)

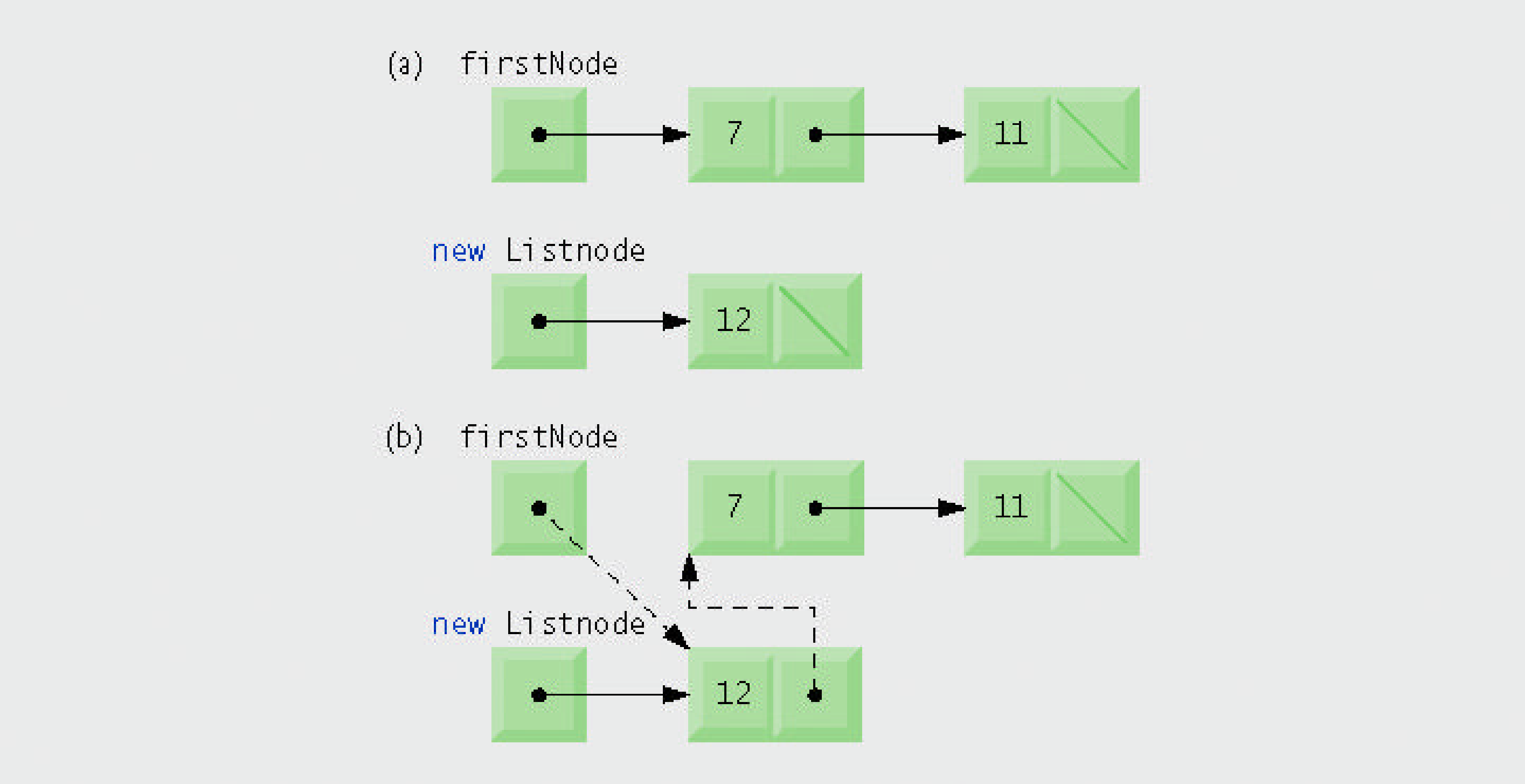
```
public class List {
    private Node head;
    private Node tail;
    public List() {
        head = null;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public void print() {  }

    public void insertAtFront( Object newObj ) { }

    public void insertAtBack( Object newObj ) { }

    public Object removeFromFirst() { }

    public Object removeFromLast() { }
    ………
} // List
```

Linked List: insertAtFront

- **Method insertAtFront's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, set firstNode to a new ListNode object and initialize that object with insertItem and firstNode
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to the ListNode passed as argument, which previously was the first node

Linked List: insertAtFront (Cont)

Graphical representation of operation `insertAtFront`



Linked List: insertAtFront (Cont)

Code of `insertAtFront`

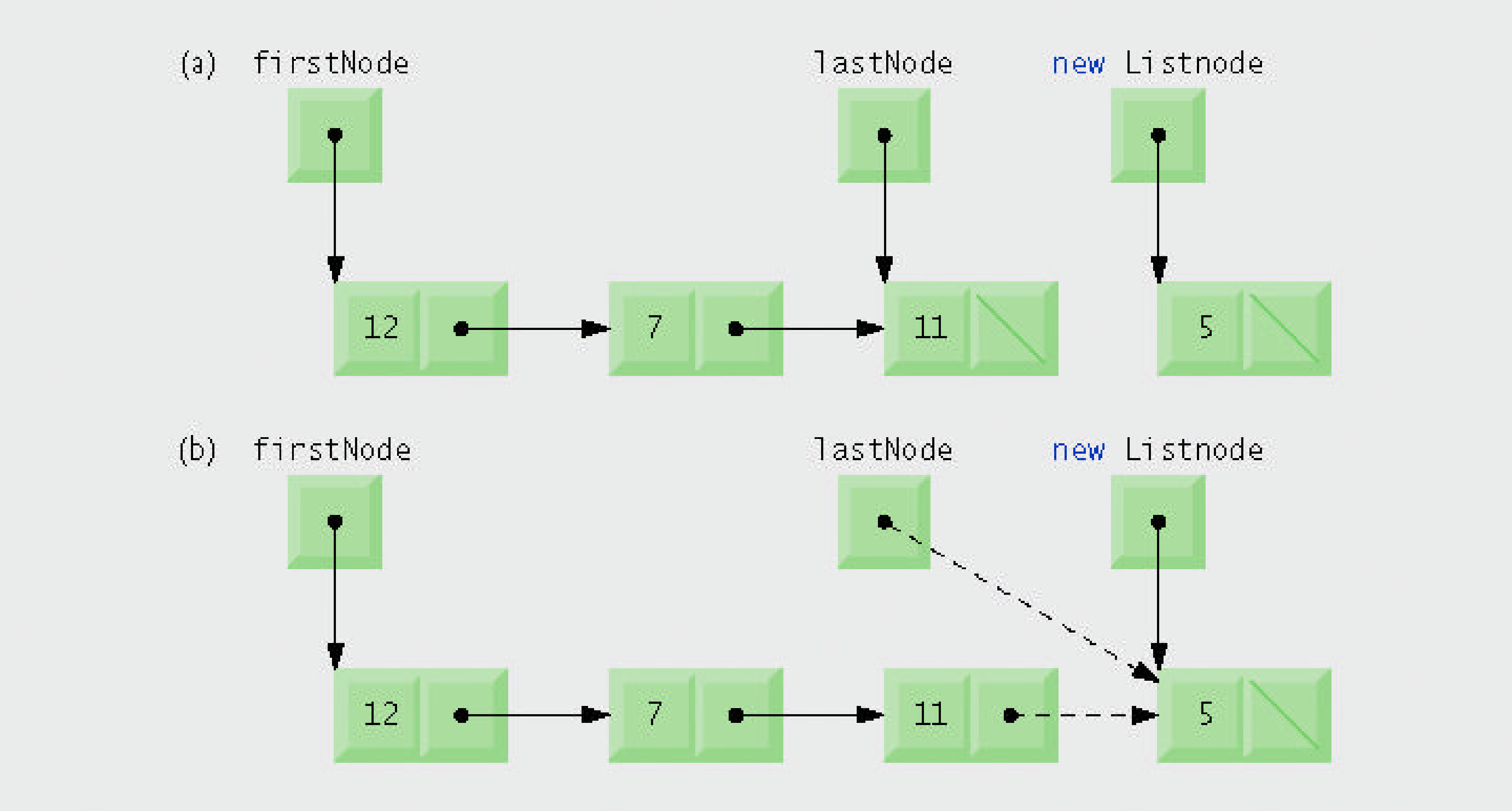
```
public void insertAtFront(Object obj) {

   Node newnode =  new Node(obj);

   newnode.setNext(head);

   if(isempty())
        head=tail= newnode;

   else

       head = newnode;

} // insertAtFront()
```

Linked List: insertAtBack

- **Method insertAtBack's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, assign to lastNode and lastNode.nextNode the reference to the new ListNode that was initialized with insertItem
    - The ListNode constructor sets data to refer to the insertItem passed as an argument and sets reference nextNode to null

Linked List: insertAtBack (Cont)

Graphical representation of operation insertAtBack.



Linked List: insertAtBack (Cont)

Code of insertAtBack

```
public void insertAtBack(Object obj) {
    if (isEmpty())
        head = tail = new Node(obj);
    else {
        Node current = head;                // Start at head of list
        while (current.getNext() != null)   // Find the end of the list
            current = current.getNext();
        current.setNext(new Node(obj));    // Insert the newObj
    }
} // insertAtRear
```

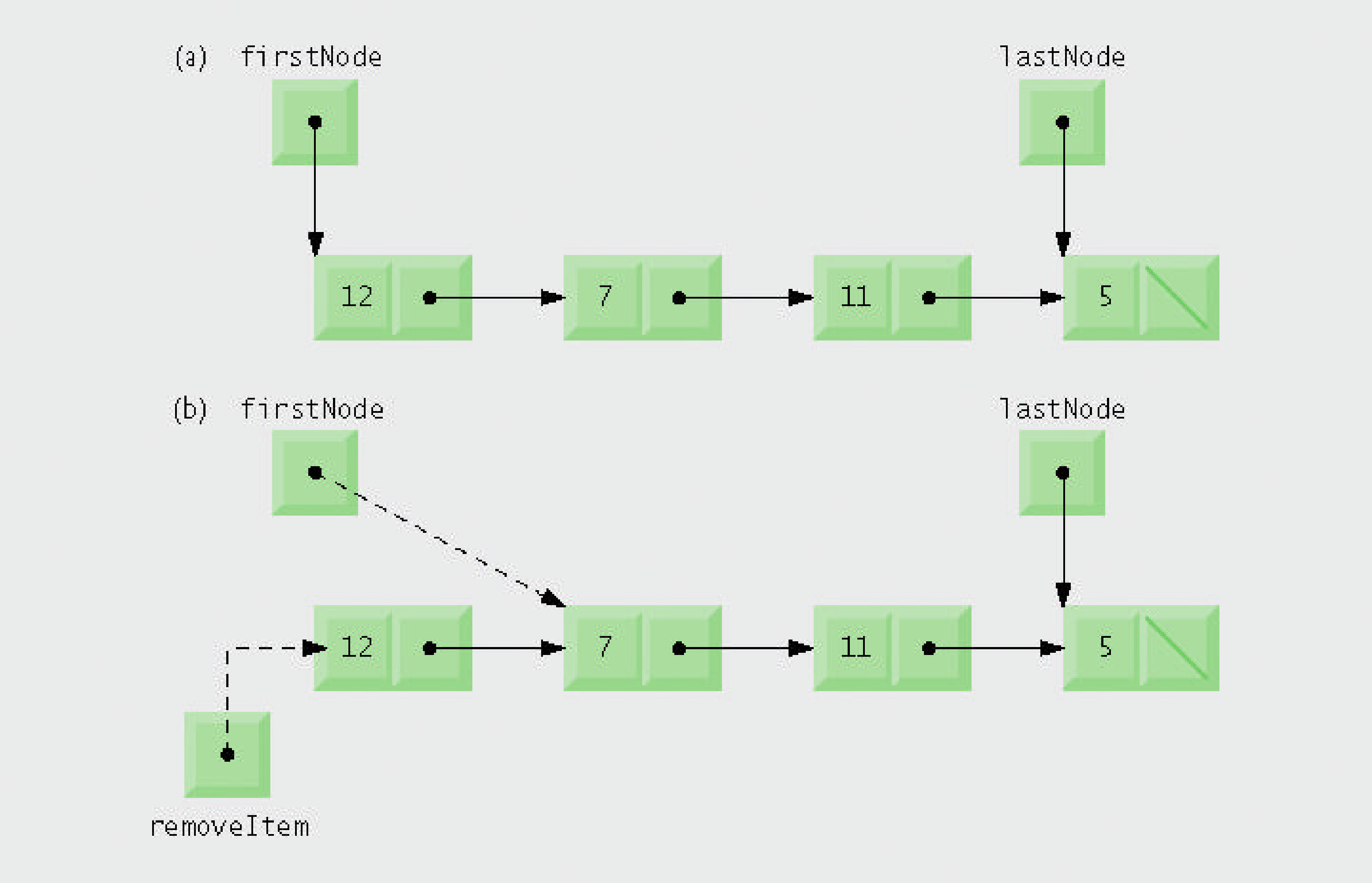
Other solution using the tail

```
public void insertAtBack(Object obj) {
      if(isempty())
          head = tail = new Node(obj);
      else{
          Node newnode =  new Node(obj);
          tail.setNext(newnode);
          tail=newnode;
      }
}
```

Linked List: removeFromFront

- **Method removeFromFront's steps**
  - Throw an EmptyListException if the list is empty
  - Assign firstNode.data to reference removedItem
  - If firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, assign the value of firstNode.nextNode to firstNode
  - Return the removedItem reference

Linked List: removeFromFront



Graphical representation of operation removeFromFront.

Linked List: removeFromFront

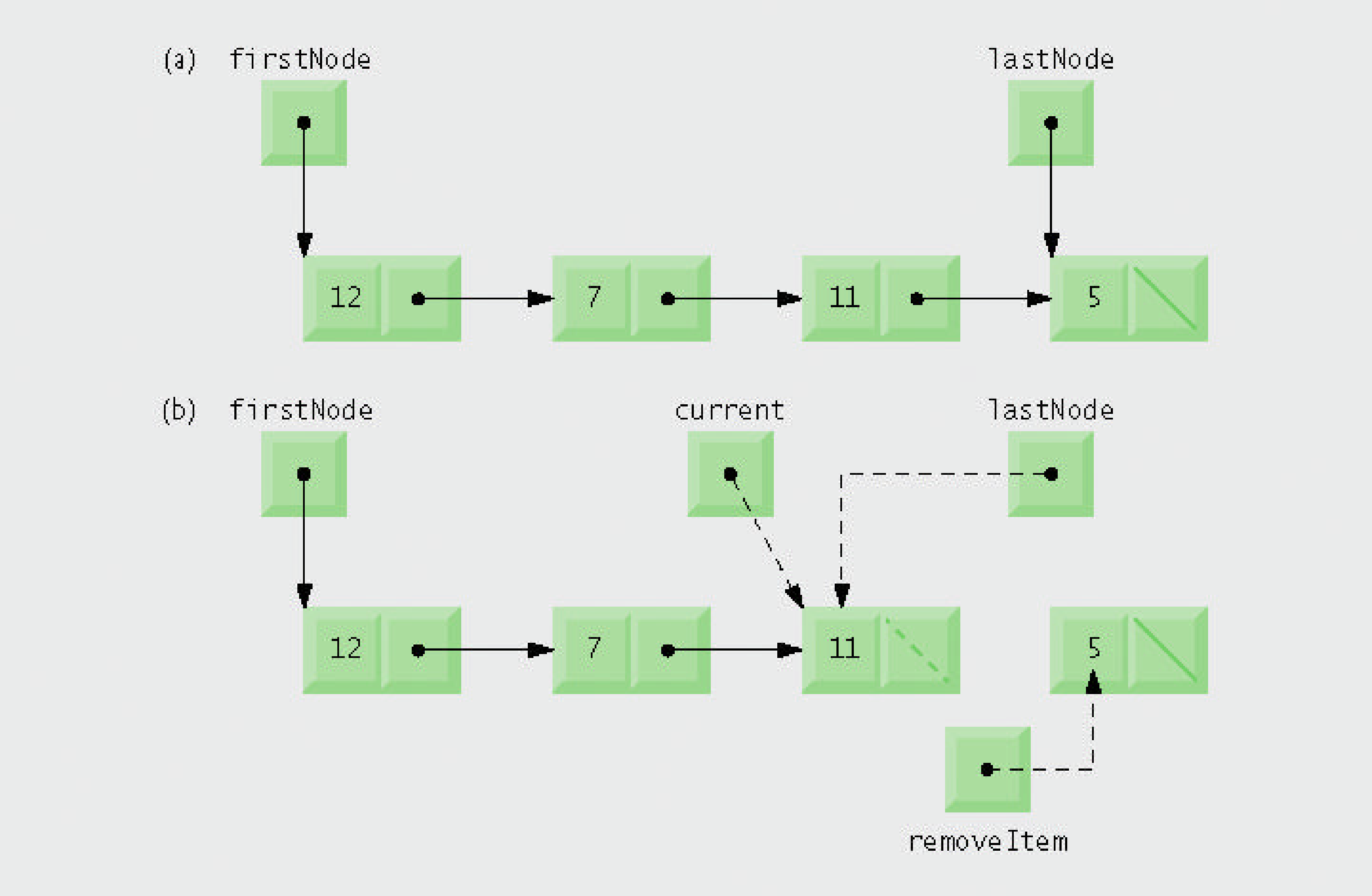
Code of removeFromFront

```
public Object removeFromFrontt() {
    if (isEmpty())
        return null;
    Node first = head;
    if head == tail
          head = tail = null;
    head = head.getNext();
    return first.getData();
}
```

Linked List: removeFromBack

- **Method removeFromBack's steps**
  - Throws an EmptyListException if the list is empty
  - Assign lastNode.data to removedItem
  - If the firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, create the ListNode reference current and assign it firstNode
  - "Walk the list" with current until it references the node before the last node
    - The while loop assigns current.nextNode to current as long as current.nextNode is not lastNode

Linked List: removeFromBack



Graphical representation of operation removeFromBack.

- –Assign `current` to `lastNode`
- –Set `current.nextNode` to `null`
- –Return the `removedItem` reference

Linked List: removeFromBack

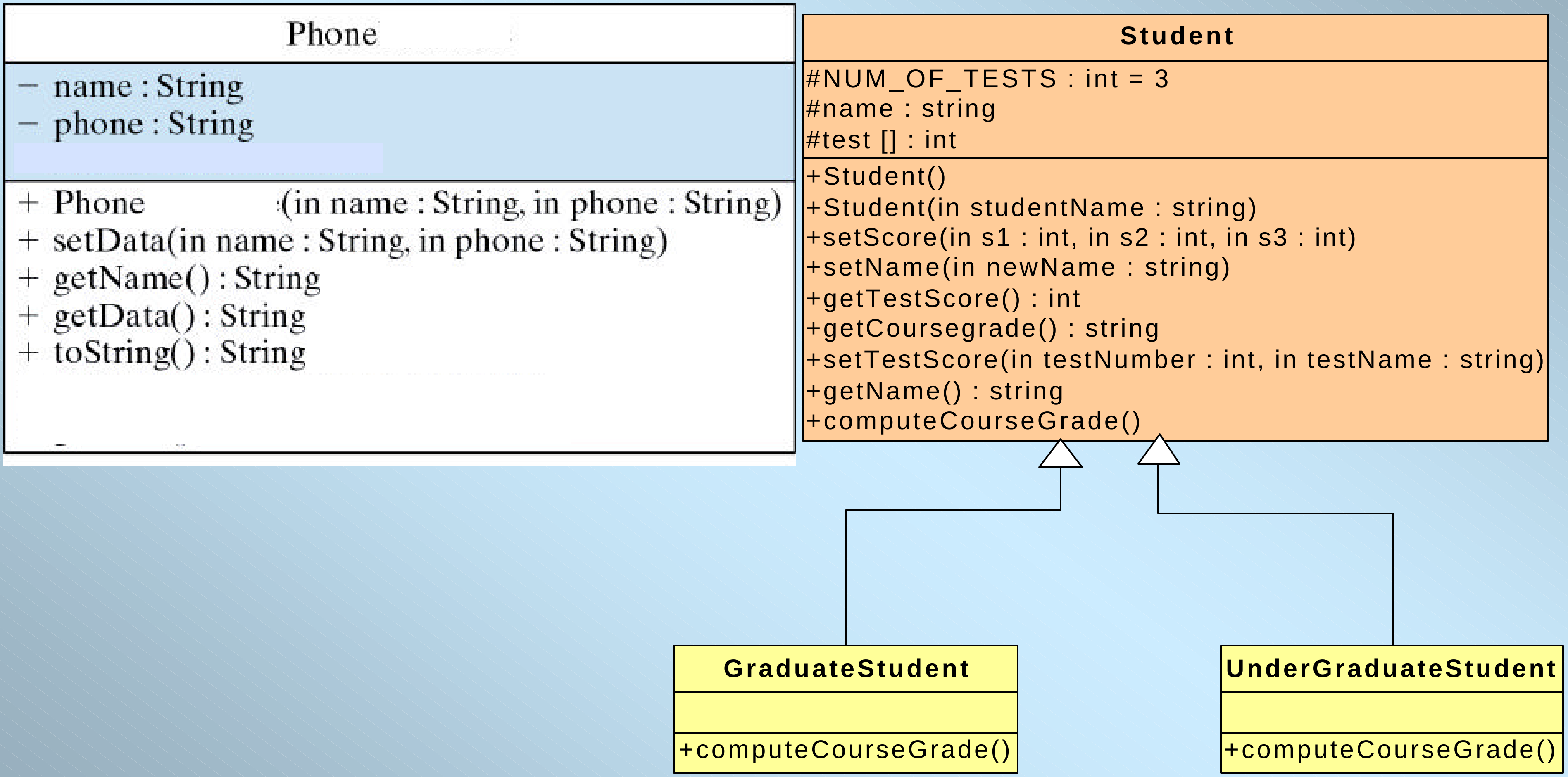
Code of removeFromBack

```
public Object removeFromBack() {
    if (isEmpty())  // Empty list
        return null;

    Node current = head;
    if (current.getNext() == null) { // Singleton list
        head = tail = null;
        return current.getData();
    }

    Node previous = null;            // All other cases
    while (current.getNext() != null) {
        previous = current;
        current = current.getNext();
    }
    previous.setNext(null);
    tail = previous;
    return current.getData();
} // removeLast()
```

Linked List: size

```
public int size()  {
    if(isempty())  return 0;
    Node current = head;;
    int c=1;
    while(current.getNext()!=null){
             current=current.getNext();
             c++;
    }
    return c;
}
```

Example: Create list and insert heterogeneous nodes



Phone

− name : String
− phone : String

+ Phone (in name : String, in phone : String)
+ setData(in name : String, in phone : String)
+ getName() : String
+ getData() : String
+ toString() : String

Student

#NUM_OF_TESTS : int = 3
#name : string
#test [] : int

+Student()
+Student(in studentName : string)
+setScore(in s1 : int, in s2 : int, in s3 : int)
+setName(in newName : string)
+getTestScore() : int
+getCoursegrade() : string
+setTestScore(in testNumber : int, in testName : string)
+getName() : string
+computeCourseGrade()

GraduateStudent

+computeCourseGrade()

UnderGraduateStudent

+computeCourseGrade()

Testing the List ADT

```
Public class Test {
  public static void main( String argv[] ) {
                // Create list and insert heterogeneous nodes
    List list = new List();

    Student s1 =new Student("Saad");

    s1.setScore(10,20,15);

    s1.computeCourseGrade();


    list.insertAtFront(s1);
    list.insertAtFront(new Phone(“Ali M", "997-0020"));
    list.insertAtFront(new Integer(8647));
    list.insertAtFront(new String("Hello World"));


    System.out.println("Generic List"); // Print the list
    list.print();
                 // Remove objects and print resulting list
    Object o;
    o = list.removeFromBack();
    System.out.println(" Removed " + o.toString());
    System.out.println("Generic List:");
    list.print();
    o = list.removeFromFirst();
    System.out.println(" Removed " +o.toString());
    System.out.println("Generic List:");
    list.print();
   } // main()
}
```

Example: Node with data Student

```java
public class Node
{
   public Student data;
   public Node nextNode;

   public Node(Student object )
   {
      this( object, null );
   }

   public Node(Student object, Node node)
   {
      data = object;
      nextNode = node;
   }
```

```java

   public Student getData()
   {
      return data;
   }

   public  Node getNext()
   {
      return nextNode;
   }
} // end class Node
```

Class Student

```
class Student
{
  private final static int NUM_OF_TESTS = 3;
  private   String           name;
  private   int[]              test;
  private   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 void computeCourseGrade()
   {if (getTotal() >= 50)
     courseGrade = "Pass";
    else { courseGrade = "NoPass";
    }
public int getTestScore(int testNumber) {
  return test[testNumber-1];  }
public void setName(String newName) {
    name = newName;  }
public void setTestScore(int tN, int tS)
   {         test[tN-1]=tS;  }
public int getTotal()
{ int total = 0;
  for (int i = 0; i < NUM_OF_TESTS; i++) {
    total += test[i]; }
  return total;
}
public void display()
{System.out.print("The student "+ name +" has
                 "+getTotal()+ " marks");
 System.out.println("  and  Course grade = "+
                          courseGrade);
}
}
```

Class List

```
// class List definition
public class List
{  private Node firstNode;
   private Node lastNode;
   private String name;
   public List()
   { this( "list" );
   }
   public List(String listName )
   {  name = listName;
      firstNode = lastNode = null;
   }
   public void insertAtFront(Student stud )
   {if ( isEmpty() )
      firstNode = lastNode = new Node(stud);
    else
     firstNode = new Node(stud, firstNode );
   }
public void insertAtBack(Student stud)
 {if ( isEmpty() )
    firstNode = lastNode = new Node(stud);
  else
   lastNode=lastNode.nextNode = new
Node(stud);
 }
```

```
public Student removeFromFront()
   { Student st = firstNode.data;
     if ( firstNode == lastNode )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;
      return st;
   }
public Student removeFromBack()
   {Student st = lastNode.data;
    if ( firstNode == lastNode )
         firstNode = lastNode = null;
    else
      { Node current = firstNode;
        while ( current.nextNode != lastNode )
            current = current.nextNode;
        lastNode = current;
         current.nextNode = null;
      }
    return st;
   }
```

Class List

```
public boolean isEmpty()
{ return firstNode == null; } // End isEmpty
public void print()
{if ( isEmpty() )
 {System.out.println("The list" + name +"
                     is empty");
  return; }
  System.out.println( "\n" );
  System.out.println( "The list : "+ name+
                     " contains : " );
  Node current = firstNode;
  while ( current != null )
  {current.data.display();
   current = current.nextNode;
   }
} // End method print

public int maximumMarks()
{if ( isEmpty() )
 {System.out.println("The list" + name +" is
                     empty");
  return -1;}
```

```
 int max=firstNode.data.getTotal();
  Node current = firstNode.nextNode;
  while ( current != null )
  {if (max < current.data.getTotal())
    max =current.data.getTotal();
   current = current.nextNode;
  }
  return max;
} // End method maximumMarks
public double averageMarks()
{if ( isEmpty() )
 {System.out.println("The list" + name +"
            is empty");
   return 0.0;}
 int sum=0, counter=0;
 Node current = firstNode;
 while ( current != null )
{sum+=current.data.getTotal();
 counter++;
 current = current.nextNode;
}
 return 1.0*sum/counter;
} // End method averageMarks
```

Class List

```
//=== this method computes the number of passed or NotPassed student
   public int numberOfPassedOrNotPassedStundent(String ss)
   {
     if ( isEmpty() )
     {
       System.out.println("The list" + name +" is empty");
       return -1;
      }
    int nb=0;
    Node current = firstNode;
    while ( current != null )
    {
     if(current.data.getCourseGrade().equals(Pass))
       nb++;
     current = current.nextNode;
    }
    return nb;
 }
} // end class List
```

Testing the List ADT

```
public class ListStudentTest
{public static void main(String args[])
 { List ob = new List("csc");
   Student s1 =new Student("Saad");
   s1.setScore(10,20,15);
   s1.computeCourseGrade();

   Student s2 =new Student("Ali");
   s2.setScore(10,50,40);
   s2.computeCourseGrade();

   Student s3 =new Student("Nabil");
   s3.setScore(30,10,15);
   s3.computeCourseGrade();

   Student s4 =new Student("Sami");
   s4.setScore(32,14,44);
   s4.computeCourseGrade();

   ob.insertAtFront(s1);
   ob.insertAtFront(s2);
   ob.insertAtFront(s3);
   ob.insertAtFront(s4);
   ob.print();
```

```
System.out.println("number of passed Students is :
  "+ob.numberOfPassedOrNotPassedStundent("Pass"));
System.out.println("number of  not passed Students
is:"+ob.numberOfPassedOrNotPassedStundent("NoPass"));
System.out.println("The max is:"+ ob.maximumMarks());
System.out.println("The avrg : "+ ob.averageMarks());
ob.removeFromFront();
System.out.println("After remov- the first node :");
ob.print();
System.out.println("number of passed Students is :
     "+ob.numberOfPassedOrNotPassedStundent("Pass"));
System.out.println("number of  not passed Students is
  :"+ob.numberOfPassedOrNotPassedStundent("NoPass"));
System.out.println("The max is:"+ ob.maximumMarks());
System.out.println("The avrg:"+ ob.averageMarks());
 }
}
```

Testing the List ADT

```
/*   output
   The list : csc contains :
    The student Sami  has 90 marks  and  Course grade = Pass
    The student Nabil  has 55 marks  and  Course grade = Pass
    The student Ali  has 100 marks  and  Course grade = Pass
    The student Saad  has 45 marks  and  Course grade = NoPass
    ====number of passed Students is : 3
    ====number of  not passed Students is : 1
    The maximum is : 100
    The average : 72.5
    After removing the first node :


    The list : csc contains :
    The student Nabil  has 55 marks  and  Course grade = Pass
    The student Ali  has 100 marks  and  Course grade = Pass
    The student Saad  has 45 marks  and  Course grade = NoPass
    ====number of passed Students is : 2
    ====number of  not passed Students is : 1
    The maximum is : 100
    The average : 66.66666666666667
  */
```

The Generic List Class: Implementation with the element type that the Node will manipulate

The Node Class: Implementation

```
public class Node<T>
{
   T data;
   Node nextNode;

   public Node( T object )
   {
      this( object, null );
   }

   public Node(T object, Node node)
   {
      data = object;
      nextNode = node;
   }
```

```
    public T getData()
   {
      return data;
   }

   public  Node getNext()
   {
      return nextNode;
   }
 } // end class Node
```

The Generic List Class: Implementation with the element type that the Node will manipulate

```
public class List<V>
{ private Node<V> firstNode;
  private Node<V> lastNode;
  private String name;
public List()
   {  this( "list" );
   }
public List(String listName )
   { name = listName;
      firstNode = lastNode = null;
   }
public void insertAtFront(V insertItem )
{ if ( isEmpty() )
     firstNode = lastNode = new Node<V>( insertItem );
  else
     firstNode = new Node<V>( insertItem, firstNode );
   }
```

```
public void insertAtBack( V insertItem )
{  if ( isEmpty() )
     firstNode = lastNode = new Node<V>(
insertItem );
   else
    lastNode = lastNode.nextNode = new Node<V>(
insertItem );
   }
public V removeFromFront()
   { V removedItem = firstNode.data;
     if ( firstNode == lastNode )
        firstNode = lastNode = null;
     else
        firstNode = firstNode.nextNode;
    return removedItem;
   }
  public V getFromFront()
   {  return firstNode.data; }
```

The Generic List Class: Implementation with the element type that the Node will manipulate

```
public V removeFromBack()
 { V removedItem = lastNode.data;
    if ( firstNode == lastNode )
        firstNode = lastNode = null;
    else {
        Node<V> current = firstNode;
       while ( current.nextNode != lastNode )
          current = current.nextNode;
        lastNode = current;
       current.nextNode = null;
      }
     return removedItem;
   }
  public boolean isEmpty()
   {   return firstNode == null;    }
```

```
  public void print()
   {
     if ( isEmpty() )
     {
       System.out.printf( "Empty %s\n", name );
       return;
     }

     System.out.printf( "The %s is: ", name );
     Node current = firstNode;

     while ( current != null )
     {
       System.out.printf( "%s ", current.data );
       current = current.nextNode;
     }

     System.out.println( "\n" );
   }

} // end class List
```