Chapter 1: Overview

Dr. Safwan Qasem

King Saud University

College of Computer and Information Sciences

Computer Science Department

Chapter 1 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 and object-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

Chapter 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

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 a simple type value (int, float), it is duplicated and a copy is provided to the method.

When we pass an object, we are actually passing the **reference** (name) of an object, not a copy.

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

Passing Objects to a Method

Student Class

```
/*     File: Student.java */
class Student {
   private String name; // Data Member
   private String email; // Data Member

   //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;  }
}
```

Library Class

```
/*      File: LibraryCard.java   */
class LibraryCard {
   private Student owner; //student owner of this card
   private int borrowCnt; //number of books borrowed

   //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 Objects to a Method

Suppose a single student owns two library cards.

Then we can make the data member *owner* of the two LibraryCard Objects to refer to the same Student object. Here's one such program

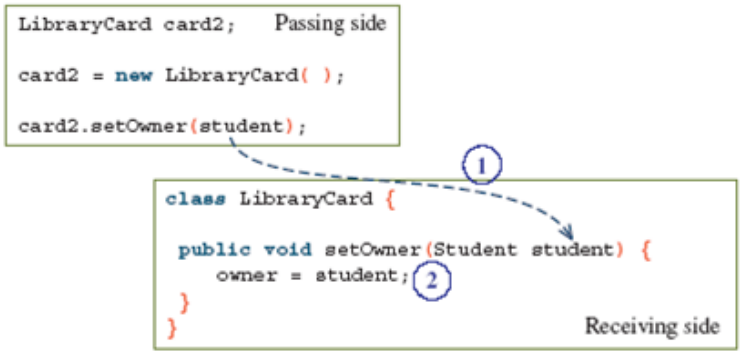
```
/*      File: Librarian.java   */
class Librarian {

   public static void main( String[] args ) {\

      Student    student;                    // Declare 1 object of type Student
      LibraryCard card1, card2;              // Declare 2 objects of type LibraryCard

     student = new Student( );               // Create an Object of type Student
      student.setName("Ali");                // Set the Student 's name
      student.setEmail("ali@ksu.edu.sa");    // Set the student's email

      card1 = new LibraryCard( );            // Create the 1st Library card
      card1.setOwner(student);               // Set the card's owner
      card1.checkOut(3);

      card2 = new LibraryCard( );            // Create the 2nd Library card
      card2.setOwner(student);               //the same student is the owner of the 2nd card, too

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

      System.out.println ("Card2 Info: ");
      System.out.println (card2.display() );
   }
}
```

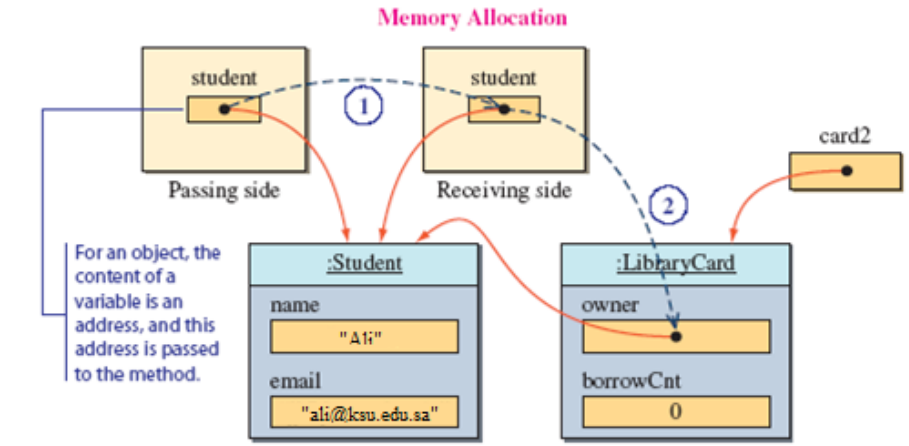
Passing Objects to a Method

When we pass an object to a method, **we are not sending a copy of an object,** but rather a reference to the object.

The memory address of the object is passed to the method.

This diagram illustrates how an objects is passed as an arguments to a method

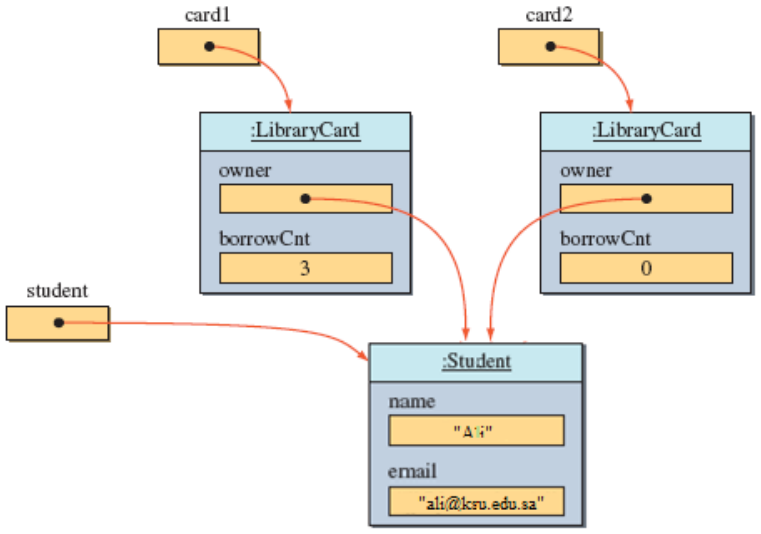




Sharing an object

We create one Student object, then two LibraryCard objects. For each LibraryCard object, we pass the same student when calling their setOwner methods.

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



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

```
Student    student;              // Declare 1 object of type Student
LibraryCard card1, card2;        // Declare 2 objects of type LibraryCard

student = new Student( );        // Create an Object of type Student
student.setName("Ali");          // Set the Student 's name
student.setEmail("ali@ksu.edu.sa");// Set the student's email

card1 = new LibraryCard( );      // Create the 1st Library card
card1.setOwner(student);         // Set the card's owner
card1.checkOut(3);

card2 = new LibraryCard( );      // Create the 2nd Library card
card2.setOwner(student);
```

Returning 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

Returning object from a 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 an instance of the Fraction class

Return an instance of the Fraction class

Returning object from a method

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

The returned object can be declared and created inside the method or received as an argument.

//----- FractionTest.java----------main program

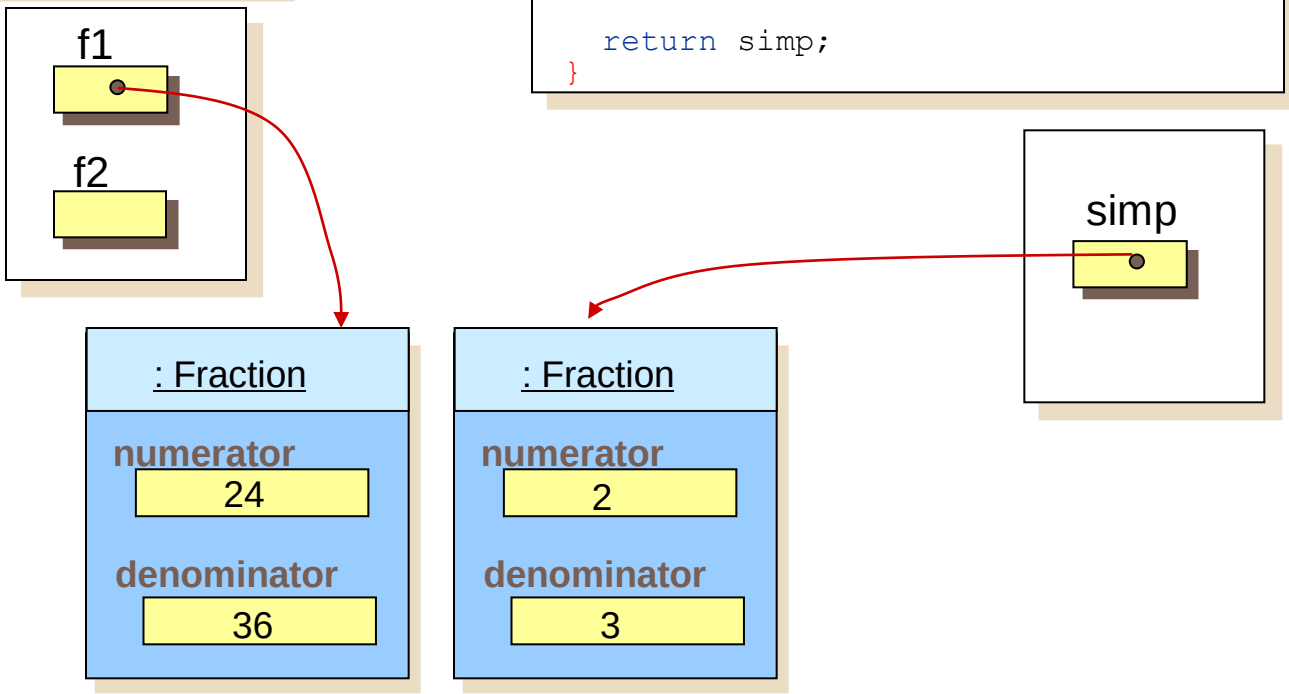
```
public class FractionTest
{
      public static void main(String[] args)
      {
            Fraction f1 = new Fraction(24,36);

            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;
}
```

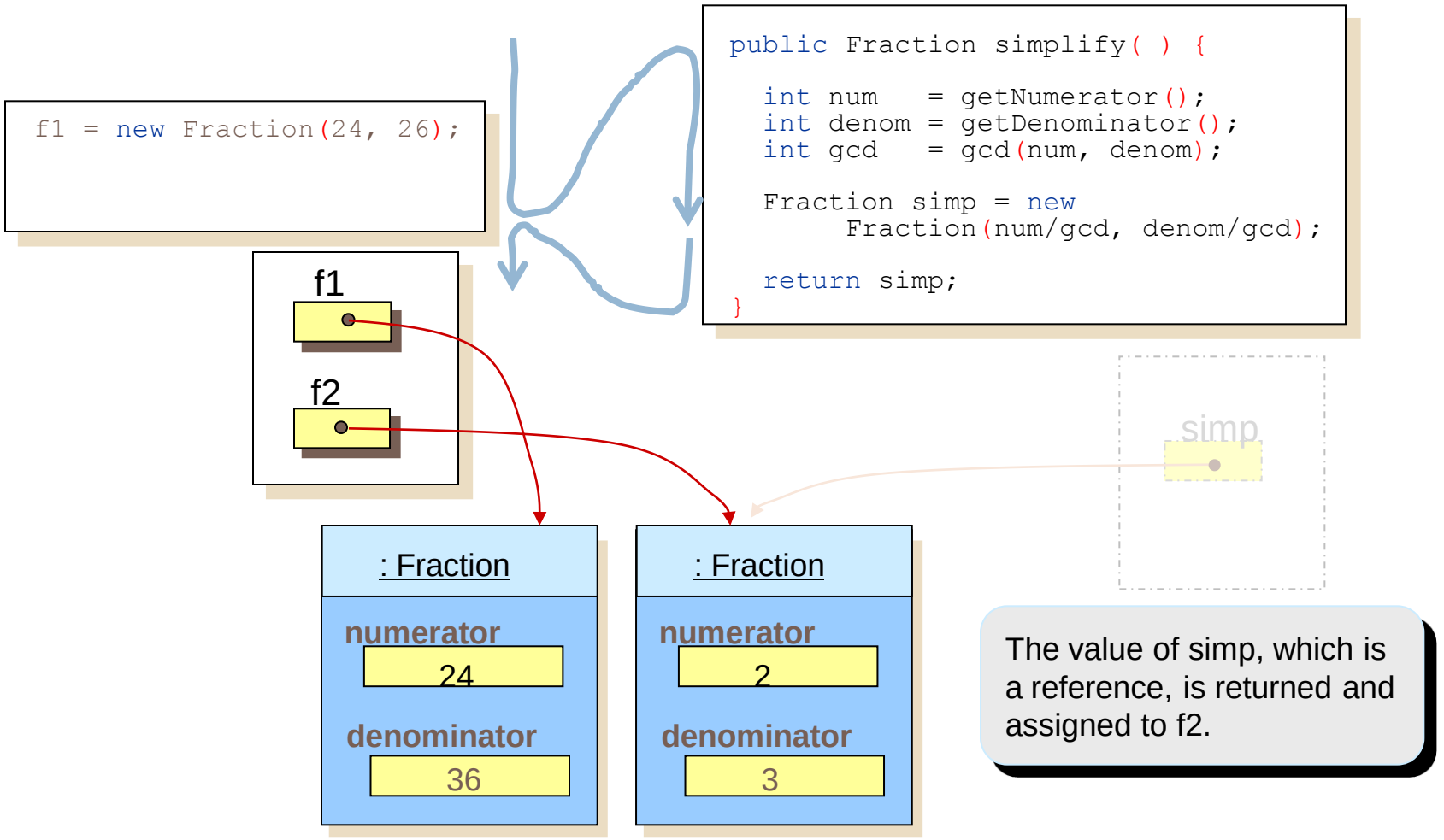
Returning object from a method


```
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;
}
```

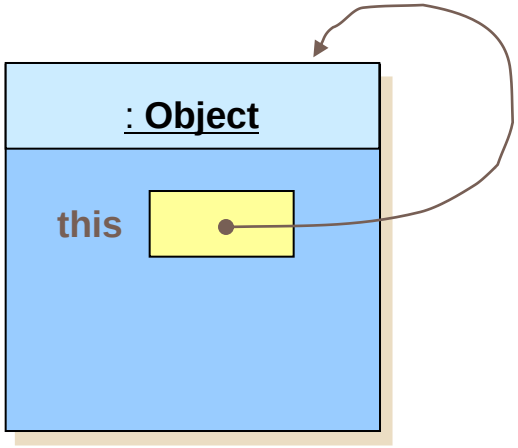


Returning object from a method



Reserved Word this

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

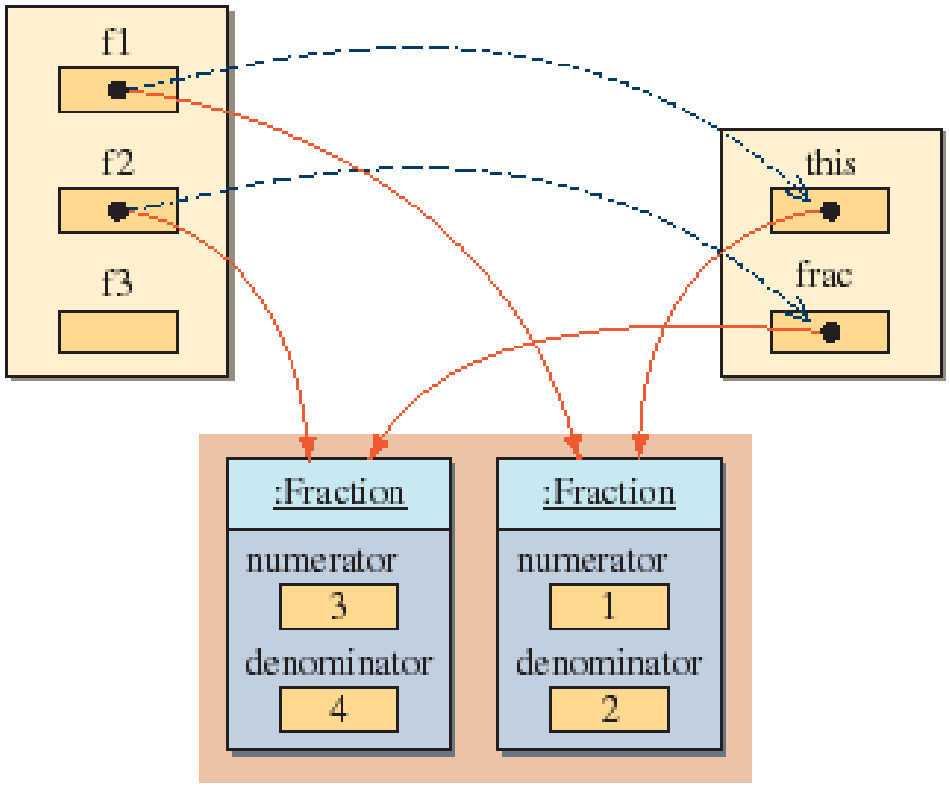


The reserved word this can be used in different ways. We will see all uses in this chapter.

Reserved Word this

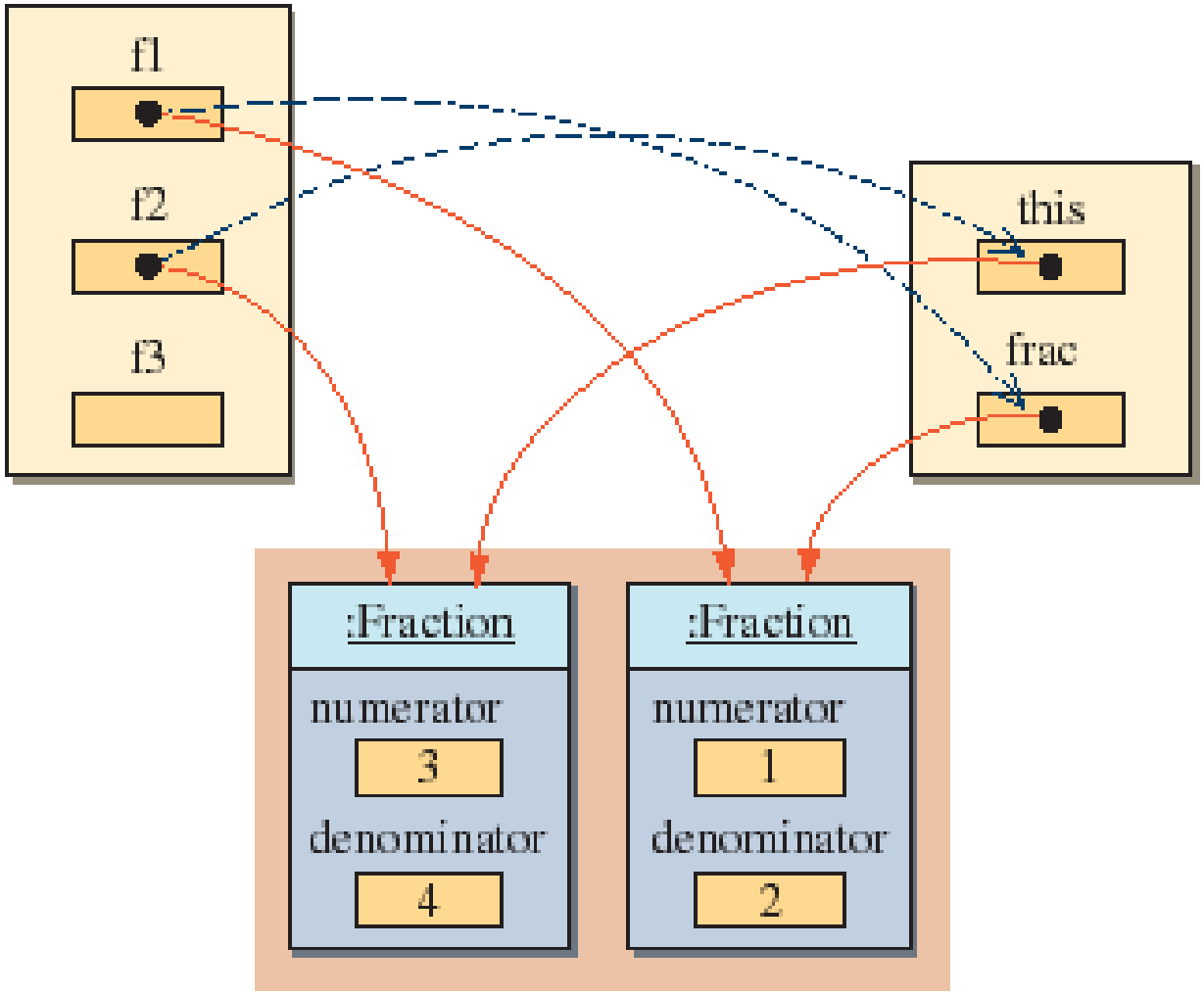

```
public Fraction add(Fraction frac) {

   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;
}
```

Reserved Word this



Because f1 is the receiving object (we're calling f1's method), so the reserved word this is referring to f1.
f3 = f1.add(f2)

Reserved Word this



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

f3 = f2.add(f1)

Reserved Word this

The reserved word this can be used to call a method of a receiving object. It can be used to refer to a data member as well.

```
class Student {

   int  age;

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

Overloaded methods

Two or more methods of the same class can share the same name as long as:

1. they have a different number of parameters

   OR

2. When the number of parameters is the same, they are of different data types.

This method is said to be overloaded, because the same method name has two or more different meanings.

Overloaded methods


```
//== 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 ============
public Fraction simplify()
{
      int num = this.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);
}
```

Overloaded methods

2 different versions of method add

```
//== 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);
   }
```

2 different methods of method multiply

```
//== Class Fraction continue and end ==========
//---- 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));
   }
}
```

Overloaded methods


```
//----- FractionTest.java----------main 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));
}
```

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.

Arrays of objects

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

```
public class Student
{               private String name;
                private int age;
                private char gender;
                public Student ()    {   age=0; name=" ";    gender=' '; }
                public Student (String na, int ag, char gen)  {
                                setAge(ag);   setName(na);   setGender(gen);
                }
                public Student (Student st)      { setstudent (st);}
                public void setStudent ( Student s)      {
                                age=s.age; gender =s.gender;
                                name=s.name. substring(0, s.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;  }
}
```

Arrays of objects: Creation

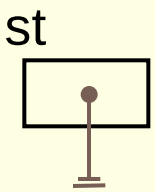
Code

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

Arrays of objects: Creation

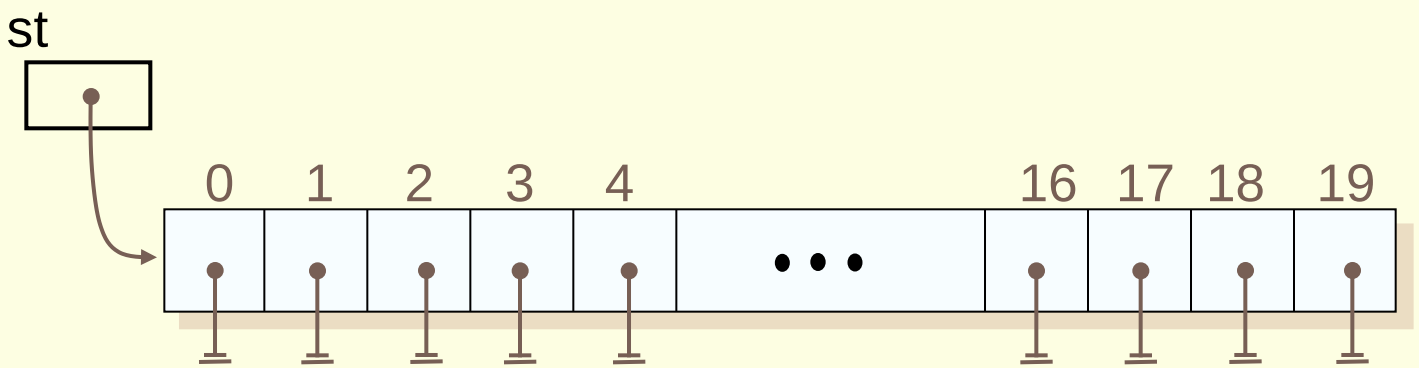
Code

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

Arrays of objects: Creation

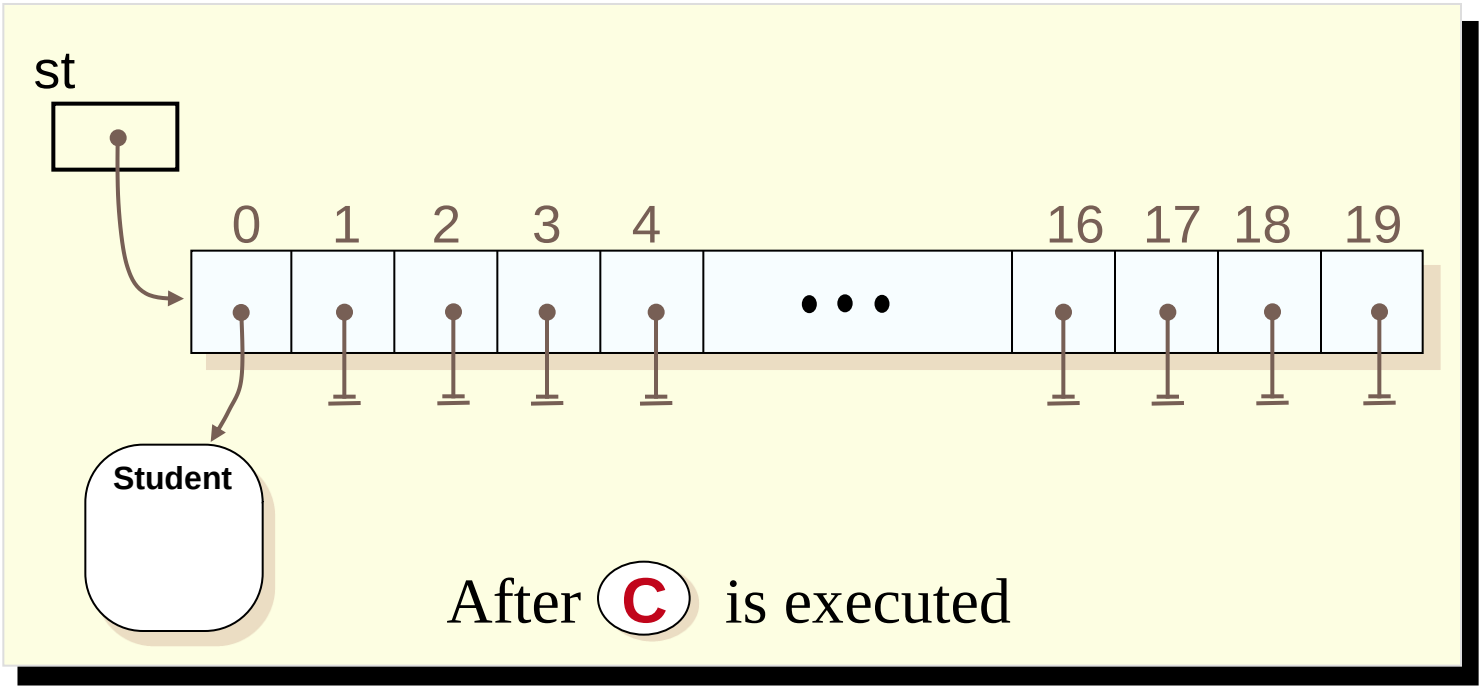
Code

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

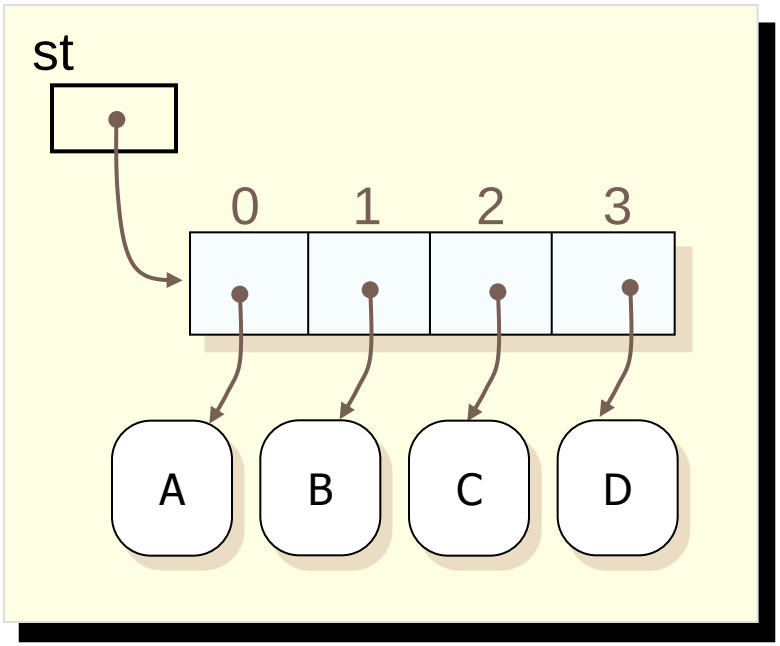


Arrays of objects: Deletion (method 1)

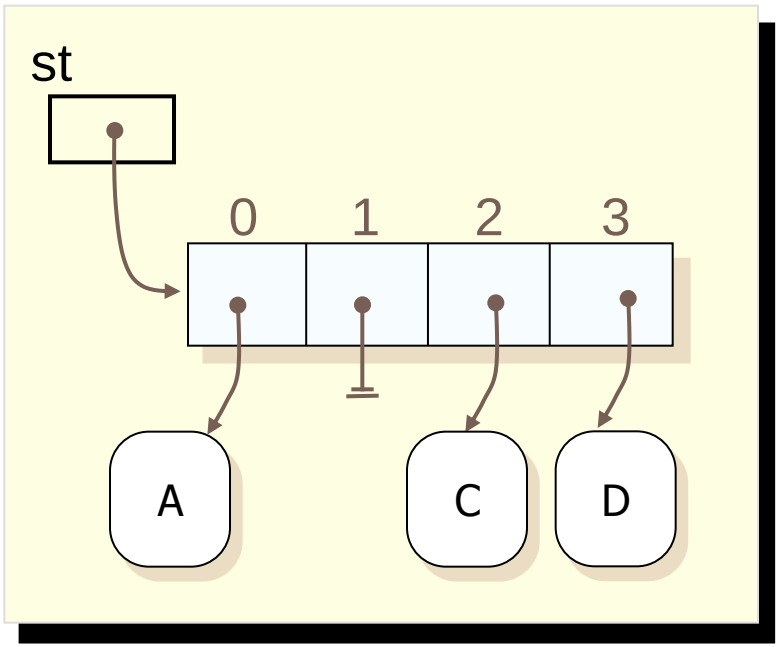
A

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

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



Before A is executed



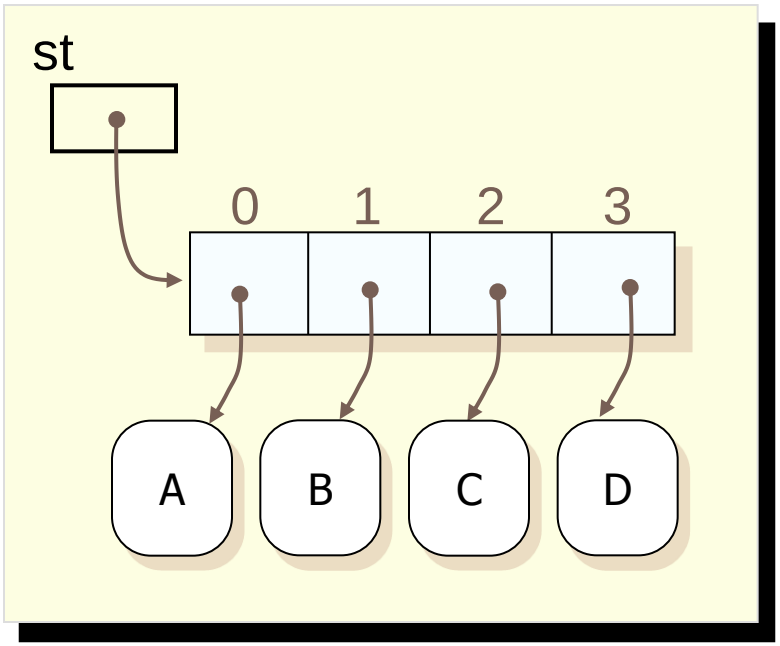
After A is executed

Arrays of objects: Deletion (method 2)

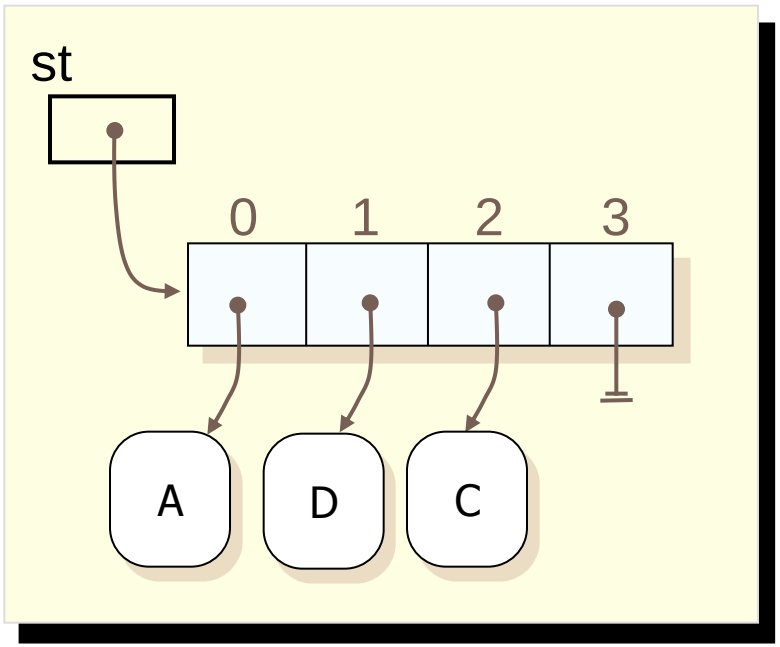
A

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

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



Before A is executed



After A is executed

Student Array Processing – Sample 1

Create Student objects and set up the p array

Constructors

```
import java.util.Scanner;

public class Section        {
      private Student p[];
      private int nbs;
      Scanner input = new Scanner(System.in);

      public Section( int size )   {
            p = new Student[size];
            nbs = 0;
      }

      public Section( Student pr[ ] ) {
          p = new Student[pr.length];
          for (int i =0; I < p.length;  i++)
              p[i]= new Student( pr[i] ); //---p[i]=pr[i] ---
          nbs = p.length;
      }
```

Setters

```
public void setSectionStudents ( Student [ ] pr ) {
      for (int i =0; (i< pr.length) && (i <p.length); i++)
      {
          p[i].setStudent( pr[i] );
          nbs++;
      }
}

public void setSectionStudents ( )        {
      String s = "";
      for (int i =0; i< p.length; i++) {
          p[i] = new Student();
          p[i].setName( input.next() + input.nextLine() );
          p[i].setAge( input.nextInt() );
          s = input.next( );
          p[i].setGender( s.charAt(0) );
      }
      nbs = p.length;
}
```

Student Array Processing – Sample 1

Add an object, parse all the array cells

Adding an object to the array

```
public boolean addStudent( Student p1 )
{
      if (nbs = = p.length)
            return false;
      p[nbs++] = p1;  // -- p[nbs] = p1;  nbs++ --
      return true;
}
```

Going through all the array cells

```
 //--- Average of all ages -----
public double averageOfAge( )
{
      double s=0.0;
      for(int i =0; i<=nbs-1; i++)
            s+=p[i].getAge( );
      return ( s/nbs );
}
```

Student Array Processing – Sample 1

Finding an object in an array

Find a student by 1 parameter

```
//---- Find the oldest  Students
public Student OldestStudent()
{
      Student old = p[0];
      for (int i =1; i<=nbs-1; i++)
            if (old.getAge() < p[i].getAge())
                              old =p[i];
      return (old);
}
//--- search  for a particular Student ----
public boolean findStudentByName(String na)
{
      for (int i=0; i<nbs; i++)  {
            if (p[i].getName().equals(na)== true)
                  return (true);
      }
      return (false);
}
```

Find a student by all its parameters

```
//---- return index of a Student if exist and -1 if
      not
  public int findStudent(Student pr)
  {
        for (int i=0; i<nbs; 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);
  }
```

Student Array Processing – Sample 1

Deleting an object from the array

Simple object deletion

```
public  boolean delete1Student(Student pr)
{
      int x = findStudent(pr)
      if (x != -1)
      {
            p[x] = p[nbs – 1];
            p[--nbs] = null;
            return true;
      }
      return false;
}
```

Object deletion with replacement

```
      public  boolean delete2Student(Student pr)
      {
            int x = findStudent(pr)
            if (x != -1)        {
                  for (int i = x; i<nbs - 1; i++)
                  p[i] = p[i + 1];
                  p[--nbs] = null;
                  return true;
            }
            return false;
      }
}
```