COP-3530
Data Structures

Instructor: Dr. Antonio Hernandez

Text: Data Structures and Algorithm Analysis in Java, 3rd Edition

Java Review

1. What do I remember?

PRACTICE

Program 01_01:

Write a Java program that asks the user to enter the x- and y-coordinates of the vertices of a 2D triangle and calculates the area. Use the Heron's formula to calculate the area. Use a loop to repeat the calculation as long as the user wants to do it.

(Note: for the area formula, visit Wikipedia)

Program 01_01

```
import java.util.Scanner;

public class Prog01_01
{
    public static void main(String[] args)
    {
        new Prog01_01();
    }

    double distance(double px, double py,
                    double qx, double qy)
    {
        return Math.sqrt(Math.pow(px-qx, 2) +
                         Math.pow(py-qy, 2));
    }
```

Program 01_01

```
public Prog01_01()
{
    Scanner in = new Scanner(System.in);
    char answer;

    do
    {
        System.out.println("Enter the three vertices:");
        double x1 = in.nextDouble();
        double y1 = in.nextDouble();
        double x2 = in.nextDouble();
        double y2 = in.nextDouble();
        double x3 = in.nextDouble();
        double y3 = in.nextDouble();

        in.nextLine();
```

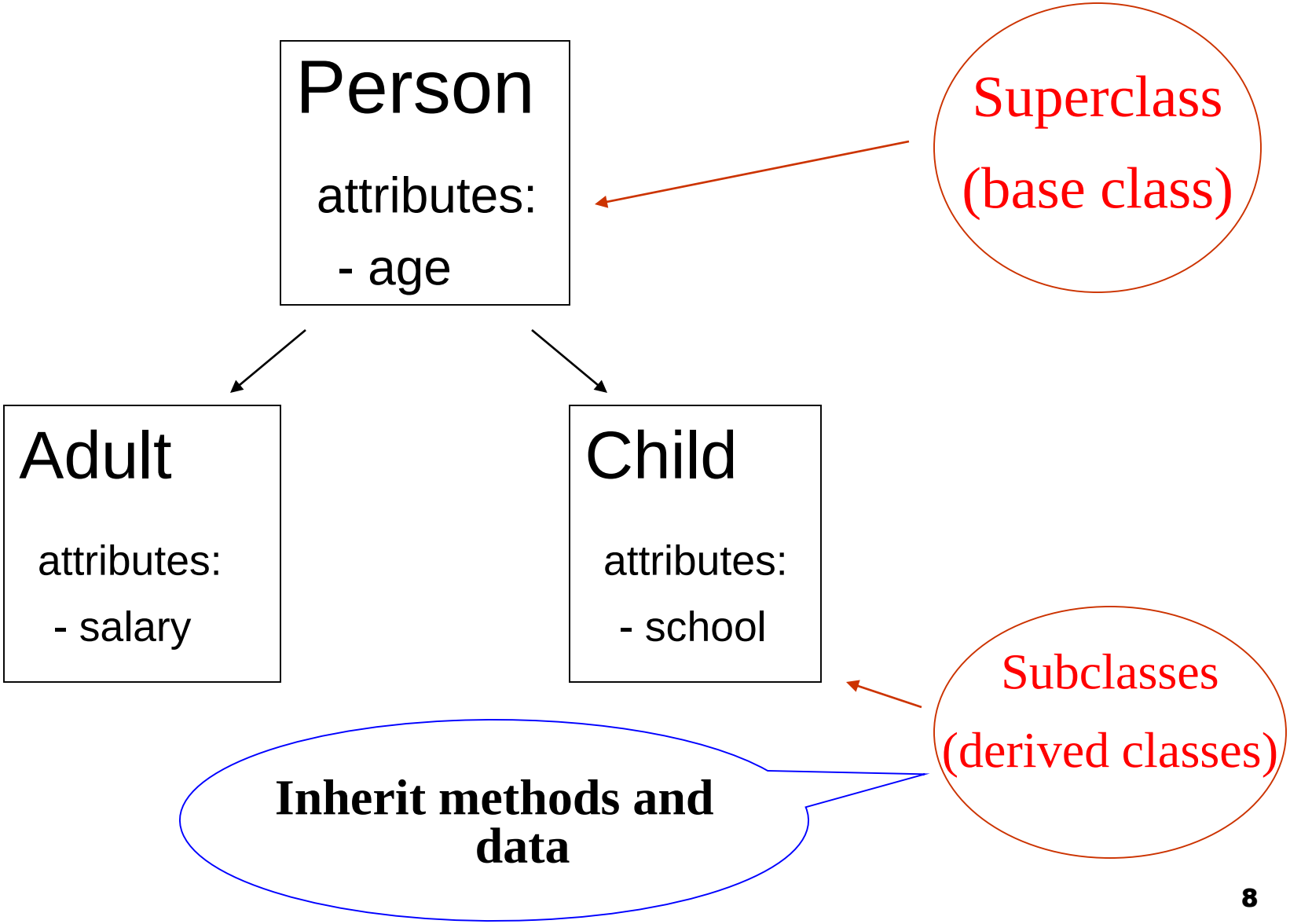
Program 01_01

```
            double a = distance(x1, y1, x2, y2);
            double b = distance(x1, y1, x3, y3);
            double c = distance(x2, y2, x3, y3);

            double s = (a+b+c)/2;
            double area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
            System.out.println("Area = " + area);

            System.out.println("Try another example? <Y/N>");
            answer = in.nextLine().toUpperCase().charAt(0);
        } while (answer == 'Y');
    }
}
```

Java Review

2. Inheritance

Subclasses and Superclasses



Inheritance in Java

Basics:

1) Inheritance in Java plays the same role as in any OOP language: let us create new classes that add functionality to existing classes
2) Remember: inheritance is used to model **`is-a`** relationship between objects
3) In Java you use the keyword **`extends`** to denote inheritance

```
class AirlineEmployee extends Employee
  {
        //added methods and fields
  }
```

Java Review

3. Abstract Classes

Abstract Classes

An ***abstract class****:* class that contains *abstract methods*.

An ***abstract method***: only its declaration is provided.

Both abstract classes and methods are specified with the keyword “abstract”.

An abstract class cannot be instantiated and its abstract methods should be implemented in the subclasses.

Abstract classes: *Example*

```
//GeometricObject class

import java.awt.*;

abstract public class GeometricObject
{
   private Color interiorColor;    //Interior color
   private Color boundaryColor;    //Boundary color

   public GeometricObject()
   {
        interiorColor = Color.white;
        boundaryColor = Color.black;
   }

   abstract public double area();
```

Abstract classes: *Example (cont.)*

```
    public Color getInteriorColor()
    {
        return interiorColor;
    }

    public void setInteriorColor(Color C)
    {
        interiorColor = C;
    }

    public Color getBoundaryColor()
    {
        return boundaryColor;
    }

    public void setBoundaryColor(Color C)
    {
        boundaryColor = C;
    }
}
```

Abstract classes: *Example (cont.)*

```
//Point class

public class Point extends GeometricObject
{
   private double x;
   private double y;

   public Point()
   {
        x = y = 0;
   }

   public Point(double x1,  double y1)
   {
        x = x1;
        y = y1;
   }
```

Abstract classes: *Example (cont.)*

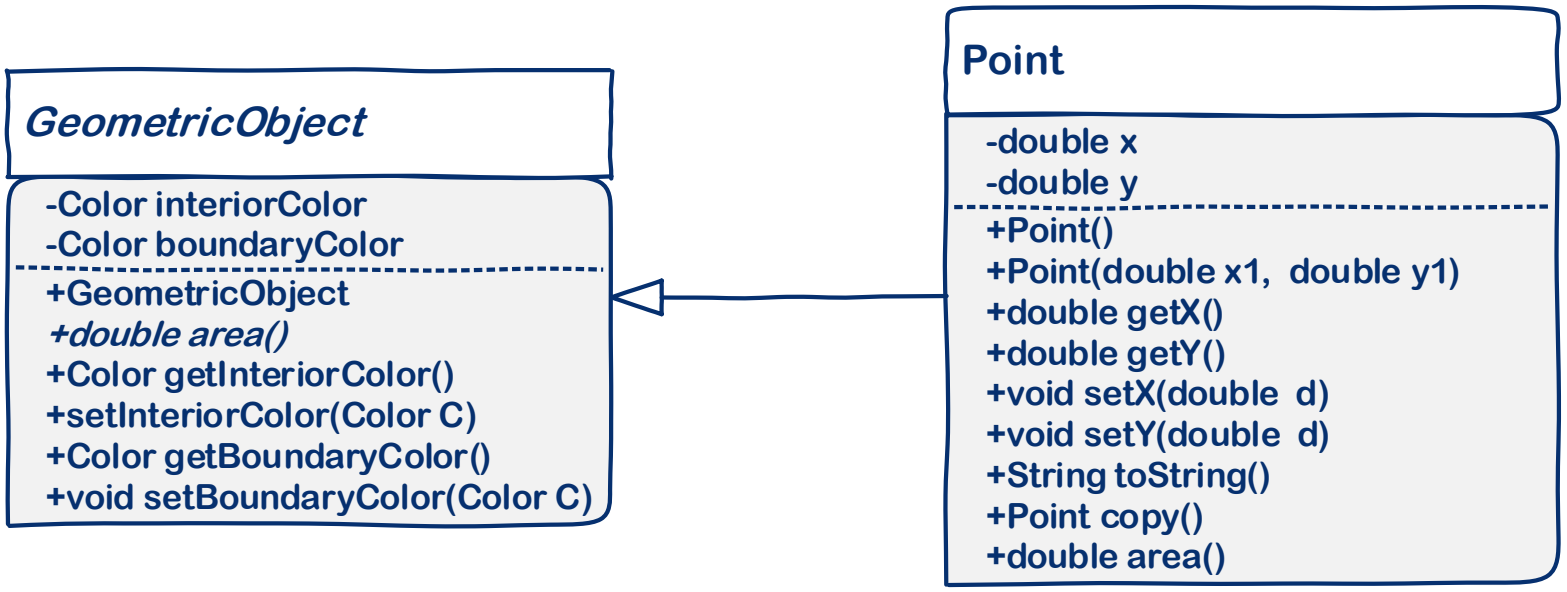
```
public double getX()
{
    return x;
}
public double getY()
{
    return y;
}
public void setX(double  d)
{
    x = d;
}
public void setY(double  d)
{
    y = d;
}
```

Abstract classes: *Example (cont.)*

```
    public String toString()
    {
        return " (" + x  + ", " + y + ") " ;
    }

    public Point copy()
    {
        return new Point(x, y);
    }

    public double area()
    {
        return 0;
    }
}
```

Abstract area: implemented in the subclass

UML Class Diagram



Method *toString*

toString(): This is a function inherited by every Java class from the *Object* class and this method is used to obtain a string representation of an object.

Whenever we call *System.out.println()* with an object name, toString is called.

toString(): It is a convenient method that it is recommended to have implemented in our classes.

Keyword: *super*

The keyword ***super*** can be used to invoke the constructors and other methods in the super class.

To invoke the constructor of the superclass, call

super(<arguments>);

(***super()*** is the default constructor)

When a method in the class overrides one method in the super class, ***super.<method>*** can be used.

The *super* keyword: *Example*

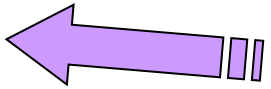
```
//Circle class

public class Circle extends Point
{
   private double radius;

   public Circle()
   {
        super();
        radius = 1;
   }

   public Circle(double r)
   {
        super();
        radius = r;
   }
```

The *super* keyword: *Example*

```
public Circle(double x,  double y, double r)
{
     super(x, y);
     radius = r;
}

public double getRadius()
{
     return radius;
}

public void setRadius (double r)
{
     radius = r;
}
```

The *super* keyword: *Example*

```
    public String toString()
    {
        return super.toString() + "[" + radius + "]" ;
    }

    public double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }
}
```



Java Review

4. References to objects

Basics of objects

1) Java distinguishes between objects and objects variables.

```
Date deadline;                ← object variable
deadline = new Date();        ← object
```

2) An object variable does not contain an object, it only refers to it.
3) Use null to indicate that an object variable refers to no object.
4) We can think of Java object variables as analogous to object pointers in C++.
5) All Java objects live in the heap.
6) Temporary objects are used:

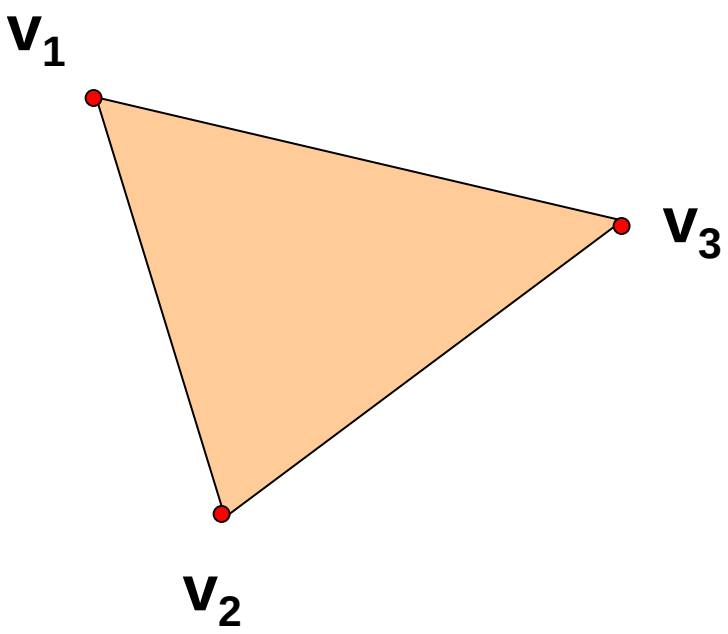
```
System.out.println ( new Date() );
new Date( ) . toString( );
```

Objects as function parameters

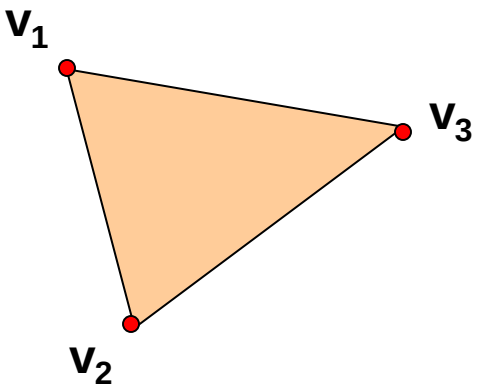
1) While the method parameter passing mechanism in C++ is by-value and by-reference, Java only possesses the passing by-value.
2) For primitive types like *int* or *double*, this means that changes done to a formal parameter in the body of a method do not affect the actual parameter.
3) When passing an object variable to a function, it is a reference to the actual object that is being passed
4) Therefore, if a mutator of a formal object parameter is invoked, the actual object parameter will be changed.

Dealing with object parameters: *Example*

A **triangle** can be represented through its vertices



Dealing with object parameters: *Example*




```
//PROGRAM: A class that models a triangle

public class Triangle
{
   private Point v1;
   private Point v2;
   private Point v3;
```

Dealing with object parameters: *Example*

```
//Constructors
public Triangle()
{
    v1 = new Point(0, 0);
    v2 = new Point(0, 1);
    v3 = new Point(1, 0);
}

public Triangle(Point p1, Point p2, Point p3)
{
    v1 = p1;
    v2 = p2;
    v3 = p3;
}
```

Dealing with object parameters: *Example*

```
public void getVertices(Point p1, Point p2, Point p3)
{
    p1.setX(v1.getX());
    p1.setY(v1.getY());

    p2.setX(v2.getX());
    p2.setY(v2.getY());

    p3.setX(v3.getX());
    p3.setY(v3.getY());
}
```

Dealing with object parameters: *Example*

```
   public void setVertices(Point p1, Point p2 , Point p3)
   {
        v1 = p1;
        v2 = p2;
        v3 = p3;
   }

   // Other class methods suchs as toString and area

}
```

Java Review

5. Shallow and deep copy

Shallow and Deep Copy

When copying objects, we have two possibilities from which to choose:

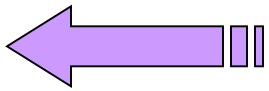
shallow or ***deep*** copy.

Shallow copy: creates a new reference to the same object

Deep copy: creates a new reference to a new object.

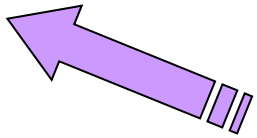
Shallow and Deep Copy: *Example*

```
Point p =  new Point(1, 2);

Point q = p;
```



Shallow copy

```
Point q = new Point(p.getX(), p.getY());
```



Deep copy

Java Review

6. Polymorphism

Polymorphism

Polymorphism: a characteristic of object programming languages in which a given entity can be assigned a different meaning or usage in different contexts.

Example: using a super class variable to refer to a subclass object.

Polymorphism: *Example*

```
Point p =  new Point(1, 2);

Circle c = new Circle(3);

//Testing polymorphism
GeometricObject g;

g = p;
System.out.println(g.area());

g = c;
System.out.println(g.area());
```

Java Review

7. Interfaces

Interfaces

Interfaces: In Java, multiple inheritance is not allowed. Interfaces are used to mimic a multiple inheritance environment

Ex.: ActionListener

Declaration:

1. **The declaration looks like the class' but the keyword *interface* is used instead of *class***

   ```
   public interface Drawable
   {       . . .      }
   ```

2. **All methods of an interface must be abstract**
3. **An interface may have data fields, but must be *final* and *static***

Interfaces

Implementing interfaces:

1) **To inherit from an interface:**

```
public class C extends B implements D
{       . . .     }
```

2) **When a class is declared to implement an interface, it must provide implementations for all methods of the interface.**

3) **A class may implement as many interfaces as needed.**

```
public class C extends B implements D, E
{       . . .     }
```

Java Review

8. Arrays

Arrays: declaration

Example:

```
double[ ]  phoneBills;
phoneBills = new double [12]
```

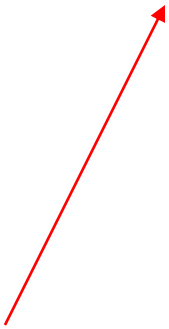
or

```
double[ ] phoneBills  = new double [12];
```

Arrays: initializing

Example:

```
String months[ ]  = { “Jan”, “Feb”, “Mar”,
                      “Apr”, “May”, “Jun”,
                      “Jul”,  “Aug”, “Sep”
                      “oct”,  “Nov”, “Dec”  };
```



No *new* or <*size*> is used

Arrays: *Example*

```
//Prog08_01 Arrays review

public class Prog08_01 {
   public static void main(String args[]) {
      double[] a = new double[5];

      // Fill the array with powers of two
      for (int i = 0; i < a.length; i++)
         a[i] = Math.pow(2, i);

      String s = "";
      for (int i = 0; i < a.length; i++)
         s = s + (int) a[i] + "\n";

      System.out.println(s);
   }
}
```

PRACTICE

Program 08_02:

Write a Java program that fills an array of 10 numbers with random integers in [0, 100) and displays it. The program will calculate the average of the numbers in the array, the minimum, and the maximum and will display them.

Note: You can use method `nextInt(n)` in java.util.Random to generate an integer in [0, n).

Program 08_02

```
//Prog08_02 Arrays review
import java.util.Random;

public class Prog08_02 {
    public static void main(String args[]) {
        int SIZE = 10;
        int[] a = new int[SIZE];

        // Fill the array with random numbers
        Random rand = new Random();
        for (int i = 0; i < a.length; i++)
            a[i] = rand.nextInt(100);

        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");

        System.out.println();
```

Program 08_02

```
        //Min. max. and average
        int sum=0, min=a[0], max=a[0];
        for (int i = 0; i < a.length; i++){
            sum = sum+a[i];

            if (min > a[i]) min = a[i];
            if (max < a[i]) max = a[i];
        }

        double ave = sum/SIZE;

        System.out.println("\nMin: " + min +
                           ", ave: " + ave +
                           ", max: " + max);

    }
}
```

Abstract Data Types

9. The List ADT

Abstract Data Types

Definition

Abstract Data Type (ADT): A mathematical model of a data structure. It specifies logical properties (*domain* and *operations*) without the implementation details

i.e. w*hat* and not *how*

Example: domain*:* *int*,
operations*:* +, -, *, /

ADT: List

- List: A collection of elements of the same type

Note: *length* of a list is the number of elements in the list

Operations performed on a list

- Create the list; initialized to an empty state
- Determine whether the list is empty
- Determine whether the list is full
- Find the size of the list
- Destroy, or clear, the list
- Determine whether an item is the same as a given list element

Operations performed on a list

- Insert an item in the list at the specified location
- Remove an item from the list at the specified location
- Replace an item at the specified location with another item
- Retrieve an item from the list at the specified location
- Search the list for a given item

Abstract Data Types

10. Array-based implementation of lists

List: Array-based implementation

```
//ArrayList class

public class ArrayList
{
        public ArrayList() {...}
        public boolean isEmpty() {...}
        public void print() {...}
        public void insert(int x) {...}
        public void removeItemAt(int pos) {...}
        private static final int SIZE = 10;

        //array to store the list items
        private int[] list = new int[SIZE];

        private int length;  //amount of items in the list
}
```

List: Array-based implementation

The constructor simply creates an empty list:

```
public ArrayList()
{
        length=0;
}
```

The function *print* traverses the array to display the elements in the list:

```
public void print()
{
        for (int i=0; i < length; i++)
                System.out.print(list[i] + " ");
        System.out.println();
}
```

List: Array-based implementation

The function ***insert*** adds an element ***x*** at the end of the list.

```
public void insert(int x)
{
        if (length == SIZE)
            System.out.println("Insertion Error: list is full");
        else
        {
                list[length] = x;
                length++;
        }
}
```

NOTE: another name for ***insert*** is ***add***

List: Array-based implementation

isEmpty returns *true* if there are no elements inserted :

```
public boolean isEmpty()
{
        return length == 0;
}
```

The function *removeItemAt (int pos)* will remove the element at location *pos* :

```
public void removeItemAt(int pos)
{
        for ( int i = pos; i < length - 1; i++ )
                list[i] = list[i+1];
        length--;
}
```

List: Array-based implementation

The main function fills 10 elements of the list *list* with random numbers:

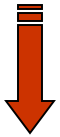
```
//PROGRAM 10_01: Testing the class ArrayList
import java.util.Random;
public class Main
{
        public static void main(String[] args)
        {
                Random rand = new Random();
                ArrayList list = new ArrayList();
                for (int i=0; i < 10; i++)
                        list.insert(rand.nextInt(100));
                list.print();
        }
}
```

Abstract Data Types

11. Array-based lists and expandable arrays

Expandable Arrays

When the internal array in the implementation of a list is completely filled with elements,



no further insertion in the list is possible.

To avoid this situation, the elements can be moved to a larger array each time the original array is full. This is known as *array expansion* or *expandable arrays*.

Expandable Arrays

Several ways to expand an array in Java:

- variant 1: instantiate a larger array, copy over the elements in a loop, and shallow-copy the new array to the old one
- variant 2: instantiate a larger array, copy over the elements using System.arraycopy, and shallow-copy the new array to the old one
- variant 3: using Arrays.copyOf

PRACTICE

Program 11_01:

Expand the array in Program08_02 to twice its size, after it has been filled with random numbers.

Program 11_01

```
//Prog11_01 Expanding the array in Prog08_02
import java.util.Arrays;
import java.util.Random;


public class Prog08_02 {
    public static void main(String args[]) {
        int SIZE = 10;
        int[] a = new int[SIZE];

        // Fill the array with random numbers
        Random rand = new Random();
        for (int i = 0; i < a.length; i++)
            a[i] = rand.nextInt(100);

        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
```

Program 11_01

```
System.out.println();

//Min. max. and average
int sum=0, min=a[0], max=a[0];
for (int i = 0; i < a.length; i++){
    sum = sum+a[i];

    if (min > a[i]) min = a[i];
    if (max < a[i]) max = a[i];
}

double ave = sum/SIZE;

System.out.println("\nMin: " + min +
                   ", ave: " + ave +
                   ", max: " + max);
```

Program 11_01

```
        //Expanding the array - variant 3
        SIZE = 2*SIZE;
        a = Arrays.copyOf(a, SIZE);

        System.out.println("\nAdding a new element:");
        a[SIZE/2] = 100;

        for(int i=0; i<SIZE/2+1; i++)
        {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }
}
```

Abstract Data Types

12. Bags

Bag ADT

List ADT*:* The notion of location or position is essential to the concept of the list ADT.

Methods involving locations: OK (removeAt, addAt, retrieveAt, etc).

Bag ADT: A bag is an abstraction of a *multiset*. Locations of items are not relevant.

Methods involving locations: are not offered as part of the public interface (~~removeAt~~, ~~addAt~~, ~~retrieveAt~~, ~~etc~~).

Bag: Array-based implementation

```
//Bag class - implemented as an array

public class Bag
{
        public Bag() {...}
        public boolean isEmpty() {...}
        public void print() {...}
        public void add(int x) {...}
        public void remove(int x) {...}
        private static final int SIZE = 10;

        //array to store the bag items
        private int[] bag = new int[SIZE];

        private int length;  //amount of items in the bag
}
```