Arrays and ArrayList

Lecture 6 in 1DV506

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

November 26, 2019

Lecture 6

- Arrays
- The `ArrayList` class
- The help classes `Arrays` and `Collections`

Reading instructions
Sections 7.1-7.9, 7.12-7.13, Chapter 8, and sections 11.11-11.12

The sections about sorting and searching (7.10, 7.11) are presented in the next course.

Handling Data Sets

```
public static void main(String[] args) {
   Scanner scan = new Scanner(System.in);
   System.out.print("How many integers: ");
   int sz = scan.nextInt();

   int[] data = new int[sz];          // create integer array of size sz
   for (int i=0; i<sz; i++) {
      System.out.print("Integer "+(i+1)+": ");
      data[i] = scan.nextInt();          // store input at position i in array
   }

   System.out.print("All integers : ");
   for ( int i=0; i<sz; i++)
      System.out.print(data[i]+" ");    // read data from array
}
// Usage
How many integers: 3
Integer 1: 7
Integer 2: -22
Integer 3: 16
All integers : 7 -22 16
```

Arrays

- **Arrays** are used to store sequences of data.
- Example:

```
int [] a = new int[3];   // Create an empty array that can store 3 integers.
a[0] = 5;
a[1] = 10;               // Three values are assigned to the array.
a[2] = 15;
System.out.println("The second value is "+a[1]);
```

- **Note:**
  - Array variables are characterized by `[]` (for example `int[] a`)
  - Arrays are objects ⇒ created using `new`
  - Array objects have a fixed size, given at creation (for example `new int[3]`)
  - Note: array variables (`int[] a`) have no size
  - Arrays have an **index**, always starting at zero
    ⇒ `maxIndex = size - 1`
  - Assign position 2 the value 7: `a[2] = 7`
  - Read the value of position 5: `int n = a[5];`
  - Access outside `[0, maxIndex]` ⇒ crash at execution

Iterating Over Arrays Using `for` Statements

```
int[] a = new int[10];
System.out.println("Size: "+a.length);    // a.length => array size

for (int i=0; i<a.length; i++)    // fill array with [0,10,20,..,90]
    a[i] = i*10;

System.out.print("Content: ");
for (int n : a)                           // for-each statement
    System.out.print(n+" ");
```

- `a.length` gives the length (size) of the array
- Ordinary `for` iteration: `for (int i=0; i<a.length; i++)`
- Simplified `for-each` iteration: `for (int n : a)`
  ⇒ `n` will take on all values in the array `a`. Starts at index zero.

The ordinary `for` iteration is more flexible. Start where you want, different step sizes, different directions. `for-each` iteration is simpler but does not have any variants.

⇒ Always all elements, from the beginning to the end.

Other Types of Arrays

We can create arrays containing any type of elements

- So far only integer arrays: `int[] a = new int[5];`
- Arrays can have elements of any type, for example `double, boolean, String, Random, MyClassA`

```
 String [] names = new String[3];
names[0] = "Olga";
names[1] = "Claes";
names[2] = "Werner";

for (String str : names)
   System.out.println("Name: "+str);

// In reverse order
for (int i=names.length-1; i >= 0; i--) {
     String name = names[i];
     System.out.println(name);
}
```

Initialisation of Arrays

```
double[] data = { 1.22, 2.34, 4.45, 5.63, 6.73};
double sum = 0.0;
for (int i=0; i<data.length; i++)
  sum = sum + data[i];
System.out.println("Mean value: "+ sum/data.length);

char[] name = { 'J', 'o', 'n', 'a', 's'};
for (char c : name)
    System.out.print(c);
```

Note

- You can create and initialize an array using `int[] a = {1,2,3,4,5,6}`
- The length of the array (`a.length`) will be the same as the number of elements (6).
- **Important:** `.length` ⇒ the size of the array is not necessarily the same as the number of inserted elements.

Array Parameters to Methods

- A method can return an array or take arrays as parameters.
- Example: Returns an array with the elements in reversed order.

```
public String[] reverseOrder(String[] in) {
    String[] out = new String[in.length];
    int count = 0;
    for (int i = in.length-1; i>=0; i--) {
        out[count++] = in[i];
    }
    return out;
}
```

- Example: Swap the strings in position p and position q

```
public void swapPosition(String[] str, int p, int q) {
    String tmp = str[p];
    str[p] = str[q];
    str[q] = tmp;
}
```

Notice that `reverseOrder` creates and returns a new array whereas `swapPosition` modifies the input array. See next slide for how they are used.

Many Methods in the main() Class

The program `MethodsInMain` uses several methods in the `main` class:

```
public class MethodsInMain {

   public static void main(String[] args) {
      String[] str = {"Do","Re","Mi","Fa","So","La"};
      str = reverseOrder(str);           // La,So,Fa,Mi,Re,Do
      swapPosition(str,1,3);             // La,Mi,Fa,So,Re,Do

      for (String s : str) {             // Print them all
         System.out.print(s+" ");
      }
   }

   private static  String[] reverseOrder(String[] in) { ... }
   private static void swapPosition(String[] str, int p, int q) { ...}
}
```

- The methods *must* be declared as `static`
- Methods are called without `variable.methodName(...)`
- `private` since only called from within the class `MethodsInMain`.

Arguments to `main(String[] args)`

```
public class ArgsMain {

    public static void main(String[] args) {
        System.out.println("Number of argument: "+args.length);  // Number of arguments
        for (int i=0;i<args.length; i++)
            System.out.println(i+".\t"+args[i]);                    // Print argument
    }
}
```

- `args` in the method `main` is the input to the program
- `String[] args` ⇒ the arguments are always strings.
  Can be transformed to integer or doubles using the wrapper classes.
- How to use:
  1. **Eclipse:** `Run --> Run Configurations --> arguments`
  2. **Command prompt:** `java lec6.ArgsMain My three arguments`

Try to run the program `ArgsMain.java` when you get home.

The Utility Class Arrays

The utility class java.util.Arrays has methods (e.g. sort, toString, fill) that can be used for sorting, printing and filling an array with values. Example:

```
// Sort and Print an Array
int[] data = {3,45,-8,9,12,6,-9,6};
Arrays.sort(data);                          // Sort array
String output = Arrays.toString(data);  // Get content string
System.out.println("Sorted Content: "+output);

// Fill and Print an Array
String[] manyJonas = new String[5];
Arrays.fill(manyJonas,"Jonas");
output = Arrays.toString(manyJonas);  // Get content string
System.out.println("String Array Content: "+output);

// Print-out
// Sorted Content: [-9, -8, 3, 6, 6, 9, 12, 45]
// String Array Content: [Jonas, Jonas, Jonas, Jonas, Jonas]
```

Old Exam Exercise

Considering the following program:

```
public class AddX {
    public static void addTwo(int n) {n = n+2;}

    public static void addOne(int[] arr) {
        for (int i=0;i<arr.length; i++)
            arr[i] = arr[i]+1;
    }

    public static void main(String[] args) {
        int a = 5;
        addTwo(a);
        System.out.println(a);        // Print-out 1

        int[] arr = {5,10,15};
        addOne(arr);
        System.out.println(arr[1]);   // Print-out 2
    }
}
```

What is printed in the two cases? Motivate your answer.

Two Types of Parameters

- Java separates: 1) Primitive types, and 2) Reference types
- They are treated differently in calls
- Method calls using primitive types

```
int a = 5;          public void addTwo(int n) { // n is assigned a copy
addTwo(a);             n = n + 2;               // of the argument value a
                    }                           // Changes in n do NOT
System.out.println(a);                          // affect the call
// Print-out 5                                  // argument a
```

- A *copy* of the value is given to method `addTwo`
- ⇒ The parameter `n` is independent of `a`
- ⇒ Changes inside the method do not affect the calling method.
- It is called **call-by-value**.

Call-by-Reference and Aliasing

- Calls using reference types

```
int[] arr =  {5,10,15};
addOne(arr);

System.out.println(arr[1]);
// Print-out 11
```

```
public void addOne(int[] n) {
   for (int i=0;i<n.length;i++)
      n[i] = n[i]+1;
}
// n is assigned a reference to the same
// object referenced by the argument arr.
// Both are referencing the same object
// ==> changes do affect the argument
// used in the code.
```

- A reference (an object address) is given to the method `addOne`
- ⇒ The parameter `n` references the same object as argument `arr`
- ⇒ Changes inside the method do affect the calling method.
- It is called **call-by-reference**.

To have many active references at a single object is called **aliasing**. It is a bit dangerous since changes using one of this references can give unexpected results at other parts of the program.

Array Variables Reference Array Objects

```
int [] a = {1,2,3,4};      // 'a' points to array object {1,2,3,4}

int [] b = a;              // 'b' points to the same object
b[1] = 7;                  // object updated ==> affects a and b

System.out.println( Arrays.toString(a));   // {1,7,3,4}
```

- Arrays are objects ⇒ variables hold a reference (their adress) to array objects
- Assignment `int[] b = a` ⇒ `a` and `b` reference the same object ⇒ changes of array using `b` affects `a`.
- **Assignments (and calls) do not make a new copy of the array!**

Variable Length Argument Lists

```
public static void main(String[] args) {
   printMax(34, 3.5, 3,2, 56.5, -99);

   double[] arr = {1, 2.5, 9.34, 7};
   printMax(arr);
}

private static void printMax(double... numbers ) {
   double max = Double.MIN_VALUE;  // Initialize max to smallest
                                   // possible double value
   for (double d : numbers) {
      if (d > max)
         max = d;
   }
   System.out.println("The max value is "+max);
}
// Print-out
The max value is 56.5
The max value is 9.34
```

Variable Length Argument Lists (cont.)

- Signature `void printMax(double... numbers )`
  ⇒ method can take a sequence **OR** an array as input argument
- The input is always treated as an array inside the method
- The variable length parameter declaration `double... numbers` must be the last parameter in the parameter list
- Examples
  - `void print(int n, int... numbers)` ⇒ Last ⇒ OK!
  - `void print(int... numbers, int n)` ⇒ Not last ⇒ Error!
  - `boolean areEqual(int... numbers, int... numbers)` ⇒ Error!
  - `int... clone(int... numbers)` ⇒ Error! Use `int[]` as return type
- Q: Why last in the parameter list?
- A: The JVM will otherwise have a hard time deciding where the list starts/ends.
- Consider the following call to `areEqual(...)` above

```
if ( areEqual(1,2,3,1,2,3) )    // Comparing {1,2,3} with {1,2,3}?
   ...                          // Or {1,2,3,1} with {2,3}?
                                // Or ...
```

A 10 minute break?

ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ

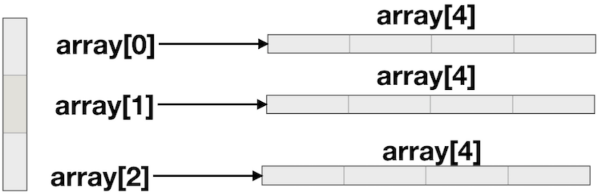
Multi Dimensional Arrays (1)

```
public static void main(String [] args) {
    int [][] m = new int [2][2];
    m [0][0] = 1;          // Upper left
    m [0][1] = 2;
    m [1][0] = 3;
    m [1][1] = 4;          // Lower right

    int n = m[1][1];
    System.out. println (n);    // 4

    int [][] a = { {1,2,3}, {4,5,6}, {7,8,9} };
    n = a [1][2];
    System.out. println (n);    // 6
}
```

- Error in code fragment above. It should be `int[][] m = ...` . That is, no space between `int` and `[][]` when declaring a multidimensional array.
- Create empty array of size 2 × 3: `int[][] m = new int[2][3];`
- Create and initialize array: `int[][] a = {{1,2,3},{4,5,6} {7,8,9}}`
- Access elements: `m[1][0] = 3` (write), `n = a[1][2]` (read)

Multi Dimensional Arrays (2)

Actually, arrays containing other arrays.



```
int[][] p = new int[3][4];              // 3 rows, 4 columns
System.out.println(p.length);           // 3

int[] arr = p[1];                       // Select row 1
System.out.println(arr.length);         // 4

arr[2] = 99;  // Add 99 to position [1][2]
```

Multi Dimensional Arrays (3)

```
int [][]  table = new int [5][7];  // row size 5, column size 7

/* Add values */
for (int i=0; i < table.length; i++) { // 5 rows
    int [] row = table[i];              // row i
    for (int j=0; j < row.length; j++)  // length
        row[j] = 5+i*j;                  // row position  j
}

/* Print values */
for (int i=0; i<table.length; i++) {     // for each row
    for (int j=0; j<table[i].length;j++) // for each row position
        System.out.print("\t"+table[i][j]);
   System.out.println ();                 // new line of print-out
}
```

- We think of 2D arrays as rectangular blocks of numbers (or data)
- We manipulated them as 1D arrays containing other arrays
- `int[][] table =  new int[5][7]` ⇒ table.length = 5 where each row is an array of size 7

The Class `java.util.ArrayList`

- The class `ArrayList` is a **list**
  ⇒ a growing sequential data structure
  ⇒ a flexible array that grows when needed.
- More about data structures in the next Java course.
- `ArrayList` belongs to the package `java.util` in the Java class library.
  ⇒ requires `import java.util.ArrayList;` or `import java.util.*;`
- `ArrayList` has methods for adding, accessing and removing data.
- `ArrayList` comes i two versions
  1. A **raw** version, where any type of elements can be stored
     ⇒ type casting (down-casting) is necessary when accessing an element
     ⇒ uses the class `Object` ⇒ all types of objects
  2. A **generic** version, where only a specified type of data can be stored
     ⇒ no type casting (down-casting) when accessing

Example `ListMain.java`

```
ArrayList  list  = new ArrayList();     // An empty, raw list
for (int i=1;i<=5;i++) {
    list .add(i );                      // add 1,2,3,4,5
}
System.out. println (  list . toString ()  );     // Print−out:  [1, 2, 3, 4, 5]

 list .add(2,99);                                 // add 99 at pos 2
System.out. println (  list . toString ()  );     // Print−out:  [1, 2, 99, 3, 4, 5]

 list .remove(1);                                 // remove 2 at pos 1
System.out. println (  list . toString ()  );     // Print−out:  [1, 99, 3, 4, 5]

System.out. println ("Size:  "+list . size ());       // Size: 5
int  first  = (Integer)  list .get(0);                // Get the element at pos 0
System.out. println ("Element 0: "+first );           // Print−out: Element 0: 1
```

Note: Accessing values requires a downcast and a type conversions

1. `Object --> Integer` (Explicit)
2. `Integer --> int` (Implicit)

Methods of the Class `ArrayList`

- `ArrayList()`: Constructor, creates an empty list
- `boolean add(Object obj)`: Adds `obj` at the end of the list
- `void add(int pos, Object obj)`: Adds `obj` at position `pos`
- `void clear()`: Removes all elements from the list
- `Object remove(int pos)`: Removes the element at position `pos`
- `Object get(int pos)`: Returns the element at position `pos`
- `int indexOf(Object obj)`: Index of the first (from the front) instance of `obj`
- `boolean contains(Object obj)`: `true` if `obj` is in the list, false otherwise
- `boolean isEmpty()`: `true` if the list is empty, false otherwise
- `int size()`: Returns the current number of elements in the list.

Note

- `Object obj` ⇒ any types of objects
- `list.add(77)` ⇒ 77 is casted into an `Integer` object before it is inserted
- Eclipse discourages the use of the raw type of `ArrayList`
  ⇒ will generate yellow warnings at the side of the code.

Example `GenericListMain.java`

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();   // An empty integer list
    for (int i=1; i<=5; i++)
        list.add(i);                         // Add 1,2,3,4,5. int --> Integer


    // Print all list elements
    for (int i=0; i<list.size(); i++) {
        int n = list.get(i);           // Integer --> int
        System.out.print(" "+n);
    }
}
```

- Generic ⇒ class with type papameter, e.g. `<Integer>`
- `ArrayList<Integer> list = new ArrayList<Integer>()`
  ⇒ we create a list that only can contain integers
  ⇒ Eclipse gets happy and stops giving yellow warnings

The Utility Class `Collections`

The utility class `java.util.Collections` has methods (e.g. `sort`, `toString`, `fill`) that can be used for sorting, printing and filling an `ArrayList` with values. Works in the same as class `java.util.Arrays` presented on slide 11.

Data Collections: Two Options

1. **Arrays**
   - Always have a fixed size.
   - Works for all type of data (primitive types and classes)
   - Data access using the [] operator.
   - Index starts at 0 and ends at `arr.length-1`
2. The class **java.util.ArrayList**
   - Has no fixed size ⇒ adjusts to the contents
   - Works for all type of data (primitive types and classes)
   - Two variants: 1) Raw, 2) Generic
   - Accessed using the ArrayList methods.
     (See the Java API for further information)
   - Index starts at 0 and ends at `list.size()-1`

Two types of `for` statements can be used on both arrays and `ArrayList`

```
for (int n : dataCollection){ || for (int i=0;i< "size" ;i++) {
  ... = n;                    ||     ...
}                             || }
```

An Old Java-test Exercise

Create a Java program `TwoMethods.java` containing a `main` method and two static methods:

- A method `ArrayList<Integer> roundOff(ArrayList<Double> input)` that returns a new integer list containing all the input doubles *correctly rounded off* to integers.
- A method `boolean hasDuplicates(int[] arr)` returning `true` if the array `arr` contains any duplicate elements (and `false` otherwise).

Also, present a `main` method that demonstrates how the two methods can be used.

Method roundOff

```
public static void main(String[] args) {
    // Round Off
    ArrayList<Double> dList = new ArrayList<Double>();
    dList.add(2.73); dList.add(3.1415);
    dList.add(-2992.64); dList.add(3.0); dList.add(4.5);

    ArrayList<Integer> iList = roundOff(dList);
    System.out.println(iList);
}

private static ArrayList<Integer> roundOff(ArrayList<Double> dL) {
    ArrayList<Integer> iL = new  ArrayList<Integer>();
    for (double d : dL) {
        int n = (int) Math.round(d);
        iL.add( n );
    }
    return iL;
}
```

Method hasDuplicates

```java
public static void main(String[] args) {
    // Has duplicates
    int[] a = {3,9,-1,6,9};
    if ( hasDuplicates(a) )
        System.out.println("a has duplicates");
}

private static boolean hasDuplicates(int[] arr) {
    for (int i=0; i<arr.length-1; i++) {
        int a = arr[i];
        for (int j=i+1;j<arr.length;j++) {
            int b = arr[j];
            if (a == b)
                return true;
        }
    }
    return false;
}
```

The Java Testet

- The first Java Test is 2019-12-13
- A practical programming test using material from lectures 1-6
  ⇒ Assignments 1 and 2.
- 2-3 exercises should be handled in 2 hours
- You are using your own laptop and your only help is the Java class library documentation.