Objektorienterad programmering

Lecture 7: multidimensional arrays and text files

Dr. Alex Gerdes | Dr. Carlo A. Furia

SP1 2017/18

Chalmers University of Technology

In the previous lecture 6

- Monodimensional arrays
- Strings

Text files: Input/Output

Data I/O (Input/Output)

- A program without input/output of data is not very useful: it can only perform a single predefined computation without communicating the result!
- There are several different ways of inputing data
  - input via keyboard
  - read the content of a file
  - through the network
  - getting the output of another program as input
- Some input sources can also be output sources
- Java supports I/O through multiple sources by means of several library classes. In this class we discuss some of them.

Input from keyboard

- In Java, `System.in` represents keyboard input, which is an object of type `InputStream`. Instead of using InputStream objects directly, it is simpler and more flexible to create objects of type `Scanner`.

Constructor	Description
`Scanner(InputStream source)`	Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters.

```java
import java.util.Scanner;

public class ReadFromKeyboard {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Give the integer numbers: ");
    int sum = 0;
    while (keyboard.hasNextInt()) {
      sum = sum + keyboard.nextInt();
    }
    System.out.println("The sum of the numbers: " + sum);
  }
}
```

Input from text files

- To read from text files, we combine library classes `File` and `Scanner`
- The constructor of Scanner may throw a `FileNotFoundException` if the file does not exist. This is an example of *exception* that has to be managed (checked).

Constructor	Description
`File(String pathname)`	Creates a new File instance by converting the given pathname string into an abstract pathname.

Constructor	Description
`Scanner(File source) throws FileNotFoundException`	Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters.

Reading numbers from a text file

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ReadFromTextFile {
  public static void main(String[] args) throws FileNotFoundException {
    File in = new File("indata.txt");
    Scanner sc = new Scanner(in);
    int sum = 0;
    while (sc.hasNext()) {
      sum = sum + sc.nextInt();
    }
    System.out.println("The sum is: " + sum);
  }
}
```

May throw
`FileNotFoundException`

Output to text files

CHALMERS

- To write to text file, we use library class `PrintWriter`

Constructor	Description
`PrintWriter(String fileName)` `throws FileNotFoundException`	Creates a new `PrintWriter`, without automatic line flushing, with the specified file name.

Operations	Description
`void close()`	Closes the stream and releases any system resources associated with it.
`void print(int i)`	Prints an integer.
`void print(double d)`	Prints a double-precision floating-point number.
…	
`void println(int i)`	Prints an integer and then terminates the line.
…	

Writing numbers to a text file

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class WriteToTextFile {
  public static void main(String[] args) throws FileNotFoundException {
    PrintWriter out = new PrintWriter("out.txt");
    for (int i = 0; i < 10; i = i + 1) {
      out.println(i);
    }
    out.close();
  }
}
```

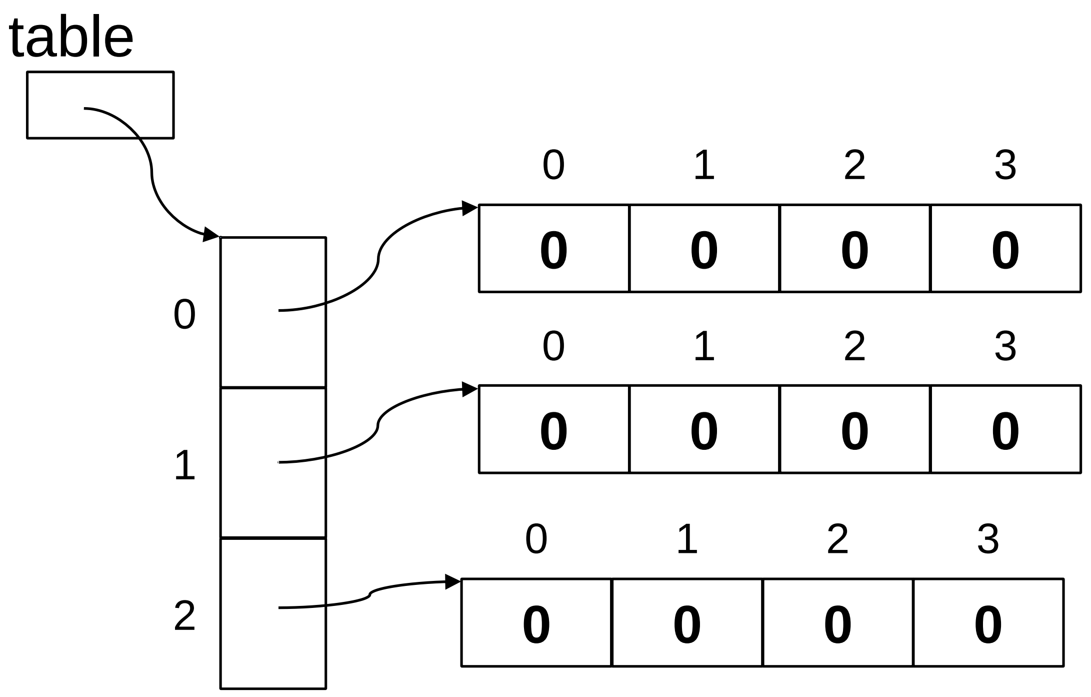
May throw `FileNotFoundException`

Multidimensional arrays

Bidimensional arrays

- A bidimensional array is an array of arrays:

```
int[][] table = new int[3][4];
```



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \end{array}$$

Bidimensional arrays

Instead of creating a bidimensional array with “new”, we can create and simultaneously fill it up with a certain content when we declare it:

```
int[][] numberTable =
   {{12, 34, 71, 9},
    {53, 43, 33, 68},
    {29, 10, 3,  42}};
```

	0	1	2	3
0	**12**	**34**	**71**	**9**
1	**53**	**43**	**33**	**68**
2	**29**	**10**	**3**	**42**

- Since a bidimensional array is just an array of references to independent arrays, the different components may have different lengths (“jagged arrays”):

```
int[][] jagged = {{12, 34, 71, 9},
                  {53, 43, 33},
                  {29, 10}};
```

	0	1	2	3
0	**12**	**34**	**71**	**9**
1	**53**	**43**	**33**	
2	**29**	**10**		

Bidimensional arrays

```
int[][] table = {{12, 34, 71, 9},
                 {53, 43, 33},
                 {29, 10}};
```

	0	1	2	3
0	**12**	**34**	**71**	**9**
1	**53**	**43**	**33**	
2	**29**	**10**		

`table[0].length` gives `4`
`table[1].length` gives `3`
`table[2].length` gives `2`

`Arrays.sort(table[0])` sorts row 0 in table
`Arrays.sort(table[1])` sorts row 1 in table
`Arrays.sort(table[2])` sorts row 2 in table

Example: symmetric matrixes

CHALMERS

- Write a program that inputs a square n x n matrix, and outputs whether the matrix is symmetric.
  Matrix A with elements aij is symmetric if:

  $$a_{ij} = a_{ji} \text{ for all indexes } i, j$$

- **Analysis**:
  - **Input**: An integer n (the matrix size) and a matrix A of size n x n
  - **Output**: Whether A is symmetric or not

- **Example**: Given

```
1   2   3
2   3   4
3   4   5
```

  the program outputs SYMMETRIC, whereas given

```
1   2   3
3   4   5
5   6   7
```

  the program outputs NOT SYMMETRIC

Design

- **Discussion**: The result can be encoded by a boolean ok, which is true if A is symmetric and false if A is not symmetric.
  Checking symmetry requires to check all pairs of elements aij, aji. Thus, we initialize ok to true and loop through all pairs of elements.If we find a pair of elements such that $a_{ij} \neq a_{ji}$ it means that the matrix is not symmetric: thus, we set ok to false and exit the loop. If we manage to complete the loop without ever setting ok to false, ok is still true, which means the matrix is indeed symmetric.

- **Algorithm**:

```
1. Read size n
2. Read matrix A
3. ok = true;
4. For every pair of indexes i, j in matrix A:
    1. if ($a_{ij} \neq a_{ji}$) ok = false;
5. if ok
       Print ”SYMMETRIC”.
   else
       Print ”NOT symmetric”
```

- **Data**:
  - `n of type int`
  - `A` of type `double[][]`

Implementation: main

```
import javax.swing.*;

public class Symmetric {
  public static void main(String[] args) {
    String input = JOptionPane.showInputDialog(“Input matrix size: ");
    int n = Integer.parseInt(input);
    double[][] A = readMatrix(n);
    if (isSymmetric(A))
      JOptionPane.showMessageDialog(null, “SYMMETRIC");
    else
      JOptionPane.showMessageDialog(null, “NOT symmetric");
  }
```

Implementation: `readMatrix`

```
public static double[][] readMatrix(int size) {
  double[][] theMatrix = new double[size][size];
  for (int row = 0; row < size; row = row + 1) {
   for (int col = 0; col < size; col = col + 1) {
    String q = “Input element  (" + row + “row, " + col + ”col )";
    String input = JOptionPane.showInputDialog(q);
    theMatrix[row][col] = Double.parseDouble(input);
   }
  }
  return theMatrix;
 }
```

Implementation: `isSymmetric`

```java
  // before: matrix != null
  public static boolean isSymmetric(double[][] matrix) {
    boolean okay = true;
    for (int row = 0; row < matrix.length; row = row + 1)
      for (int col = 0; col < matrix[row].length; col = col + 1)
        if (matrix[row][col] != matrix[col][row])
          okay = false;
    return okay;
  }
}
```

Implementation: `isSymmetric – ver. 2`

With the previous implementation, we end up checking each pair aij, aji twice. We can avoid that by only looping over the lower-half of matrix A.

```java
// before: matrix != null
public static boolean isSymmetric(double[][] matrix) {
 boolean okay = true;
 for (int row = 0; row < matrix.length; row = row + 1)
   for (int col = 0; col <= row && col < matrix[row].length; col = col + 1)
     if (matrix[row][col] != matrix[col][row])
         okay = false;
 return okay;
}
```

Other implementation of `readMatrix`

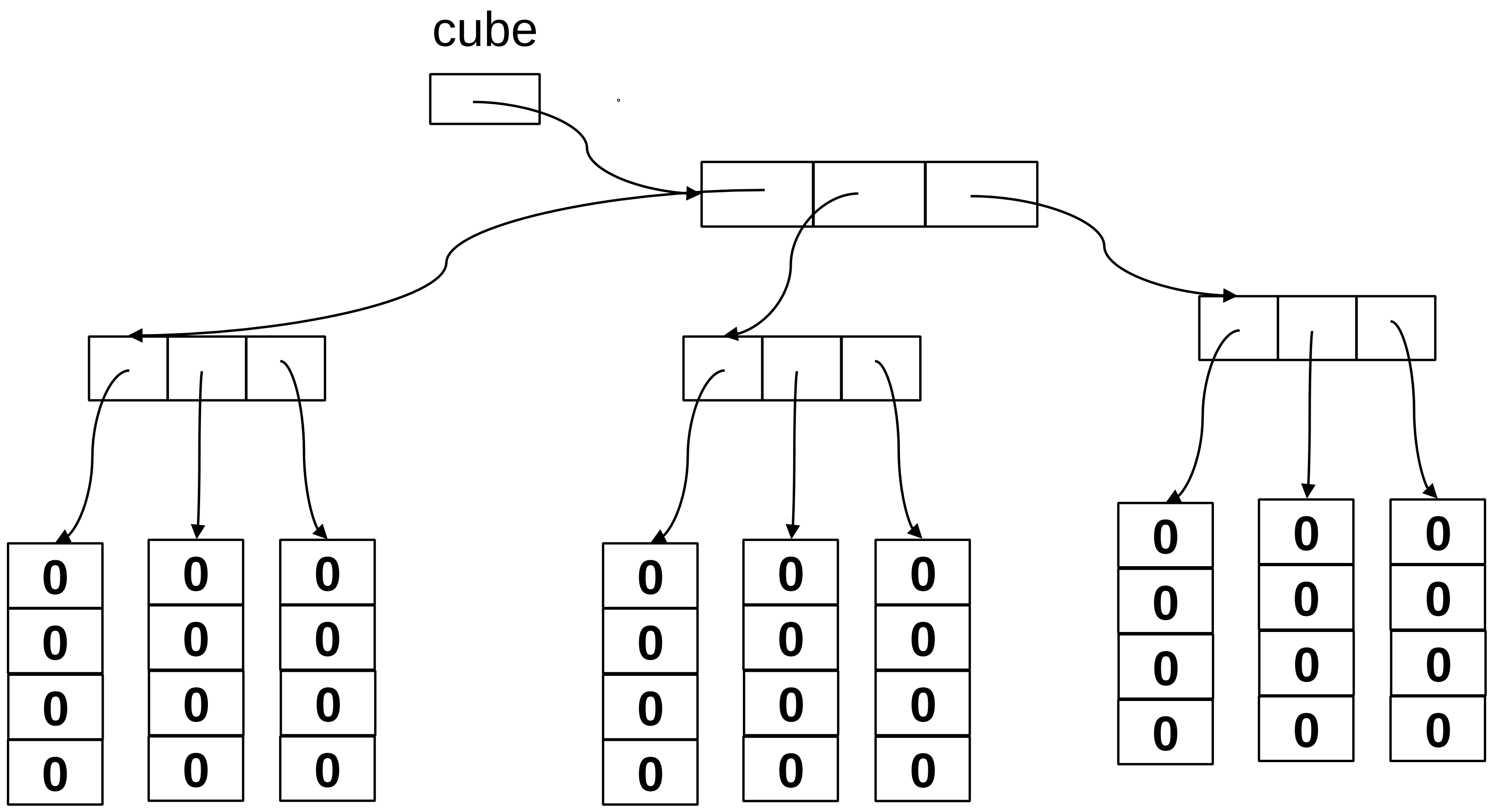
Using a
`Scanner object`

```
import java.util.*;
...
public static double[][] readMatrix(int size) {
  double[][] theMatrix = new double[size][size];
  String input = JOptionPane.showInputDialog( "Input elements: ");
  Scanner sc = new Scanner(input);
  for (int row = 0; row < size; row = row + 1) {
    int col = 0;
    while (col < size) {
      if (sc.hasNextDouble()) {
        theMatrix[row][col] = sc.nextDouble();
        col = col + 1;
      } else {
        input = JOptionPane.showInputDialog( "Input more elements: ");
        sc = new Scanner(input);
      }
    }
  }
  return theMatrix;
}
```

Multidimensional arrays

- Arrays can have more than two dimensions: an array of arrays of arrays of....

```
int[][][] cube = new int[3][3][4];
```



Multidimensional arrays

- A picture can be represented as a bidimensional array of color dots ("pixels")
- In black-and-white (grayscale) pictures, each pixel is a value in the interval [0, 255], where 0 is black and 255 is white
- In color pictures, each pixel is three values in the interval [0, 255], representing the intensity of red, green, and blue
- Example: grayscale and color pictures of size 800x600 pixels:

```
int[][] grayImage = new int[800][600];
int[][][] colorImage = new int[800][600][3];
```

ArrayList

Class `ArrayList`

- An array is a *static data structure*, whose size is fixed upon creation and cannot be changed while the program executes. In some applications, we may not know the size of the data when the program starts, and thus we need *dynamic data structures*, whose size can grow and shrink as the program needs.

- We could “simulate” a dynamic structure using an array, for example:
  - Create an array that is as large as the maximum data size
  - Keep track of how which elements are actually added to the array
  - To add an element: use an empty slot
  - To remove an element: mark a slot as empty

- Class `ArrayList` is a library class that provides a flexible implementation of dynamic list data structure. Whenever we need dynamic data management, it’s usually much simpler to use `ArrayList` instead of (monodimensional) arrays

- `ArrayList` is in package `java.util`

Class `ArrayList`

- Clarr `ArrayList` is *generic*.This means that we can create lists of elements of any type (like for arrays). When we declare a variable of type ArrayList, we also declare the type of its elements. Examples:

```
ArrayList<String> words = new ArrayList<String>();
ArrayList<Integer> values = new ArrayList<Integer>();
ArrayList<BigInteger> bigValues =
                   new ArrayList<BigInteger>();
ArrayList<Person> members = new ArrayList<Person>();
```

- `ArrayList` can only store object/reference types, not primitive types (such as `int`, `double`, `boolean` and `char`)

- When we need a list with elements of a primitive type, we use its corresponding *wrapper* type instead

Class `ArrayList<E>`

Operation	Description
`ArrayList<E>()`	Create an empty `ArrayList` for elements of type `E`
`void add(E elem)`	Add elem as last element of the list (after all other elements)
`void add(int pos, E elem)`	Insert elem at position pos in the list, shifting all other elements at the insertion point to the right
…	…
`E get(int pos)`	Return element at position pos
`E set(int pos, E elem)`	Replace the element currently at pos with elem
`E remove(int pos)`	Remove the element at position pos, shifting all other elements at the removal point to the left

Class `ArrayList<E>`

Operation	Description
`int size()`	Return the number of elements in the list
`boolean isEmpty()`	Return true if the list is empty, otherwise return false
`int indexOf(Object elem)`	Return the position (index) of elem in the list; if elem is not in the list, return -1
`boolean contains(Object elem)`	Return true of elem is in the list, otherwise return false
`void clear()`	Remove all elements in the list
`String toString()`	Return a textual representation of the list content in the form [e_1, e_2, . . . , e_n]

Methods indexOf and contains compare elem to the elements in the list using a method

```
public boolean equals(Object obj)
```

which must be defined for the class E. All standard classes such as `String`, `Integer` and `Double`, include a definition of equals.

Autoboxing and autounboxing

- We can often mix primitive types and their corresponding wrapper types thanks to *autoboxing* and *autounboxing*

-
  ```
  Integer talObjekt = new Integer(10);
                 // without autoboxing
  ...
  int tal = talObjekt.toValue();
                // without autounboxing
  ```

Equivalently, and more simply:

```
Integer talObjekt = 10;     // autoboxing
...
int tal = talObjekt;        // auto-unboxing
```

For loop over collections (for each)

- A special form of the for loop is convenient to loop over every element of an array, or of a list like ArrayList

```
double[] values = new double[100];
ArrayList<String> listan = new ArrayList<String>();

// For loops with explicit index
for (int index = 0; index < values.length; index = index +1) {
  System.out.println(values[index]);
}
for (int pos = 0; pos < listan.size(); pos = pos +1) {
  System.out.println(listan.get(pos));
}

// For loops with "for each"
for (double v : values)          // for each v in values
  System.out.println(v);

for (String str : listan)        // for each str in listan
  System.out.println(str);
```

Example: read a set of numbers

CHALMERS

- Write a method with signature

  ```
  private static ArrayList<Integer> readSet()
  ```

  that reads integers in any order, and returns a list where all read integers appear exactly once:

  - If an element is read multiple times, it appears only once in the output
  - If an element is read once, it appears once in the output
  - If an element is not read, it does not appear in the output

- **Example**: input integers
  1, 4, 1, 2, 4, 5, 12, 3, 2, 4, 1

  output list:
  1, 4, 2, 5, 12, 3

Analysis and implementation

- **Algorithm**:
  1. while (there are more integers)
     1. Read the next number
     2. if (the number is not already in the output list)
        add the number to the list;
  2. Return the output list

```
public static ArrayList<Integer> readSet() {
  ArrayList<Integer> set = new ArrayList<Integer>();
  Scanner in = new Scanner(System.in);
  while (in.hasNextInt()) {
    int value = in.nextInt();
    if (!set.contains(value)) {
      set.add(value);
    }
  }
  return set;
}
```

Class `PhoneBook` implemented with arrays

```
public class Entry {
  private String name;
  private String number;
  public Entry(String name, String number) {
    this.name = name;
    this.number = number;
  }
  public String getName() {
    return name;
  }
  public String getNumber() {
    return number;
  }
}
```

```
public  class PhoneBook {
  private Entry[] book;
  private int count;
  public PhoneBook(int size) {
    book = new Entry[size];
    count = 0;
  }
  public void put(String name, String nr) {
    book[count] = new Entry(name, nr);
    count = count + 1;
  }
  public String get(String name) {
    String res = null;
    for (int i = 0; i < count; i = i +1)
      if (name.equals(book[i].getName()))
        res = book[i].getNumber();
    return res;
  }
}
```

Maximum number of entries

Actual number of stored elements

Runtime error if count >= size

Class PhoneBook implemented with ArrayList

```java
public class Entry {
  private String name;
  private String number;
  public Entry(String name, String number) {
    this.name = name;
    this.number = number;
  }
  public String getName() {
    return name;
  }
  public String getNumber() {
    return number;
  }
}
```

```java
import java.util.ArrayList;

public class PhoneBook {
  private ArrayList<Entry> book = new ArrayList<Entry>();

  public void put(String name, String nr) {
    book.add(new Entry(name, nr));
  }

  public String get(String name) {
    String res = null;
    for (Entry e : book)
      if (name.equals(e.getName()))
        res = e.getNumber();
    return res;
  }
}
```

Shorthand operators

Shorthand operators

- Shorthand operators are more concise forms of assignments
- There are shorthand operators for *increment* and *decrement*, each in *prefix* and *postfix* version

Shorthand	***Full form***
`++x`	`x + 1`
`--x`	`x - 1`
`x++`	`x + 1`
`x--`	`x - 1`
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Shorthand operators

- The difference between the prefix and postfix operators is *when* the increment or decrement is executed within an expression

- With the *prefix* operators, the increment/decrement occurs first, and then the whole expression is evaluated:

```
firstNumber = 10;
secondNumber = ++firstNumber;
```

  In the end, firstNumber == secondNumber == 11

- With the *prostfix* operators, the whole expression is evaluated first, and then the increment/decrement occurs (without affecting the value of the expression)

```
firstNumber = 10;
secondNumber = firstNumber++;
```

  In the end, secondNumber == 10 and firstNumber == 11

The most common, and simple, usage of the prefix/postfix operators is as stand-alone statements:

```
++firstNumber; firstNumber++
```

- The behavior of complex combinations of pre- and postfix operators can be quite tricky. Rule of thumb: only use them in simple expressions!