From Procedural Programming to Object-Oriented Programming (OOP)

A preliminary step by step approach

ISEP/LEI/ESOFT

Adapted from Paulo Maio's original version

Content Overview

- **Procedural Programming**
  - **Revision**
- Systematization
- Raising the need for OOP
  - Classes as Data Structures
- Towards OOP
  - Primary Concepts and Principles

While practicing
  - Main Software Engineering Activities
  - Promoted Working Method

Procedural Programming

Revision

Assumed Knowledge (1/3)

- Algorithm
  - Is a finite set of instructions, executed in a sequence controlled by
    - Conditional structures (e.g. if, switch)
    - Repetitive structures (e.g. for, while)

- Variable
  - Is a mean to store value(s) of a given data type
  - Scope (e.g. local vs. global)
  - Data Type
    - e.g. int, long, char, double, boolean

Assumed Knowledge (2/3)

- Operator
  - Attribution: used to set values to variables (=)
  - Arithmetic: i.e. sum (+), subtraction (-), multiplication (*), division (/)
  - Relational: e.g. equal (==), different (!=), greater than (>), lesser that (<)
  - Logic: i.e. *and*, *or*, *not*

- Array
  - Used to store multiple values of the same data type
  - *Array,* referring to a uni-dimensional array (e.g. *int[] v = {56, 21, 3}*)
  - *Matrix,* referring to a bi-dimensional array (e.g. *int[][] m = { {5, 6, 7}, {3, 2, 1} }*)

Assumed Knowledge (3/3)

- Problem Decomposition
  - Splitting a problem in a set of smaller sub-problems, and so on…
    - Single, but large, method/function → multiple, but thin, methods/functions
  - Concern separation between
    - Data Input – reading data from application users
    - Data Processing – computing the data to achieve results
    - Data Output – presenting the results to application users

- Automatic Tests
  - As a means to validate data processing methods
  - As a means to early detect regression in the software being developed

Function Parameters and Data Types

- Primitive types (e.g. int, char) – changes made inside the functions (e.g. *changeValues*) **are not visible** from the outside (e.g. *main*)
- Reference types (e.g. arrays) – changes made inside the functions **are visible** from the outside

```
private static void changeValues(int a, int[] array) {
   a = a * 3;
   array[0] = 7;
}
```

Check the class **FunctionParameters** in the package **slides**

```
public static void main(String[] args) {
   int a = 4;
   int[] array = {2, 9};
   changeValues(a, array);
   System.out.println(a);  // The output is "4" and not "12"
   System.out.println(array[0]);  // The output is "7" and not "2"
}
```

Function Parameters and Data Types

- Primitive types (e.g. int, char) – changes made inside the functions (e.g. *changeValues*) **are not visible** from the outside (e.g. *main*)
- Reference types (e.g. arrays) – changes made inside the functions **are visible** from the outside

```
private static void changeValues(int a, int[] array) {
   a = a * 3;
   array[0] = 7;
}
```

```
public static void main(String[] args) {
   int a = 4;
   int[] array = {2, 9};
   changeValues(a, array);
   System.out.println(a);  // The output is “4” and not “12”
   System.out.println(array[0]);  // The output is “7” and not “2”
}
```

Check the class **FunctionParameters** in the package **slides**

Change the parameter names of *changeValues* function. What happens?

Problem I

Focus on Software Engineering Activities and Working Method

SW Development using TDD

Requirements

- Goal
  - Develop a method to ascendingly sort a given array of integer numbers
- Acceptance Criteria
  - AC1. Method should return the same array as the one received as input
  - AC2. There should be a set of tests to verify that such method is correct
- Suggested approach
  - Adopt a TDD approach (tests are written first than code)

Analysis & Design

- Analysis
  - Input: an integers array
  - Output: the same integers array, but sorted (from AC1)

- Design (Method Header)

```
public static int[] sortArrayAscending(int[] array)
```

- Check Acceptance Criteria
  - The design contributes to satisfy AC1 but not ensures it

Instructions on how to proceed (1/2)

- First take a look at the following test cases

Test Cases – Considered Scenarios

- Possibilities for inputs
  - No array of numbers is provided (using “null”)
  - An empty array
  - An array with just one element
  - An array with two elements already sorted
  - An array with two elements incorrectly sorted
  - An array with several elements incorrectly sorted
  - (more…)
- Possibilities for outputs
  - ~~A distinct array of the one received as input, thus violating AC1~~

Test Cases – Specification (1/6)

- Scenario
  - No array of numbers is provided (using “null”)
- Decisions with impact in Design/Implementation
  - The return should also be “null”

```
@Test
public void ensureSortingNullArrayReturnsNull() {
   // Act
   int[] result = ProblemOne.sortArrayAscending(null);
   // Assert
   assertNull(result); // check result is null
}
```

Test Cases – Specification (2/6)

- Scenario
  - An empty array
  - AC1. Method should return the same array as the one received as input

```
@Test
public void ensureSortingAnEmptyArrayWorks() {
   // Arrange
   int[] data = {};
   int[] expected = {};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the
same as output

   assertEquals(expected.length, result.length); // check
dimension

   assertArrayEquals(expected, result); // check array
content
}
```

Test Cases – Specification (2/6)

- Scenario
  - An empty array
  - AC1. Method should return the same array as the one received as input

Focus/Check AC1

```
@Test
public void ensureSortingAnEmptyArrayWorks() {
   // Arrange
   int[] data = {};
   int[] expected = {};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the same as output

   assertEquals(expected.length, result.length); // check dimension

   assertArrayEquals(expected, result); // check array content
}
```

Test Cases – Specification (3/6)

- Scenario
  - An array with just one element

```
@Test
public void ensureSortingOneElementArrayWorks() {
   // Arrange
   int[] data = {4};
   int[] expected = {4};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the
same as output

   assertEquals(expected.length, result.length); // check
dimension

   assertArrayEquals(expected, result); // check array
content
}
```

Test Cases – Specification (4/6)

- Scenario
  - An array with two (already sorted) elements

```
@Test
public void ensureArrayWithTwoSortedElementsWorks() {
   // Arrange
   int[] data = {-1, 4};
   int[] expected = {-1, 4};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the same as output

   assertEquals(expected.length, result.length); // check dimension

   assertArrayEquals(expected, result); // check array content
}
```

Test Cases – Specification (5/6)

- Scenario
  - An array with two (unsorted) elements

```
@Test
public void ensureArrayWithTwoUnsortedElementsWorks() {
   // Arrange
   int[] data = {30, 25};
   int[] expected = {25, 30};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the same as output

   assertEquals(expected.length, result.length); // check dimension

   assertArrayEquals(expected, result); // check array content
}
```

Test Cases – Specification (6/6)

- Scenario
  - An array with several (unsorted) elements

```
@Test
public void ensureArrayWithSeveralUnsortedElementsWorks() {
   // Arrange
   int[] data = {30, 25, 25, -1, 20};
   int[] expected = {-1, 20, 25, 25, 30};

   // Act
   int[] result = ProblemOne.sortArrayAscending(data);

   // Assert
   assertSame(data, result); // check the input array is the same as output

   assertEquals(expected.length, result.length); // check dimension

   assertArrayEquals(expected, result); // check array content
}
```

Instructions on how to proceed (2/2)

- Go to the class **ProblemOneTest** in the **problem.one.version.one** test package of the lab project
- Run the tests
  - All tests have been disabled using the @Disabled annotation
  - All will be skipped
- Go one test at a time
  - Delete the @Disabled annotating from the first test and run your tests
  - Once you have implemented enough code for the test to pass, proceed to the next one
- Don’t forget – always run all your class tests, to ensure you don’t break any previous tests

Problem I – Implementation

- Once you start running the tests, you will see that some do not require any code changes for the test method to pass.

- On the next slide you can check a possible solution for the problem. Try to achieve it on your own.

Problem I – Solution

- Check out this solution, implemented in the class **ProblemOneSolution**, in the **problem.one.version.one** package of the lab project.
- It is not perfect and could be better.
- Do you think you can make it more modular?

```
public static int[] sortArrayAscending(int[] array) {
   if (array != null) {
      int temp = 0;

      int arraySize = array.length;
      //Sort the array in ascending order using two for loops
      for (int i = 0; i < arraySize; i++) {
         for (int j = 0; j < arraySize - i - 1; j++) {
            if (array[j] > array[j + 1]) {
               //swap elements if not in order
               temp = array[j];
               array[j] = array[j + 1];
               array[j + 1] = temp;
            }
         }
      }
   }
   return array;
}
```

Problem I – Better Solutions

- Try to split the method into different modules. To do this, implement the missing code in subsequent versions (two, three and four) of the lab project.

- If you need to, check the **ProblemOneSolution** classes for some possible solutions, available in the following packages:
  - **problem.one.version.two**
  - **problem.one.version.three**
  - **problem.one.version.four**