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

Systematization

User Stories

User Story – What is it?

- A way to express functional requirements of an application
  - Although, it can be used to express non-functional requirements too
- Written from the perspective of an application user (end-users)
  - Roles and/or personas
- Using informal natural language and domain/business jargon
- Follows a common and well-known template:
  - As a <user role>, I <something to do>.
  - As a <user role>, I <something to do>, so that <benefit>.
- Desirably, some acceptance criteria must exist

User Story – What is it?

- A way to express functional requirements of an application
  - Although, it can be used to express non-functional requirements too
- Written from the perspective of an application user (end-users)
  - Roles and/or personas
- Using informal natural language and domain/business jargon
- Follows a common and well-known template:
  - As a <user role>, I <something to do>.
  - As a <user role>, I <something to do>, so that <benefit>.
- Desirably, some acceptance criteria must exist

Raising the need for OOP

Classes as Data Structures

Problem II

Requirements

- Context:
  - An application to manage the students (number and name) of a course unit and their grades is required. Intended functionalities comprehends adding, updating, removing and listing students sorted and/or filtered by some criteria, among others.

- User Stories (US):
  - US01 – As a user, I want to list students sorted by their number (ascendent).
  - US02 – As a user, I want to list students sorted by their grades (descendent).

- Remarks and Acceptance Criteria (common to all US):
  - No User Interface (UI) is required
  - Solution must have a testing coverage and mutation coverage of, at least, 90%

Analysis

- Data to be managed (example)

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
....	....	....
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

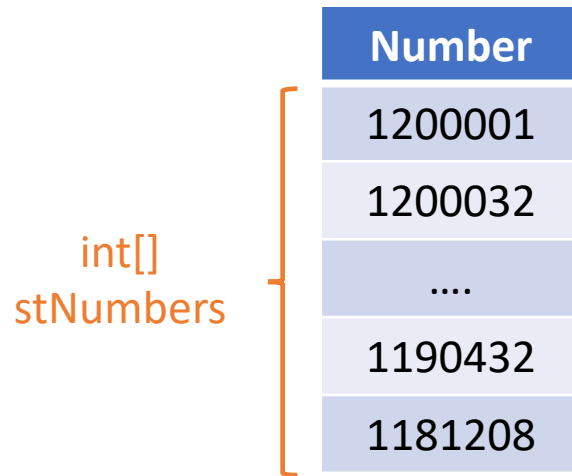- Used as input on US01 and US02
- Used as output on US01 and US02

Design – How to represent the students data? (using known knowledge, so far)

- Through a matrix?
  - Is it possible?
  - Of which data type? Integers? Strings? Other?

Design – How to represent the students data? (using known knowledge, so far)

- Through a matrix?
  - Is it possible?
  - Of which data type? Integers? Strings? Other?
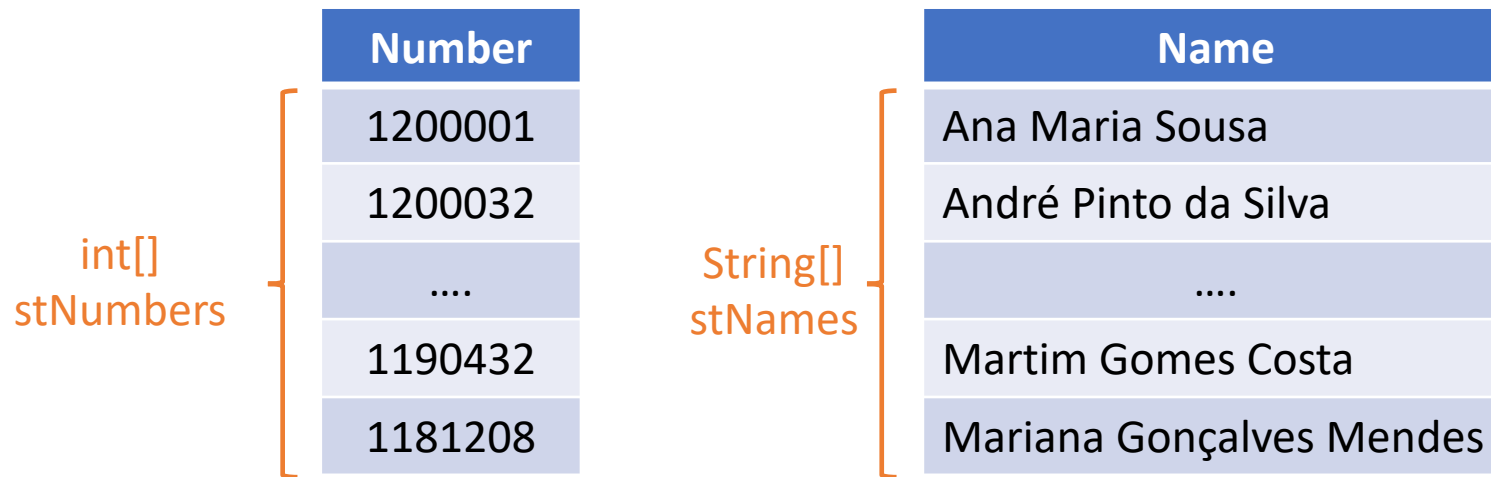- Through 3 arrays:

int[]
stNumbers

Number
1200001
1200032
….
1190432
1181208

Design – How to represent the students data? (using known knowledge, so far)

- Through a matrix?
  - Is it possible?
  - Of which data type? Integers? Strings? Other?
- Through 3 arrays:

Number
1200001
1200032
….
1190432
1181208

int[] stNumbers

Name
Ana Maria Sousa
André Pinto da Silva
….
Martim Gomes Costa
Mariana Gonçalves Mendes

String[] stNames

Design – How to represent the students data? (using known knowledge, so far)

- Through a matrix?
  - Is it possible?
  - Of which data type? Integers? Strings? Other?
- Through 3 arrays:

int[] stNumbers	String[] stNames	int[] stGrades
Number	**Name**	**Grade**
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
….	….	….
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Design – How to represent the students data? (using known knowledge, so far)

- Through a matrix?
  - Is it possible?
  - Of which data type? Integers? Strings? Other?
- Through 3 arrays:

Number		Name		Grade
1200001		Ana Maria Sousa		16
1200032		André Pinto da Silva		12
….		….		….
1190432		Martim Gomes Costa		17
1181208		Mariana Gonçalves Mendes		14

int[] stNumbers

String[] stNames

int[] stGrades

Information of a single student is split by 3 distinct data structures
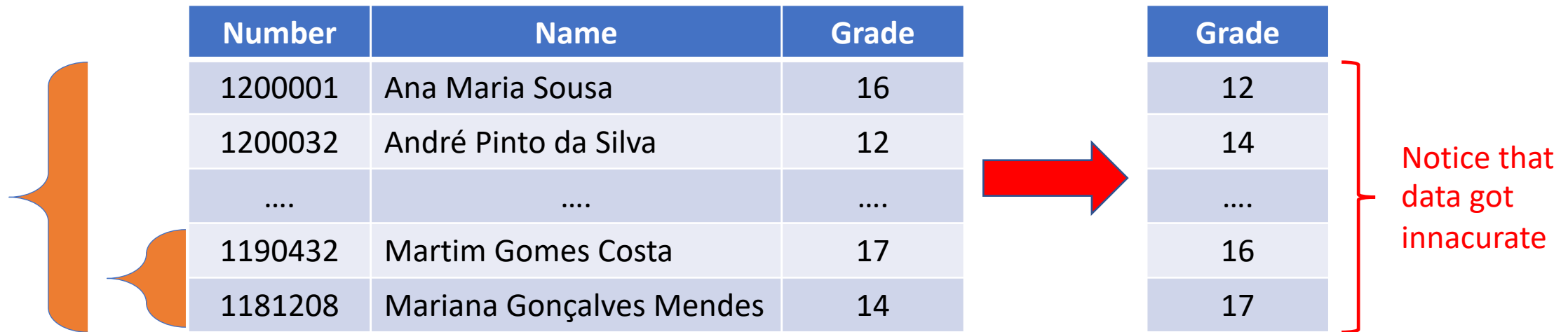
Design – (cont.)

- Can we apply the method "*sortArrayAscending*" to each array to accomplish US01?

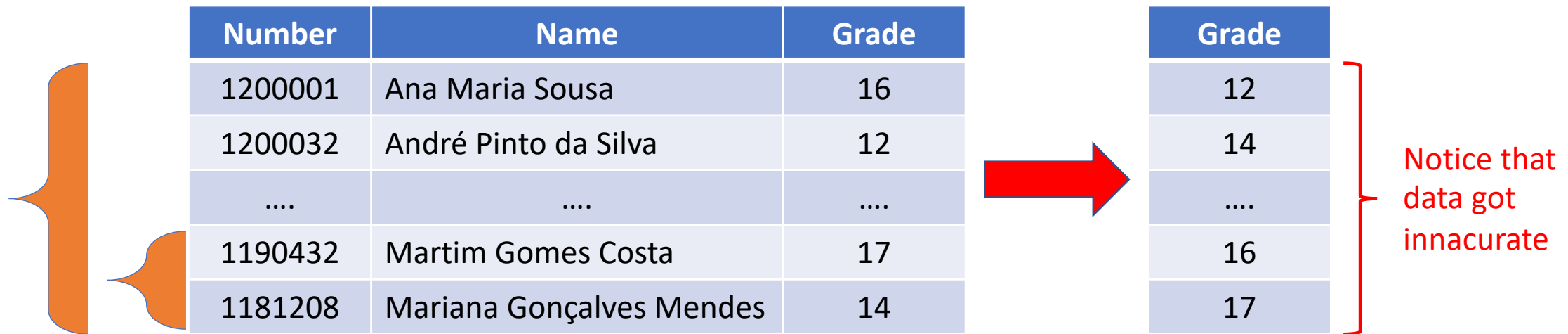| Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
....	....	....
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Design – (cont.)

- Can we apply the method "*sortArrayAscending*" to each array to accomplish US01?
  - No! Why? What would happen?

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
….	….	….
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Design – (cont.)

- Can we apply the method "*sortArrayAscending*" to each array to accomplish US01?
 - No! Why? What would happen?

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
….	….	….
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Grade
12
14
….
16
17

Notice that data got innacurate

Design – (cont.)

- Can we apply the method "*sortArrayAscending*" to each array to accomplish US01?
  - No! Why? What would happen?

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
….	….	….
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Grade
12
14
….
16
17

Notice that data got innacurate

- What "links" the student number to its name and/or grade? Or vice-versa?

Design – (cont.)

- Can we apply the method "*sortArrayAscending*" to each array to accomplish US01?
  - No! Why? What would happen?

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
….	….	….
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Grade
12
14
….
16
17

Notice that data got innacurate
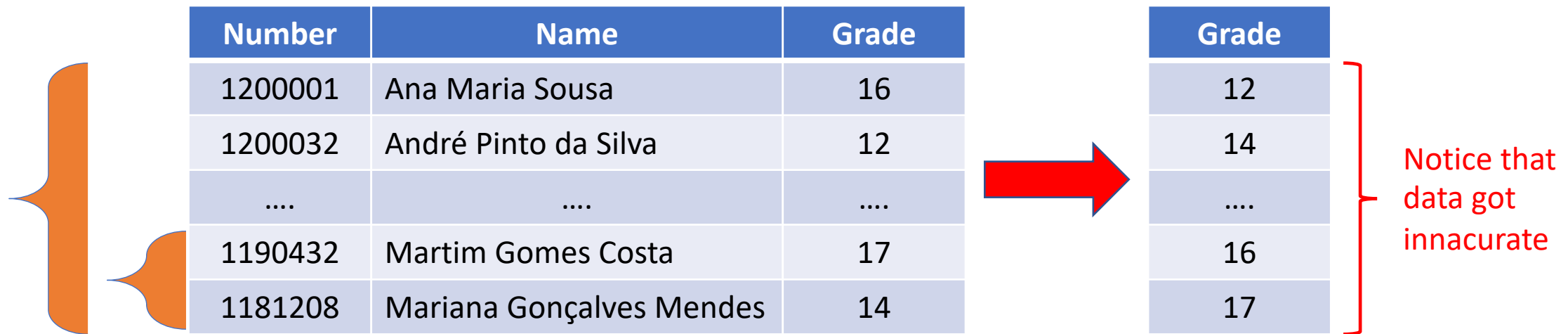
- What "links" the student number to its name and/or grade? Or vice-versa?
  - Having the **same position** into the arrays used to represent students data. **Is it a reliable approach?**

Design – This would result in the following

Number
1200001
1200032
….
1190432
1181208

int[]
stNumbers

Name
Ana Maria Sousa
André Pinto da Silva
….
Martim Gomes Costa
Mariana Gonçalves Mendes

String[]
stNames

Grade
16
12
….
17
14

int[]
stGrades

Design – This would result in the following

Number
1200001
1200032
….
1190432
1181208

int[] stNumbers

Name
Ana Maria Sousa
André Pinto da Silva
….
Martim Gomes Costa
Mariana Gonçalves Mendes

String[] stNames

Grade
16
12
….
17
14

int[] stGrades

Number
1181208
1190432
….
1200001
1200032

Name
Ana Maria Sousa
André Pinto da Silva
….
Mariana Gonçalves Mendes
Martim Gomes Costa

Grade
12
14
….
16
17

Design – This would result in the following

int[]
stNumbers

String[]
stNames

int[]
stGrades

Number	Name	Grade
1200001	Ana Maria Sousa	16
1200032	André Pinto da Silva	12
....	....	
1190432	Martim Gomes Costa	17
1181208	Mariana Gonçalves Mendes	14

Check the classes

ProblemTwo and **ProblemTwoTest**

in both <u>package</u> and <u>test package</u> **problem.two.version.one**

for understanding this problem.

Number	Name	Grade
1181208	Ana Maria Sousa	12
1190432	André Pinto da Silva	14
....	....	....
1200001	Mariana Gonçalves Mendes	16
1200032	Martim Gomes Costa	17
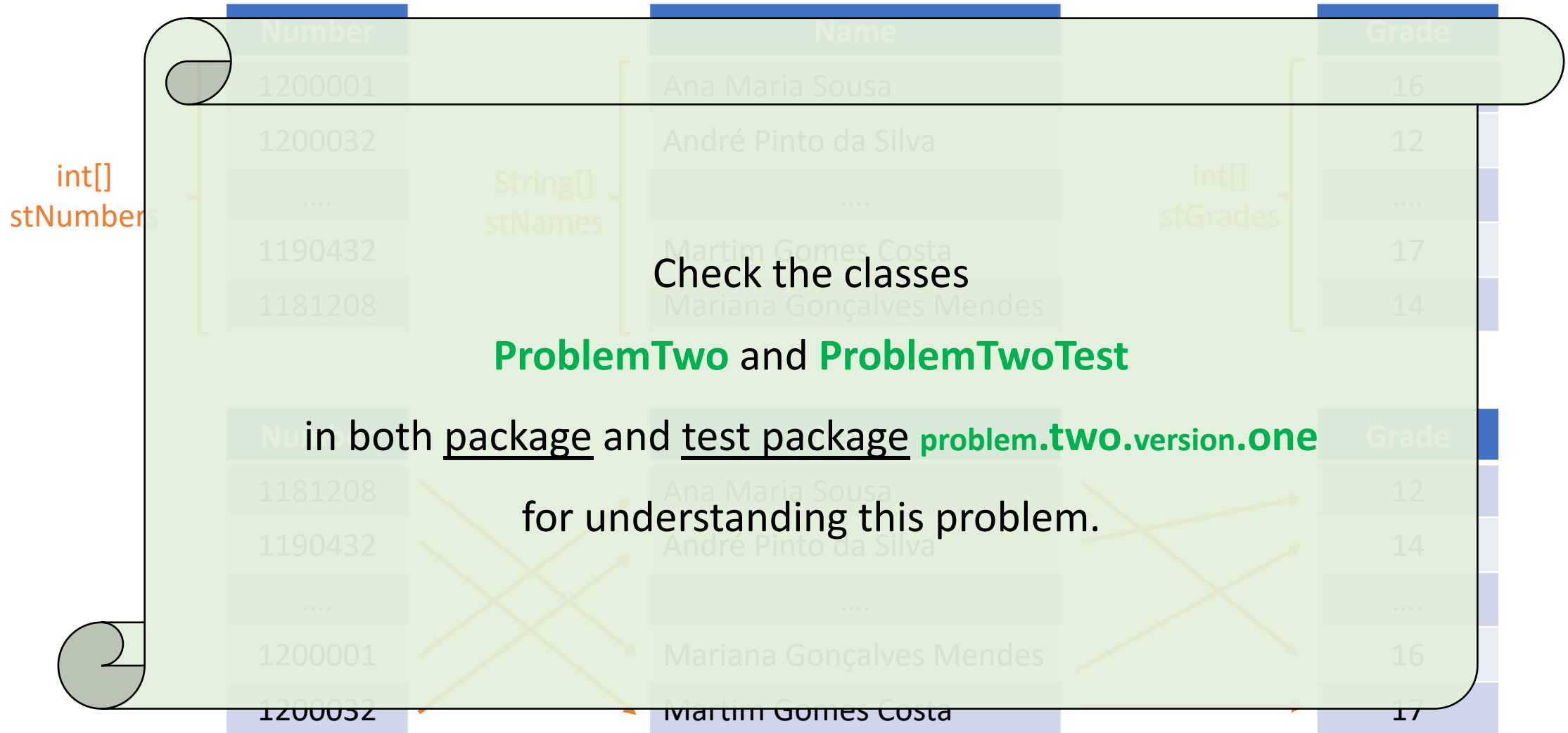
Design Alternatives (Method Signature)

- Alternative I:

  void sortStudentsByAscendingNumber(int[] studentNumbers, String[] studentNames, int[] studentGrades)

- Alternative II:

  boolean sortStudentsByAscendingNumber(int[] studentNumbers, String[] studentNames, int[] studentGrades)

- In both, all arrays are required and passed by reference as so changes made inside the method are visible from the outside
- Return value might be used to signal that something went wrong while sorting

Instructions on how to proceed

- Run the tests of the class **ProblemTwoTest** in the **problem.two.version.two** test package
  - 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
  - Implement the class **ProblemTwo** in the package **problem.two.version.two** with enough code for the tests to pass
  - Once you have implemented, proceed to the next test

- Don't forget, always run all your class tests, to ensure you don't break any previous tests

Possible Implementation for version two

```java
public static boolean sortStudentsByAscendingNumber(int[] studentNumbers, String[] studentNames,
                                                    int[] studentGrades) {
    if ((studentNumbers == null) || (studentNames == null) || (studentGrades == null)) {
        return false;
    }

    int arraySize = studentNumbers.length;
    //Sort the studentNumbers in ascending order using two for loops
    for (int i = 0; i < arraySize; i++) {
        for (int j = 0; j < arraySize - i - 1; j++) {
            if (studentNumbers[j] > studentNumbers[j + 1]) {
                //swap elements if not in order - in studentNumbers
                swapIntArrayElements(studentNumbers, j, j + 1);
                //swap elements jf not jn order - in studentNames
                swapStringArrayElements(studentNames, j, j + 1);
                //swap elements jf not jn order - in studentGrades
                swapIntArrayElements(studentGrades, j, j + 1);
            }
        }
    }
    return true;
}
```

Possible Implementation for version two

```java
public static boolean sortStudentsByAscendingNumber(int[] studentNumbers, String[] studentNames,
                                                    int[] studentGrades) {
    if ((studentNumbers == null) || (studentNames == null) || (studentGrades == null)) {
        return false;
    }

    int arraySize = studentNumbers.length;
    //Sort the studentNumbers in ascending order using two for loops
    for (int i = 0; i < arraySize; i++) {
        for (int j = 0; j < arraySize - i - 1; j++) {
            if (studentNumbers[j] > studentNumbers[j + 1]) {
                //swap elements if not in order - in studentNumbers
                swapIntArrayElements(studentNumbers, j, j + 1);
                //swap elements jf not jn order - in studentNames
                swapStringArrayElements(studentNames, j, j + 1);
                //swap elements jf not jn order - in studentGrades
                swapIntArrayElements(studentGrades, j, j + 1);
            }
        }
    }
    return true;
}
```

Swapping elements is required for all arrays

Reflection about this approach (1/4)

- Is it error prone?
  - A lot!
  - Arrays may have different sizes!
  - We are considering three individual arrays as a unit or work
    - All arrays are coupled to each other
  - Adding one element to an array requires adding an element to all the other
    - Syncing this operation is really hard this way

Reflection about this approach (2/4)

- Is it easy to maintain and evolve?
    - Not really!
    - Tests are difficult to achieve, they seem to have a lot of duplicate code!
    - If we need an array of a different type (i.e. Names) makes this really difficult.

Reflection about this approach (3/4)

- How would it be if one need to store several other student attributes (e.g. birthdate, address, email, id card)?
  - Very repetitive!
  - Hard to manage.
  - Sorting method keeps increasing.

Reflection about this approach (4/4)

- Is it close to the human-beings conceptual level?
  - No!

- A more natural approach, closer to human-beings conceptual level, would be better? And, would that be expectable?
  - Yes! Yes!

So, what if we…

- Could define novel data types to represent the things (e.g. students, course units) that one observe in the (business) domain we are dealing with?

- Could manage those data types (e.g. Student) similarly to the "*built-in*" data types (e.g. int, char, boolean)?

- Could express and operate closer to the human-beings conceptual level?

Object Oriented (OO)

Object-Oriented (OO)

- More than a programming style, it is a paradigm

- Probably the most adopted paradigm for software development
  - Up-to-date development processes rely and promote OO
  - Architectural styles and patterns apply and share OO fundamental principals

- Transversal to several software engineering activities
  - Analysis → Analysis OO
  - Design → Design OO
  - Coding → Coding OO
  - Testing → Testing OO

Object-Oriented (OO)

- More than a programming style, it is a paradigm

- Probably the most adopted paradigm for software development
  - Up-to-date development processes rely and promote OO
  - Architectural styles and patterns apply and share OO fundamental principals

- Transversal to several software engineering activities
  - Analysis → Analysis OO
  - Design → Design OO
  - Coding → Coding OO
  - Testing → Testing OO

  Consume and produce OO artifacts

33

OO – Class Definition

- Groups things that show some common characteristics as attributes and behaviors

- It might be thought as a category to which these existing things in our (business) domain belong to

- It provides the blueprint/template to individually represent each of those things
  - e.g., a "Student" class can be used as a template to represent each student in the classroom

OO – Object (or Instance) Definition

- It represents/refers to a concrete thing (physical or conceptual or (in)tangible) of our (business) domain in accordance with one class
    - e.g., each student in the room can be represented as an object of class "Student"

- Each object has a state determining the value of its attributes (e.g. the student's name) and exhibit behaviors (or operations) that when executed might change (or not) its internal state
    - e.g., a method called isApproved() stating its approval status

Using Classes as Data Structures: A rudimentary approach

- All attributes are public

- Constructor with no arguments (default)

- No operations

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student(String name) {
        this.name = name;
        //used to initialize attributes that identify the student
    }
}
```

Class Structure – Through example

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

Class Structure – Through example

The name of the class. Further, can be used as a data type.

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

Class Structure – Through example

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

The name of the class. Further, can be used as a data type.

List of the attributes that all objects of this class has. Each attribute has an accessor (*public* or *protected* or *private*), a **data type** and a *designation*.

Class Structure – Through example

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

The name of the class. Further, can be used as a data type.

List of the attributes that all objects of this class has. Each attribute has an accessor (*public* or *protected* or *private*), a **data type** and a *designation*.

The constructor. Invoked to create every object of the class. At least one *public* constructor is usually provided, but several might exist and with other accessors.

Class Structure – Through example

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

The name of the class. Further, can be used as a data type.

List of the attributes that all objects of this class has. Each attribute has an accessor (*public* or *protected* or *private*), a **data type** and a *designation*.

The constructor. Invoked to create every object of the class. At least one *public* constructor is usually provided, but several might exist and with other accessors.

Several methods might be defined to exhibit and capture object behaviours. Accessors constraint method accessability.

Class Structure – Through example

```java
public class Student {

    // Attributes
    public int number;
    public String name;
    public int grade;

    // Constructor
    public Student() {

    }

    // Operations (Behavior)
    public boolean isApproved() {
        return (this.grade >= 10);
    }
}
```

The name of the class. Further, can be used as a data type.

List of the attributes that all objects of this class has. Each attribute has an accessor (*public* or *protected* or *private*), a **data type** and a *designation*.

The constructor. Invoked to create every object of the class. At least one *public* constructor is usually provided, but several might exist and with other accessors.

Several methods might be defined to exhibit and capture object behaviours. Accessors constraint method accessability.

Reserved word used to refer to this object's attributes and operations.

Evolving Design – Applying Classes

- Classes
  - Student (cf. defined previously)

- US01 Method
  boolean sortStudentsByAscendingNumber(Student[] students)

- US02 Method
  boolean sortStudentsByDescendingGrade(Student[] students)

Evolving Design – Applying Classes

- Classes
  - Student (cf. defined previously)

- US01 Method
  boolean sortStudentsByAscendingNumber(Student[] students)

Student being used as a novel data type.
Each methods requires, as input, an array of students.

- US02 Method
  boolean sortStudentsByDescendingGrade(Student[] students)

Instructions on how to proceed

- Check the test cases on the following slides

Testing example

```java
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;    studentOne.name = "Ana Maria Sousa";    studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;    studentTwo.name = "André Pinto da Silva";    studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

Testing example

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;    studentOne.name = "Ana Maria Sousa";     studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;    studentTwo.name = "André Pinto da Silva";     studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;    studentOne.name = "Ana Maria Sousa";    studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;    studentTwo.name = "André Pinto da Silva";    studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

48

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange
```

One student object →

*Sets the **value** of attribute **number** on object **studentOne***

```
    Student studentOne = new Student();
    studentOne.number = 1200001;   studentOne.name = "Ana Maria Sousa";     studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;   studentTwo.name = "André Pinto da Silva";     studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

49

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;   studentOne.name = "Ana Maria Sousa";     studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;   studentTwo.name = "André Pinto da Silva";     studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

*Sets the **value** of attribute **number** on object **studentOne***

Another student object

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;    studentOne.name = "Ana Maria Sousa";    studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;    studentTwo.name = "André Pinto da Silva";    studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

*Sets the **value** of attribute **number** on object **studentOne***

Another student object

*Sets the **value** of attribute **grade** on object **studentTwo***

51

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;   studentOne.name = "Ana Maria Sousa";   studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;   studentTwo.name = "André Pinto da Silva";   studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

*Sets the **value** of attribute **number** on object **studentOne***

Another student object

*Sets the **value** of attribute **grade** on object **studentTwo***

*Sets the **value** of attribute **name** on object **studentTwo***

52

Testing example

Creates a new object of the Student class by invoking the constructor

```
public void ensureSortingTwoUnsortedElementArraysByNumberWorks() {
    //Arrange

    Student studentOne = new Student();
    studentOne.number = 1200001;   studentOne.name = "Ana Maria Sousa";   studentOne.grade = 16;

    Student studentTwo = new Student();
    studentTwo.number = 1200032;   studentTwo.name = "André Pinto da Silva";   studentTwo.grade = 12;

    Student[] students = {studentTwo, studentOne};
    Student[] expectedStudents = {studentOne, studentTwo};

    // Act
    boolean result = ProblemTwo.sortStudentsByAscendingNumber(students);

    //Assert
    assertTrue(result);
    //check dimension
    assertEquals(expectedStudents.length, students.length);
    // check array content
    assertArrayEquals(expectedStudents, students);
}
```

One student object

Create Arrays

*Sets the **value** of attribute **number** on object **studentOne***

Another student object

*Sets the **value** of attribute **grade** on object **studentTwo***

*Sets the **value** of attribute **name** on object **studentTwo***

53

Instructions on how to proceed

- Run the tests of the class **ProblemTwoTest** in the **problem.two.version.three** test package
  - 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
  - Implement the class **ProblemTwo** in the package **problem.two.version.three** with enough code for the tests to pass
  - Once you have implemented, proceed to the next test

- Don't forget, always run all your class tests, to ensure you don't break any previous tests

Implementation

```java
public static boolean sortStudentsByAscendingNumber(Student[] students) {
    if (students == null) {
        return false;
    }

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

Implementation

```java
public static boolean sortStudentsByAscendingNumber(Student[] students) {
    if (students == null) {
        return false;
    }

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

Now the comparison is between the number attribute

Implementation

```java
public static boolean sortStudentsByAscendingNumber(Student[] students) {
    if (students == null) {
        return false;
    }

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

Now the comparison is between the number attribute

Swapping students is similar to swapping integers and strings.

Comparing Approaches (1/5)

- How both approaches compare to each other?

  - Version Two : Using individual arrays

  - Version Three: Using a Student class and only one array

Comparing Approaches (2/5)

- Which one is more error prone? Why?

Comparing Approaches (2/5)

- Which one is more error prone? Why?
  - **Version Two: Using individual arrays**
    - students' data being split by 3 arrays
    - Arrays are "linked" by their position (indexes) on such arrays
    - Index is not a reliable option for this coupling

Comparing Approaches (3/5)

- Which one is easier to maintain and evolve? Why?

Comparing Approaches (3/5)

- Which one is easier to maintain and evolve? Why?
  - **Version Three: Using a Student class and only one array**
    - (i) a single array has all the data about students
    - (ii) each student owns its own data

- Check next question/answer

Comparing Approaches (4/5)

- How well each solution would handle if one need to store several other student attributes (e.g. birthdate, address, email, id card)?

Comparing Approaches (4/5)

- How well each solution would handle if one need to store several other student attributes (e.g. birthdate, address, email, id card)?
  - **Version Two: Using individual arrays**
    - it implies adding and managing a new array by each attribute and revising sorting methods.

  - **Version Three: Using a Student class and only one array**
    - it implies adding such attributes on Student class but sorting methods do not need to be revised.

Comparing Approaches (5/5)

- Which one is closer to the human-beings conceptual level?

Comparing Approaches (5/5)

- Which one is closer to the human-beings conceptual level?
  - Definitely the latter one.
  - But, hopefully there is still room for improvements.

Revising previous "So, what if we…"

- Could define novel data types to represent the things (e.g. students, course units) that one observe in the (business) domain we are dealing with?

- Could manage those data types (e.g. Student) similarly to the "*built-in*" data types (e.g. int, char, boolean)?

- Could express and operate closer to the human-beings conceptual level?

Revising previous "So, what if we…"

- Could define novel data types to represent the things (e.g. students, course units) that one observe in the (business) domain we are dealing with?

- Could manage those data types (e.g. Student) similarly to the "*built-in*" data types (e.g. int, char, boolean)?

- Could express and operate closer to the human-beings conceptual level?

Revising previous "So, what if we…"

- Could define novel data types to represent the things (e.g. students, course units) that one observe in the (business) domain we are dealing with?

- Could manage those data types (e.g. Student) similarly to the "*built-in*" data types (e.g. int, char, boolean)?

- Could express and operate closer to the human-beings conceptual level?

Revising previous "So, what if we…"

- Could define novel data types to represent the things (e.g. students, course units) that one observe in the (business) domain we are dealing with?

- Could manage those data types (e.g. Student) similarly to the "*built-in*" data types (e.g. int, char, boolean)?

- Could express and operate closer to the human-beings conceptual level?

Summary

- Two OO fundamental notions were introduced
  - Classes
  - Objects (or instances)

- OO was applied (by now…)
  - As data structures – a very simple and rudimentary approach
  - On software engineering activities: Design and Coding
  - To develop an (alternative) solution to Problem II

- Comparison between developed solutions was done
  - Advantages of adopting OO are becoming clear and evident

What's next on OO?

- Some OO concepts (or principles)
  - Encapsulation
  - Abstraction

- Some OO Design principles and patterns
  - Tell, Don't Ask
  - Information Expert

- Association between objects

- Objects' equality