

From Procedural Programming to Object-Oriented Programming (OOP)

A preliminary step by step approach

ISEP/LEI/ESOF

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

Towards OOP

Primary Concepts and Principles

Encapsulation

Encapsulation (1/2)

- It is a fundamental concept of OOP
- It is based on the ideas of
 - Restricting the outside (other objects) from directly accessing to object's components, namely those where object's data is stored (e.g. attributes)
 - Hiding object's internal representations from the outside
 - Keeping the object's state private
 - Controlling access (e.g. read/write) to object's state
 - Bundling object's data and methods operating on such data
 - Other objects can only interact with the object by invoking its public methods

Encapsulation (2/2)

Advantages

- Over time, objects' internal representations may change without (negative) impacting on other objects as long as its public API (set of public methods) is stable.

Drawback

- More methods are required
- Cloning (performance hit)
 - Clone method is used to create copies of provided objects
 - Clone method is used to create copies of returned objects
 - Does not apply immutable objects

An Object's Internal State

- What is it?
 - Expression used for referring to the data that an object has in the current moment
 - Objects' data means/refers to the values of all object's attributes
 - e.g., values of attributes number, name and grade of a Student object
- Object's state evolves over time, as long as the object's attributes/values change
 - e.g., the value of a Student's grade might change from 8 to 13

Object's State – Some Principles


- State must be kept private
 - Other objects must not be able to directly change the state of another object
 - Prevents unauthorized direct access to object's data
- State must be always consistent
 - An object must be always in a valid state
 - Allowed changes lead the object from one valid state to another valid state
 - Changes leading to an invalid state must be rejected or not allowed at all
 - Valid states are determined by the domain business rules
 - e.g.: Are there students without a number and/or a name?
 - e.g.: Does a student always have a grade?

How to keep state private and consistent?

- Define attributes using the ***private accessor*** instead of ***public accessor***
 - In some cases, ***protected accessor*** is also admissible
- **Constructors** – only constructors leading to an initial valid state should exist
- Attribute values are changed only through public methods
 - **Setter** methods – used to change the value of a single attribute
 - Not all attributes should have a setter method
 - E.g. “*setGrade(int grade)*” might be used to change the value of attribute *grade* of a *Student* object
 - **Business** methods – used to change at once the value of one or more attributes
 - Method names are inspired and reflect business processes/operations
 - E.g., “*changeName(String name)*” might be used to modify the value of attribute *name* of a *Student* object instead of “*setName(String name)*”
 - E.g., “*doImprovement(int result)*” might be used to capture a business process (evaluation improvement) and its rules and, therefore, at the end it might change or not the value of attribute *grade*

Business processes and rules – a small note

- Business processes and rules are known by the client
- The development team should interact continuously with the client to identify, understand, synthesize and systematize such processes and rules, namely during activity of
 - (Elicitation of) Requirements
 - Analysis
- Both, should be formally captured in (e.g.)
 - Use Cases and/or User Stories
 - Acceptance criteria
 - Activity Diagrams



More about this,
later on!

Problem III

Problem III – Some business rules

- R1 – A student always has a number and name
- R2 – Student number is always a 7 digits number (e.g. 1190001)
- R3 – Length of a student name cannot be shorter than 5 chars
- R4 – Student grade ranges between 0 and 20 (no decimals)
- R5 – Student grade results from an evaluation process (which is out of scope in this application, but only occurs once)

Checking impact on our *Student* class


- From objects' state principles
 - Attributes *number*, *name* and *grade* need to become private
 - Since the default constructor (no arguments) leads to an invalid state (R1), its use should be avoided → Not exist!
- From business rules
 - R1 → requires a constructor to specify *number* and *name*
 - R2 → involves validating the student number (new method)
 - R3 → involves validating the student name (new method)
 - R4 → involves validating the student grade (new method)
 - R5 →
 - implies a method to know the grade resulting from the evaluation process
 - a way to distinguish if a student has been (or not) evaluated need to be envisaged

Design – Updating *Student* class (not implementing constructor and methods)

```
public class Student {  
    // Attributes  
    private int number;  
    private String name;  
    private int grade;  
  
    // Constructor  
    public Student(int number, String name) {...}  
  
    // Operations  
    private boolean isValidNumber(int number) {...}  
  
    private boolean isValidName(String name) {...}  
  
    private boolean isValidGrade(int grade) {...}  
  
    private boolean isEvaluated() {...}  
  
    public void doEvaluation(int grade) {...}  
}
```

Design – Updating *Student* class (not implementing constructor and methods)

```
public class Student {  
    // Attributes  
    private int number;  
    private String name;  
    private int grade;  
  
    // Constructor  
    public Student(int number, String name) {...}  
  
    // Operations  
    private boolean isValidNumber(int number) {...}  
  
    private boolean isValidName(String name) {...}  
  
    private boolean isValidGrade(int grade) {...}  
  
    private boolean isEvaluated() {...}  
  
    public void doEvaluation(int grade) {...}  
}
```



Design – Updating *Student* class (not implementing constructor and methods)

```
public class Student {
```

```
// Attributes
```

```
private int number;  
private String name;  
private int grade;
```



```
// Constructor
```

```
public Student(int number, String name) {...}
```



```
// Operations
```

```
private boolean isValidNumber(int number) {...}
```

```
private boolean isValidName(String name) {...}
```

```
private boolean isValidGrade(int grade) {...}
```

```
private boolean isEvaluated() {...}
```

```
public void doEvaluation(int grade) {...}
```

```
}
```


Design – Updating *Student* class (not implementing constructor and methods)

```
public class Student {
```

```
// Attributes
```

```
private int number;  
private String name;  
private int grade;
```



```
// Constructor
```

```
public Student(int number, String name) {...}
```



```
// Operations
```

```
private boolean isValidNumber(int number) {...}  
  
private boolean isValidName(String name) {...}  
  
private boolean isValidGrade(int grade) {...}  
  
private boolean isEvaluated() {...}
```



Till now it was not found any reason/need justifying these methods to be accessed by other objects. When that happens, methods should be private as well.

```
public void doEvaluation(int grade) {...}
```

```
}
```

Design – Updating *Student* class (not implementing constructor and methods)

```
public class Student {
```

```
// Attributes
```

```
private int number;  
private String name;  
private int grade;
```



```
// Constructor
```

```
public Student(int number, String name) {...}
```



```
// Operations
```

```
private boolean isValidNumber(int number) {...}
```

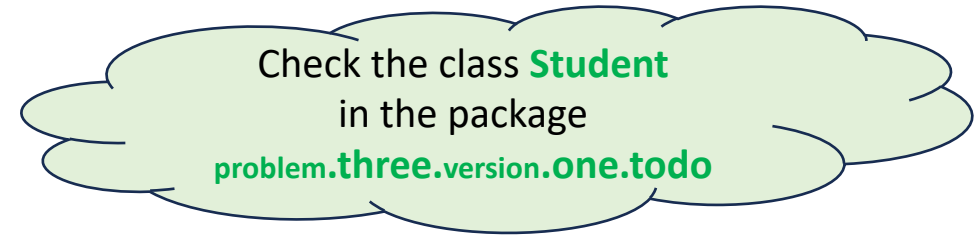
```
private boolean isValidName(String name) {...}
```

```
private boolean isValidGrade(int grade) {...}
```

```
private boolean isEvaluated() {...}
```



```
public void doEvaluation(int grade) {...}
```



Till now it was not found any reason/need justifying these methods to be accessed by other objects. When that happens, methods should be private as well.

Used to set the students' grade.
Other plausible name would be "setGrade".

Test Cases – Considered Scenarios

- Constructor
 - Invalid number → several reasons (e.g. negative, less/more than 7 digits)
 - Invalid name → several reasons (e.g. null, less than 5 chars, left/right spaces)
 - Invalid number and name → mix of above reasons
 - Number and name are valid
- doEvaluation
 - Grade out of limits → lower and upper limit
 - Doing evaluation more than once

→ Several testing methods should be specified

Example – Some testing methods

@Test

```
public void ensureCreateValidStudentWorks(){  
    // Arrange + Act  
    Student student = new Student( 1190001, "Paulo");  
    // Assert  
    assertNotNull(student);  
}
```

@Test

```
public void ensureCreateStudentWith6DigitsNegativeNumberFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(-190001, "Paulo Maio");  
    });  
}
```

@Test

```
public void ensureCreateStudentWith7DigitsNegativeNumberFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(-1190001, "Paulo Maio");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithLongerDigitsNumberFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(11900013, "Paulo Maio");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithEmptyNameFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, "");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithNameFullOfSpacesFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, " ");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithShortNameLengthFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, "Rui");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithValidNumberButInvalidNameFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1980398, "Bia");  
    });  
}
```

Example – Some testing methods

@Test

```
public void ensureCreateValidStudentWorks(){  
    // Arrange + Act  
    Student student = new Student( 1190001, "Paulo");  
    // Assert  
    assertNotNull(student);  
}
```

@Test

```
public void ensureCreateStudentWith6DigitsNegativeNumberFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(-190001, "Paulo Maio");  
    });  
}
```

@Test

```
public void ensureCreateStudentWith7DigitsNegativeNumberFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(-1190001, "Paulo Maio");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithLongerDigitsNumberFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(11900013, "Paulo Maio");  
    });  
}
```

States that an error (exception) of such type is expected to be raised while running the test

@Test

```
public void ensureCreateStudentWithEmptyNameFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, "");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithNameFullOfSpacesFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, " ");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithShortNameLengthFails() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1190001, "Rui");  
    });  
}
```

@Test

```
public void ensureCreateStudentWithValidNumberButInvalidNameFails(){  
    assertThrows(IllegalArgumentException.class, () -> {  
        Student student = new Student(1980398, "Bia");  
    });  
}
```

Check the class **StudentTest**
in the test package
problem.three.version.one.todo

Implementation (Partial)

// Constructor

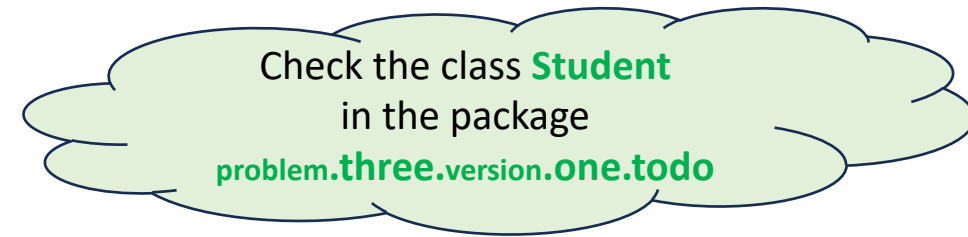
```
public Student(int number, String name) {  
    if (!this.isValidNumber(number))  
        throw new IllegalArgumentException("Student number needs to be 7 digits number (e.g. 1190001).");  
    if (!this.isValidName(name))  
        throw new IllegalArgumentException("Student name cannot be shorter than 5 chars.");  
    this.number = number;  
    this.name = name.trim();  
    this.grade = -1; // -1 means student hasn't been evaluated  
}
```

// Operations

```
private boolean isValidNumber(int number) {  
    if (number != Math.abs(number))  
        return false;  
    String strNumber = Integer.toString(number);  
    return (strNumber.length() == 7);  
}
```

```
private boolean isValidName(String name) {  
    if (name == null)  
        return false;  
    name = name.trim();  
    return !(name.length() < 5);  
}  
  
public void doEvaluation(int grade) {  
    if (!this.isEvaluated() && this.isValidGrade(grade))  
        this.grade = grade;  
}
```

Implementation (Partial)



// Constructor

```
public Student(int number, String name) {  
    if (!this.isValidNumber(number))  
        throw new IllegalArgumentException("Student number needs to be 7 digits number (e.g. 1190001).");  
    if (!this.isValidName(name))  
        throw new IllegalArgumentException("Student name cannot be shorter than 5 chars.");  
    this.number = number;  
    this.name = name.trim();  
    this.grade = -1; // -1 means student hasn't been evaluated  
}
```

Throws an error/exception, avoiding the object being created

// Operations

```
private boolean isValidNumber(int number) {  
    if (number != Math.abs(number))  
        return false;  
    String strNumber = Integer.toString(number);  
    return (strNumber.length() == 7);  
}
```

```
private boolean isValidName(String name) {  
    if (name == null)  
        return false;  
    name = name.trim();  
    return !(name.length() < 5);  
}  
  
public void doEvaluation(int grade) {  
    if (!this.isEvaluated() && this.isValidGrade(grade))  
        this.grade = grade;  
}
```

Now, how good is our solution?

- Business rules applying on student attributes are being applied
 - Several methods on class Student were added to ensure that
 - Student constructor with no arguments was removed
 - Student class got a new constructor with 2 arguments (number and name)
- Student objects are able to keep their state private and consistent
 - Student attributes accessor become private
 - Student attributes values are just set through constructor and public methods, all of them, enforcing known business rules
- But, sorting methods (e.g. *sortStudentsByAscendingNumber*) are not working since attribute values are no longer accessible

How to fix this? Two alternatives:

- **Getters methods** – used to read/get the value of a single attribute
 - Not all attributes should have a getter method
 - E.g. “*int getNumber()*” might be used to get the value of attribute *number* of a *Student* object
 - **Warning:** be careful to avoid providing, inadvertently, access to write on attributes → preferably return a copy of the attribute value
- **“Tell” methods** – used to tell an object to perform a given operation over its own data
 - Result from applying “*Tell, Don’t Ask*” and “*Information Expert*” principles
 - E.g. “*int compareToByNumber(Student other)*” might be used to compare the student number of an object (*this*) to another object (*other*)
 - **Preferable over getters methods**

Fixing... tests first

@Test

```
public void ensureSortingUnsortedArraysByNumberWorks() {
```

```
    // Arrange
```

```
    Student studentOne = new Student(1200001, "Ana Maria Sousa");
```

```
    studentOne.doEvaluation(16);
```

```
    Student studentTwo = new Student(1200032, "André Pinto da Silva");
```

```
    studentTwo.doEvaluation(12);
```

```
    Student studentThree = new Student(1190432, "Martim Gomes Costa");
```

```
    studentThree.doEvaluation(17);
```

```
    Student studentFour = new Student(1181208, "Mariana Gonçalves Mendes");
```

```
    studentFour.doEvaluation(14);
```

```
    Student[] students = {studentOne, studentTwo, studentThree, studentFour}; // Unordered
```

```
    Student[] expectedStudents = {studentFour, studentThree, studentOne, studentTwo}; // Ordered
```

```
    // Act
```

```
    boolean result = ProblemThree.sortStudentsByAscendingNumber(students);
```

```
    //Assert
```

```
    assertTrue(result); // check if sort runs with no errors
```

```
    assertEquals(expectedStudents.length, students.length); // check dimension
```

```
    assertEquals(expectedStudents, students); // check students is sorted
```

```
}
```

Fixing... tests first

@Test

```
public void ensureSortingUnsortedArraysByNumberWorks() {
```

Just an example. It is necessary to fix several tests.

```
// Arrange
```

```
Student studentOne = new Student(1200001, "Ana Maria Sousa");
```

```
studentOne.doEvaluation(16);
```

```
Student studentTwo = new Student(1200032, "André Pinto da Silva");
```

```
studentTwo.doEvaluation(12);
```

```
Student studentThree = new Student(1190432, "Martim Gomes Costa");
```

```
studentThree.doEvaluation(17);
```

```
Student studentFour = new Student(1181208, "Mariana Gonçalves Mendes");
```

```
studentFour.doEvaluation(14);
```

```
Student[] students = {studentOne, studentTwo, studentThree, studentFour}; // Unordered
```

```
Student[] expectedStudents = {studentFour, studentThree, studentOne, studentTwo}; // Ordered
```

```
// Act
```

```
boolean result = ProblemThree.sortStudentsByAscendingNumber(students);
```

```
//Assert
```

```
assertTrue(result); // check if sort runs with no errors
```

```
assertEquals(expectedStudents.length, students.length); // check dimension
```

```
assertArrayEquals(expectedStudents, students); // check students is sorted
```

```
}
```

Fixing... tests first

@Test

```
public void ensureSortingUnsortedArraysByNumberWorks() {
```

Just an example. It is necessary to fix several tests.

// Arrange

```
Student studentOne = new Student(1200001,"Ana Maria Sousa");
studentOne.doEvaluation(16);
Student studentTwo = new Student(1200032,"André Pinto da Silva");
studentTwo.doEvaluation(12);
Student studentThree = new Student(1190432,"Martim Gomes Costa");
studentThree.doEvaluation(17);
Student studentFour = new Student(1181208,"Mariana Gonçalves Mendes");
studentFour.doEvaluation(14);
```

Just the “arrange” part of tests need to be fixed. Now, one makes use of the constructor.

```
Student[] students = {studentOne, studentTwo, studentThree, studentFour}; // Unordered
Student[] expectedStudents = {studentFour, studentThree, studentOne, studentTwo}; // Ordered
```

// Act

```
boolean result = ProblemThree.sortStudentsByAscendingNumber(students);
```

//Assert

```
assertTrue(result); // check if sort runs with no errors
assertEquals(expectedStudents.length, students.length); // check dimension
assertArrayEquals(expectedStudents, students); // check students is sorted
```

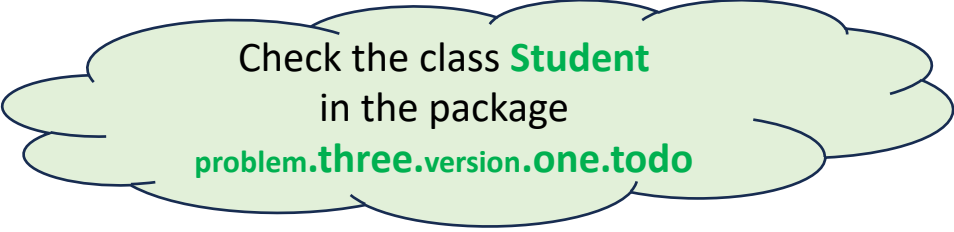
```
}
```

Check the class **ProblemThreeTest**
in the test package
problem.three.version.one.todo

Fixing... implementation – Getter approach

- In *Student* class add

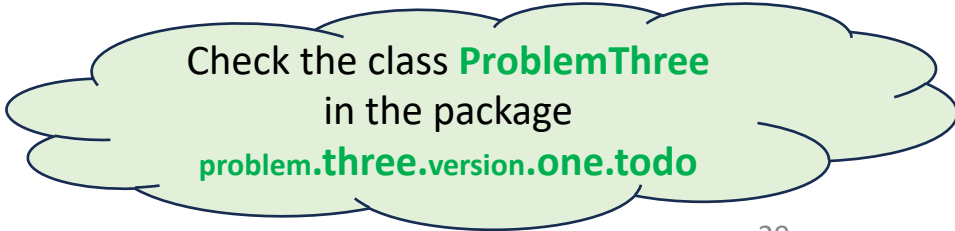
```
public int getNumber(){  
    return this.number; // type is "int" so a copy of the value is return  
}
```



Check the class **Student**
in the package
problem.three.version.one.todo

- Update sort method to

```
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].getNumber() > students[j + 1].getNumber()) {  
                //swap elements if not in order - in students  
                swapStudentArrayElements(students, j, j + 1);  
            }  
        }  
    }  
    return true;  
}
```

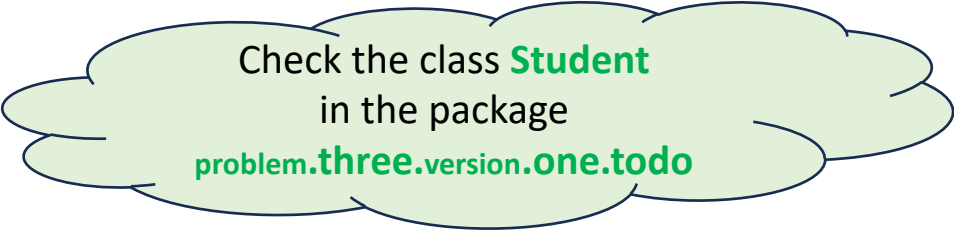


Check the class **ProblemThree**
in the package
problem.three.version.one.todo

Fixing... implementation – “Tell” approach

- In *Student* class add

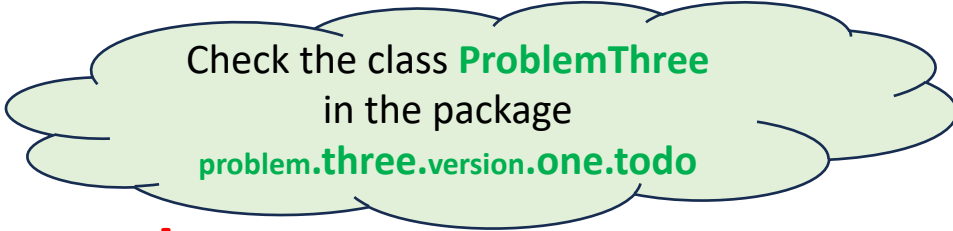
```
public int compareToByNumber(Student other) {  
    if (this.number < other.number) {  
        return -1; // less than other  
    }  
    if (this.number > other.number) {  
        return 1; // greater than other  
    }  
    return 0; // both have the same value  
}
```



Check the class **Student**
in the package
problem.three.version.one.todo

- Update sort method to

```
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].compareToByNumber(students[j+1])>0) {  
                //swap elements if not in order - in students  
                swapStudentArrayElements(students, j, j + 1);  
            }  
        }  
    }  
    return true;  
}
```



Check the class **ProblemThree**
in the package
problem.three.version.one.todo

Preferable over getter approach

Tell Don't Ask

Tell, Don't Ask

- Principle
 - You **should not ask** an object for its own data (*state*) and further act on that data to make some decisions
 - Instead, **you should tell** an object what to do, i.e. send commands to it
- Advantages
 - Encourages to move behavior into an object to go with the data
 - Solution becomes:
 - Clear
 - Easy to maintain
 - Flexible enough to add new features

Information Expert

Information Expert

- **Problem:** what is the general principle for assigning responsibilities to objects?
- **Solution:** assign responsibility to the information expert
 - I.e., the class that contains the information needed to fulfil that responsibility
 - Which class?
 - Get inspired by the domain concepts
 - Promote domain concepts to software classes
 - E.g., in scope of Problem III, student is a domain concept (i.e. client knows and speaks about it and cannot be represented as a primitive data type) that gives rise to the class Student

But, what are responsibilities?

- An abstraction of what each class/object do
 - Are obligations and behaviors of an object depending on the role it performs
 - A responsibility is not the same as a method, but methods implement responsibilities
-
- Metaphor: to help in OO development (Design and Code activities)
 - Analogy: thinking about software objects in a similar way to thinking about people with responsibilities, who collaborate with others to get their work done

Which kind of responsibilities are there?

- Responsibilities of “knowing”
 - Know its own (private) information
 - Know related objects
 - Know how to obtain or calculate new information
- Responsibilities of “doing”
 - Do it yourself, create an object, do calculations, etc.
 - Delegate, initiate actions on other objects
 - Control and coordinate activities on other objects

Back to Problem III – Other Responsibilities

Regarding students management, which class should have the responsibility of

- Adding a new student?
- Removing a student?
- Updating a student info?
- Sorting students?
- Filtering students?



Checking our current design (Mix of Procedural + OO)

- Adding a new student?
 - Adding an element in students array (e.g. static ***Student[] students***)
- Removing a student?
 - Removing an element in students array
- Updating a student info?
 - Updating data of a student object → done in Student class
- Sorting students?
 - Sorting elements in students array
 - Two static methods: ***sortStudentsByAscendingNumber, sortStudentsByDescendingGrade***
- Filtering students?
 - Getting a subset of students array, but no criteria neither US defined yet!

Updating our Design (1/2)

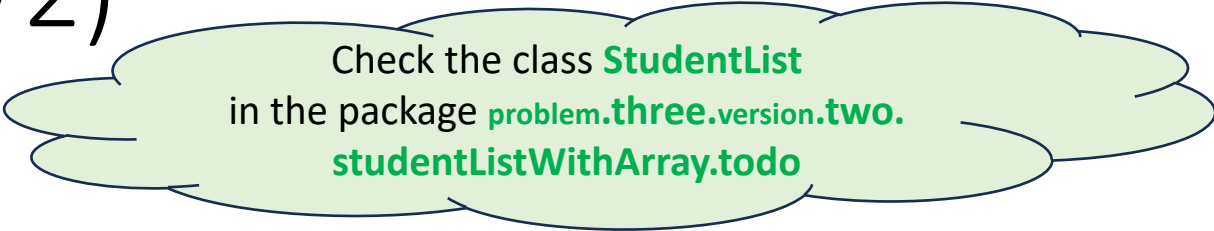
- No static methods and primitive data structures → fully adopt OO
- Creating a new class “StudentList”
 - To manage students array content by
 - Preserving the known students
 - Adding students → might be required, but no User Story for it yet
 - Removing students → might be required, but no User Story for it yet
 - To sort known students
 - By number ascendent (US01)
 - By grade descendent (US02)
 - To filter known students
 - No criteria or US defined yet!

Updating our Design (2/2)

```
public class StudentList {  
    // Attributes  
    private Student[] students;  
  
    //Constructors  
    public StudentList() {}  
  
    public StudentList(Student[] students) {}  
  
    // Operations  
    public void sortByAscendingNumber() {}  
  
    public void sortByDescendingGrade() {}  
  
    public Student[] toArray() {}  
}
```


Updating our Design (2/2)

```
public class StudentList {  
    // Attributes  
    private Student[] students;  
  
    //Constructors  
    public StudentList() {}  
  
    public StudentList(Student[] students) {}  
  
    // Operations  
    public void sortByAscendingNumber() {}  
  
    public void sortByDescendingGrade() {}  
  
    public Student[] toArray() {}  
}
```



Check the class **StudentList**
in the package **problem.three.version.two.**
studentListWithArray.todo

- *Internal list of students – stored as an array*
- *Creates an empty StudentList*
- *Creates a new StudentList and populates it with students*
- *Sorts the list using the students numbers (US01)*
- *Sorts the list using the students grades (US02)*
- *Returns a copy of students stored in the list as an array*

As seen before, other methods might be need to support further User Stories.

Test Cases – Considered Scenarios

- Constructor
 - No Arguments → OK! Students list must be empty
 - With Arguments → Error if input is *null*, otherwise it should have the same elements of input array but no further changes on input array are reflected on the students list
- *sortByAscendingNumber* and *sortByDescendingGrade*
 - All scenarios considered previously (i.e. before adopting OO)
- *toArray*
 - Used to verify other methods only (returning students list in the form of an array)

→ **Several testing methods should be specified**

Example – Some testing methods

@Test

```
public void ensureCreateStudentListWorks() {  
    // Act  
    StudentList stList = new StudentList(); // StudentList empty  
    Student[] result = stList.toArray();  
    //Assert  
    assertEquals(0, result.length); // check array  
}
```

@Test

```
public void ensureCreateStudentListWithSomeElementsWorks() {  
    // Arrange  
    Student studentOne = new Student(1200001, "Ana Maria Sousa");  
    Student studentTwo = new Student(1200032, "André Pinto da Silva");  
    Student studentThree = new Student(1190432, "Martim Gomes Costa");  
    Student[] students = {studentOne, studentTwo, studentThree}; // Some order  
    Student[] expected = {studentOne, studentTwo, studentThree}; // A copy of students  
    // Act  
    StudentList stList = new StudentList(students);  
    students[2] = studentOne; // change the original array  
    Student[] result = stList.toArray();  
    //Assert  
    assertEquals(expected, result); // check students are the same  
    assertNotSame(students, result);  
}
```

@Test

```
public void ensureSortingUnsortedArraysByGradeWorks() {  
    // Arrange  
    Student studentOne = new Student(1200001, "Ana Maria Sousa");  
    studentOne.doEvaluation(16);  
    Student studentTwo = new Student(1200032, "André Pinto da Silva");  
    studentTwo.doEvaluation(12);  
    Student studentThree = new Student(1190432, "Martim Gomes Costa");  
    studentThree.doEvaluation(17);  
    Student[] students = {studentOne, studentTwo, studentThree}; // Unordered  
    StudentList stList = new StudentList(students);  
    Student[] expected = {studentThree, studentOne, studentTwo}; // Ordered  
    // Act  
    stList.sortByDescendingGrade();  
    Student[] result = stList.toArray();  
    //Assert  
    assertEquals(expected, result); // check students are sorted  
}
```

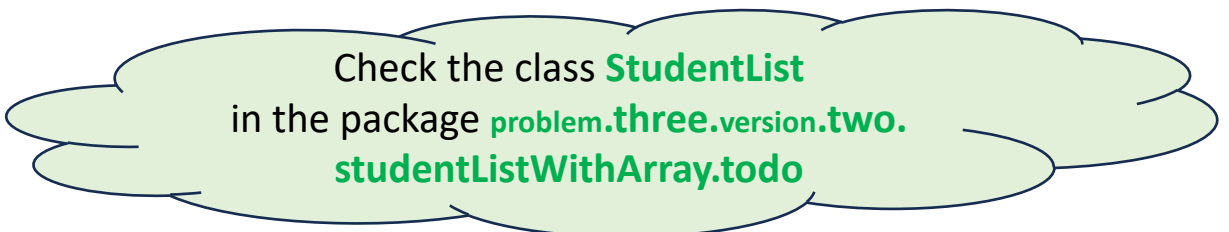
Check the class **StudentListTest**
in the test package **problem.three.**
version.two.studentListWithArray.todo

StudentList – *sortByAscendingNumber*

- Reusing previous/existing implementation
- Adapting it:
 - Acting on students array of the class (private attribute)
 - No need to return a boolean since no errors are expected to occur

```
public void sortByAscendingNumber() {  
    //Sort the students in ascending order using two for loops  
    for (int i = 0; i < this.students.length; i++) {  
        for (int j = 0; j < this.students.length-i-1; j++) {  
            if(this.students[j].compareToByNumber(this.students[j+1])>0) {  
                //swap elements if not in order – in students  
                swapStudentArrayElements(students, j, j + 1);  
            }  
        }  
    }  
}
```

```
private void swapStudentArrayElements(Student[] array, int indexOne, int indexTwo) {  
    Student temp;  
    temp = array[indexOne];  
    array[indexOne] = array[indexTwo];  
    array[indexTwo] = temp;  
}
```



Check the class **StudentList**
in the package **problem.three.version.two.**
studentListWithArray.todo

Similar approach to implement *sortByDescendingGrade*

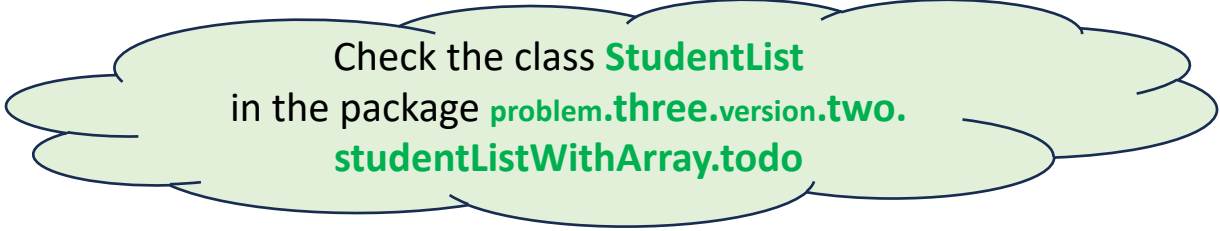
StudentList – Other methods

// Constructors

```
public StudentList() {  
    this.students = new Student[0];  
}  
public StudentList(Student[] students) {  
    if (students == null)  
        throw new IllegalArgumentException("Students array should not be null");  
    this.students = copyStudentsFromArray(students, students.length);  
}
```

// Operations

```
public Student[] toArray() {  
    return this.copyStudentsFromArray(this.students, this.students.length);  
}  
  
private Student[] copyStudentsFromArray(Student[] students, int size) {  
    Student[] copyArray = new Student[size];  
    for(int count=0; (count < size) && (count < students.length); count++){  
        copyArray[count] = students[count];  
    }  
    return copyArray;  
}
```



Check the class **StudentList**
in the package **problem.three.version.two.**
studentListWithArray.todo

Associations

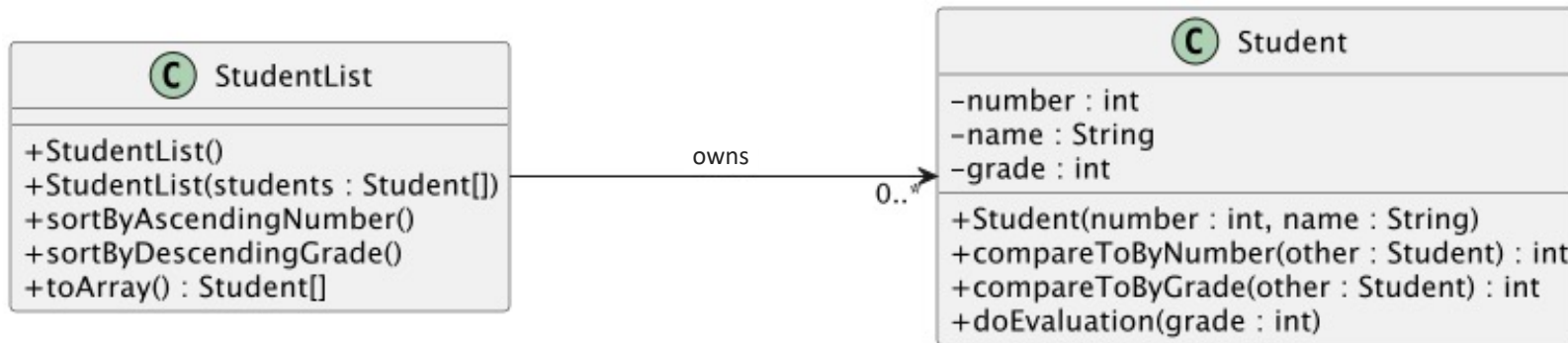
Association between Objects

- An association relationship occurs between two (or more) objects
 - One object (source) is **directly connected** to one or more (target) objects
 - Source object can invoke public methods of target objects
- Two main characteristics:
 - Can have a name (e.g. *owns*) to allow the same kind of target objects to play distinct roles in the source object → the association name is the role name
 - Multiplicity: expressed by a lower and an upper limit (e.g. “0..5”). E.g.:
 - “0..*” or simply “*” – meaning multiplicity ranges between zero or more (infinite)
 - “1..*” – meaning multiplicity ranges between one or more
 - “2..7” – meaning multiplicity might be any integer value between two and seven
 - “3, 5, 7” – meaning it might be exactly three or five or seven
 - “4” – meaning it is exactly four

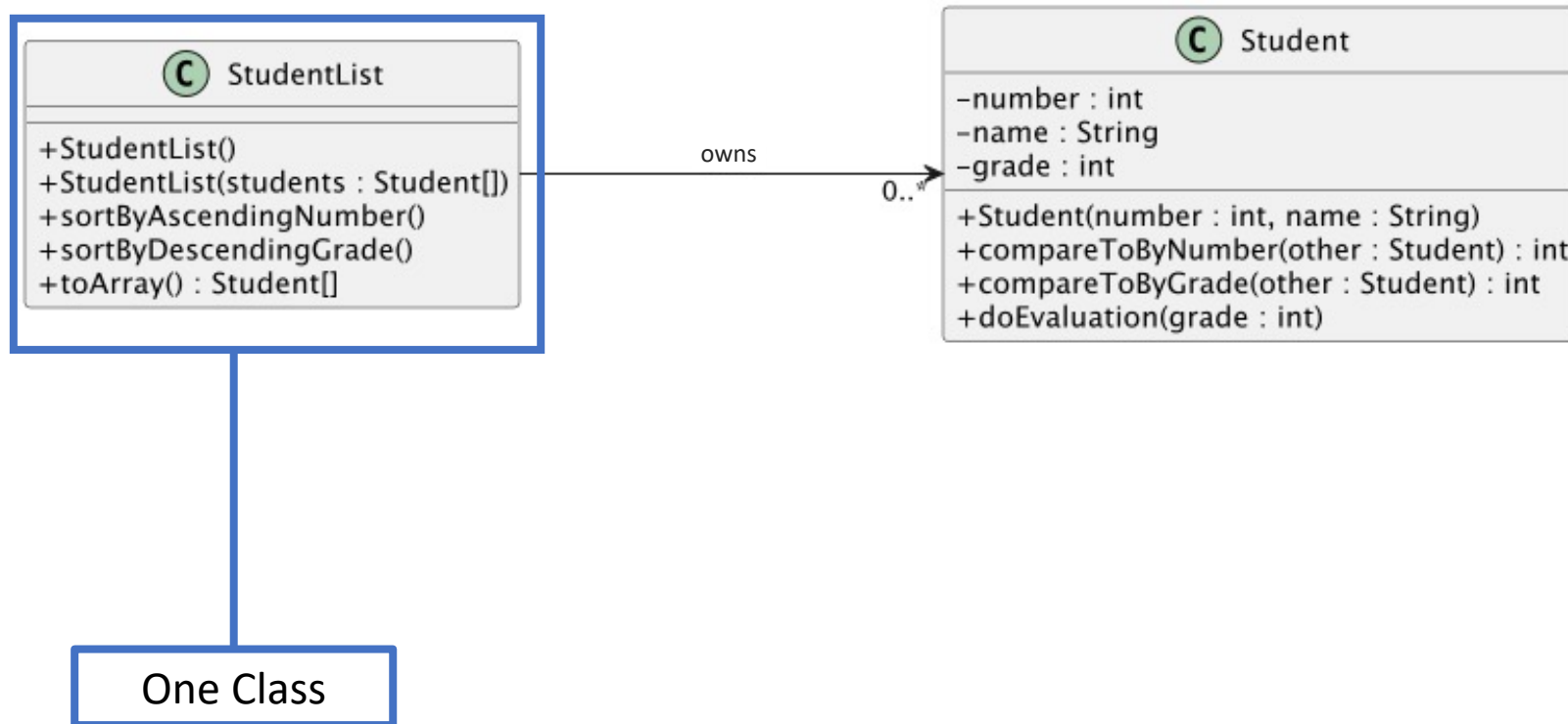
Association between Objects - Examples

- ***StudentList*** ---owns--> [0..*] ***Student***, i.e. any object of type *StudentList* has an association relationship, named *owns* and whose multiplicity ranges from *zero* to *infinite*, with objects of type *Student*
- ***StudentList*** ---hasClassDelegate--> [0..1] ***Student***, i.e. any object of type *StudentList* has an association relationship, named *hasClassDelegate* and whose multiplicity ranges from *zero* to *one*, with objects of type *Student*
- ***Student*** ---worksWith--> [0..3] ***Student***, i.e. any object of type *Student* has an association relationship, named *worksWith* and whose multiplicity ranges from *zero* to *three*, with objects of type *Student*

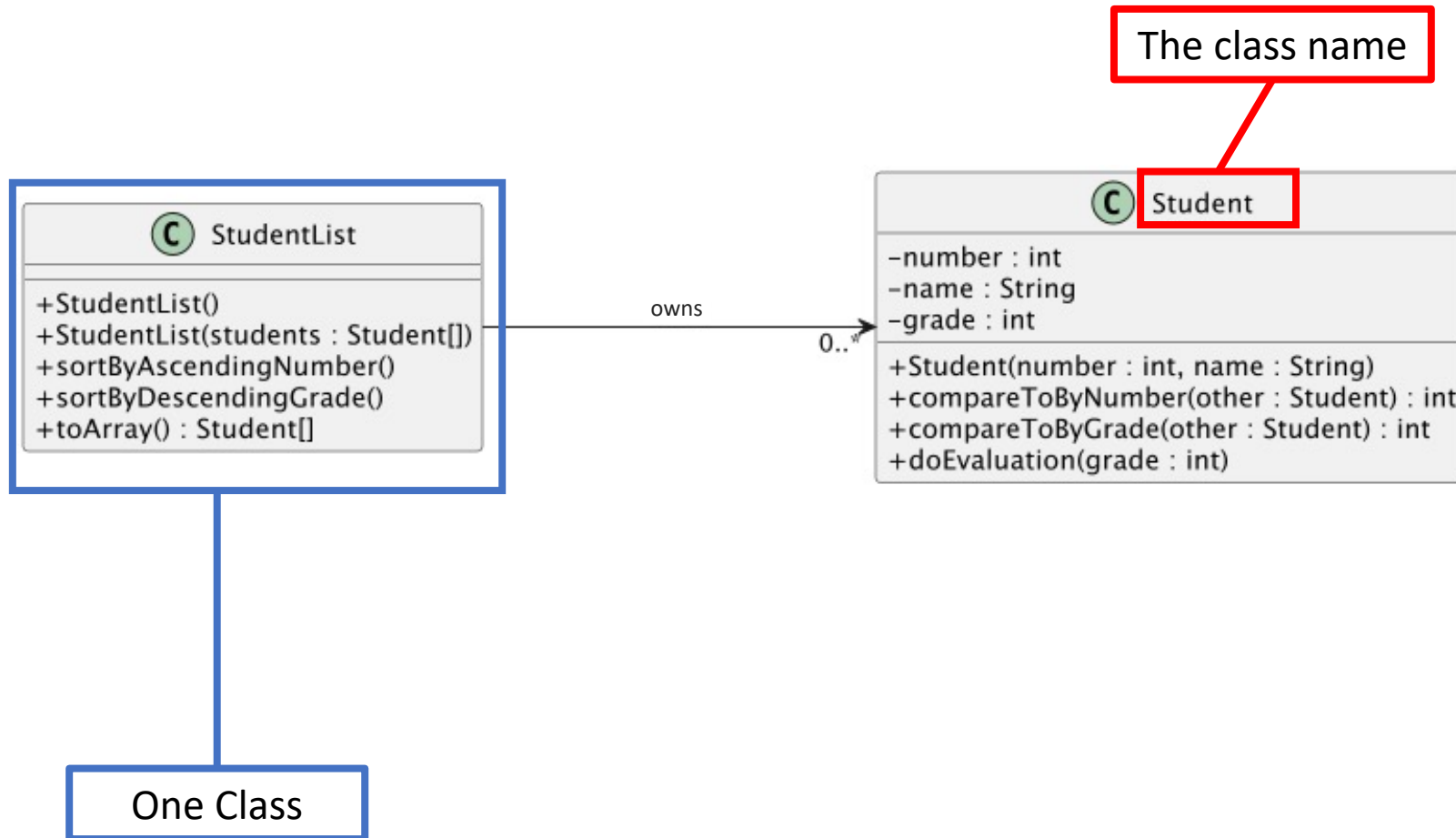
A visual representation of Classes and Associations through UML Class Diagram



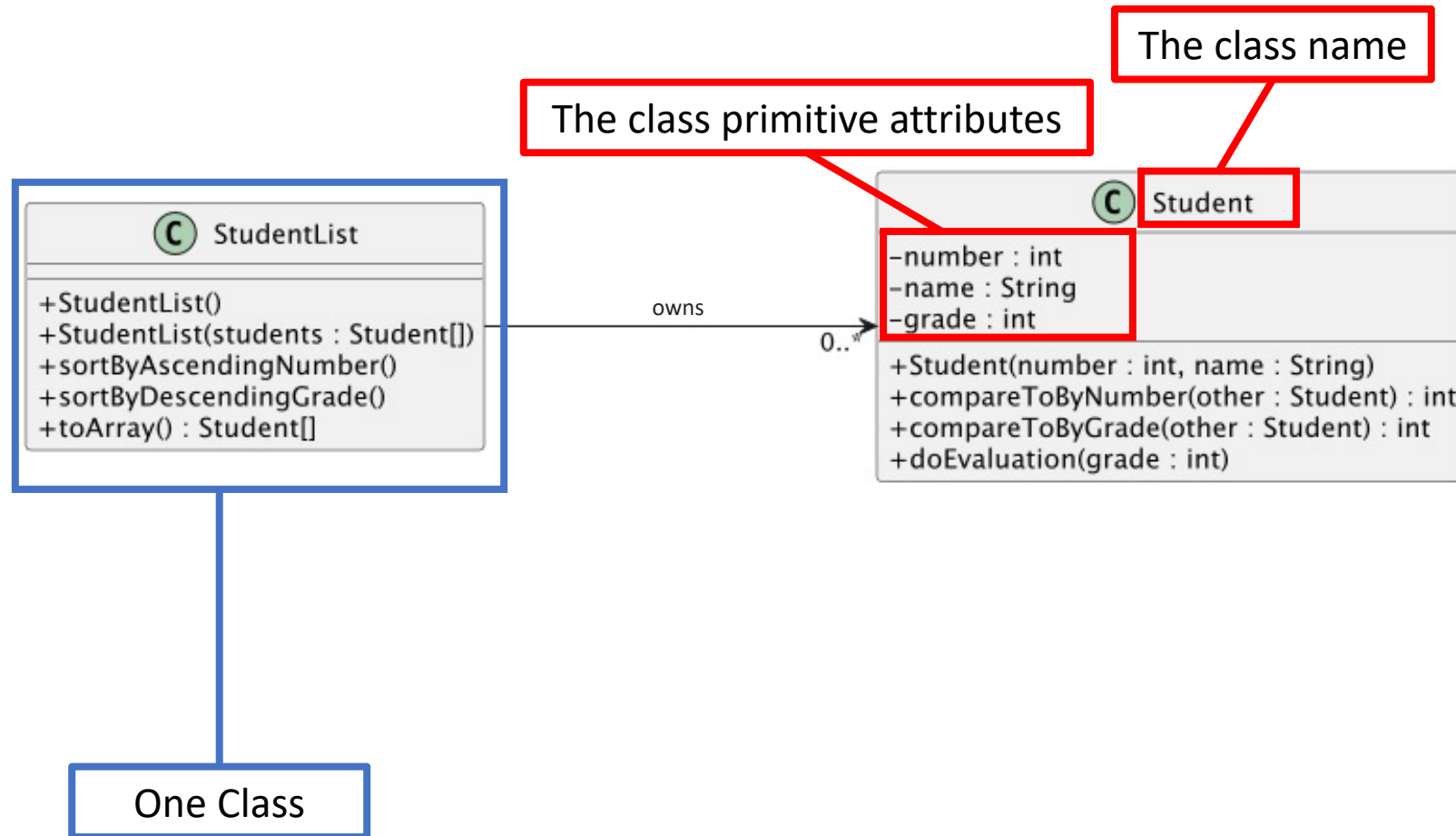
A visual representation of Classes and Associations through UML Class Diagram



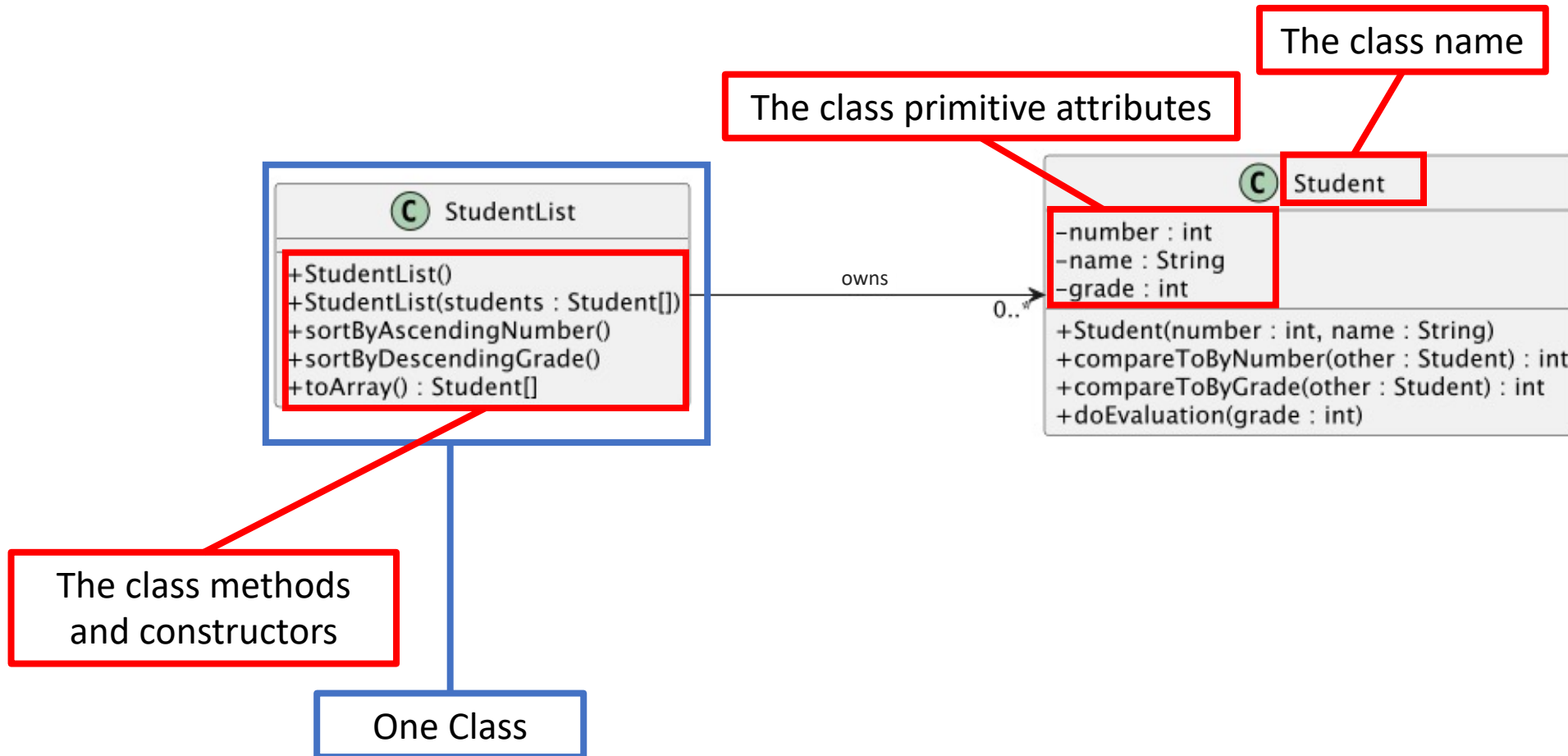
A visual representation of Classes and Associations through UML Class Diagram



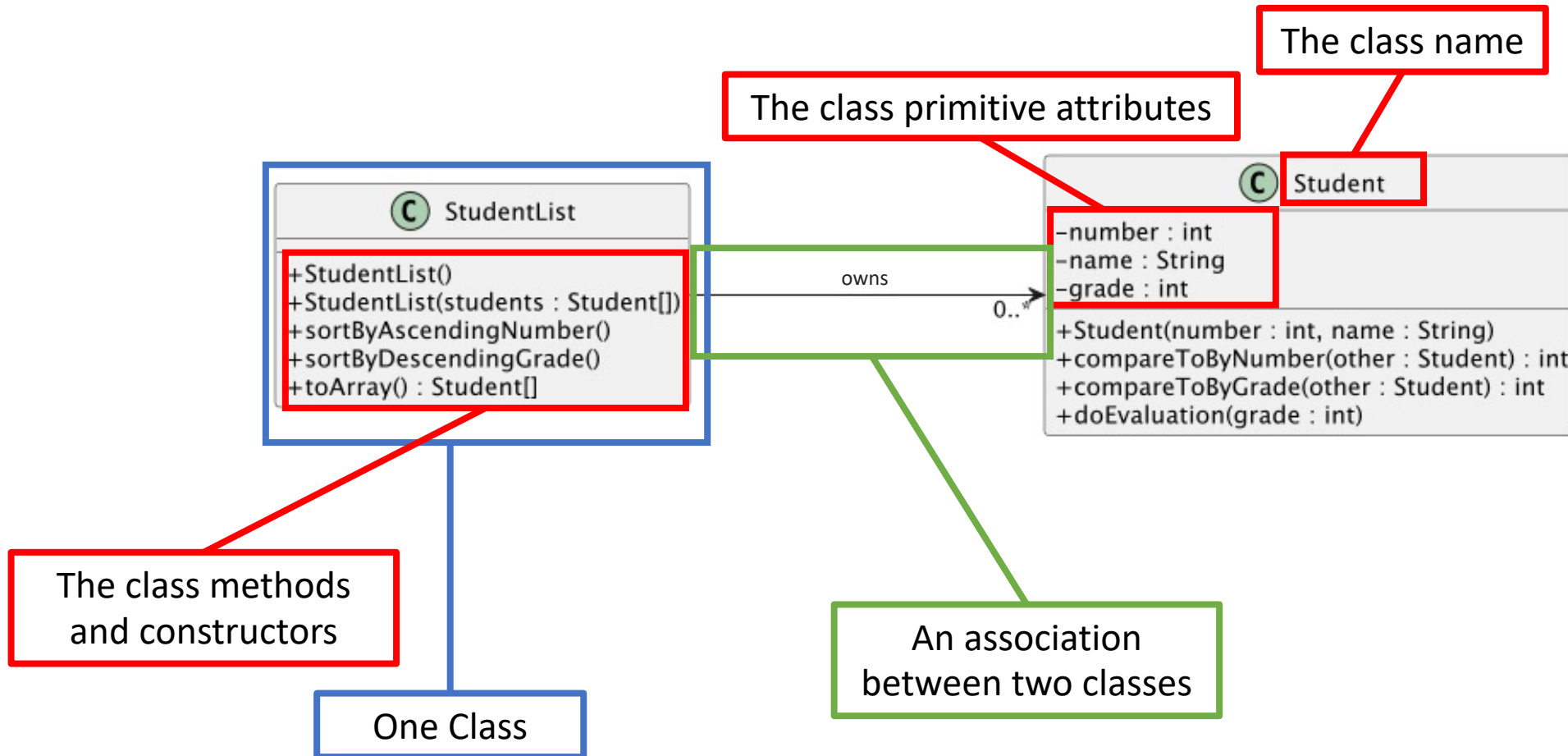
A visual representation of Classes and Associations through UML Class Diagram



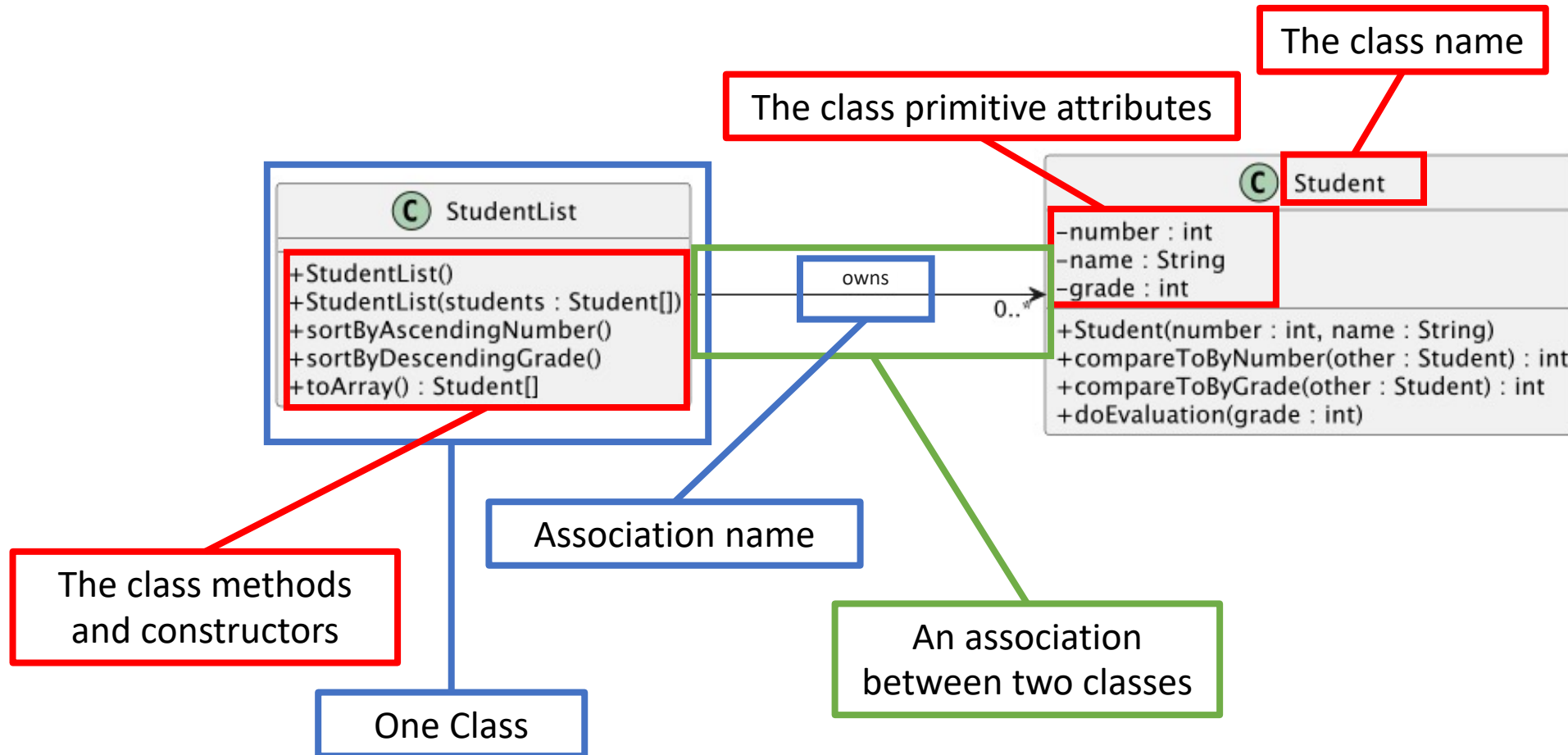
A visual representation of Classes and Associations through UML Class Diagram



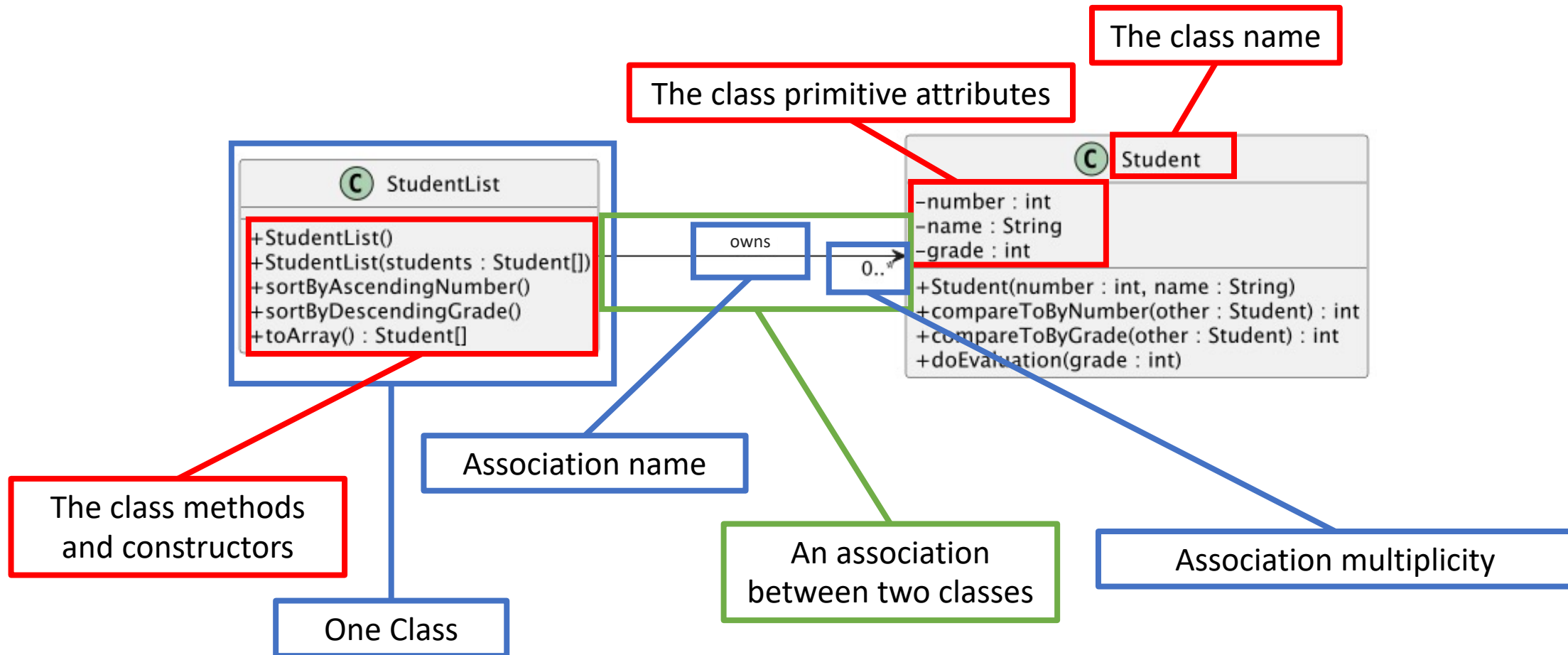
A visual representation of Classes and Associations through UML Class Diagram



A visual representation of Classes and Associations through UML Class Diagram



A visual representation of Classes and Associations through UML Class Diagram



UML Class Diagram – A few notes

- Static representation of classes (not objects) and their relations
 - Association is a kind of relation between classes
 - There are other kind of relations (e.g. dependency, inheritance)
- Presented information about each class might vary in completeness
 - It depends on the purpose the diagram is being used (e.g. domain model, design model)
 - Showing attributes only
 - Showing methods and constructors only
 - Showing attributes, methods and constructors
 - Showing public (denoted as “+”) or private (denoted as “-”) elements

Problem III – More Requirements

- User Stories (US):
 - US03 – As a user, I want to add a student to the students list, so that I can evaluated him/her later.
 - AC01: no duplicate students should exist in the list. Student number is unique, i.e. there are not two students with the same number.
 - US04 – As a user, I want to remove a given student from the students list, so that he/she is no longer enrolled in the course unit.

Analysis & Design (1/2)

- Which class(es) should have the responsibility of adding a new student?
- Which class(es) should have the responsibility to ensure AC01 of US03?
- Which class(es) should have the responsibility of removing a student?

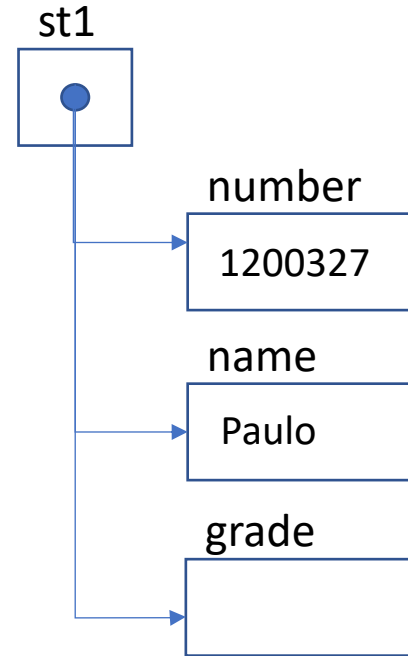
Analysis & Design (2/2)

- Which class(es) should have the responsibility of adding a new student?
 - To know (and store) the new student → ***StudentList***
 - To know (and store) the data of the new student → ***Student***
- Which class(es) should have the responsibility to ensure AC01 of US03?
 - ***StudentList***, since it knows the existing students
 - ***Student***, in the sense that each student knows its own number and, therefore, it should be able to compare with another student's number
- Which class(es) should have the responsibility of removing a student?
 - ***StudentList***, since it knows (and stores) all the existing students
 - ***Student***, in the sense that each student knows if it corresponds to the one being removed

Equals

Objects Equality (1/3)

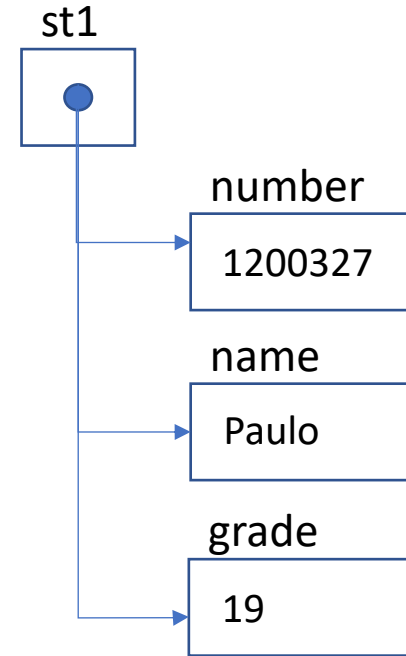
```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
}
```



Objects Equality (1/3)

```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

}
```

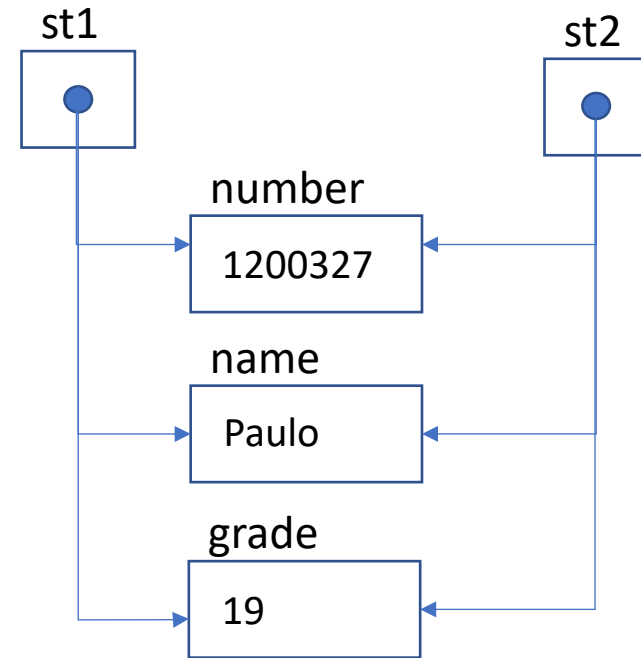


Objects Equality (1/3)

```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

    Student st2 = st1;

}
```



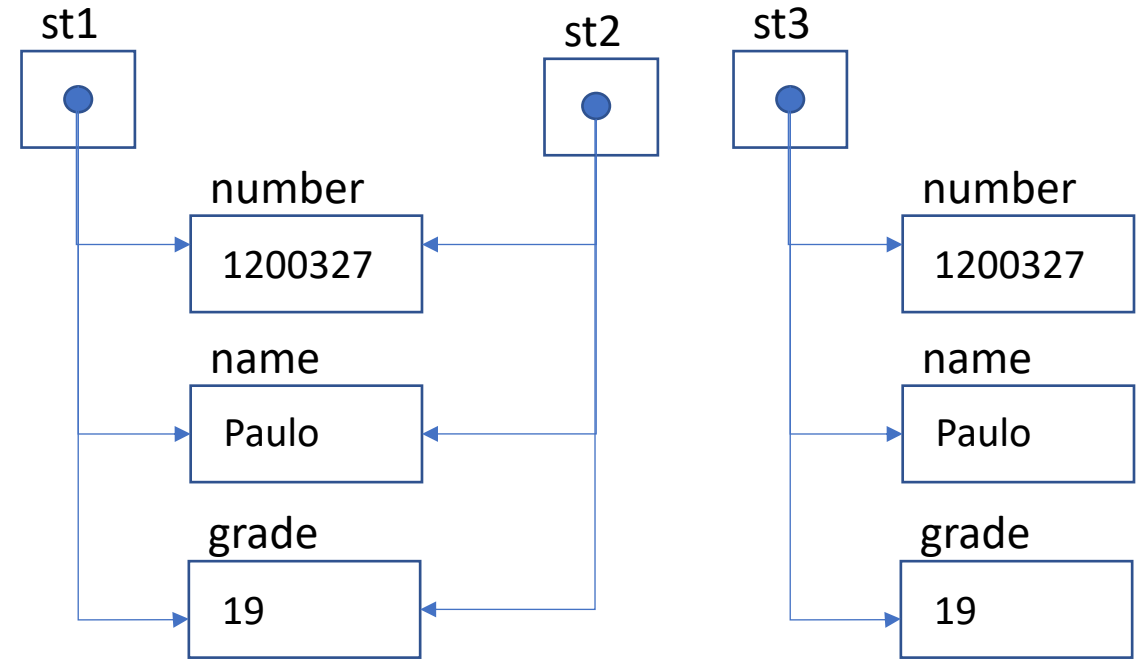
Objects Equality (1/3)

```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

    Student st2 = st1;

    Student st3 = new Student(1200327, "Paulo" );
    st3.doEvaluation(19);

}
```



Objects Equality (1/3)

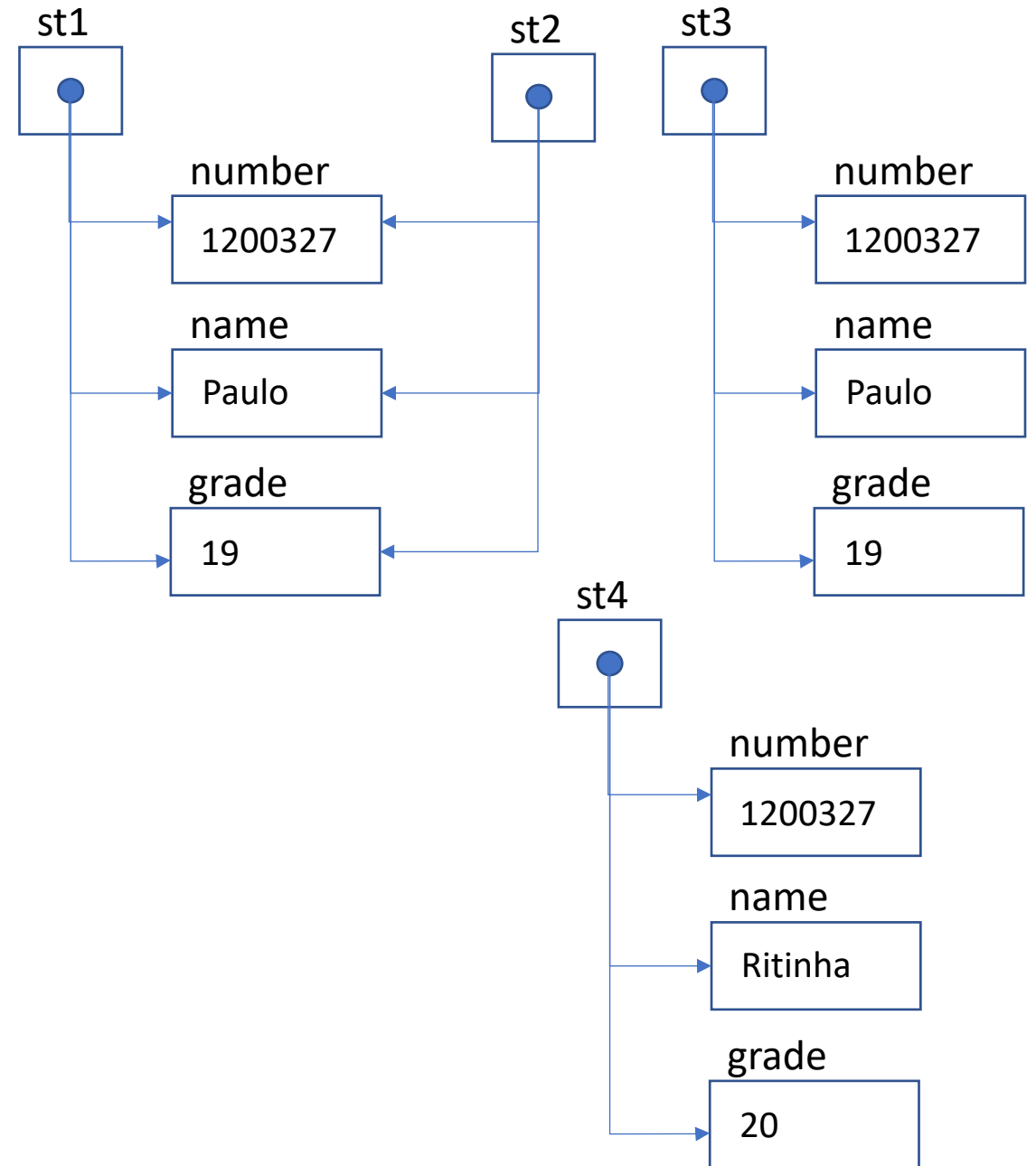
```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

    Student st2 = st1;

    Student st3 = new Student(1200327, "Paulo" );
    st3.doEvaluation(19);

    Student st4 = new Student(1200327, "Ritinha" );
    st4.doEvaluation(20);

}
```



Objects Equality (1/3)

```
public static void checkingObjectEquality()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

    Student st2 = st1;

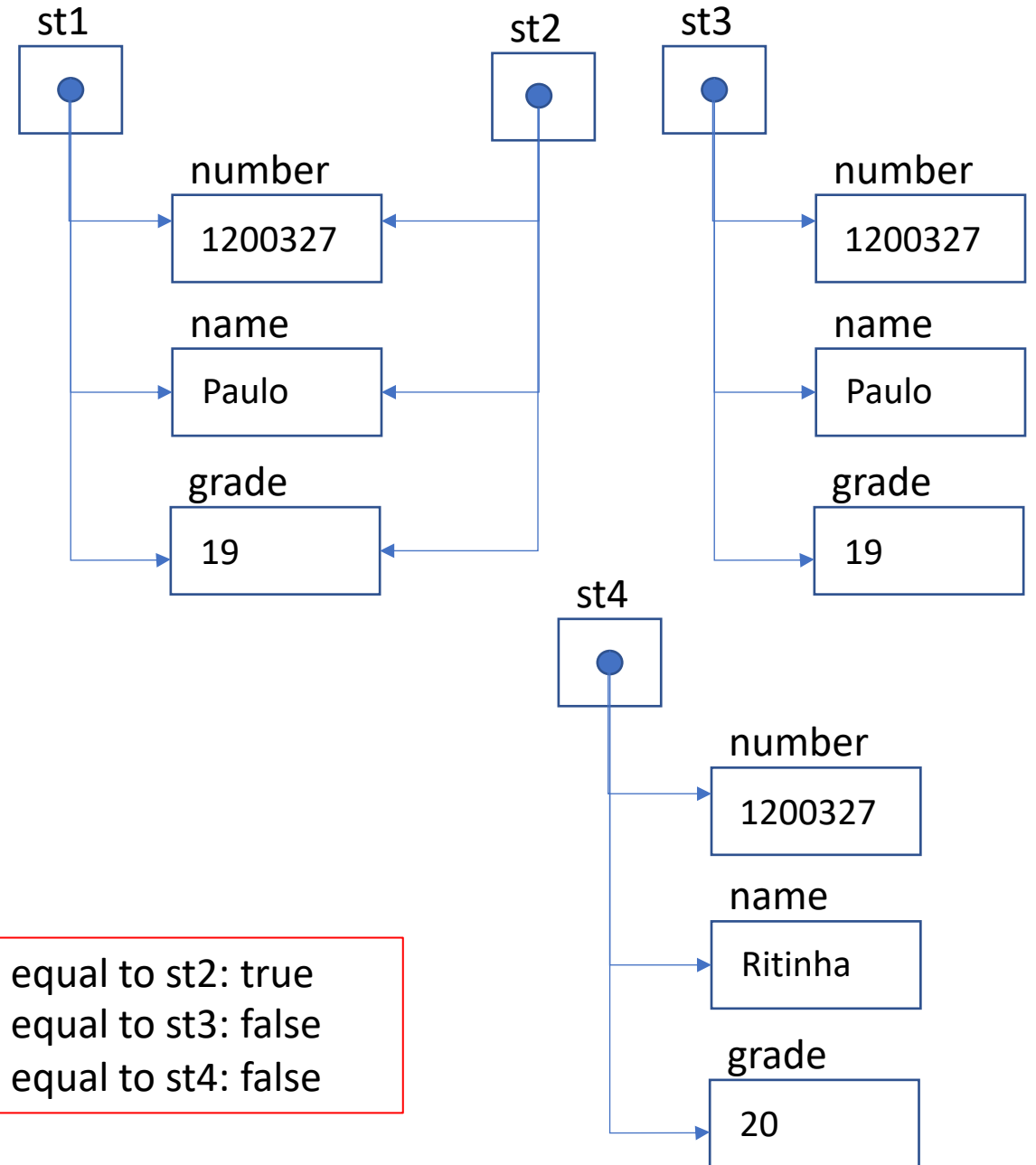
    Student st3 = new Student(1200327, "Paulo" );
    st3.doEvaluation(19);

    Student st4 = new Student(1200327, "Ritinha" );
    st4.doEvaluation(20);

    System.out.println("st1 is equal to st2:" + (st1==st2));
    System.out.println("st1 is equal to st3:" + (st1==st3));
    System.out.println("st1 is equal to st4:" + (st1==st4));
}
```

Check the class **MainToEquals**
in the package **problem.three.version.two**.
studentListWithArray.todo

st1 is equal to st2: true
st1 is equal to st3: false
st1 is equal to st4: false



Objects Equality (2/3)

- Assignment Operator (“=”) → sets two objects with same memory address
- Relational Operator (e.g. “==”) → only checks memory address
- To check objects data use the “*equals*” method
 - By default, all objects have this method (no attribute is checked)
 - Override this method on each class you need to behave differently
 - Check all the necessary attribute data
 - Check object type also

Objects Equality (3/3)

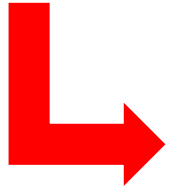
```
public static void checkingObjectEquality2()
{
    Student st1 = new Student(1200327, "Paulo" );
    st1.doEvaluation(19);

    Student st2 = st1;

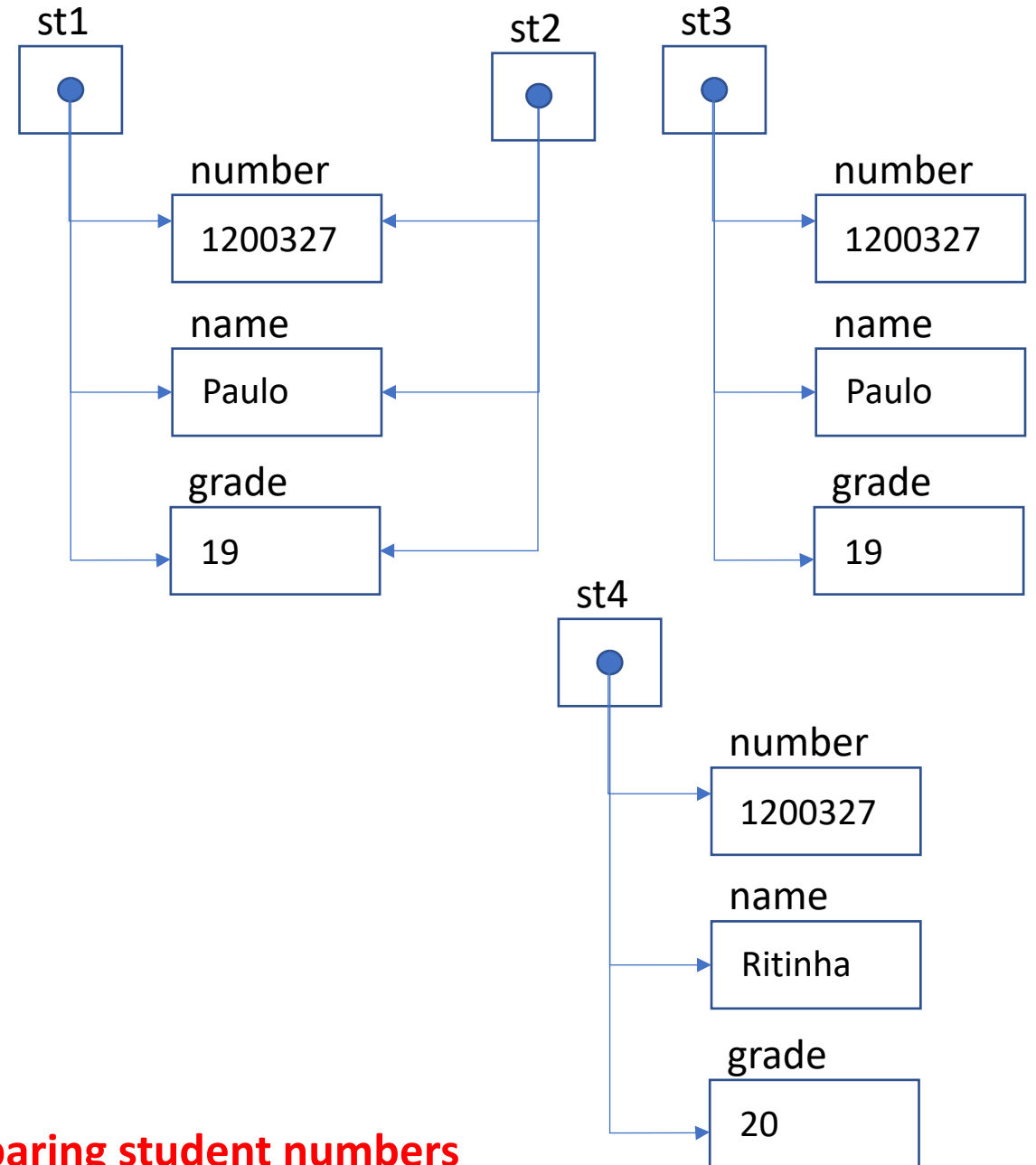
    Student st3 = new Student(1200327, "Paulo" );
    st3.doEvaluation(19);

    Student st4 = new Student(1200327, "Ritinha" );
    st4.doEvaluation(20);

    System.out.println("st1 is equal to st2:" + (st1.equals(st2)));
    System.out.println("st1 is equal to st3:" + (st1.equals(st3)));
    System.out.println("st1 is equal to st4:" + (st1.equals(st4)));
}
```

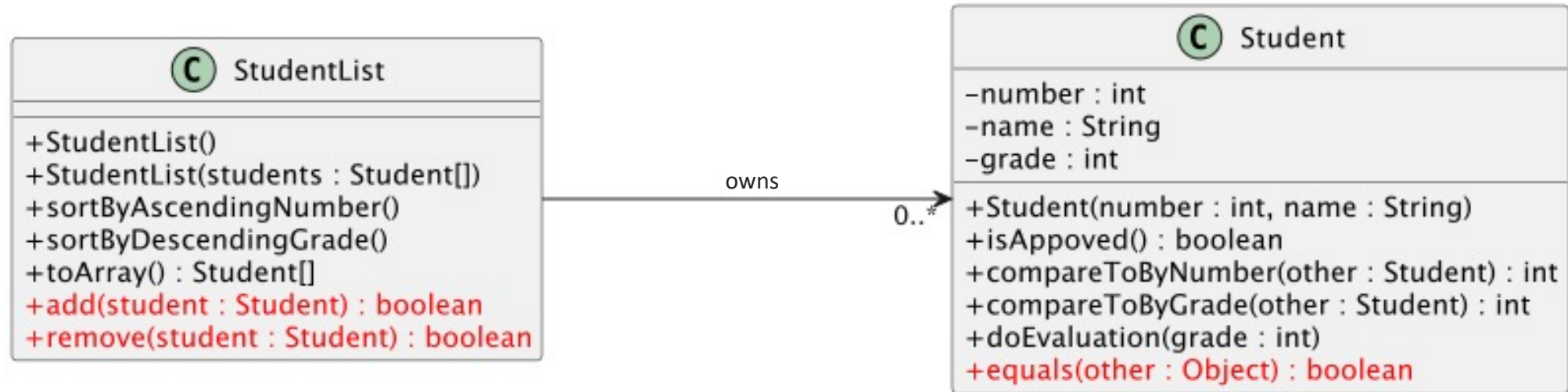


st1 is equal to st2: true
st1 is equal to st3: true
st1 is equal to st4: true



In this case, Student.equals method is also comparing student numbers

Proposed Design



Test Cases – on *Student* class (1/2)

- *equals* method returns
 - *False*, when receiving:
 - a *null value*; or
 - any object that is not of type *Student*; or
 - a student whose number is not equal to the number of the student evaluating equality
 - *True*, when receiving:
 - a student whose number is equal to the number of the student evaluating equality

Notice that on each point several (sub)scenarios can also be considered

Test Cases – on *Student* class (2/2)

```
@Test
public void ensureEqualsTrueDueToSameNumber() {
    // Arrange
    Student studentOne = new Student( 1980398, "Beatriz");
    Student studentTwo = new Student( 1980398, "Beatriz Costa");
    // Act
    boolean result = studentOne.equals(studentTwo);
    // Assert
    assertTrue(result);
}
```

```
@Test
public void ensureEqualsTrueToItself() {
    // Arrange
    Student studentOne = new Student( 1980398, "Beatriz");
    // Act + Assert
    assertTrue(studentOne.equals(studentOne));
}
```

```
@Test
public void ensureEqualsFalseDueToNull() {
    // Arrange
    Student studentOne = new Student( 1980398, "Beatriz");
    // Act
    boolean result = studentOne.equals(null);
    // Assert
    assertFalse(result);
}
```

```
@Test
public void ensureEqualsFalseDueToDifferentType() {
    // Arrange
    Student studentOne = new Student( 1980398, "Beatriz");
    // Act
    boolean result = studentOne.equals(new String("1980398"));
    // Assert
    assertFalse(result);
}
```

```
@Test
public void ensureEqualsFalseDueToDifferentNumbers() {
    // Arrange
    Student studentOne = new Student( 1980398, "Beatriz");
    Student studentTwo = new Student( 1980399, "Beatriz Costa");
    // Act
    boolean result = studentOne.equals(studentTwo);
    // Assert
    assertFalse(result);
}
```

Check the class **StudentTest**
in the test package **problem.three.**
version.two.studentListWithArray.todo

Other tests must be specified

Test Cases – on *StudentList* class (1/3)

- *add* method returns
 - *False*, when receiving:
 - a *null value*; or
 - an already known student (i.e. equals to another one)
 - *True*, when receiving:
 - a not known student (i.e. not equals to anyone known)
- *remove* method returns
 - *True*, when receiving:
 - an already known student (i.e. equals to another one)
 - *False*, when receiving:
 - a *null value*; or
 - a not known student (i.e. not equals to anyone known)

Notice that on each point several (sub)scenarios can also be considered

Test Cases – on *StudentList* class (2/3)

@Test

```
public void ensureAddDifferentStudentWhenArrayNotEmptyWorks(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    Student studentTwo = new Student(1200145, "Sampaio");  
    StudentList stList = new StudentList();  
    stList.add(studentOne);  
    // Act  
    boolean result = stList.add(studentTwo);  
    Student[] content = stList.toArray();  
    //Assert  
    assertTrue(result);  
    assertEquals(2,content.length);  
}
```

@Test

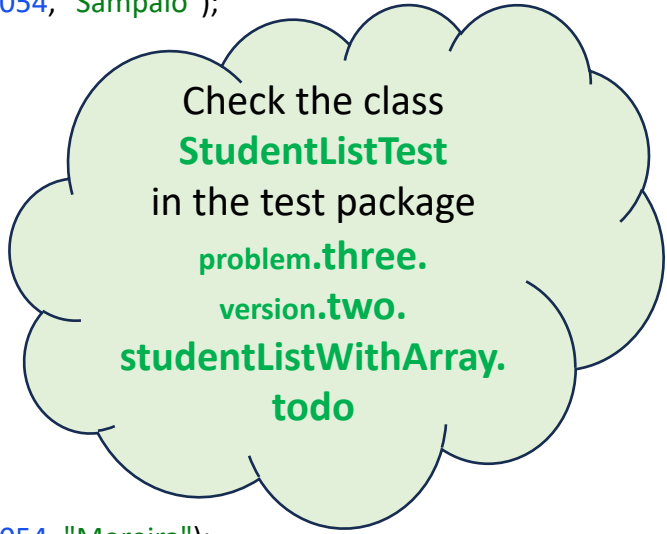
```
public void ensureAddSameStudentTwiceFails(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    StudentList stList = new StudentList();  
    stList.add(studentOne);  
    // Act  
    boolean result = stList.add(studentOne);  
    Student[] content = stList.toArray();  
    //Assert  
    assertFalse(result);  
    assertEquals(1,content.length);  
}
```

@Test

```
public void ensureAddStudentWithSameNumberFails(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    Student studentTwo = new Student(1200054, "Sampaio");  
    StudentList stList = new StudentList();  
    stList.add(studentOne);  
    // Act  
    boolean result = stList.add(studentTwo);  
    Student[] content = stList.toArray();  
    //Assert  
    assertFalse(result);  
    assertEquals(1,content.length);  
}
```

@Test

```
public void ensureAddNullFails(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    StudentList stList = new StudentList();  
    stList.add(studentOne);  
    // Act  
    boolean result = stList.add(null);  
    Student[] content = stList.toArray();  
    //Assert  
    assertFalse(result);  
    assertEquals(1,content.length);  
}
```



Check the class
StudentListTest
in the test package
problem.three.
version.two.
studentListWithArray.
todo

Other tests must be specified

Test Cases – on *StudentList* class (3/3)

@Test

```
public void ensureRemoveMiddleStudentInSeveralWorks(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    Student studentTwo = new Student(1200154, "Sampaio");  
    Student studentThree = new Student(1201154, "Costa");  
    Student studentFour = new Student(1201354, "Lidia");  
    Student studentFive = new Student(1101354, "Maria");  
  
    Student[] all = {studentOne, studentTwo, studentThree, studentFour, studentFive};  
    Student[] expected = {studentOne, studentTwo, studentThree, studentFive};  
    StudentList stList = new StudentList(all);  
    // Act  
    boolean result = stList.remove(studentFour);  
    Student[] content = stList.toArray();  
    //Assert  
    assertTrue(result);  
    assertEquals(expected, content);  
}
```

@Test

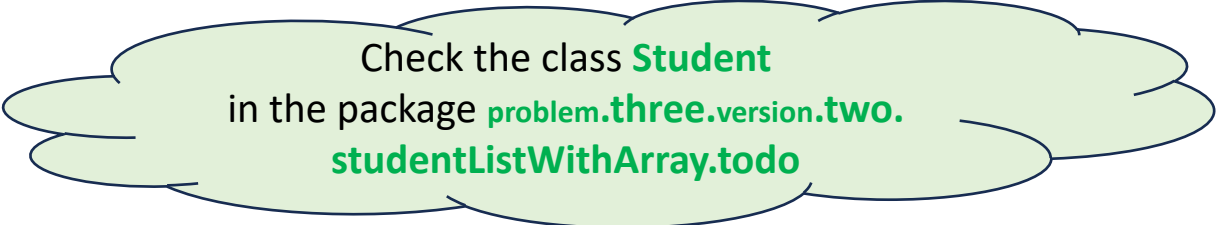
```
public void ensureRemoveSameStudentTwiceReturnsFalse(){  
    // Arrange  
    Student studentOne = new Student(1200054, "Moreira");  
    StudentList stList = new StudentList();  
    stList.add(studentOne);  
    // Act  
    boolean resultOne = stList.remove(studentOne);  
    boolean resultTwo = stList.remove(studentOne);  
    Student[] content = stList.toArray();  
    //Assert  
    assertTrue(resultOne);  
    assertFalse(resultTwo);  
    assertEquals(0, content.length);  
}
```

Other tests must be specified

Implementation – Student class

@Override

```
public boolean equals(Object other) {  
    if (this == other) return true;  
    if (other == null || this.getClass() != other.getClass()) return false;  
    Student student = (Student) other;  
    return (this.number == student.number);  
}
```

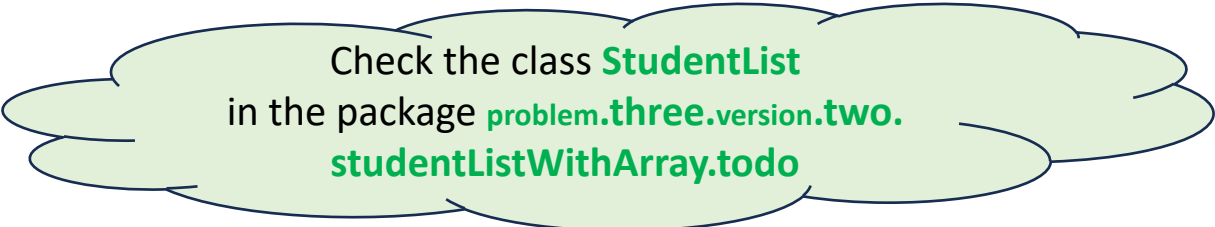


Check the class **Student**
in the package **problem.three.version.two.**
studentListWithArray.todo

Implementation – StudentList class (1/2)

```
public boolean add(Student student) {
    if (student == null)
        return false;
    if (this.exists(student))
        return false;
    this.students = copyStudentsFromArray(this.students, this.students.length+1);
    this.students[this.students.length-1] = student;
    return true;
}

public boolean remove(Student student) {
    if (student == null)
        return false;
    int idx = this.indexOf(student);
    if (idx < 0)
        return false;
    Student[] leftSide = this.copyStudentsFromArray(this.students, idx);
    Student[] rightSide = this.copyStudentsFromArray(this.students, idx+1, this.students.length-idx-1);
    this.students = this.join(leftSide, rightSide);
    return true;
}
```



Check the class **StudentList**
in the package **problem.three.version.two.**
studentListWithArray.todo

Implementation – StudentList class (2/2)

```
private Student[] copyStudentsFromArray(Student[] students, int size) {  
    return this.copyStudentsFromArray(students, 0, size);  
}
```

```
private Student[] copyStudentsFromArray(Student[] students, int start, int size) {  
    Student[] copyArray = new Student[size];  
    int count = 0;  
    for(int idx=start; (count < size) && (idx < students.length); idx++){  
        copyArray[idx-start] = students[idx];  
        count++;  
    }  
    return copyArray;  
}
```

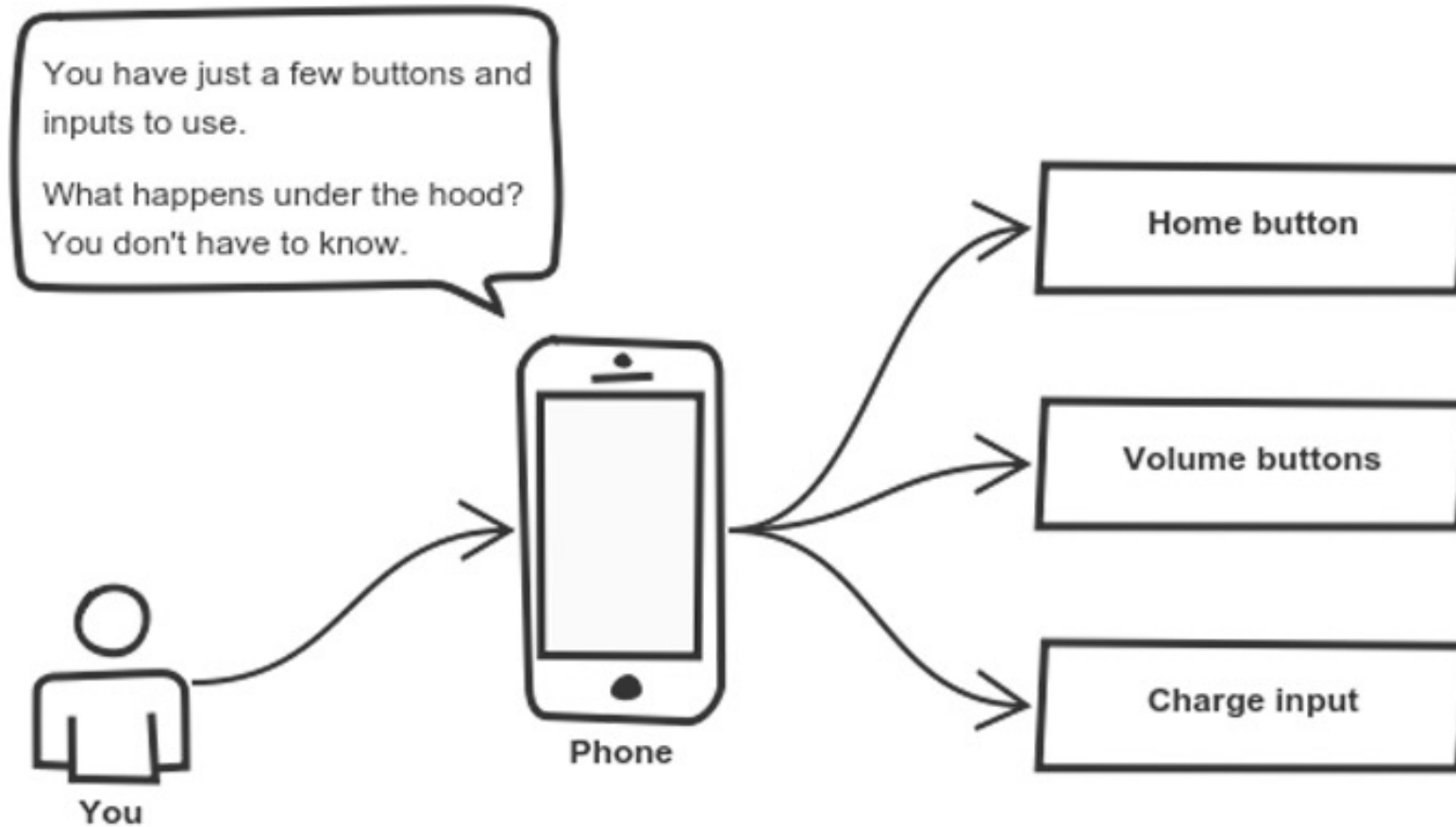
```
private Student[] join(Student[] students1, Student[] students2) {  
    int size = students1.length+students2.length;  
    Student[] copyArray = new Student[size];  
  
    for(int idx=0; (idx < students1.length); idx++){  
        copyArray[idx] = students1[idx];  
    }  
    for(int idx=0; (idx < students2.length); idx++){  
        copyArray[idx+students1.length] = students2[idx];  
    }  
    return copyArray;  
}
```

Abstraction

Abstraction

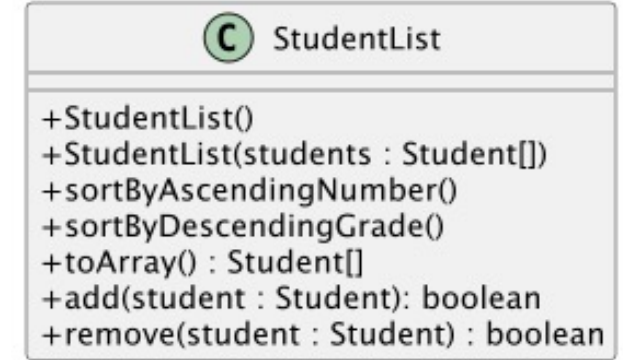
- It is a fundamental concept of OOP
- It is a natural extension of the Encapsulation concept
- It consists on objects
 - **Only** exposing a high-level mechanism for other objects using it (i.e. public API)
 - Preferably, should be easy to use and
 - Should rarely change over time
 - Hiding its own internal complexity and implementation details to the outside
- Advantage(s)
 - Objects implementation might change without breaking the application (i.e. other objects)
 - Eases software maintenance and evolution over time

Abstraction – a real-life example



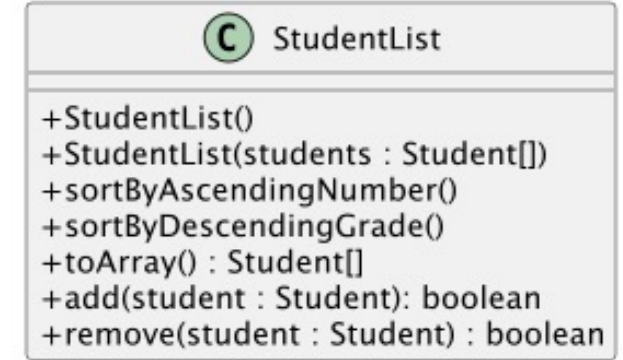
Abstraction – Checking out *StudentList*

- No public attributes
- Two constructors
- 5 public methods
- Internally, it is using
 - An array to store student objects
 - Several private methods to manipulate the students array



Abstraction – Checking out *StudentList*

- No public attributes
- Two constructors
- 5 public methods
- Internally, it is using
 - An array to store student objects
 - Several private methods to manipulate the students array
- Challenge
 - Change internal implementation to use a Java Collection object such as an ArrayList
 - Existing tests should not break
 - No compilation error(s)
 - All tests should run with success



Collections – Some Java classes

- **ArrayList**
- EnumSet
- HashSet
- LinkedHashSet
- LinkedList
- TreeSet
- Vector


Collections – Some Java methods

Return	Method Name
boolean	add(E e)
void	clear()
boolean	contains(Object o)
boolean	equals(Object o)
boolean	isEmpty()
Iterator<E>	iterator()
boolean	remove(Object o)
int	size()
Object[]	toArray()
...	...

New StudentList Implementation (1/2)

```
public class StudentList {  
    // Attributes  
    private ArrayList<Student> students;  
  
    //Constructor  
    public StudentList() {  
        this.students = new ArrayList();  
    }  
    public StudentList(Student[] students) {  
        if (students == null)  
            throw new IllegalArgumentException("Students arraylist should not be null");  
        this.students = new ArrayList();  
        for (Student st:students)  
            this.add(st);  
    }  
  
    // Operations  
    public boolean add(Student student) {  
        if (student == null)  
            return false;  
        if (this.students.contains(student))  
            return false;  
        return this.students.add(student);  
    }  
    ...  
}
```

```
...  
public boolean remove(Student student) {  
    if (student == null)  
        return false;  
    if (!this.students.contains(student))  
        return false;  
    return this.students.remove(student);  
}  
  
public void sortByAscendingNumber() {  
    this.students.sort(new SortByAscendingNumberComparator());  
}  
  
public void sortByDescendingGrade() {  
    this.students.sort(new SortByDescendingGradeComparator());  
}  
  
public Student[] toArray() {  
    Student[] array = new Student[this.students.size()];  
    return this.students.toArray(array);  
}  
...
```



Check the class **StudentList** in the package
problem.three.version.three.
studentListWithArrayList.todo

New StudentList Implementation (2/2)

...

```
private class SortByAscendingNumberComparator implements Comparator<Student> {  
    public int compare(Student st1, Student st2) {  
        return st1.compareToByNumber(st2);  
    }  
}
```

```
private class SortByDescendingGradeComparator implements Comparator<Student> {  
    public int compare(Student st1, Student st2) {  
        return st1.compareToByGrade(st2)*(-1);  
    }  
}
```

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.
- Do all the tests run with success? Why?

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.
- Do all the tests run with success? Why?
 - Yes! Because the externalized behaviors did not change.

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.
- Do all the tests run with success? Why?
 - Yes! Because the externalized behaviors did not change.
 - No! Due to an assumed implicit behavior: tests assume that “toArray()” return students by the order they were added when no “sort” method was used before.
 - If that behavior was the intended one, so you must revise *StudentList* implementation, otherwise you need to update tests to not assume such thing → Avoid implicit behavior

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.
- Do all the tests run with success? Why?
 - Yes! Because the externalized behaviors did not change.
 - No! Due to an assumed implicit behavior: tests assume that “toArray()” return students by the order they were added when no “sort” method was used before.
 - If that behavior was the intended one, so you must revise *StudentList* implementation, otherwise you need to update tests to not assume such thing → Avoid implicit behavior
- Now, do you realize Abstraction advantages?

Abstraction – Challenge Quiz

- Do *StudentList* public API (methods) change? Why?
 - No! There was no need to do it. The public API is stable and is not grounded on internal implementation details.
- Do the tests got any compilation error? Why?
 - No! Because the *StudentList* public API did not change.
- Do all the tests run with success? Why?
 - Yes! Because the externalized behaviors did not change.
 - No! Due to an assumed implicit behavior: tests assume that “toArray()” return students by the order they were added when no “sort” method was used before.
 - If that behavior was the intended one, so you must revise *StudentList* implementation, otherwise you need to update tests to not assume such thing → Avoid implicit behavior
- Now, do you realize Abstraction advantages?
 - Yes! And such advantages are great.

Summary

Summary (1/3)

- Terminology - Distinguishing between
 - Class(es) – as template(s) for things one intends to represent and manipulate
 - Object or Instance – as a representation of something concrete
- Two Fundamental OO concepts
 - Encapsulation – about object's state being private and consistent
 - Abstraction – about object's public API, hiding internal implementation details and complexity

Summary (2/3)

- Designing Classes/Objects
 - Based on the notion of Responsibility
 - Setter methods vs. Business methods (i.e. names inspired on business processes)
 - Getter methods vs. “Tell methods” (i.e. acting on the object’s data)
 - By default, keeping private everything that does not need to be public
- OO Design principles and patterns
 - Tell, Don’t Ask
 - Information Expert
 - (Based on the notion of Responsibility)

Summary (3/3)

- Association between objects
 - Captures a direct relationship between objects
 - Has a (role) name and a multiplicity
- Object equality
 - Assignment Operator ("=") → sets memory address
 - Relational Operator ("==") → compares memory addresses
 - Equals method → compares the attributes of the desired objects
- Collections
 - As a valid alternative to rudimentary data structures (e.g. arrays)
 - Available in all OO languages (e.g. Java)

A final thought:

Procedural vs. Object-Oriented

- *“Procedural code gets information then makes decisions. Object-oriented code tells objects to do things.”*

Alec Sharp

- *“Object-orientation is about bundling data with the functions that operate on that data.”*

Martin Fowler

Upcoming studying topics... (More on)

- OOP fundamental concepts
 - Inheritance
 - Polymorphism
- Analysis OO – the domain model artifact
 - As a means to understand the business
 - Bridging the gap between Requirements and Design OO
- Design OO
 - Principles and patterns
 - GRASP – General Responsibility Assignment Software Patterns
 - SOLID Principles
 - Gang of Four (GoF) Patterns
 - UML based artifacts
 - Sequence Diagrams - to represent interaction between objects
 - Class Diagrams – to represent a static view between classes