Programare avansata pe obiecte - laborator 2 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

1) **Development environment:**
   a) Choose an IDE that can assist with using class libraries and frameworks: **eg**: IntelliJ (https://www.jetbrains.com/idea/), Eclipse or NetBeans

---

Object oriented programming - lab 2

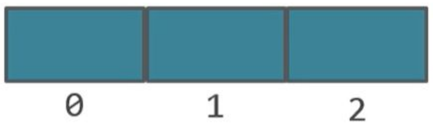
Arrays:

- provide an ordered collection of elements => they allow us to store multiple values of a common type under a single name.

```
package com.paolabs.lab2;

public class Exemplul1 {

   public static void main(String[] args) {
       float[] values = new float[3];
   }
}
```

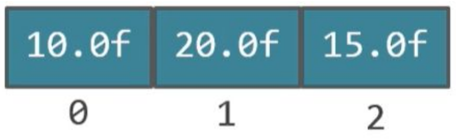
- square bracket right after float indicates that we're declaring values as an array that allocates out a name that can access the array
- we use the keyword "new" and then float[3] to allocate space to store 3 float values, accessible under the name, values
- each element is accessed via an index - and indexes range from 0 to number of elements in the array minus 1

values



```java
package com.paolabs.lab2;

public class Exemplul1 {

   public static void main(String[] args) {
       // Arrays - provide an ordered collection of elements
       float[] values = new float[3];

       values[0] = 10.0f; // this stores 10.0 in the zero position
       values[1] = 20.0f;
       values[2] = 15.0f;
   }
}
```

values



- arrays expose a value called **length** to tell us how many elements are in the array

```java
package com.paolabs.lab2;

public class Exemplul1 {

   public static void main(String[] args) {
       // Arrays - provide an ordered collection of elements
       float[] values = new float[3];

       values[0] = 10.0f; // this stores 10.0 in the zero position
       values[1] = 20.0f;
       values[2] = 15.0f;

       // go through the values array and add everything up
       float sum = 0.0f; // allocate some space to store the result

       // arrays makes it easy to move through these values using loops
       for (int i = 0; i < values.length; i++) {
           sum += values[i];
```

```
        }

        System.out.println(sum);
    }
}
```

```
package com.paolabs.lab2;

public class Exemplul2 {

    public static void main(String[] args) {

        float[] values = {10.0f, 20.0f, 15.0f};
        // after we declare the array name, we can use an open bracket
        // and then just list the values and enclose it
        // => It automatically allocates the right number of spaces,
        // in this case, three, and initializes the value in each space

        float sum = 0.0f;

        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }

        System.out.println(sum);
    }
}
```

Representing Complex Types with Classes:

Java is an object-oriented language.

Objects encapsulate the data, operations, and usage semantics.

An object represents what we want to work on and how we work with it.

- Allows storage and manipulation details to be hidden
- Separates "what" is to be done from "how" it is done

Classes - provide a structure for describing and creating objects.

A class is a **template** for creating an **object**.

- Declared with the keyword “class” followed by the class name
- Body of the class is contained within brackets

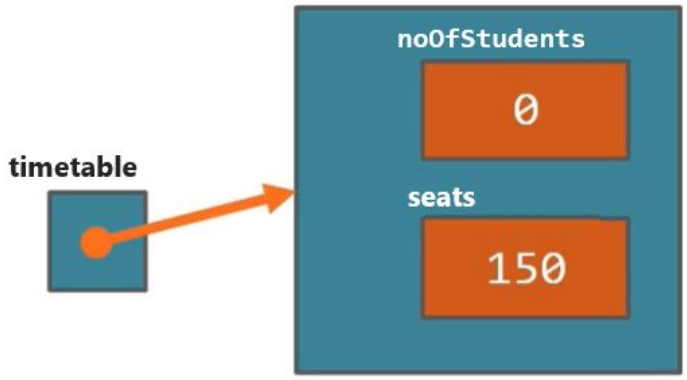
A class is made up of both state and executable code:

- **Fields** = store object state
- **Methods** = executable code that manipulates state and performs operations
- **Constructors** = executable code used during object creation to set initial state

```java
package com.paolabs.lab2.exemplul3;

class Timetable {

   // fields - store object state
   int noOfStudents;
   int seats;

   /**
    * constructor
    */
   public Timetable() {
       this.noOfStudents = 0;
       this.seats = 150;
   }

   /**
    * methods - executable code that manipulates state
    * and performs operations
    */
   void addStudent() {
       if (noOfStudents < seats) {
           noOfStudents++; // <=> noOfStudents = noOfStudents + 1;
       }
   }

}
```

Using Classes:

- Declaring a variable of type **Timetable** simply allocates space to store what we call a reference to the actual object we want to use. To create an instance of our class, our object Timetable, we have to use the new keyword.



```
package com.paolabs.lab2.exemplul3;

public class Exemplul3 {

   public static void main(String args[]) {

       // declare a variable of type Timetable
       Timetable timetable;
       timetable = new Timetable();
       // allocates the memory described by the class
       // returns a reference to the allocated memory

       // declare the variable and assigned the object in a single
       // statement
       Timetable uniBucTimetable = new Timetable();
   }
}
```

Classes are reference types:

```java
package com.paolabs.lab2.exemplul3;

public class Exemplul4 {

   public static void main(String args[]) {

       // Classes are reference types
       Timetable uniBucTimetable = new Timetable();
       Timetable poliTimeTable = new Timetable();

       poliTimeTable.addStudent();
       System.out.println(poliTimeTable.noOfStudents); // => 1

       poliTimeTable = uniBucTimetable;
       System.out.println(poliTimeTable.noOfStudents); // => 0

       uniBucTimetable.addStudent();
       uniBucTimetable.addStudent();

       System.out.println(poliTimeTable.noOfStudents); // => 2
   }
}
```

- If we declare a variable uniBucTimetable that points to a new Timetable, that allocates out to variable uniBucTimetable, allocates that to the instance of our object and puts a reference to that object into uniBucTimetable.
- We do the same thing for poliTimeTable. That creates a separate object and assigns it into a separate variable, poliTimeTable.
- Those two objects are completely separate from one another. If we call addStudent on poliTimeTable, the object instance pointed to by poliTimeTable will have its noOfStudents increment from 0 to 1.
- If we print out poliTimeTable noOfStudents, it prints out 1.
- If we assign uniBucTimetable to poliTimeTable, that doesn't copy the entire object pointing from by uniBucTimetable into poliTimeTable. All that does is reassign the reference so that poliTimeTable, instead of pointing to the object it currently points to, will now point to the same object that uniBucTimetable points to.

- Now once that object previously pointed by poliTimeTable is no longer referenced, that object goes away, and poliTimeTable and uniBucTimetable are pointing now to the exact same object.
- If we print out poliTimeTable noOfStudents, it prints out 0 because that's the value of noOfStudents in the object it's pointing to now.

Encapsulation and Access Modifiers:

The internal representation of objects should generally be hidden. The user of an object shouldn't have to know a whole lot about the way that object is built. The idea of hiding this internal representation is called **encapsulation**. In order to achieve encapsulation we need to use **access modifiers**.

The most basic access modifier at all is to have **no access** modifier. If a class or a class member does not have an access modifier on it, then it's considered what we call *package private, only usable within its own package.*

A common access modifier is the **public access modifier.** If a class or a class member is marked as public, that means it *can be used anywhere in the program.*

Private access modifier: If a class member is marked as private, it's *only accessible from within the class where it's declared.*

```
package com.paolabs.lab2.exemplul5;

public class Flight {

   private int passengers;
   private int seats;

   public Flight() {
       seats = 150;
       passengers = 0;
   }

   public void addPassenger() {
       if (passengers < seats) {
           passengers++;
       } else {
           handleTooManyPassengers();
       }
   }
```

```
    private void handleTooManyPassengers() {
        System.out.println("There are too many passengers.");
    }
}
```

Method Basics:

```
access-modifier return-type name (typed-parameter-list) {
  statements
}
```

- Executable code that manipulates state and performs operations:
    - Name: same rules and conventions as variables; should be a verb or an action
    - Return type: use void when no value is returned
    - Typed parameter list: can be empty
    - Body contained with brackets

- Exiting from the method:
    - The end of the method is reached
    - A return statement is encountered
    - An error occurs

```
public void showSum(float x, float y, int count) {
   if (count < 1) {
       return;
   }
   float sum = x + y;
   for (int i = 0; i < count; i++) {
       System.out.println(sum);
   }
}
```

- A method returns a single value:
    - A primitive value
    - A reference to an object
    - A reference to an array (Arrays are objects)

Special References: this and null:

- The “**this**” reference is used to refer to the current object:
    - useful for any time we want to be very explicit and indicate that we're referring to the current object

- it's also used for forward cases where an object may need to pass references to itself to other methods.

```
public boolean hasRoom(Flight flight2) {
   int total = this.passengers + flight2.passengers;
   return total <= seats;
}
```

- Null is a reference, and this is a literal:

- Null refers to an uncreated object
- can be assigned to any reference variable => this reference variable isn't pointing to anything yet

Field Encapsulation, Accessors, and Mutators:

- In most cases, a class fields should not be directly accessible outside of the class
- Field Encapsulation
    - Helps to hide implementation details;
    - Use methods to control fields access
- Use the Accessor/ Mutator pattern to control field access
    - Accessors retrieve field value: also called **getter;** create a method name: getFieldName
    - Mutator modifies field value: also called **setter**; create a method name: setFieldName

```
package com.paolabs.lab2.exemplu5;

public class Flight {

   private int passengers;
   private int seats;

   public Flight() {
       seats = 150;
       passengers = 0;
   }

   public int getPassengers() {
       return passengers;
   }
```

```
    public void setPassengers(int passengers) {
        this.passengers = passengers;
    }

    public int getSeats() {
        return seats;
    }

    public void setSeats(int seats) {
        this.seats = seats;
    }

}
```

Applications:

1. Write a Java application that allows the user to enter up to 20 integer grades into an array. When the user enters -1 the reading of grades stops and the program displays the average of the grades.

2. Write a program to create a **Person** object, with the following attributes: name as string, surname as string, age as string, identity number as long, type as string. Define a constructor for this class as well as accessors and mutators for all the attributes. Create two objects of type Person (John, Doe, 24, 1123444, student) and (Jane, Roe, 56, 2233444, teacher) and display the information for them on separate lines.

3. Write a program to create a **Room** object, the attributes of this object are room **number**, room **type** and room **floor**. Define a constructor for this class as well as accessors and mutators for all the attributes. Create two objects of type Room (with room number: 12A and 12B, room type: normal and tech and room floor: 3 and 7) and display the information for them on separate lines.

4. Write a program to create an object **Subject** with the following attributes: room as Room, noOfStudents as integer, teacher as Person. Define a constructor for this class as well as accessors and mutators for all the attributes. Create two objects of type Subject and display the information for them on separate lines.