Programare avansata pe obiecte - laborator 4 (231)

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

Object oriented programming - following lab 3

Polymorphism:

Polymorphism is the ability of an object to take on many forms.

- **Static Polymorphism:**
    - Eg. The operator *"+"* can be used in different contexts: with integers numbers, real numbers, strings. An operator with this behavior is also called **overloaded (operator supraîncarcat)**

```
package com.paolabs.lab4.ex1;

public class Exemplul1 {

    public static void main(String[] args) {
        // static polymorphism
        System.out.println(1+2); // adding integer numbers
        System.out.println(3.14 + 0.00015); // adding real numbers
        System.out.println("lab " + "PAO"); // string concatenation
    }

}
```

    - **Method Overloading** - when methods with the same name have different algorithms. Eg: In class **Product** the method **computePrice**:

```
package com.paolabs.lab4.ex1;

public class Product {

   private String name;
   private String code;
   private double price;
```

```
    private double sellingPrice;

    public Product() {
    }

    public double computePrice(int noOfItems) {
        return sellingPrice * noOfItems;
    }

    public double computePrice(int noOfItems, double discount) {
        sellingPrice -= (sellingPrice * discount) / 100;
        return sellingPrice * noOfItems;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public double getSellingPrice() {
        return sellingPrice;
    }
```

```
    public void setSellingPrice(double sellingPrice) {
        this.sellingPrice = sellingPrice;
    }
}
```

```
package com.paolabs.lab4.ex1;

public class Exemplul2 {

    public static void main(String[] args) {
        // static polymorphism
        Product p1 = new Product();
        p1.setName("Flour");
        p1.setPrice(12.0);
        p1.setSellingPrice(23.3);

        System.out.println("Ten items of product = " + p1.computePrice(10));
        System.out.println("Ten items of product with discount = " +
p1.computePrice(10, 10));
    }

}
```

- **Dynamic Polymorphism**
    - Method overriding - when an overridden method is called through a superclass reference, the interpreter determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.

```
package com.paolabs.lab4.ex1.exemplul3;

public class Shape {

    public Shape() {
    }

    public void draw() {
        System.out.println("Draw a shape");
    }
```

```
    public void delete() {
        System.out.println("Delete the shape");
    }
}
```

```
package com.paolabs.lab4.ex1.exemplul3;

public class Circle extends Shape {

    @Override
    public void draw() {
        System.out.println("Draw a circle");
    }

    @Override
    public void delete() {
        System.out.println("Delete a circle");
    }
}
```

```
package com.paolabs.lab4.ex1.exemplul3;

public class Triangle extends Shape {

    @Override
    public void draw() {
        System.out.println("Draw a triangle.");
    }

    @Override
    public void delete() {
        System.out.println("Delete the triangle");
    }
}
```

```
package com.paolabs.lab4.ex1.exemplul3;

public class Exemplul3 {
```

```
    public static void main(String[] args) {
        // dynamic polymorphism
        Shape shape = new Shape();
        Circle circle = new Circle();
        Triangle triangle = new Triangle();

        Shape ref;

        ref = shape;
        ref.draw();
        ref.delete();

        ref = circle;
        ref.draw();
        ref.delete();

        ref = triangle;
        ref.draw();
        ref.delete();

    }

}
```

Result:

```
Draw a shape
Delete the shape
Draw a circle
Delete a circle
Draw a triangle.
Delete the triangle
```

Object oriented programming - lab 4

Association, Composition and Aggregation

We call **association** those relationships whose objects have an independent lifecycle and where there is no ownership between the objects. => Association is a relationship between two separate classes and the association can be of any type eg. one to one, one to many etc.

```java
package com.paolabs.lab4.ex2;

public class Article {

   private int quantity;
   private ArticleDetails articleDetails;

   public Article(String name, double unitPrice, int quantity) {
       this.articleDetails = new ArticleDetails(name, unitPrice);
       this.quantity = quantity;
   }

   public int getQuantity() {
       return quantity;
   }

   public void setQuantity(int quantity) {
       this.quantity = quantity;
   }

   public ArticleDetails getArticleDetails() {
       return articleDetails;
   }

   public void setArticleDetails(ArticleDetails articleDetails) {
       this.articleDetails = articleDetails;
   }

   public double computePrice() {
       return this.articleDetails.getUnitPrice() * this.quantity;
   }
}
```

```java
package com.paolabs.lab4.ex2;

public class ArticleDetails {

   private String name;
   private double unitPrice;

   public ArticleDetails(String name, double unitPrice) {
```

```
        this.name = name;
        this.unitPrice = unitPrice;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getUnitPrice() {
        return unitPrice;
    }

    public void setUnitPrice(double unitPrice) {
        this.unitPrice = unitPrice;
    }
}
```

```
package com.paolabs.lab4.ex2;

public class TestArticle {

    public static void main(String[] args) {
        Article[] articles = new Article[3];
        articles[0] = new Article("Mouse", 23.2, 2);
        articles[1] = new Article("MousePad", 5.2, 3);
        articles[2] = new Article("Printer", 58.8, 1);

        for (int i=0; i< articles.length; i++) {
            System.out.print(articles[i].getArticleDetails().getName());
            System.out.print("->");

System.out.println(articles[i].getArticleDetails().getUnitPrice());
        }
    }
}
```

In Object-Oriented programming, an Object communicates with other Objects to use functionality and services provided by that object.
Composition and **Aggregation** are the two forms of association.

Aggregation:

- represents a Has-A **relationship**.
- It is a unidirectional association, eg: a department can have students
- both entries can survive individually which means ending one entity will not affect the other entity

```
package com.paolabs.lab4.ex3;

public class Student {

   private String name;
   private String surname;
   private int id;
   private int age;

   public Student(String name, String surname, int id) {
       this.name = name;
       this.surname = surname;
       this.id = id;
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }

   public String getSurname() {
       return surname;
   }

   public void setSurname(String surname) {
       this.surname = surname;
   }
```

```
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
package com.paolabs.lab4.ex3;

public class Department {

    private String name;
    private Student[] students;

    public Department(String name, Student[] students) {
        this.name = name;
        this.students = students;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Student[] getStudents() {
        return students;
    }
```

```
    public void setStudents(Student[] students) {
        this.students = students;
    }
}
```

```
package com.paolabs.lab4.ex3;

public class TestDepartment {

    public static void main(String[] args) {
        Student s1 = new Student("Silvia", "Butan", 7);
        Student s2 = new Student("Maria", "Popescu", 8);

        Student[] students = {s1, s2};

        Department department = new Department("IT", students);

        System.out.println(department.getName());
        System.out.println(department.getStudents().length);
    }
}
```

Composition:

- is a restricted form of Aggregation in which two entities are highly dependent on each other.
- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object cannot exist without the other entity.

Eg: Article and ArticleDetails remain the same:

```
package com.paolabs.lab4.ex4;

public class Sale {

    private int tva = 20;
    private double total;
    private Article[] articles;
    private int currentNo;
```

```java
public Sale(int noArticles) {
    this.articles = new Article[noArticles];
}

public void setTva(int tva) {
    this.tva = tva;
}

public int getTva() {
    return tva;
}

public double getTotal() {
    return total;
}

public void setTotal(double total) {
    this.total = total;
}

public Article[] getArticles() {
    return articles;
}

public void setArticles(Article[] articles) {
    this.articles = articles;
}

public int getCurrentNo() {
    return currentNo;
}

public void setCurrentNo(int currentNo) {
    this.currentNo = currentNo;
}

public void addArticle(String name, double price, int quantity) {
    if (this.currentNo < this.articles.length) {
        this.articles[currentNo++] = new Article(name, price, quantity);
    }
}
```

```
    public double computeTotal() {
        for (int i = 0; i < this.articles.length; i++) {
            this.total += this.articles[i].computePrice();
        }
        return total + (total * tva / 100);
    }
}
```

```
package com.paolabs.lab4.ex4;

public class TestSale {

    public static void main(String[] args) {

        Sale sale = new Sale(2);
        sale.addArticle("mouse", 120, 1);
        sale.addArticle("mousePad", 23, 2);
        System.out.println(sale.computeTotal());

    }
}
```

Strings - String class:

Strings are Objects that are backed internally by a char array. Since arrays are immutable (cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

```
package com.paolabs.lab4.ex5;

public class Exemplul1 {

    public static void main(String[] args) {
        // String literal
        String s1 = "this is a string";

        // using the new keyword
        String s2 = new String("this is another string");
    }
}
```

```java
package com.paolabs.lab4.ex5;

public class Exemplul2 {

   public static void main(String[] args) {
       String s1 =  new String("this is a String");
       s1.toUpperCase();

       System.out.println(s1);
   }
}
```

Result:

```
this is a String
```

```java
package com.paolabs.lab4.ex5;

public class Exemplul3 {

   public static void main(String[] args) {
       String s1 = "A chain is only as strong as its weakest link";

       int length = s1.length();
       System.out.println("Length: " + length);

       char charAt = s1.charAt(2);
       System.out.println("The char value at the index 2:" + charAt);

       String[] strings = s1.split(" ");
       for (int i = 0; i < strings.length; i++) {
           System.out.println(strings[i]);
       }

       String substring = s1.substring(2, 7);
       System.out.println(substring);
   }
}
```

More info:
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

StringBuilder class:

The StringBuilder represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.

```
package com.paolabs.lab4.ex5;

public class Exemplul4 {

   public static void main(String[] args) {
       StringBuilder sb = new StringBuilder();
       sb.append("A journey of a thousand miles begins with a single
step");

       System.out.println(sb);
       sb.reverse();
       System.out.println(sb);
   }
}
```

Result:

```
A journey of a thousand miles begins with a single step
pets elgnis a htiw snigeb selim dnasuoht a fo yenruoj A
```

More info:
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuilder.html

StringBuffer class:

The StringBuffer represents a mutable sequence of characters. String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

```
package com.paolabs.lab4.ex5;
```

```
public class Exemplul5 {

   public static void main(String[] args) {
       StringBuffer stringBuffer = new StringBuffer();
       stringBuffer.append("A journey of a thousand miles begins with a
single step");

       System.out.println(stringBuffer);
       stringBuffer.replace(2,9, "walk");
       System.out.println(stringBuffer);
   }
}
```

Immutable classes:

- ➔ Once an object is created, we cannot change its content. All the wrapper classes (like Integer, Boolean, Byte, Short) and String class are immutable.
- ➔ We can create our own immutable class as well by following the rules:
    - ◆ Make your class final, so that no other classes can extend it.
    - ◆ Data members in the class must be declared as final (so that they're initialized only once inside the constructor and never modified afterward)
    - ◆ Define a parameterized constructor
    - ◆ Getter methods for all the variables in the class
    - ◆ Don't expose setter methods (to not have the option to change the value of the instance variable)
    - ◆ When exposing methods which modify the state of the class, you must always return a new instance of the class.
    - ◆ If the class holds a mutable object:
        - Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through the constructor, this is to prevent others who pass the object from modifying it afterwards.
        - Make sure to always return a clone copy of the field and never return the real object instance.

```
package com.paolabs.lab4.ex6;

public final class ImmutableStudent {

   private final int id;
```

```
    private final String name;

    public ImmutableStudent(int id, String name) {
        this.name = name;
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

```
package com.paolabs.lab4.ex7;

public class Age {

    private int day;
    private int month;
    private int year;

    public Age() {
    }

    public Age(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public int getDay() {
        return day;
    }

    public void setDay(int day) {
        this.day = day;
```

```
    }

    public int getMonth() {
        return month;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}
```

```
package com.paolabs.lab4.ex7;

public final class ImmutableStudent {

    private final int id;
    private final String name;
    private final Age age;

    public ImmutableStudent(int id, String name, Age age) {
        this.name = name;
        this.id = id;
        this.age = new Age(age.getDay(), age.getMonth(), age.getYear());
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
```

```
    public Age getAge() {
        return age;
    }
}
```