

Tutorial - Week 03 5COSC019W – Object Oriented Programming – Java

UML notation and objects relationships, static and not static context

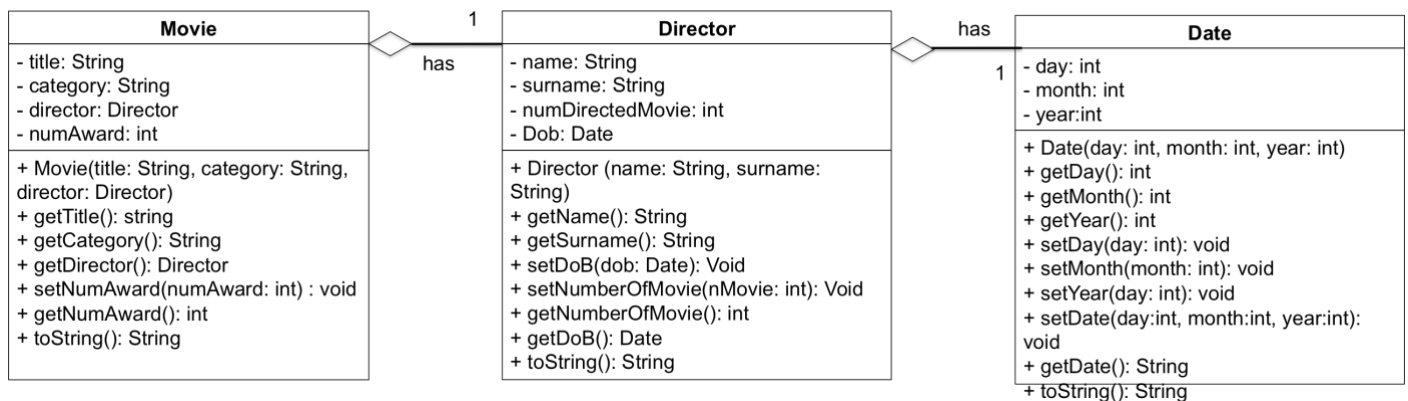
09/10/2023 – 13/10/2023

Association, aggregation and composition

In Object-oriented programming, one object can be related to other objects to use functionalities and services provided by those objects.

- **Association:** relationship between two objects and is depicted by an arrow in Unified Modelling language or UML.
- **Aggregation:** special form of association. It is a directional association, which means it is strictly a *one-way* association. It represents a *Has-A* relationship.
- **Composition:** stronger form of aggregation. One class owns other class and other class cannot meaningfully exist. It represents an *Owns-A* relationship.

- 1) Analyse the following UML diagram and explain the relationship between the classes. Which association there is between the classes and why?



Date Class

A class `Date` has been designed to model a date as shown in the class diagram. Implement the class `Date` as described in the UML class diagram.

This class contains:

- Three *private instance variables*. They represent day (int), month (int), and year (int).
- One *constructor* to initialize the date with values:

```
public Date (int day, int month, int year) { //write code here }
```
- Public *getters* and *setters*:

```
public void setDay(int day){ // write code here}
public void setMonth(int month){ // write code here}
public void setYear(int year){ // write code here}
public int getDay(){ //write code here}
public int getMonth(){ //write code here}
public int getYear(){ //write code here}
```
- A *toString()* method that returns the string `"Date[day = ?, month = ?, year = ?]"`

Example: "Date [day = 23, month = 11, year = 2020]"

If you don't remember the functionality of the "toString()" method go to revise it in the previous tutorial

Director Class

A class Director has been designed to model a movie's director as shown in the class diagram.

This class contains:

- Four *private instance variables*. They represent name (string), surname (string), the number of movies directed (int) and the date of birth (Date) of the director.

NOTE that one of the instance variable is an object (Date). You can see how the object Director and the object Date are linked by a relationship: a director object has a date of birth, which is represented by an object Date.

- One *constructor* to initialize the name and the surname of the Director with values:

```
public Director (String name, String Surname) { ... }
```

- Public *getters* and *setters*:

```
public String getName() { ... }
```

```
public String getSurname() { ... }
```

```
public Date getDoB() { ... }
```

```
public int getNumberOfMovie() { ... }
```

```
public void setDoB(Date date) { ... }
```

```
public void setNumberOfMovie (int num) { ... }
```

There are no setter methods for name and surname because we don't want to change these attributes.

- A *toString()* method that returns the string "Director[name = ?, surname = ?, dob = ?, movies directed = ?]"

Example: "Director [name = James, surname = Cameron, dob = 16/8/1954, movies directed = 23]"

- 2) Write the class Director base on the UML diagram. Note that, as mentioned earlier, the class Director uses a Date object to represent the date of birth of the director. You can use the implementation of the class Date that you wrote above.
- 3) Write a test class to test the public methods you implemented. Note that you should create an instance of Date before you can construct an instance of Director:

```
/*
 * A test class for the Director class.
 */
public class Test {
    public static void main(String[] args) {

        // Test constructor
        Director director = new Director("James", "Cameron");

        // Test Setters and Getters

        // Create an object Date to represent the dob
        Date dob = new Date(16, 8, 1954);
        director.setDoB(dob);
    }
}
```

```

        director.setNumberOfMovie(23);

        System.out.println(director); // toString()
        System.out.println("name is: " + director.getName());
        System.out.println("surname is: " + director.getSurname());
        System.out.println("dob is: " + director.getDoB().getDate());
        System.out.println("number of directed movies is: " +
            director.getNumberOfMovie());
    }
}

```

Movie Class

- 4) As you can see from the class diagram, a class called *Movie* is designed to model a movie directed by a *director*. This class contains:

- Four *private instance variables*. They represent the title of the movie (string), the category (string), the number of awards (int) and the director (Director).
- One *constructor* to initialize the title and the category, the name of the Director, with values:

```
public Movie (String title, String category, Director director) { ... }
```

- Public *getters* and *setters*:

```

public String getTitle() { ... }
public String getCategory() { ... }
public Director getDirector() { ... }
public void setNumAwards(int numAwards) { ... }
public int getNumAwards() { ... }

```

- A *toString()* method that returns the string "Movie[title = ?, category = ?, director name = ?, director surname = ?, number of awards = ?]"

Example: "Movie [title = Avatar, category = Fantasy, director name = James, director surname = Cameron, number of awards = 3]"

- 5) Write the class *Movie* base on the UML diagram. This class uses the *Director* class written earlier.
- 6) In the Test class you can test all the public methods in the class *Movie*. Note that you have to create an instance of *Director* before you can construct an instance of *Movie*.

```

/*
 * A test program for the Movie class.
 */
public class Test {
    public static void main(String[] args) {

        // We need a Director instance to create a Movie instance
        Director director = new Director("James", "Cameron");
        Date dob = new Date(16, 8, 1954);
        director.setDoB(dob);
        director.setNumberOfMovie(23);
    }
}

```

```

        System.out.println(director); // Director's toString()

        // Test Movie's constructor and toString()
        Movie movie = new Movie("Avatar", "Fantasy",director);
        System.out.println(movie); // Movie's toString()

        // Test Setters and Getters
        movie.setNumAwards(23);

        System.out.println(movie); // Book's toString()
        System.out.println("title is: " + movie.getTitle());
        System.out.println("category is: " + movie.getCategory ());
        System.out.println("name of director is: " + movie.getDirector().getName());
        System.out.println("surname of director is: " + movie.getDirector().getSurname());
        System.out.println("number of awards is " + movie.getNumAwards());
    }
}

```

7) Improve the class Date:

- Add the input validation when the user inserts the date. Consider that a day can be from 1 to 31, a month from 1 to 12 and a year from 1980 to 2079.
- Modify the method getDate() in order to return the date in the form gg/mm/yyyy. Add a leading zero when is needed.

Static variables and static methods

Static Methods can access class variables without using object of the class. A method may be declared with the *static* keyword. Static methods live at *class level*, not at *object level*. Static methods *may access* static variables and methods, but not dynamic ones.

Static variables are also known as *Class Variables*. The main characteristics of these variables are:

- Data stored in static variables is common for all the objects (or instances) of that Class.
- Memory allocation for such variables only happens once when the class is loaded in the memory.
- These variables can be accessed in any other class using class name.

8) Consider the following class:

```

public class IdentifyVariables {
    public static int x = 7;
    public int y = 3;
}

```

Which are the class variables?

Which are the instance variables?

Based on the class in 8), which is the output of the following code?

```

IdentifyVariables a = new IdentifyVariables ();
IdentifyVariables b = new IdentifyVariables ();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);

```

```
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyVariables.x = " + IdentifyVariables.x);
```

Some extra exercises

Extend the UML diagram in section 1), adding a class *Actor*. This class should provide instance variables, constructors and setter and getter methods. In the class *Movie*, add an instance variable that represents a list of actors playing in that movie. Since more than one actor plays a movie you should create a variable `actor:Actor[]`. This is an array of *Actor* objects.

You can declare and use an array of object in the following way:

```
Actor listActor[] = new Actor[10]; // it will create an array of 10 actors
listActor[0] = new Actor(); // it will create an object Actor and save in position 0
listActor[0].setName();
...
```

Write a Test class to test the new *Actor* class and the usage of the class within the *Movie* class.