Lab 2

[valid 2020-2021]

Starting from this week...:

- Comment and properly format the source code of your programs (otherwise: -0.5 points)
- Use the naming conventions for writing Java code. "Naming conventions make programs more understandable by making them easier to read." (otherwise: -0.5 points)
- Use the API documentation: Java Platform, Standard Edition 8 API Specification (otherwise...)

The Transportation Problem

An instance of the Transportation Problem consists of *source* and *destinations*.

- Each source has a given *capacity*, i.e. how many units of a *commodity* it is able to supply to the destinations.
- Each destination demands a certain amount of commodities.
- The cost of transporting a unit of commodity from each source to each destination is given by a *cost* matrix (or function).

We consider the problem of determining the quantities to be transported from sources to destinations, in order to minimize the total transportation cost. The supply and demand constraints must be satisfied. (We may assume that all the values are integer).

Consider the following example.

	D1	D2	D3	Supply
S1	2	3	1	**10**
S2	5	4	8	**35**
S3	5	6	8	**25**

Demand	**20**	**25**	**25**	

A solution may be something like that:

```
S1 -> D3: 10 units * cost 1 = 10
S2 -> D2: 25 units * cost 4 = 100
S2 -> D3: 10*8 = 80
S3 -> D1: 20*5 = 100
S3 -> D3: 5*8 = 40
Total cost: 330
```

The main specifications of the application are:

Compulsory (1p)

- Create an object-oriented model of the problem. You should have (at least) the following classes: *Source, Destination, Problem*.
  The sources and the destinations have *names*. The sources will also have the property *type*. The available types will be implemented as an *enum* . For example:
- `public enum SourceType {`
- `    WAREHOUSE, FACTORY;`
- `}`

  Assume S1 is a factory and S2, S3 are warehouses.

- Each class should have appropriate constructors, getters and setters.
  Use the IDE features for code generation, such as generating getters and setters.
- The *toString* method form the *Object* class must be properly overridden for all the classes.
  Use the IDE features for code generation, for example (in NetBeans) press *Alt+Ins* or invoke the context menu, select "Insert Code" and then "toString()" (or simply start typing "toString" and then press *Ctrl+Space*).
- Create and print on the screen the instance of the problem described in the example.

Optional (2p)

- Override the *equals* method form the *Object* class for the *Source*, *Destination* classes. The problem should not allow adding the same source or destination twice.
- Instead of using an *enum*, create dedicated classes for warehouses and factories. *Source* will become *abstract*.
- Create a class to describe the *solution*.
- Implement a simple algorithm for creating a feasible solution to the problem (one that satisfies the supply and demand constraints).
- Write doc comments in your source code and generate the class documentation using javadoc.

Bonus (2p)

- Implement an algorithm in order to minimize the total cost, using either:
  - an heuristic, for example the Vogel's approximation method
  - an exact algorithm based on minimum cost network flows.
  - your own idea (!?)
- Generate large random instances and analyze the performance of your algorithm (running times, memory consumption). Identify the *hot-spots* in your code.

(**Warning**: No points are awarded unless the implementation can be clearly explained).

Resources

- Slides
- Tutorial: Object-Oriented Programming Concepts
- Tutorial: Classes and Objects
- Java Language Specification: Classes
- Enum Types
- The Date-Time package

Objectives

- Create a project containing multiple classes.
- Instantiate classes and manipulate objects.
- Understand the concepts of: object identity, object state, encapsulation, property, accessors/mutators.
- Override methods of the *Object* class.

- Understand uni- and bi-directional relations among objects.
- Understand the notion of multiplicity (one-to-one, one-to-many, many-to-many).
- Implement instance-level relationships among objects (association).
- Implement class-level relationships (generalization)
- Work with abstract classes.
- Get used to the naming conventions of the Java language.
- Generate documentation using javadoc.