One dimension table, 2 dimension tables (ie matrices)

Emmanuelle Becker
Chapitre 1

Definition

When programming, **encapsulation** refers to the principle of grouping raw data with a set of routines for reading or manipulating them. This principle is often accompanied by information hiding of these raw data to ensure that the user does not bypass the interface that is intended for him.

1.1 Example

```java
public class Beverage {
    private String name /* the name of the beverage */
    private float sugar /* the sugar content of a 33cl glass */
    private float alcohol /* the alcohol degree of the beverage */

    /* other methods */
    /* now just imagine (this is not true) that the taxes are 20% when the alcohol is above 10 degrees or the sugar above 10 grams, and 10% otherwise. We would like a function that returns the tax that apply to the current beverage... */
    public computeTax(){
        if ((getAlcohol() > 20) || (getSugar > 10)) { return 20}
        else { return 10}
    }
    /* please note, and this is very important, that we used the accessors, and not the attributes themselves : this should become a reflex */
}
```
Chapitre 2

Exercices

2.1 Exercice 1

Le programme ci-dessous décrit un objet représentant les employés d’une entreprise. Dans l’entreprise, chaque employé est représenté par son nom, son prénom, et son échelon. Le salaire annuel de la personne est directement calculé à partir de cet échelon.

/* La classe des employés */
public class Employe {
    /* attributs */
    private String nom;
    private String prenom;
    private int echelon;

    /* constructeur */
    public Employe(String nomInitial, String prenomInitial, int echelonInitial) {
        nom = nomInitial;
        prenom = prenomInitial;
        echelon = echelonInitial;
    }

    /* méthodes */
    public void affiche() {
        System.out.println(nom + " " + prenom + " " + echelon + " (" + calculeSalaire() + ")");
    }

    public void donnePromotion() {
        echelon = echelon+1;
    }

    public int calculeSalaire() {
        return 100000+echelon*12000;
    }
}
La classe suivante n’est pas correctement encapsulée. En effet, on constate qu’au sein de la classe employé, il n’y a pas de méthode dédiées à la manipulation des attributs nom, prenom et echelon :

1. copiez-collez le code suivant dans un fichier Employes.java ;
2. ajoutez les accesseurs et mutateurs de ces trois attributs ;
3. dans le code de la classe Employes, remplacez tous les accès directs aux attributs par des accès via l’accesseur que vous avez implémenté.

Ajoutez ensuite un programme principal pour vérifier que tout fonctionne, en copiant-collant le code ci-dessous dans un fichier MainEmploye.java :

```java
public class MainEmployes {
    public static void main(String[] args) {
        Employe empl1 = new Employe("Hogg", "Boss", 1);
        Employe empl2 = new Employe("Coltrane", "Rosco", 1);

        empl1.affiche();
        empl2.affiche();
        empl1.donnePromotion();
        empl2.donnePromotion();
        empl1.affiche();
        empl2.affiche();
    }
}
```

### 2.2 Exercice 2

Le programme ci-dessous décrit un objet représentant un patient dans le fichier patients d’un ophtalmologue. Chaque patient est caractérisé par :

1. son nom
2. son prénom
3. son médecin traitant
4. son numéro de sécurité sociale
5. la date de son dernier RV ;
6. la date de son prochain RV.

Un médecin traitant est quant à lui représenté par un object Medecin dont les attributs sont les suivants :

- son nom ;
- son adresse ;
— son numéro de téléphone ;
— son numéro au registre de l’ordre.

Vous pouvez visualiser un squelette des classes ci-dessous.

```
truc
```
Chapitre 3

Short-project : Spread of a forest fire

The purpose of this exercise is to model the spread of a fire in a forest, depending in particular on the density of trees in the forest.

The forest will be modeled by a square matrix of dimension $n$ (we will impose that $n \geq 10$). Each cell of the matrix will take an integer value according to its state:

- value 0 for the "empty" cells (no tree);
- value 1 for the "tree" cells;
- value 5 for the "tree on fire" cells;
- value $-1$ for the "ash" cells (a tree after burning).

3.1 Creating the Automata and TestAutomata classes

Create a Automata class with two private fields:

- an array of integers named matrix, which will store our modeled forest;
- an integer named dimension that will store the dimension of the matrix.

We must now build a forest, i.e, fill our matrix with 0 and 1 so as to represent a specific forest. Add the following constructors (reminder, we impose that $n \geq 10$):

1. a first public constructor Automata(int n, double p), which creates a random forest of dimension $n$, and whose tree density is $p$ (which means that each cell has a probability $p$ of containing a tree$^1$;
2. a second public constructor Automata (double p), which creates a random forest of dimension $n = 10$, and whose tree density is $p$.

Add the following public methods:

1. a method called forestDisplay(), which returns nothing but displays the current matrix so that:
   - the "empty" cells are represented by ".",
   - trees by 'T',
   - trees on fire with 'O',

---

1. use the Math.Random() package
— and finally the ashes by 'l'.

2. a method isRazed(), which returns the false boolean if the forest has at least one tree, and true if there are no trees left in the forest.

3. a method isOnFire(), which returns the true boolean if one of the trees in the forest is on fire, and false otherwise.

3.2 Testing your basic Automata class

To test your Automata class, which already contains some interesting methods, create another class in the same package called TestAutomata, which contains a public static void main method. In this main method, create some automaton of the dimension you want, display them and test the few written methods.

For example:

```java
iAC=new Automata(0.3);
System.out.println("Is the forest on fire? "+iAC.isOnFire());
System.out.println("Is the forest completely razed? "+iAC.isRazed());
iAC.forestDisplay();

jAC=new Automata(25,0.8);
System.out.println("Is the forest on fire? "+jAC.isOnFire());
System.out.println("Is the forest completely razed? "+jAC.isRazed());
jAC.forestDisplay();

kAC=new Automata(20,0.7);
System.out.println("Is the forest on fire? "+kAC.isOnFire());
System.out.println("Is the forest completely razed? "+kAC.isRazed());
kAC.forestDisplay();
```

Add something to test that you took correctly into account the hypothesis that the dimension of the forest is at least 10.

3.3 Fire!

If we want to study the spread of a fire, we will have to fire a cell in our matrix. Create the following two methods (again an illustration of polymorphism):

1. a public method putFire(int i, int j) that modifies the matrix so that the cell i,j is on fire (which means that \(matrix[i][j] = 5\));

2. a public method putFire() that modifies the matrix to put a random cell on fire (even if there is no tree there).

You can now, from the TestAutomata class, simultaneously test your putFire() methods and your isOnFire() method... It's your turn to play!
3.4 Spreading of the fire

We propose the following rules to determine the future state of a cell according to its present state:

1. if the cell is "tree on fire" in its present state, then its next state will be "ash";
2. if the cell is "tree" in its present state and if one of its neighbors is currently in the "tree on fire" state, then its next state will be "tree on fire";
3. in other cases, the state of the cell does not change.

To make your code as explicit as possible, you can start by coding the following two private methods:

1. the private method isTree(int i, int j) which returns the true boolean if and only if matrix[i][j] currently contains a tree (i.e. a tree that is not on fire), and false otherwise;
2. the private method isOnFire(int i, int j) which returns the true boolean if and only if matrix[i][j] currently contains a tree on fire, and false otherwise (again an illustration of the polymorphism ...)

We arrive at the heart of the subject, with the following methods:

1. the private method hasNeighborOnFire(int i, int j), which returns the true boolean if one of the cells around the cell (i,j) is on fire, and false otherwise (please do not consider diagonals as neighbors, because it will complexify the function; so a cell has at most 4 neighbors);
2. the private method nextState(int i, int j), which returns an integer coding for the next state of the cell (i,j) according to its current state and the current state of its neighbors;
3. finally, the private method propagateFire1() which calculates the next matrix according to the current matrix. Warning: You can not modify the matrix as you calculate the next states (try to understand why)! You will have to declare a new matrix, allocate the space correctly, and fill this new matrix with the next states. Once all the next states are calculated, you can assign to this.matrix the new matrix.

3.5 Testing these methods

From the TestAutomata class, you can test your method propagateFire1() (it will be necessary to think of passing the method from private to public the time of the test), for example with the following lines of code:

```java
iAC=new Automata(15,0.8) ;
iAC.displayForest() ;
iAC.putFire(5,5) ;
iAC.displayForest() ;
iAC.propagateFire1() ;
iAC.displayForest() ;
```
3.6 Spreading over a long period

We know how to go from the current state to the next state. Let’s assume now that this is the evolution of our fire after an hour. We now want to calculate the evolution of the fire for several hours, or the evolution of the fire until the fire is extinguished ...

1. The public method `propagateFire(int n)`, which displays the evolution of the fire hour by hour for `n` hours. In order for the visualization to be easier, it can be interesting to put the program in "pause" between the different hours represented, what you can do with the following piece of code :

   ```java
   try {
       Thread.sleep (5000);
   } catch (InterruptedException ex) {
       Thread.currentThread () interrupt ().
   }
   ```

2. The public `propagateFire()` method, which displays the evolution of the fire hour by hour until the fire is extinguished.

3. Test these methods from the TestAutomata class, varying the scenarios.

3.7 The importance of tree density

Does the density of trees in the forest affect the spread of fire? This is what we will test now.

```java
for (double d=0.05; d<=1;d=d+0.05){
    int c=0 ;
    for (int j=0; j<100;j++){
        iAC=new Automata(25,d) ;
        iAC.putFire();
        iAC.propagateFire() ;
        if iAC.isRazed() { c=c + 1 ; }
    }
    System.out.println("With density "+d+" : "+c+" forest completely razed")
}
```

3.8 Bonus 1 : changing the spreading rule

Some modifications can be made to the current model. For example, think about how to take all neighbors into account when calculating the next state (neighbors diagonally included), or how to take wind into account.

3.9 Bonus 2 : the epidemy spreading

Strongly based on the previous exercise, imagine modeling a polio outbreak. You will place the individuals on a matrix, even if it is not very realistic. The cells will be populated :

— healthy unvaccinated individuals,
— vaccinated healthy individuals,
— individuals suffering from polio,
— recovering individuals,
— dead individuals
— nobody.

Initially, you will populate your matrix with a given density of individuals, of which a percentage $p$ will be vaccinated. Then you will contaminate with polio a random person.

You will calculate the propagation with the following rules:

— An individual in the current state of illness becomes either re-established or died in the next stage, with a probability $m$;

— Healthy unvaccinated individuals, one of whose neighbors is ill in the current state, become sick in the next stage.

— Recovered individuals can no longer be sick.

— Vaccinated individuals can not be sick.

— Empty cells remain empty.

Good luck!