Rappels Java*

1 Classes, instanciation, objets, main

Java est un langage pour la programmation orientée objet (POO). La programmation par objets permet de structurer les programmes non en termes de fichiers, de modules, de fonctions et de procédures, mais en termes de structures (les objets) qui approchent plus les choses du monde réel.

Exemple : vous voulez modéliser une voiture, vous allez définir une structure `Voiture`. La voiture a quatre roues, un volant, quatre portes, deux sièges avant, deux sièges arrière, etc. Vous allez définir, à l'intérieur de la structure `Voiture` d'autres structures pour les différentes parties. La voiture peut démarrer, s'arrêter, prendre de l'essence, avoir un accident : vous aller définir des actions pour manipuler la structure `Voiture`. Dans votre programme vous allez vouloir gérer différentes voitures (vous êtes vendeur de voitures), alors il va falloir pouvoir créer et manipuler différentes structures `Voiture`.

Pour faire ceci en Java, nous allons définir une classe `Voiture` qui va définir la structure générique d'une voiture. Dans cette classe, nous allons définir des *attributs* pour dire quelles sont les parties d'une voiture. Nous allons également définir des *méthodes* pour les différentes actions que nous pouvons faire avec une voiture. Ce qui nous donne :

```
class Voiture {

    //attributs
    Roue roueAvG, roueAvD, roueArG, roueArD;
    Volant volant;
    Siege sieges[4];
    String marque;
    int nbKilometres;
    int litresEssence;
    String etatVoiture;
```

*Cette documentation est basée sur un support de V. Danjean et V. Marangozova-Martin.

```
    ...
    //methodes
    void demarrer() {etatVoiture="demarree";}
    void arreter() {etatVoiture="arretee";}
    ...
}
```

Le programme principal (celui qui est exécuté lorsqu'on lance le programme Java), est contenu dans une méthode `main`, définie également dans une classe. Par exemple, si nous avons notre magasin de voitures, nous pouvons définir une classe `Magasin` comme suit :

```
class Magasin {

    static final int MAXSTOCK = 100;

    //attributs
    Voiture[MAXSTOCK] stock;
    int nbVoitures = 0;
    ...

    //methodes
    void recevoirVoiture(int kilometres,
                         int essence,
                         String marque) {

        Voiture nouvelle =
            new Voiture (kilometres, essence, marque);
        if (nbVoitures < MAXSTOCK )
            stock[nbVoitures]=nouvelle;
            nbVoitures++;
        } else {
            System.out.println("Reception impossible car stock complet.");
        }
    }

    void vendreVoiture(int numero) {
        for (int i = numero; i < nbVoitures-1; i++)
            stock[i]=stock[i+1];
        nbVoitures--;
    }
    ...

    //programme principal
```

```
    void main() {
        while (true) {
            System.out.println("Menu principal\n"+
                "1 : recevoir Voiture\n" +
                "2 : vendre Voiture\n"+
                ...);
            ...
            //creation de voiture
            if (choix==1) {
                ...
                recevoirVoiture(...);

            }
        }
    }
}
```

Dans cette classe `Magasin` nous voyons que la création de voitures se fait à l'aide d'une opération spéciale : `new`. Avec cette opération on appelle une méthode spéciale de la classe `Voiture` : son *constructeur*. Si on peut dire que la classe `Voiture` définit un type (quelque chose d'abstrait), quand on appelle son constructeur nous créons un objet concret qui peut être manipulé dans le programme. On dit que la classe est *instanciée*.

Le constructeur dans la classe `Voiture` est défini comme suit :

```
class Voiture {
    String marque;
    int nbKilometres;
    int litresEssence;
    ...
    Voiture (int k, int e, String m) {
        marque = m;
        nbKilometres = k;
        litresEssence=e;
    }
    ...
}
```

2 Compilation et exécution

Compilation Pour compiler notre programme Java, il faut exécuter la com-

mande suivante.

```
javac Voiture.java Magasin.java
```

Exécution Pour exécuter :

```
java Magasin //la classe qui contient le main
```

Définition de `classpath` : On peut compiler les classes Java séparément. Dans ce cas, il faut indiquer au compilateur où il peut trouver les classes référencées dans les classes à compiler. Pour cela, on utilise le `classpath`. Par exemple :

```
javac Voiture.java //crée Voiture.class ds répértoire courant
javac -classpath . Magasin.java
```

Répertoire destination pour classes compilées. Pour créer les classes non dans le répertoire courant mais dans un répertoire destination :

```
javac -d dest Voiture.java //crée Voiture.class dans dest
javac -classpath dest Magasin.java
//compile Magasin en utilisant la classe compilée dans dest
```

Remarques complémentaires concernant le `classpath` :

- La notion de `classpath` est utilisée par le compilateur `javac` (cf. ci-dessus) mais aussi par le programme `java` (machine virtuelle) lors du lancement d'un programme : le `classpath` indique une liste de chemins dans lesquels le code des classes nécessaires à l'exécution pourra être trouvé (code de la classe passé en paramètre à `java`, code des classes utilisées par cette dernière, etc.).
- Au lieu de passer le `classpath` en paramètre d'une ligne de commande `javac` ou `java`, on peut aussi le configurer via la variable d'environnement `CLASSPATH`, ce qui est souvent plus pratique.
  - Exemple de commande avec le shell tcsh :
    `setenv CLASSPATH chemin1:chemin2:chemin3`
  - Exemple de commande avec le shell bash :
    `export CLASSPATH=chemin1:chemin2:chemin3`

La compilation d'une classe Java ne génère pas de langage machine (compréhensible par le processeur de la machine sur laquelle on travaille) mais crée un format intermédiaire (*bytecode Java*) qui est interprété avant d'être exécuté sur le processeur. L'interprétation est faite par la *machine virtuelle Java*. L'avantage est que tout programme Java peut être exécuté sur toutes les machines où est installée la machine virtuelle (portabilité et indépendance du matériel). Toutefois, si un nouveau matériel doit être pris en compte, la machine virtuelle doit être réécrite.

3 Compléments

3.1 Héritage

Imaginons que le magasin ne vend pas uniquement des voitures, mais également des camions. Dans ce cas on voudrait avoir une entité générique `Vehicule` (on aura toujours quatre roues, une marque, avoir les mêmes actions). Par contre pour les voitures on voudrait spécifier PermisB, alors que pour les camions PermisC.

```
    class Vehicule {
        Roue ...
        Volant
        int nbKilometres;
        int litresEssence
        String marque

        void demarrer()...
        void arreter()
        ...
    }

    class Voiture extends Vehicule {
        String permis;

        Voiture(...) {
            super();
            permis="B";
        }
    }

    class Camion extends Vehicule {
        String permis;

        Voiture(...) {
        super();
        permis="C";
    }
}
```

La classe `Voiture` hérite de `Vehicule`, ce qui veut dire qu'elle dispose des attributs et des méthodes définis dans `Vehicule` (réutilisation de code).

3.2 Accès aux attributs et aux méthodes

3.2.1 Critères de visibilité

Specifier	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

3.2.2 Mot clé `static`

Pour utiliser des méthodes et des valeurs sans créer des objets, caractérise les classes (par opposition aux instances).

```
public class Calcul {

    //calcul de la moyenne de valeurs entières
    public static double moyenne(int[] valeurs) {
        int sum = 0;  // sommeéléments
        for (int i=0; i<p.length; i++) {
            sum += p[i];
        }
        return ((double)sum) / p.length;
    }
}

public class MonCalcul {
    public static void main() {
        int [] vals = {3,4,15,20};
        System.out.println(Calcul.moyenne(vals));
    }
}
```

3.2.3 Interfaces

Mêmes fonctions, différentes implémentations.

```
interface SommeProduit{
    int somme();
    int produit();
}
```

```
class Doublet implements SommeProduit{
    int a, b;
    ...
    public int somme(){return a + b;}
    public int produit(){return a * b;}
}
class Triplet implements SommeProduit{
    int a, b, c;
    ...
    public int somme(){return a + b +c;}
    public int produit(){return a * b *c;}
}
```

3.3 Comparaison de références et de valeurs

L'opérateur == sert à comparer des références vers des objets (les deux références pointent-elles vers la même instance d'objet ?). Si l'on souhaite comparer les valeurs de deux objets, il faut utiliser la méthode `equals`. Ainsi, si l'on souhaite par exemple vérifier que deux chaînes de caractères sont identiques, il faut utiliser la méthode `equals`.

Remarque : La méthode `equals` est définie par la classe `Object` et peut être redéfinie pour chaque classe. La plupart des classes fournies par la bibliothèque Java implémentent `equals` conformément à ce qui est indiqué ci-dessus (comparaisons de valeurs plutôt que de références d'objets). Cependant, pour les classes définies par l'utilisateur, l'implémentation par défaut de la méthode `equals` effectue une comparaison de références.