JAVA et les objets

Table des matières

La classe score

Création

Créons un premier objet ou une première classe qui représentera le score d'un joueur ou la note d'un élève. Cette classe, nommée score, devra gérer au moins trois données : le nom du joueur, son score, le score maximal autorisé.

Pour la créer, on utilise le fichier score.java :

```
public class score {
 String nom;
 int leScore;
 int scoMax;
}
```

Le fichier score.java étant créé, on le compile pour obtenir le fichier score.class.

Utilisation

Pour pouvoir utiliser la classe score, il faut créer une variable de type score dans un programme.

Créons donc un programme test1 (qui est aussi une classe, mais celle-ci a la particularité de posséder une méthode main).

```
public class test1 {
 public static void main(String args[]) {
  //déclaration
  score sc;
  //création ou instanciation
  sc=new score();
  //initialisation des données
  sc.nom="Pierre";
  sc.leScore=10;
  sc.scoMax=100;
  System.out.println(sc.nom+" : "+sc.leScore);
 }
}
```

Compilons et exécutons test1.

Méthodes

Comme nous serons souvent appelés à afficher le contenu de l'objet, écrivons une méthode affiche() qui se charge de ce travail. On l'ajoute dans le fichier score.java.

```
void affiche() {
 System.out.println(nom+" : "+leScore);
}
```

Après avoir recompilé score.java, on peut créer un nouveau fichier test2.java semblable à test1 mais en remplaçant la ligne System.out..... par sc.affiche(), puis tester le tout.

Un premier constructeur

Les constructeurs sont des méthodes particulières qui prennent le nom de la classe et qui permettent d'initialiser les objets. La classe score possède déjà un constructeur qui a été utilisé en écrivant

```
sc= new score();
```

Ce constructeur existe par défaut pour toute classe; il est possible de le réécrire, par exemple pour que le nom par défaut soit "inconnu" et pour que le score initial soit 0 et que le score maximal soit 100. On obtient :

```
score() {
 nom="inconnu";
 leScore=0;
 scoMax=100;
}
```

L'exécution de test1 ou de test2 ne donne pas de changement. Pour vérifier l'effet de ce nouveau constructeur, créons le fichier test3 à partir de test2 en supprimant la partie initialisation des données.

```
public class test3 {
 public static void main(String args[]) {
  //déclaration
  score sc;
  //création ou instanciation
  sc=new score();
  //affichage
  sc.affiche();
 }
}
```

On obtient cette fois "inconnu : 0" à l'affichage.

Un second constructeur

Pour ne pas créer que des inconnus, écrivons un second constructeur qui fixera le nom du joueur.

```
score(String nom) {
 this.nom=nom;
 leScore=0;
 scoMax=100;
}
```

Le mot-clé *this* représente la classe elle-même : cela permet de distinguer le champ nom de la classe et le paramètre nom passé au constructeur.

On peut alors modifier test3 pour créer l'objet score de Henri en écrivant :

```
sc=new score("Henri");
```

La gestion de plusieurs joueurs se fait de façon simple en créant plusieurs instances de la classe score. Par exemple, pour deux joueurs nommés Henri et Lise on pourra écrire le fichier test4.

```
public class test4 {
 public static void main(String args[]) {
  //déclaration
  score scHenri,scLise;
  //création ou instanciation
  scHenri=new score("Henri");
  scLise=new score("Lise");
  //affichage
  scHenri.affiche();
  scLise.affiche();
 }
}
```

Un problème de cohérence

Nous avons jusqu'ici présupposé que le score était toujours un nombre positif inférieur ou égal à scoMax. Cependant un programme peut attribuer n'importe quelle valeur au champ leScore. Pour empêcher cela nous allons déclarer ce champ avec le préfixe "private". Ce qui donne :

```
private int leScore;
```

Dans ces conditions, aucun programme ne peut lire ou modifier le contenu du champ leScore. Pour permettre d'agir sur lui en tenant compte de ses limites nous devrons ajouter deux nouvelles méthodes : l'une de lecture et l'autre d'écriture.

```
//méthode de lecture
public int getScore() {
 return leScore;
}

//méthode d'écriture
public void setScore(int sco) {
 if (sco>scoMax) leScore=scoMax;
 else if (sco<0) leScore=0;
 else leScore=sco;
}
```

Public ou privé

Les champs et les méthodes d'une classe peuvent être publics (mot-clé public) ou privés (mot-clé private). Comment choisir ?

1. Les classes et leurs constructeurs sont en général publics, leur raison d'être est justement d'être utilisés par d'autres.
2. Les champs contenant les données doivent en général être privés.
3. L'accès aux champs se fait par des méthodes de lecture (préfixées par get) et d'écriture (préfixées par set) qui doivent être publiques.
4. Pour les autres méthodes on décide au cas par cas.

Une dernière méthode

Terminons la construction de la classe score en lui attribuant une nouvelle méthode permettant d'ajouter des points au score.

```
public void ajoute(int points) {
 setScore(leScore+points);
}
```

Nous disposons désormais d'une classe permettant de gérer le score d'un joueur pour de nombreux jeux. Mais que se passe-t-il si pour une application particulière nous souhaitons doter la gestion du score de nouvelles possibilités ?

C'est ce que nous allons voir en créant une classe dérivée de la classe score.

Une classe dérivée de score

Classe dérivée

Pour les besoins d'un programme de jeu particulier, nous voudrions pouvoir gérer le score d'un joueur, mais aussi le nombre d'essais qui ont été effectués pour obtenir ce score. Nous disposons déjà d'une classe score pour la gestion du score proprement dit, mais celle-ci ne prend pas en compte le nombre d'essais. Pour éviter de refaire le travail effectué sur la classe score, nous allons créé une classe dérivée de la classe score : celle-ci aura d'une part tous les champs (non privés) et toutes les méthodes (non privées) de la classe score (c'est ce qu'on appelle l'héritage) et d'autre part de nouveaux champs et de nouvelles méthodes.

La déclaration de cette nouvelle classe score2 se fait en utilisant le mot-clé **extends** de la manière suivante :

```
public class score2 extends score {
 private int nbEssais;
}
```

La classe score2 hérite des champs et des méthodes de la classe score, certaines devront être complétées.

Surcharge des constructeurs

Les deux constructeurs de la classe score doivent être complétés pour initialiser le champ nbEssais.

```
public score2() {
 super();
 nbEssais=0;
}

public score2(String nom) {
 super(nom);
```

```
 nbEssais=0;
}
```

Le mot-clé **super** permet de faire référence à la classe parente, c'est à dire ici la classe score. Ainsi on commence par appeler le constructeur hérité, puis on initialise le champ nbEssais.

Surcharge de méthodes

Nous allons surcharger la méthode setScore de la classe score pour que chaque mise à jour du score s'accompagne d'une augmentation du nombre d'essais.

```
public void setScore(int sco) {
 super.setScore(sco);
 nbEssais++;
}
```

On utilise la même déclaration que dans la classe parente et on utilise la méthode héritée grâce au mot-clé **super**.

Ajout de méthodes

Ajoutons une méthode getNbEssais qui permet de lire le contenu du champ nbEssais qui a été déclaré private.

```
public int getNbEssais() {
 return nbEssais;
}
```

La nouvelle classe score2 peut maintenant être utilisée.

Exemple d'utilisation

Le programme suivant simule un jeu qui consiste à lancer des dés jusqu'au moment où on obtient un total de 21 points.

```
public class jeu21 {

 public static void main(String args[]) {
  Double D;
  int tirage;
  score2 pierre=new score2("Pierre");
  while (pierre.getScore()<21) {
   D=new Double(6*Math.random()+1);
   tirage=D.intValue();
   pierre.ajoute(tirage);
   pierre.affiche();
  }
  System.out.println("Pierre a gagné en "+
   pierre.getNbEssais()+" coups.");
 }
}
```

Copier, compiler et exécuter ce programme.

Conclusion

La possibilité de construire des classes dérivées de classes existantes est particulièrement importante : elle nous permet de récupérer les nombreuses classes définies dans le JDK et nous dispense ainsi de l'écriture d'une importante quantité de code.

Les classes du JDK

Le JDK fournit un grand nombre de classes regroupées en packages nommés java.*.

Le package java.lang

Le package java.lang est chargé automatiquement, ses classes sont donc toujours utilisables.

On y trouve, entre autres :

- la classe Object dont dérivent toutes les autres classes
- les classes représentant les types numériques de bases : Boolean, Byte, Double, Float, Integer, Long
- la classe Math qui fournit des méthodes de calcul des fonctions usuelles en mathématiques
- les classes Character, String et StringBuffer pour la gestion des caractères et des chaînes de caractères
- la classe System que nous utilisons pour afficher du texte sur la console DOS.

Les autres packages

Les autres packages doivent être déclarés (mot clé import) pour pouvoir être utilisés.

Nous aurons à étudier entre autres, les packages

- java.awt pour la création d'interfaces graphiques
- java.awt.event pour la gestion des évènements
- java.util pour certaines structures de données
- java.io pour la gestion des fichiers
- java.net pour le téléchargement de fichiers

Utiliser la documentation du JDK

Utiliser la documentation du JDK est indispensable pour connaître le contenu de ces classes. Effectuer les deux recherches suivantes :

1. trouver comment convertir une chaîne de caractères en variable de type double à l'aide de la classe Double

2. trouver dans la classe Math une méthode permettant de tirer des nombres au hasard; l'utiliser pour écrire un programme de pile ou face.

Résumé

Déclaration de classes

Déclaration simple

On écrit le mot-clé **class** suivi du nom de la classe puis d'un bloc entre accolades contenant les déclarations de champs et de méthodes.

```
class NomDeLaClasse {
 ...;
}
```

Toute classe hérite de façon implicite de la classe Object qui est donc à la racine de toute arborescence de classes.

Champs

Les données d'une classe sont contenues dans des champs qui sont des variables. La déclaration se fait en donnant le type du champ suivi de son nom.
Par exemple :

```
class ajout {
 double raison;
}
```

Méthodes

La déclaration se fait en donnant le type de retour suivi du nom et de parenthèses entourant les paramètres. Attention, les parenthèses doivent être présentes même s'il n'y a pas de paramètres.
La déclaration est suivie d'un bloc d'instructions entre accolades qui définit ce que fait la méthode. La valeur de retour est introduite par le mot-clé **return**. Celui-ci peut être omis si le type de retour est **void**.
Par exemple, si nous ajoutons la méthode image à la classe ajout :

```
class ajout {
 double raison;
 double image(double d) {
  return d+raison;
 }
}
```

Les paramètres sont toujours transmis par valeur pour les types de données de base. Par contre les instances passées en paramètres sont transmises par référence et sont donc susceptibles d'être modifiées si leur contrôle de visibilité le permet.

Constructeurs

Les constructeurs permettent d'initialiser les champs d'une classe. On les déclarent en utilisant le nom de la classe suivi de parenthèses entourant des paramètres.
Par exemple :

```
class ajout {
 double raison;
 ajout(double d) {
  raison=d;
 }
 double image(double d) {
  return d+raison;
 }
}
```

Héritage

Une classe peut être dérivée d'une classe préexistante qui est alors sa classe parente. La déclaration utilise alors le mot-clé **extends**. Par exemple :

```
class NomDeLaClasse extends ClasseParente {
 ...;
}
```

La classe dérivée hérite des champs et des méthodes visibles de la classe parente. Elle peut aussi surcharger certaines méthodes.

Contrôle de visibilité

Il existe 4 niveaux de protection des champs et des méthodes d'une classe.

1. **package**
   C'est le niveau par défaut (pas de déclaration à faire). La visibilité est limitée aux classes du package courant. Les champs sont accessibles en lecture et écriture.
2. **public**
   La déclaration est introduite par le mot-clé public. Il n'y a aucune restriction d'accès.
3. **protected** La déclaration est introduite par le mot-clé protected. Les champs et méthodes déclarés protected sont visibles par toutes classes du package courant et toutes les sous-classes qu'elles fassent partie ou non du package.
4. **private**
   La déclaration est introduite par le mot-clé private. Seule la classe peut voir les champs (encapsulation des données) et les méthodes déclarés private. En général on prévoit des méthodes publiques d'accès aux données.

Mots-clés final et static

Mot-clé static

Il permet d'introduire des *variables et des méthodes de classe* par opposition aux *variables et aux méthodes d'instance*.

Les variables et méthodes de classe peuvent être utilisées sans création d'une instance de classe.
La méthode main d'une classe est toujours static.

Mot-clé final

Il a l'effet suivant :

- pour une classe, il interdit d'en hériter
- pour une variable, il la rend constante
- pour une méthode, il interdit de la redéfinir dans une sous-classe

Note : les méthodes private sont déjà final.

Constantes

L'association des mots-clés static et final permet de définir des constantes de classe. Par exemple :

```
public static final int an=2000;
```

Création d'instances

Mot-clé new

La création d'une instance se fait au travers d'un **new** qui procède à l'allocation en mémoire et à l'appel d'un constructeur.
L'allocation en mémoire est dynamique et un garbage collector se charge de récupérer la mémoire allouée qui n'est plus référencée.

Quand on fait appel à un constructeur sans paramètres, une instance de base est créée avec des valeurs par défaut.
Quand le constructeur attend des arguments, ils déterminent les valeurs initiales des (tous ou certains) champs.

Accès aux champs et aux méthodes

On accède aux champs et aux méthodes en utilisant la notation pointée.

Pour les champs et les méthodes déclarés static on utilise le nom de la classe. Par exemple, avec la classe Math :

```
double mon_pi=Math.PI;
```

Dans les autres cas on utilise le nom d'une variable d'instance. Par exemple, en utilisant la classe ajout :

```
ajout a=new(12.5);
```

```
        double d=a.raison;
        double rep=a.image(5);
```

Mots-clés this et super

Le mot-clé **this** référence l'instance courante.

Le mot-clé **super** référence la classe parente de la classe courante. On l'utilise lorsqu'on surcharge des constructeurs ou des méthodes de la classe parente.