BPOO - Sujet TD 2

Dut/Info-S2/M2103

Table des matières

PreReq	Cours 1 : approche objet. S1.
ObjTD	**Utiliser des objets en Java**.
Durée	**1 séance** de 1,5h

1. Préliminaires et rappels

1.1. En-tête de sous-programme en Java

Un en-tête de sous-programme définit :

- l'identificateur du sous-programme,
- le nombre ET le type ET l'ordre des paramètres formels : une suite de couples `type identificateur` séparés par des virgules,
- un éventuel type de valeur retournée, `void` si rien n'est retourné.

  Notez que `void`, du point de vue du compilateur, ne veut pas dire "rien" mais est un type (il existe une classe Void) qui n'a aucune valeur et veut donc dire "retourne aucune valeur".

La **signature ou prototype** du sous-programme est ce qui permet de le reconnaître. Elle est formé de :

- l'identificateur du sous-programme,
- le nombre ET le type ET l'ordre des paramètres formels.

Exemple :

```
/**
* Calcul de la puissance puiss de la valeur v (calcul de v puissance puiss).
* Ne fonctionne que si puiss >= 0
*
* @param    pfV      valeur dont on cherche la puissance
* @param    pfPuiss  puissance à laquelle on va élever pfV
* @return   pfV élevé à la puissance pfPuiss
**/
double  puissance (double pfV, int pfPuiss) {
        double resultat ;
        ...
        return resultat;
}
```

1.2. Appel d'un sous-programme en Java

L'appel d'un sous-programme fait intervenir :

- l'identificateur du sous-programme,
- des paramètres effectifs en nombre ET ordre ET type correspondant au prototype du sous-programme.

Les paramètres effectifs peuvent être toute expression donnant une valeur du type attendu.

Lors de l'appel, d'éventuelles conversions de types, si elles sont nécessaires, peuvent intervenir. Les plus "classiques" sont les conversions des types de base : char → int → long → float → double.

Exercice Pour les appels suivants, expliquez si : i) l'appel est correct, ii) les calculs effectués pour réaliser l'appel. On considérera l'écriture des appels comme correcte même si nous reviendrons plus tard sur la syntaxe exacte d'appel.

1. Q : `double puissance (double v, int puiss)`

   ```
   double x;                    // Rappel : double  puissance (double pfV, int pfPuiss)
   x = puissance (10.5, 1+1);
   ```

   - Correct ?
   - Calculs ?
2. Q : `double puissance (double v, int puiss)`

```
double x;                    // Rappel : double  puissance (double pfV, int pfPuiss)
x = puissance (10.5, 2, 2+2);
x = puissance (10.5);
```

- Correct ?
- Calculs ?

3. Q : `double puissance (double v, int puiss)`

```
double x;                    // Rappel : double  puissance (double pfV, int pfPuiss)
x = puissance (10, 2);
```

- Correct ?
- Calculs ?

4. Q : `double puissance (double v, int puiss)`

```
double x;                    // Rappel : double  puissance (double pfV, int pfPuiss)
x = puissance (10.5, 2.5);
```

- Correct ?
- Calculs ?

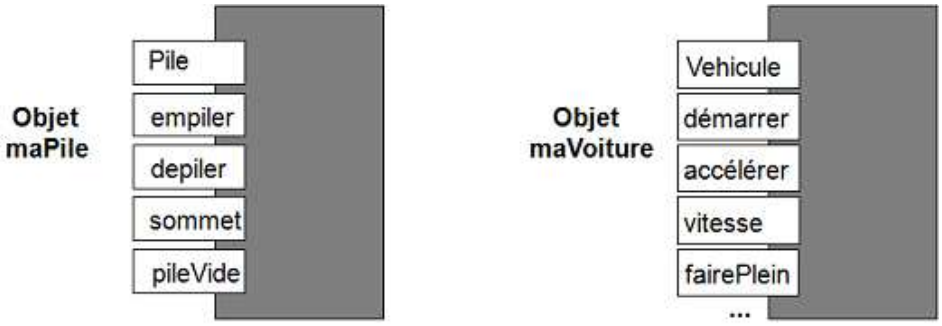
---

2. Utiliser des objets en Java

2.1. Rappel objets

Objet = identité + état + comportement

- Identité : Un objet forme un tout identifiable
- Un état : ensemble de propriétés/caractéristiques (attributs) ayant une valeur
- Comportement : ensemble des traitements (méthodes) que peut accomplir l'objet si on le lui « demande »



Par principe :

- Un objet **encapsule** son état
  - Etat : accessible que par l'objet lui-même (méthodes)
  - Principe objet majeur à respecter ⇒ **abstraction**
- Un objet reçoit des **messages** ⇒ appel de méthode appliqué à l'objet
- Un objet est créé à partir d'une classe qui fournit sa description.

Un objet reçoit des messages ⇒ appel de méthode appliqué à l'objet

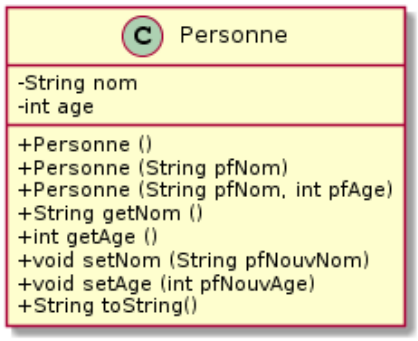
- Idée : c'est l'objet qui est "actif" et qui réagit (exécute la méthode)
- On peut "penser" objet comme cela …
- … mais la mise en oeuvre est différente.

2.2. Principe des classes en Java

Pour décrire une classe d'objet en Java, on définit une classe qui déclare :

- des attributs pour les objets (private),
- des méthodes pour les objets (public) + éventuellement des méthodes internes (private),
  - des méthodes observateurs (appelées aussi pour certaines getters),
  - des méthodes transformateurs (appelées aussi pour certaines setters),
- un/des constructeurs (public) pour construire des objets.

Exemple :



```
class Personne
{
                // Attributs
                private  String nom;
                private  int age;

        // Constructeurs
        public  Personne ()  { ... }
        public  Personne (String pfNom)   { ... }
        public  Personne (String pfNom, int pfAge) { ... }

        // Méthodes
        public  String getNom () { ... }
        public  int getAge ()   { ... }
        public  void setNom (String pfNouvNom) { ... }
        public  void setAge (int pfNouvAge)  { ... }
        public  String toString() { ... }
} // Fin Classe Personne
```

Principes de syntaxe :

1. Attributs : **toujours** déclarés private. Règle à appliquer pour construire des objets (et non des enregistrements).
2. Constructeur :
   - Porte le nom de la classe.
   - Pas de valeur de retour.
   - Plusieurs constructeurs possibles (surcharge) : différenciés par leur signature.

La classe définit un type :

1. On peut déclarer des variables de ce type.
2. Ces variables contiennent des **références** vers des objets de ce type (une référence est typée).

2.3. Création d'objet

Utiliser l'opérateur **new** : Exemple `new Personne ("Jean", 39);`

- réserve la mémoire nécessaire : environ le nombre d'octets pour stocker les attributs définis par la classe,
- initialise chaque attribut (`0` pour les nombres, `false` pour les booleans, `null` pour les objets et les tableaux),
- applique le constructeur spécifié (signature à partir des paramètres),
- **renvoie la référence** de l'objet créé dans le tas.

```
 new  Personne  ("Jean", 39)
|-------------| ..................> L'objet à créer -> avec les attributs définis par la
classe

 new  Personne  ("Jean", 39)
     |----------------------| ....> Signature du constructeur à appliquer -> Personne
(String, int)
```

Autre façon de créer des objets ? oui, méthode clone() appliquée à un objet … mais sera vu plus tard.

2.4. Identité d’objet en Java

L’identité d’objet en Java est automatique → c’est la référence de l’objet créée par **new**.

Elle est unique et permet de désigner l’objet indépendamment des valeurs des attributs.

Les variables objet contiennent la référence vers l’objet … donc son identité.

2.5. Envoi de message

Syntaxe : `expressionReferenceObjet . identificateurMethode ( [parametreEffectif1, …] )`

2.6. Exemple

```
class ClasseEssai
{
        public static  void main (String[]  argv) {
                int i;
                Personne  p1, p2, p3;
                Personne[]  tabPers;

                p1 = new Personne ("Jean", 39);
                p2 = new Personne ("Jean", 39);
                p3 = new Personne ("Pierre");

                p3.setAge (45);
                p3.setNom ("Durand");

                System.out.println (p1.toString());
                System.out.println (p1.getNom () );
                System.out.println (p2);   // Implicitement fera appel à p2.toString()
                System.out.println (p3);   // Idem

                if  (p1 != p2)
                        System.out.println ("Non Egalité des références");

                if (p1.getNom().equals(p2.getNom()))
                        System.out.println ("Egalité des noms des Personnes");

                tabPers = new Personne [3];
                tabPers[0] = p1;
                tabPers[1] = p2;
                tabPers[2] = new Personne ("Albert", 30);

                for (i=0; i<tabPers.length; i++) {
                        System.out.println (tabPers[i]);
                }
        }// Fin Main
}// Fin Classe ClasseEssai
```

3. Utilisation de classe

3.1. Exercice Pile

Soit la classe Pile suivante dont on ne donne que les constructeurs et les méthodes (sans les corps) :

```
class Pile {
        private String[] elements;
        private int indiceSommet;

        public Pile() { ... }
        public Pile(int pfTaille) {
                this.elements = new String [pfTaille];
                this.indiceSommet = -1
        }

        public boolean estVide() { ... }

        public void empiler(String pfElement) throws Exception { ... }

        public void depiler() throws Exception { ... }

        public String sommet() throws Exception { ... }
}
```

A faire :

1. Étudier le code de la classe Pile.
2. Ecrire (feuille jointe) un programme main() utilisant **obligatoirement des objets `Pile`** permettant : a) de saisir 10 chaînes de caractères, b) qui sépare d'un côté celles plus petites que "moto" et de l'autre celles plus grandes, et c) qui affiche ces deux "sous-listes" dans l'ordre inverse de leur saisie.

   Exemple : saisie de "a" "b" "n" "o" "p" "c" "m" "d" "e" "q" affichera :

   ⇒ e d m c b a

   ⇒ q p o n

NB : pour comparer deux chaînes, on dispose de la méthode suivante dans la classe `String` :

- `public int compareTo (String otherString)` : compare la chaîne avec `otherString` et renvoie `0` si elles sont égales, une valeur <0 si la chaîne est plus petite que `otherString`, une valeur >0 si la chaîne est plus grande que `otherString`.

3.2. Exercice Devinette

Nous allons utiliser une classe Devinette pour jouer, cf. ci-après.

Au jeu de la "devinette" :

- On suppose une valeur entière à trouver, tirée au sort par le programme, entre une borne basse (10<>20) et haute (40<>50).
- L'utilisateur cherche la valeur par essais successifs, la machine répondant à chaque coup si le nombre recherché est supérieur ou inférieur au nombre soumis.
- Le jeu s'arrête quand la valeur est trouvée.

Exemple d’exécution :

```
Coup : (13-49) : 33 // ❶
Plus Haut ... Coup : (13-49) : 40 // ❷
Plus Haut ... Coup : (13-49) : 49 // ❸
Plus Bas ... Coup : (13-49) : 48
Plus Bas ... Coup : (13-49) : 41
Plus Haut ... Coup : (13-49) : 47
Plus Bas ... Coup : (13-49) : 44
Plus Haut ... Coup : (13-49) : 46
Bravo, en 8 coups.
```

❶ Saisie utilisateur de 33

❷ Saisie utilisateur de 40

❸ …

A faire :

1. Étudier la documentation de la classe Devinette (constructeurs/méthodes).
2. Un objet Devinette, instance de la classe Devinette, est un objet qui :
   - lors de sa création, détermine des bornes min et max au jeu et choisit la valeur à chercher,
     - tirer au sort la borne supérieure (40<>50) et la borne inférieure (10<>20) du jeu.
     - tirer au sort la valeur à rechercher entre les bornes choisies ci-dessus.
 Exemple : borne inf 12, bone sup 47, valeur à chercher 22.
   - dispose de méthodes permettant de jouer et d’interroger l’état de la partie (nombre de coups joués, résultat d’un coup, …).
3. Écrire un programme main permettant de faire une partie de devinette en utilisant la classe Devinette : tirage au sort des valeurs, demander une valeur comprise entre les deux bornes et répondre à l’utilisateur sur la validité de son coup, afficher « partie gagnée » à la fin ainsi que le nombre de coups joués (essais).
4. Écrire un autre programme main permettant de jouer successivement 3 parties de devinette. Écrire d’abord un sous-programme `void jouer (Devinette d)` permettant de "dérouler" une partie de devinette avec l’objet d en paramètre. Puis écrire le programme demandé (quelques lignes).

3.3. Documentation classe Devinette

Retour sujet

Package (../classestps/package-summary.html)	**Class**	**Use** (class-use/Devinette.html)	**Tree** (package-tree.html)	**Deprecated** (../deprecated-list.html)	**Index** (../index-files/index-1.html)	**Help** (../help-doc.html)

classestps

Class Devinette

```
java.lang.Object
  ex classestps.Devinette
```

```
public class Devinette
extends java.lang.Object
```

Classe Devinette permettant de faire des jeux de "devinette de nombres".
A la création, un objet Devinette est configuré avec les 2 bornes du jeu (haute : 40<>50, basse : 10<>20) et la valeur à rechercher. Ces valeurs ne sont pas modifiables par la suite.
On peut soumettre un coup à une Devinette ("jouer") et tester ensuite si le dernier coup joué est gagnant, trop haut ou trop bas.
Lorsque la valeur à chercher a été trouvée, la partie est considérée "terminée" et on ne devra plus soumettre de coup. Les autres opérations de tests restent disponibles.

Exemple :

```
Devinette d;
d = new Devinette();
d.soumettreCoup(35);
if (d.isDernierCoupGagnant())
System.out.println ("Gagné");
```

Version:

2.02

Author:

André Péninou

Constructor Summary

Devinette (../classestps/Devinette.html#Devinette())()	Constructeur de Devinette qui initialise un objet Devinette prêt à être utilisé.

Method Summary

int	**getBas (../classestps/Devinette.html#getBas())()** Permet d'obtenir la borne basse du jeu tirée au sort à la création de l'objet.

int	**getHaut (../classestps/Devinette.html#getHaut())()** Permet d'obtenir la borne haute du jeu tirée au sort à la création de l'objet.
int	**getNbCoupsJoues (../classestps/Devinette.html#getNbCoupsJoues())()** Permet d'obtenir le nombre de coups déjà joués à une devinette.
boolean	**isDernierCoupGagnant (../classestps/Devinette.html#isDernierCoupGagnant())()** Permet de tester si le dernier coup joué est gagnant ou pas.
boolean	**isDernierCoupTropBas (../classestps/Devinette.html#isDernierCoupTropBas())()** Permet de tester si le dernier coup joué est trop bas.
boolean	**isDernierCoupTropHaut (../classestps/Devinette.html#isDernierCoupTropHaut())()** Permet de tester si le dernier coup joué est trop haut.
boolean	**isValeurDansBornes (../classestps/Devinette.html#isValeurDansBornes(int))(int val)** Permet de tester si la valeur val est située entre les bornes du jeu (incluses).
void	**soumettreCoup (../classestps/Devinette.html#soumettreCoup(int))(int valeurJouee)** Soumettre le coup valeurJouee ("jouer" une valeur) à une devinette.

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Devinette

```
public Devinette()
```

Constructeur de Devinette qui initialise un objet Devinette prêt à être utilisé. Permet de :

- Tirer au sort la borne supérieure (40<>50) et la borne inférieure (10<>20) du jeu.
- Tirer au sort la valeur à rechercher entre les bornes choisies ci-dessus.

Exemple : borne inf 12, bone sup 47, valeur à chercher 22.

Method Detail

soumettreCoup

```
public void soumettreCoup(int valeurJouee)
                   throws ErreurExecutionDevinette (../classestps
/ErreurExecutionDevinette.html)
```

Soumettre le coup valeurJouee ("jouer" une valeur) à une devinette.

Permet de soumettre un coup dans le cas où :

- Soit aucun coup n'a encore été joué (début de partie).
- Soit le dernier coup joué n'était pas gagnant (partie en cours, non finie).

ATTENTION : Lève une exception si le coup précédent était gagnant.
C'est dire que si **`isDernierCoupGagnant()`** vaut **`true`**, car la partie est considérée terminée.

Parameters:

`valeurJouee` - coup (valeur) que l'on souhaite soummettre à la devinette.

Throws:

`ErreurExecutionDevinette (../classestps/ErreurExecutionDevinette.html)` - lorsque on joue un coup ET que la partie était terminée (valeur trouvée au soumettreCoup() précédent)

See Also:

`isDernierCoupGagnant()` (../classestps/Devinette.html#isDernierCoupGagnant()),
`isDernierCoupTropHaut()` (../classestps/Devinette.html#isDernierCoupTropHaut()),
`isDernierCoupTropBas()` (../classestps/Devinette.html#isDernierCoupTropBas())

---

isValeurDansBornes

```
public boolean isValeurDansBornes(int val)
```

Permet de tester si la valeur val est située entre les bornes du jeu (incluses).

Parameters:

`val` - valeur que l'on souhaite tester.

Returns:

`true` ou **`false`** selon le cas.

---

isDernierCoupGagnant

```
public boolean isDernierCoupGagnant()
```

Permet de tester si le dernier coup joué est gagnant ou pas.
Permet de tester si le dernier coup joué par la méthode **`soumettreCoup`** a été gagnant ou pas.
Retourne **`false`** si aucun coup n'a encore été joué.

Returns:

`true` ou **`false`** selon le cas.

See Also:

`soumettreCoup(int)` (../classestps/Devinette.html#soumettreCoup(int))

---

isDernierCoupTropHaut

```
public boolean isDernierCoupTropHaut()
```

Permet de tester si le dernier coup joué est trop haut.

Permet de savoir si le dernier coup joué par la méthode **soumettreCoup** a été trop haut (bonne valeur plus petite).

Retourne **false** si aucun coup n'a encore été joué.

Returns:

true ou **false** selon le cas.

See Also:

soumettreCoup(int) (../classestps/Devinette.html#soumettreCoup(int))

---

isDernierCoupTropBas

```
public boolean isDernierCoupTropBas()
```

Permet de tester si le dernier coup joué est trop bas.

Permet de savoir si le dernier coup joué par la méthode **soumettreCoup** a été trop bas (bonne valeur plus grande).

Retourne **false** si aucun coup n'a encore été joué.

Returns:

true ou **false** selon le cas.

See Also:

soumettreCoup(int) (../classestps/Devinette.html#soumettreCoup(int))

---

getNbCoupsJoues

```
public int getNbCoupsJoues()
```

Permet d'obtenir le nombre de coups déjà joués à une devinette.

Returns:

nombre de coups joués.

---

getHaut

```
public int getHaut()
```

Permet d'obtenir la borne haute du jeu tirée au sort à la création de l'objet.

Returns:

borne haute du jeu.

---

getBas

```
public int getBas()
```

Permet d'obtenir la borne basse du jeu tirée au sort à la création de l'objet.

Returns:

borne basse du jeu.

Dernière mise à jour 2017-01-24 14:33:37 CET