INFO-H-200
Programmation orientée objet

Séance d'exercices 3
Packages, exceptions et tests unitaires

Université libre de Bruxelles
École polytechnique de Bruxelles

Professeur : Hugues Bersini

2017-2018

Packages

Un package Java est un ensemble de types (classes, interfaces, énumérations,...).

Par convention, le nom d'un package commence par une minuscule et est souvent un nom de domaine inversé (p.e : *be.ac.ulb.code*). Ce mécanisme est utilisé pour éviter les conflits de nommage, pour mieux contrôler les accès, pour simplifier le parcours et l'utilisation des classes plus aisé.

Par exemple, les classes fondamentales se trouvent dans *java.lang* qui est importé par défaut, les classes d'entrée/sortie se trouvent dans *java.io*,...

Une règle de bonne pratique veut que l'on rassemble dans un même package des classes liées.

A la première ligne d'un fichier, il faut préciser le package concerné, sinon il sera dans le package par défaut : **package nomDuPackage;**

Package (suite)

Pour utiliser des types d'un package dans un fichier d'un autre package :

- soit on importe tous les types d'un package au début du fichier :

```
import tp1.* ;
```

- soit on importe un type contenu dans un package au début du fichier :

```
import tp1.Date ;
```

- soit on utilise le nom complet dans le code :

```
tp1.Date date = new tp1.Date(12,11,09) ;
```

Encapsulation

L'encapsulation est un concept OO qui consiste à protéger l'information contenue dans un objet.

En java, les attributs et méthodes peuvent être :

- *public* : accessibles de partout (intérieur et extérieur de la classe)
- *private* : accessibles seulement à l'intérieur de la classe.
- *package-private* : (par défaut si on ne précise rien) : accessibles à l'intérieur de la classe et à l'intérieur d'un package.
- *protected* : accessibles à l'intérieur de la classe, du package mais aussi depuis l'extérieur pour les sous-classes. Ex : Si l'on déclare une variable comme 'protected' alors n'importe qui peut l'employer à condition d'étendre la classe concernée.

Une classe doit être le vigile de l'intégrité de ses objets. Pour ce faire, on va généralement déclarer tous ses attributs en *private* et gérer leur intégrité via des accesseurs. Attention, il ne faut faire des accesseurs que si c'est nécessaire !

Niveaux de protection d'une classes

Les niveaux de protection pour une classe sont :

- *public* : la classe peut être vue par tout le monde
- *private-package (si rien n'est précisé)* : la classe n'est vue qu'à l'intérieur de son package.

Il ne peut y avoir au maximum qu'une classe publique par fichier *.java*

Mot clé : ‘final’

- *variables:* Toutes les variables locales sont ’final’. Une fois une variable déclarée ’final’, vous ne pouvez plus la changer. Le compilateur verifiera cela et produira une erreur de compilation si nécessaire. On peut aussi dire qu’elles sont en ’lecture seule’.
- *méthodes:* Une méthode peut aussi être ’final’. Dans ce cas, elle ne peut être ré-écrite dans une sous-classe. Celles-ci sont par ailleurs plus rapide car elle ne nécessitent pas d’être résolue durant l’exécution, elles sont liées durant la compilation.
- *classes:* Finalement, une classe peut être ’final’. Cela implique qu’elle ne peut être sous-classés ou héritée.

Le mot clé *static*

Un attribut ou une méthode statique se réfère à une classe et non à une instance particulière.

Exemples :

- Pas besoin d'instancier un objet de type Math

```
Math.PI
```

- Pas besoin d'instancier un objet de type Math

```
Math.max(int i, int j)
```

- Méthode dans laquelle on va créer les premiers objets

```
static void main (...)
```

Une méthode statique dans une classe ne peut accéder qu'aux membres statiques de cette classe.

Méthode statique

```
public Date{
   ...
   private static int[] daysInMonths =  new int[]
       {31,28,31,30,31,30,31,31,30,31,30,31};

   private static int daysInMonth(int month, int
       year){
     int value = 0;
     if (isLeapYear(year) && month == 2){
       value = 0;
     }else{
       value = 1;
     }
     return daysInMonths[month-1] + value;
   }
}

...

int daysInFeb12 = Date.daysInMonth(2,2012);    //
    Appel sur la classe et non sur une instance
```

Objets et références

Les objets sont toujours utilisés par références.

Une instance = un *new*

```
Point originOne = new Point(23,94);
Rectangle rectOne = new Rectangle(originOne,100,200);
```

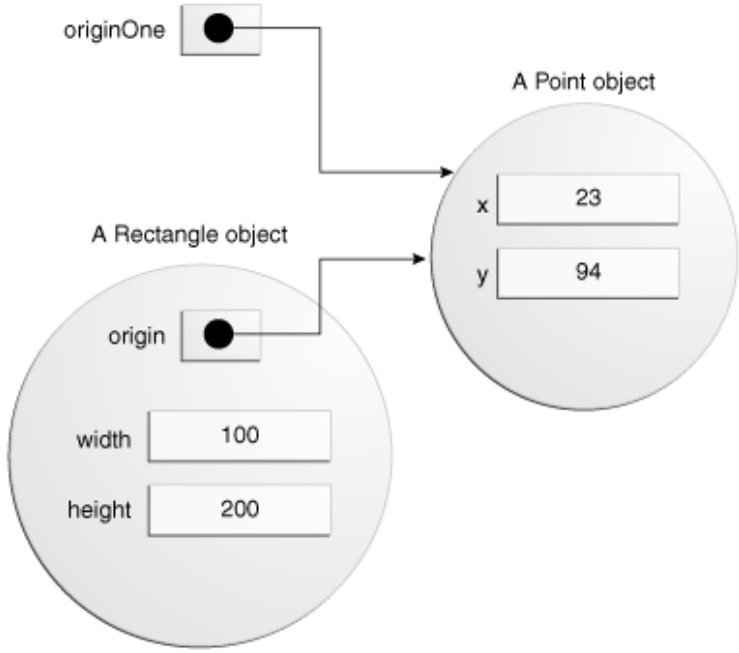

Figure: https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html

Les exceptions en Java

En Java, quand quelque chose se passe mal, une exception peut être déclenchée :

- Une condition d'exécution anormale a été détectée par la machine virtuelle (division par zéro, dépassement d'un tableau,...)
- Une exception a été lancée par une méthode via l'instruction *throw*
- Une exception asynchrone a été déclenchée (voir plus loin)

Une exception est un objet d'une classe ou d'une sous classe de la classe *Throwable*. Par exemple un objet de type *Exception*.

Atraper une exception

On peut attraper (*catch*) une exception en Java à l'aide de l'instruction *try catch*.

Exemple :

```
try{
   // code pouvant déclencer une exception
   // si une exception se déclenche,
   // on saute dans le bloc catch
} catch (Exception e) {
   // gestion de l'exception
   // par exemple :
   System.out.println(e.getMessage());
}
```

Atraper une exception (suite)

L'instruction *try catch finally* permet d'exécuter du code qu'une exception ait été déclenchée ou non.

Exemple :

```
try{
   // code pouvant déclencer une exception
   // si une exception se déclenche,
   // on saute dans le bloc catch
} catch (Exception e) {
   // gestion de l'exception
} finally {
   // code exécuté de quoiqu'il arrive
}
```

Checked / Unchecked Exceptions

Exception **unchecked** :

- Error (plutôt hardware) : Erreurs qui peuvent arriver à beaucoup d'endroits. Difficiles ou impossibles à réparer. (division par zéro, sortir d'un tableau,...)
- RuntimeException (plutôt software) : Exceptions qui ne devraient pas arriver comme un pointeur null mais qui pourraient arriver tellement souvent que les gérer serait fastidieux.

Exception **checked** : Toutes les autres exceptions

Déclencher une exception

L'instruction *throw* permet de déclencher une exception et sortir de la méthode courante.

Les méthodes qui peuvent déclencher une exception **checked** doivent le déclarer dans leur signature via le mot clé *throws*. Et donc le code qui appelle ces méthodes doivent attraper cette exception **checked**.

Exemple :

```
public void aMethod() throws Exception{
   ...

   if(somethingHappens()){
     throw new Exception("Un problème est survenu
         !");
   }
   ...
}
```

Les tests unitaires

Procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme. D'où le nom "test unitaire" pour désigner une 'unité' de votre programme.

Utilisation de la librairie JUnit

Libraires devant être importées :

```
import static org.junit.Assert.*;
import org.junit.Test;
```

Les tests unitaires : Exemple

Fichier ValueInt.java

```
public class ValueInt {
  private Integer value;

  public ValueInt(int value){
    this.value = value;
  }

  public int getValue(){
    return value;
  }

  public void increment(int inc) {
    value = value + inc;
  }
}
```

Fichier ValueIntTest.java

```
import static org.junit.Assert.*;
import org.junit.Test;

public class ValueIntTest {

  @Test
  public void testIncrement() {
    Integer initial = 0;
    Integer increment = 2;
    ValueInt value = new ValueInt(initial);
    value.increment(increment);
    assertEquals(initial+increment,value.getValue());
  }
}
```