AP3 – Programmation en Java
TD 03

Mickael Montassier et Arnaud Pêcher

6 septembre 2011

Résumé

Présentation des exceptions, de la sérialisation.

L'ensemble des fichiers de cette séance est disponible dans :
`/net/Bibliotheque/AP3/TD03/code/`

1 Cours

1.1 Contexte

Considérons la classe suivante décrivant une **Ville** :

```
public class Ville{
    private String nom;
    private int nbHabitants;

    public Ville(String nom){
        this.nom = nom;
    }
    public Ville(String nom,int nbHabitants){
        this(nom);
        setNbHabitants(nbHabitants);
    }
    public final void setNbHabitants(int nbHabitants){
        if(nbHabitants <= 0)
            erreur("nombre d'habitants doit être > 0");
        this.nbHabitants=nbHabitants;
    }
    protected void erreur(String message){
        System.out.println("ERREUR : classe : " + getClass().getName() +
                           " : " + message);
        System.exit(1);
    }
...
```

Et voici la classe de test :

```
public final class VilleApp{
    public static void main (String args []){
        Ville v = new Ville("Shanghai",-1);
    }
}
```

1. Que se passe t-il lors de l'exécution ?

Lors d'un appel à `setHabitants` avec un paramètre incorrect :
- la méthode `erreur` est appelée,
- un message décrivant l'erreur est affiché,
- le programme s'interrompt (brutalement).

Dans le cas présent, l'arrêt brutal n'est pas justifié ; nous aimerions refaire un appel avec un paramêtre correct et continuer l'exécution du programmme. **MAIS**.... nous n'avons pas envie de surcharger le code avec des instructions rendant les algorithmes illisibles. Vous avez certainement constaté la multiplication de clauses `if(...)...` en `C++` afin de gérer les cas limites qui peuvent intervenir dans vos programmes : division par zéro,...

Important : Le plus souvent, la méthode **constatant l'erreur** n'est pas habilitée à prendre la meilleure décision pour **la traiter** :
- je redemande une saisie ?
- je m'arrête ?
- j'affiche un message et je continue (vaille que vaille) ?

D'où le besoin d'un mécanisme de gestion des erreurs : besoin d'un moyen de **détection**, besoin d'un moyen de **gestion**, besoin d'un moyen de reprise.

Revenons à notre exemple. Ajoutons le code suivant :

```
public class NbHabException extends Exception{
    private int nbErr;

    public NbHabException(int nbErr){
        this.nbErr = nbErr;
    }
    public String toString(){
        return "Nb habitants erroné : " + nbErr ;
    }
}

public class Ville{
...
    public final void setNbHabitants (int nbHabitants) throws NbHabException{
        if(nbHabitants <= 0)
            throw new NbHabException(nbHabitants);
        this.nbHabitants=nbHabitants;
    }
```

Notez la création de la classe **NbHabException** spécialisant la classe **Exception** et la modification de la méthode `setNbHabitants`.

Et ça change quoi ?
- On détecte l'erreur.
- On prévient l'environnement.

La méthode ayant détectée l'erreur ne la gère pas : il y a **séparation** entre détection et gestion. Plus précisement que se passe t'il lors de l'exécution de la ligne suivante ?

```
...
            throw new NbHabException(nbHabitants);
...
```

i. L'argument est évalué.

ii. Un objet de la classe `NbHabException` est instancié.

iii. L'instruction `throw` **lève** une exception.

iv. La méthode courante (`setHabitants`) est stoppée.

v. Le gestionnaire d'exceptions prend la main : `setHabitants` a levé une exception. Il enlève de la pile d'exécution l'environnement associé à la méthode `setHabitants`.

vi. Il rend (éventuellement) la main à la méthode ayant appelé `setHabitants`. En fait, le gestionnaire d'exceptions va remonter la pile d'appels jusqu'à trouver une méthode capable de gérer l'exception.

```
public final void setNbHabitants (int nbHabitants) throws NbHabException
{
    if(nbHabitants <= 0)
        throw new NbHabException(nbHabitants);
...
```

La signature a un double objectif :
- documentaire : la méthode est susceptible de lever une exception du type `NbHabException`.
- elle précise au compilateur que toute méthode appelant `setHabitants` devra se préoccuper de cette exception : soit **en la traitant**, soit **en la propageant**.

2. Compiler et observer.

On observe une erreur lors de la création de l'instance :

```
public Ville(String nom,int nbHabitants) {
    this(nom);
    setNbHabitants(nbHabitants);
}
```

NbHabException must be caught or it must be declared in the throws clause of this method

ERREUR : Le constructeur `Ville` appelant `setHabitants` ne gère pas l'exception `NbHabException` susceptible d'être levée. Le constructeur doit soit **traiter** l'exception soit la **propager**.

Modifions l'entête du constructeur `Ville` afin de propager l'exception :

```
public Ville(String nom,int nbHabitants) throws NbHabException {
    this(nom);
    setNbHabitants(nbHabitants);
}
```

OK... nous propageons l'exception **mais qui la traite et comment ?**
Cela se fait via le bloc d'instruction `try... catch... finally` :

```
public final class VilleApp{
    public static void main (String args []){
        try { // Capture de l'exception
            Ville v = new Ville("Shanghai",-1);
        }
        catch(NbHabException e){
              // l'exception capturée, on la traite
            System.out.println(e);
        }
     /* finally{
              // bloc exécuté quoiqu'il advienne
              // permet de faire du nettoyage au besoin
            System.out.println("finally");
        } */
    }
}
```

Étape par étape :

i. appel du constructeur

ii. appel de `setNbHabitants(-1)`, qui lève une exception

iii. dépilement de l'environnement de `setNbHabitants` et propagation de l'exception au constructeur

iv. le constructeur ne capturant pas l'exception, son environnement est dépilé (aucun objet instancié) ; l'exception est propagée à la méthode appelante (ici `main`)

v. la clause `catch` de `main` capture l'exception et provoque l'exécution du bloc qui lui est associé.

1.2 Résumé

- Une exception est un objet qui est instancié lors d'un incident : **une exception est levée**.
- Le traitement du code de la méthode est interrompu et l'exception est **propagée** à travers la pile d'exécution de méthode appelée en méthode appelante.
- Si aucune méthode ne **capture** l'exception : celle-ci remonte l'ensemble de la pile d'exécution ; l'exécution se termine avec une indication d'erreur.
- La **capture** est effectuée avec les clauses **try... catch**.
- La clause **try** définit un bloc d'instructions pour lequel on souhaite capturer les exceptions éventuellement levées. Si plusieurs exceptions peuvent se produire, l'exécution du bloc est interrompue lorsque la première est levée. Le contrôle est alors passé à la clause **catch**.
- La clause **catch** définit l'exception à capturer, en référençant l'objet de cette exception par un paramètre puis le bloc à exécuter en cas de capture.

1.3 Pour aller plus loin

Toutes les exceptions héritent de la classe **Throwable**.

- `Throwable()`
- `Throwable(String)` : la chaîne passée en paramètre sert à décrire l'incident
- `getMessage()` : info
- `printStackTrace()` : exception + état de la pile d'exécution

Throwable possêde deux classes filles **Error** et **Exception** :

- Error : erreurs graves de la machine virtuelle (état instable, récursivité infinie, classe en chargement non trouvée,...).
- Exception : ensemble des erreurs pouvant être gérées par le programmeur (`RunTimeExeption` (`NullPointerException`, `IndexOutOfBoundsException`, `ArithmeticException`), `IOException`, ...).

Java n'oblige la déclaration explicite d'une levée d'exception que pour les exceptions dites contrôlées (en anglais **checked**) (`... throws NbHabException`).

- unchecked : RunTimeException,... (toutes les exceptions définies dans l'API), Error
- checked : les autres (celles du programmeur)

1.4 Exercice : ThrowTest

1. Écrire la classe `ThrowTest` : celle-ci contient une méthode `main`. La méthode `main` récupère l'argument passé en ligne de commande au moyen de :

   ```
   i=Integer.parseInt(argv[0])
   ```

2. La méthode `main` capture les exceptions eventuelles `ArrayIndexOutOfBoundsException` et `NumberFormatException`. Si une de ces exceptions est levée, alors affichez un message et terminez l'exécution.

Un peu d'aide ? :

```
public class ThrowTest {
    public static void main (String argv[]){
        int i;
        try{
            i = Integer.parseInt(argv[0]);
        }
        catch(ArrayIndexOutOfBoundsException e){// argv est vide
            System.out.println("Vous devez saisir un argument");
            return;
        }
        catch(NumberFormatException e){// l'argument n'est pas un entier
            System.out.println("L'argument spécifié doit être un entier");
            return;
        }
    }
}
```

3. Définissez trois exceptions :
   `MyException` étendant Exception
   `MyOtherException` étendant Exception
   `MySubException` étendant `MyException`

Par exemple :

```
class MyException extends Exception{
    public MyException(){
        super();
    }
    public MyException(String s){
        super(s);
    }
}
```

4. Dans la classe `ThrowTest`, ajouter les méthodes suivantes :
   - Méthode `c(int i)` :
     si l'entier i vaut 0, alors la méthode `c` lève une exception du type `MyException` ;
     si l'entier i vaut 1, alors la méthode `c` lève une exception du type `MySubException` ;
     si l'entier i vaut 2, alors la méthode `c` lève une exception du type `MyOtherException` ;
     pour toute autre valeur de i le message "input ok" est affiché.
   - Méthode `b(int i)` : appelle `c(i)` et ne capture que les exceptions de type `MyOtherException`.
   - Méthode `a(int i)` : appelle `b(i)` et capture toutes les exceptions restantes à capturer.
   - La méthode `main` appelle `a(i)`.

   Un peu d'aide ? :

```
public static void c (int i) throws MyException, MyOtherException{
    switch(i)
        {
        case 0:
            throw new MyException("input 0 -> MyException");
        case 1:
            throw new MySubException("input 1 -> MySubException");
        case 2:
            throw new MyOtherException("input 2 -> MyOtherException");
        default:
            System.out.println("input ok : " + i);
        }
}


public static void b (int i) throws MyException{
    try{
        c(i);
    }
    catch(MyOtherException e){
        System.out.println("MyOtherException");
        System.out.println(e.getMessage());
        System.out.println("handled at point 2");
    }
}

public static void a (int i){
    try{
        b(i);
    }
    catch(MyException e){
        if (e instanceof MySubException)
            System.out.println("MySubException");
        else
            System.out.println("MyException");
        System.out.println(e);
        System.out.println("handled at point 1");
    }
}
```

5. Écrire la trace du programme en fonction des valeurs de i.
6. Observer les mécanismes d'exceptions suivant les valeurs de i.

1.5 Sérialisation

Voici une exemple de sérialisation, où comment sauvegarder un objet dans un fichier binaire.

```
// People.java
import java.io.Serializable;
import java.util.Calendar;
import java.util.Date;

public class People implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String firstName;
    private final String name;
    private final Date birthday;

    public People(String firstName, String name, int year, int month, int day) {
        this.firstName = firstName;
        this.name = name;
        Calendar cal = Calendar.getInstance();
        cal.set(year, month, day);
        this.birthday = cal.getTime();
    }

    public final String firstName() {
        return firstName;
    }

    public final String name() {
        return name;
    }

    public final Date birthday() {
        return birthday;
    }

    public String toString() {
        return firstName + " " + name + " " + birthday.toString();
    }
}

// Test.java
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;


public class Test {

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        People dupont = new People("Jean", "Dupont", 1960, 11, 25);
        System.out.println(dupont);
        String dataFile = "essai";
        ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(new
```

```
                                        BufferedOutputStream(new FileOutputStream(dataFile)));
            out.writeObject(dupont);
        } finally {
            out.close();
        }

        People dupontBis;
        ObjectInputStream in = null;
        try {
            in = new ObjectInputStream(new
                                       BufferedInputStream(new FileInputStream(dataFile)));
            dupontBis = (People) in.readObject();
        } finally {
            in.close();
        }
        System.out.println(dupontBis);
    }
}
```

Pour plus de détails, voir :
`http://download.oracle.com/javase/tutorial/essential/io/objectstreams.html`

2 Applications

Voici une nouvelle description d'un labyrinthe et de Bob :

```
public interface Salle {
    // indique si cette salle est une sortie
    public boolean sortie();

    // salles voisines.
    public Collection<Salle> voisines();

    // informations de la salle
    public String toString();
}

public interface Labyrinthe {
    // salles composant le labyrinthe
    public Collection<Salle> salles();

    // renvoie la salle d'entree
    public Salle entree();

    // renvoie vrai ssi il existe un passage de s1 à s2
    public boolean passage(Salle s1, Salle s2);
}

public interface Personnage {
    // positionne un personnage dans la salle d'entree d'un labyrinthe.
    public void entrer(Labyrinthe l);

    // labyrinthe ou se trouve le personnage.
    public Labyrinthe labyrinthe();

    // salle ou se trouve le personnage.
    public Salle position();

    // deplace le personnage si il existe un passage entre sa position
    // et la nouvelle salle.
    public void aller(Salle s) throws DeplacementInterditException;

    // vrai si et seulement si la position du personnage est une sortie.
    public boolean estSorti();
}
```

Proposer un jeu de labyrinthe utilisant ces interfaces : faîtes progresser Bob jusqu'à la sortie !