Introduction à la programmation en JAVA

technofutur TIC

WAVENET
Intelligent IT Solutions

Table des matières

technofutur TIC

© Wavenet 2015

WAVENET
Intelligent IT Solutions

La gestion des exceptions

Aperçu du chapitre

I . La gestion des exceptions

Une **exception** est un **signal** qui se déclenche en cas de **problème**.

Les exceptions permettent de **gérer les cas d'erreur** et de rétablir une situation stable (ce qui veut dire, dans certains cas, quitter l'application proprement).

La gestion des exceptions se décompose en deux phases :

- la **levée** de l'exception
- le **traitement** de l'exception

En Java, une exception est représentée par une **classe**.

Toutes les exceptions dérivent de la classe **Exception** qui dérive de la classe **Throwable**.

I . La gestion des exceptions

En Java, il existe 3 sortes d'exception :

- Les exceptions vérifiées (Exception) : **exceptions prévisibles** dans le programme et traitées par le programmeur (panne réseau, absence d'un fichier, ...).
- Les exceptions runtime (RuntimeException) : **exceptions internes** au programme et qui résultent de **bugs** de programmation.
- Les erreurs (Error) : **exceptions externes** au programme et impossible à prévoir que le programmeur n'est pas censé traiter.

Aperçu du chapitre

II . Lever une exception : throw – throws

Une exception est **levée** grâce à l'instruction **throw**.

```
if (k < 0) throw new MonException("Message");
```

Une exception peut être traitée directement par la méthode dans laquelle elle est levée, mais elle peut également être **envoyée à la méthode appelante** grâce à l'instruction **throws**.

```
public void maMethode(int entier) throws MonException {
    //code de la methode
}
```

Dans cet exemple, si une exception de type *MonException* est levée durant l'exécution de *maMethode*, l'exception sera envoyée à la méthode appellant *maMethode*, qui devra la traiter.

Aperçu du chapitre

III . Traiter une exception : le bloc try – catch – finally

Le **traitement** des exceptions se fait à l'aide de la séquence d'instructions **try**...**catch**...**finally**

- L'instruction **try** indique qu'une instruction (ou plus généralement un bloc d'instructions) susceptible de lever une (des) exception(s) débute
- L'instruction **catch** indique le traitement pour un type particulier d'exceptions
- L'instruction **finally**, qui est optionnelle, sert à définir un bloc de code à exécuter dans tous les cas, exception levée ou non

III . Traiter une exception : le bloc try – catch – finally

```
public void ajouterPersonne(Personne p) {
        try {
            for (int i = 0 ; i < nbPersonnes ; i++) {
                if (p.equals(tab[i])) {
                        throw new PersonneDejaExistanteException(p);
                }
            }

            tab[nbPersonnes] = p;
            nbPersonnes++;
        }
        catch (PersonneDejaExistanteException ex) {
                System.err.println(ex.getMessage());
        }
        finally {
                System.out.println("Fin de méthode");
        }
}
```


III . Traiter une exception : le bloc try – catch – finally

Le bloc **catch** (PersonneDejaExistanteException ex) capture toute exception du type PersonneDejaExistanteException et crée une instance de cette classe: ex.

La méthode **getMessage**() (appelée sur ex) permet de récupérer le message d'erreur lié à l'exception. Celui-ci est défini dans le constructeur de PersonneDejaExistanteException.

Le bloc **finally** est exécuté quoi qui se passe (exception ou non).

Aperçu du chapitre

IV . La hiérarchie des exceptions

L'**ordre** des blocs **catch** est important : il faut placer les **sous-classes avant leur super-classe**. Dans le cas contraire, le compilateur génère l'erreur exception:

classe_exception has already been caught

```
try {
        FileReader lecteur = new FileReader(nomDeFichier);
}
 // Capture IOException et ses sous-classes
catch (IOException ioex) {
        System.err.println("IOException catched");
        ioex.printStackTrace();
}
// Erreur car FileNotFoundException hérite de IOException, déjà capturée par le premier bloc catch
catch (FileNotFoundException fnfe) {
        System.err.println("FileNotFoundException catched");
        fnfe.printStackTrace();
}
```

Aperçu du chapitre

V . Relancer une exception

Relancer une exception consiste simplement à utiliser l'instruction throw avec l'objet exception que l'on a capturé

```
public void ajouterPersonne(Personne p) throws PersonneDejaExistanteException {
        try {
            for (int i = 0 ; i < nbPersonnes ; i++) {
                if (p.equals(tab[i])) {
                        throw new PersonneDejaExistanteException(p);
                }
            }
            tab[nbPersonnes] = p;
            nbPersonnes++;
        }
        catch (PersonneDejaExistanteException ex) {
                // Traitement partielle de l'exception
                …
                // On relance l'exception (le traitement sera terminé ailleurs)
                throw ex;
        }
}
```

Aperçu du chapitre

VI . Créer une classe d'exception

Il est également possible d'étendre une classe d'exception pour **spécialiser un type d'erreur**, ajouter une information dans l'objet exception, ...

```
public class PersonneDejaExistanteException extends Exception {

        public PersonneDejaExistanteException(Personne p) {
                super("La personne " + p + " existe déjà.");
        }
}
```

La chaîne de caractères passée en argument lors de l'appel du constructeur parent sert à définir le message d'erreur à afficher.

VI . Créer une classe d'exception

Une instance de cette classe peut ensuite être lancée de la manière suivante :

```
public void ajouterPersonne(Personne p) throws PersonneDejaExistanteException {
        ...
        throw new PersonneDejaExistanteException(p);
}
```

et capturée comme suit :

```
try {
        registre.ajouterPersonne(p4);
 }
catch(PersonneDejaExistanteException e) {
        System.err.println(e.getMessage());
}
```


Aperçu du chapitre

technofutur TIC
© Wavenet 2015
WAVENET Intelligent IT Solutions

VII . Exercices

1. Vous demandez à un utilisateur d'encoder la longueur du côté d'un carré et vous affichez l'aire de ce carré. Vous demandez à l'utilisateur d'encoder une valeur jusqu'à ce qu'il fournisse une longueur supérieur à 0.

2. Vous demandez à un utilisateur d'encoder un nombre réel. Une exception risque de se produire si l'information encodée par l'utilisateur n'est pas un nombre réel. Vous demandez à l'utilisateur d'encoder une valeur jusqu'à ce qu'il fournisse un nombre réel.

VII . Exercices

3. Toutou est une classe avec deux propriétés privées (nom et nombrePuces). Ecrire un constructeur Toutou(String n, int np) qui propage des exceptions lorsque le nom est null ou lorsque le nombre de puces np est négatif.

4. Créez une classe Ville. Une ville est caractérisée par un nom et un nombre d'habitants.

Vous allez prévoir deux constructeurs :

- Le premier pour lequel on ne donne que le nom (nombre d'habitants = 1000)
- Le second pour lequel on donne le nom et le nombre d'habitants

Il faut prévoir deux types d'exception :

- Un nombre d'habitants négatif
- Un nombre d'habitants nul (= 0)

Dans les deux cas, il faut afficher un message d'exception adapté.

VII . Exercices

5. Créez une classe Etudiant.

Un étudiant sera caractérisé par un nom, un prénom et un tableau reprenant la liste des cours qu'il suit (l'étudiant peut suivre un maximum de 10 cours).
Vous devez prévoir un constructeur garnissant le nom et le prénom (le nombre de cours sera nul).
Il faut prévoir une méthode ajouterCours ajoutant un cours dans la liste des cours d'un étudiant.
L'ajout d'un doublon (cours déjà présent dans la liste) ainsi que la tentative d'ajouter un 11e cours doivent être gérés par des exceptions.
Créez également une méthode getCours qui retourne le libellé du cours lorsqu'on donne son numéro. Vous devez gérer pour cette méthodes 3 types d'exceptions : indice négatif, indice dépassant le nombre de cours suivis par l'étudiant, indice dépassant la taille maximale de 10 cours.
Un cours est caractérisé par un libellé, l'année dans laquelle il est donné et un nombre d'heures données. Deux cours sont les mêmes s'ils possèdent le même libellé et la même année dans laquelle ils sont donnés.

VII . Exercices

6. Construisez une classe abstraite TabTrie qui correspond à un tableau trié d'objets. Cette classe doit notamment contenir :
   - un tableau d'Object, tab, initialisé avec une capacité définie
   - une variable contenant l'indice du premier élément vide du tableau (tous les éléments vides doivent être à la fin!)
   - une méthode plusGrand qui compare deux Object et renvoie true si le premier est plus grand que le deuxième
   - une méthode ajouter qui insère un Object dans le tableau en respectant l'ordre croissant
   - une méthode toString qui renvoie une chaîne de caractères représentant le tableau

Lorsque la capacité du tableau est atteinte, l'insertion d'un nouveau élément lancera une exception TabPlein.
Construisez la classe TabTriPoint qui hérite de TabTri et ordonne des objets de type Point (toute comparaison se fait d'abord sur X, puis sur Y).

VII . Exercices

7. Un Polygone est composé d'un tableau de Point représentant ses sommets et d'une variable contenant l'indice du premier élément vide de ce tableau. Le constructeur reçoit comme argument le nombre de sommets qui définit le Polygone. Il doit être possible d'ajouter un nouveau sommet au Polygone en envoyant un Point ou en envoyant des coordonnées X et Y (attention aux exceptions!). Enfin, prévoir des méthodes pour afficher l'état du Polygone et de faire subir une translation.

La classe Rectangle hérite de Polygone et possède une surface. Son constructeur reçoit en arguments deux sommets opposés. C'est lui qui s'occupe d'ajouter les quatre points dans le tableau des sommets et on doit pouvoir l'appeler sans devoir utiliser de try-catch.
La classe Carre hérite de Rectangle. Son constructeur recoit un sommet et la longeur d'un côté.