Les exceptions

Christina Boura, Stéphane Lopes

christina.boura@prism.uvsq.fr,
stephane.lopes@prism.uvsq.fr

9 mars 2015

Qu'est-ce que c'est une exception ?

Une exception est un événement qui se produit lors de l'execution d'un programme et qui interrompe la succession normale des instructions.

- Pourquoi dit-on "exception" ?

Il s'agit des problèmes qui se produisent rarement. Ce sont donc *l'exception à la règle*.

Une erreur se produit pendant l'exécution

- L'erreur se produit normalement dans une méthode qui a été appelée par une autre méthode et ainsi de suite jusqu'à la méthode initiale `main`.
- Quand l'erreur se produit, l'exécution du programme s'arrête.
- Java crée un objet de type `Exception` qui contient des informations relatives à l'erreur.

On dit qu'une exception est levée.

- L'exception remonte la séquence des appels de méthodes (se propage) jusqu'à la méthode main initiale.
- Un message d'erreur significatif s'affiche sur l'écran en prenant en compte les informations collectées par l'objet `Exception` pendant sa propagation.

Diviser par zéro

```java
import java.util.Scanner;

public class DiviserParZero {

    public static int quotient(int numerateur, int denominateur)
    {
        return numerateur/denominateur;

    } // fin method quotient

    public static void main(String[] args) {

    int numerateur, denominateur, resultat;

    Scanner scanner = new Scanner(System.in);

    System.out.println("Entrez un numérateur: " );
    numerateur = scanner.nextInt();
    System.out.println("Entrez un dénominateur: " );
    denominateur = scanner.nextInt();

    resultat = quotient(numerateur, denominateur);
    System.out.println("Le résultat est :" + resultat);

    scanner.close();
    } //fin main

} //fin class DiviserParZero
```

Lors de l'exécution

```
Entrez un numérateur:
8
Entrez un dénominateur:
4
Le résultat est :2
```

Une exception est levée

```
Entrez un numérateur:
3
Entrez un dénominateur:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DiviserParZero.quotient(DiviserParZero.java:7)
        at DiviserParZero.main(DiviserParZero.java:22)
```

- L'utilisateur saisit 0 comme dénominateur.
- La division par 0 sur les entiers est interdite, donc une exception est levée.
- Type de l'exception : `ArithmeticException`.

Stack trace de l'exception

```
Entrez un numérateur:
3
Entrez un dénominateur:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DiviserParZero.quotient(DiviserParZero.java:7)
        at DiviserParZero.main(DiviserParZero.java:22)
```

La *stack trace* représente la pile d'appel du programme au moment où l'exception a été générée. Elle contient :

1. Le type de l'exception ainsi qu'un message descriptif de l'erreur.
2. Une liste de noms de classes et des méthodes accompagnés d'un numéro de ligne, indiquant à quel endroit du programme l'erreur a été produite.

InputMismatchException

```
Entrez un numérateur:
8
Entrez un dénominateur:
quatre
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at DiviserParZero.main(DiviserParZero.java:20)
```

- Type de l'exception : `InputMismatchException`
  (L'utilisateur a saisi une entrée qui n'est pas une valeur numérique.)

Autres types d'exceptions communes

Exceptions habituelles en Java :

- `NullPointerException`
  - Accès à un champ ou appel d'une méthode non statique sur un objet `null`.
  - Utilisation de `length` ou accès à une case d'un tableau `null`.
- `ArrayIndexOutOfBoundsException`
  - Accès à une case inexistante d'un tableau.
  - Accès au k-ième caractère d'une chaîne de caractères de taille inférieure à k.
  - Essai de création d'un tableau de taille négative.

Gérer une exception : Un petit exemple

Une classe `RacineCarree` qui calcule la racine carrée d'un nombre réel passé en paramètre :

```
public class RacineCarre() {

    public static double racineCarree(double entier) {
        return Math.sqrt(entier);
    }
}
```

Cette méthode fonctionne bien sauf si l'utilisateur fournisse un entier négatif.

Gérer une exception : Première approche

Afficher un message et renvoyer la racine carrée de l'opposé de l'entier fourni (car il faut bien renvoyer quelque chose) :

```
public class RacineCarre() {

    public static double racineCarree(double entier) {
        if (entier < 0) {
            System.out.println("Pas possible de calculer la racine d'un nombre négatif.");
            System.out.println("Calcul de racineCarree("+(-entier)+") a la place");
            entier = (-1)*entier;
        }
        return Math.sqrt(entier);
    }
}
```

Critique : L'utilisateur ne se rend pas compte facilement que quelque chose d'anormal vient de se produire.

Gérer une exception : Deuxième approche

Définir et renvoyer un `code d'erreur` pour signaler le problème.

```
public class RacineCarre() {

    public static double racineCarree(double entier) {
        if (entier < 0)
            return -1;
        return Math.sqrt(entier);
    }
}
```

Avantage par rapport à la première solution : L'utilisateur a maintenant un moyen de savoir si la méthode s'est bien déroulée ou pas.

Une première difficulté

Envoyer un code d'erreur n'est pas toujours évident.

La méthode `quotient` renvoie le quotient de deux nombres réels :

```
public static double quotient(double numerateur, double denominateur) {
    return numerateur/denominateur;
}
```

Problème : Comment faire la différence entre un "-1" signalant un problème lors d'une division par 0 d'un "-1" résultant de la division de -2 par 2 ?

Un autre exemple : Lire un fichier (I)

Méthode qui ouvre un fichier et le charge dans la mémoire.

Opérations :

- Ouvrir le fichier.
- Déterminer sa taille.
- Allouer de la mémoire.
- Lire le fichier.
- Fermer le fichier.

Un autre exemple : Lire un fichier (II)

Différents problèmes peuvent apparaître :

- Le fichier ne peut pas s'ouvrir.
- La longueur du fichier ne peut pas être déterminée.
- Il n'y a pas assez de mémoire disponible.
- La lecture échoue.
- Le fichier ne peut pas se fermer.

Comment procéder ?

Une solution avec des codes d'erreurs

```
errorCodeType lireFichier {
    initialize codeErreur = 0;

    open the file;
    if (fichierEstOuver) {
        determiner la longueur du fichier;
        if (longueurDeterminee) {
            allouer tant de memoire;
            if (memoireSuffisante) {
                lire le fichier;
                if (lectureEchouee) {
                    codeErreur = -1;
                }
            } else {
                codeErreur = -2;
            }
        } else {
            codeErreur = -3;
        }
        fermer le fichier;
        if (FichierPasFermee && codeErreur == 0) {
            errorCode = -4;
        } else {
            codeErreur += -4;
        }
    } else {
        codeErreur = -5;
    }
    return codeErreur;
}
```

Critique de cette approche

- Le programme peut rapidement devenir illisible. Les lignes essentielles du code sont perdus parmi tous ces tests.
- Le programme devient difficile à maintenir et à déboger.
- Difficile de se rendre compte si le programme fait toujours ce qui lui a été démandé.

Traiter des erreurs de façon systématique

`Java` fournit un mécanisme permettant de gérer les erreurs.

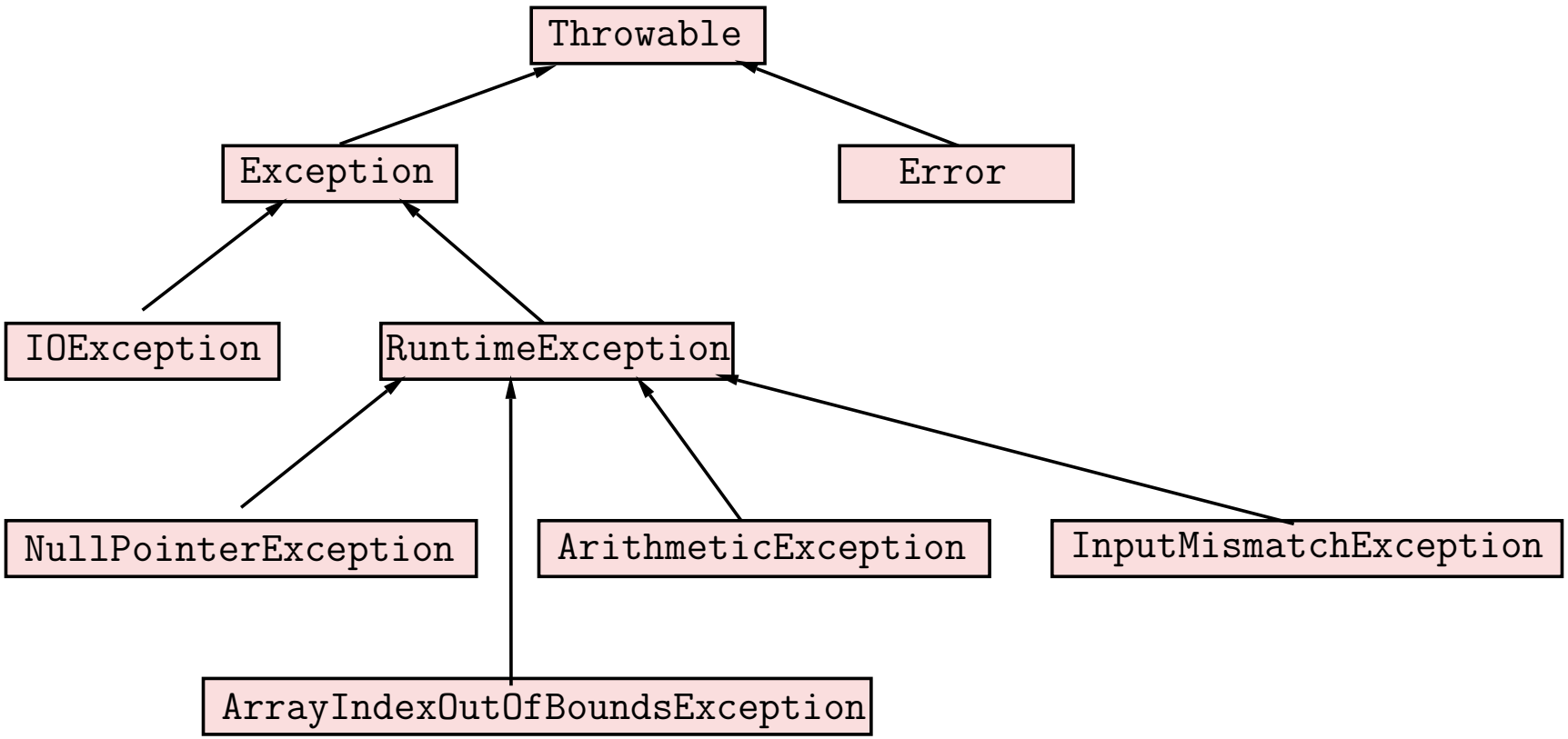
Idée :

- Séparer le code de gestion d'erreurs du code "normal".
  - pas d'empilements d'instructions conditionnelles
  - code plus propre et lisible
- Utiliser une construction syntaxique permettant de *rattraper* une exception, et d'exécuter un morceau de code avant de reprendre l'exécution normale du programme.
- Regrouper des types d'erreurs et gérer ensemble les erreurs du même type.

Nature des exceptions en `Java`

- Une exception est un objet, instance de la classe `Exception`.
- Cette classe hérite de la classe `Throwable`.

La hiérarchie des exceptions



La classe `Error`

La classe `Error` et ses sous-classes represent des situations anormales qui pourraient se produire dans la JVM.

Un exemple classique : `OutOfMemoryError`

```
public class ErreurMemoire {

    public static void main(String[] args) {
        int[] tableau = new int[1000000000];
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at ErreurMemoire.main(ErreurMemoire.java:5)
```

La classe `Error` (II)

Quelques sous-classes de la classe `Error` :

- `OutOfMemoryError`, `ThreadDeath`, `IOError`, `AnnotationFormatError`, ...

Les `Errors` correspondent à des problèmes graves et ne doivent pas être rattrapées par l'application. Il est en général impossible pour les applications de récupérer après des erreurs de ce type.

Le méchanisme `try-catch`

```
// Instructions anodynes (affectations, ...)
try {
    // Instructions susceptibles de provoquer des erreurs
}
catch (TypeException1 e1) {
    // Instructions de traitement des exceptions de type TypeException1
}
catch (TypeException2 e2) {
    // Instructions de traitement des exceptions de type TypeException2
}
finally {
// Instructions qui seront toujours executées
}
```

- La clause `try` doit entourer les instructions qui risquent de provoquer une exception.
- En cas d'exception ce sont les instructions du bloc `catch` correspondant qui seront exécutées.
- Les instructions bu bloc `finally` seront toujours exécutées.

Le méchanisme `try-catch` -Exemple

```
public static void main(String[] args) {

    int numerateur, denominateur, resultat;

    Scanner scanner = new Scanner(System.in);
    boolean continueBoucle = true;

    do {
        try { //lire deux nombres et calculer le quotient de la division
            System.out.println("Entrez un numérateur: " );
            numerateur = scanner.nextInt();
            System.out.println("Entrez un dénominateur: " );
            denominateur = scanner.nextInt();

            resultat = quotient(numerateur, denominateur);
            System.out.println("Le résultat est " + resultat);
            continueBoucle = false; //saisie correcte, fin de la boucle
        } //fin try
        catch (ArithmeticException arithmeticException) {
            System.out.println("Exception: " + arithmeticException);
            System.out.println("Le dénominateur doit être différent de zéro. Essayez encore");
        } // catch
    } while(continueBoucle); //fin do...while


    scanner.close();
} //fin main
```

Le bloc `try`

```
try { //lire deux nombres et calculer le quotient de la division
    System.out.println("Entrez un numérateur: " );
    numerateur = scanner.nextInt();
    System.out.println("Entrez un dénominateur: " );
    denominateur = scanner.nextInt();

    resultat = quotient(numerateur, denominateur);
    System.out.println("Le résultat est " + resultat);
    continueBoucle = false; //saisie correcte, fin de la boucle
} //fin try
```

- Le bloc `try` contient le code susceptible de lever une exception.
- Si une exception est levée, le reste du code dans le bloc `try` est ignoré.

Le bloc catch

```
catch (ArithmeticException arithmeticException) {
    System.out.println("Exception: " + arithmeticException);
    System.out.println("Le dénominateur doit être différent de zéro. Essayez encore");
} // catch
```

- Un bloc `try` doit toujours être suivi par au moins un bloc `catch` (ou d'un bloc `finally`).
- L'instruction catch prend un unique paramètre

  `catch(<type> <variable>)`

  - `<type>` représente le type de l'exception et doit être une sous-classe de `Throwable`
  - `<variable>` est le nom de la variable (locale) liée à l'exception

Lors de l'exécution

```
catch (ArithmeticException arithmeticException) {
    System.out.println("Exception: " + arithmeticException);
    System.out.println("Le dénominateur doit être différent de zéro. Essayez encore");
} // catch
```

```
Entrez un numérateur:
8
Entrez un dénominateur:
0
Exception: java.lang.ArithmeticException: / by zero
Le dénominateur doit être différent de zéro. Essayez encore
Entrez un numérateur:
8
Entrez un dénominateur:
4
Le résultat est 2
```

Traiter plusieurs types d'exception

```
try { //lire deux nombres et calculer le quotient de la division
    System.out.println("Entrez un numérateur: " );
    numerateur = scanner.nextInt();
    System.out.println("Entrez un dénominateur: " );
    denominateur = scanner.nextInt();

    resultat = quotient(numerateur, denominateur);
    System.out.println("Le résultat est " + resultat);
    continueBoucle = false; //saisie correcte, fin de la boucle
} //fin try
catch (ArithmeticException arithmeticException) {
    System.out.println("Exception: " + arithmeticException);
    System.out.println("Le dénominateur doit être différent de zéro. Essayez encore.");
} // fin catch
catch (InputMismatchException inputMismatchException) {
    System.out.println("Exception: " + inputMismatchException);
    scanner.nextLine(); //Éliminer l'entrée pour que l'utilisateur puisse re-essayer.
    System.out.println("Le dénominateur doit être un entier. Essayez encore.");
} // fin catch
```

Lors de l'exécution

```
Entrez un numérateur:
8
Entrez un dénominateur:
bonjour
Exception: java.util.InputMismatchException
Le dénominateur doit être un entier. Essayez encore.
Entrez un numérateur:
3
Entrez un dénominateur:
0
Exception: java.lang.ArithmeticException: / by zero
Le dénominateur doit être différent de zéro. Essayez encore.
Entrez un numérateur:
3
Entrez un dénominateur:
2
Le résultat est 1
```

Remarques

- Après que l'exception levée est gérée, le programme continue après le bloc `catch`.
- C'est une erreur syntaxique d'écrire du code entre le bloc `try` et ses blocs `catch` correspondants.
- Chaque bloc `catch` ne peut gérer qu'un seul type d'exception. Écrire

```
catch (ExceptionType e1, ExceptionType e2)
```

est une erreur syntaxique.

Le bloc `finally`

1. Le bloc `finally` fournit un mécanisme pour nettoyer l'état du programme et libérer des ressources.
2. Il s'agit d'un bloc optionnel.
3. Le bloc `finally` se place après le dernier bloc `catch`.
4. Les instructions du bloc `finally` sont toujours exécutées (sauf si la méthode `System.exit()` est appelée dans le bloc `try`).

Exemple d'utilisation du bloc `finally`

- Le bloc `finally` peut par exemple être utilisé pour fermer des fichiers ouverts dans la clause `try`.
- Même si un `return`, un `break` ou un `continue` est présent dans le bloc `try`, le bloc `finally` sera toujours exécuté.

```
finally {
    if (fichier != null) {
        System.out.println("Fermer le fichier");
        fichier.close();
    }
    else {
        System.out.println("Le fichier n'est pas ouvert");
    }
}
```

Respecter la hiérarchie des exceptions

```
try {
    String chemin="Chemin/vers/une/classe/qui/n'existe/pas";
    Class.forName(chemin);//levée d'une ClassNotFoundException
    System.out.println("Fin du programme");
}
catch(Exception e) {
    //traitement
}
catch(ClassNotFoundException ex) {
    System.out.println("Une exception est survenue");
}
```

- Les blocs `catch` sont testés par ordre d'écriture.
- Ce code ne compilera pas.
- Il faut respecter l'hiérarchie des exceptions, du plus spécifique au plus générique.

Respecter la hiérarchie des exceptions (II)

Il faut écrire à la place :

```
public static void main(String[] args) {

    try {
        String chemin="Chemin/vers/une/classe/qui/n'existe/pas";
        Class.forName(chemin);//levée d'une ClassNotFoundException
        System.out.println("Fin du programme");
    }
    catch(ClassNotFoundException ex) {
        System.out.println("Une exception est survenue");
    }
    catch(Exception e) {
        //traitement
    }
}
```

La classe `Throwable`

- La classe `Throwable` est la super-classe (classe mère) de toutes les exceptions ou erreurs en Java.
- Seules les instances de cette classe ou d'une de ces sous-classe peuvent être lancées.
- Une instance de `Throwable` peut avoir une cause qui est une autre instance de `Throwable` à l'origine de la création de cette instance (*chaînage des exceptions*).

Les constructeurs de la classe `Throwable`

- `Throwable`()
  - Le constructeur par défaut.
- `Throwable`(`String` message)
  - Le constructeur avec comme seul paramètre le message d'erreur.
- `Throwable`(`Throwable` cause)
  - Le constructeur avec comme seul paramètre la cause de l'exception.
- `Throwable`(`String` message, `Throwable` cause)
  - Le constructeur avec deux paramètres, le message d'erreur et la cause de l'exception.

Quelques méthodes de la classe `Throwable`

- `String getMessage()` : Retourne le message descriptif associé, stocké dans l'exception.
- `Throwable getCause()`: Retourne la cause de cette exception ou `null` si la cause n'existe pas ou si elle est inconnue.
- `void printStackTrace()` : Affiche l'exception et la stack trace dans le fichier d'erreur `err`.

La méthode getStackTrace()

Retourne un tableau représentant l'état de la pile, au moment où l'exception a été levée. Un élément de stack, contient les informations suivantes :

- le nom de la méthode
- le numéro de ligne où a été levée l'exception
- le nom de la classe
- le nom du fichier

La clause `throws`

- Souvent, lorsque l'erreur se produit, la méthode n'a pas les moyens de la corriger et doit le signaler à l'utilisateur.
- Il faut ajouter à la déclaration de la méthode une clause `throws` indiquant le type d'erreur auquel l'utilisateur doit s'attendre.
- La clause `throws` s'ajoute à la signature de la méthode.

```
typeRetour nomMethode throws typeException1, typeException2
```

- Cette déclaration n'est pas obligatoire pour les exceptions du type `Error` ou `RuntimeException`.
- Avec cette clause on laisse le soin à l'appelant de régler le problème.

`throws` ou `catch` ?

Lors de l'appel d'une méthode qui lève une exception :

- `catch` si l'on peut reprendre sur l'erreur et faire quelque chose de cohérent sinon
- `throws` propage l'exception vers celui qui l'a appelé et qui traitera l'erreur.

Lancer une exception

- L'instruction `throw` est utilisée pour lancer une exception.
- Le mot-clé `throw` doit être suivi d'un objet d'une classe dérivée de `Throwable`.

```
throw objet Throwable;
```

- Une exception peut être relancée à partir d'un bloc `catch`.

Un exemple

```
public String calculeNomComplet(String prenom, String nom) throws Exception {

    if (prenom == null || prenom.length() == 0)
        throw new Exception("Le prenom n'est pas valide.");

    if (nom == null || nom.length() == 0)
        throw new Exception("Le nom n'est pas valide.");

    return prenom + " " + nom;
}
```

Créer des classes exceptions

- Définir les classes exceptions est une étape importante.
- Il faut déterminer `dans quelles méthodes` et `sous quelles conditions` des exceptions seront lancées.
- Deux alternatives sont possibles pour choisir le type de chaque exception :
  - utiliser une exception existante
  - en créer une nouvelle
- Il reste à choisir quelle sera la super-classe des exceptions définies (`Exception` ou l'une de ses sous-classes)

Création de notre propre classe d'exception

```
public class PrenomPasValideException extends Exception {
    private final String nom ;

    public PrenomPasValideException(String nom)
    {
        super("Prenom pas valide: " + nom) ;
        this.nom = nom ;
    }

    public final String getNom()
    {
        return nom ;
    }
}
```

Création d'une deuxième classe d'exception

```
public class NomPasValideException extends Exception {
    private final String nom ;

    public NomPasValideException(String nom)
    {
        super("Nom pas valide: " + nom) ;
        this.nom = nom ;
    }

    public final String getNom()
    {
        return nom ;
    }
}
```

Utiliser ses propres exceptions

```
public String calculeNomComplet(String prenom, String nom) throws
                        PrenomPasValideException, NomPasValideException {

    if (prenom == null || prenom.length() == 0)
        throw new PrenomPasValideException(prenom);

    if (nom == null || nom.length() == 0)
        throw new NomPasValideException(nom);

    return prenom + " " + nom;
}
```

Attraper ses propres exceptions

```
try {
    ...
    String nomComplet = calculeNomComplet(prenom,nom) ;
    ...
}
catch (PrenomPasValideException e) {
    ...
    String nom = e.getNom() ;
    ...
}
catch (NomPasValideException e) {
    ...
}
catch (Exception e) {
    ...
}
```

La bonne façon de faire

- Utiliser les exceptions standard dès que possible.
- Eviter de réinventer la roue.
- Tout le monde les connait, le code sera donc plus facilement lisible. De plus elles sont documentées et adaptées aux cas prévus.

Propagation des exceptions

```
class MonException extends Exception {
    MonException() {
        System.out.println("Me voila");
    }
}

public class Propagation {

    static void methode3() throws MonException {
        try {
            if(true) throw new MonException();
            System.out.println("et moi?");
        }
        finally {
            System.out.println("niveau 3");
        }
        System.out.println("instruction inutile");
    }

    static void methode2() throws MonException {
        try {
            methode3();
            System.out.println("et ici?");
        }
        finally {
            System.out.println("niveau 2");
        }
    }

    static void methode1() {
        try {
            methode2();
            System.out.println("et ici?");
        }
        catch(MonException e) {
            System.out.println("exception attrapee...");
            e.printStackTrace();
        }
        System.out.println("reprise du cours normal");
    }
    public static void main(String[] args) {
        methode1();
    }
}
```

Exécution

```
Me voila
niveau 3
niveau 2
exception attrapee...
MonException
reprise du cours normal
        at Propagation.methode3(Propagation.java:11)
        at Propagation.methode2(Propagation.java:22)
        at Propagation.methode1(Propagation.java:32)
        at Propagation.main(Propagation.java:42)
```

Exemple dans *"Le langage Java"* de Irène Charon

Un avantage

Regrouper et différentier les différents types d'erreurs.

- Les exceptions sont des objets, les grouper et les categoriser est donc naturel.

Exemple : La classe `IOException` et ses sous-classes.

- Traiter tout type de problème lié aux entrées-sorties.

Le programmeur peut choisir quel type d'exception il souhaite traiter.

- `catch (FileNotFoundException e)` : Traiter un seul type d'exception.
- `catch (IOException e)` : Traiter tout type d'exception lié aux entrées sorties.
- `catch (Exception e)` : Beaucoup trop général. A utiliser si la seule chose qui nous intéresse est imprimer un message d'erreur et terminer le programme.

Conclusion

- Le mécanisme des exceptions permet de traiter proprement les erreurs en séparant leur traitement du traitement des cas normaux.
- Il permet au programmeur d'éviter de faire des tests.
- Le programmeur laisse l'erreur éventuelle se produire puis récupère l'exception à l'aide d'un bloc `try-catch` afin de l'analyser.
- Le code est plus lisible, plus comprehensible et plus facile à déboger.
- Les exceptions sont des objets, on bénéficie donc de toute la puissance de l'héritage.