
Programmation orientée objet en Java

# Gestion des exceptions et logging

**Dominique Blouin**
**Télécom Paris, Institut Polytechnique de Paris**
**dominique.blouin@telecom-paris.fr**

# Objectifs d'apprentissage

- Erreurs lors de l'exécution d'un programme
- Exceptions en Java
- Exceptions applicatives
- Logging

# Traitement des erreurs en programmation

- Traiter les erreurs pouvant se produire lors de l'exécution d'un programme est **essentiel**.
- Une mauvaise (ou absence) de traitement des erreurs peut être la cause de **plusieurs problèmes**.
- C'est un aspect qui est souvent **négligé** dans plusieurs applications logicielles.
- Cela peut causer des comportements **inattendus** du programme…

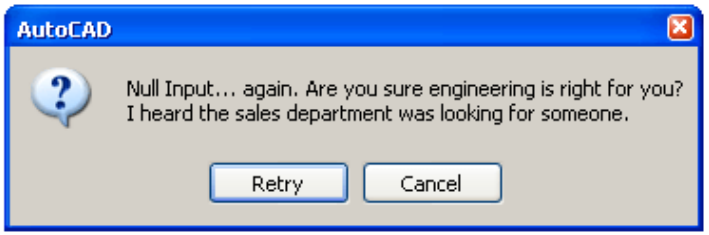
# Exemples de mauvais traitements d'erreurs

- Ou de messages d'erreur décourageants…

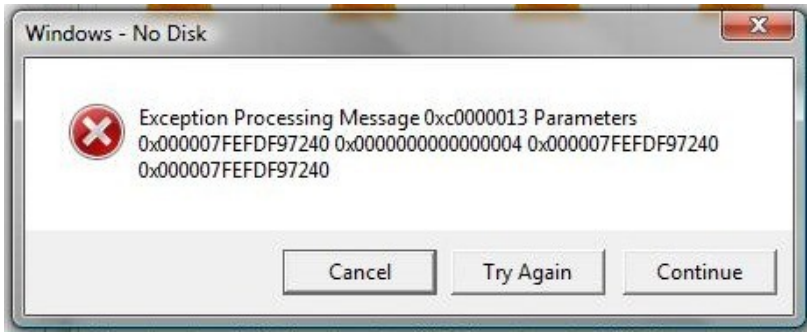
The following errors were found, please correct them and re-submit the form:

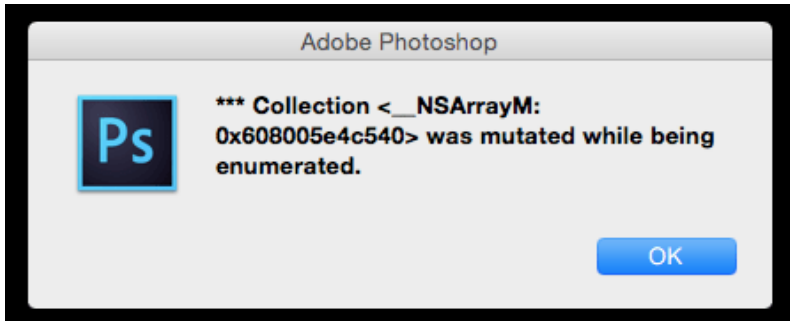
- Fields must not be the same:
- Password must be at least 6 characters
- Forename is a required field.
- Please enter a valid email address.
- Surname is a required field.
- Confirm Email is a required field.
- Password is a required field.
- Confirm Password is a required field.
- Job Title is a required field.
- Name of organisation is a required field.
- Head office based in is a required field.
- Company Sector is a required field.
- No. of employees is a required field.
- Company Turnover is a required field.
- The value you entered for the captcha validation was incorrect.

- Ou condescendant…



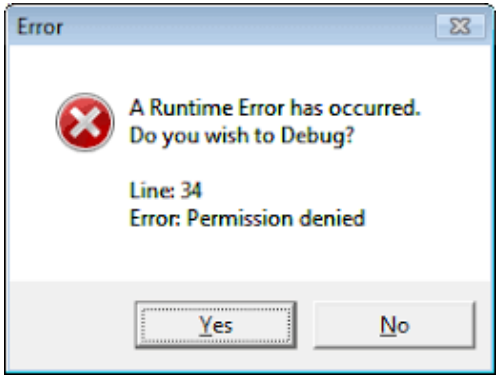

# Messages d'erreur incompréhensibles (techniques) ou qui ne donnent pas d'information

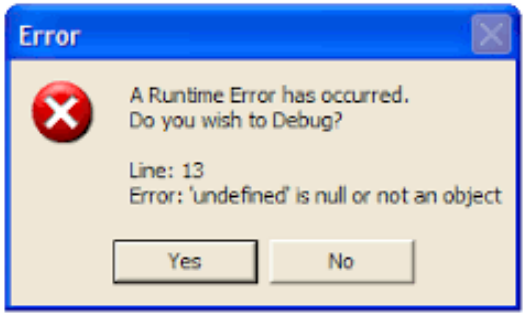


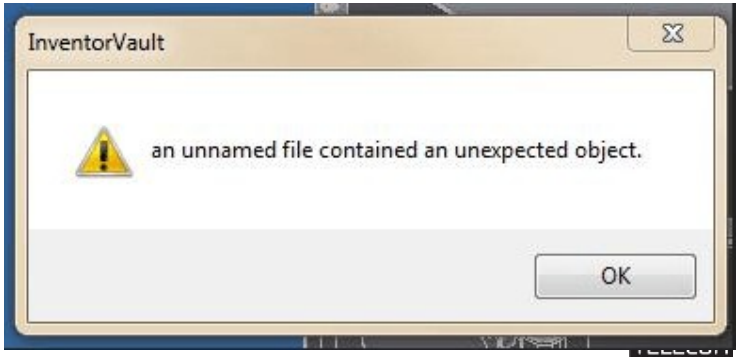





- Ou qui n'ont rien à voir avec le vrai problème…



# Traitement d'erreur qui potentiellement révèle des données confidentielles (sécurité)

```
1   try {
2   openDbConnection();
3   }
4   //print exception message that includes exception message and configuration file location
5   catch (Exception $e) {
6   echo 'Caught exception: ', $e->getMessage(), '\n';
7   echo 'Check credentials in config file at: ', $Mysql_config_location, '\n';
8   }
9
```

# Traitement des erreurs en Java

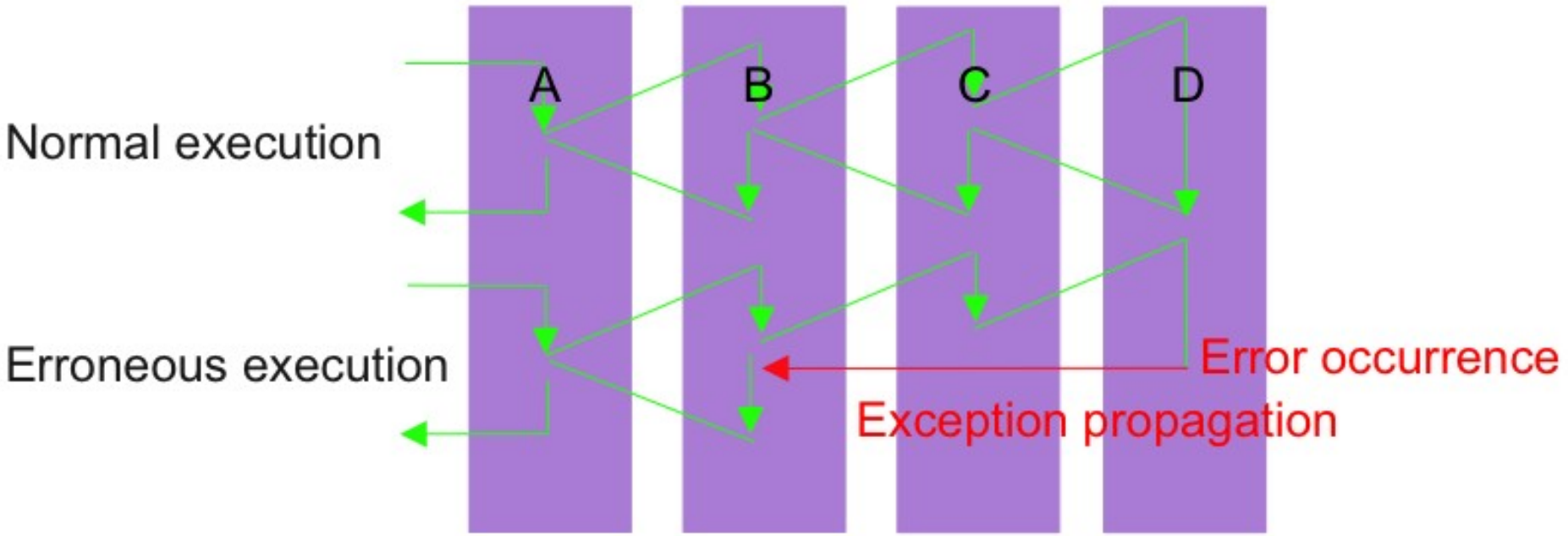
- Une erreur se produit normalement dans une méthode, qui a été appelée par une méthode, qui a été appelée par une méthode, et ainsi de suite jusqu'à la méthode initiale **`main`** qui a été appelée lors du démarrage du programme.

- Quand l'erreur se produit, 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 jusqu'à la méthode **`main`** initiale.
  - On dit que l'exception est **propagée** jusqu'à la méthode **`main`**.

- Le programme s'arrête et un message d'erreur s'affiche à la console en prenant en compte les informations collectées par l'objet **`exception`** pendant sa propagation.

# Trace d'exécution d'un programme, occurrence d'une erreur et propagation d'une exception



# Une erreur fréquente en Java

```
java.lang.Exception: java.lang.NullPointerException
        at utils.CheckData.checkNotEmpty(CheckData.java:63)
        at submission.Submission.handle(Submission.java:146)
        at Main.main(Main.java:16)
```

- On voit les **appels imbriqués** de méthodes : **`main`** (en bas) a appelé **`handle`** qui a appelé **`checkNotEmpty`** où l'erreur s'est prroduite à la ligne 63.
- On voit également les **numéros de lignes** de ces appels dans les fichiers source.
- On voit le **nom de l'erreur** et le **numéro de ligne du fichier** où elle s'est produite.
- Note: dans Eclipse, un clic sur la trace de l'exception ouvre automatiquement le fichier concerné à la ligne donnée par la trace.

# Différents types d'erreurs en Java

- Premier type : les erreurs **graves**.
  - Elles ne peuvent pas être traitées par le programmeur et provoquent presque obligatoirement **l'arrêt du programme**.
- Elles sont représentées par des objets de la classe **`Error`**
- Exemples :
  - Manque de mémoire de la machine virtuelle (**`OutOfMemoryError`**).

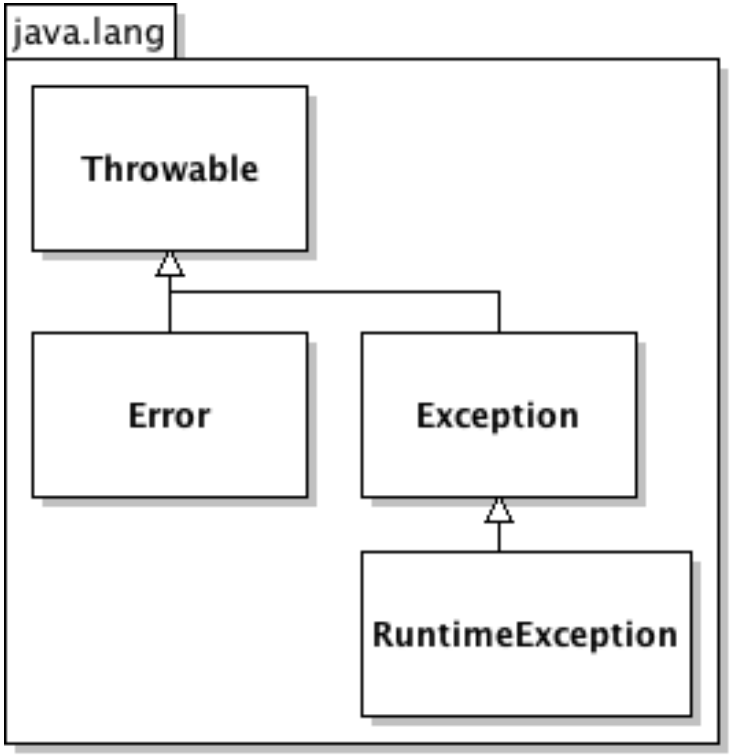
# Différents types d'erreurs en Java

- Deuxième type : les erreurs devant être **gérées** par le **programmeur**.
- Elles sont représentées par des objets de la classe **`Exception`**.
- Exemple :
  - Erreurs d'entrée / sortie (**`IOException`**).

# Différents types d'erreurs en Java

- Troisième type : les erreurs liées au **langage**.
- C'est un cas particulier du deuxième type.
- Représentées par des objets de la classe **`RuntimeException`**
  - Qui est une sous-classe de la classe **`Exception`**.
- Exemple :
  - Erreurs d'arithmétique (division par 0).
    - **`ArithmeticException`**

# Diagramme de classe des exceptions




- Toutes les erreurs **propagées** sont des sous-classes de la classe **`Throwable`**.

# Propagation des erreurs

- Quand une erreur survient, un objet de la classe **`Throwable`** est créé et propagé jusqu'à la méthode **`main`** initiale.

- Cet objet peut être de type :
  - **`Error`** (erreurs graves).
  - **`Exception`** (erreurs communes).
  - Sous-classe de **`Exception`** (erreurs particulières).
  - **`RuntimeException`** (erreurs langage).
  - Sous-classe de **`RuntimeException`** (erreurs particulières).

- Les sous-classes de **`Exception`** ou de **`RuntimeException`** sont des spécialisations représentant des **cas particuliers d'erreurs**.

- Recherche : JAVA SE **`Exception`**

# Traitement des erreurs

- Il est possible **d'arrêter la propagation** d'une erreur à l'aide d'une instruction **`try-catch`** :

```
try {
   ... // Instructions pouvant déclencher
   ... // une erreur
}
catch (Exception ex) {
   ... // Instructions à exécuter en
   ... // cas d'erreur
}
```

- Lorsque l'on **arrête** la propagation d'une erreur, on **exécute** des instructions de **substitution** situées dans le bloc **`catch`** où l'on récupère l'objet de type **`Exception`**.

# Traitement des erreurs

- Lors de l'exécution d'un bloc :

```
try {
    ...
}
catch (Exception ex) {
    ...
}
...
```

- Les instructions du bloc **`try`** sont exécutées.
  - Si aucune exception n'est levée et propagée durant l'exécution de ce bloc, les instructions après le bloc `catch` sont exécutées.
  - Si une exception est propagée, elle est stoppée et se retrouve dans la variable `ex` et l'on exécute le code du bloc `catch`.

- Dans le bloc **`catch`**, on peut :
  - Afficher un message d'erreur spécifique.
  - Corriger les conditions de l'erreur et faire un autre calcul.
  - Etc.

# Exemple de traitement d'erreur grave (classe Error)

- Ces erreurs **ne peuvent être réparées**, alors on se contente d'afficher un message d'erreur.

```
public static void main(String[] args) {
        int size = Integer.parseInt(args[0]);

            try {
                List<Integer> manyInts = new ArrayList<Integer>();

                for (int index = 0; index < size; index++) {
                    manyInts.add(new Integer(0));
                }
            }
            catch (Error err) {
                System.out.println("Error: " + err);
            }
    }
}
```

- Si la valeur de **`size`** est assez grande on aura :
  - **`Error: java.lang.OutOfMemoryError: Java heap space`**

# Traitement d'erreur de langage

- Les exceptions de type **`RuntimeException`** correspondent à des erreurs ordinaires liées à l'exécution du programme.

- Recherche : JAVA SE **`ArrayList`**
  - Regardons la méthode **`get(int index)`**.

- Cette méthode peut déclencher une exception de type **`IndexOutOfBoundsException`** qui est une sous-classe de **`RuntimeException`**.

# Traitement sélectif des erreurs

- Il est possible d'arrêter **sélectivement** les exceptions.

```
try {
    ...
}
catch (IndexOutOfBoundsException ex) {
    ...
}
catch (ArithmeticException ex) {
    ...
}
catch (Exception ex) {
    ...
}
```

- Seul le premier bloc **`catch`** dont le type correspond à l'exception propagée sera exécuté.

- Donc l'**ordre** des blocs **`catch`** est important.

# Clause `finally`

- La clause **`finally`** est positionnée en **fin** d'un bloc **`try-catch`**.

- Elle permet de spécifier une suite d'instructions qui sera exécutée **après** l'exécution les instructions du bloc **`try-catch` quelle que soit la manière** dont s'est déroulée l'exécution de ce bloc.

```
try {
    ...
}
catch (IndexOutOfBoundsException ex) {
    ...
}
catch (Exception ex) {
    ...
}
finally {
    ...
}
```

# Exemple classique d'utilisation de la clause `finally`

```
try {
    ... // Ouverture d'un fichier en lecture
    ... // Lecture et traitement du fichier
}
catch (IOException ex) {
    ... // Affichage d'un message d'erreur I/O
}
catch (Exception ex) {
    ... // Gestion des autres erreurs
}
finally {
    ... // Fermeture du fichier
}
```

- Cela permet de fermer le fichier **quoi qu'il arrive**.

# Affichage de la trace d'appels de la pile

- Recherche : JAVA SE **Exception**

- Une méthode importante est **printStackTrace()** qui affiche la séquence des appels de méthodes au moment où s'est produit l'erreur.

```
try {
    ...
}
catch (Exception ex) {
    ...
    ex.printStackTrace();
    ...
}
```

# Programmer le déclenchement d'une exception

- Le programmeur peut **déclencher** une exception à l'aide de l'instruction **`throw`** :

```
public String computeFullName(String firstName,
                              String lastName) {
    if (firstName == null || firstName.isEmpty()) {
        throw new Exception("First name is invalid");
    }

    if (lastName == null || lastName.isEmpty()) {
        throw new Exception("Last name is invalid");
    }

    return firstName + " " + lastName;
}
```

# Exceptions *vérifiées*

- La méthode **`computeFullName()`** précédente **ne compile pas**!

- Toutes les exceptions autres que **`RuntimeException`** sont dites de type **vérifiées** (**checked**).
  - **`RuntimeException`** (et ses sous-classes) : non vérifiée (unchecked).

- Si une méthode peu lever une exception de type **vérifié**, il faudra alors le **déclarer** avec le mot clé **`throws`**:

```
public String computeFullName(String firstName,
                              String lastName)
throws Exception {
    if (firstName == null || firstName.isEmpty()) {
        throw new Exception("First name is invalid");
    }

    if (lastName == null || lastName.isEmpty()) {
        throw new Exception("Last name is invalid");
    }

    return firstName + " " + lastName;
}
```

- Sinon il y aura une **erreur de compilation**.

# Définir ses propres exceptions

- Le programmeur peut définir ses **propres** exceptions en tant que **sous-classes** de la classe **`Exception`**.

```
public class InvalidFirstNameException extends Exception {
    private final String name; // Data embedded into the exception

    public InvalidFirstNameException(String name) {
        super("Invalid first name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

# Définir ses propres exceptions

- Une autre classe d'**`Exception`**.

```
public class InvalidLastNameException extends Exception {
    private final String name; // Data embedded into the exception

    public InvalidLastNameException(String name) {
        super("Invalid last name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

# Utiliser ses propres exceptions

- Le programmeur peut **lever** ses propres exceptions :

```
public String computeFullName(String firstName,
                              String lastName)
throws InvalidFirstNameException,
       InvalidLastNameException {
    if (firstName == null || firstName.isEmpty()) {
        throw new InvalidFirstNameException(firstName);
    }

    if (lastName == null || lastName.isEmpty()) {
        throw new InvalidLastNameException(lastName);
    }

    return firstName + " " + lastName;
}
```

# Traiter ses propres exceptions

```
try {
    ...
    String fullName = computeFullName(firstName, lastName);
    ...
}
catch (InvalidFirstNameException ex) {
     String name = ex.getName();
     ... // Display error message to log file or error dialog
box
}
catch (InvalidLastNameException ex) {
    ...
}
catch (Exception ex) {
    ...
}
```

# Exercice : proposer une *meilleure modélisation* de ces deux exceptions

```
public class InvalidFirstNameException extends Exception {
    private final String name; // Data embedded into the exception

    public InvalidFirstNameException(String name) {
        super("Invalid first name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

```
public class InvalidLastNameException extends Exception {
    private final String name; // Data embedded into the
exception

    public InvalidLastNameException(String name) {
        super("Invalid last name: " + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}
```

# Solution : définir une exception parente commune

```
public class InvalidNameException extends Exception {
    private final String name; // Data embedded into the exception

    public InvalidNameException(String messagePrefix,
                                String name) {
        super(messagePrefix + name);

        this.name = name;
    }

    public final String getName() {
        return name;
    }
}

public class InvalidFirstNameException extends
InvalidNameException {

    public InvalidFirstNameException(String name) {
        super("Invalid first name ", name);
}
```

# Bon usage des exceptions

- Cette implémentation est-elle bonne?

```
public Student findStudent(int index) {
    try {
        return studentsList.get(index);
    }
    catch (IndexOutOfBoundsException ex) {
        return null;
    }
}
```

N'utiliser les exceptions que pour la gestion des exceptions!

```
public Student findStudent(int index) {
    if (index < 0 || index >= studentsList.size()) {
        return null;
    }

    return studentsList.get(index);
}
```

Utiliser les structures de contrôle standard (meilleures performances)

# Logging

- Consiste à gérer l'émission et le stockage de **messages** (traces) à la suite de différents événements qui se produisent lors de l'exécution d'un programme.

- Jusqu'à maintenant, nous avons principalement utilisé la méthode **`System.out.println()`** pour afficher les messages d'exécution à la console.

- Qu'en est-il si on souhaite **conserver ces messages** (dans un fichier par exemple), que ce soit pour :
  - Déboguer.
  - Obtenir des traces d'exécution (démarrage/arrêt, informations, avertissements, erreurs d'exécution, ...).
  - Ou faciliter la recherche d'une source d'anomalie (stacktrace, ...).

- **`System.out.println()`** est alors clairement **insuffisant**...

# Logging

- Pour solutionner ce problème, on utilise une API de **Logging**.

- Celle-ci fait généralement intervenir trois principaux composants :
  - **Logger** : pour émettre un message généralement avec un niveau de gravité associé.
  - **Formatter** : pour formater le contenu des messages.
  - **Appender** : pour envoyer le message à une **cible de stockage** donnée (console, fichier, base de données, email, etc.).

- Différentes bibliothèques de logging :
  - **Apache Log4j** : une des premières…
  - **Java Logging** : inclue dans le JDK à partir de Java 4.
  - **SLF4J** (Simple Logging Facade for Java).
  - Etc.

# Exemple avec la bibliothèque de logging du JDK

```
import java.util.logging.Logger;

public class Main {

    private static final Logger LOGGER =
Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        LOGGER.info("Starting the simulation...");

        LOGGER.config("With parameters '" + Arrays.toString(args) + "'...");
                ...

        try {
            ...
        }
        catch (Exception ex) {
            LOGGER.log(Level.SEVERE, "Error…", ex);
        }
    }
...
```

# Niveaux de logging et affichage des messages

- Différents **niveaux** de logging :
  - Level.SEVERE : initialisée avec la valeur 1000
  - Level.WARNING : initialisée avec la valeur 900
  - Level.INFO : initialisée avec la valeur 800
  - Level.CONFIG : initialisée avec la valeur 700
  - Level.FINE : initialisée avec la valeur 500
  - Level.FINER : initialisée avec la valeur 400
  - Level.FINEST : initialisée avec la valeur 300

- Ensuite, on configure le niveau du logger pour l'**application** (ou par appender ou handler).

```
LOGGER.setLevel(Level.INFO);
```

- Pour chaque appel à la méthode **`log`**, le message ne sera affiché que si le niveau de logging du message est égal ou plus élevé que celui de l'application :

```
LOGGER.info("Starting the simulation..."); // Will be written
LOGGER.config("With parameters '" + Arrays.toString(args) + "'..."); // Will not be written
```

# Définir le niveau de logging de l'application

- Dans un fichier de configuration nommé **`logging.properties`** :

```
# Limit the messages that are printed on the console to FINE and above.
java.util.logging.ConsoleHandler.level = FINE
```

- Spécifier l'**endroit** du fichier de configuration :
  - Par défaut, un fichier est fourni dans le répertoire **`jre/conf/logging.properties`**
  - On peut redéfinir l'emplacement et le nom du fichier via un argument de la JVM :
    - -Djava.util.logging.config.file=<my application dir>/config/logging.properties

- Définir le niveau de logging à partir du code :

```
LOGGER.setLevel(Level.INFO);
```

# Exemple avec le logging du JDK

- La classe **`Handler`** :
  - **`StreamHandler`** : envoie des messages dans un flux de sortie.
  - **`ConsoleHandler`** : envoie des messages sur la sortie standard (console).
  - **`FileHandler`** : envoie des messages sur un fichier.
  - **`SocketHandler`** : envoie des messages dans une socket réseau.

- Exemple avec un fichier :

```
Handler fileHandler;

try {
    fileHandler = new FileHandler("TestLogging.log");
    LOGGER.addHandler(fileHandler);
}
 // Catches any of SecurityException or IOException
catch (SecurityException | IOException ex) {
    LOGGER.log(Level.SEVERE, "Error…", ex);
}
```

# Conclusion

- Le mécanisme de gestion des erreurs de Java est très puissant.
- Il permet au programmeur de traiter les erreurs plus facilement.
  - Le programmeur laisse l'erreur éventuelle se produire puis récupère l'exception à l'aide d'un bloc **`try-catch`** afin de la traiter.
- L'utilisation des exceptions doit être **réservée au traitement de situations exceptionnelles** pour lesquelles les performances ne sont pas essentielles.
- Utiliser les **logger** autant que possible. C'est une fonctionnalité essentielle pour les applications complexes.
- Eviter d'utiliser directement **`System.out.println(…)`** et **`ex.printStackTrace()`**
  - C'est le **`Logger`** qui appellera ces méthodes en interne au besoin.