# Exceptions en Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

## Table des matières

1 Objectifs d'apprentissage ..... 1
2 Introduction aux Exceptions ..... 2
2.1 Objectifs de la Gestion des Exceptions ..... 2
3 Pourquoi sécuriser un programme ? ..... 2
4 Sécuriser un programme **sans gestion spécifique des erreurs** ..... 2
4.1 Défis ..... 3
5 Approche de Java pour la sécurité ..... 3
5.1 La classe `Exception` ..... 3
5.2 Types d'Exceptions ..... 4
6 Syntaxe de base de la Gestion des Exceptions ..... 5
6.1 Exemple avec l'ouverture d'un fichier ..... 5
7 Création d'Exceptions Personnalisées ..... 5
7.1 Autoclôture des Ressources ..... 6
7.2 Exemple avec l'ouverture d'un fichier ..... 6
8 Propagation des Exceptions ..... 7
8.1 Exemple d'Enveloppement d'Exception ..... 7
8.2 Avantages de la Propagation des Exceptions ..... 8
9 Assertions ..... 8
9.1 Exemple d'utilisation ..... 8
9.2 Meilleures Pratiques ..... 9
10 Exemple 1 : Gestion des Entrées/Sorties de Fichiers ..... 9
10.1 Exemple 2 : Exception Personnalisée ..... 9
10.2 Exemple 3 : Utilisation de try-with-resources ..... 10
11 Conclusion ..... 10

## 1 Objectifs d'apprentissage

- Comprendre le concept des exceptions en Java
- Apprendre à gérer les exceptions efficacement

- Créer des exceptions personnalisées
- Mettre en oeuvre les meilleures pratiques pour la gestion des exceptions

# 2 Introduction aux Exceptions

- **Qu'est-ce qu'une exception ?**
  - Une exception est un événement qui interrompt le flux normal d'un programme.
- **Pourquoi les exceptions sont-elles importantes ?**
  - Elles permettent de gérer les erreurs de manière structurée et de maintenir le flux normal du programme.

## 2.1 Objectifs de la Gestion des Exceptions

- **Anticiper les erreurs** : Identifier les scénarios d'erreur potentiels.
- **Détecter les erreurs** : Implémenter des mécanismes pour capturer les erreurs lorsqu'elles se produisent.
- **Réagir aux erreurs** : Définir comment le programme doit répondre à différents types d'erreurs.

# 3 Pourquoi sécuriser un programme ?

- **Anticiper les erreurs** : Identifier les scénarios d'erreur potentiels.
- **Détecter les erreurs** : Implémenter des mécanismes pour capturer les erreurs lorsqu'elles se produisent.
- **Réagir aux erreurs** : Définir comment le programme doit répondre à différents types d'erreurs.

# 4 Sécuriser un programme sans gestion spécifique des erreurs

- **Prédiction des erreurs** : Analyser le code et l'utilisation pour identifier les erreurs possibles.
- **Ajout de code de prévention/détection** : Utiliser des vérifications de conditions (`if` statements) et des assertions pour détecter les erreurs.
- **Réactions aux erreurs** : Définir comment gérer les erreurs :
  - **Avertissements** : Informer l'utilisateur ou le développeur.
  - **Corrections dynamiques** : Tenter de corriger l'erreur automatiquement.
  - **Arrêts contrôlés** : Arrêter le programme de manière sécurisée.
- **Propagation des erreurs** : Décider si l'appelant ou l'appelé gère l'erreur.

---

```java
public boolean validateInput(String input) {
    if (input == null || input.isEmpty()) {
         System.err.println("Entrée invalide : l'entrée ne doit pas être nulle
```

```
ou vide.");
            return false;
        }
        return true;
    }

    public void processInput(String input) {
        if (validateInput(input)) {
            // Traitement de l'entrée
            System.out.println("Traitement de l'entrée : " + input);
        }
    }

    processInput(null);
```

### 4.1 Défis

- **Lisibilité** : Mélanger le code de test avec le code de l'application rend la lecture difficile.
- **Maintenabilité** : Difficile de maintenir le code de gestion des erreurs.
- **Responsabilité** : Incertitude sur qui doit gérer l'erreur, l'appelant ou l'appelé.

## 5 Approche de Java pour la sécurité

- Java propose un mécanisme robuste pour gérer les erreurs via les exceptions.
- **Avantages** :
  - Sépare le code de gestion des erreurs du code régulier.
  - Propage les erreurs dans la pile d'appels.
  - Regroupe et différencie les types d'erreurs.

### 5.1 La classe `Exception`

- **Définition** : La classe `Exception` est la classe de base pour toutes les exceptions en Java.
  - Les exceptions sont des objets qui encapsulent des informations sur les erreurs.
- **Constructeurs** :
  - `Exception(String message)`: Construit une nouvelle exception avec le message détaillé spécifié.
  - `Exception(String message, Throwable cause)`: Construit une nouvelle exception avec le message détaillé spécifié et la cause.
    - La « cause » est éventuellement une autre exception qui a conduit à l'exception actuelle.

  - Cela permet de chaîner les exceptions pour mieux comprendre la séquence d'erreurs qui a conduit à un problème.

- Exceptions prédéfinies : `IllegalArgumentException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc.
- **Méthodes** :
  - `String getMessage()`: Retourne le message détaillé de l'exception.
  - `Throwable getCause()`: Retourne la cause de l'exception.
- printstacktrace() : Affiche la trace de la pile

---

```
Exception e = new Exception("Message d'erreur");
System.out.println("Le message d'erreur est : %s".formatted(e.getMessage()));
e.printStackTrace(); // Affiche la trace de la pile
```

## 5.2 Types d'Exceptions

- **Exceptions vérifiées (Checked Exceptions)** :
  - Vérifiées à la compilation.
  - Héritent de `Exception` (mais pas de `RuntimeException`) qui hérite de `Throwable`.
  - Doivent être gérées ou déclarées dans la signature de la méthode avec `throws`.
  - Exemple : `IOException`, `SQLException`
- **Exceptions non vérifiées (Unchecked Exceptions)** :
  - Vérifiées à l'exécution.
  - Héritent de `RuntimeException` (qui hérite de `Exception`).
  - Exemple : `NullPointerException`, `ArrayIndexOutOfBoundsException`
- **Erreurs (Errors)** :
  - Problèmes sérieux qu'une application ne devrait pas essayer de capturer.
  - Héritent de `Error` qui hérite de `Throwable`.
  - Exemple : `OutOfMemoryError`, `StackOverflowError`

---

```
// Hiérarchie des exceptions
Throwable
    ├── Exception
    │   ├── IOException
    │   ├── SQLException
    │   └── RuntimeException
    │       └── NullPointerException
    └── Error
            ├── OutOfMemoryError
            └── StackOverflowError
```

# 6 Syntaxe de base de la Gestion des Exceptions

- **Bloc `try-catch`** : Utilisé pour entourer le code qui peut générer une exception.
- **Bloc `try-finally`** : Utilisé pour exécuter un code qui doit s'exécuter, qu'une exception soit levée ou non.
- **Bloc `try-catch-finally`** : Combinaison des deux blocs précédents.

```
try {
    // Code pouvant générer une exception
} catch (ExceptionType1 e1) {
    // Gérer l'exception de type ExceptionType1
} catch (ExceptionType2 e2) {
    // Gérer l'exception de type ExceptionType2
} finally {
    // Code qui s'exécute toujours
}
```

## 6.1 Exemple avec l'ouverture d'un fichier

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

Logger logger = Logger.getLogger("Notebook");

import java.io.*;
```

```
try (BufferedReader reader = new BufferedReader(new FileReader("/etc/
issue.net"))) {
    String line;
    while ((line = reader.readLine()) ≠ null) {
        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    logger.severe("Erreur de fichier non trouvé %s".formatted(e.getMessage()));
} catch (IOException e) {
    logger.severe("Erreur de lecture du fichier %s".formatted(e.getMessage()));
}
```

# 7 Création d'Exceptions Personnalisées

- Les exceptions personnalisées permettent de définir des erreurs spécifiques à une application.
- Elles héritent de la classe `Exception` ou d'une de ses sous-classes.
- **Définir une nouvelle classe d'exception** :

```java
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
```

- `throws` : indique que la méthode peut lancer une exception qui doit être gérée par l'appelant.
- **Lancer une exception personnalisée** :

```java
boolean someCondition = true;

public void someMethod() throws CustomException {
    if (someCondition) {
        throw new CustomException("Message d'erreur personnalisé");
    }
}

someMethod();
```

## 7.1 Autoclôture des Ressources

- Java 7 a introduit le concept d'autoclôture des ressources pour simplifier la gestion des ressources.
- Les ressources qui implantent l'interface `AutoCloseable` peuvent être utilisées dans un bloc `try-with-resources`.
- Les ressources sont automatiquement fermées à la fin du bloc `try`.
- Pas besoin de bloc `finally` pour fermer les ressources.

## 7.2 Exemple avec l'ouverture d'un fichier

```java
public class MyDataException extends Exception {
    public MyDataException(String message, Throwable cause) {
        super(message, cause);
    }
}

public void readIssueFile(String filepath) throws MyDataException {
    try (BufferedReader reader = new BufferedReader(new FileReader(filepath))) {
        String line;
        while ((line = reader.readLine()) ≠ null) {
            System.out.println(line);
        }
```

```
    } catch (FileNotFoundException e) {
        throw new MyDataException("Erreur de fichier non trouvé", e);
    } catch (IOException e) {
        throw new MyDataException("Erreur de lecture du fichier", e);
    }
}
```

---

```
readIssueFile("/etc/issue.net");
```

```
readIssueFile("/etc/missingfile");
```

# 8 Propagation des Exceptions

- **Définition** : La propagation des exceptions est le processus par lequel une exception est transmise dans la pile d'appels jusqu'à ce qu'elle soit capturée par un bloc `catch` approprié ou jusqu'à ce que le programme se termine.
- **Envelopper une exception** : Une méthode peut envelopper une exception dans une autre exception plus spécifique avant de la propager. Cela permet de fournir plus de contexte sur l'erreur.
- **Traitement total ou partiel** : Un bloc `catch` peut traiter une exception partiellement et ensuite la propager ou en propager une autre. Cela permet de gérer certaines parties de l'erreur tout en laissant d'autres parties être gérées par des niveaux supérieurs.

## 8.1 Exemple d'Enveloppement d'Exception

```
    public class CauseException extends Exception {
        public CauseException(String message) {
            super(message);
        }
    }

    public void method2() throws CauseException {
        throw new CauseException("Erreur dans method2");
    }

    public class CustomException extends Exception {
        public CustomException(String message, Throwable cause) {
            super(message, cause);
        }
```

```
    }

    public void method1() throws CustomException {
        try {
            method2();
        } catch (CauseException e) {
            throw new CustomException("Erreur enveloppée dans method1", e);
        }
    }

    public void test() {
        try {
            method1();
        } catch (CustomException e) {
            System.err.println("Exception capturée : " + e.getMessage());
            e.printStackTrace();
        }
    }

    test();
```

### 8.2 Avantages de la Propagation des Exceptions

- **Séparation des préoccupations** : Permet de séparer le code de gestion des erreurs du code de logique métier.
- **Réutilisabilité** : Facilite la réutilisation des méthodes sans avoir à dupliquer le code de gestion des erreurs.
- **Lisibilité** : Améliore la lisibilité du code en réduisant le nombre de blocs `try-catch` imbriqués.

## 9 Assertions

- Les assertions sont des outils de vérification utilisés pour tester les hypothèses faites par le programmeur.
- Elles permettent de détecter les erreurs de logique dans le code pendant le développement et les tests.
- Utilisation de l'instruction `assert` :

```
assert condition : message;
```

### 9.1 Exemple d'utilisation

- Vérification d'une condition simple :

```
int age = -1;
assert age ⩾ 0 : "L'âge ne peut pas être négatif : " + age;
System.out.println("L'âge est : " + age);
```

- Par défaut, les assertions sont désactivées à l'exécution.
- Pour les activer, utilisez l'option `-ea` (ou `-enableassertions`) lors de l'exécution du programme :

```
java -ea AssertionExample
```

### 9.2 Meilleures Pratiques

- **Utiliser des exceptions spécifiques** : Capturer des exceptions spécifiques plutôt qu'une exception générique.
- **Éviter de masquer les exceptions** : Ne pas capturer une exception sans la gérer.
- **Journaliser les exceptions** : Toujours journaliser les exceptions pour le débogage.
- **Nettoyer les ressources** : Utiliser le bloc `finally` ou try-with-resources pour nettoyer les ressources.
- **Documenter les exceptions** : Utiliser Javadoc pour documenter les exceptions qu'une méthode peut lancer.

## 10 Exemple 1 : Gestion des Entrées/Sorties de Fichiers

```
public class FileHandler {
    public void readFile(String filePath) {
      try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
{
            String line;
            while ((line = reader.readLine()) ≠ null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.err.println("Fichier non trouvé : " + e.getMessage());
        } catch (IOException e) {
          System.err.println("Erreur de lecture du fichier : " + e.getMessage());
        }
    }
}
```

### 10.1 Exemple 2 : Exception Personnalisée

```
public class AgeValidator {
    public void validateAge(int age) throws InvalidAgeException {
```

```
        if (age < 0 || age > 150) {
            throw new InvalidAgeException("Âge invalide : " + age);
        }
    }
}

public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

### 10.2 Exemple 3 : Utilisation de try-with-resources

```
import java.sql.*;
```

```
    public void connectToDatabase() {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String user = "root";
        String password = "password";

       try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement()) {
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
            while (rs.next()) {
                System.out.println(rs.getString("column1"));
            }
        } catch (SQLException e) {
           System.err.println("Erreur de base de données : " + e.getMessage());
        }
    }
```

## 11 Conclusion

- La gestion des exceptions est cruciale pour construire des applications robustes et sécurisées.
- Java fournit un mécanisme complet pour gérer les exceptions.
- Suivre les meilleures pratiques assure un code maintenable et lisible.
- Références
  - Documentation Java sur les Exceptions