Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG
DUT- IDIA 1 - S2
M122 - Ingénierie Logicielle avec Java et UML
Année universitaire: 2024-2025

Support de cours

Ingénierie logicielle avec Java et UML

Partie I: Programmation Orientée Objet en Java

Chapitre III: Gestion des exceptions

Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- **INTRODUCTION**
- **CONCEPT DE BASE**
- **MANIPULATION DES EXCEPTIONS**
- **PERSONNALISATION DES EXCEPTIONS**

Introduction

- Les **exceptions** sont des éléments fondamentaux en programmation, représentant des erreurs pouvant interrompre le déroulement normal d'un programme.

Exemple:

Division par zéro : Lorsqu'une division par zéro est tentée, elle génère une exception **`ArithmeticException`**.

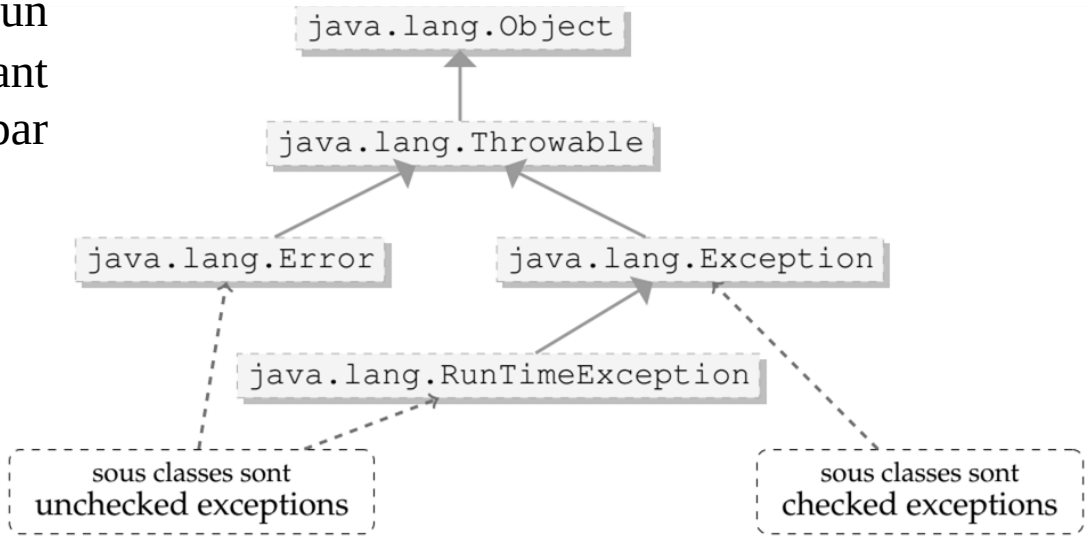
```
int numerator = 10;
int denominator = 0;

try {
    // Cette ligne provoquera une exception
    int result = numerator / denominator;
} catch (ArithmeticException e) {
    System.out.println("Erreur : Division par zéro !");
}
```

→ Gestion des erreurs

- **`Java`** possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la sécurité du code. On peut avoir différents niveaux de problèmes :

- L'erreur (**`Error`**) représente un dysfonctionnement grave survenant dans la machine virtuelle (par exemple, **OutOfMemoryError**).

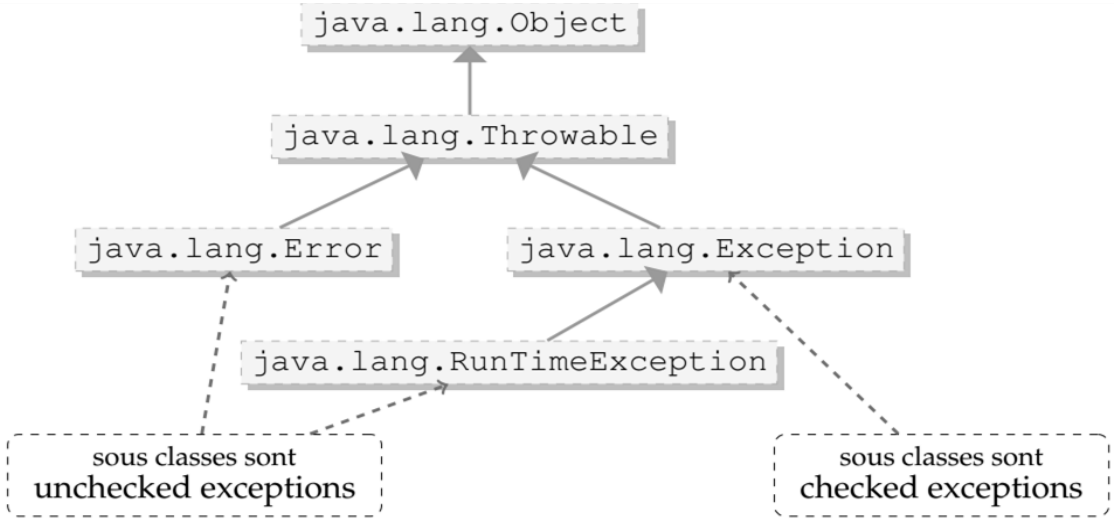
- La classe **`Exception`** quant à elle représente des anomalies ou des situations imprévues dans le programme, lesquelles sont signalées au développeur.



- Le développeur a la possibilité de **gérer** ces erreurs (**`Exception`**) et ainsi **éviter** une cessation soudaine de l'application.
- Nous souhaitons principalement gérer les erreurs **anticipées**, celles que l'on peut prévoir et traiter dans le code.

Concept de base

→ Gestion des erreurs



- Les exceptions vérifiées (**`checked Exception`**) représentent des situations où l'on peut anticiper le problème, telles que les opérations d'entrée-sortie (**`IOException`**), nécessitant une gestion explicite dans le code.
- Les **`RunTimeException`** ne peuvent pas être détectées lors de la compilation. Elles surviennent généralement en raison d'erreurs de programmation telles que **`NullPointerException`** et diffèrent des erreurs provoquées par des sources incontrôlables comme les opérations d'entrée-sortie (**`IOException`**).

Concept de base

→ **Gestion des erreurs**

- Les exceptions sont à utiliser pour des conditions véritablement exceptionnelles. Les exceptions vérifiées (**checked exception**) correspondent à des situations pour lesquelles on peut raisonnablement espérer pouvoir récupérer et continuer l'exécution du programme.

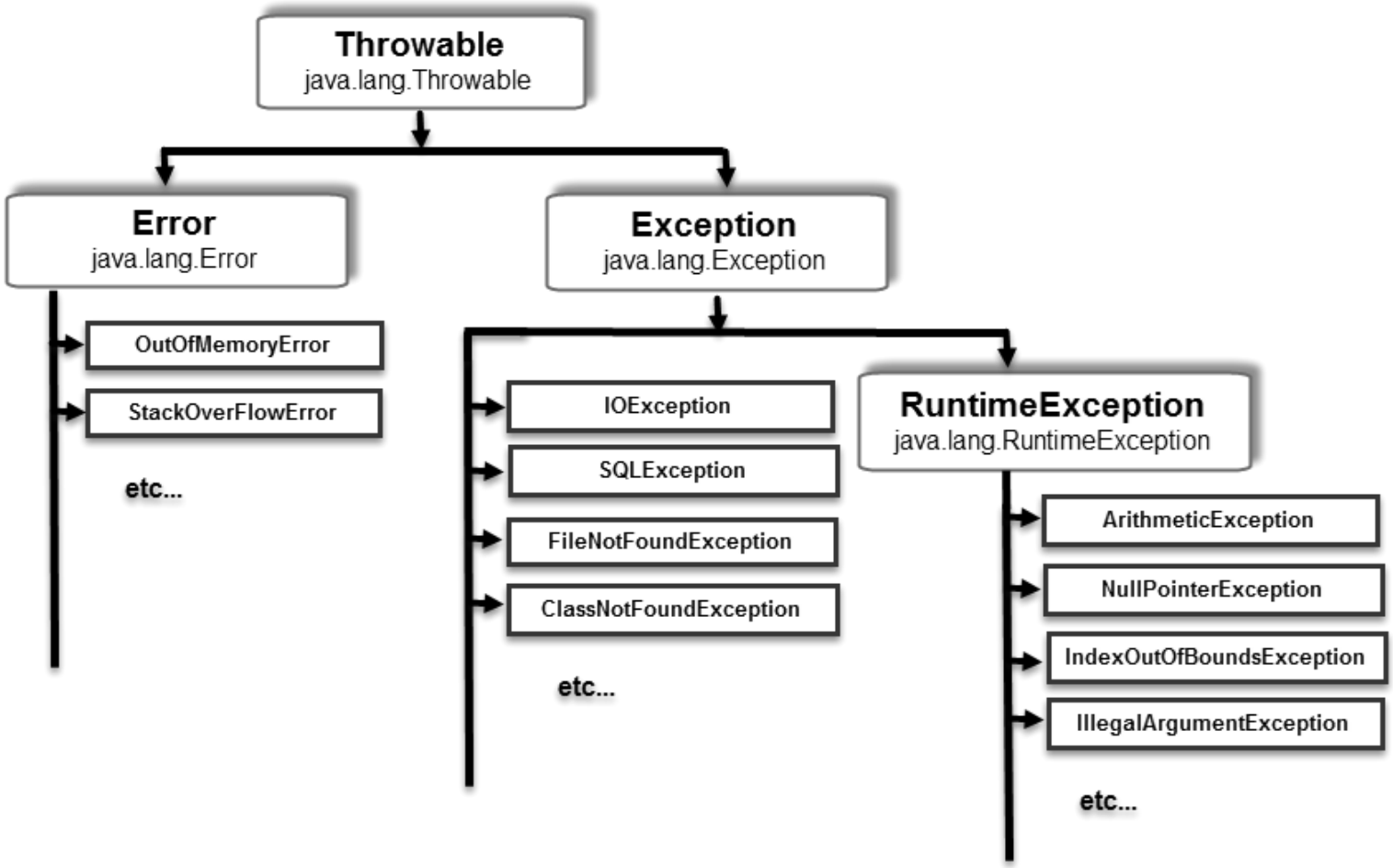
➔ Ces exceptions obligent le développeur à gérer des situations exceptionnelles, l'empêchant ainsi de les ignorer.

➔ Les "**exceptions d'utilisation**" doivent être déclarées dans la signature de la fonction si elles ne sont pas capturées à l'intérieur de la fonction.

- Les **`RunTimeException`** signalent généralement des erreurs de programmation. Par exemple, **`NullPointerException`** est une exception "**unchecked**" (***non vérifiée***). Si elle n'est pas traitée spécifiquement, elle peut potentiellement survenir à n'importe quel endroit du code.

Concept de base

→ Utilisation de la Bibliothèque Standard pour la Gestion des Exceptions



→ Utilisation de la Bibliothèque Standard pour la Gestion des Exceptions

- **`IllegalArgumentException`** : Correspond à un argument ayant une valeur inappropriée.
- **`IllegalStateException`** : Indique que l'objet n'est pas dans un état adéquat pour effectuer l'appel.

Il est possible d'utiliser principalement ces deux exceptions, car la plupart des erreurs sont liées à un argument ou à un état illégal :

- **`NullPointerException`**
- **`IndexOutOfBoundsException`**
- **`ArithmeticException`**
- **`NumberFormatException`**

Cependant, déterminer quelles exceptions utiliser n'est pas une science exacte

→ Utilisation de la Bibliothèque Standard pour la Gestion des Exceptions

Exemple

Supposons un programme de gestion d'inventaire pour un magasin. Imaginons qu'il comporte une fonctionnalité permettant l'ajout d'articles au stock. Cette fonction, nommée ***ajouterArticle***, prend en paramètres le nom de l'article (***String nomArticle***) ainsi que la quantité d'articles à ajouter (***int quantite***).

```
public void ajouterArticle(String nomArticle, int quantite);
```

- **`IllegalArgumentException`** pourrait être levée si la quantité passée en paramètre est négative.

- **`IllegalStateException`** pourrait être déclenchée si la fonction est appelée alors que la connexion à la base de données ou une ressource externe nécessaire n'est pas établie correctement

→ Déclaration des Exceptions

- La clause **`throws`** est utilisée pour déclarer une méthode dans laquelle l'échec peut être anticipé.

Exemple:

```
public void write(Object obj, String nomFichier)
    throws IOException, ReflectiveOperationException {
    /* Code de la méthode susceptible
        de lancer IOException ou ReflectiveOperationException */
}
```

→ Dans cette déclaration, il est possible de regrouper les exceptions en utilisant une superclasse commune ou de les énumérer individuellement.

→ Dans l'exemple, nous déclenchons des exceptions de type **`IOException`**. Toutefois, il est également envisageable de spécifier de manière plus précise tous les types d'exceptions possibles, notamment les sous-classes spécifiques de **`IOException`**."

→ Lever une exception

Lorsqu'une erreur est détectée dans le code,

- → un objet héritant de la classe **Exception** est créé, ce processus étant appelé "**lever une exception**".
- → L'exception créée est propagée à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.

Exemple 1

```
int[] tableau = new int[5];
tableau[5] = 0;
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
At Personne.main(Personne.java:2)
```

- **`ArrayIndexOutOfBoundsException`** : Se produit lorsqu'on tente d'accéder à un index non valide dans un tableau.

→ Lever une exception

Exemple 2

```
int d = 10, t1 = 5, t2 = 5;
System.out.println("vitesse : " + d / (t2 - t1));
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Personne.main(Personne.java:21)
```

- **`ArithmeticException` :** Se produit lors d'une opération arithmétique invalide, comme une division par zéro.

Ces exemples illustrent comment les exceptions, telles que **ArrayIndexOutOfBoundsException** et **ArithmeticException**, sont générées et signalées dans le code Java, fournissant des informations sur leur origine dans la **pile d'exécution** (**stack trace**)."

La **pile d'exécution**, ou **stack trace**, est une structure de données qui enregistre l'ordre chronologique des appels de fonctions ou de méthodes dans un programme en cours d'exécution.

Manipulation des Exceptions

→ **Le bloc** *try … catch - finally*

- Le bloc **`try`** englobe le code pouvant générer des erreurs.
- Les exceptions sont récupérées et gérées via le bloc **`catch`**.
- Plusieurs blocs **`catch`** peuvent être utilisés pour attraper différents types d'erreurs.
- En option, le bloc **`finally`** est ajouté pour exécuter des instructions, qu'une exception ait été levée ou non.
- Lorsqu'une erreur survient dans le bloc **`try`**, le reste du code du bloc est abandonné.
- Les blocs **`catch`** sont testés séquentiellement, et le premier bloc correspondant à l'erreur est exécuté.
- L'ordre des blocs **`catch`** doit aller de l'erreur la plus spécifique à la plus générale, établissant ainsi une hiérarchie d'exceptions à attraper.

→ Ce mécanisme permet de **filtrer** et de **gérer** les exceptions de manière ordonnée et adaptée à chaque type d'erreur.

→ Le bloc *try … catch - finally*

Exemple

```
public class GInfo {
    public static void main(String[] args) {
        try {
            // Code susceptible de générer une exception
            int[] tableau = new int[5];
        // Cette ligne provoquera ArrayIndexOutOfBoundsException
            tableau[5] = 10;
        } catch (ArrayIndexOutOfBoundsException e) {
            // Capture et gestion de l'exception spécifique
            System.out.println("Erreur : Accès à un index non valide dans le tableau !");
        } catch (Exception e) {
            // Capture de toute autre exception non spécifiée
            System.out.println("Une erreur s'est produite !");
        } finally {
            // Bloc finally : exécuté que l'exception soit levée ou non
            System.out.println("Fin du programme");
        }
    }
}
```

Les instructions présentes dans le bloc **`finally`** sont toujours exécutées, même si une exception est levée dans le bloc **`try`**. Cela est utile pour effectuer des actions qui doivent être réalisées quelle que soit la situation, telles que la fermeture de fichiers, la libération de ressources, ou des opérations de nettoyage.

→ **Le bloc** *try … catch - finally*

Une méthode peut lever plusieurs exceptions :

- Il peut être tentant de les regrouper dans une classe mère d'exception, mais cela entraîne une perte partielle d'informations spécifiques à chaque type d'erreur.
- Il est possible de déclarer chaque exception individuellement, bien que le langage ne requière pas explicitement cette déclaration pour chaque exception potentielle du code.
- Cette pratique encourage le développeur à être plus attentif à certains aspects du code.
- La balise Javadoc '@throws' permet de documenter spécifiquement chaque type d'exception qu'une méthode peut lever, fournissant ainsi une documentation claire sur les erreurs possibles.
- Depuis **Java 7**, il est possible de gérer plusieurs exceptions dans une seule clause `catch` à l'aide de la syntaxe

```
catch(ExceptionType1 | ExceptionType2 | ExceptionType3 e)  {... }
```

→ **Le bloc** *try … catch - finally*

Exemple

```
int d = 10, t1 = 5, t2 = 5;
try {
    System.out.println("vitesse: " + d / (t2 - t1));
} catch (ArithmeticException e) {
    System.out.println("Vitesse non valide");
} catch (Exception e) {
    e.printStackTrace();
}
```

Une tentative de division (d / (t2 - t1)) est effectuée.

→ Si une **`ArithmeticException`** est levée (dans ce cas, une division par zéro), le premier bloc **`catch`** sera exécuté pour afficher "*Vitesse non valide*".

→ Si une autre exception est levée, le deuxième bloc **`catch`** utilisera la méthode ***printStackTrace***() pour afficher les détails de cette exception dans la sortie d'erreur.

Personnalisation des exceptions

→ Créer une Exception Personnalisée

- Pour créer une **exception personnalisée** en Java, vous pouvez définir votre propre classe qui étend une classe d'exception existante, comme **`Exception`** ou l'une de ses sous-classes.

Exemple (1/2)

```
// Création d'une exception personnalisée en étendant la classe Exception
class MonExceptionPersonnalisee extends Exception {
    // Constructeur prenant un message d'erreur en paramètre
    public MonExceptionPersonnalisee(String message) {
        super(message);
    }
}
```

- Dans cet exemple, **`MonExceptionPersonnalisee`** étend la classe **`Exception`**. Vous pouvez personnaliser cette classe en ajoutant des attributs ou des méthodes supplémentaires selon vos besoins.
- Le constructeur ***MonExceptionPersonnalisee*** prend en paramètre un message d'erreur, qui sera utilisé pour décrire l'exception lorsqu'elle est levée.

Personnalisation des exceptions

→ Créer une Exception Personnalisée

Exemple (2/2) : Voici comment utiliser cette exception personnalisée :

```
class ExempleUtilisation {
    public static void main(String[] args) {
        try {
            // Simuler le déclenchement de l'exception personnalisée
            throw new MonExceptionPersonnalisee("Erreur personnalisée !");
        } catch (MonExceptionPersonnalisee e) {
            // Attraper et gérer l'exception personnalisée
            System.out.println("Exception attrapée : " + e.getMessage());
        }
    }
}
```

En utilisant **`throw new`** ***MonExceptionPersonnalisee***(*"Erreur personnalisée !"*); , vous pouvez déclencher votre exception personnalisée. Ensuite, avec un bloc **`catch`**, vous pouvez l'attraper et afficher son message.