



Introduction à la programmation et au développement d'applications en Java

Exceptions et gestion d'erreurs

Introduction

- ❑ Le rôle des exceptions
- ❑ L'instruction `try/catch/finally`
- ❑ Responsabilités de gestion d'exceptions
- ❑ Lancer une exception (*throw*)
- ❑ Définir nos propres types d'exceptions

Gestion d'erreurs avec Exceptions

- ❑ La plus géniale application du monde n'aura aucun succès si elle est bourrée d'erreurs et « crashe » toutes les trois minutes
- ❑ La **gestion d'erreurs** doit être une partie intégrante du développement
- ❑ Avant : codes d'erreurs/*flags* qui devaient être posés et vérifiés partout où du code pouvait potentiellement générer une erreur
- ❑ Le principe des **exceptions** est au contraire **non-intrusif**
- ❑ On écrit le code **comme si tout se passait bien**, et si une erreur est rencontrée, une **exception est lancée**, et il faut alors **la récupérer**
- ❑ C'est sur ce principe que fonctionne l'instruction **try/catch** : on *essaye* (*try*) d'exécuter le code en conditions normales, et en cas de problème on *récupère* (*catch*) l'exception lancée et on la traite

try/catch/finally

- ❑ `try { ... }`
 - ❑ contient le code « normal », ce qu'on veut faire
 - ❑ s'exécute jusqu'à la fin sauf si une exception est lancée
- ❑ `catch (Exception e) { ... }`
 - ❑ contient le code de gestion de l'erreur éventuellement rencontrée
 - ❑ s'exécute seulement si une exception du type indiqué est rencontrée
- ❑ `finally { ... }`
 - ❑ contient le code de « nettoyage »
 - ❑ s'exécute **toujours**, qu'une exception ait été lancée ou non

try/catch/finally: principe

```
int a = 12;  
int b = 2;  
  
int res = a / (b - 2);  
System.out.println(res);
```

try/catch/finally : principe

```
int a = 12;
int b = 2;

try {
    int res = a / (b - 2);
    System.out.println(res);
}
catch (Exception e) {
    System.out.println("Erreur : " + e.getMessage());
    e.printStackTrace();
}
```

Exemple de lecture de fichier

```
Buffered reader = null;  
int total = 0;  
  
    reader = new BufferedReader(new FileReader("c:\\nombres.txt"));  
    String ligne = null;  
    while ((ligne = reader.readLine()) != null)  
        total += Integer.valueOf(ligne);  
    System.out.println("Total = " + total);
```

Exemple de lecture de fichier

```
Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (Exception e) {
    System.out.println("Erreur : " + e.getMessage());
}
```


Exemple de lecture de fichier

```
Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (Exception e) {
    System.out.println("Erreur : " + e.getMessage());
} finally {

    if (reader != null)
        reader.close();

}
```

Exemple de lecture de fichier

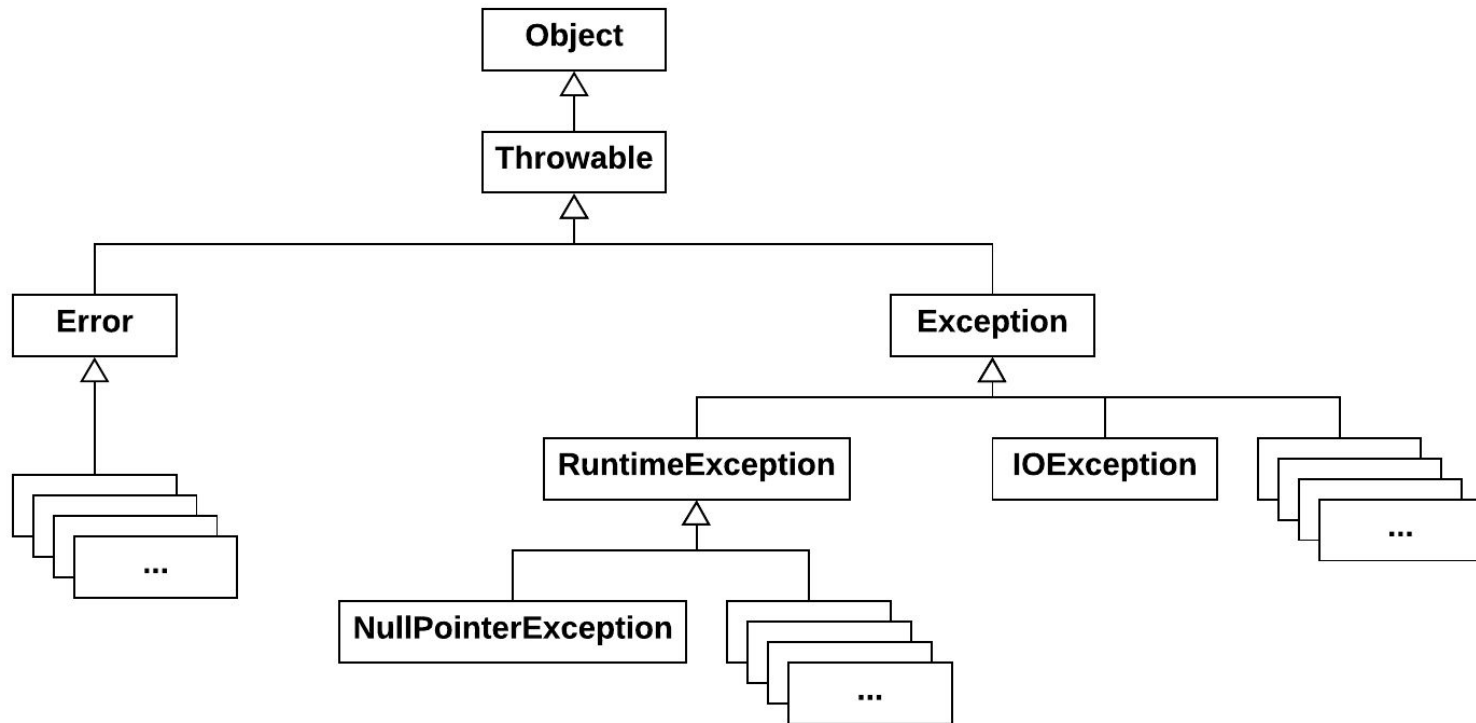
```
Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (Exception e) {
    System.out.println("Erreur : " + e.getMessage());
} finally {
    try {
        if (reader != null)
            reader.close();
    } catch (Exception e) {
        System.out.println("Erreur fermeture : " + e.getMessage());
    }
}
```

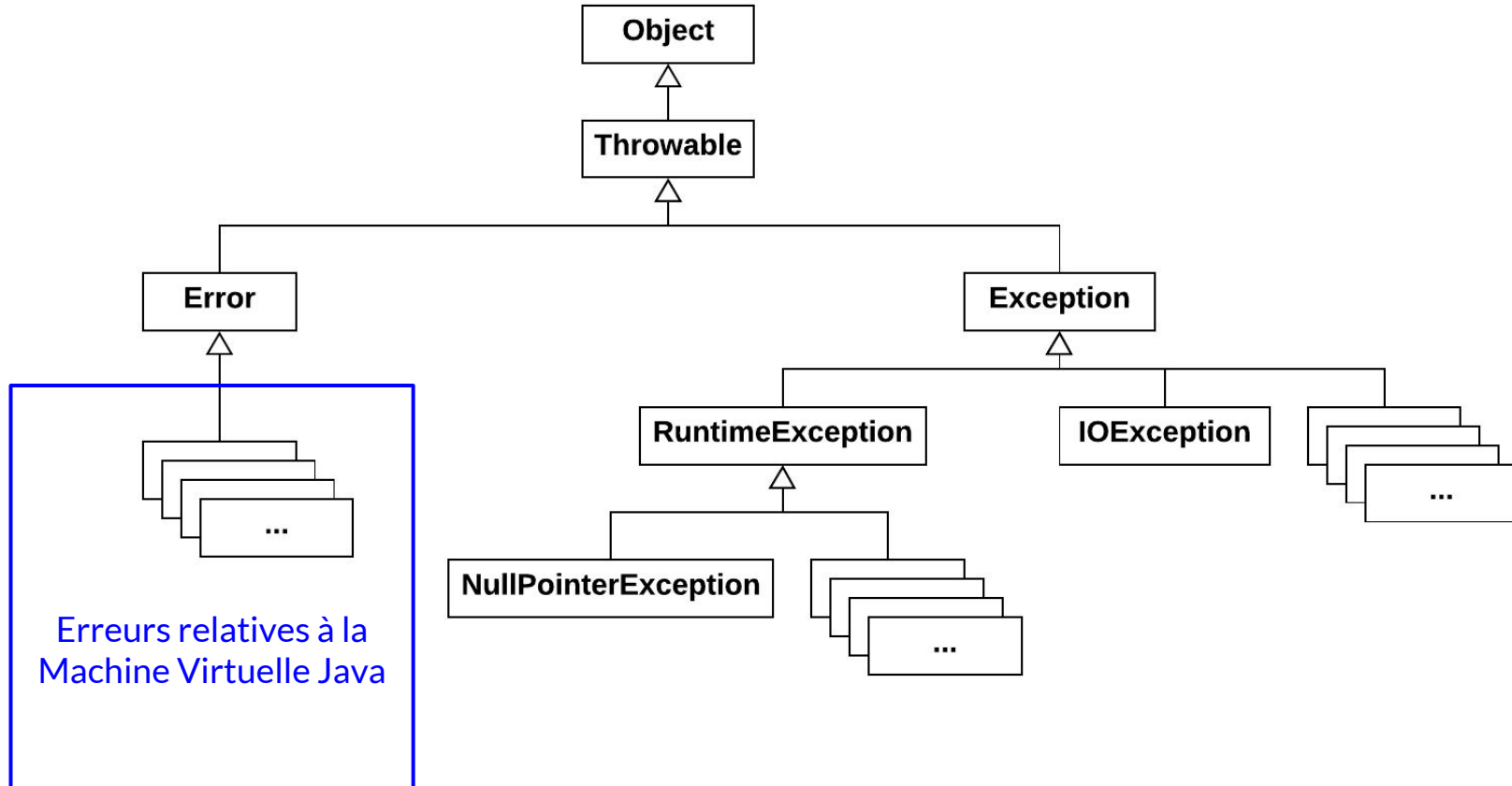
Hiérarchie des classes d'erreurs

- ❑ **Une exception est un objet** (on peut appeler par exemple `getMessage()` ou `printStackTrace()` dessus)
- ❑ Et donc, chaque type d'exception est décrit dans une classe
- ❑ Ces classes font partie d'une **hiérarchie de classes** Java, qui divise les exceptions en différentes « catégories »

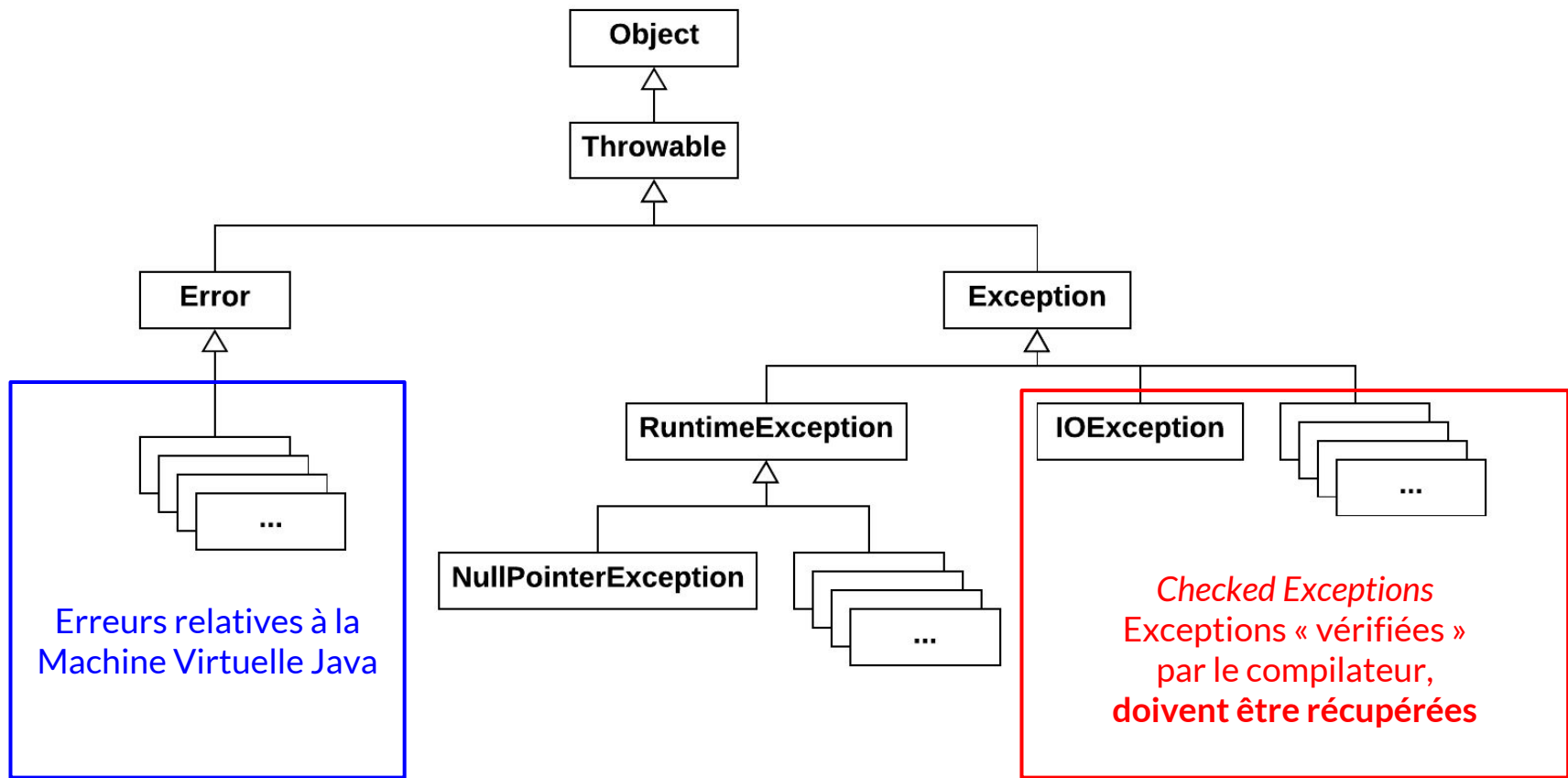
Hiérarchie des classes d'erreurs



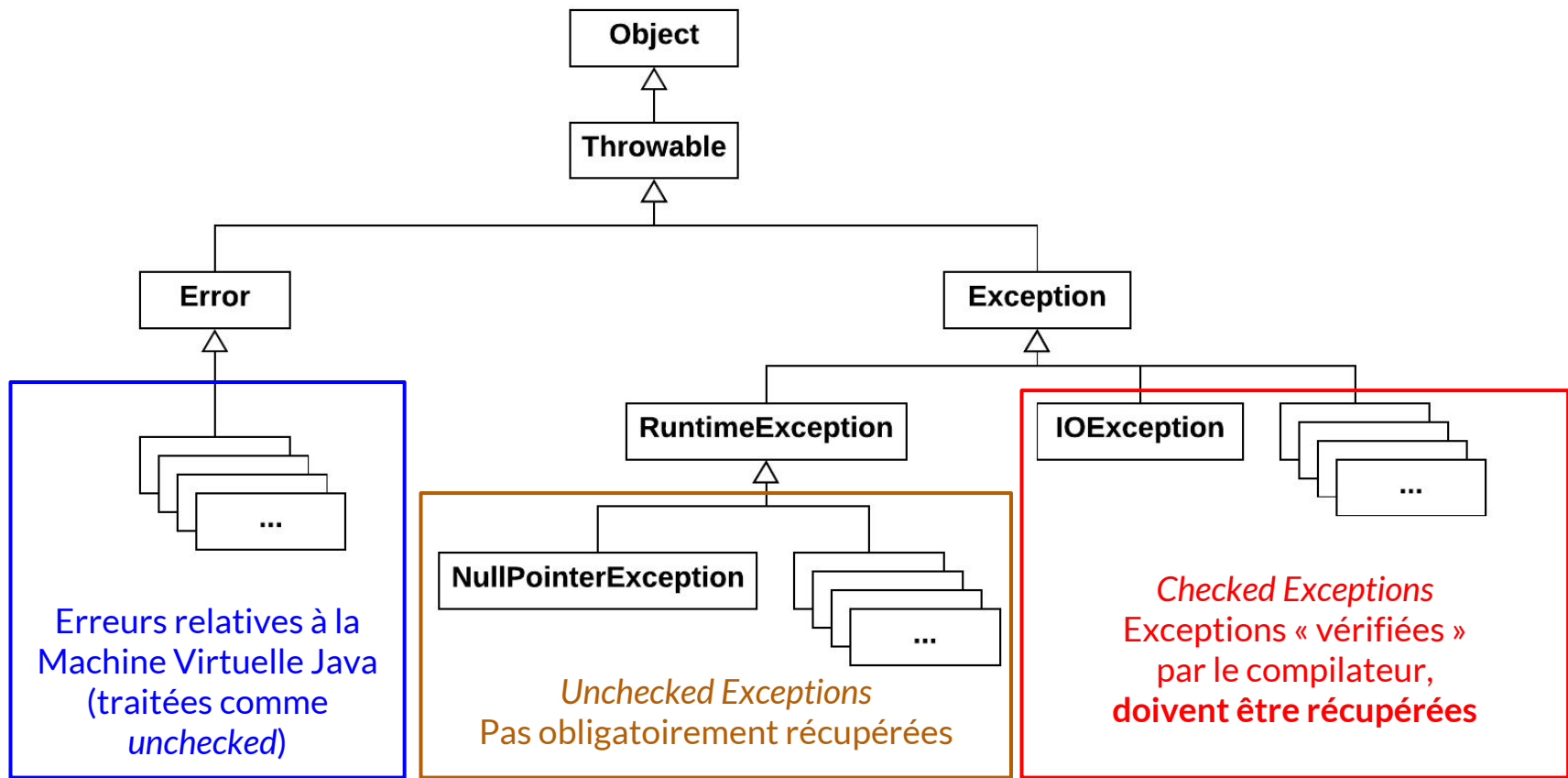
Hiérarchie des classes d'erreurs



Hiérarchie des classes d'erreurs



Hiérarchie des classes d'erreurs



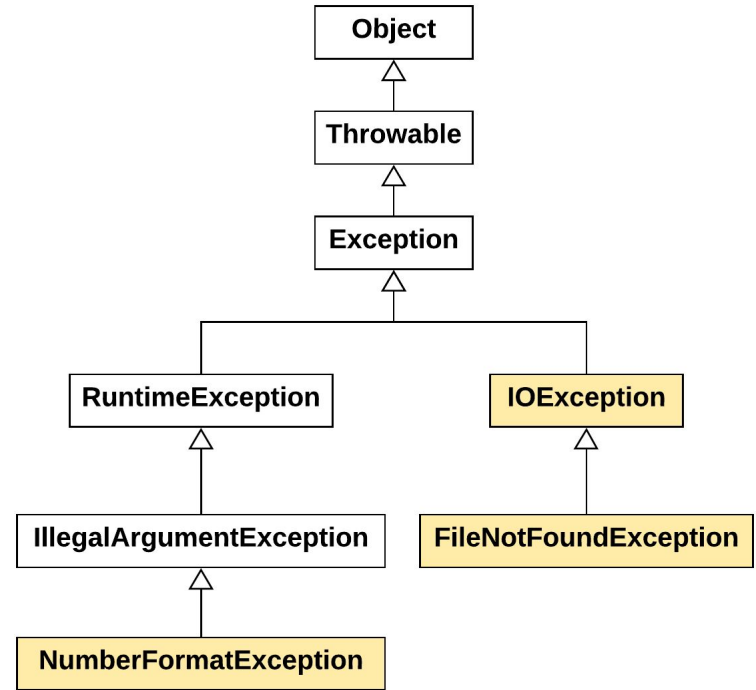
Exceptions typées

- ❑ Les exceptions sont **récupérées par types** :
 - ❑ chaque type d'exception pourra avoir **son propre bloc `catch`**
 - ❑ chaque bloc **`catch`** est testé, **dans l'ordre donné**, pour voir s'il correspond à l'exception lancée
 - ❑ **le premier qui correspond** (même s'il y en a plusieurs) **est sélectionné**, et les autres sont ignorés
- ❑ Il faut donc spécifier **en premier** les types d'exceptions **les plus spécifiques** (les classes les plus basses dans la hiérarchie d'héritage)


```
Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(
        new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (Exception e) {
    System.out.println("Erreur : "
        + e.getMessage());
} catch (NumberFormatException e) {
    System.out.println("Valeur invalide : "
        + e.getMessage());
}

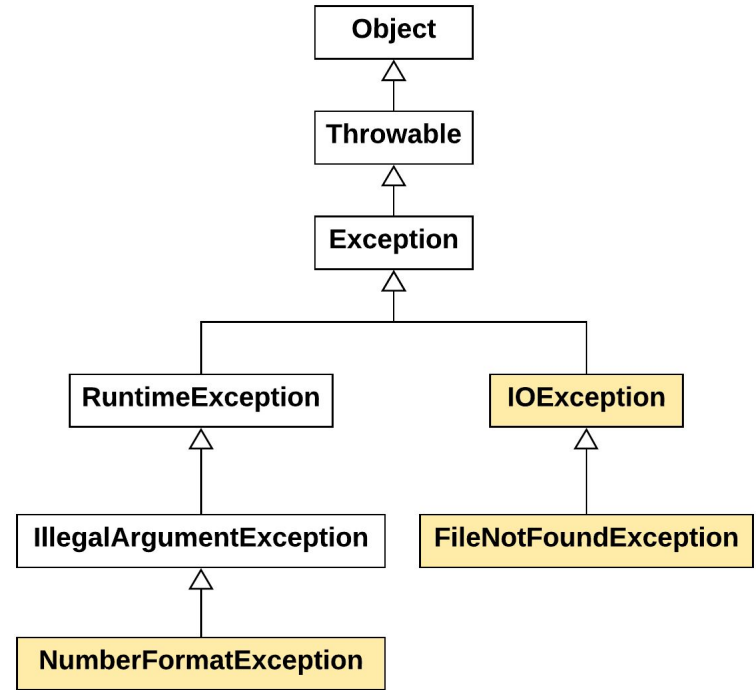
finally {
    ...
}
```



```
Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(
        new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (NumberFormatException e) {
    System.out.println("Valeur invalide : "
        + e.getMessage());
} catch (Exception e) {
    System.out.println("Erreur : "
        + e.getMessage());
}

finally {
    ...
}
```



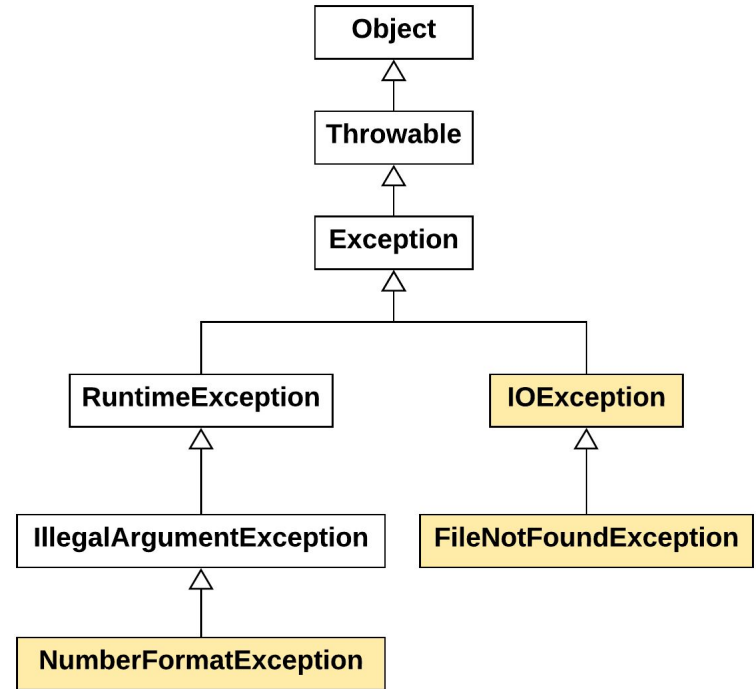
```

Buffered reader = null;
int total = 0;

try {
    reader = new BufferedReader(
        new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (NumberFormatException e) {
    System.out.println("Valeur invalide : "
        + e.getMessage());
} catch (FileNotFoundException e) {
    System.out.println("Fichier non trouvé : "
        + e.getMessage());
}

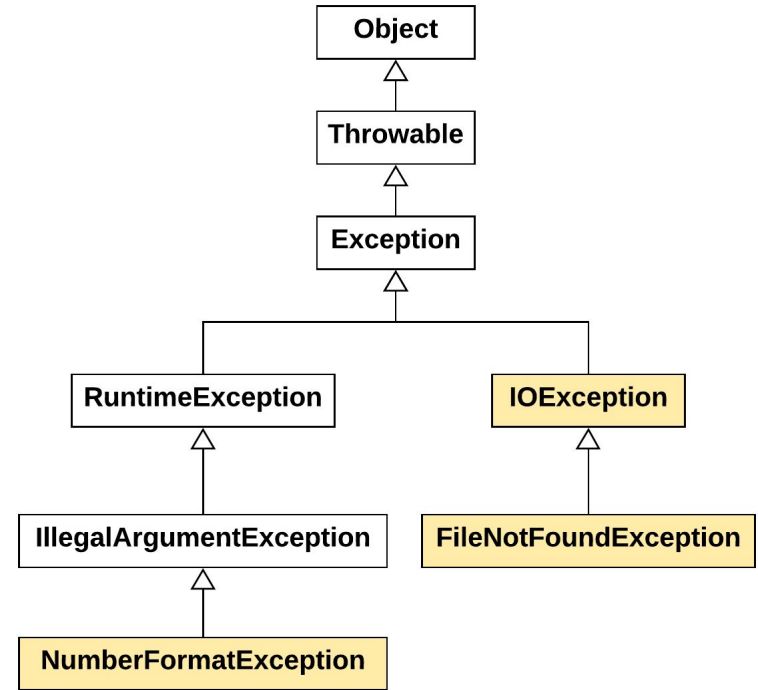
finally {
    ...
}

```



```
BufferedReader reader = null;
int total = 0;

try {
    reader = new BufferedReader(
        new FileReader("c:\\nombres.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null)
        total += Integer.valueOf(ligne);
    System.out.println("Total = " + total);
} catch (NumberFormatException e) {
    System.out.println("Valeur invalide : "
        + e.getMessage());
} catch (FileNotFoundException e) {
    System.out.println("Fichier non trouvé : "
        + e.getMessage());
} catch (IOException e) {
    System.out.println("Erreur d'accès : "
        + e.getMessage());
} finally {
    ...
}
```



Propagation d'exceptions

- ❑ Disons que la méthode `main` appelle la méthode A qui appelle B qui appelle C
- ❑ Une exception est lancée pendant l'exécution de C
- ❑ Java va rechercher (en remontant la pile d'appels) **la première méthode qui indique pouvoir récupérer ce type d'exception** précis (ou un de ses parents)
- ❑ Il va donc chercher un bloc `catch` correspondant dans C, puis dans B, puis dans A et enfin dans `main`
- ❑ Si aucun bloc **`catch`** correspondant n'est trouvé, **le programme s'arrête** en indiquant l'erreur ; c'est une très mauvaise expérience utilisateur

Propagation d'exceptions

- ❑ Une exception peut donc traverser les « frontières » des méthodes en remontant la pile d'appels
- ❑ Comment la méthode `main` sait-elle qu'elle pourrait être responsable de récupérer une exception lancée dans C, trois niveaux d'appels plus bas ?
- ❑ En Java, les exceptions que lance une méthode font partie du **contrat** de la méthode, au même titre que sa signature
- ❑ Une méthode est **responsable de toute « exception vérifiée »** (*checked exception*) qui pourrait se produire ; elle a alors deux options :
 - ❑ *récupérer* l'exception (clause **catch**)
 - ❑ *déclarer* qu'une exception peut se produire (clause **throws**) et déléguer la responsabilité en laissant l'exception remonter la pile d'appel

```
public class Vol {  
    private int nbPassagers;  
    // autre membres...  
  
    public void ajouterPassagers(String fichier) {  
        BufferedReader reader = null;  
  
        reader = new BufferedReader(new FileReader(fichier));  
        String ligne = null;  
        while ((ligne = reader.readLine()) != null) {  
            String[] parties = ligne.split(" ");  
            nbPassagers += Integer.valueOf(parties[0]);  
        }  
  
    }  
}
```

ListePassagers.txt

```
2 Durand  
5 Dupont  
4 Dupuis  
7 Dupond  
...
```

```
public class Vol {  
    private int nbPassagers;  
    // autre membres...
```

```
    public void ajouterPassagers(String fichier) {  
        BufferedReader reader = null;
```

```
        reader = new BufferedReader( new FileReader(fichier) );
```

```
        String ligne = null;
```

```
        while ((ligne = reader.readLine()) != null) {
```

```
            String[] parties = ligne.split(" ");
```

```
            nbPassagers += Integer.valueOf(parties[0]);
```

```
        }
```

```
    }
```

```
}
```

FileNotFoundException



IOException


```
public class Vol {  
    private int nbPassagers;  
    // autre membres...  
  
    public void ajouterPassagers(String fichier) throws IOException {  
        BufferedReader reader = null;  
  
        reader = new BufferedReader( new FileReader(fichier) );  
        String ligne = null;  
        while ((ligne = reader.readLine()) != null) {  
            String[] parties = ligne.split(" ");  
            nbPassagers += Integer.valueOf(parties[0]);  
        }  
  
    }  
}
```

```
public class Vol {  
    private int nbPassagers;  
    // autre membres...  
  
    public void ajouterPassagers(String fichier) throws IOException {  
        BufferedReader reader = null;  
        try {  
            reader = new BufferedReader(new FileReader(fichier));  
            String ligne = null;  
            while ((ligne = reader.readLine()) != null) {  
                String[] parties = ligne.split(" ");  
                nbPassagers += Integer.valueOf(parties[0]);  
            }  
        } finally {  
            if (reader != null)  
                reader.close();  
        }  
    }  
}
```

```
public class Vol {  
    private int nbPassagers;  
    // autre membres...  
  
    public void ajouterPassagers(String fichier) throws IOException {  
        BufferedReader reader = null;  
        try {  
            reader = new BufferedReader(new FileReader(fichier));  
            String ligne = null;  
            while ((ligne = reader.readLine()) != null) {  
                String[] parties = ligne.split(" ");  
                nbPassagers += Integer.valueOf(parties[0]);  
            }  
        } catch (NumberFormatException e) {  
            System.out.println("Mauvais format de fichier : "  
                + e.getMessage());  
        } finally {  
            if (reader != null)  
                reader.close();  
        }  
    }  
}
```

Exceptions et redéfinition de méthodes

- ❑ La clause **throws** d'une méthode redéfinie (*overridden*) doit être **compatible** avec la clause **throws** de la méthode de base (*overriding*)
- ❑ La méthode redéfinie est **compatible** si elle ne peut pas lancer d'exception qui n'était pas déjà « connue » de la méthode de base. Trois cas possibles ; elle lance :
 - ❑ aucune exception (pas de clause **throws**)
 - ❑ les **mêmes exceptions** que la méthode de base
 - ❑ des **exceptions dérivées** des exceptions déclarées dans la méthode de base
- ❑ Ex.: `ajouterDesPassagers(String fichier) throws IOException`
ou bien : `throws FileNotFoundException`

Lancer des exceptions

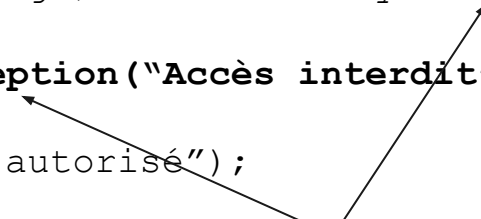
- ❑ En plus de simplement récupérer des exceptions lancées, notre code peut lui-même **lancer des exceptions** (mot-clé **throw**) :
 - ❑ parce qu'on a détecté une erreur
 - ❑ `throw new IndexOutOfBoundsException()`
 - ❑ parce qu'on a récupéré une exception et qu'on veut en relancer une différente qui fournirait plus d'informations de contexte
 - ❑ `throw new IllegalArgumentException("Le tableau rib[] devrait comporter exactement cinq éléments", e);`
- ❑ Une exception est un **objet**, elle doit donc être **instanciée avant d'être lancée**
- ❑ On fournit les détails pertinents qu'on **encapsule** dans l'exception (celui qui récupère l'exception doit avoir un maximum d'informations sur ce qui s'est produit)

Lancer des exceptions

- ❑ Comment encapsuler les détails ?
 - ❑ La plupart des classes d'exceptions fournissent un constructeur qui accepte un **message** dans lequel on va pouvoir inclure ce qu'on veut (string)
 - ❑ On peut également **relancer une exception existante en l'encapsulant dans un autre type d'exception** qui va inclure des informations supplémentaires
 - ❑ il faut alors appeler la méthode **`initCause()`** pour indiquer l'exception d'origine (toutes les classes d'exceptions fournissent **`initCause()`**)
 - ❑ certaines classes disposent même d'un constructeur qui prendra directement l'exception d'origine

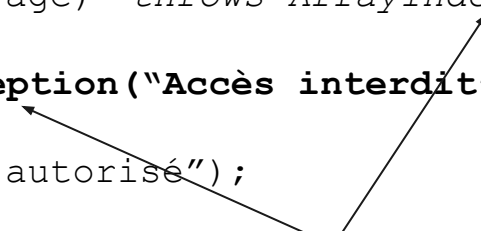
```
public void verifierAge(int age) throws ArrayIndexOutOfBoundsException {  
    if (age < 18) {  
        throw new ArithmeticException("Accès interdit");  
    }  
    System.out.println("Accès autorisé");  
}
```

```
public void verifierAge(int age) throws ArrayIndexOutOfBoundsException {  
    if (age < 18) {  
        throw new ArithmeticException("Accès interdit");  
    }  
    System.out.println("Accès autorisé");  
}
```



ArrayIndexOutOfBoundsException est *unchecked*, donc la clause **throws** est facultative, et même déconseillée. En fait, vous devriez traiter les exceptions *unchecked* dérivées de **RuntimeException**, qui sont de votre responsabilité, le plus vite possible.

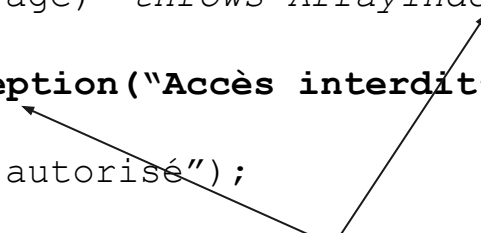

```
public void verifierAge(int age)  throws ArrayIndexOutOfBoundsException {  
    if (age < 18) {  
        throw new ArithmeticException("Accès interdit");  
    }  
    System.out.println("Accès autorisé");  
}
```



ArrayIndexOutOfBoundsException est *unchecked*, donc la clause **throws** est facultative, et même déconseillée. En fait, vous devriez traiter les exceptions *unchecked* dérivées de **RuntimeException**, qui sont de votre responsabilité, le plus vite possible.

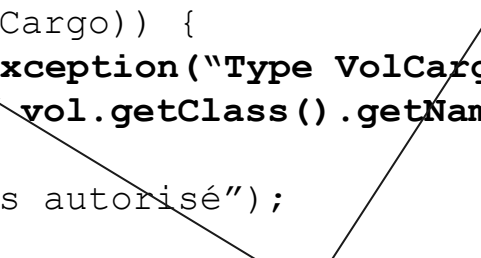
```
public boolean attribuer(Vol vol)  throws InvalidTypeException {  
    if (!(vol instanceof VolCargo)) {  
        throw new InvalidTypeException("Type VolCargo attendu, mais  
l'argument est de type " + vol.getClass().getName());  
    }  
    System.out.println("Accès autorisé");  
}
```

```
public void verifierAge(int age) throws ArrayIndexOutOfBoundsException {  
    if (age < 18) {  
        throw new ArithmeticException("Accès interdit");  
    }  
    System.out.println("Accès autorisé");  
}
```



ArrayIndexOutOfBoundsException est *unchecked*, donc la clause **throws** est facultative, et même déconseillée. En fait, vous devriez traiter les exceptions *unchecked* dérivées de **RuntimeException**, qui sont de votre responsabilité, le plus vite possible.

```
public boolean attribuer(Vol vol) throws InvalidTypeException {  
    if (!(vol instanceof VolCargo)) {  
        throw new InvalidTypeException("Type VolCargo attendu, mais  
l'argument est de type " + vol.getClass().getName());  
    }  
    System.out.println("Accès autorisé");  
}
```



InvalidTypeException est *checked*, donc la clause **throws InvalidTypeException** est obligatoire.

Créer son propre type d'exception

- ❑ Dans la plupart des cas, on lance des exceptions existantes dans la JCL
- ❑ Mais vous pouvez **créer vos propres types d'exceptions** dans le cas où vous avez des informations à propager qui ne s'encapsulent pas aisément dans les types prédéfinis
- ❑ En général on dérive alors directement la classe **Exception**, ce qui rend automatiquement nos exceptions *checked* par le compilateur
- ❑ En général toujours, les seuls membres qu'on va y définir sont des constructeurs et des attributs (les autres fonctionnalités - *message*, *stack trace*... - sont héritées)
 - ❑ un constructeur prendra en arguments **les détails** (message et autres)
 - ❑ un constructeur prendra **en plus l'exception d'origine**

Créer son propre type d'exception

```
class TemperatureException extends Exception {  
  
    private int temperature;  
  
    public TemperatureException(string msg, int temp) {  
        super(msg) ;  
        this.temperature = temp;  
    }  
  
    public TemperatureException(string msg, int temp, Exception originalCause) {  
        this(msg, temp) ;  
        this.initCause(originalCause);  
    }  
  
    public int getTemperature() {  
        return temperature;  
    }  
}
```

Exemple d'utilisation

```
public void boireCafe(Tasse tasse) throws TemperatureException {  
    int temperature = tasse.getTemperature();  
    if (temperature <= limiteInferieure) {  
        throw new TemperatureException("Café trop froid !", temperature);  
    } else if (temperature >= tooHot) {  
        throw new TemperatureException("Café trop chaud !", temperature);  
    }  
  
    // ... consommer le café ...  
  
}
```

Résumé

- ❑ Les exceptions fournissent un **moyen non-intrusif** de gérer les erreurs
- ❑ **try/catch/finally** : permet de **structurer** clairement la gestion d'erreurs
- ❑ Les exceptions sont **récupérées suivant leur type** :
 - ❑ **plusieurs clauses catch** peuvent être spécifiées pour gérer différents types d'erreurs
 - ❑ elles doivent être **ordonnées** en fonction de la hiérarchie des classes d'exceptions (du plus spécifique au plus général)
- ❑ On peut **lancer une exception** grâce au mot-clé **throw**
- ❑ Les méthodes doivent déclarer les (*checked*) exceptions qu'elle peuvent propager avec la clause **throws**
- ❑ Si les exceptions de la JCL ne suffisent pas, on peut créer nos propres types d'exceptions, qui hériteront en général de la classe **Exception**