



Le langage Java

Exceptions

Objectifs Pédagogiques

À l'issue de cette formation, vous serez en mesure de :

- ✓ Utiliser les exceptions lors des développements

Programme détaillé ou sommaire

Approche conventionnelle

Principaux rôles des exceptions

Pile d'appels

Générer une erreur

Traiter les erreurs

Bonne pratique

Approche conventionnelle

- Voici une méthode qui divise 2 nombres.
- Sachant qu'une division par 0 est interdite, que faire dans le cas où b est égal à 0 ?

```
double diviser(int a, int b) {  
    if (b == 0) {  
        ???  
    }  
    return a / b;  
}
```

Approche conventionnelle

- Voici une méthode qui édite un PDF contenant les informations du compte bancaire d'un client.
- Que faire si le numéro du client n'est pas connu ? Il faudrait pouvoir sortir de la méthode avant d'arriver au bout et de signaler une anomalie.

```
void editerComptePdf(String numeroClient) {  
    if (numeroClient == null) {  
        ???  
    }  
    Document document = new Document();  
    ...  
}
```

Principaux rôles des exceptions

- **Générer une erreur** si une situation empêche le bon déroulement d'une méthode
 - ❑ Permet une sortie anticipée de la méthode
- **Propager les erreurs** dans la pile d'appels
 - ❑ Permet de faire remonter une erreur dans une méthode appelante
- **Traiter les erreurs**
 - ❑ Là où elle se produit ou dans une méthode appelante

Pile d'appels

- Java connaît à tout moment quelle suite d'appels l'a conduit dans la méthode courante.
- Si la pile d'appels est "pleine" : `StackOverflowError`
- Exemple:
 - ❑ `void Application.main(String[] args)`
 - ❑ `int RechercherPopulationDeptServices.rechercher(String codeDept)`
 - ❑ `List<Ville> RechercherPopulationVillesDao.extraireVilles(String codeDept)`

Générer une erreur

- Permet de sortir de manière anticipée d'une méthode
- Pour ce faire, on «jette» un objet de type Exception
 - ❑ L'idée est de remonter l'erreur à la méthode appelante
- La méthode appelante peut alors : soit traiter l'erreur, soit la remonter un cran plus haut dans la pile d'appels.

Traiter les erreurs

- En Java on choisit où l'erreur doit être traitée dans la pile d'appels
- On peut donc soit, à chaque niveau de la pile d'appels, décider de :
 - ❑ Soit traiter l'erreur dans la méthode
 - ❑ Soit la propager en la remontant d'un cran
- ❑ `void Application.main(String[] args)` **Traite**
- ❑ `int RechercherPopulationDeptServices.rechercher(String codeDept)` **remonte**
- ❑ `List<Ville> RechercherPopulationVillesDao.extraireVilles(String codeDept)` **remonte**
Exception jetée ici

Bonne pratique

- L'exception est générée au niveau le plus bas
- L'exception est traitée au niveau le plus haut (méthode main dans l'exemple)

❑ `void Application.main(String[] args)` **On traite ici**

❑ `int RechercherPopulationDeptServices.rechercher(String codeDept)` **On remonte**

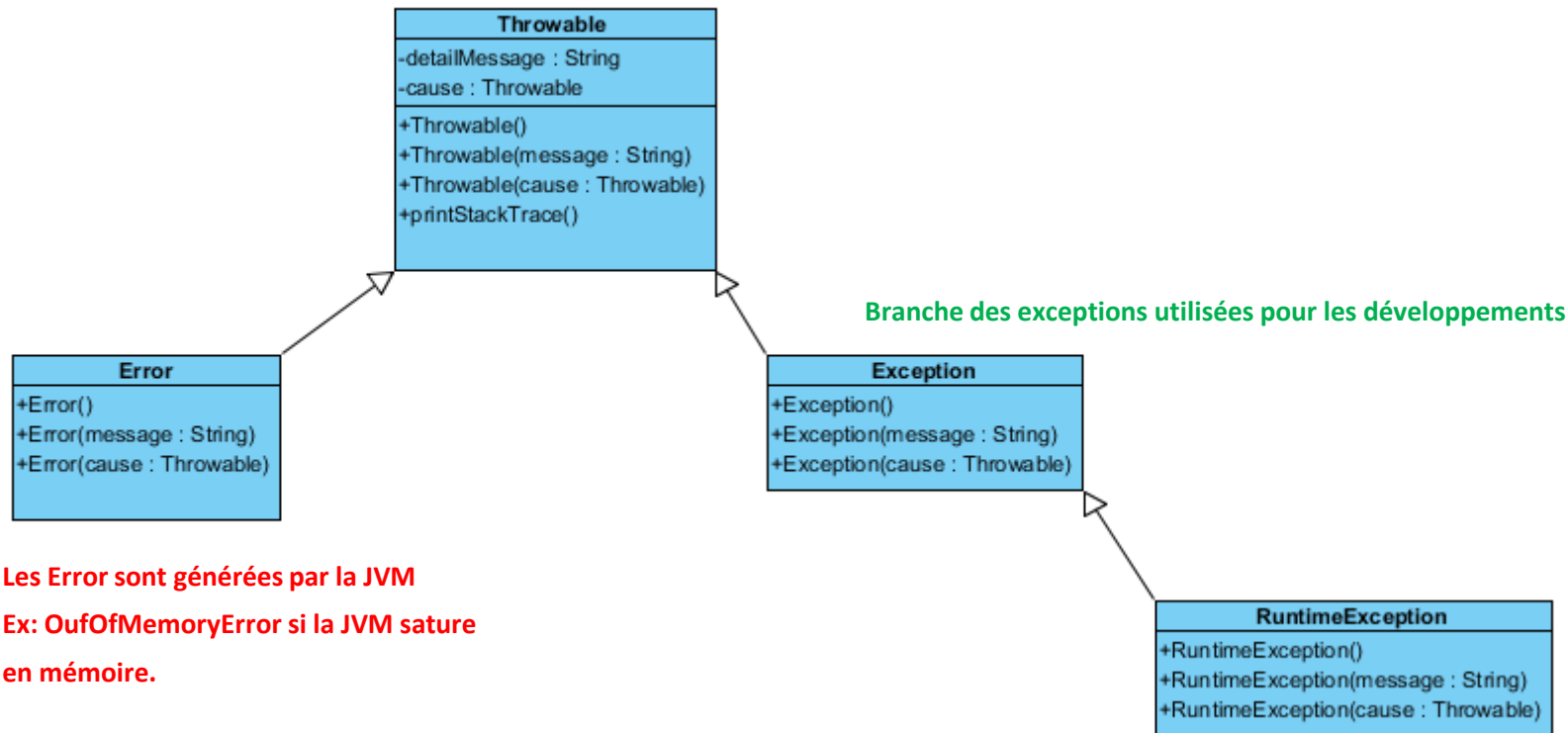
❑ `List<Ville> RechercherPopulationVillesDao.extraireVilles(String codeDept)` **On remonte**

On jette au plus bas: là

Avant de rentrer dans le détail, un zoom
sur les exceptions

Hiérarchie des exceptions

- Les exceptions sont des classes !!



Création de l'exception

➤ Etape 1

- ❑ Création d'une classe de type exception qui hérite de la classe **Exception**
- ❑ Classe fille de la classe **java.lang.Exception**

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

Remontée de l'exception

➤ Etape 2


- ❑ Jeter l'exception là où une anomalie potentielle doit être traitée
- ❑ Avec l'instruction **throw** dans le corps de la méthode
- ❑ Avec l'instruction **throws** dans la signature de la méthode
- ❑ On "throw" une instance de l'exception (opérateur new)

```
public class RechercherPopulationVillesDao {  
  
    public List<Ville> extraireVilles(String codeDept) throws CodeDeptException {  
  
        if (codeDept==null){  
            throw new CodeDeptException("Veuillez renseigner un code département.");  
        }  
  
        // Partie exécutée seulement si l'exception ne s'est pas produite  
  
    }  
}
```

Déclaration des exceptions lancées

- La méthode peut remonter plusieurs exceptions potentielles.
- Dans la signature, il faut toutes les indiquer avec le séparateur virgule

```
typeRetour nomMethode(...) throws ClasseException, ... {  
    ...  
}
```



Pour traiter une erreur

➤ Blocs try / catch

- ❑ On invoque la méthode qui remonte l'exception dans le bloc **try**. C'est le bloc de code nominal.
- ❑ Si l'exception est remontée, c'est le bloc **catch** qui sera exécuté. C'est le bloc alternatif.

```
public void rechercher(String codeDept){  
  
    RechercherPopulationVillesDao dao = new RechercherPopulationVillesDao();  
    try {  
        List<Ville> villes = dao.extraireVilles(codeDept);  
        //TODO code nominal ici  
    }  
    catch (CodeDeptException e){  
        System.err.println(e.getMessage());  
        //TODO code alternatif ici  
    }  
}
```


Pour remonter une erreur

- Remontée de l'exception dans la classe appelante
 - ❑ Si on ne souhaite pas traiter l'exception dans la méthode **rechercher** (cf. exemple pas précédente), il faut la remonter avec **throws** dans la signature.
 - ❑ On ajoute la clause **throws** dans la signature de cette méthode

```
public void rechercher(String codeDept) throws CodeDeptException {  
    List<Ville> villes = dao.extraireVilles(codeDept);  
    //TODO Suite code ici  
}
```

Pour traiter une erreur : à retenir

- La clause **throws** placée dans la signature permet de remonter l'exception
- Le bloc **try / catch** permet de traiter l'exception
 - ❑ On invoque la méthode dans le bloc try ainsi que tout le code
 - ❑ On met en place le traitement de l'exception dans le corps du catch
- La méthode de plus haut niveau (ex: main) peut également comporter la clause throws dans sa signature.
 - ❑ Dans ce cas l'exception est traitée nulle part. Si elle se produit l'application tombe en erreur.

Que faire dans un bloc catch ?

- Une exception fournit des informations exploitables
- Exemples :

```
e.getMessage()      // correspond au message de l'exception  
e.printStackTrace() // méthode qui affiche la pile d'appels  
e.getCause()        // correspond à la cause racine de l'erreur.
```

...

```
catch (Exception e) {  
    e.printStackTrace();  
    System.err.println(e.getMessage());  
}
```

Catches multiples

- Il est possible de mettre en place plusieurs blocs **catch**.
- Le bloc **finally** est exécutée à la suite du **try** ou à la suite du **catch**.

```
try {  
    // code à risque  
}  
catch (CodeDeptNullException e) {  
    // Traitement de ce type d'exception  
}  
catch (Exception e) {  
    // Traitement de tous les autres types  
}
```

Catches multiples avec un OR

- Il est possible de traiter plusieurs exceptions avec un **catch multiple**.
- En utilisant l'opérateur |

```
try {  
    // code à risque  
}  
catch (CodeDeptNullException | CodeRegionNullException e) {  
    // Traitement de ces types d'exceptions  
}  
catch (Exception e) {  
    // Traitement de tous les autres types  
}
```

Bloc finally

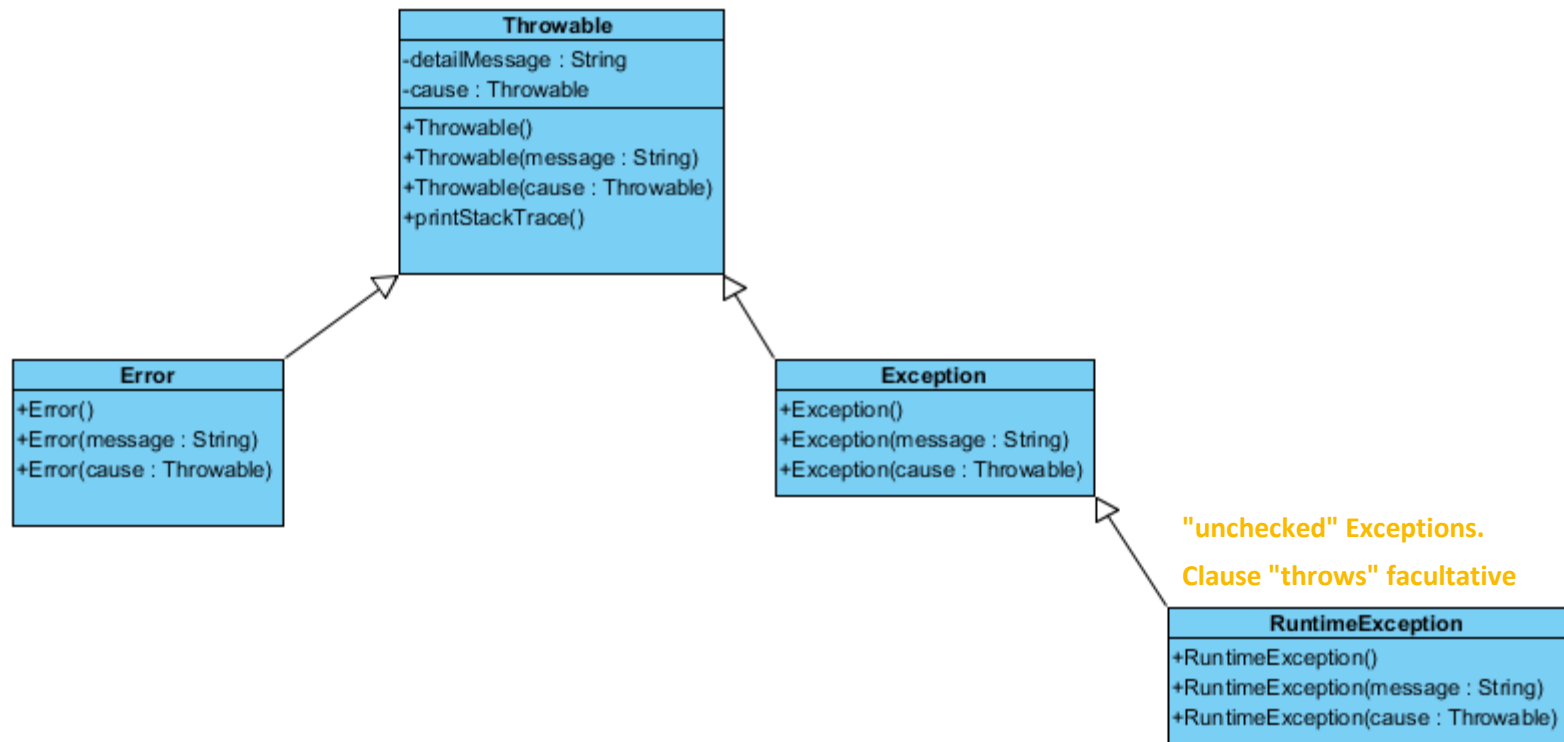
- Garantit l'exécution d'un bloc d'instructions quoiqu'il se produise.
- Le bloc finally est exécutée à la suite du try ou à la suite du catch.

```
try {  
    // code à risque  
}  
catch (CodeDeptNullException e) {  
    // Traitement de ce type d'exception  
}  
catch (Exception e) {  
    // Traitement de tous les autres types  
}  
finally {  
    // bloc toujours exécuté  
}
```

Atelier (TP)

- Objectifs du TP: mettre en place une gestion des exceptions
- Description du TP:
 - Dans ce TP, vous allez mettre en place une gestion d'exceptions.

Annexes : RuntimeException



Qu'est-ce qu'une RuntimeException ?

- Elle représente une exception technique grave
- Son objectif est de stopper brutalement une application.
- Cela peut se justifier quand du code est impossible à exécuter.
- Exemples de cas :
 - ❑ Un tableau est accédé avec un index trop grand
 - ❑ Une méthode est invoquée sur un objet null
 - ❑ La pile d'appels est saturée (appel récursif infini)
 - ❑ Une base de données, sur laquelle repose l'application, est inaccessible

Quelles différences avec une Exception ?

- Beaucoup plus simple à mettre en œuvre.
- La clause `throw` suffit.
 - ❑ La clause `throws` dans la signature est facultative. L'exception remonte toute seule.
 - ❑ La clause `try / catch` est facultative. L'exception n'est pas censée être traitée.
- C'est une exception dite non-checkée, ou non vérifiée.

Créer une exception Runtime

➤ Etape 1

- ❑ Création d'une exception qui hérite de **RuntimeException**

```
public class CodeDeptNullException extends RuntimeException {  
    public CodeDeptNullException(String msg) {  
        super(msg);  
    }  
}
```

Ni throws, ni try / catch

- **Etape 2:** l'exception est jetée avec la clause **throw**, mais :
 - ❑ **throws** n'est pas obligatoire dans la signature
 - ❑ Le traitement de l'exception est facultatif dans les méthodes appelantes.

```
public class RechercherPopulationVillesDao {  
  
    public List<Ville> extraireVilles(String codeDept) {  
  
        Connection connexionDB = getConnection("jdbc:mysql://localhost:3306/mabase", "root", "");  
        if (connexionDB==null){  
            throw new DatabaseStoppedException("La base de données est arrêtée. Veuillez la démarrer.");  
        }  
  
        // Partie exécutée seulement si l'exception ne s'est pas produite  
    }  
}
```

Annexes : Exceptions et redéfinition

- Si une méthode abstraite **throws** une **exception** (signature de méthode)
- Dans une classes fille, la méthode redéfinie **throws** une exception qui **hérite de cette exception**.

```
public abstract class AbstractDao {  
    public abstract List<Data> extraireData(String code) throws VerificationException;  
}  
  
public class RechercherPopulationVillesDao extends AbstractDao {  
    public List<Data> extraireData(String code) throws CodeNullException {  
    }  
}
```

