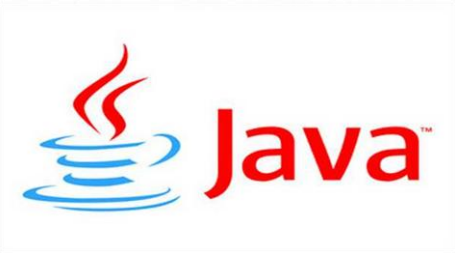
Faculté des Sciences
كلية العلوم
جامعة ابن طفيل Université Ibn Tofaïl

Département d'Informatique
Filière SMI - Semestre 6
2020-2021

Programmation Orientée Objet

Application avec Java



Pr. Khadija Louzaoui

Chapitre 6
Gestion des exceptions

La fiabilité en Java

Problématique de la fiabilité

- Tout programme comporte des erreurs (bugs) ou est susceptible de générer des erreurs (e.g suite à une action de l'utilisateur, de l'environnement, etc ...).

- La fiabilité d'un logiciel peut se décomposer en deux grandes propriétés :

  - la robustesse qui est la capacité du logiciel à continuer de fonctionner en présence d'événements exceptionnels tels que la saisie d'informations erronées par l'utilisateur ;
  - la correction qui est la capacité d'un logiciel à donner des résultats corrects lorsqu'il fonctionne normalement.

La fiabilité en Java

Assurer la fiabilité en Java

- Le langage Java inclus plusieurs mécanismes permettant d'améliorer la fiabilité des programmes :
  - les **exceptions** pour la robustesse
  - les **assertions** pour la correction.
- A ces mécanismes viennent s'ajouter des outils complémentaires tels :
  - des outils de test unitaire comme **JUnit** ;
  - des outils de debuggage comme **jdb**.

Les exceptions

La gestion d'erreurs en Java

- Le langage Java propose un mécanisme particulier pour gérer les erreurs : **les exceptions**.

- Ce mécanisme repose sur deux principes :

  – Les différents types d'erreurs sont modélisées par des classes.

  – Les instructions susceptibles de générer des erreurs sont séparées du traitement de ces erreurs : concept de **bloc d'essai** et de **bloc de traitement d'erreur**.

Les exceptions

Exemple

```
public class Test {
public static void main(String[] args) {
   int x = 0;
   int y = 5/x;
        System.out.print(y);
        System.out.println("Fin de calcul");
}
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:20)
```

Le message Fin de calcul n'a pas été affiché
La division par zéro déclenche une exception ArithmeticException

Les exceptions

Définition

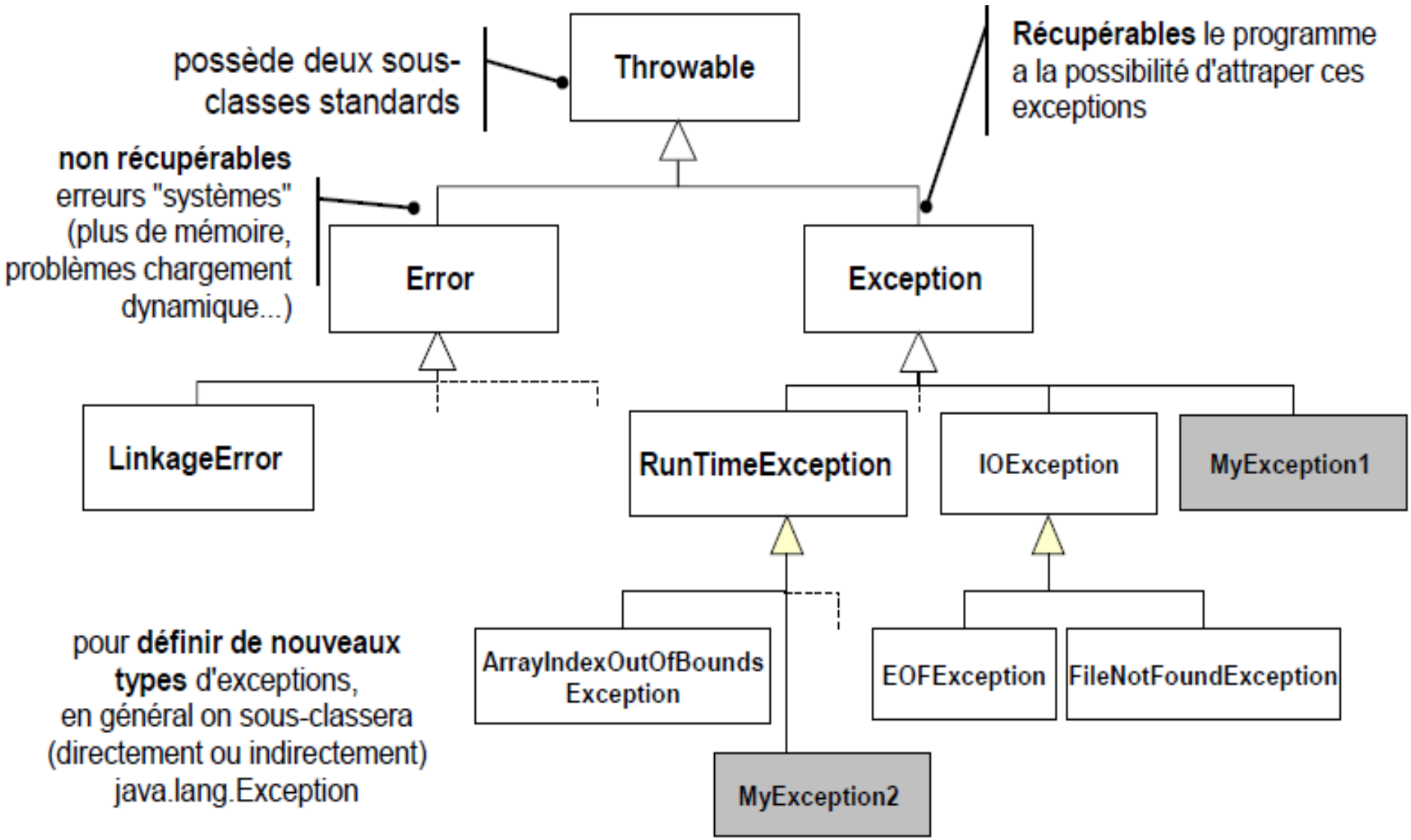
Le terme **exception** désigne tout événement arrivant durant l'exécution d'un programme interrompant son fonctionnement normal.

- En java, les exceptions sont matérialisées par des instances de classes héritant de la classe java.lang.Throwable.

- A chaque évènement correspond une sous-classe précise, ce qui peut permettre d'y associer un traitement approprié.

Les exceptions

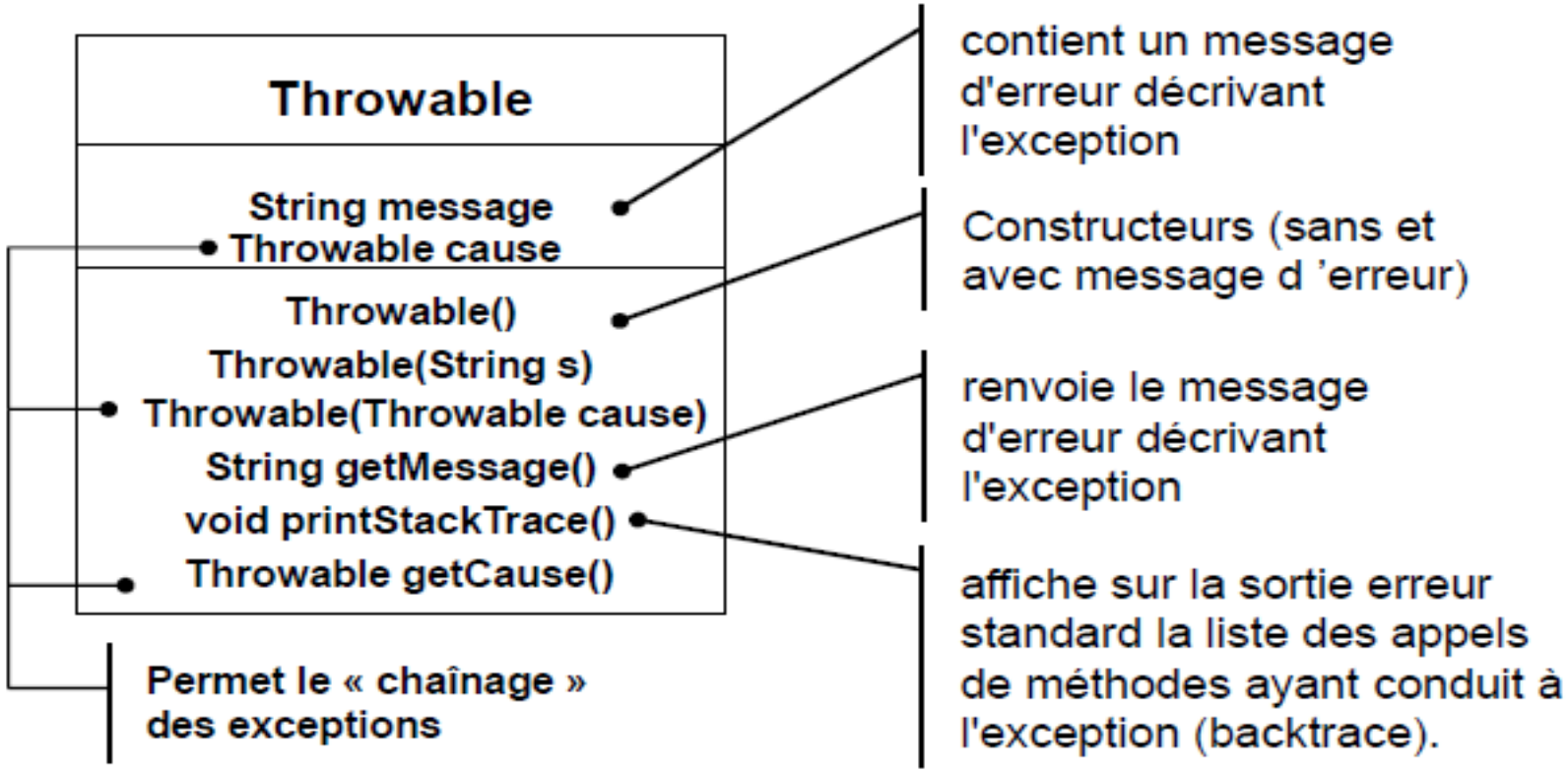
Arbre d'héritage des exceptions



Les exceptions

Arbre d'héritage des exceptions

- Puisqu'elles **sont des objets** les exceptions peuvent contenir :
  - *des attributs particuliers,*
  - *des méthodes.*
- Attributs et méthodes standards (définis dans `java.lang.Throwable`)



Les exceptions

Comment faire pour poursuivre l'exécution?

- Repérer les blocs pouvant générer une exception
- Capturer l'exception correspondante
- Afficher un message relatif à cette exception
- Continuer l'exécution

Définition

Lorsqu'un bloc de traitement d'erreur est déclenché par une exception, on dit qu'il **traite** *(***capture***) cette exception.*

Définition

Lorsqu'une instruction du bloc d'essai génère une erreur et y associe une exception, on dit qu'elle **lève** *(***lance***) cette exception.*

Les exceptions

Capturer une exception

- La clause **try** s'applique à un bloc d'instructions correspondant au fonctionnement normal mais pouvant générer des erreurs.

- La clause **catch** s'applique à un bloc d'instructions définissant le traitement d'un type d'erreur. Ce traitement sera lancé sur une instance de la classe d'exception passée en paramètre.

```
try{
...
   }
catch(TypeErreur1 e) {
...
   }
catch(TypeErreur2 e) {
...
   }
```

Les exceptions

Capturer une exception

- Tout bloc **try** doit être suivit par au moins un bloc **catch** ou par un bloc **finally**.
- Tout bloc **catch** doit être précédé par un autre bloc **catch** ou par un bloc **try**.

- un **seul bloc catch peut être exécuté**: le premier susceptible "d'attraper" l'exception.

- l'**ordre** des blocs **catch** est donc très important.

Les exceptions

Capturer une exception

Utiliser un bloc try { ... } catch {... }

```
public static void main(String[] args) {
int x = 5, y = 0;
try {
        System.out.println(x/y);
    }
catch (ArithmeticException e) {
        System.out.println("Exception : Division par zero ");
    }
System.out.println("Fin de calcul");
}
```

```
Exception : Division par zero
Fin de calcul
```

Les exceptions

Capturer une exception

Et si on ne connais pas le type d'exception

```
public static void main(String[] args) {
int x = 5, y = 0;
try {
        System.out.println(x/y);
    }
catch (Exception e) {
        System.out.println("Exception : Division par zero ");
    }
System.out.println("Fin de calcul");
}
```

```
Exception : Division par zero
Fin de calcul
```

Les exceptions

Capturer une exception

Utiliser des méthodes de la classe **Exception**

```
public static void main(String[] args) {
int x = 5, y = 0;
try {
        System.out.println(x/y);
    }
catch (Exception e) {
        System.out.println("Exception : " + e.getMessage());
    }
System.out.println("Fin de calcul");
}
```

```
Exception : / by zero
Fin de calcul
```

Les exceptions

Capturer une exception

Utiliser des méthodes de la classe **Exception**

```
public static void main(String[] args) {
int x = 5, y = 0;
try {
        System.out.println(x/y);
    }
catch (Exception e) {
        e.printStackTrace();
    }
System.out.println("Fin de calcul");
}
```

```
java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:21)
Fin de calcul
```

Les exceptions

Finally

- La clause **finally** définit un bloc d'instruction qui sera exécuté même si une exception est lancée dans le bloc d'essai.
- Elle permet de forcer la bonne terminaison d'un traitement en présence d'erreur, par exemple : la fermeture des fichiers ouverts.

```
try {
...
}
catch (...) {
...
}
finally {
...
}
```

- Le bloc **finally** peut s'avérer intéressant si le catch contient un return qui forcera l'arrêt de l'exécution du code. Malgré cela, ce bloc (**finally**) sera exécuté.

Les exceptions

Lever une exception

- Pour lancer une exception, on peut utiliser la clause **throw**.
- l'instruction **throw unObjetException** permet de lancer une exception
- **unObjetException** doit être une référence vers une instance d'une sous-classe de **Throwable**
- quand une exception est lancée,
  - 1. l'exécution normale du programme est interrompue,
  - 2. la JVM recherche la clause **catch** la plus proche permettant de traiter l'exception lancée,
  - 3. cette recherche se propage au travers des blocs englobants et remonte les appels de méthodes jusqu'à ce qu'un gestionnaire de l'exception soit trouvé,
  - 4.**tous** les blocs **finally** rencontrés au cours de cette propagation sont exécutés.

Les exceptions

Lever une exception

- Toute exception contrôlée, du JDK ou personnalisée, pouvant être émise dans une méthode doit être :

  – soit levée dans cette méthode. Elle est alors lancée dans un bloc try auquel est associé un catch lui correspondant.

  – soit être indiquées dans le prototype de la méthode à l'aide du mot clé throws.

Attention : ne confondez pas

Throw (sans s) pour lever une exception

Avec

Throws (avec s) pour indiquer quelles exceptions sont levées

Les exceptions

Lever une exception

```
public void f() {
try {
      FileInputStream monFichier ;
// Ouvrir un fichier peut générer une exception
monFichier = new FileInputStream("./essai.txt");
} catch (FileNotFoundException e) {
      System.out.println(e);
      }
}
public static void main(String[] args) {
      f();
}
```

Les exceptions

Lever une exception

```
public void f() throws FileNotFoundException {
        FileInputStream monFichier ;
// Ouvrir un fichier peut générer une exception
        monFichier = new FileInputStream("./essai.txt");
}
public static void main(String[] args) {
try {
// Un appel a f() peut générer une exception
        f();
} catch (FileNotFoundException e) {
System.out.println(e);
        }
}
```

Les exceptions

Exceptions prédéfinies

Les exceptions levées par la VM correspondent :

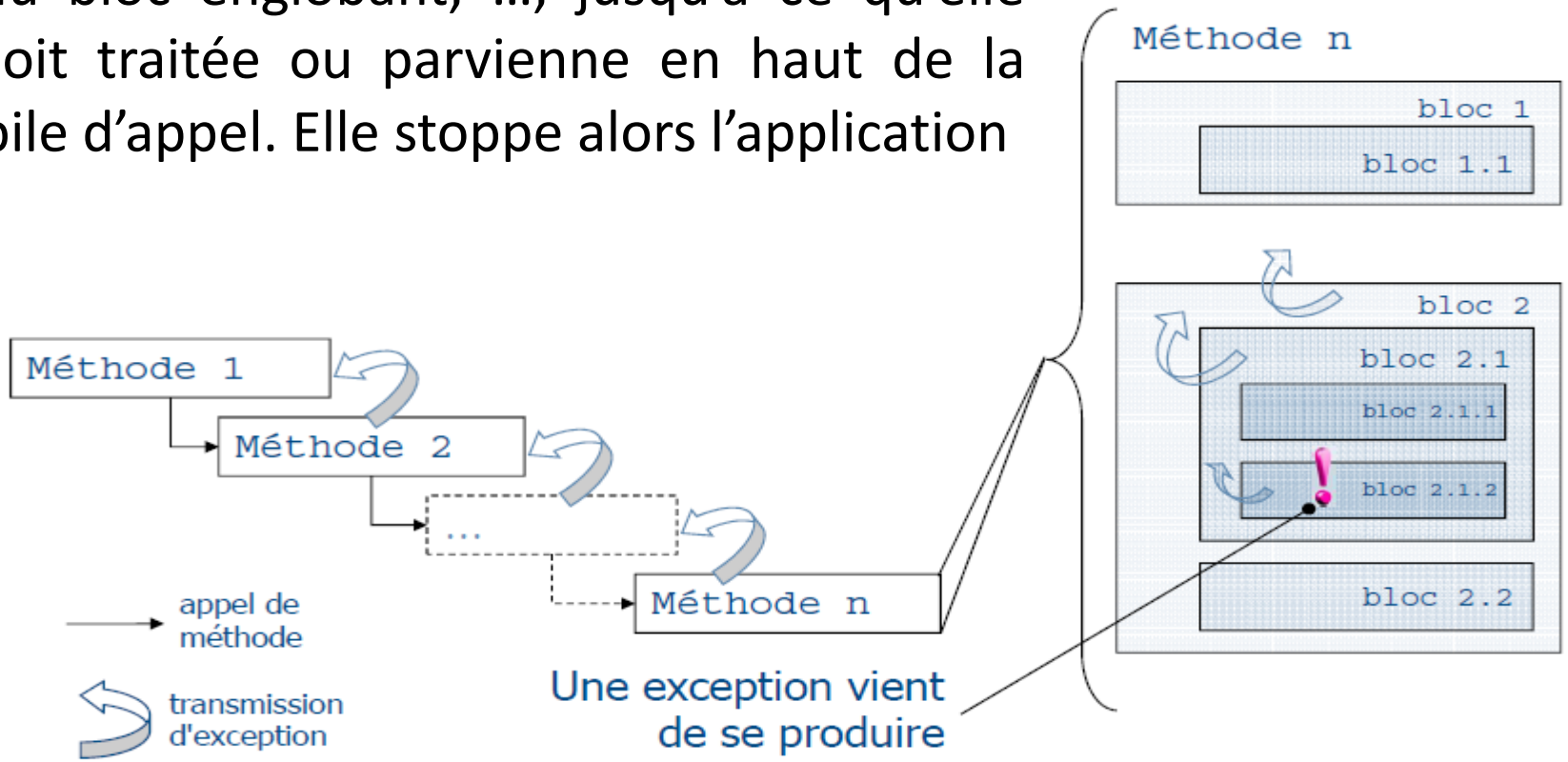
- Erreur de compilation ou de lancement
  - NoClassDefFoundError, ClassFormatError
- problème d'entrée/sortie :
  - IOException, AWTException
- problème de ressource :
  - OutOfMemoryError, StackOverflowError
- des erreurs de programmation (runtime)
  - NullPointerException, ArrayIndexOutOfBoundsException, ArithmethicException,

- On peut aussi définir ses exceptions personnalisées

Les exceptions

Propagation des Exception

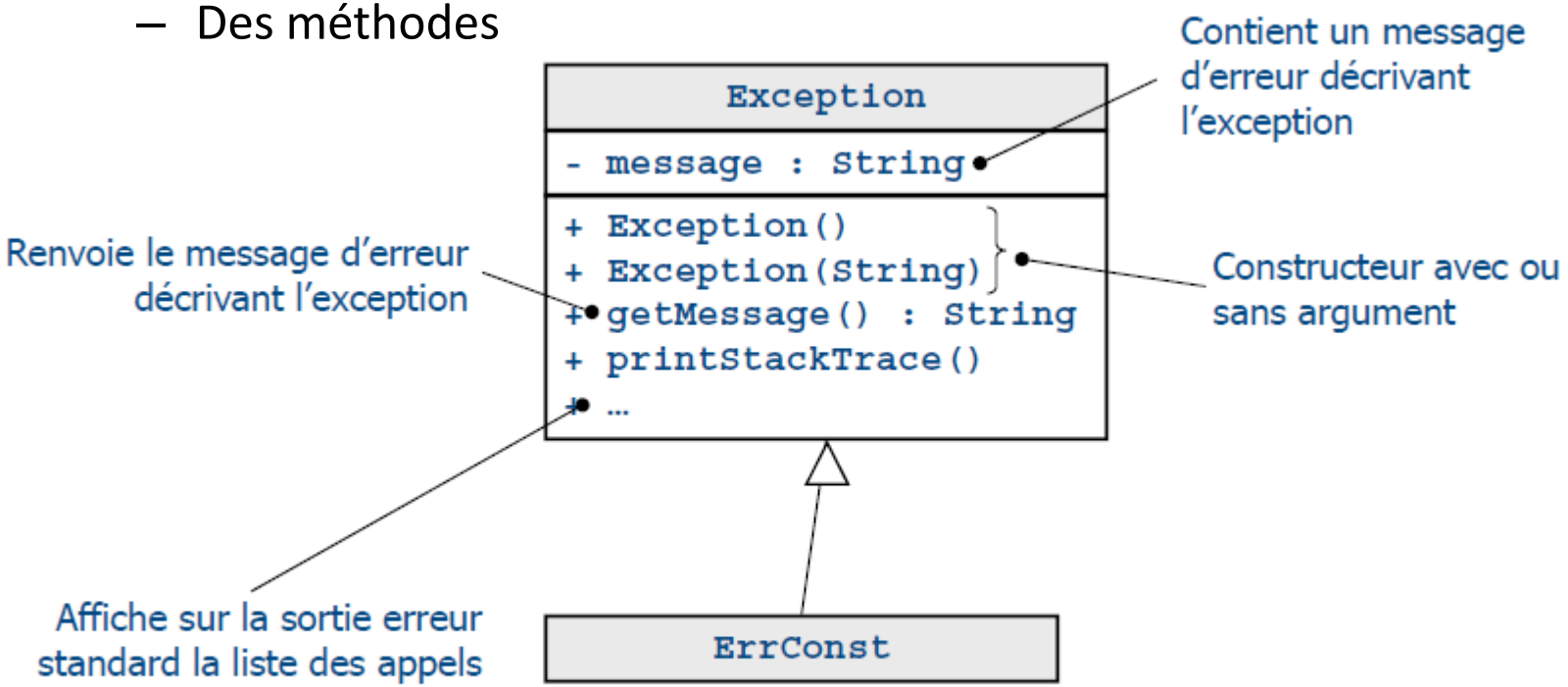
- Lorsqu'une situation exceptionnelle est rencontrée, une exception est lancée
- Si elle n'est pas traitée, elle est transmise au bloc englobant, …, jusqu'à ce qu'elle soit traitée ou parvienne en haut de la pile d'appel. Elle stoppe alors l'application



Les exceptions

Modélisation

- Les exceptions sont des objets nous pouvons donc définir
  - Des attributs particuliers
  - Des méthodes




```
public class ErrConst extends Exception {
     public ErrConst() { }
     public void afficherErreur(){
          System.out.println("Erreur de création");
     }}
```

Les exceptions

Les exceptions personnalisées

Exemple: classe Adresse

```
public class Adresse {
  private String rue;
  private String ville;
  private String codePostal;
public Adresse(String rue, String ville, String codePostal) {
  this.rue = rue;
  this.ville = ville;
  this.codePostal = codePostal;
}
// ensuite les getters/setters et autres méthodes
...
}
```

Supposons que **codePostal** doit contenir exactement **5** chiffres

Les exceptions

Les exceptions personnalisées

Démarche à faire

- Créer notre propre exception (qui doit étendre la classe **Exception**)
- Dans **le constructeur** de Adresse, on lance une exception si codePostal ne contient pas 5 chiffres

Créons l'exception CodePostalException

```
public class CodePostalException extends Exception {
// le constructeur de cette nouvelle exception
public CodePostalException (){
System.out.println("Le code postal doit contenir
                    exactement 5 chiffres");
      }
}
```

Les exceptions

Les exceptions personnalisées

Modifions le constructeur de la classe Adresse

```
public class Adresse {
// apres les attributs
public Adresse(String rue, String ville, String codePostal)
throws CodePostalException
{
if (codePostal.length()!=5)
throw new CodePostalException ();
this.rue = rue;
this.ville = ville;
this.codePostal = codePostal;
}
}
// il faut faire pareil dans setCodePostal()
```

Les exceptions

Les exceptions personnalisées

Testons tout cela dans le main()

```
public static void main(String[] args) {
Adresse a = null;
try {
a = new Adresse ("rue de El Qods", "kenitra","1400");
}
catch(CodePostalException cpe) {
cpe.printStackTrace();
}
}
```

Le message affiché est :
Le code postal doit contenir exactement 5 chiffres

Les exceptions

Les instructions multi-catch

On peut rajouter une deuxième condition

- codePostal doit contenir exactement 5 chiffres
- rue doit être une chaîne en majuscule

Créons une deuxième exception RueException

```
public class RueException extends Exception {
public RueException() {
System.out.print("Le nom de la rue doit etre en majuscule");
}
}
```

Les exceptions

Les instructions multi-catch

Modifions le constructeur de la classe Adresse

```
public class Adresse {
// après les attributs
public Adresse(String rue, String ville, String codePostal)
throws CodePostalexception, RueException {
if (codePostal.length()!=5)
      throw new CodePostalException();
if(!rue.equals(rue.toUpperCase()))
      throw new RueException();
      this.rue = rue;
      this.ville = ville;
      this.codePostal = codePostal;
}
}
```

Les exceptions

Les instructions multi-catch

Re-testons tout cela dans le main()

```
public static void main(String[] args) {
Adresse a = null;
try {
a = new Adresse ("rue de El Qods", "kenitra","1400");
}
catch(CodePostalException cpe) {
      cpe.printStackTrace();
}
catch(RueException re) {
      re.printStackTrace();
}
}
```

Les exceptions

Les instructions multi-catch

Java 7 ajoute quelques nouveautés au langage Java pour faciliter la gestion des exceptions :

- récupération de plusieurs types d'exception dans un même bloc catch
  - évite d'avoir à répéter des blocs catchs identiques pour des exceptions de types différents

```
public static void main(String[] args) {
Adresse a = null;
try {
a = new Adresse ("rue de El Qods", "kenitra","1400");
}
catch(CodePostalException | RueException e) {
      e.printStackTrace();
}}
```

Les exceptions

Les exceptions paramétrées

Si on voudrait afficher les valeurs qui ont déclenché l'exception dans le message

Modifions les premières exceptions

```
public class CodePostalException extends Exception {
// le constructeur de la nouvelle exception
public CodePostalException (String cp){
System.out.println("Le code postal"+cp+" doit contenir
                    exactement 5 chiffres");}
}
```

```
public class RueException extends Exception {
public RueException(String rue) {
System.out.print("Le nom de la rue"+rue+" doit
             être en majuscule");}
 }
```

Les exceptions

Les exceptions paramétrées

Modifions le constructeur de la classe Adresse

```
public class Adresse {
// après les attributs
public Adresse(String rue, String ville, String codePostal)
throws CodePostalexception, RueException {
if (codePostal.length()!=5)
      throw new CodePostalException(codePostal);
if(!rue.equals(rue.toUpperCase()))
      throw new RueException(rue);
      this.rue = rue;
      this.ville = ville;
      this.codePostal = codePostal;
}
}
```

Les exceptions

Les exceptions paramétrées

Re-testons tout cela dans le main()

```
public static void main(String[] args) {
Adresse a = null;
try {
a = new Adresse ("rue de El Qods", "kenitra","1400");
}
catch(CodePostalException | RueException e) {
      e.printStackTrace();
}}
```

Les exceptions

Les exceptions paramétrées

Créer une nouvelle classe d'exception

AdresseException

pour fusionner et remplacer les deux exceptions

CodePostalException et **RueException**

Les exceptions

Les exceptions paramétrées

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

public void afficherErreur(Throwable cause) {
System.err.println("Erreur de Création :\n L'origine de
        l'erreur est \n \t"+cause);
}
```

Les exceptions

Les exceptions paramétrées

```
public class Adresse {
private String rue;
  private String ville;
  private String codePostal;
public Adresse(String rue, String ville, String codePostal)
throws AdresseException {
if(!rue.equals(rue.toUpperCase()))
throw new AdresseException ("Le nom de la rue "+rue+"
                            doit être en majuscule");
if(codePostal.length()!=5)
 throw new AdresseException ("Le code postal "+codePostal+"
                     doit contenir exactement 5 chiffres");
  this.rue = rue;
  this.ville = ville;
  this.codePostal = codePostal;
... }
```

Les exceptions

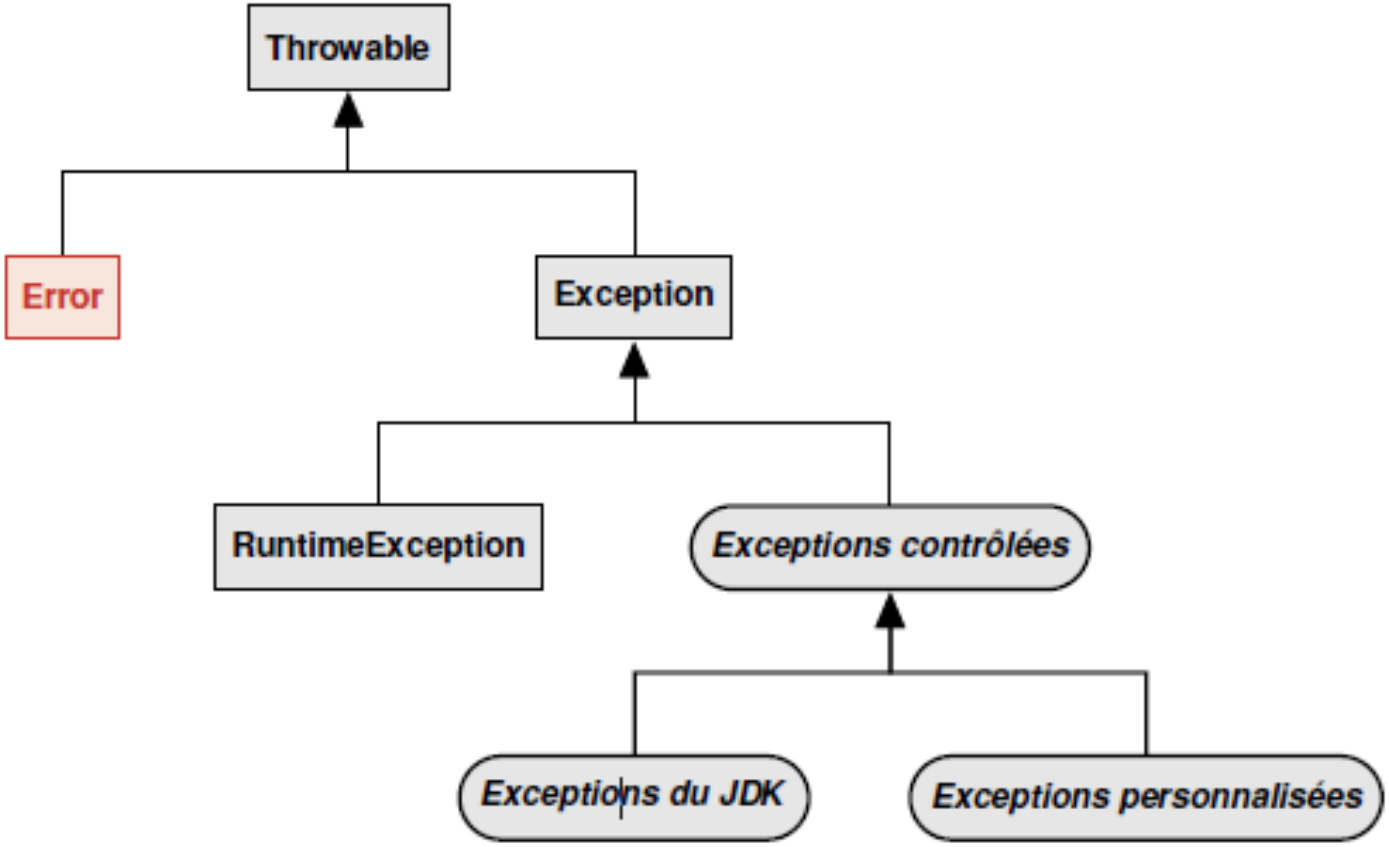
Les exceptions paramétrées

```
public class TestAdresse {

public static void main(String[] args) {
Adresse adr=null;
try {
adr = new Adresse("ELQODS", "Kenitra ", "14000");
      }
catch (AdresseException e) {
 e.afficherErreur(e);
}
System.out.println(adr); }
}
```

Les exceptions

Types des Exceptions



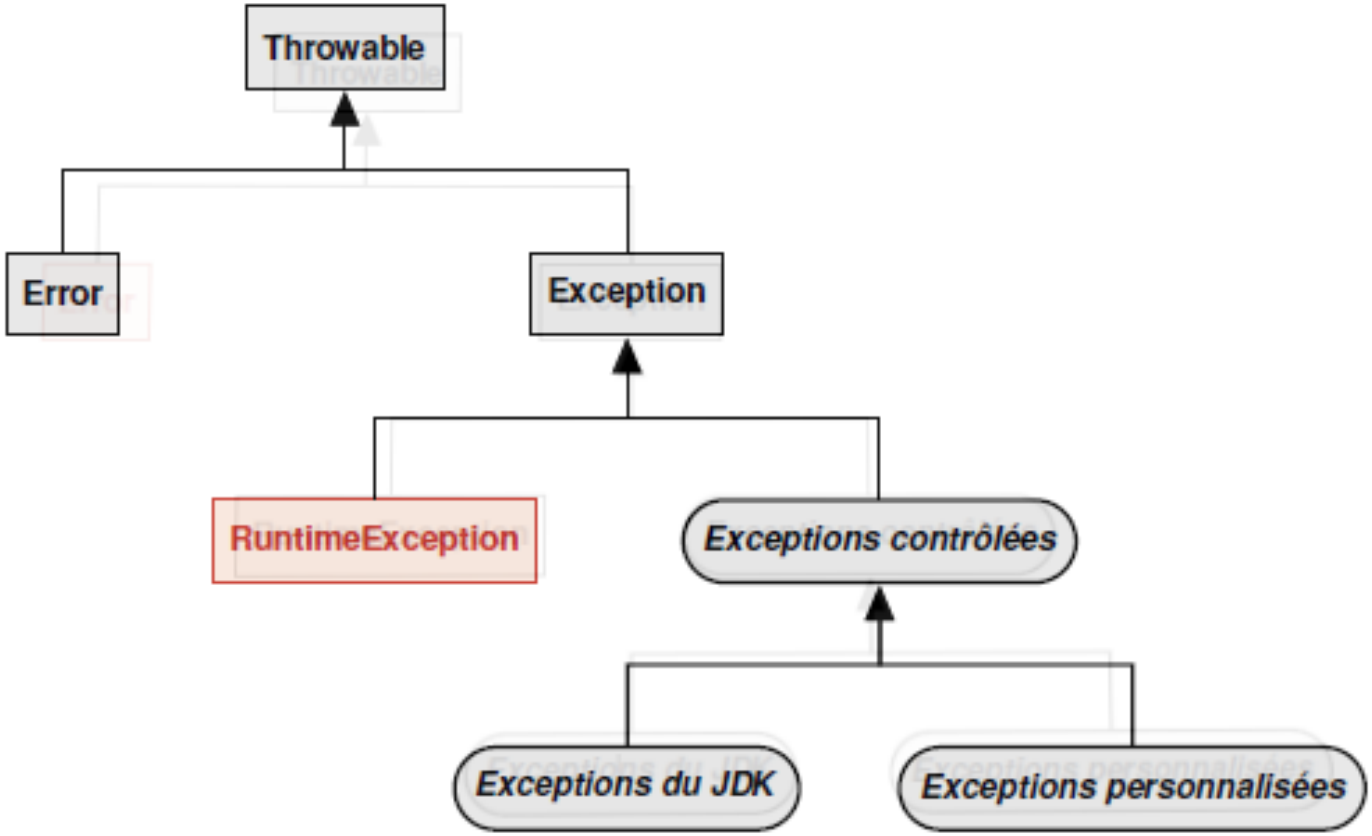
Les exceptions

Les Exceptions de type: Error

- Les exceptions de type **Error** sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM. Elles peuvent survenir dans toutes les portions du codes.

- Java définit de nombreuses sous-classes de **Error** :
  - **OutOfMemoryError** : survient lorsque la machine virtuelle n'a plus de place pour faire une allocation et que le GC ne peut en libérer.
  - **NoSuchMethodError** : survient lorsque la machine virtuelle ne peut trouver l'implémentation de la méthode appelée.
  - **StackOverflowError** : survient lorsque la pile déborde après une série d'appel récursif trop profond.
  - etc...

Les exceptions



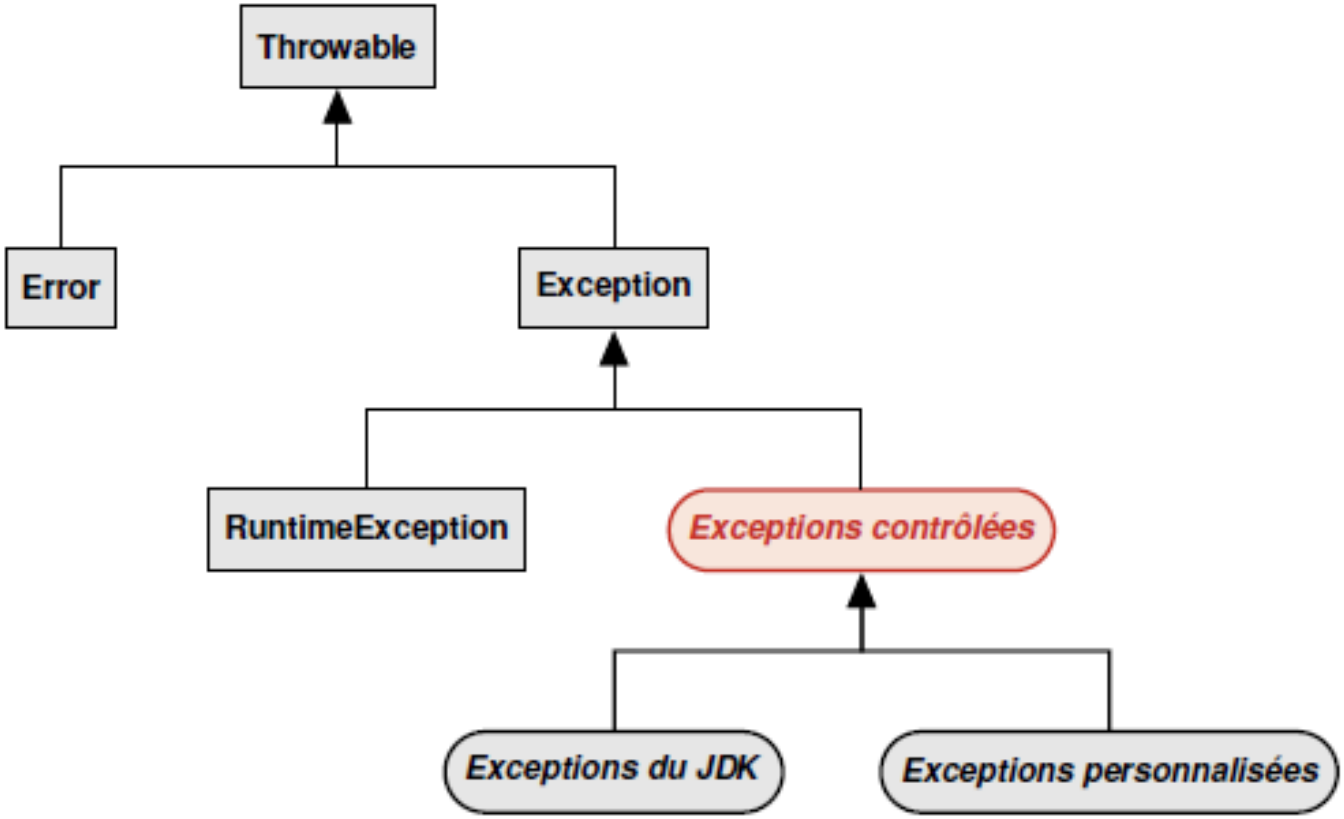
Les exceptions

Les Exceptions de type: RuntimeException

- Les exceptions de type **RuntimeException** correspondent à des erreurs qui peuvent survenir dans toutes les portions du codes.
- Java définit de nombreuses sous-classes de RuntimeException :

  - **ArithmeticException** : division par zéro (entiers), etc ....
  - **IndexOutOfBoundsException** : dépassement d'indice dans un tableau.
  - **NullPointerException** : référence **null** alors qu'on attendait une référence vers une instance.
  - etc...

Les exceptions

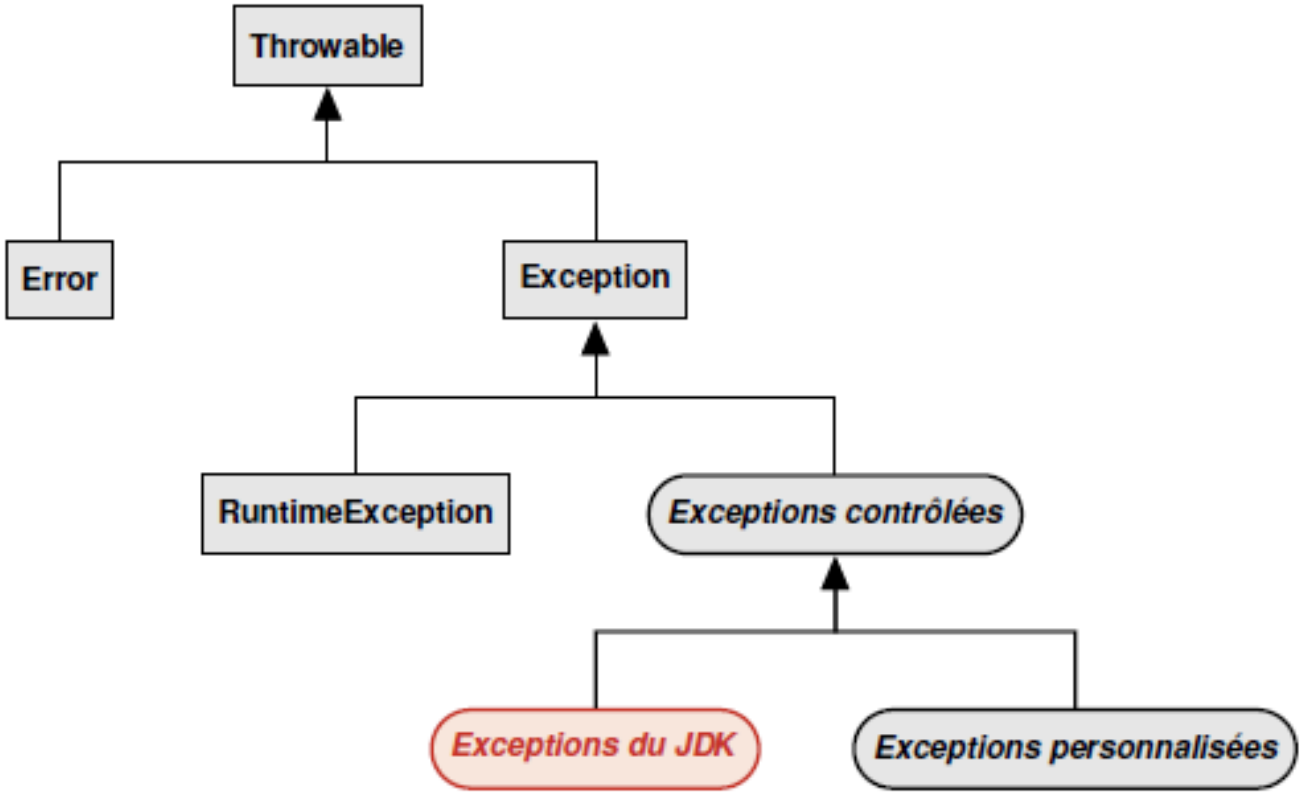


Les exceptions

Les Exceptions de type: Exception contrôlée

- On appelle **exception contrôlée**, toute exception qui hérite de la classe Exception et qui n'est pas une RuntimeException. Elle est dite contrôlée car le compilateur vérifie que toutes les méthodes l'utilisent correctement.
- Le JDK définit de nombreuses exceptions :
  - **EOFException** : fin de fichier.
  - **FileNotFoundException** : erreur dans l'ouverture d'un fichier.
  - **ClassNotFoundException** : erreur dans le chargement d'une classe.
  - etc...

Les exceptions



Les exceptions

Les Exceptions de type: Exception du JDK

- Le JDK contient des API qui abusent des **exceptions contrôlées** En effet, elles utilisent ces exceptions alors que la méthode appelante ne pourra pas résoudre le problème.
- Par exemple, *JDBC* (liée au langage *SQL*) et les *entrées-sorties*.
- Dans ce cas, une solution est d'attraper l'**exception contrôlée** et de renvoyer une **exception non contrôlée**

Les exceptions

En Résumé

- les exceptions rendent la gestion des erreurs **plus simple** et **plus lisible**
- le code pour gérer les erreurs peut être **regroupé en un seul endroit**: là où ont a besoin de traiter l'erreur
- possibilité de **se concentrer sur l'algorithme** plutôt que de s'inquiéter à chaque instruction de ce qui peut mal fonctionner,
- les erreurs **remontent la hiérarchie d'appels** grâce à l'exécutif du langage et non plus grâce à la bonne volonté des programmeurs.