Programmation et Algorithmique II

Ch.6 – Exceptions

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

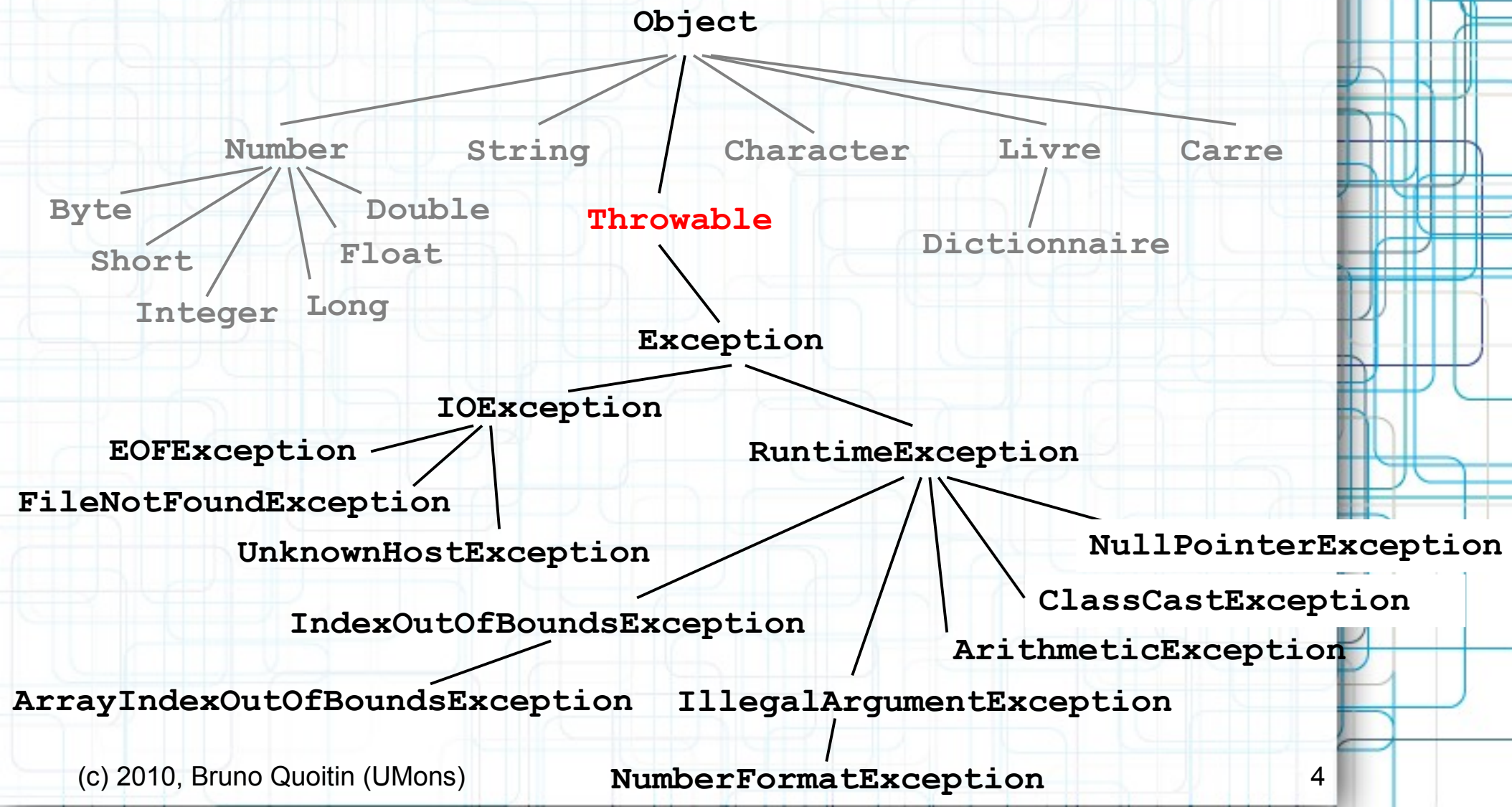
Exceptions

- **Introduction**
  - Une **exception** est un mécanisme permettant de signaler des conditions exceptionnelles durant l'exécution du programme. Typiquement, les exceptions sont utilisées pour gérer des erreurs à l'exécution.
  - En Java, **une exception est un *objet*** représentant les informations associées à cette condition exceptionnelle.Il existe différents types (classes) d'exceptions.
  - Plusieurs mécanismes entrent en jeu
    - le déclenchement d'exception(s)
    - la capture d'exception(s)
    - la déclaration d'exceptions contrôlées

n'a pas d'équivalent en Python

Exceptions

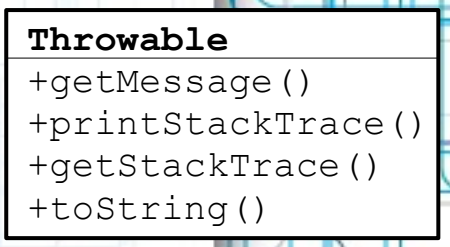
- **Hiérarchie des classes d'exceptions**



Exceptions

- **Classe `Throwable`**
  - La classe **`Throwable`** est à la base de toutes les exceptions. Elle fournit les services suivants:
    - contient un **message** (d'erreur)
    - permet de **chaîner les exceptions**
      - une exception causée par une autre exception contient une référence vers cette dernière.
    - permet d'obtenir la **trace de la pile d'exécution**
      - liste des appels de méthodes qui ont conduit au déclenchement de l'exception.
  - En pratique, toutes les exceptions en Java descendent de **`Exception`**, une sous-classe de `Throwable`.

Exceptions



- **Classe `Throwable`**
  - Constructeur (il y en a d'autres)

```
public Throwable(String message);
```

  - Récupération du message (d'erreur)

```
public String getMessage();
```

  - Obtention/affichage de la trace de pile d'exécution

```
public void printStackTrace();
public StackTraceElement[] getStackTrace();
```

  - Méthode `toString` surchargée

```
public String toString();
```

Exceptions

- **Trace de la pile d'exécution**
  - La **trace de la pile d'exécution** indique les appels successifs de méthodes (et leur localisation dans le code source) qui ont mené au déclenchement de l'exception.
  - Exemple
    - L'appel initial est celui le plus bas (`main`).
    - L'exception a eu lieu dans l'appel le plus haut (`forInputString`).

```
java.lang.NumberFormatException: For input string: "123toto"
        at java.lang.NumberFormatException.forInputString(
             NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:458)
        at java.lang.Integer.parseInt(Integer.java:499)
        at MethExcPropag.maMethode(MethExcPropag.java:5)
        at MethExcPropag.main(MethExcPropag.java:11)
```

Table des Matières

Déclenchement

- **Déclenchement d'exception**
  - Le mot-réservé **`throw`** déclenche une exception
  - Il prend en argument une instance de l'exception déclenchée (p.ex. créée avec **`new`**).
  - Syntaxe

```
throw instanceException ;
```

  - Lorsqu'une exception est déclenchée, l'exécution du programme est déroutée. Les instructions qui suivent la clause **`throw`** ne sont pas exécutées.
    - soit l'exception est capturée par un gestionnaire
    - soit le programme est terminé

Déclenchement

- **Déclenchement d'exception**
  - Exemple
    - Supposons une classe gérant un *compte bancaire* et où il n'est pas permis d'avoir un *solde négatif*. La méthode suivante permet d'effectuer un retrait.

```
public void retirer(double montant) {
  if (montant > solde)
    throw new IllegalArgumentException(
                "le montant dépasse le solde");
  solde = solde - montant;
}
```

Déclencher une exception comprend deux parties

- **instanciation** de l'exception (avec **new**)
- **déclenchement** de l'exception (avec **throw**)

Le code qui suit le déclenchement de l'exception n'est pas exécuté.

La machine virtuelle passe les instructions suivantes et retourne éventuellement aux méthodes plus basses sur la pile d'appel, jusqu'à rencontrer un gestionnaire d'exception.

Déclenchement

- **Déclenchement d'exception**
  - Exemple
    - Supposons une interface utilisateur qui demande de fournir un entier, sous forme d'une chaîne de caractères. Si la chaîne ne représente pas un entier, une exception est générée.

```
System.out.println("Quel montant retirer ?");
Scanner scanner = new Scanner(System.in);
String s = scanner.next();
int valeur = Integer.parseInt(s);
compte.retrait(valeur);
```

La méthode de classe `parseInt()` déclenche une exception `NumberFormatException` si la chaîne de caractères ne représente pas un entier.

Le code qui suit le déclenchement de l'exception n'est pas exécuté.

Capture

- **Capture d'exception**
  - La capture d'une exception est réalisée à l'aide de la directive **`try-catch`**, composée de deux blocs :
    - le bloc **`try`** entoure le code susceptible de générer une exception
    - le bloc **`catch`** permet de capturer certaines exceptions
  - Syntaxe

```
try {
  code susceptible de
  déclencher une exception
} catch ( nomClasse nomVariable ) {
  code qui traite l'exception
}
```

nomClasse désigne une sous-classe d'`Exception`

Le gestionnaire capture les instances d'exception qui sont des sous-classes de *nomClasse*.

nomVariable référence l'instance de l'exception capturée. Il s'agit d'une variable locale au bloc **`catch`**.

Capture

- **Capture d'exception**

Si aucune exception n'est générée dans le bloc **try**, alors le bloc **catch** n'est pas exécuté.

```
try {
   // pas d'exception
   int m= Integer.parseInt(s);
   // code suivant exécuté
   compte.retirer(m);

} catch (SomeException e) {
   // pas exécuté
   System.out.println(…);
}
```

Si une exception **A** est générée dans le bloc **try**, les instructions suivantes du bloc ne sont pas exécutées.

Si (**A** ***is-a*** **B**) et le bloc **catch** capture les exceptions de type **B**, alors le bloc **catch** est exécuté.

```
try {
   // exception déclenchée
   int m= Integer.parseInt(s);
   // code suivant pas exécuté
   compte.retirer(m);

} catch (SomeException e) {
   // gestion exception
   System.out.println(…);
}
```

exception de type **A**

Capture

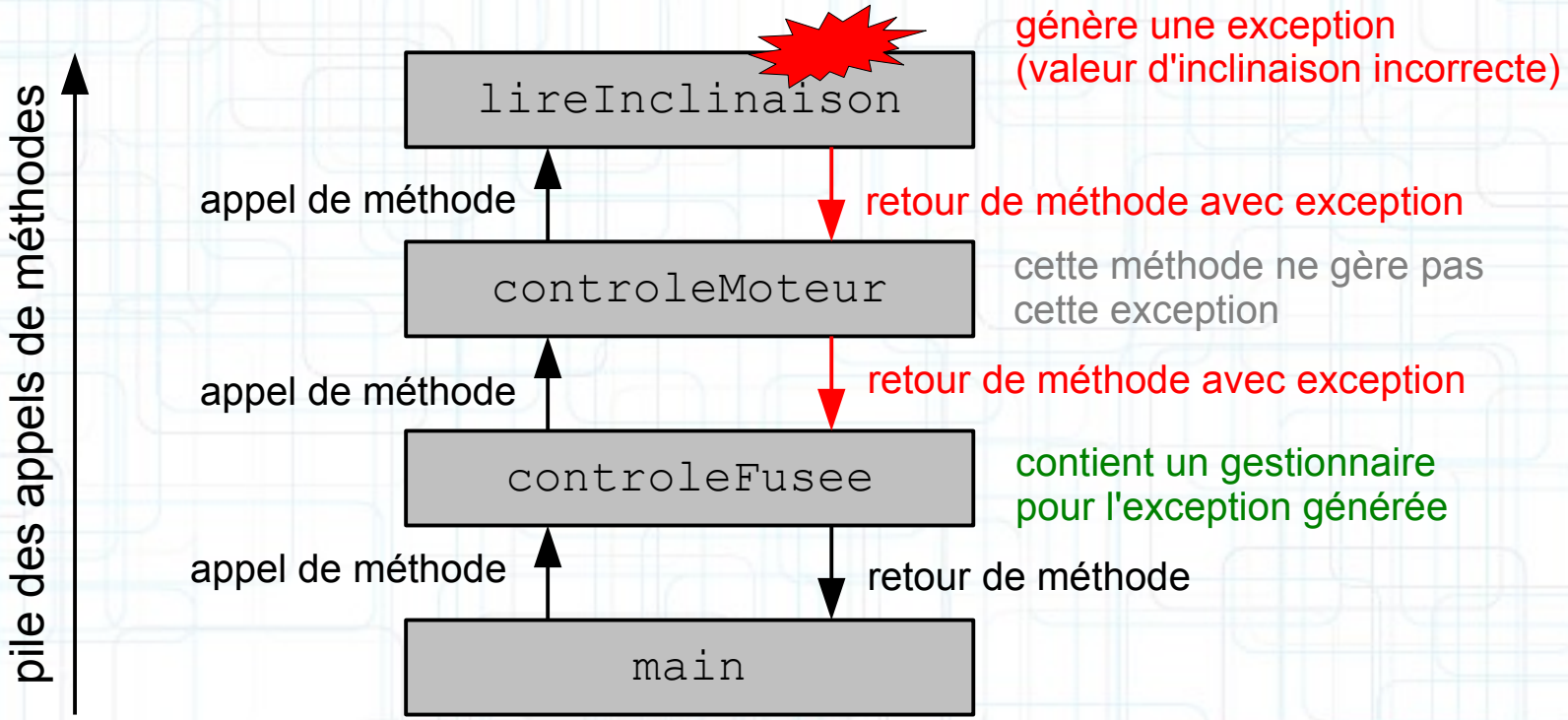
- **Capture d'exception**
  - Dans le bloc **`catch`**, la référence à l'instance de l'exception capturée est placée dans la variable déclarée dans la clause **`catch`**.
  - Exemple

```
try {
   System.out.println("Quel montant retirer ?");
   Scanner scanner = new Scanner(System.in);
   String s = scanner.next();
   int montant = Integer.parseInt(s);
   compte.retrait(montant);
} catch (NumberFormatException e) {
   System.err.println("Erreur: \"" + e.getMessage() + "\"");
   System.err.println("Le retrait n'a pas eu lieu");
}
```

```
Quel montant retirer ?
100plop
Erreur: "For input string "100plop""
Le retrait n'a pas eu lieu
```

Déclenchement → Capture

- **Déclenchement d'exception**
  - Lorsqu'une exception est générée, la machine virtuelle recherche un gestionnaire d'exception le long de la pile des appels. Si le type de l'exception correspond à celui du gestionnaire, l'exception est délivrée à ce gestionnaire.



Capture

- **Capture d'exceptions multiples**
  - Une séquence d'instructions peut déclencher des exceptions de plusieurs types. Dans ce cas, il est possible de les capturer toutes avec une seule clause **`try`** et plusieurs clauses **`catch`**.
  - Syntaxe

```
try {
  ... code qui peut déclencher une exception ...
} catch ( nomClasse1 nomVariable ) {
  ... code qui traite l'exception de type 1...
} catch ( nomClasse2 nomVariable ) {
  ... code qui traite l'exception de type 2...
} ... {
  ...
} catch ( nomClasseN nomVariable ) {
  ...
}
```

Capture

- **Capture d'exceptions multiples**
  - Les clauses **`catch`** sont évaluées dans l'ordre de leur écriture.
  - Le traitement de plusieurs types d'exceptions dans un seul bloc **`try`** nécessite quelques précautions dans le cas où l'instance de l'exception déclenchée est compatible (en relation ***is-a***) avec plusieurs clauses **`catch`**.

Ordre d'évaluation

```
try {
  ... code qui peut déclencher une exception ...
} catch ( nomClasse1 nomVariable ) {
  ... code qui traite l'exception de type 1...
} catch ( nomClasse2 nomVariable ) {
  ... code qui traite l'exception de type 2....
}
```

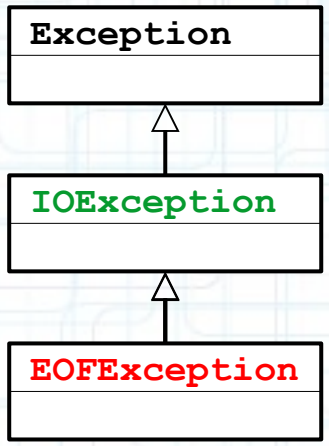
Capture

- **Capture d'exceptions multiples**
  - Plusieurs clauses **`catch`** peuvent correspondre à une exception dans le cas où la classe mentionnée dans une clause **`catch`** est une sous-classe d'une classe mentionnée dans une autre clause **`catch`**.
  - Exemple

```
try {
   ...
} catch (IOException e) {

} catch (EOFException e) {

}
```



La seconde clause catch ne sera jamais utilisée car les instances de `EOFException` correspondent à la classe parent `IOException` dans la clause qui précède !
Le compilateur génère une erreur dans ce cas.

Capture

- **Capture d'exceptions multiples**
  - La capture de multiples exceptions avec un seul bloc `try` est sensiblement différente de la forme suivante. **Quelle est la différence ?**

```
try {
    ...
    try {
        try {
          ... code qui peut déclencher plusieurs exceptions ...
        } catch ( nomClasse1 nomVariable ) {
          ... code qui traite l'exception de type 1...
        }
    } catch ( nomClasse2 nomVariable ) {
      ... code qui traite l'exception de type 2...
    }
    ...
} catch ( nomClasseN nomVariable ) {
  ... code qui traite l'exception de type N...
}
```

Note : cette forme ci peut être utile si un bloc `catch` plus imbriqué est lui-même susceptible de générer une exception !

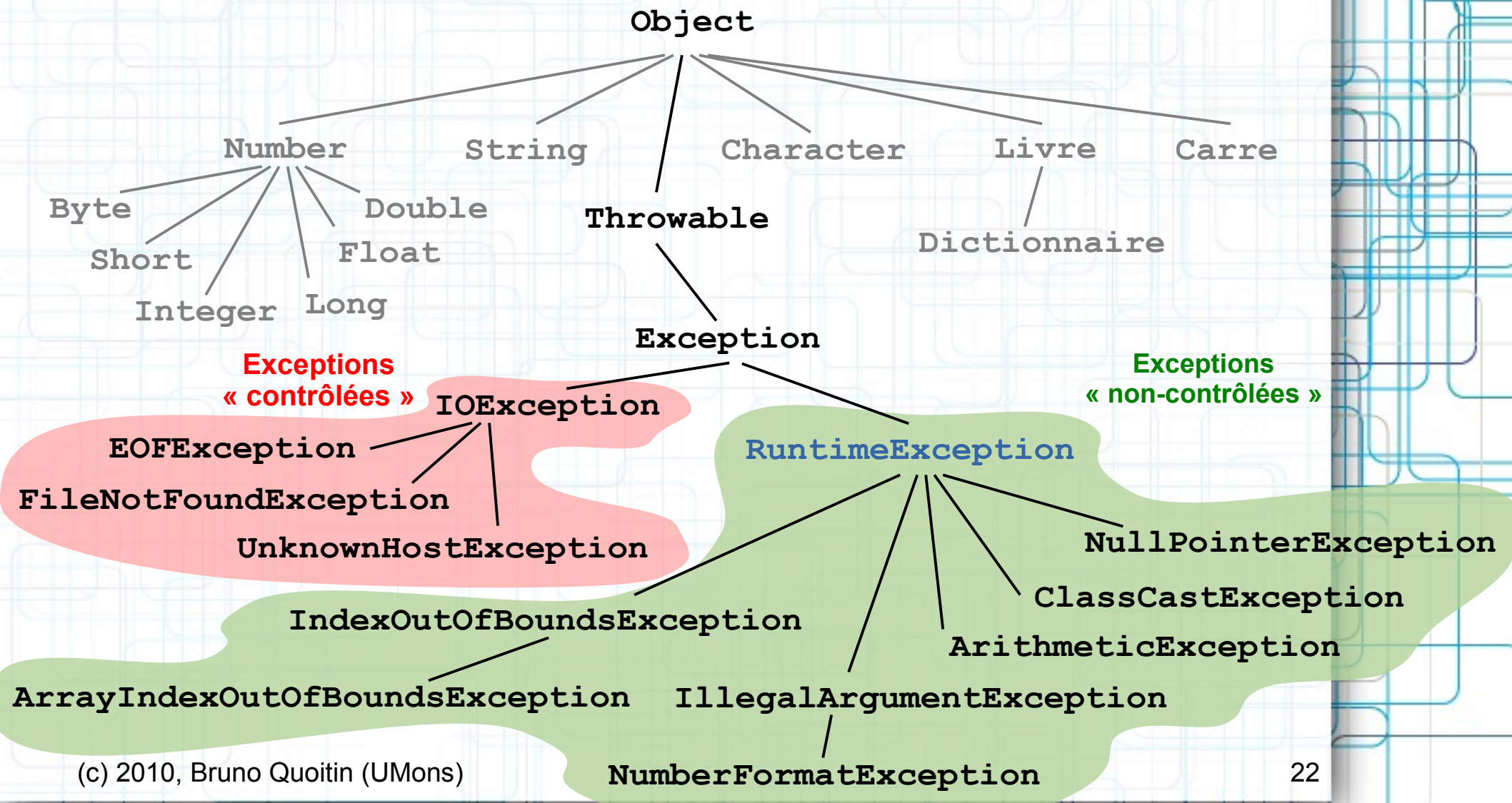
Table des Matières

Contrôlée ou non ?

- ***Checked* vs *Unchecked***
  - Il existe deux catégories d'exceptions en Java
  1. **Exceptions « contrôlées »** (*checked exceptions*).
     - Exceptions qui sont prévues par un contrat établi entre le client et l'implémentation. Le client est **forcé de vérifier** la survenance de telles exceptions.
     - Le compilateur vérifie qu'une *checked exception* est gérée.
     - Typiquement utilisées pour des erreurs récupérables.
  2. **Exceptions « non-contrôlées »** (*unchecked exceptions*).
     - Exceptions dont la survenance n'est pas explicitement prévue, comme par exemple la division par 0. Il s'agit en général d'erreurs de programmation, mais pas toujours.
     - Le compilateur n'effectue pas de vérification.
     - Sous-classes de `RuntimeException`.

Contrôlée ou non ?

- **Hiérarchie des classes d'exceptions**



Contrôlée ou non ?

- **Déclaration**
  - Les exceptions *contrôlées* **doivent être déclarées** par les méthodes susceptibles de les déclencher. Cette déclaration est effectuée à l'aide du mot-réservé **`throws`**(1).
    - Si une méthode/un constructeur déclare qu'elle/il peut déclencher une exception A, en pratique elle/il pourra déclencher toute exception B telle que (B ***is-a*** A).
  - Syntaxe pour une méthode

```
specificateurAcces  typeRetour  nomMethode  ( parametres )
throws nomClasseException1, ..., nomClasseExceptionN ;
```

  - Syntaxe pour un constructeur

```
specificateurAcces  nomClasse  ( parametres )
throws nomClasseException1, ..., nomClasseExceptionN ;
```

(1) Attention : il s'agit d'un mot-clé différent de **`throw`** (sans « s » final) qui sert à déclencher une exception.

Contrôlée ou non ?

- **Déclaration**
  - Exemple
    - La classe FileInputStream permet de créer un flux de lecture à partir d'un fichier. Plusieurs de ses méthodes déclarent qu'elles sont susceptibles de générer des exceptions.

```
public class FileInputStream {
   ...
   public FileInputStream(String name)
      throws FileNotFoundException
   { /* ... */ }

   public byte read()
      throws IOException
   { /* ... */ }

   ...
}
```

Déclare que le constructeur est susceptible de déclencher une exception de type `FileNotFoundException`.

Déclare que la méthode est susceptible de déclencher une exception de type `IOException`.

Contrôlée ou non ?

- **Vérification**
  - Dans le cas où le programme fait appel à une méthode qui déclare pouvoir déclencher une « *checked* » exception d'un type donné, le compilateur vérifie que le programme capture l'exception. Si ça n'est pas le cas, une erreur de compilation est générée.
  - Exemple

```
public void ouvreFichier(String s)
{
  FileInputStream fis=
    new FileInputStream("/tmp/toto.txt");
  ...
}
```

Le constructeur de `FileInputStream` peut déclencher une exception.

Par conséquent, en absence d'un gestionnaire d'exception, la méthode `ouvreFichier` est aussi susceptible de générer une exception.

```
bash-3.2$ javac FIS.java
FIS.java:7: unreported exception java.io.FileNotFoundException; must
be caught or declared to be thrown
    FileInputStream fis= new FileInputStream("/tmp/toto.txt");
```

Contrôlée ou non ?

- **Propager l'exception**
  - Le client peut ne pas traiter l'exception mais la propager à l'appelant. Il lui faut pour cela déclarer qu'il peut également déclencher l'exception, via la clause **`throws`**.
    - L'exception déclenchée doit être en relation ***is-a*** avec l'exception propagée.
  - Exemple

```
public void ouvreFichier(String s)
  throws IOException
{
  FileInputStream fis=
    new FileInputStream("/tmp/toto.txt");
  ...
}
```

La méthode `ouvreFichier` ne gère pas l'exception localement. Elle propage l'exception éventuelle à l'appelant.

Le constructeur `FileInputStream` peut déclencher une *checked exception* de type `FileNotFoundException`.

Contrôlée ou non ?

- **Exceptions *non-contrôlées***
  - Il est possible de déclarer une *unchecked exception* dans une clause **`throws`**.
  - Cependant, cela n'en fait pas une exception contrôlée (*checked*). Le compilateur ne fera pas de vérification de la capture/propagation.
  - Exemple

```
public class Integer extends Number {

  public static int parseInt(String s)
    throws NumberFormatException;

}
```

Le mot-clé **`throws`** indique que la méthode `parseInt` est susceptible de déclencher une exception de type `NumberFormatException`. Cependant, il s'agit d'une *unchecked exception* (sous-classe de `RuntimeException`)

→ **le compilateur n'impose pas que l'exception soit capturée ou propagée !**

Table des Matières

Bonnes pratiques

- **Quand utiliser une exception ?**
  - L'utilisation d'exceptions doit rester... ***exceptionnelle***... c'est-à-dire n'être utilisée que pour signaler une condition inhabituelle du programme, généralement dépendante de l'environnement ou des entrées/sorties.
    - entrée utilisateur / argument invalide
    - fichier inexistant, erreur de lecture, format non-respecté
    - ...
  - On peut aussi noter que **déclencher une exception a un coût** qui peut correspondre au temps d'exécution de plusieurs dizaines de lignes de code(1).

(1) Ce coût provient majoritairement de l'instanciation de l'exception : durant cette instanciation, la trace de la pile d'exécution est établie. Source : **Java Performance Tuning (2nd edition)**, Jack Shirazi, O'Reilly, 2003

Bonnes pratiques

- **Ne pas capturer toutes les exceptions**
  - Il est recommandé de ne pas capturer « aveuglément » l'ensemble des exceptions qui peuvent survenir.
  - Exemples

```
try {
   ...
} catch (Exception e) {
  System.out.println("Une exception a eu lieu:"+
    e.getMessage());
}
```

```
try {
   ...
} catch (Exception e) {
  e.printStackTrace();
}
```

Ces approches ne sont pas recommandées !

  - Il est préférable d'être sélectif dans la capture en spécifiant les classes que l'on veut capturer.

Bonnes pratiques

- **Ne pas mettre de blocs `catch` vides**
  - Une tentation des débutants en programmation est de se « débarrasser » de la gestion des exceptions en définissant des blocs `catch` vides ou en se limitant à afficher un message d'erreur.
  - Exemple

```
try {
   ...
} catch (Exception e) {
}
```

Capture toute instance d'une sous-classe d'`Exception`, i.e. toute exception même *unchecked* !

Puis se comporte comme si rien ne s'était passé !...

Il s'agit d'une très mauvaise pratique !

  - Si on ne sait pas quoi faire d'une exception, il est préferable de la « convertir » en *unchecked exception* !

```
try {
   ...
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

Bonnes pratiques

- **Quand ne pas utiliser une exception ?**
  - Exemple à éviter : utilisation d'une exception plutôt que les mécanismes normaux de contrôle de flux du programme (conditions d'arrêt des boucles, etc.)

```
public void stupid() {
  try {
    while (true)
      incrementerCompteur();
  } catch (MaximumCountReachedException ex) { }
  // Continuer execution
}

public void incrementerCompteur() throws MaximumCountReachedException
{
  compteur++;
  if (compteur >= 5000)
    throw new MaximumCountReachedException();
}
```

Note : exemple inspiré de « **Best Practices for Exception Handling** », Gunjan Doshi, http://onjava.com/pub/a/onjava/2003/11/19/exceptions.html

Bonnes pratiques

- **Exceptions *non-contrôlées***
  - Les ***exception non-contrôlées*** ne doivent généralement pas être capturées car elles signalent des *erreurs irrécupérables*.
  - Exemple
    - La *division par zéro*, l'*accès en dehors des bornes* d'un tableau ou l'usage d'une *référence nulle* génèrent des *exceptions non-contrôlées*.

```
double x= 0;
// --> ArithmeticException
double y= 1/x;
```

```
int[] array= new int[10];
// --> ArrayIndexOutOfBoundsException
array[10]= 123;
```

Les *unchecked exceptions* signalent des *erreurs irrécupérables* (typ. des erreurs de programmation). **Il n'est généralement pas approprié de capturer ces exceptions !**

  - Pour cette raison, le compilateur ne force pas le programmeur à déclarer, capturer ou propager ces exceptions.
  - Exception à cette règle : `NumberFormatException`

Bonnes pratiques

- **Capturer ou propager ?**
  - Décision difficile !
  - **Capturer une exception**
    - Uniquement si on sait réagir.
    - Sinon, convertir en *unchecked exception* si on ne souhaite pas propager explicitement (p.ex. `RuntimeException`).
  - **Propager une exception**
    - Peut casser l'encapsulation en révélant le fonctionnement interne d'une classe (par exemple propager `SQLException` indique que la classe se base sur un gestionnaire de bases de données et utilise SQL pour l'interroger).
    - Eventuellement, convertir en une autre exception afin de cacher les détails d'implémentation. Il peut être utile de définir sa propre classe d'exceptions pour cela.

Bonnes pratiques

- **Capturer ou propager ?**
  - Conception de ses propres exceptions
    - Se poser la question : « n'existe-t-il pas déjà une classe d'exception appropriée dans la bibliothèque java ? »
    - Eviter de définir une nouvelle classe d'exceptions si elle ne comporte pas d'information supplémentaire permettant de réagir à l'exception.
    - Décider *checked* ou *unchecked* ? Si le client peut essayer de récupérer après l'erreur, utiliser *checked*. Sinon *unchecked*.
    - Exception pour signaler des erreurs de programmation → préférer *unchecked*.