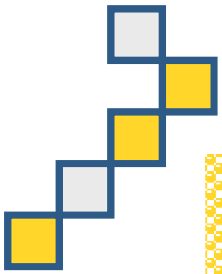


Informatique / Programmation

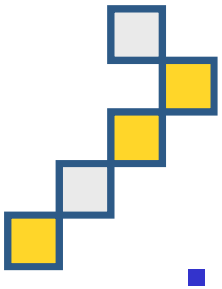
Programmation orientée objet avec Java

06 : Les exceptions et leur traitement

J. + F. Bapst

frederic.bapst@hefr.ch

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg



Exceptions

- Les **exceptions** représentent des **événements** qui peuvent survenir durant l'exécution d'un programme et qui **rompent le flux normal** des instructions.
- Les **exceptions** sont principalement utilisées pour représenter des **erreurs** de différents types : des erreurs matérielles (crash du disque, ...) des erreurs de programmation (indice d'un tableau hors limites, ...) des erreurs liées à l'environnement d'exécution (mémoire insuffisante, ...), des erreurs spécifiques à une librairie (matrice singulière) ou à une application (numéro d'article non-défini), etc.
- Les **exceptions** peuvent également représenter des événements qui en sont pas à proprement parler des erreurs, mais qui correspondent à des **situations exceptionnelles** (prévues) qui doivent être traitées de manières différentes du flux normal des opérations (par exemple la fin d'un fichier, un mot de passe incorrect, des ressources momentanément non-disponibles, …).

Avantages des exceptions

- Augmentent la **lisibilité** du code en séparant les instructions qui traitent le cas normal des instructions nécessaires au traitement des erreurs ou des événements exceptionnels.
- Permettent (voire imposent) la **déclaration explicite** des exceptions qu'une méthode peut lever (fait partie de la signature).
- Forcent le programmeur à **prendre en compte** (traiter ou propager) les cas exceptionnels (erreurs ou autres événements) qui sont déclarés dans les méthodes qu'il invoque.
- Permettent de créer une **hiérarchie** d'événements et de les traiter avec une granularité différente selon les situations.
- Offrent un mécanisme de **propagation automatique** ce qui permet au programmeur de choisir à quel niveau il souhaite traiter l'exception (celui auquel il est à même de prendre les mesures adéquates).

Gestion des exceptions

- En Java, les exceptions sont représentées par des objets de type **`Throwable`**. Il existe plusieurs familles d'exceptions représentées par différentes sous-classes de `Throwable` (par ex. **`Exception`**).
- Différentes instructions permettent de gérer les exceptions.
- L'instruction **`throw`** sert à **générer une exception** (on dit également **lever une exception**, **lancer une exception**, …).
- L'instruction **`try`** / **`catch`** / **`finally`** constitue le cadre dans lequel les exceptions peuvent être détectées (capturées) et traitées.
- Le mot-clé **`throws`** (à ne pas confondre avec `throw`) est utilisé dans la déclaration de méthode (signature) pour **annoncer la liste des exceptions** que la méthode peut générer. L'utilisateur de la méthode est ainsi informé des exceptions qui peuvent survenir lors de son invocation et peut prendre les mesures nécessaires pour gérer ces événements exceptionnels (c'est-à-dire les **traiter** ou les **propager**).

Générer (lever) une exception [1]

- Les exceptions peuvent être générées soit :
  - par le **système**, lors de l'exécution de certaines instructions
  - par l'exécution de l'**instruction** `throw` (dans les classes pré-définies de la plate-forme Java [librairies] ou dans le code que l'on écrit soi-même)
- Parmi les instructions qui génèrent des exceptions, on peut citer :
  - La division entière par zéro qui génère `ArithmeticException`
  - L'indexation d'un tableau hors de ses limites qui génère `ArrayIndexOutOfBoundsException`
  - L'utilisation d'un tableau ou objet dont la référence vaut `null` qui génère `NullPointerException`
  - La création d'un tableau avec une taille négative qui génère `NegativeArraySizeException`
  - La conversion d'un objet dans un type non compatible qui génère `ClassCastException`
  - . . .

Générer (lever) une exception [2]

- Pour générer explicitement une exception, on utilise l'instruction **`throw`** :

```
throw exception_object ;
```

- L'expression qui suit l'instruction `throw` doit être un objet qui représente une exception (un objet de type `Throwable`).
- Lors de la création de l'objet exception, on peut généralement lui associer un **message** (`String`) qui décrit l'événement.

```
public static long factorial(int x) throws Exception {
  long result = 1;
  if (x<0)
    throw new Exception("x doit être >= 0");
  while (x > 1) {
    result *= x;
    x--;
  }
  return result;
}
```

Traiter une exception [1]

- Les instructions susceptibles de lever des exceptions peuvent être insérées dans un bloc **try** / **catch** qui se présente de la manière suivante :

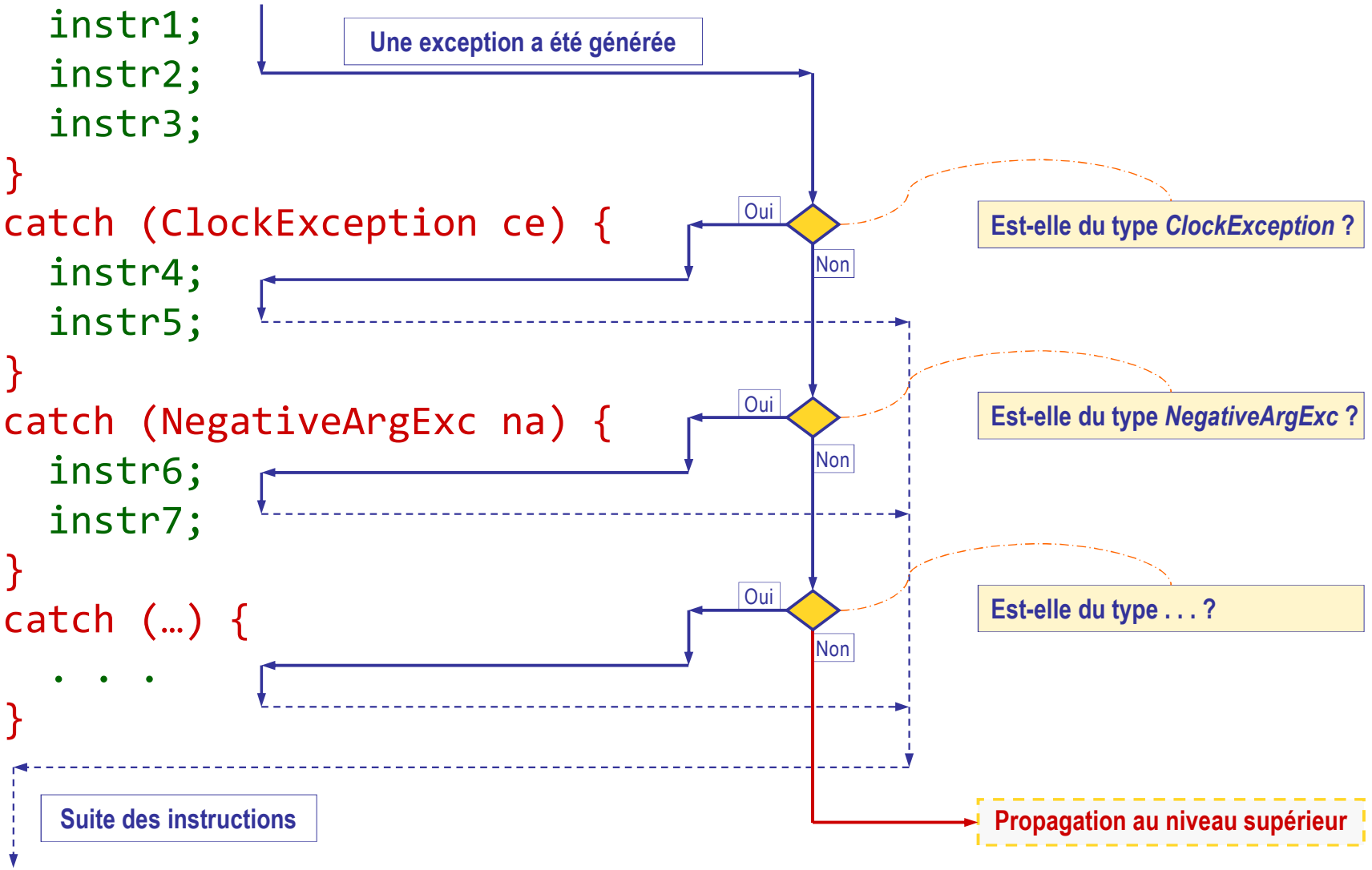
```
try {
  // Bloc contenant des instructions pouvant générer des exceptions.
}
catch (ExceptionType1 e1) {
  // Bloc contenant les instructions qui traitent (capturent) les exceptions
  // du type ExceptionType1 (ou d'une de ses sous-classes)
  // On peut référencer l'objet exception à l'aide de la variable e1.
}
catch (ExceptionType2 e2) {
  // Bloc contenant les instructions qui traitent (capturent) les exceptions
  // du type ExceptionType2 (ou d'une de ses sous-classes).
  // On peut référencer l'objet exception à l'aide de la variable e2.
}
catch (…) {
  …                    // Et ainsi de suite…
}
```

Traiter une exception [2]

- Si une des instructions contenues dans le bloc `try`, lève une exception, le contrôle est passé au premier bloc `catch` dont le type d'exception correspond à l'exception qui a été levée (même classe ou classe parente de l'exception levée).
- Après l'exécution de la dernière instruction du bloc `catch` considéré, le contrôle est passé à l'instruction qui suit le dernier bloc `catch` (ou à la clause `finally` s'il y en a une).
- Si aucun bloc `catch` ne correspond au type d'exception qui a été levée, **l'exception est propagée** au niveau supérieur c'est-à-dire que le contrôle est transféré au traitement d'exception de la méthode invoquante ou du bloc englobant. Si aucun traitement n'existe au niveau supérieur pour ce type d'exception, la propagation se poursuit jusqu'à trouver un bloc `catch` traitant cette exception. Si ce n'est pas le cas, le programme se termine avec un message d'erreur sur la console de sortie *(Stack Trace)*.

Traiter une exception [3]

```
try {
   instr1;
   instr2;
   instr3;
}
catch (ClockException ce) {
   instr4;
   instr5;
}
catch (NegativeArgExc na) {
   instr6;
   instr7;
}
catch (…) {
   . . .
}
```



Traiter une exception [4]

- Les divers types d'exception peuvent appartenir à **différentes familles** (voir hiérarchie de classes, en fin de chapitre).
- Une exception spécialisée (par ex. `FileNotFoundException`) peut faire partie d'une famille plus vaste (`IOException`) qui elle-même fait partie d'une famille plus générale (`Exception`) etc.
- Si plusieurs clause `catch` sont compatibles avec le type d'exception qui a été levée (font partie de la même famille), **c'est la première qui capturera l'exception** et se chargera du traitement.

```
try {
   . . .
} catch (FileNotFoundException notFound) {
   . . .
} catch (IOException ioErr) {
   . . .
} catch (Exception genErr) {
   . . .
}
```

L'ordre des clauses `catch` est important !

Traiter une exception [5]

- Il est possible de traiter plusieurs types d'exceptions dans une seule clause `catch` en utilisant le symbole '`|`' comme séparateur.
- Les exceptions mentionnées dans une telle liste doivent être "indépendantes" sur le plan de leur relation hiérarchique. Une des exceptions ne peut pas être un ancêtre *(super-classe)* d'une autre (dans ce cas, mentionner l'exception ancêtre suffit pour traiter toutes celles de sa "famille").

```
try {
   . . .
} catch (UnknownUser | IncorrectPassword loginError) {
   . . .
} catch (Exception otherError) {
   . . .
}
```

Traiter une exception [6]

- Lorsqu'un problème a été détecté (une exception a été générée) et que l'on est en mesure de le traiter (au moins partiellement), il y a **différentes mesures** que l'on peut envisager, selon les cas, pour régler la situation.
- Dans un traitement d'exception (bloc `catch`) on peut par exemple :
  - Régler le problème et recommencer le traitement (nécessite généralement un boucle). Idéal mais pas toujours possible.
  - Faire quelque chose d'autre à la place (algorithme de substitution).
  - Sortir de l'application (`System.exit()`) après affichage d'un message (et/ou de la *Stack-Trace*), écriture dans un fichier *log*, etc.
  - Re-générer l'exception (après avoir effectué certaines opérations).
  - Générer une nouvelle exception (après avoir éventuellement effectué certaines opérations).
  - Retourner une valeur spéciale ou valeur par défaut (pour une fonction).
  - Terminer la méthode (si elle n'a pas de valeur de retour).
  - Ne rien faire (ou afficher un message) et continuer (c'est rarement une bonne solution).

Propagation des exceptions [1]

```
public class PropEx {

  public static void main(String[] args) {
    System.out.println("main starts");
    b(5);
    b(0);
    System.out.println("main ends");
  }

  //-----------------------------------------------
  //  b
  //-----------------------------------------------
  public static void b(int i) {
    System.out.println("b starts");
    try {
      System.out.println("b step 1");
      c(i);
      System.out.println("b step 2");
    }
    catch (Exception e) {
      System.out.println("b catches " + e);
    }
    System.out.println("b ends");
  }
```

// Suite...

Propagation des exceptions [2]

// …Suite

```
  //-----------------------------------------------
  //  c
  //-----------------------------------------------
  public static void c(int i) throws Exception {
    System.out.println("c starts");
    d(i);
    System.out.println("c ends");
  }

  //-----------------------------------------------
  //  d
  //-----------------------------------------------
  public static void d(int i) throws Exception {
    System.out.println("d starts");
    int a=10/i;    // Cette instruction peut générer une exception
    System.out.println("d ends");
  }
}
```

Propagation des exceptions [3]

- Dans le programme `PropEx`, lors de la deuxième invocation de la méthode `b()`, une exception sera levée dans la méthode **`d()`** (division par zéro).
- Le déroulement de l'application sera alors altéré et **l'exception sera propagée** jusqu'à la méthode `b()` qui dispose d'un bloc **`catch`** pour traiter cette exception.

`b(5)` 1ère invocation	**`b(0)`** 2ème invocation

`main starts`
`b starts`
`b step 1`
`c starts`
`d starts`
`d ends`
`c ends`
`b step 2`
`b ends`
…
	…
	`b starts`
	`b step 1`
	`c starts`
	`d starts`
	`b catches ArithmeticException: / by zero`
	`b ends`
	`main ends`

Interprétation de la *Stack-Trace* [1]

- Si, dans l'exemple précédent (**`PropEx`**), on supprime tout traitement d'exception, l'exception provoquée par la division par zéro (dans la méthode `d()`) sera propagée jusqu'à la méthode `main()` qui sera alors interrompue avec affichage du nom de l'exception et de l'état de la pile des appels (***Stack-Trace***) :

```
java.lang.ArithmeticException: / by zero
        at javabasic.PropEx.d(PropEx.java:43)
        at javabasic.PropEx.c(PropEx.java:37)
        at javabasic.PropEx.b(PropEx.java:24)
        at javabasic.PropEx.main(PropEx.java:13)

Exception in thread "main"
```

- Il est important de savoir interpréter ces informations de manière à localiser rapidement la cause du problème qui a causé l'interruption de l'application et agir au bon endroit.
- La page suivante décrit comment interpréter cette *Stack-Trace*.

Interprétation de la *Stack-Trace* [2]

- La ***Stack-Trace*** décrit (dans l'ordre inverse) la séquence (pile) des appels qui ont conduit à l'interruption de l'application.

```
java.lang.ArithmeticException: / by zero
      at javabasic.PropEx.d(PropEx.java:43)
      at javabasic.PropEx.c(PropEx.java:37)
      at javabasic.PropEx.b(PropEx.java:24)
      at javabasic.PropEx.main(PropEx.java:13)

Exception in thread "main"
```

- La première ligne affichée correspond au type d'exception qui a été générée (avec, éventuellement, le texte du message qui lui est associé).
- On trouve ensuite l'enchaînement des invocations de méthodes où chacune des lignes est structurée de la manière suivante :

```
. . .
at Package.Classe.Méthode(Fichier_Source:No_de_ligne)
. . .
```

- Dans le cas d'une application, on trouvera sur la dernière ligne de la séquence des appels, la méthode `main()` qui constitue le point d'entrée du programme.

Clause `finally` [1]

- Une clause `finally` peut être ajoutée à une instruction `try` / `catch`.
- Cette clause définit un bloc d'instructions qui seront exécutées à la fin de l'instruction `try` / `catch` indépendamment du fait que des exceptions aient été levées ou non. Le bloc `finally` sera exécuté **dans tous les cas**.
- Si aucune exception n'est levée dans le bloc `try` , le bloc `finally` sera exécuté après la dernière instruction du bloc `try`.
- Si une exception est levée dans le bloc `try` et est capturée par un bloc `catch`, le bloc `finally` sera exécuté après la dernière instruction du bloc `catch`.
- Si une exception est levée dans le bloc `try` et n'est pas capturée par un bloc `catch`, le bloc `finally` sera exécuté avant la propagation de l'exception au niveau supérieur.

Clause `finally` [2]

- Un bloc `finally` est généralement utilisé pour effectuer des opérations de conclusion (fermeture de fichiers, de connexion réseau, de base de données, etc.) qui devraient être effectuées dans tous les cas de figure.

  Le bloc `finally` évite donc de devoir placer ces instructions de conclusion dans le bloc `try` et dans tous les blocs `catch`.

- L'utilisation des instructions `break`, `continue`, `return` ou `throw` (dans les blocs `try` ou `catch`) n'empêche pas l'exécution préalable du bloc `finally`.

- Le bloc `finally` est optionnel mais un bloc `try` doit obligatoirement être accompagné d'au moins un bloc `catch` ou d'un bloc `finally` (ou naturellement des deux).

Clause `finally` [3]

- Exemple d'utilisation de la clause `finally` :

```
try {
  openFile(f);
  content= readFile(f);
  print(content);
}
catch (InvalidData invdat) {
  System.out.println("Les données du fichier sont erronées");
}
catch (PrintError prerr) {
  System.out.println("Erreur durant l'impression");
}
finally {
  closeFile(f);   //--- Effectué dans tous les cas
}
```

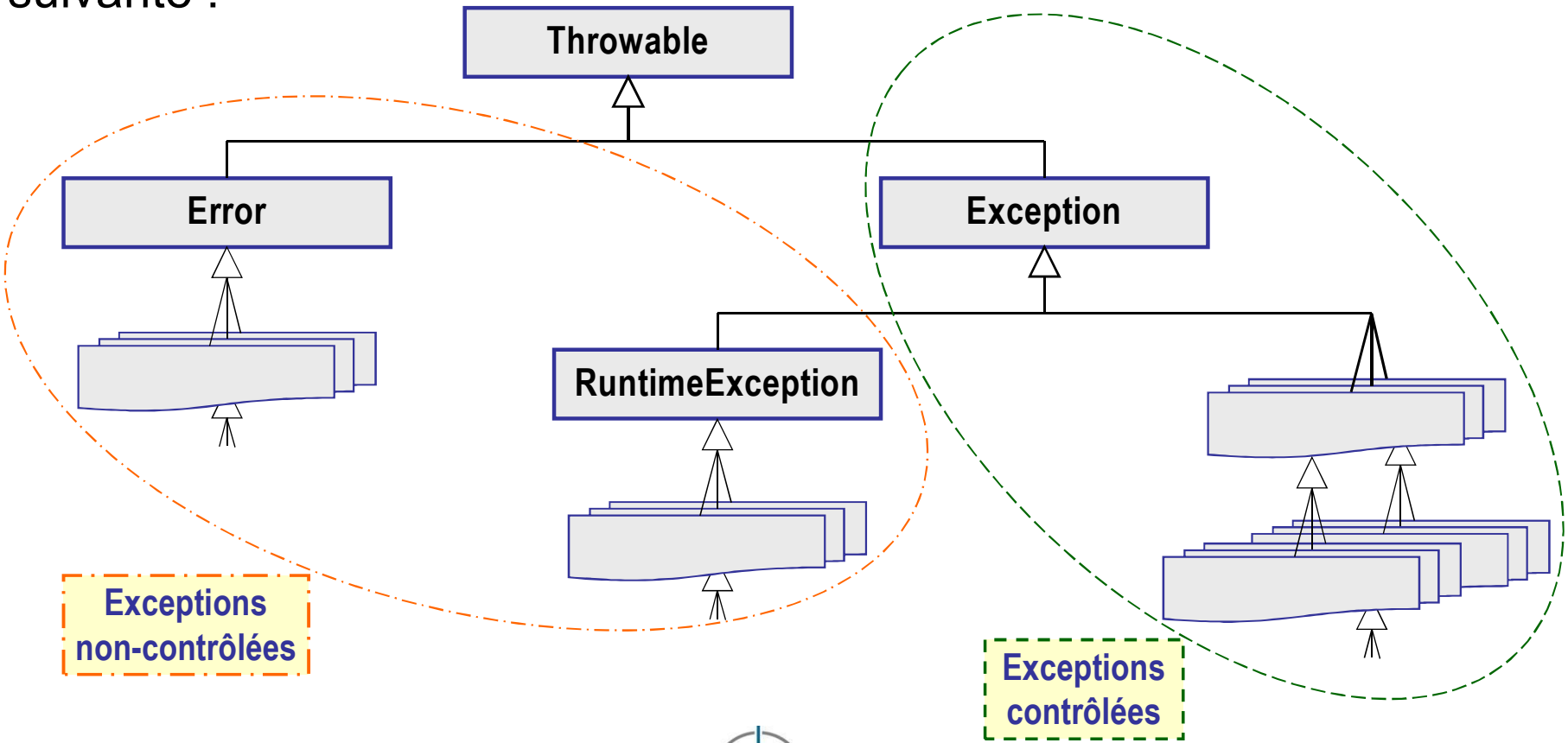
Objet exception

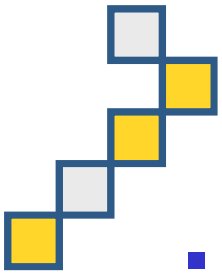
- L'objet exception sert principalement de marqueur pour l'événement qui lui est associé.
- Dans un traitement d'exception (bloc `catch`), l'objet exception est transmis et il peut être utilisé pour obtenir plus d'informations concernant l'événement.
- Quelques méthodes que l'on peut utiliser avec les objets exception :

`getMessage()`	:	retourne un `String` contenant le message (texte) associé à l'exception
`toString()`	:	retourne un `String` contenant le nom de l'exception suivi du message associé
`printStackTrace()`	:	affiche sur la console de sortie le nom de l'exception, le message associé ainsi que l'état de la pile des appels qui ont conduit au traitement de l'exception *(Stack-Trace)*
`getStackTrace()`	:	retourne les informations de la *Stack-Trace* sous forme d'un tableau de `StackTraceElement`

Hiérarchie de classes

- Dans le langage *Java*, toutes les exceptions sont représentées par des objets qui spécialisent la classe de base `Throwable`.
- Trois sous-classes de `Throwable` sont pré-définies : `Error`, `Exception` et `RuntimeException` selon l'arborescence suivante :





Exceptions contrôlées / non-contrôlées

- Les **exceptions contrôlées *(checked)*** [du type *Exception*] doivent être soit traitées (dans une clause `catch`), soit propagées pour être traitées à un niveau supérieur (annoncée avec `throws` dans l'en-tête).
- Elle ne peuvent pas être ignorées (le compilateur génère une erreur dans ce cas).

- Pour les **exceptions non-contrôlées *(unchecked)*** [du type *RuntimeException* ou *Error*], le programmeur a le choix de les traiter ou de les ignorer (sans erreur à la compilation).
- Si une telle exception survient et qu'elle n'est traitée nulle part, elle sera alors propagée et "remontera" jusqu'à la méthode `main()` interrompant ainsi brutalement l'application.

La classe Error

- La classe `Error` représente généralement des erreurs sévères qui surviennent dans la machine virtuelle.
- Le langage n'impose par que les programmeurs traitent ce type d'exceptions car elles ne devraient pas survenir dans un environnement sain (machine virtuelle correctement installée).
- On ne traite généralement pas ce type d'exceptions dans une application standard sauf éventuellement au niveau le plus haut pour informer l'utilisateur (si c'est encore possible) et éviter le crash brutal de l'application.
- D'autre part, les applications ne devraient pas générer (dans des instructions `throw`) ce genre d'exceptions.
- Exemples : `OutOfMemoryError`, `StackOverflowError`, `AssertionError` (Erreur de conception)

La classe Exception

- La classe **`Exception`** représente la classe de base de la plupart des exceptions qui sont générées (`throw`) et traitées (`catch`) dans les applications (exceptions contrôlées).
- Un grand nombre de sous-classes de `Exception` sont pré-définies dans les classes de la plate-forme *Java*
- Exemples : `PrintException, IOException`
- Les exceptions spécifiques à une librairie ou à une application sont généralement représentées par des sous-classes de `Exception`.
- La classe `RuntimeException` représente une sous-classe particulière de `Exception` (voir page suivante).
- Le langage impose que les exceptions de type `Exception` (qui ne sont pas une sous-classe de `RuntimeException`) soient traitées par les applications (en plaçant les instructions critiques dans un bloc `try` / `catch`, ou en déclarant l'exception dans la signature de la méthode à l'aide du mot-clé `throws` déléguant ainsi le traitement au niveau supérieur).

La classe RuntimeException

- La classe **`RuntimeException`** représente une sous-classe particulière (exceptions non-contrôlées) de la classe `Exception`.
- Elle représente des erreurs qui sont détectées par la machine virtuelle durant l'exécution de l'application.
  Par exemple, l'exception `NullPointerException` indique que la référence à un objet n'est pas définie (`null`).
  Cette exception peut potentiellement survenir lors de l'exécution de n'importe quelle instruction qui manipule un objet.
- De ce fait, le langage n'impose pas que les exceptions de type `RuntimeException` (et de ses sous-classes) soient traitées par les applications (ce qui serait extrêmement lourd et contraignant).
- Si nécessaire, les applications peuvent cependant traiter ce type d'exceptions (dans un bloc `try` / `catch`) et prendre ainsi les mesures adéquates.
- Exemples : `ArithmeticException`, `ArrayIndexOutOfBoundsException`

Créer de nouveaux types d'exceptions

- Pour créer ses **propres types d'exceptions**, il suffit de dériver (spécialiser) une classe de type `Throwable` (choisir une des classes existantes proche de celle que l'on souhaite créer ou, sinon, dériver la classe générale `Exception`).
- Généralement, dans cette sous-classe, on crée uniquement deux constructeurs : un constructeur sans paramètre et un constructeur qui prend un message (`String`) en paramètre.
- On crée rarement de nouveaux champs ou de nouvelles méthodes.

```
public class ClockException extends Exception {

  public ClockException() {
    super();
  }

  public ClockException(String message) {
    super(message);
  }
}
```

A prendre tel quel pour l'instant *(sera étudié ultérieurement)*

Utiliser des exceptions personnalisées

- L'utilisation des types d'exceptions que l'on a créés est identique à l'utilisation des types d'exceptions pré-définis.

```
. . .
if (hour < 0 || hour > 23) {
  throw new ClockException("Heure incorrecte");
}
. . .
```

```
. . .
try {
  . . .
} catch (ClockException clockErr) {
  System.out.println(clockErr);
  . . .
} catch (Exception otherErr) {
  . . .
}
. . .
```

Try-with-resources

Complément

- Depuis Java 1.7, il y a une variante très utile/élégante d'instruction `try`, avec invocation automatique de la méthode `close()`
  (très pratique quand on fait des entrées/sorties)

```
public static String firstLine(String filename)
                                        throws IOException {

    try ( FileReader     fp = new FileReader(filename);
          BufferedReader br = new BufferedReader(fp) ) {

        return br.readLine();

    }
    // this will also invoke br.close() and fp.close();
  }
```

- Le **try()** attend des déclarations de variables d'un type offrant une méthode `close()` (toute classe implémentant `AutoCloseable`)
- Ce **try()** peut aussi être combiné avec `catch` et `finally`