Java - 1ère année

Exceptions, gestion d'erreur

Cours 9

Exception

- Une exception est littéralement un « événement exceptionnel ».
- Un événement exceptionnel est une situation « anormale » du programme, une erreur, à un certain moment (***une division par zéro provoque une exception de type arithmeticException).***
- A un certain moment du programme, nous sommes dans une certaine fonction.
- La succession des fonctions constitue la **Pile d'appels**

Traitement d’erreur (2)

- Exemple : une fonction qui cherche un nom dans un fichier.
  - fin normale positive : je trouve le nom
  - fin normale négative : je ne trouve pas le nom
  - fin anormale :
    - je ne peux pas ouvrir le fichier ou
    - erreur de lecture
- En programmation simple, toutes les situations d’erreur utilisent le retour de la fonction.

Exemple

```
// DivParZero.java
class DivParZero {
    public static void main (String argv[] ) {
        int val=0;
        val = 1997/val;
        System.out.println("Fin du programme");
    }
}
```

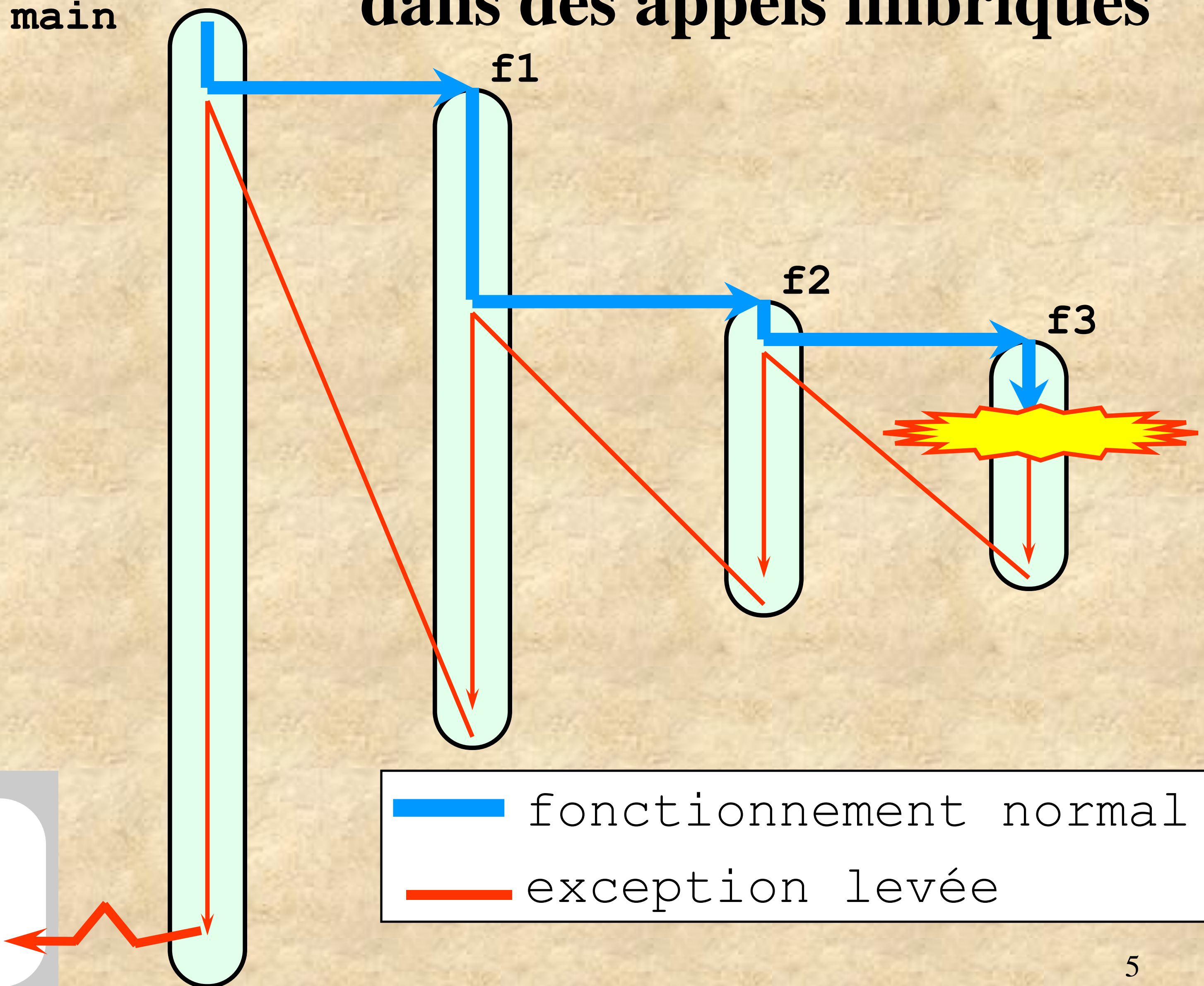
Exécution de ce programme , on obtient l'affichage suivant : (message par défaut)

java.lang.ArithmeticException: / by zero
at DivParZero.main(DivParZero.java:6)

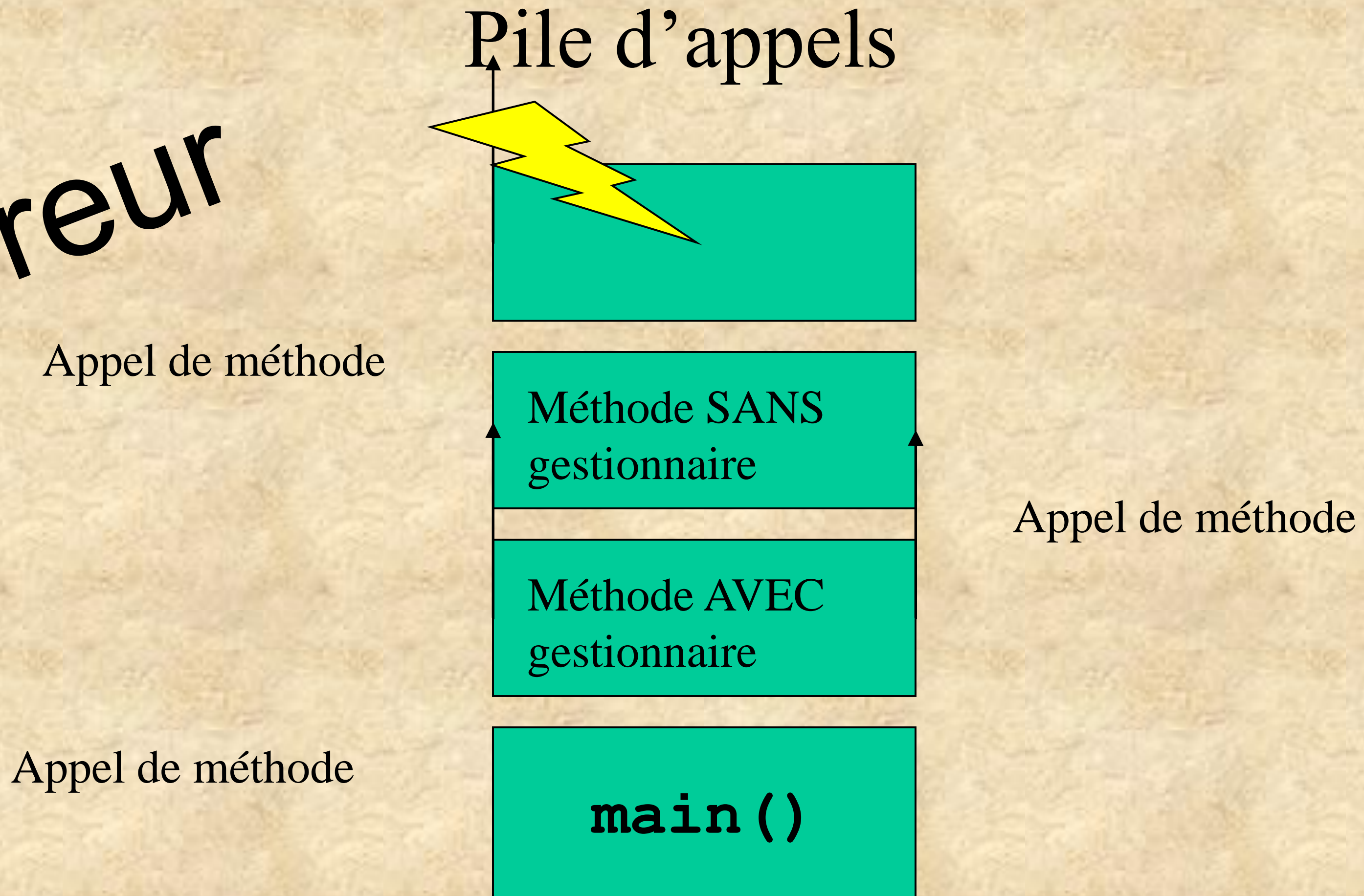
On distingue:

le nom complet de l'exception qui a été *levée*,
un message précisant la cause de cette erreur est envoyé par l'objet de type exception (/by zero),
l'indication de la classe, de la méthode et du numéro de ligne où s'est produite cette exception.

Propagation d'une exception dans des appels imbriqués

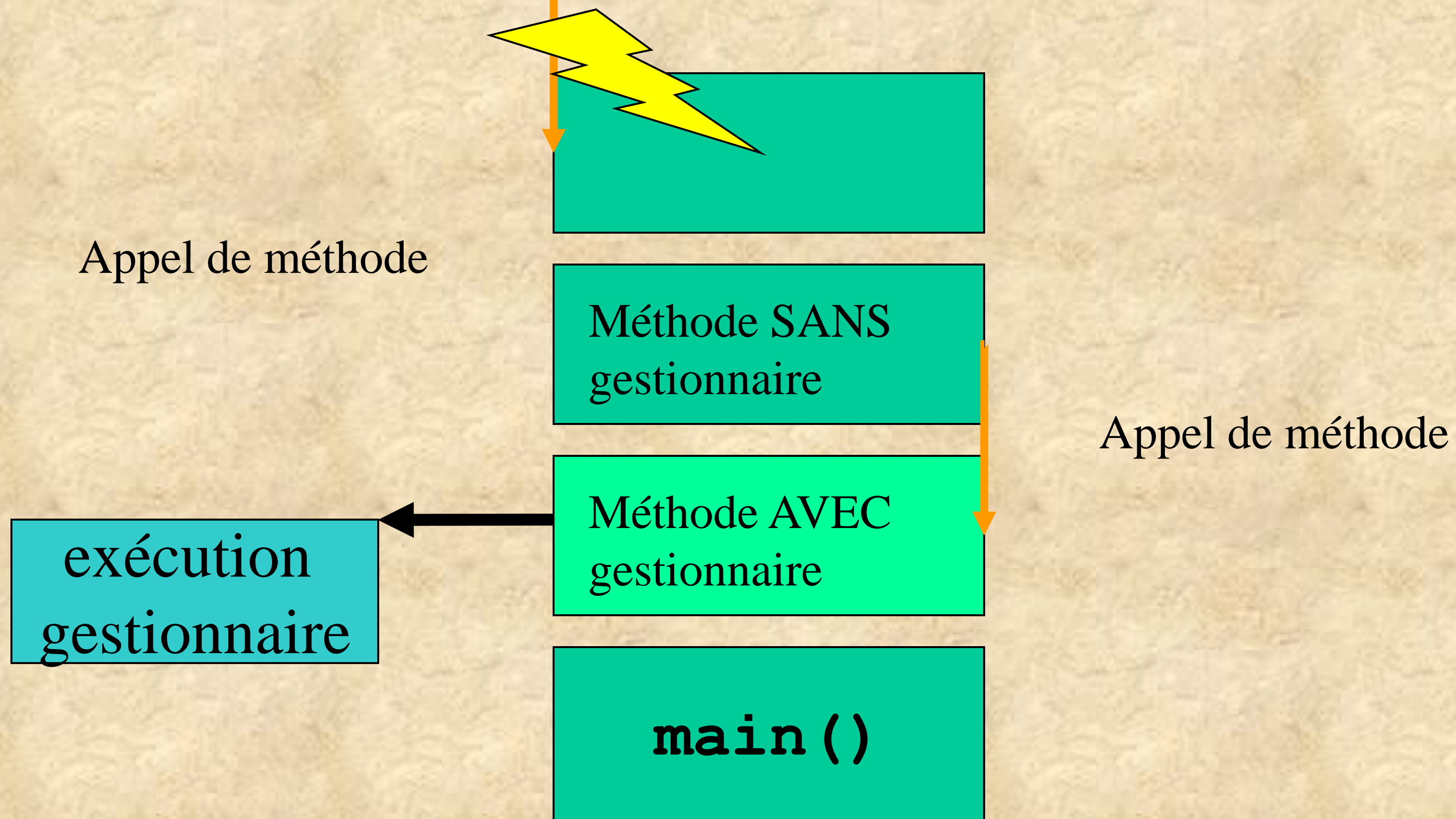


Principe des exceptions

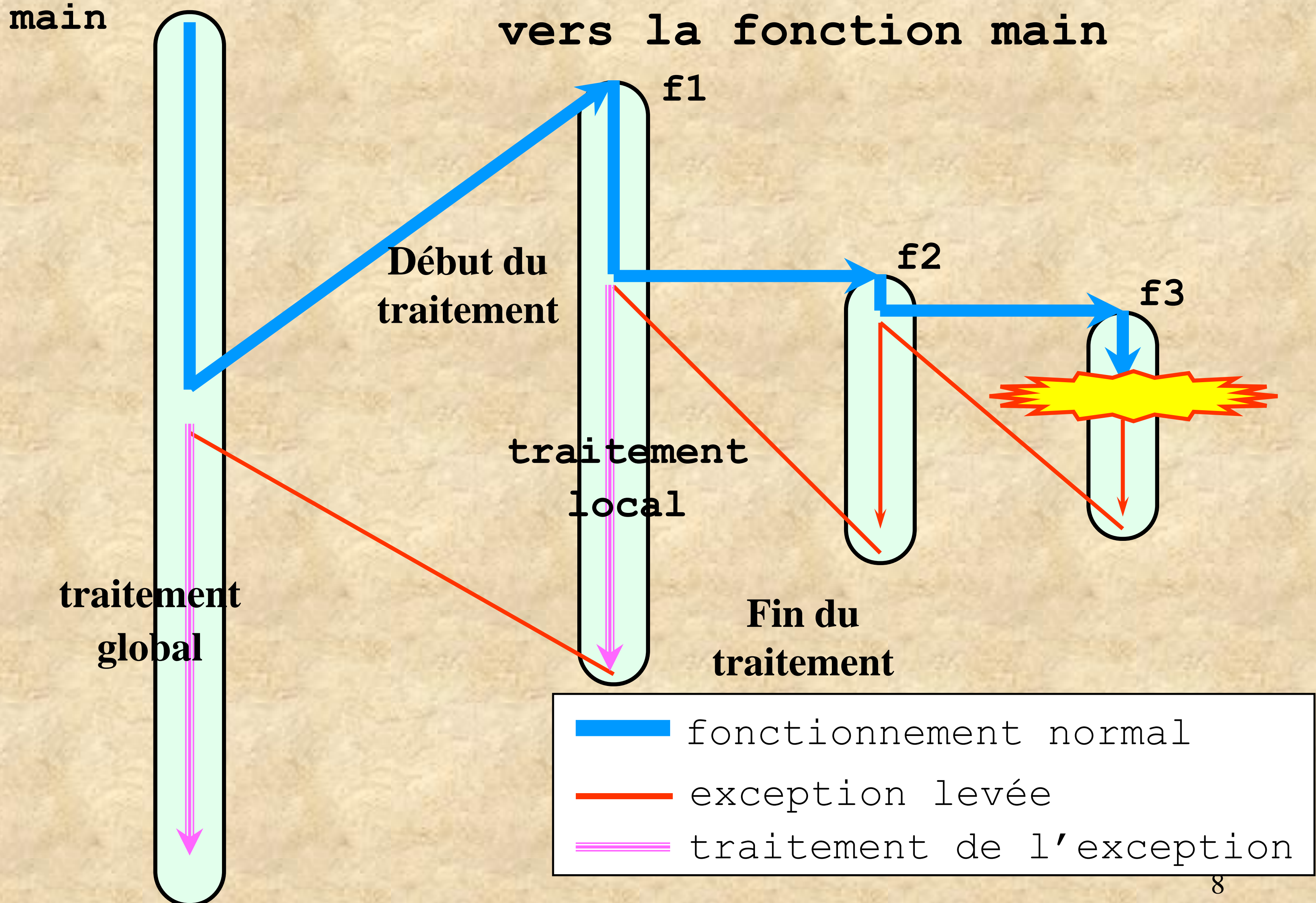


Principe des exceptions (2)

Recherche du gestionnaire



traitement d'une exception dans une fonction et propagation vers la fonction main



Obligation de capture ou de transfert

- Il existe trois catégories d'exceptions
  - Les erreurs (problèmes de la JVM)
  - Les exceptions non vérifiées (Runtime)
  - Les exceptions vérifiées (la majorité)
- Java <u>oblige</u> au traitement des exceptions vérifiées. Soit :
  - On doit trouver un gestionnaire
  - On doit transférer l'exception au-dessus

Comment on gère les exceptions

- Une exception est un objet (instance) envoyé par la machine virtuelle lorsqu'une certaine condition d'erreur apparaît.
- On met le code qui appelle la fonction où l'erreur peut se produire « sous contrôle », dans une structure

```
try {} catch {} finally {}
```

Le bloc `try`

- On enserre un code non fiable dans un bloc `try` => bloc de « mise à l'épreuve »
- Le bloc `try` sera suivi de blocs gestionnaires (clauses `catch`) locaux.
- Plusieurs instructions du bloc `try` peuvent générer une exception.

```
try{
        ...code...
}
```

Les blocs `catch`

- Les blocs **`catch`** sont des gestionnaires pour un certain type d'exception.

=> un bloc de « mise à l'épreuve » peut être associé à plusieurs blocs gestionnaires.

- Un bloc gestionnaire reçoit un paramètre.

```
catch(ExceptionType e){
    ...code...
}
```

Les blocs `catch` (2)

- Un bloc `try` a nécessairement (au moins) un bloc `catch`
- Un bloc gestionnaire est choisi si et seulement si :
  - le contrôle de la recherche de gestionnaire considère ce bloc (dans l'ordre d'apparition du source)
  - le type du paramètre reçu est compatible avec l'objet d'exception d'origine

Le bloc `finally`

- Le bloc `finally` n'est pas obligatoire
- Le bloc `finally` s'exécute QUELLE QUE SOIT L'ISSUE de la structure `try`…`catch`

La structure complète

- Try -> obligatoire si exceptions vérifiées
- Catch -> au moins un si try
- Finally -> optionnel, mais conseillé.

```
try{
   … something …
} catch(AnErrorType e){
   … do what possible to fix …
} finally{
   … do something anyway …
}
```

Scénario sans erreur

```
try{

    … something all good …

} catch(anErrorType e){

    … do what possible to fix …

} finally{
    … do something anyway …
}
```

Exemple

```
// DivParZero.java
class DivParZero {
  public static void main (String argv[] ) {
      int val=0;
      try {
            val = 1997/val;
      }
      catch (ArithmeticException e ) {
             System.out.println("Une exception arithmétique a été levée");
             System.out.println("Message : " + e.getMessage());
             System.out.println("Pile :");
            e.printStackTrace();
       }
      finally {
               System.out.println("Fin du calcul");
       }
      System.out.println("Fin du programme");
   }//fin de main
}//fin de la classe
```

A l'exécution, on obtient :

```
Une exception arithmétique a été levée
Message : / by zero
Pile :
java.lang.ArithmeticException: / by zero
at DivParZero.main(DivParZero.java:5)
Fin du calcul
Fin du programme
```

Scénario avec erreur spécifique

```
try{
   … something good …
   … something goes wrong …
   … something follows …
} catch(AnErrorType e){
   … do what possible to fix …
} finally{
   … do something anyway …
}
```

Exemple

```
class DivParZero {
  public static void main (String argv[] ) {
     int val=0;
     try {
          val = 1997/val;
     }
    catch (Exception e ) {
        System.out.println("Pile :");
    }
    catch (ArithmeticException e ) {
      System.out.println("Une exception arithmétique a été levée");
      System.out.println("Message : " + e.getMessage());
      System.out.println("Pile :");
      e.printStackTrace();
    }
  }
}
```

Un tel segment de code compile-t-il ? Précisez.

Non, car le deuxième catch capture une exception plus spécialisée que la première : le code est inatteignable

Exemple

```
class DivParZero {
   public static void main (String argv[] ) {
      int val=0;
      try {
            if (val==0) throw new ArithmeticException ("Division par zéro");

          val = 1997/val;
      }
     catch (ArithmeticException e ) {
        System.out.println("Une exception arithmétique a été levée");
        System.out.println("Message : " + e.getMessage());
        System.out.println("Pile :");
        e.printStackTrace();
      }
   }
}
```

```
Une exception arithmétique a été levée
Message : Division par zéro
Pile :
java.lang.ArithmeticException: Division par zéro
at DivParZero.main(DivParZero.java:8)
```

Scénario avec erreur non spécifique

```
try{
   … something good …
   … something goes wrong …
   … something follows …
} catch(AnErrorType e){
   … do what possible to fix …
} catch(AnMoreGeneralError e){
   … do what possible to fix that …
} finally{
   … do something anyway …
}
```

Scénario avec erreur non traitée

```
void aMethod() throws SomeException {
    try{
        … something good …
        … something goes wrong with SomeException …
        … something follows …
    } catch(AnExceptionType e){
        … do what possible to fix …
    } finally{
        … do something anyway … (?)
    }
}
```

Spécifier le transfert d'une exception

- Si on ne capture pas une exception attendue, il faut la transmettre dans la pile d'appel.
- On doit déclarer cette transmission.
- On peut déclarer plusieurs transmissions.
- On déclare le type des objets que l'on transmet

```
public void uneMethode() throws
   IOException,ArrayIndexOutOfBoundsException {
```

- On peut déclarer un type plus général que ce que l'on émettra réellement.

Lever une exception

- Si utiliser une certaine méthode présente le risque d'exceptions, c'est qu'on sait les émettre.
- On obtient des exceptions par certaines méthodes de l'API standard.
- On peut lever soi-même explicitement dans son code des exceptions :
  - On crée un objet adéquat (= **`Throwable`**)
  - On le « jette » (en l'air) => **`throw`** !!
- Ce qui peut se résumer en une ligne par :

```
throw new MyException();
```

A supposer que MyException est une Throwable !!

Lever une exception (2)

- Qu’est ce qu’une Throwable ?
  - La classe mère de tous les objets utilisables dans le mécanisme d’exception.
  - Deux sous-classes :
    - Error
    - Exception
      - RuntimeException ...
      - ...
- Une Error peut être levée, mais elle n’oblige pas à capture.
- Une RuntimeException peut être levée, mais n ’oblige pas à capture (non vérifiée)

Exceptions en cascade

- Une exception appelle un gestionnaire (handler)
- -> Un gestionnaire contient du code Java
- => Un gestionnaire peut lever une exception
- => Qui gère l'exception levée par un gestionnaire ?

```
try{
   … something …
} catch(AnExceptionType e){
   throw new AnotherException("Message", e);
}
```

Exception en cascade (2)

- Solution 1 :
  - Capturée par un gestionnaire situé après
- Solution 2 :
  - Capturée par un gestionnaire d'un bloc try…catch au dessus dans la pile d'appel

Exception en cascade (3)

- Simplement lancer un rebond d’exception n ’est pas une cascade.
- Une cascade récupère l’information de l’exception initiale en utilisant l’API du Throwable :
  - getCause()
  - initCause(Throwable)
- Une exception de cascade accumule les informations tout au long de la cascade : nous connaissons la pile de tous ces empilements :

```
StackTraceElement[] getStackTrace()
```

Créer des classes d'exception

- La plupart des cas génériques d'exception existent dans l'API.
- A un type correspond une sémantique.
- Une instance peut être « contextualisée » par un message que j'y range dedans.
- => Je peux créer des classes d 'exception pour « mes erreurs à moi ».

Créer des classes d'exception (2)

- Mes classes d'exception doivent être Throwable
- Mes classes d'exception devraient être « vérifiées »
- Mes classes d 'exception sont des filles de **`Exception`**

Créer des classes d'exception (3)

- Structure minimale d'une classe d'exception :

```
public class MyException extends Exception {
}
```

- Structure acceptable d'une classe d'exception :

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

Hiérarchies d'exception

- On utilise l'héritage pour regrouper des exceptions en une catégorie plus générale.
- Exemple :
  - CouleurException
    - RougeException extends CouleurException
    - BleueException extends CouleurException
    - JauneException extends CouleurException

Exceptions vérifiées / non vérifiées

- Les exceptions vérifiées obligent à leur traitement : leur usage améliore la qualité logicielle.
- Les exceptions non vérifiées peuvent être ignorées, mais cela laisse des cas non sécurisés. C ’est plus facile pour le programmeur (en apparence).
- Il est rare qu’un programmeur écrive des Error.

Avantage des exceptions

- Clarté du code, même lorsque les erreurs sont prises en charge
- Mécanisme de pile : économie de code pour des situations compliquées
- Catalogage des erreurs en familles par l’héritage : économie de code par traitement d’erreurs générales.

FIN DU COURS