Les Exceptions

Exceptions

Exceptions ???

- mécanisme utilisé très fréquemment dans le langage Java
- les exceptions sont rencontrées dans de nombreuses situations

certaines exécutions peuvent faire apparaître (lever) des exceptions

```
C:>java throwtest 3
 i = 3
 java.lang.ArrayIndexOutOfBoundsException: 3
        at throwtest.b(throwtest.java:92)
        at throwtest.a(throwtest.java:65)
        at throwtest.main(throwtest.java:58)
```

instructions **`try catch`** dans les programmes

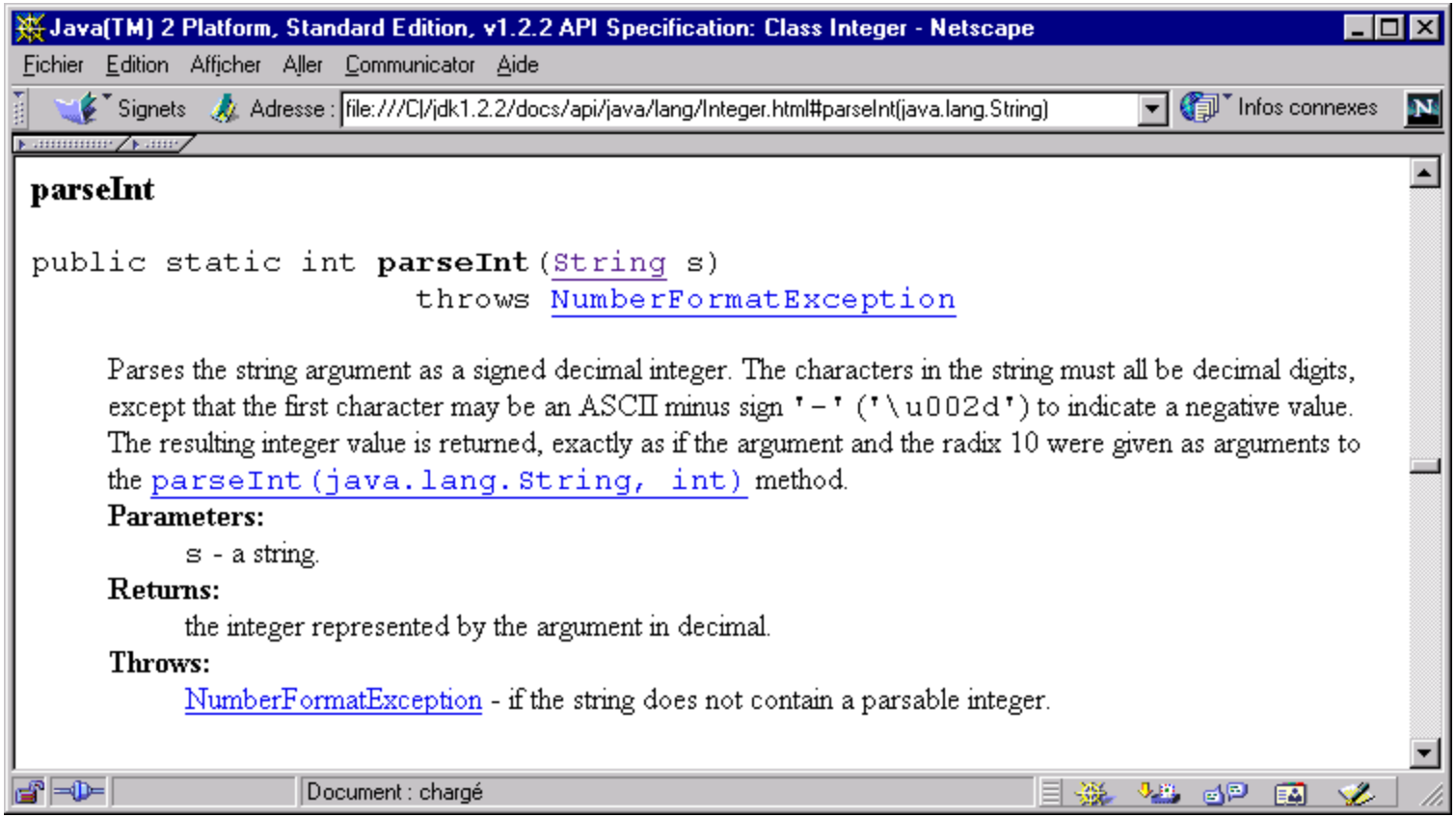
```
try {
    valSurface = Integer.parseInt(surface.getText());
}
catch (NumberFormatException execpt)
{
    surface.setText("ENTIER !!!");
    return; // on sort sans creer d'instance
}
```

Exceptions

Exceptions ???

- les exceptions sont rencontrées dans de nombreuses situations

Dans la documentation de certaines méthodes



Exceptions ???

- **Mais alors qu'est-ce qu'une exception ?**

- Un exception est un **signal**
  - *qui indique que quelque chose d'exceptionnel (par exemple une erreur) s'est produit,*
  - *qui interrompt le flot d'exécution normal du programme.*

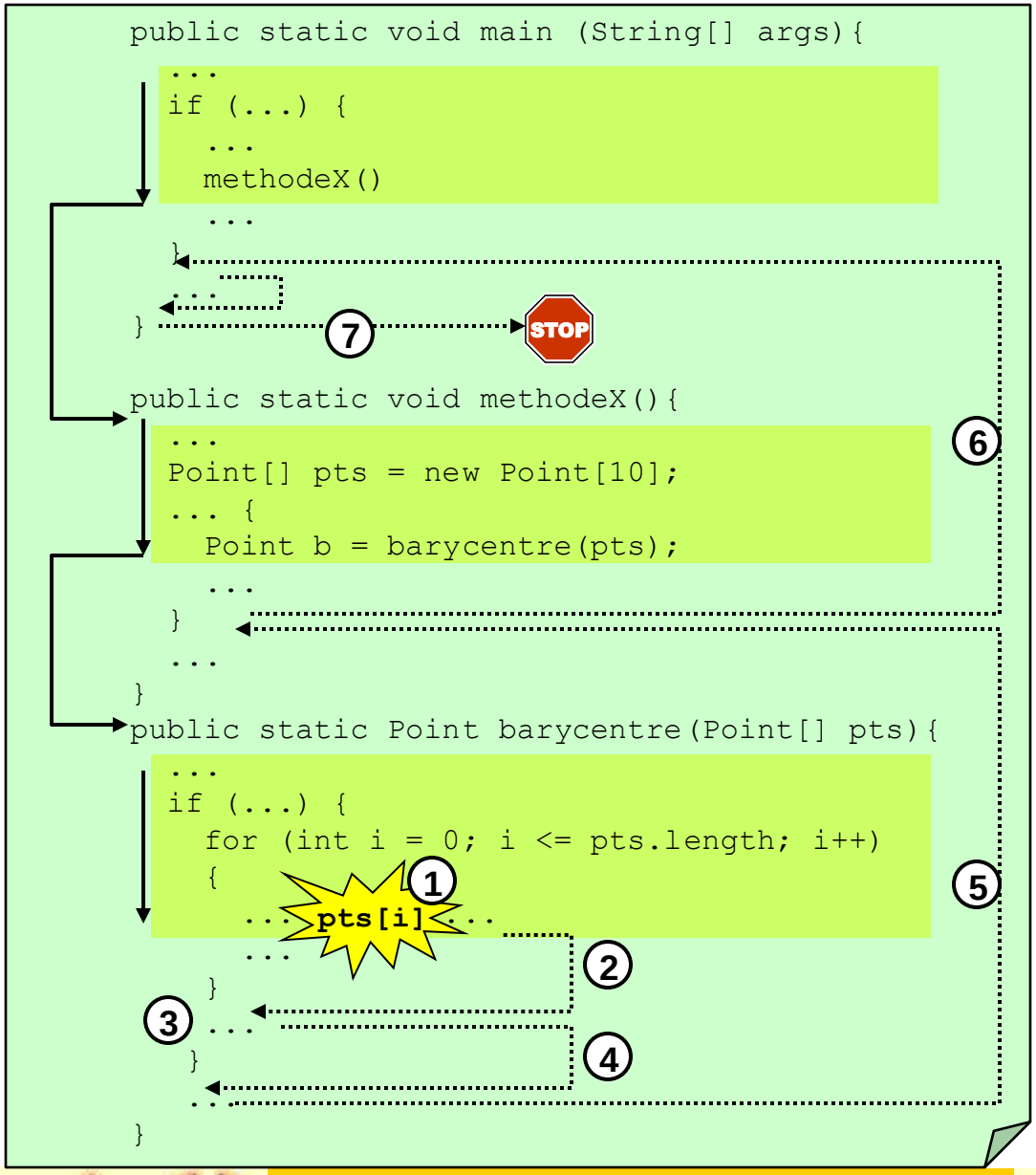
- **lancer** (throw) une exception consiste à signaler ce quelque chose,

- **attraper** (catch) une exception permet d'exécuter les actions nécessaires pour traiter cette situation.

- lorsqu'une situation exceptionnelle est rencontrée, une exception est lancée
- si cette exception n'est pas attrapée dans le bloc de code où elle a été lancée, elle est propagée au niveau du bloc englobant,
- si celui-ci ne l'attrape pas elle est transmise au bloc de niveau supérieur et ainsi de suite...
- si l'exception n'est pas attrapée dans la méthode qui la lance, elle est propagée dans la méthode qui invoque cette dernière.
- si la structure de bloc de la méthode d'invocation ne contient aucune instruction attrapant l'exception, celle-ci est à nouveau propagée vers la méthode de niveau supérieur.
- si une exception n'est jamais attrapée :
  - *propagation jusqu'à la méthode main() à partir de laquelle l'exécution du progamme a débutée,*
  - *affichage d'un message d'erreur et de la trace de la pile des appels (call stack),*
  - *arrêt de l'exécution du programme.*

Exceptions

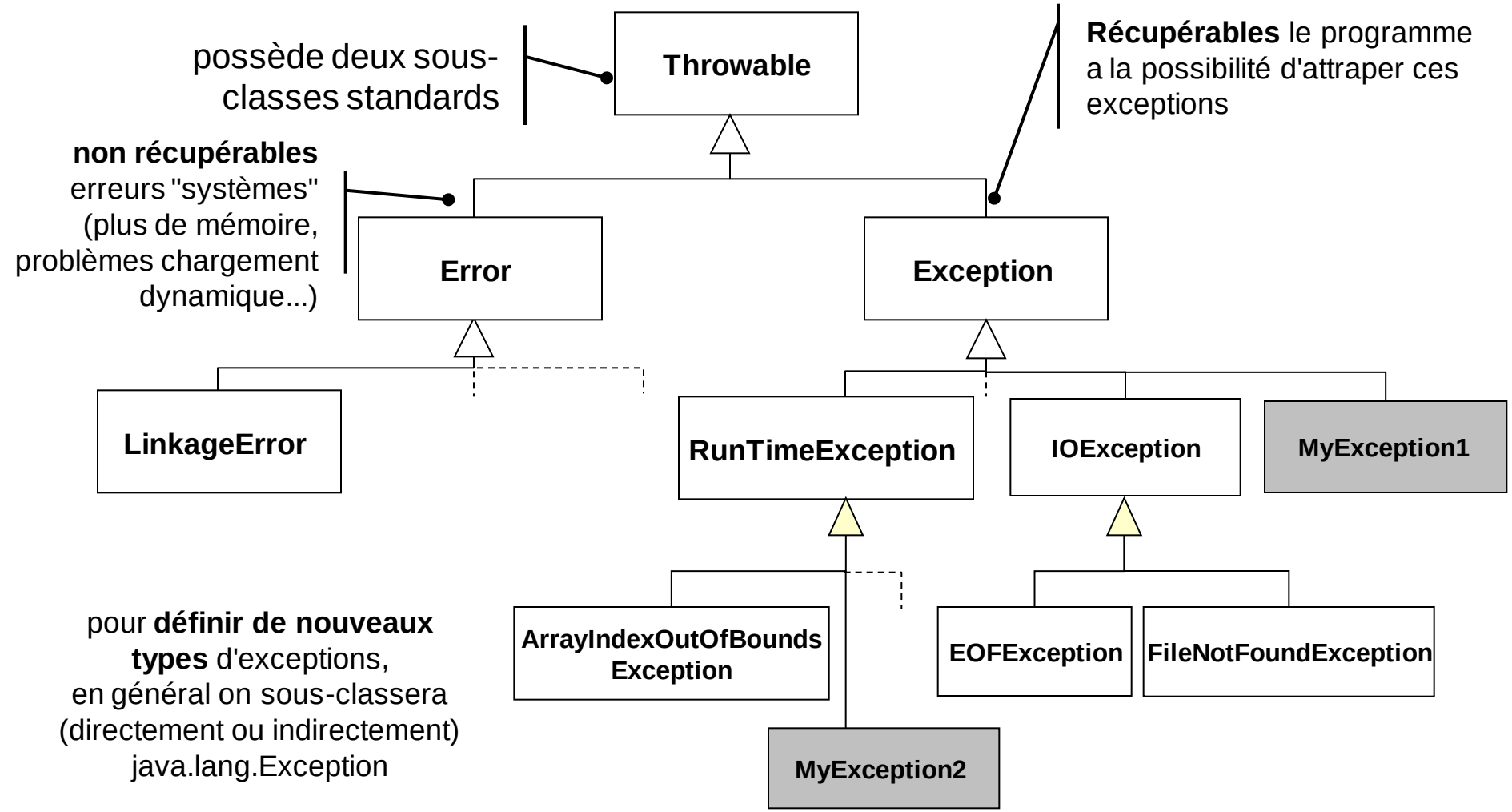
Mécanisme



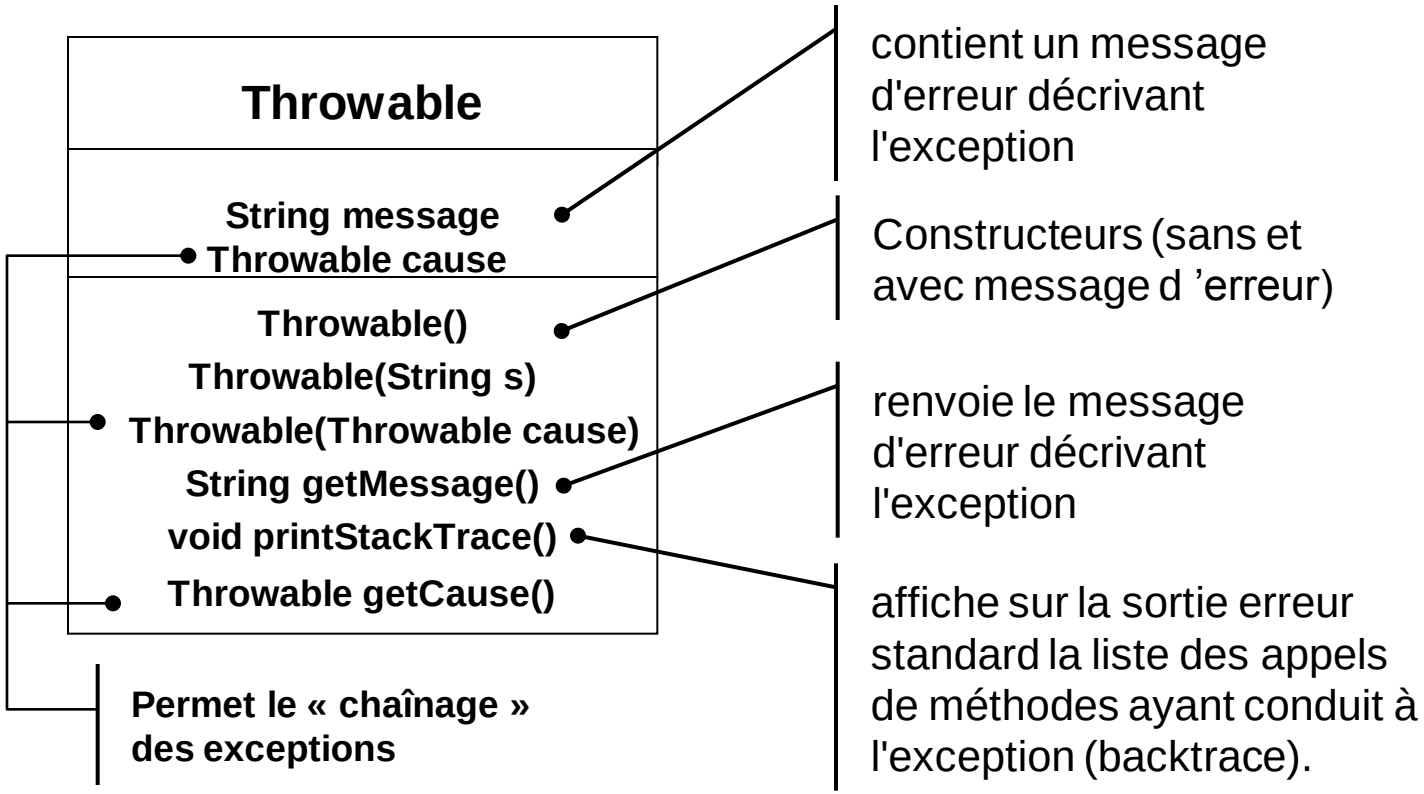
1. Dépassement d'index, arrêt de l'exécution normale et *lancement* de `ArrayOutOfBoundsException`
2. Transmission du contrôle au bloc de niveau supérieur
3. Si il y a du code pour traiter (*attraper*) l'exception reprise du flot d 'exécution normal
4. Sinon on recommence comme en 2
5. Si aucun code dans la méthode pour traiter l'exception le contrôle est transmis au niveau de la méthode appelante et on recommence comme en 2
6. Et ainsi de suite...
7. Jusqu'à ce que l'on aboutisse au bloc du programme principal, alors arrêt de l'exécution et impression de la pile des appels.

Exceptions — Objets exceptions

- en JAVA les **exceptions sont des objets**
  - *toute exception doit être une instance d'une sous-classe de la classe* ***`java.lang.Throwable`***



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



- La représentation des exceptions sous forme d'objets permet de **mieux structurer** la **description** et le **traitement** des erreurs

"Because all exceptions that are thrown within a Java program are **first-class objects**, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be instances of Throwable or any Throwable descendant. As for other Java classes, you can create subclasses of the Throwable class and subclasses of your subclasses. Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions. "

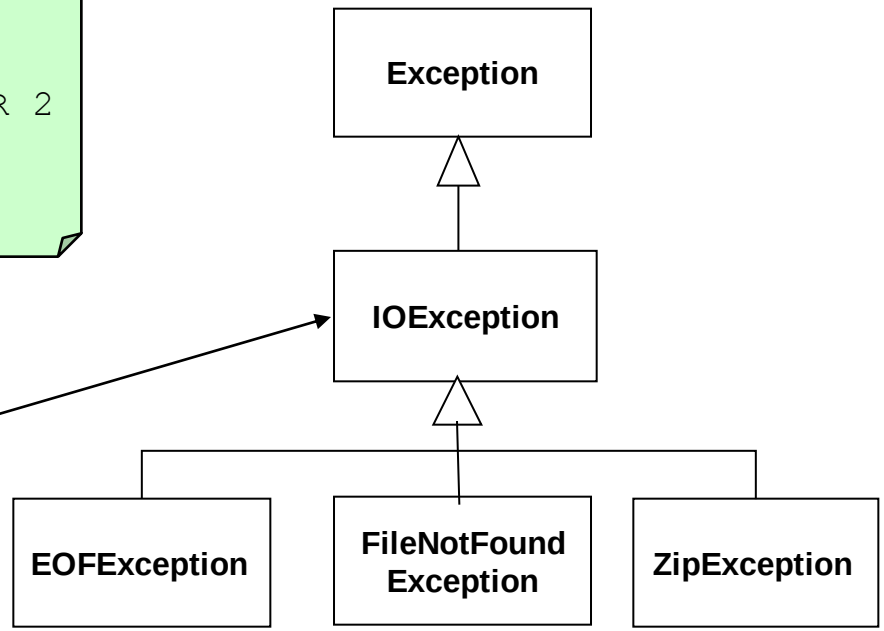
[The Java Tutorial, Campione &Walrath 97]

- **Structuration de la description** des exceptions

Dans ce bon vieux C

```
#define EOF_ERROR 1
#define FILENOTFOUND_ERROR 2
#define ZIP_ERROR 3
```

En Java

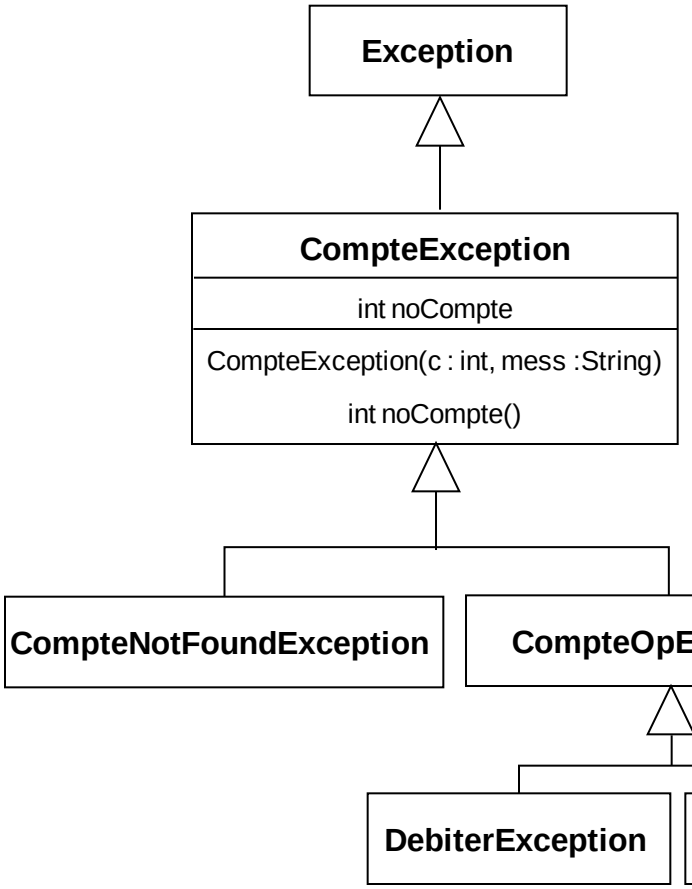


Niveau d'abstraction qui permet de désigner toutes les exceptions liées au E/S

Les exceptions étant des objets, leur structuration en une hiérarchie de classes permet de les traiter à différents niveaux d'abstraction (via polymorphisme)

Exceptions — Objets exception

- On peut être amené à définir une hiérarchie de classes (sans nécessairement définir de nouveaux attributs et nouvelles méthodes) uniquement dans un souci de **structuration** et de **typage** des exceptions



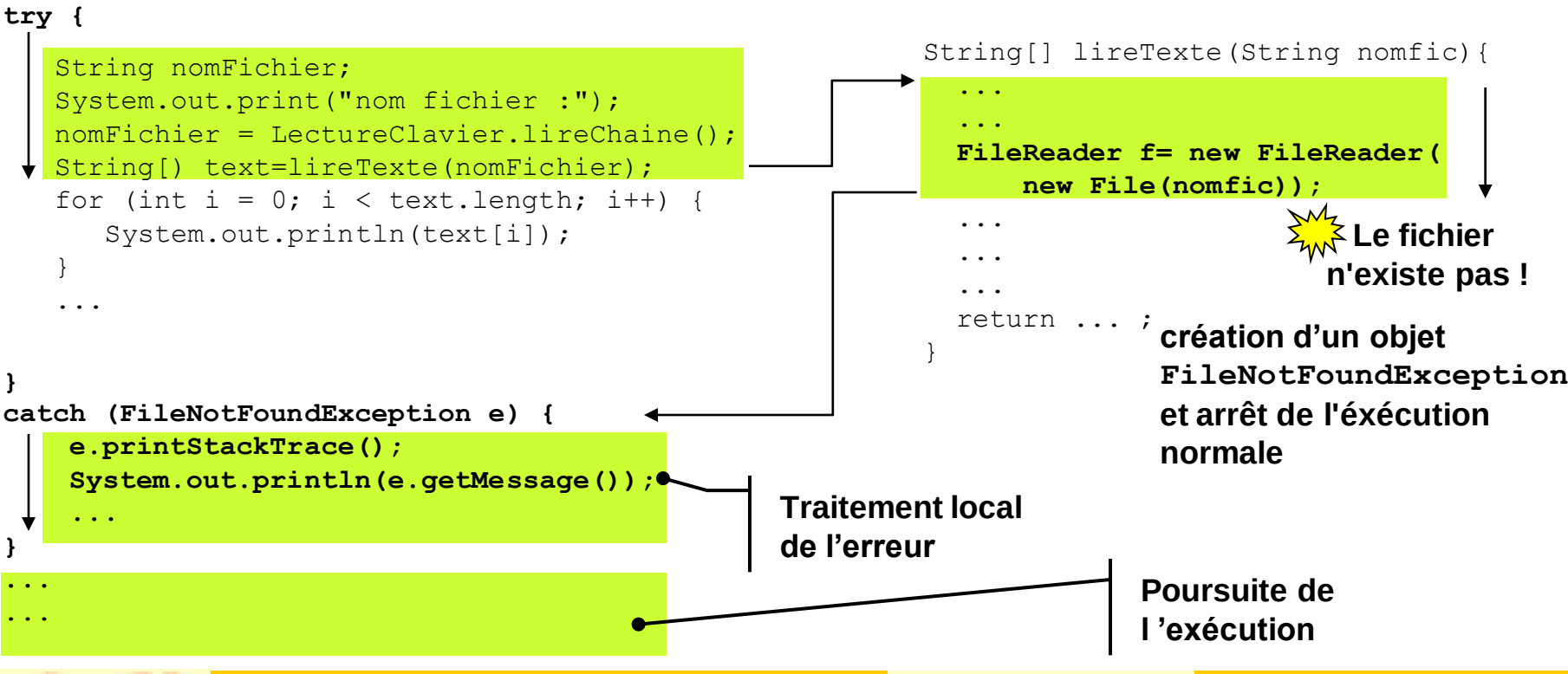

```
class CompteException extends Exception {
   private int noCompte;

   public CompteException(int c, String mess) {
      super(mess);
      noCompte = c;
   }

   public int noCompte() {
     return noCompte;
   }
}
```

```
class CompteNotFoundException extends CompteException {
public CompteNotFoundException(int c, String mess) {
      super(c,mess);
}
}
```

```
class CompteOpException extends CompteException {
   public CompteOpException(int c, String mess) {
      super(c,mess);
}
}
```

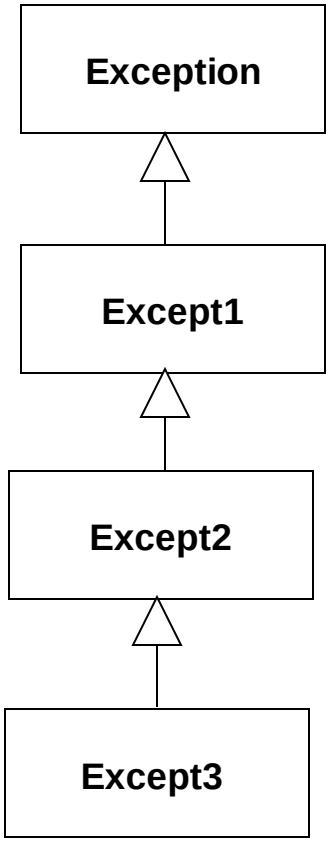
Exceptions — Traitement

- **`try{ … }`** délimite un ensemble d'instructions susceptibles de déclencher une(des) exception(s) pour la(les)quelles une gestion est mise en œuvre
- cette gestion est réalisée par des blocs **`catch(TypeDexception e){ … }`** qui suivent le bloc **`try`**
  - *permettent d'intercepter ("attraper") les exceptions dont le type est spécifié et d'exécuter alors du code spécifique.*



- un bloc **`try`** est suivi par une ou plusieurs clauses **`catch`** qui permettent d'intercepter ("attraper") les exceptions dont le type est spécifié (dans la clause **`catch`**) et d'exécuter alors du code spécifique.
- un **seul bloc `catch` peut être exécuté** : le premier suceptible "d'attraper" l'exception.
  - *chaque clause catch doit être déclarée avec un argument de type **`Throwable`** ou une sous-classe de **`Throwable`***
  - *quand une exception est levée dans le bloc **`try`**, la première clause **`catch`** dont le type de l'argument correspond à celui de l'exception levée est invoquée*
    - *clause **`catch`** dont l'argument est de même classe que l'exception levée,*
    - *clause **`catch`** dont l'argument est une super-classe de la classe de l'exception levée.*
- l'**ordre** des blocs **`catch`** est donc très important.

Exemple : d'après "*Programmation Java*", J.F. Macary, N. Cédric, Ed. Eyrolles 1996

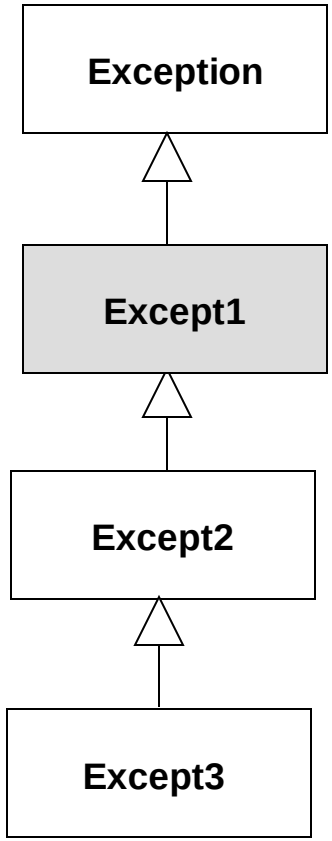


Exception de type :**Except1** ou **Except2** ou **Except3**

quel bloc `catch` va la traiter ?

?

```
...
try {

    // séquence de code pouvant lever
    //des exceptions de type Except1,
    // Except2 ou Except3
    ...
}
catch (Except1 e1)
{
    // e1 est l'exception qui a été levée
    // dans le bloc try
    System.out.println(e1.getMessage());
    e1.printStackTrace();
    // ...
}
catch (Except2 e2)
{
  // ...
}
catch (Except3 e3)
{
  // ...
}
```

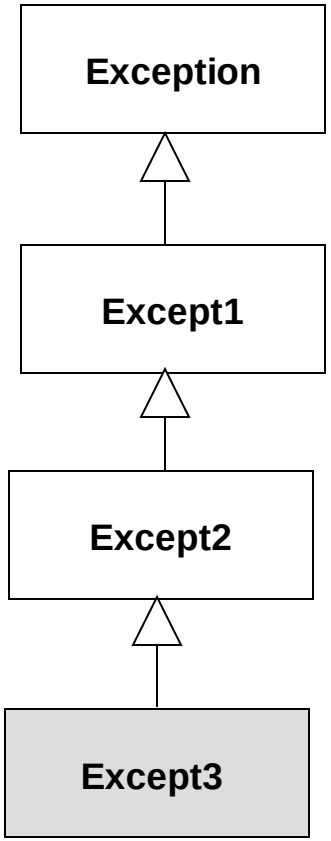


Exception de type :
Except1

type Except2 ? catch (Except2 e2)

type Except1 ? catch (Except1 e1)

```
...
try {

    // séquence de code pouvant lever
    //des exceptions de type Except1,
    // Except2 ou Except3
    ...
}
catch (Except2 e2)
{
    // e2 est l'exception qui a été levée
    // dans le bloc try
    System.out.println(e2.getMessage());
    e2.printStackTrace();
    // ...
}
catch (Except1 e1)
{
  // ...
}
catch (Except3 e3)
{
  // ...
}
```



Exception de type :
Except3

type Except2 ? `catch (Except2 e2)`

Une Except3
est une Except2

le bloc Except3
ne peut **jamais**
être atteint

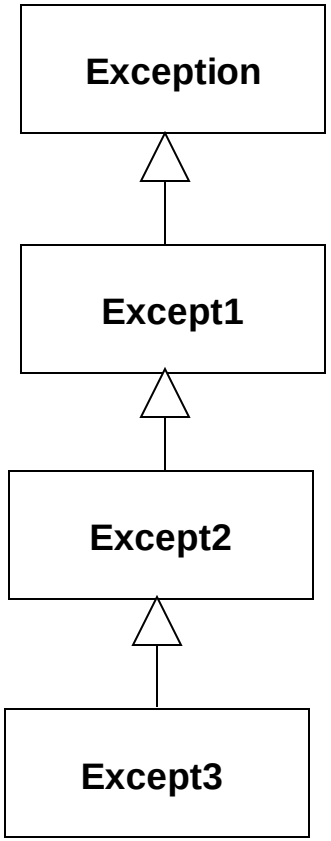
d'ailleurs le
compilateur le signale

```
...
try {

    // séquence de code pouvant lever
    //des exceptions de type Except1,
    // Except3 ou Except2
    ...
}
catch (Except2 e2)
{
    // e2 est l'exception qui a été levée
    // dans le bloc try
    System.out.println(e2.getMessage());
    e2.printStackTrace();
    // ...
}
catch (Except1 e1)
{
  // ...
}
catch (Except3 e3)
{
  // ...
}
```

```
c:>\Philippe\Java\Tests\>javac TestException.java
 TestException.java:24: cath not reached
 catch (Except3 e3) {
 ^
 1 error
```

Exceptions

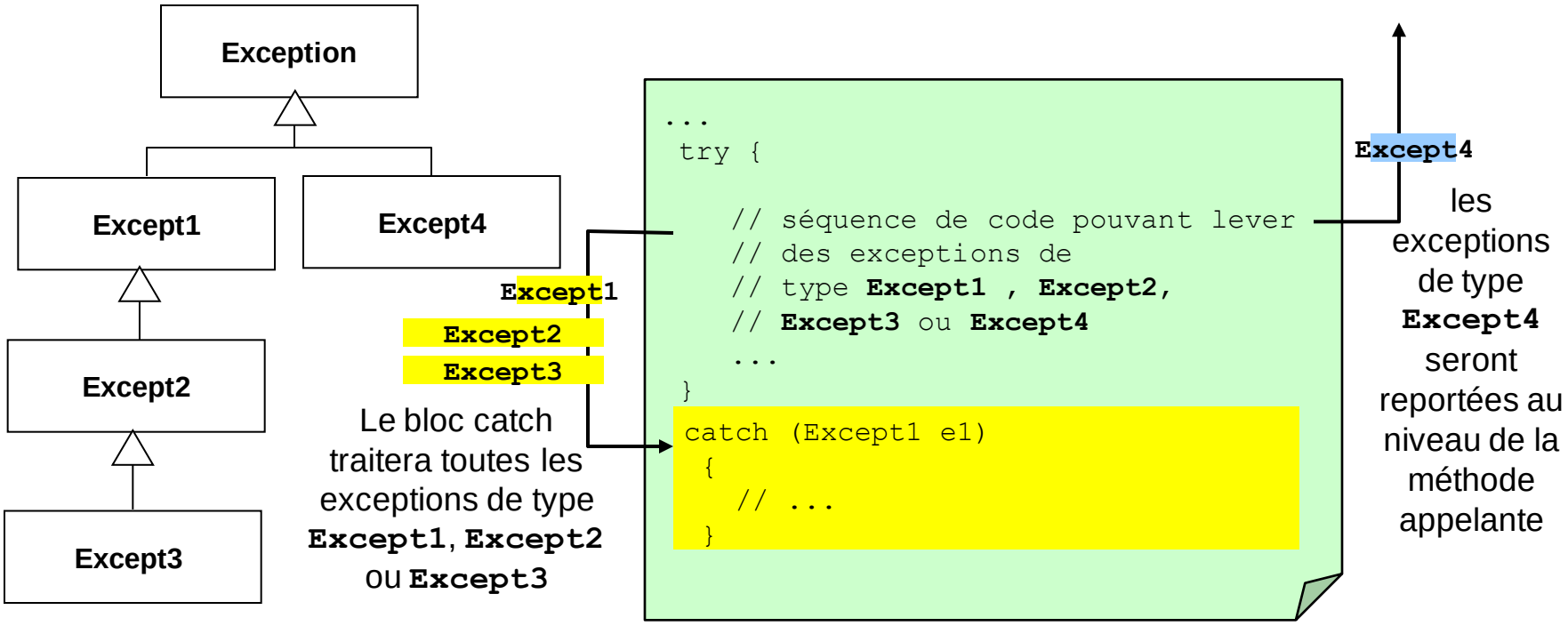
Traitement



L 'ordre des blocs `catch` doit respecter l'ordre inverse de l'ordre d'héritage entre les classes d'exception

```
...
 try {

    // séquence de code pouvant lever
    //des exceptions de type Except1,
    // Except2 ou Except3
    ...
 }
  catch (Except3 e3)
  {
    // ...
  }
  catch (Except2 e2)
  {
     //e2 est l'exception qui a été levée
     // dans le bloc try
     System.out.println(e2.getMessage());
     e2.printStackTrace();
     // ...
  }
  catch (Except1 e1)
  {
    // ...
  }
```

- pas nécessaire d'avoir clause **`catch`** pour chaque type possible d'exception pouvant être levée.
  - *si aucun bloc **`catch`** ne permet d'attraper l'exception, celle-ci est propagée vers la méthode appelante qui a alors la charge de la traiter ou non*
    - *si une exception n'est attrapée par aucune des méthodes présentes dans la pile des appels alors un message d'erreur ainsi que la trace de la pile d'appels sont affichés et l'exécution du programme est interrompue.*



- les clauses **`catch`** sont suivies de manière optionnelle par un bloc **`finally`** qui contient du code qui sera exécuté quelle que soit la manière dont le bloc **`try`** a été quitté

- le bloc **`finally`** permet de spécifier du **code dont l'exécution est garantie** quoi qu'il arrive :
  - *le bloc **`try`** s'exécute normalement sans qu'aucune exception ne soit levée*
    - *la fin du bloc **`try`** a été atteinte,*
    - *contrôle quitte le bloc **`try`** suite à une instruction return, continue, break*

    *(d'ou parfois l'utilisation d'un bloc **`try`** avec un bloc **`finally`** sans clauses **`catch`**).*
  - *le bloc **`try`** lève une exception attrapée par l'un des blocs **`catch`**.*
  - *le bloc **`try`** lève une exception qui n'est attrapée par aucun des blocs **`catch`** qui le suivent.*

- interêt double :
  - *permet de rassembler dans un seul bloc un ensemble d'instructions qui autrement auraient du être dupliquées*
  - *permet d'effectuer des traitements après le bloc* ***try****, même si une exception a été levée et non attrapée par les blocs catch*

```
...
 try {
    // ouvrir un fichier
    // effectuer des traitements
    // susceptibles
    // de lever une exception
    // fermer le fichier
 }
 catch (CertaineException e)
 {
   // traiter l'exception
   // fermer le fichier
 }
 catch (AutreTypeException e)
 {
   // traiter l'exception
   // fermer le fichier
 }
```

```
...
 try {
    // ouvrir un fichier
    // effectuer des traitements
    //susceptibles
    // de lever une exception
}
 catch (CertaineException e)
 {
   // traiter l'exception
 }
 catch (AutreTypeException e)
 {
   // traiter l'exception
 }
 finally {
   //fermer le fichier
 }
```

Le bloc finally est toujours exécuté

- 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.*

- Exemple : violation d'une précondition

```
class OperationBancaireException extends Exception
{
 /**
  * compte pour lequel l'opération a échouée
  */
  private Compte c;

  public OperationBancaireException(Compte c,String s)
  {
     super(s);
     this.c = c;
  }
}
```

2. Définition d'une nouvelle classe d 'exception

4. L 'exception doit être signalée dans la signature de la méthode

```
public class Compte {
   protected double solde = 0;
   ...

  /**
  * Dépôt d'argent sur le compte
  * @param s la somme à déposer,  s doit être >= 0
  * @throws OperationBancaireException si s <=0
  */
  public void créditer(double s)  throws OperationBancaireException {

    if (s < 0)
          throw new OperationBancaireException(this,
                    "dépôt incorrect");

     solde += s;
  }
  ...
}
```

1. L'opération ne peut être effectuée que si le dépôt est >= 0

3. Si la précondition n 'est pas vérifiée création et lancement d 'une exception

Déclaration

- Toute méthode susceptible de lever une exception "normale" doit
  - *soit **l'attraper**,*
  - *soit **la déclarer explicitement**, c'est à dire comporter dans sa signature l'indication que l'exception peut être provoquée : clause* **`throws`**
- Les exceptions déclarées dans la clause **`throws`** d 'une méthode sont :
  - *les exception levées dans la méthode et non attrapées par celle-ci,*
  - *les exceptions levées dans des méthodes appelées par la méthode et non attrapées par celle-ci.*

Exceptions

Déclaration

```
class MonException extends Exception
 {
     public MonException()
     {
       super();
     }
     public MonException(String s)
     {
       super(s);
     }
 }
```

```
public class TestExcep {

    public void  method1() throws MonException
    {
      throw (new MonException());
    }

   public void  method2()
    {
      method1();
    }

 }
```

si on oublie une de traiter une exception le compilateur le signale :-)

```
BASH.EXE-2.02$ javac TestExcep.java
      TestExcep.java:24: Exception MonException must be caught,
      or it must be declared in the throws clause of this method.
            method1();
                  ^
1 error
BASH.EXE-2.02$
```

```
BASH.EXE-2.02$ javac TestExcep.java
      TestExcep.java:24: Exception MonException must be caught,
      or it must be declared in the throws clause of this method.
            method1();
                  ^
1 error
BASH.EXE-2.02$
```

```
public class TestExcep {

    public void  method1() throws MonException
    {
      throw (new MonException());
    }

    public void  method2()
    {
      method1();
    }

}
```

Pour avoir une compilation correcte il faut modifier méthode 2

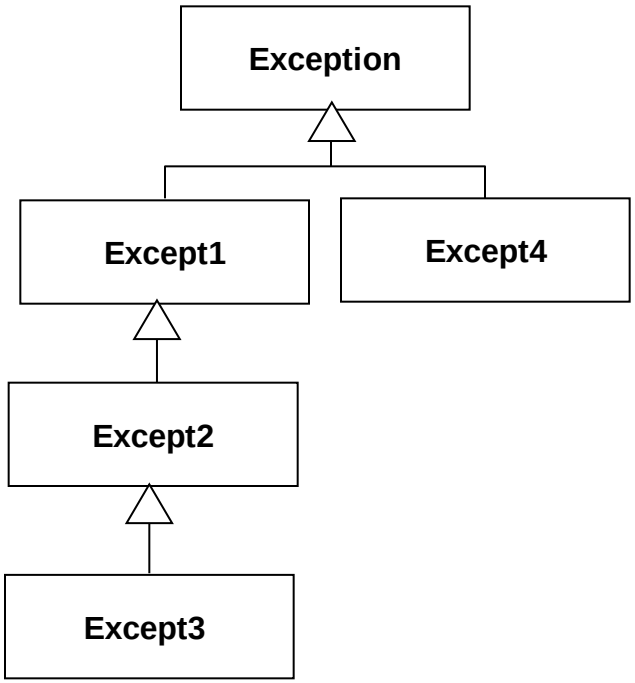
soit en en déclarant explicitement que **method2** est susceptible de lancer une exception de type **MonException**

```
public void  method2() throws MonException
{
  method1();
}
```

soit en "attrapant" l'exception

```
public void  method2()
    {
      try {
        method1();
      }
      catch (MonException e)
      {
          e.printStackTrace();
      }
    }
```

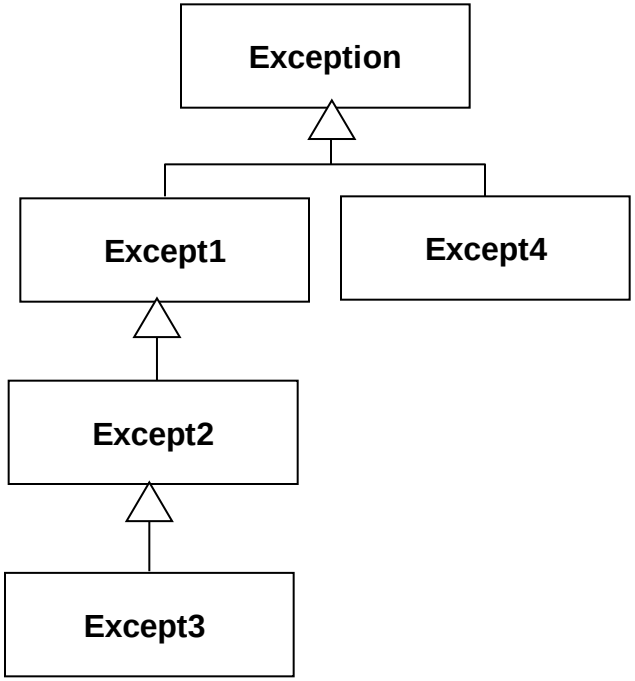
- **plusieurs classes** d 'exceptions peuvent être indiquées dans la clause **`throws`** d'une déclaration de méthode




```
void methodeQuelconque()  throws Except1,Except4  {
   try {

      // séquence de code pouvant
      // lever des exceptions
      // de type Except1, Except2,
      // Except3 ou Except4
      ...
   }
   catch (Except2 m)
   {
     // intercepte les exceptions
     // de type Except2 et Except3
     ...
   }
   finally
   {
     // ...
   }
 }
```

Les exceptions de type *Except1* et *Except4* ne sont pas traitées dans le corps de la méthode, elles doivent être déclarées dans la clause *throws*

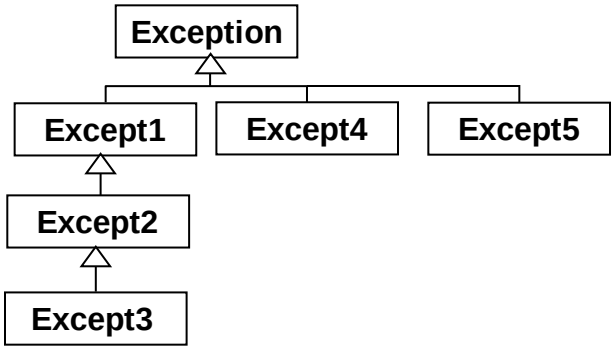
- la classe utilisée pour les exceptions dans la clause **`throws`** peut être une **superclasse** de la classe de l'exception effectivement lancée




```
void methodeQuelconque()
      throws Except1, Except4                    {

       // séquence de code pouvant
       // lever des exceptions
       // de type Except1,
       // Except2, Except3 ou Except4
       ...


 }
```

Les exceptions de type *Except1*, *Except2*, *Except3* et *Except4* ne sont pas traitées dans le corps de la méthode, elles doivent être déclarées dans la clause *throws*

Exceptions — Redéfinition de méthodes



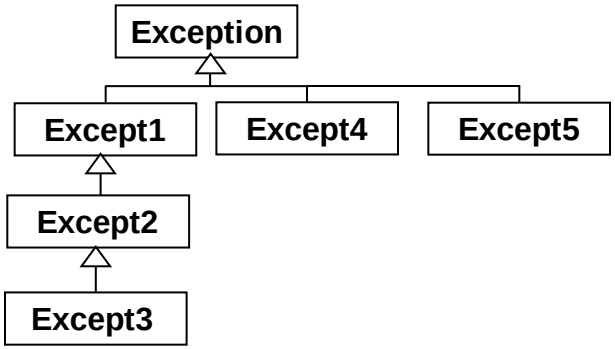

```
public Class A {

      void methodeX() throws Except1, Except4 {
         // séquence de code pouvant
         // lever des exceptions
         // de type Except1,
         // ou Except4
         ...
      }

      ...
 }
```

Lorsque l'on redéfinit la méthode peut-on lancer de nouveaux types d'exception ?

- La méthode redéfinie **doit respecter le contrat défini par la méthode originale**. Elle ne peut rajouter de nouveaux types d'exception

```
methodeX(int) in ClasseB cannot override methodeX(int) in ClassA; overridden method does not throw Except5

public Class B extends A {
      void methodeX() throws Except1, Except4 , Except5 {
         super.methodeX();
         ...
         // séquence de code pouvant lever une
         // exception de type Except5
         ...

      }
      ...
 }
```

Exceptions — Redéfinition de méthodes




```
public Class A {

      void methodeX() throws Except1, Except4 {
         // séquence de code pouvant
         // lever des exceptions
         // de type Except1,
         // ou Except4
         ...
      }

      ...
 }
```

La clause throws eut être "spécialisée"

(typage covariant de la clause throws)

Mais alors, la méthode redéfinie doit-elle avoir nécessairement la même clause *throws* ?

Pas nécessairement !

```
public Class B extends A {
      void methodeX() throws  Except3 , Except4    {
         ...
         ...
         // séquence de code pouvant lever une
         // exception de type Except3 ou Except4
         ...

      }
      ...
}
```

- La méthode redéfinie peut lancer des exceptions spécialisant les exceptions définies dans la clause `throws` de la méthode originale

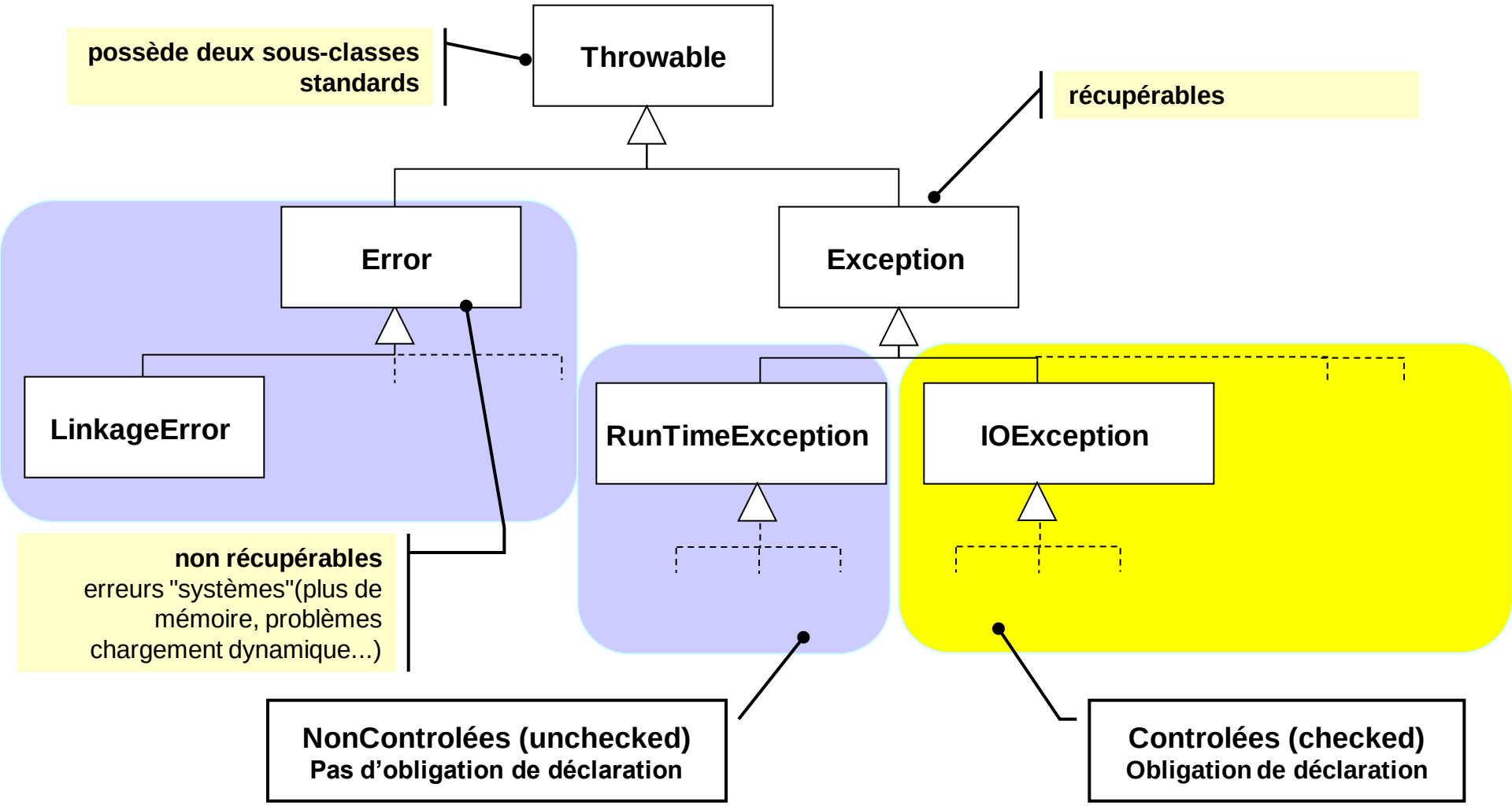
```
public Class C extends A {

      void methodeX() {

         // séquence de code ne levant
         // pas d'exception
         ...
      }

      ...
 }
```

- **l'obligation de déclaration des exceptions** présente un double intérêt ("*Programmation Java*", J.F. Macary, N. Cédric, Ed. Eyrolles 1996)
  - ***celui qui écrit*** *une méthode doit être conscient de toutes les exceptions levées par les méthodes qu'il appelle. Pour ces exceptions il doit choisir entre les traiter ou les déclarer. Il ne peut les ignorer.*
  - ***celui qui utilise*** *la méthode apprend grâce aux clauses throws quelles sont les exceptions susceptibles d'être levées par cette méthode et les méthodes appelées.*

- pour simplifier écriture des programmes (et permettre une extensibilité future) les exceptions "standards" non pas besoin d'être déclarées
  - *exceptions définies comme sous classes de* ***`Error`***
  - *exceptions définies comme sous classes de* ***`RuntimeException`*** *(exemple* ***`ArrayOutOfBoundException`****,* ***`NullPointerException`****...)*

Exceptions

Objets exceptions



```
try{

 ...

} catch (SomeException e) {
}
```

Code très, très suspect…

Bloc catch vide, détruit la finalité des exceptions qui est de vous obliger à traiter les conditions exceptionnelles qu'elles sont supposer représenter.

```
catch (SomeException e) {
   System.out.println(…);
   e.printStackTrace();
   System.exit(0);
}
```

```
catch (SomeException e) {
   // commentaire
   // justifiant le fait de
   // ne rien faire
}
```

Séparation du code de gestion des erreurs du code « normal »

- Exemple : écriture d'une méthode qui ouvre et charge en mémoire un fichier

d'après "The Java Tutorial" de Mary Campione et Kathy Walrath, ed. Adisson-Wesley

```
void readFile() {
     open the file;
     determine its size;
     allocate that much memory;
     read the file into memory;
     close the file;
 }
```

MAIS que se passe t'il si

- le fichier ne peut être ouvert ?
- si sa taille ne peut être déterminée ?
- si il n'y a pas assez de mémoire disponible ?
- si il se produit une erreur de lecture ?
- si le fichier ne peut être fermé ?

Exceptions

Intérêt

Séparation du code de gestion des erreurs du code « normal »

```
errorCodeType readFile {
     initialize errorCode = 0;
     open the file;
     if (theFileIsOpen) {
         determine the length of the file;
         if (gotTheFileLength) {
           allocate that much memory;
             if (gotEnoughMemory) {
                 read the file into memory;
                 if (readFailed) {
                     errorCode = -1;
                 }
             } else {
                 errorCode = -2;
             }
         } else {
             errorCode = -3;
         }
         close the file;
         if (theFileDidntClose && errorCode == 0) {
             errorCode = -4;
         } else {
             errorCode = errorCode and -4;
         }
     } else {
         errorCode = -5;
     }
     return errorCode;
 }
```

- ajouter dans le code de la méthode des instructions pour détecter, rapporter et gérer les erreurs, utilisation de code d ’erreur en retour des fonctions
- augmentation conséquente de la taille du code (de 7 lignes de codes on passe à 29 lignes, augmentation de plus de 400% !!)
- grande perte de lisibilité (le code devient un plat de spaghetti)
- que faire pour les fonctions qui doivent renvoyer un résultat ?

➔ **gestion des erreurs souvent négligée par les programmeurs**

Intérêt

Séparation du code de gestion des erreurs du code « normal »

```
readFile {
     try {
         open the file;
         determine its size;
         allocate that much memory;
         read the file into memory;
         close the file;
     } catch (fileOpenFailed) {
         doSomething;
     } catch (sizeDeterminationFailed) {
         doSomething;
     } catch (memoryAllocationFailed) {
         doSomething;
     } catch (readFailed) {
         doSomething;
     } catch (fileCloseFailed) {
         doSomething;
     }
 }
```

- Avantages du mécanisme d ’exceptions
  - *concision*
  - *lisibilité*

Exceptions

Interêt

- pouvoir propager les erreurs en remontant la pile des appels de méthodes
  - *possibilité de mettre en oeuvre une procédure de traitement de l'erreur à un niveau plus élevé que à l'endroit où elle s'est produite*

Approche « classique »

```
method1 {
     errorCodeType error;
     error = call method2;
     if (error)
         doErrorProcessing;
     else
         proceed;
 }
 errorCodeType method2 {
     errorCodeType error;
     error = call method3;
     if (error)
         return error;
     else
         proceed;
 }
 errorCodeType method3 {
     errorCodeType error;
     error = call readFile;
     if (error)
         return error;
     else
         proceed;
 }
```

on est contraint de forcer **method2** et **method3** à propager les codes d'erreur retournés par **readFile**

```
method1 {
     call method2;
 }
 method2 {
     call method3;
 }
 method3 {
     call readFile;
 }
```

seule **method1** intéressée par les erreurs pouvant intervenir dans **readFile**

Avec les exceptions

```
method1 {
     try {
         call method2;
     } catch (exception) {
         doErrorProcessing;
     }
 }
 method2 throws exception {
     call method3;
 }
 method3 throws exception {
     call readFile;
 }
```

propagation automatique dans la pile des appels, seules les méthodes qui se soucient des erreurs ont à les détecter

- Java 7 ajoute quelques nouveautés aux langage Java pour faciliter la gestion des exceptions
  - *récupération de plusieurs types d'exception dans un même bloc catch*
    - *évite d'avoir de répéter des blocs catchs identiques pour des exceptions de types différents*
  - *instruction "try-with-resources"*
    - *évite d'avoir un bloc finally pour la fermeture des ressources*
  - *gestion des types plus fine lors du "relancement" des exceptions*

Exceptions
Catching Multiple Exception Types

- a single catch block can handle more than one type of exception.
  - ***reduce code duplication and lessen the temptation to catch an overly broad exception.***

```
catch (IOException ex) {
        logger.log(ex);
           throw ex;
catch (SQLException ex) {
        logger.log(ex);
           throw ex;
            }
```

Code duplication


```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Bytecode generated will be smaller (and thus superior) than compiling many catch blocks that handle only one exception type each. A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler.

Exceptions

The try-with-resources Statement

- a try statement that declares one or more ***resources***, i.e. an object that must be closed after the program is finished with it.
  - ***ensures that each resource is closed at the end of the statement.***

```
static void copy(String src, String dest) throws IOException {
  InputStream in = new FileInputStream(src);
  try {
    OutputStream out = new FileOutputStream(dest);
    try {
      byte[] buf = new byte[8 * 1024];
      int n;
      while ((n = in.read(buf)) >= 0)
        out.write(buf, 0, n);
    } finally {
      out.close();
    }
  } finally {
    in.close();
  }
}
```

Resources must implement the new `java.lang.AutoCloseable` *interface.*

`InputStream`, `OutputStream`, `Reader`, `Writer` (`java.io`), `Connection`, `Statement`, *and* `ResultSet` (`java.sql`) *have been retrofitted to implement it*


```
static void copy(String src, String dest) throws IOException {
  try (InputStream in = new FileInputStream(src);
       OutputStream out = new FileOutputStream(dest)) {
    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
  }
}
```

close methods of resources are called in the opposite order of their creation.

The try-with-resources Statement


```
public static void viewTable(Connection con) throws SQLException {

    String query =
          "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {

      ResultSet rs = stmt.executeQuery(query);

      while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price +
                           ", " + sales + ", " + total);
      }

    } catch (SQLException e) {
      JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.

In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed..

Rethrowing Exceptions with More Inclusive Type Checking

- Java SE 7 compiler performs more precise analysis of rethrown exceptions than earlier releases of Java SE.
  - ***enables you to specify more specific exception types in the throws clause of a method declaration.***

```
class FirstException extends Exception { }
class SecondException extends Exception { }

...
 public void rethrowException(String exceptionName) throws Exception {
    try {
      if (exceptionName.equals("First")) {
        throw new FirstException();
      } else {
        throw new SecondException();
      }
    } catch (Exception e) {
      throw e;
    }
}
```


```
public void rethrowException(String exceptionName)
 throws FirstException, SecondException {
   try {
     // ...
   }
   catch (Exception e) {
     throw e;
   }
 }
```

Even though the exception parameter of the catch clause, e, is type Exception, the compiler can determine that it is an instance of either FirstException or SecondException

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. 

***Exceptions in Java* : Bill Venners**
javaworld juillet 1998
`http://www.javawolrd.com/javaworld/jw-07-1998/jw-07-exceptions.html`