Les Exceptions

Préparé par : Larbi Hassouni

Introduction aux exceptions

- Toutes les applications peuvent rencontrer des erreurs lorsqu'elles s'exécutent. Par exemple:
  - Un utilisateur peut fournir une donné inappropriée (Une chaine au lieu d'un nombre);
  - Un fichier dont a besoin l'application peut être déplacé ou supprimé.

- Ce type d'erreurs peut causer à une application "pauvrement" codée de se planter et de perdre les données de l'utilisateur
- En revanche, lorsqu'une erreur survient dans une application bien codée, l'application avertit l'utilisateur et essaie de se récupérer après l'erreur.
  - Si elle ne peut pas se récupérer, elle sauvegarde le plus possible de données, nettoie les ressources, et arrête son exécution le plus proprement possible.

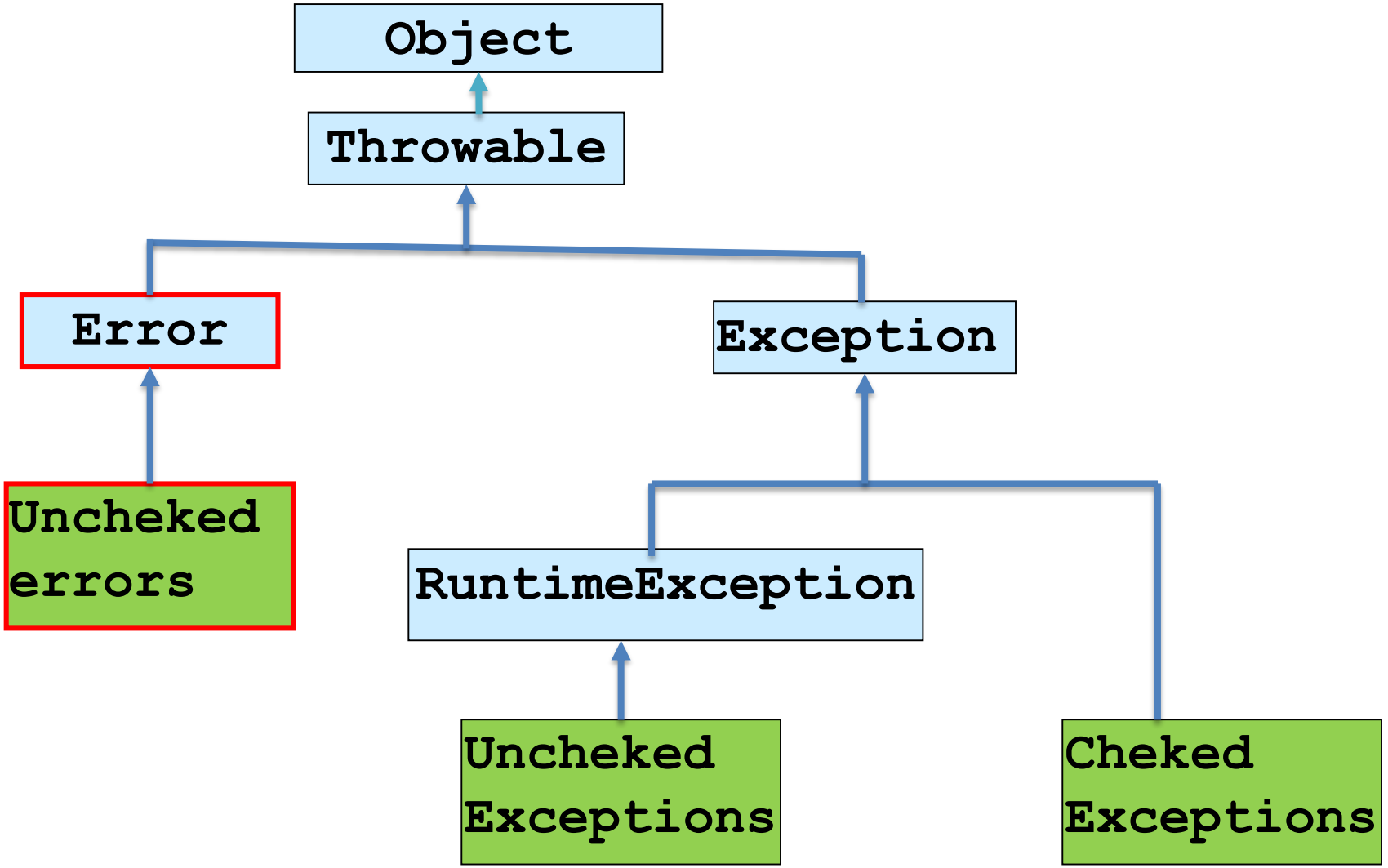
- Les anciens langages n'offraient pas de moyens pour contrôler les erreurs.
- Pire encore, ils ne permettaient pas de communiquer ces erreurs aux autres parties de l'application qui peuvent avoir besoin d'avoir des informations dessus.
- Pour résoudre ce problème, la plupart des langages modernes, y compris java, fournissent un mécanisme de gestion des erreurs, connu sous le nom d'exceptions.

- Les exceptions vous permette d'écrire du code robuste capable de gérer les erreurs de manière facile et fiable.
- Avant de présenter comment gérer les erreurs, nous allons présenter :
  - La hiérarchie des exceptions
  - Le mécanisme de gestion des exceptions

Hiérarchie des exceptions

- En Java, une exception est un objet instance de la classe Exception ou de l'une de ses sous-classes.
- Une exception représente une erreur qui s'est produit lors de l'exécution d'une application, et contient des informations sur cette erreur.
- Comme le montre la figure de la diapositive suivante. Toutes les exceptions dérivent de la classe Throwable.

Arbre d'héritage des exceptions



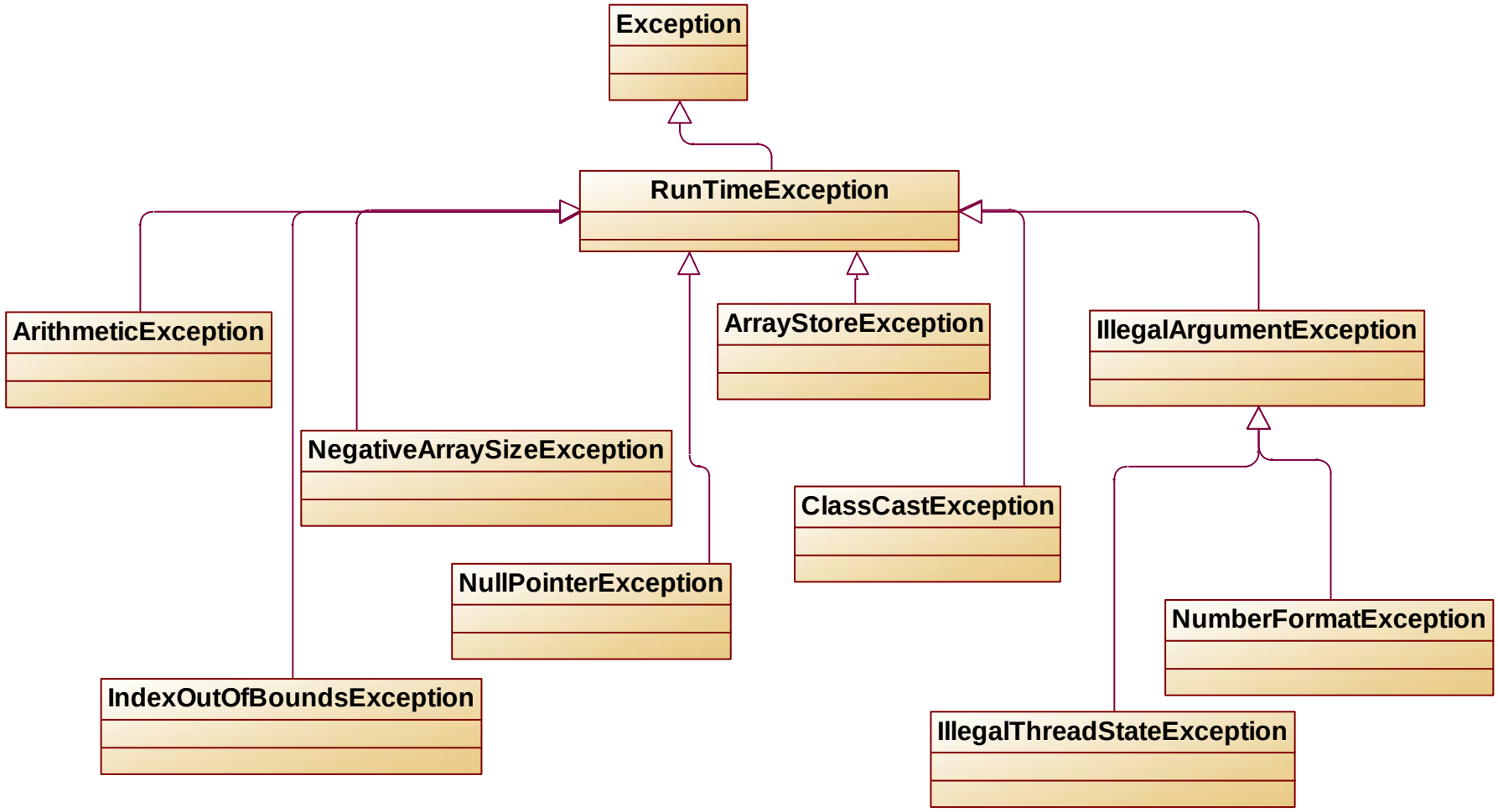
- Comme le montre le schéma, deux classes héritent directement de la classe Throwable:
  - Error
  - Exception
- Les classes qui héritent de la classe Error représentent des erreurs internes contre lesquelles, vous ne pouvez rien faire, comme les erreurs de l'environnement d'exécution (JVM).
- Par conséquent, vous pouvez ignorer ce type d'erreurs dans la plupart du temps.
- En revanche, vous avez besoin de gérer les exceptions qui héritent de la classe Exception.

- Les classes qui dérivent de la classe Exception se divisent en deux catégorie:
  - Les exceptions qui dérivent de la classe RunTimeException
  - Les autres sous classes qui ne dérivent pas de RunTimeException.
- Les exceptions qui dérivent de RunTimeException sont appelées des **exceptions non contrôlées** puisque le compilateur ne vous forcent pas de les gérer
- Par contre, le compilateur exige qu'on gère de façons explicites les exceptions qui ne dérivent pas de RunTimeException. Elles sont alors appelées **Exceptions contrôlées.**

- Les exceptions non contrôlées survient souvent à cause d'une erreur de codage.
- Par exemple, si une application essaie d'accéder à un tableau avec un indice incorrect (par exemple, supérieur à la longueur du tableau) Java lance un objet exception instance de ArrayIndexOutOfBoundsException qui est sous classe de IndexOutOfBoundsException.
- Si vous êtes attentif lorsque vous écrivez votre code, vous pouvez généralement éviter ce type d'exception d'être lancé.

- Les exceptions contrôlées, surviennent dans des circonstances qui sont en dehors du contrôle du programmeur.
  - Comme par exemple un fichier manquant, ou une mauvaise connexion réseau.
- Bien que vous ne pouvez pas éviter ces exceptions, vous pouvez écrire le code qui les traite lorsqu'ils se produisent.

Sous classes de la classe RunTimeException
Dans le package java.lang



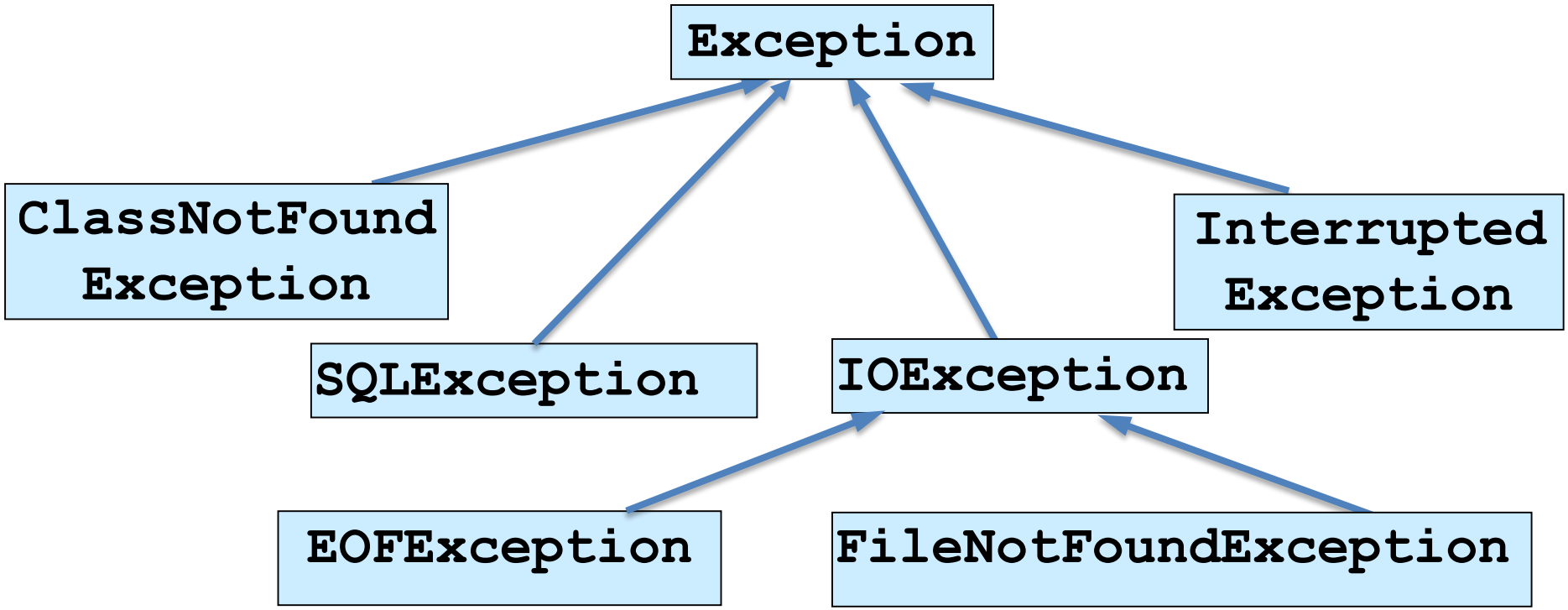
Nom de la classe	Condition de lancement de l'exception
ArithmeticException	Opération arithmétique invalide telle que la division d'un entier par zéro.
IndexOutOfBoundsException	Utilisation d'un indice qui est en dehors des bornes autorisés . L'indice peut être d'un tableau, d'une String ou d'un objet Vector. Cette classe possède deux sous classes : ArrayIndexOutOfBoundsException , et StringOutOfBoundsException.

Nom de la classe	Condition de lancement de l'exception
NegativeArraySizeException	Essai de créer un tableau avec une dimension négative
NullPointerException	Essai d'utilisation d'un objet dont la valeur est null, pour accéder à un attribut ou appeler une méthode
ArrayStoreException	Essai de stocker un objet dans un tableau dont le type des éléments ne l'autorise pas

Nom de la classe	Condition de lancement de l'exception
ClassCastException	Essai d'effectuer un cast d'un objet vers un type invalide. L'objet n'est pas du type de la classe, d'une sous classe, ou de la superclasse de la classe vers laquelle on veut effectuer le cast.
IllegalArgumentException	Essai de passer un argument à une méthode qui n'est pas du type du paramètre correspondant. C'est une classe de base des sous classes IllegalThreadStateException, et NumberFormatException.

Nom de la classe	Condition de lancement de l'exception
llegalThreadStateException	est lancée lors d'une opération qui est illégale dans l'état actuel du thread.
NumberFormatException	est lancée avec les méthodes **valueOf()**, et **decode()** qui se trouvent dans les classes qui représentent les entiers (Byte, Short, Integer, et Long). Les méthodes **parseXXX()** présentes dans ces classes peuvent égament lancer NumberFormatException. L'exception est lancée si l'objet String passé en argument contient un caractère invalide qui empêche la conversion vers un entier.

Quelques exceptions du JDK, contrôlées par le compilateur



Exceptions : Exemple 1

```
class ExempleSansExcep1{
    public static void main(String[] args){
        int n ;
        if(args.length > 0){
           n = Integer.parseInt(args[0]);
            System.out.println("C'est trés bien vous avez fourni un entier : " +n);
        }
    }
}
```

```
java ExempleSansExcep1 5
C'est tres bien vous avez fourni un entier : 5

java ExempleSansExcep1 a
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at ExempleSansExcep1.main(ExempleSansExcep1.java:5)
```

Exceptions : Exemple 2

```
public class Div {

  public static int divint (int x, int y) {
     return (x/y);
  }
    static void main (String [] args) {
       int c=0,a=1,b=0;
       c= divint(a,b);
       System.out.println("res: " + c);
       System.exit(0);
    }
}
```

Le système affiche l'erreur suivante:

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Div.divint(Div.java:3)
at Div.main(Div.java:7)

Une erreur s'est produite à la ligne 3 de la méthode divint appelée par la méthode main à la ligne 7.

Exceptions : Exemple 3

```
class Point{
        private int x, y;
        public int getX(){ return x;      }
        public void setX(int x){ this.x = x;  }
}
public class ExempleExcep2{
   public static void main(String[] args){
     Point p1 = null;
     p1.setX(5);
     System.out.println("l'abscisse du point est : " + p1.getX());
   }
}
```

```
java ExempleSansExcep2

Exception in thread "main" java.lang.NullPointerException
    at ExempleSansExcep2.main(ExempleSansExcep2.java:9)
```

Exceptions : Exemple 4

```
class ExempleExcep1{
   public static void main(String[] args){
     int n = -1;
     try{
            if(args.length > 0){
               n = Integer.parseInt(args[0]);
               System.out.println("C'est trés bien vous avez fourni un entier : " +n);
            }
     }
     catch (NumberFormatException e){
        System.out.println("Vous n'avez pas fourni un entier :");
        System.out.println(e.getMessage());
      }
     finally{
         System.out.println("Je suis toujours exécutée!!!");
     }
     }
}
```

Exceptions : Exemple 5

```
class Point{
            private int x, y;
            public int getX(){ return x;      }
            public void setX(int x){ this.x = x;  }
}
public class ExempleExcep2{
   public static void main(String[] args){
     Point p1 = null;
     try{
         p1.setX(5);
         System.out.println("l'abscisse du point est : " + p1.getX());
      }
       catch (NullPointerException e){
         System.out.println("Vous avez oublie de creer l'objet p1");
         System.out.println(e.getMessage());
     }
     finally{
         System.out.println("Je suis toujours exécutée!!!");
     }
   }}
```

Exceptions : Exemple 6

```
class Point{
        private int x, y;
        public int getX(){ return x; }
        public void setX(int x){  this.x = x; }}

public class ExempleExcep3{
   public static void main(String[] args){
     Point p1 = null;   int n;
      try{
            n = Integer.parseInt(args[0]);
            System.out.println("Vous avez bien saisi un entier : " + n);
            p1.setX(5);
            System.out.println("l'abscisse du point est : " + p1.getX());
       }
     catch (NumberFormatException e){
            System.out.println("Vous n'avez pas fourni un entier :");
            System.out.println(e.getMessage());
     }
      catch (NullPointerException e){
            System.out.println("Vous avez oublie de creer l'objet p1");
            System.out.println(e.getMessage());
     } }}
```

Exceptions : Exemple 7

```
class ErrConst extends Exception{
}
```

```
java TestEntNat1
n1 = 20
**Erreur construction entier naturel**
```

```
class EntNat{
  private int n;
  public EntNat (int n) throws ErrConst {
       if (n<0) throw new ErrConst();
       this.n = n;
   }
  public int getN(){ return n; }
}
```

```
public class TestEntNat{
  public static void main(String[] args){
    try{
              EntNat n1 = new EntNat(20);
              System.out.println("n1 = " + n1.getN());
              EntNat n2 = new EntNat(-20);
              System.out.println("n2 = " + n2.getN());
            }
    catch(ErrConst e){
              System.out.println("**Erreur construction entier naturel);
              System.exit(-1);
    }
  }}
```

Exceptions : Exemple 8

```
class ErrConst extends Exception{
   private int valeur;
   public ErrConst(int valeur){
    this.valeur = valeur;}
   public int getValeur(){ return valeur; }
}
```

```
class EntNat{
   private int n;
   public EntNat (int n) throws ErrConst {
        if (n<0) throw new ErrConst(n);
        this.n = n;
    }
   public int getN(){ return n; }
}
```

```
public class TestEntNat{
  public static void main(String[] args){
    try{
              EntNat n1 = new EntNat(20);
              System.out.println("n1 = " + n1.getN());
              EntNat n2 = new EntNat(-20);
              System.out.println("n2 = " + n2.getN());
            }
    catch(ErrConst e){
             System.out.println("**Err: construction entier naturel avec:"+e.getValeur());
             System.exit(-1);
    }
  }}
```

Exceptions contrôlées

Pour toutes les autres classes (différentes de RunTimeException) qui dérivent de la classe Exception, le compilateur vérifie que l'exception est soit traitée par la méthode où elle a été lancée, soit que vous avez indiqué que cette méthode peut lancer une telle exception. Si vous ne faites ni l'une ni l 'autre, votre code ne se compilera pas.

Gestion des exceptions

Si votre code peut lancer une exception qui n'est pas de type Error, ou de type RunTimeException (ou une sous classe de Error ou RunTimeException) vous devez vous en occuper en suivant l'une des deux méthodes suivantes:

1. Spécifier que votre méthode peut lancer l'exception pour qu'elle soit traitée par la méthode appelante
2. Ecrire le code dans la méthode où est lancée l'exception pour la traiter.

Spécification des exceptions qu'une méthode peut lancer

Pour déclarer qu'une méthode peut lancer des exceptions, il suffit de mettre, juste après la liste de ses paramètres, le mot clé **throws** suivi de la liste des classes exceptions séparées par des virgules.

Exemple:

```
double myMethode() throws EOFException, FileNotFoundException{
        //corps de la méthode
}
```

Attention:

La méthode qui appelle une méthode qui lance des exceptions doit à son tour les lancer ou contenir du code qui les traite. Dans le cas contraire, le compilateur génère une erreur, et le code n'est pas compilé.

Traitement des exceptions

Pour traiter une exception, il existe trois types de blocs de code que vous pouvez inclure dans la méthode qui va se charger du traitement:

- **try**
- **catch**
- **finally**

Le bloc **try** délimite le code qui est sensé lancer une ou plusieurs exceptions.

Le bloc **catch** délimite le code qui est destiné à traiter les exceptions d'un type particulier qui pourraient être lancée par le code du bloc try.

Le bloc **finally** délimite un code qui sera toujours exécuté qu'il y ait ou non lancement d'exceptions par le code du bloc try, et traitement ou non d'une éventuelle exception par un bloc catch.

Bloc try

Lorsque vous voulez attraper une exception, vous devez placer le code susceptible de lancer une telle exception dans un bloc try.

Un bloc try consiste simplement en le mot clé try suivi de deux accolades qui englobent le code susceptible de lancer l'exception.

```
try {
        // code qui peut lancer une ou plusieurs exceptions
}
```

Il n'est pas obligatoire de mettre un code qui peut causer des exceptions dans un bloc try. Seulement, la méthode qui contient un tel code ne peut pas attraper ces exceptions si elles sont lancées, et il faut alors que la méthode déclare avec throws qu'elle peut lancer les types d'exceptions qui ne peuvent pas être attrapées.

Même si on est en train de discuter des exceptions que nous devons traiter, un bloc try est aussi nécessaire si vous voulez attraper des exceptions de type Error ou RunTimeException.

Bloc catch

Un bloc catch englobe le code qui traite un type donné d'exception.
Le bloc catch doit suivre immédiatement le bloc try qui contient le code qui peut avoir causé ce type d'exception.

Un bloc catch consiste en le mot clé catch, suivi d'un paramètre entre parenthèses qui identifie le type d'exception que le bloc doit traiter, suivi du code, qui traite l'exception, placé entre des accolades.

```
Catch (ClasseException e){
        // Code  qui traite l'exception de type ClasseException
}
```

En général le paramètre du bloc catch doit être de type throwable ou d'une des sous classes de throwable.
Le bloc catch traitera les exceptions de type de la classe du paramètre, et de toutes les sous classes de cette classe.
Par exemple, si vous spécifiez un paramètre de type RunTimeException, le code du bloc catch sera invoqué pour traiter les exceptions de type RunTimeException, et celles de ses sous classes.

Exemple :

```
Try {
        // Code qui peut lancer une ou plusieurs exceptions
}
Catch (ArithmeticException e) {
        // Code qui traite l'exception de type ArithmeticException et pas les autres
}
```

Ce bloc catch traite uniquement les exceptions de type ArithmeticException. Ceci implique que ce sont les seuls exceptions qui peuvent être lancés dans le bloc try.

Si d'autres exceptions peuvent être lancés, le code ne s'exécutera pas.

Nous verrons ultérieurement comment traiter plusieurs exceptions qui sont lancées par un même bloc try.

```
public class TestTryCatch {
  public static void main(String[] args) {
    int i = 1;
    int j = 0;

    try {
      System.out.println("Try block entered " + "i = "+ i + " j = "+j);
      System.out.println(i/j);        // Divide by 0 - exception thrown
      System.out.println("Ending try block");

    } catch(ArithmeticException e) { // Catch the exception
      System.out.println("Arithmetic exception caught");
    }

    System.out.println("After try block");
    return;
  }
}
```

Exécution

```
Try block entered i = 1 j = 0
Arithmetic exception caught
After try block
```

L’exécution montre que le bloc try n'a pas été entièrement exécuté, puisque l'instruction println(i/j) lance une exception, le bloc try s'arrête et l'exécution continue au bloc catch associé.

Remarque:

En ce qui concerne le scope des variables, les blocs try et catch ont le même comportement qu’un bloc standard.

Par conséquent, une variable déclarée dans un bloc try ou catch n’est visible que dans ce bloc, elle n’est pas visible à l’extérieur.

Liaison des blocs try/catch

Les blocs try et catch vont ensemble. Vous ne devez pas les séparer par des instructions ou par quoi que ce soit. Si vous avez une boucle qui est aussi un bloc try, le bloc catch qui suit doit être aussi une partie de la boucle.

Que fait ce programme ?

```
public class TestLoopTryCatch {
  public static void main(String[] args) {
    int i = 12;


    for(int j=3 ;j>=-1 ; j--) {
      try {
        System.out.println("Try block entered " + "i = "+ i + " j = "+j);
        System.out.println(i/j);        // Divide by 0 - exception thrown
       System.out.println("Ending try block");

      } catch(ArithmeticException e) {     // Catch the exception
        System.out.println("Arithmetic exception caught");
      }
    }

    System.out.println("After try block");
    return;
  }
}
```

Exécution :

```
Try block entered i = 12 j = 3
4
Ending try block
Try block entered i = 12 j = 2
6
Ending try block
Try block entered i = 12 j = 1
12
Ending try block
Try block entered i = 12 j = 0
Arithmetic exception caught
Try block entered i = 12 j = -1
-12
Ending try block
After try block
```

Que fait ce programme ?

```
public class TestLoopTryCatch2 {
  public static void main(String[] args) {
    int i = 12;

    try {
      System.out.println("Try block entered.");
      for(int j=3 ;j>=-1 ; j--) {
        System.out.println("Loop entered " + "i = "+ i + " j = "+j);
        System.out.println(i/j);        // Divide by 0 - exception thrown
        }
      System.out.println("Ending try block");

    } catch(ArithmeticException e) {    // Catch the exception
      System.out.println("Arithmetic exception caught");
    }
  }
}
```

Bloc try avec de multiple blocs catch

Si un bloc try peut lancer plusieurs types différents d'exception, vous pouvez placer plusieurs blocs catch pour les traiter.

```
try {
        // Code qui peut lancer plusieurs types d'exceptions
}

catch (ArithmeticException e){
        // Code qui traite les exceptions de type ArithmeticException
}

catch (IndexoutOfBoundsException e){
        // Code qui traite les exceptions de type IndexoutOfBoundsException
}

// L'exécution continue ici ….
```

Les exceptions de type ArithmeticException seront attrapées par le premier catch, et celles de type IndexoutOfBoundsException seront attrapées par le deuxième catch.

Si une exception de type ArithmeticException est lancée, seul le code du bloc catch correspondant est exécuté. L'exécution se poursuit ensuite à l'instruction qui suit le dernier bloc catch.

Lorsque vous voulez attraper des exceptions de plusieurs types différents qui peuvent être lancé par un même bloc try, l'ordre des blocs catch est important.

Lorsqu'une exception est lancée, elle sera attrapée par le premier bloc catch dont le type du paramètre est le même que celui de l'exception, ou de type d'une superclasse de la classe type de l'exception.

Le cas extrême serait si vous spécifiez un bloc catch avec un paramètre de type Exception, ce bloc attrapera n'importe quelle exception de type Exception ou de type d'une sous classe de la classe Exception.

Les blocs catch doivent être placés avec le bloc dont le paramètre est de type le plus dérivé en premier, et le bloc catch dont le paramètre est de type de la classe de base en dernier**. Dans le cas contraire, le code ne sera pas compilé.**

La raison de cela est que si un bloc catch qui attrape une exception d'un type donné précède le bloc catch qui attrape une exception d'un type qui dérive de type précédent, le second bloc catch ne sera jamais exécuté.

```
// Sequence invalide de blocs catch

Try {
        // Code qui peut lancer des exceptions de type
        // ArithmeticException et d'autres exception de type sous classes
        // de Exception
}

Catch (Exception e) {
        // Code qui attrapera toute exception de type Exception
        // ou de toute classe dérivé de la classe Exception
}

Catch (ArithmeticException e) {
        // Code qui attrape uniquement les exceptions de type
        // ArithmeticException et de ses sous classes
}
```

Ce code ne compilera pas parce que le deuxième catch ne s'exécutera jamais, puisque le premier catch attrapera toutes les exceptions qui dérivent de la classe Exception y compris ArithmeticException.

Le bloc finally

Lorsqu'une exception est lancée dans un bloc try, l'exécution du bloc est arrétée à l'instruction qui cause l'exception, quel que soit l'importance du code qui vient après.
Ceci peut avoir comme effet de laisser les choses dans un état non satisfaisant, car il est possible d'avoir ouvert un fichier où établir une connexion réseau, et le lancement d'une exception ne laissera pas les instructions destinées à fermer les fihiers et les connexions s'exécuter.

Le bloc finally fournit le moyen de faire le nettoyage après l'exécution d'un bloc try.
Vous utilisez un bloc finally lorsque vous voulez être sûr qu'un code sera exécuté avant de quitter la méthode qu'il y ait ou non lancement d'exceptions dans le bloc try.

Un bloc finally est toujours exécuté quel que soit ce qui arrive dans une méthode.

Le bloc finally a une structure simple:

```
Finally  {
            // code de nettoyage toujours exécuté et en dernier
}
```

Comme un bloc catch, un bloc finally doit toujours être associé à un bloc try, et doit être placé après le dernier catch associé au même try.

S'il n'ya aucun catch associé à un bloc try, le bloc finally doît être placé immédiatement après le bloc try.

Il est bien sûr possible d'avoir un bloc try avec uniquement des blocs catch sans qu'il n'y ait de bloc finally.

Structure d'une méthode

```
Double doSomething(int aParam)
throws exceptionType1, ExceptionType2
{
  //Code qui ne lance pas d'exception
//ensemble de blocs try/catch/finally
//Code qui ne lance pas d'exception
 //ensemble de blocs try/catch/finally
//Code qui ne lance pas d'exception
 //ensemble de blocs try/catch/finally
//Code qui ne lance pas d'exception

//….

}
```

```
try {
        // Code qui peut lancer des exceptions
}
```

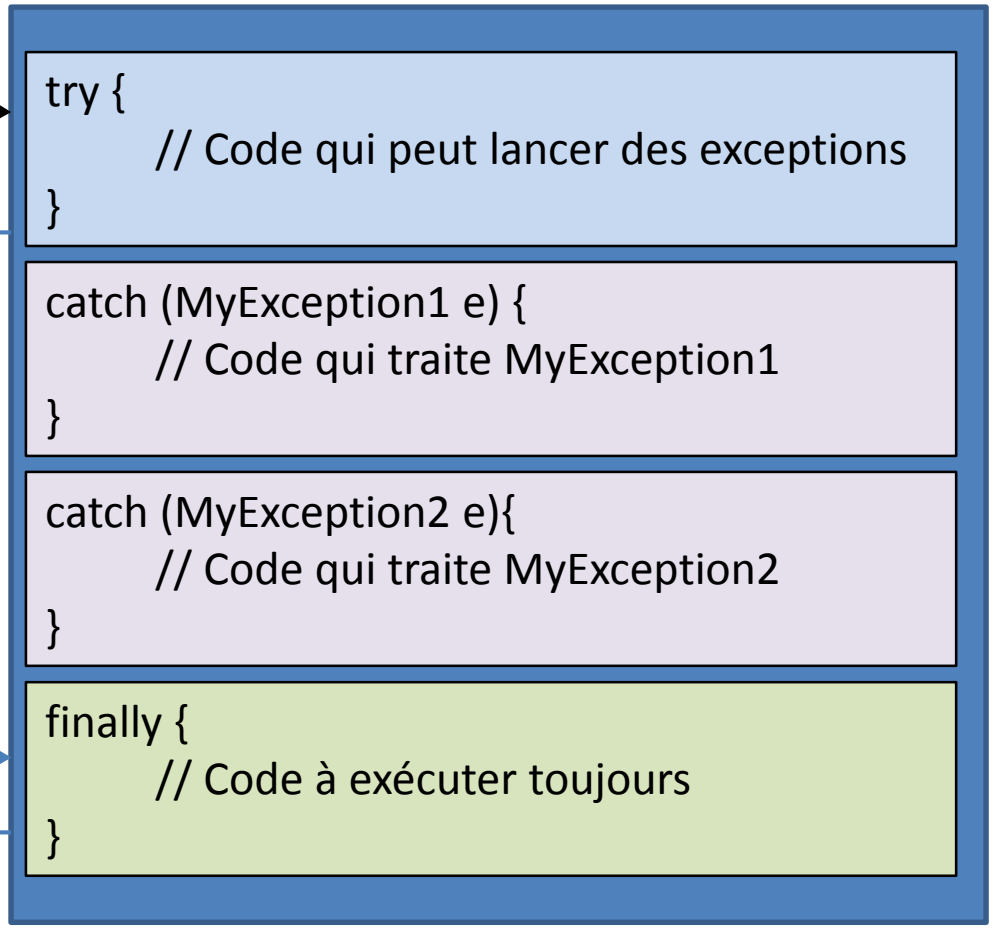
```
catch (MyException1 e) {
        // Code qui traite MyException1
}
```

```
catch (MyException2 e){
        // Code qui traite MyException2
}
```

```
finally{
        // Code à exécuter toujours
}
```

Exécution normale d'une méthode (aucun lancement d'exception)

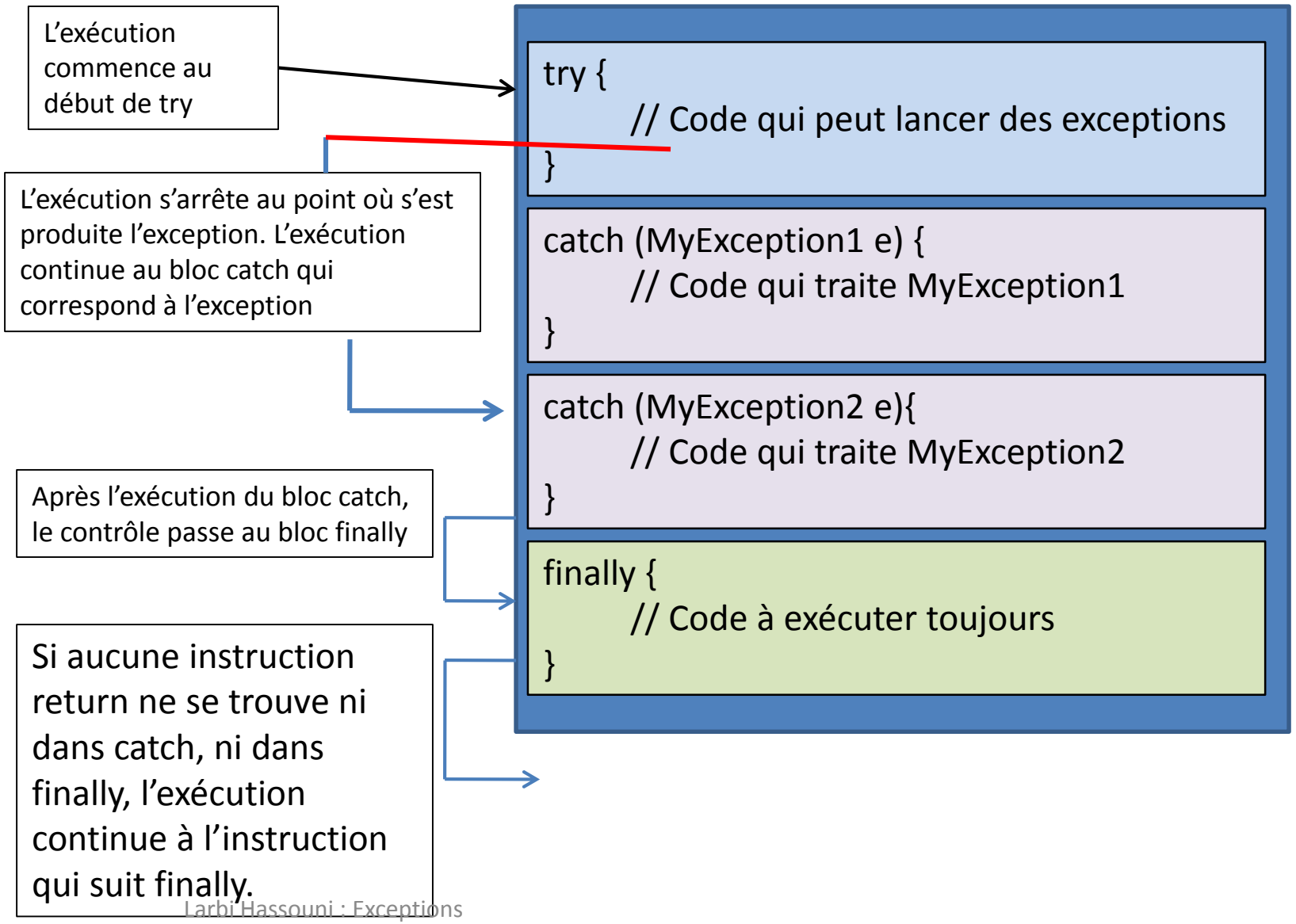
L'exécution commence au début de try



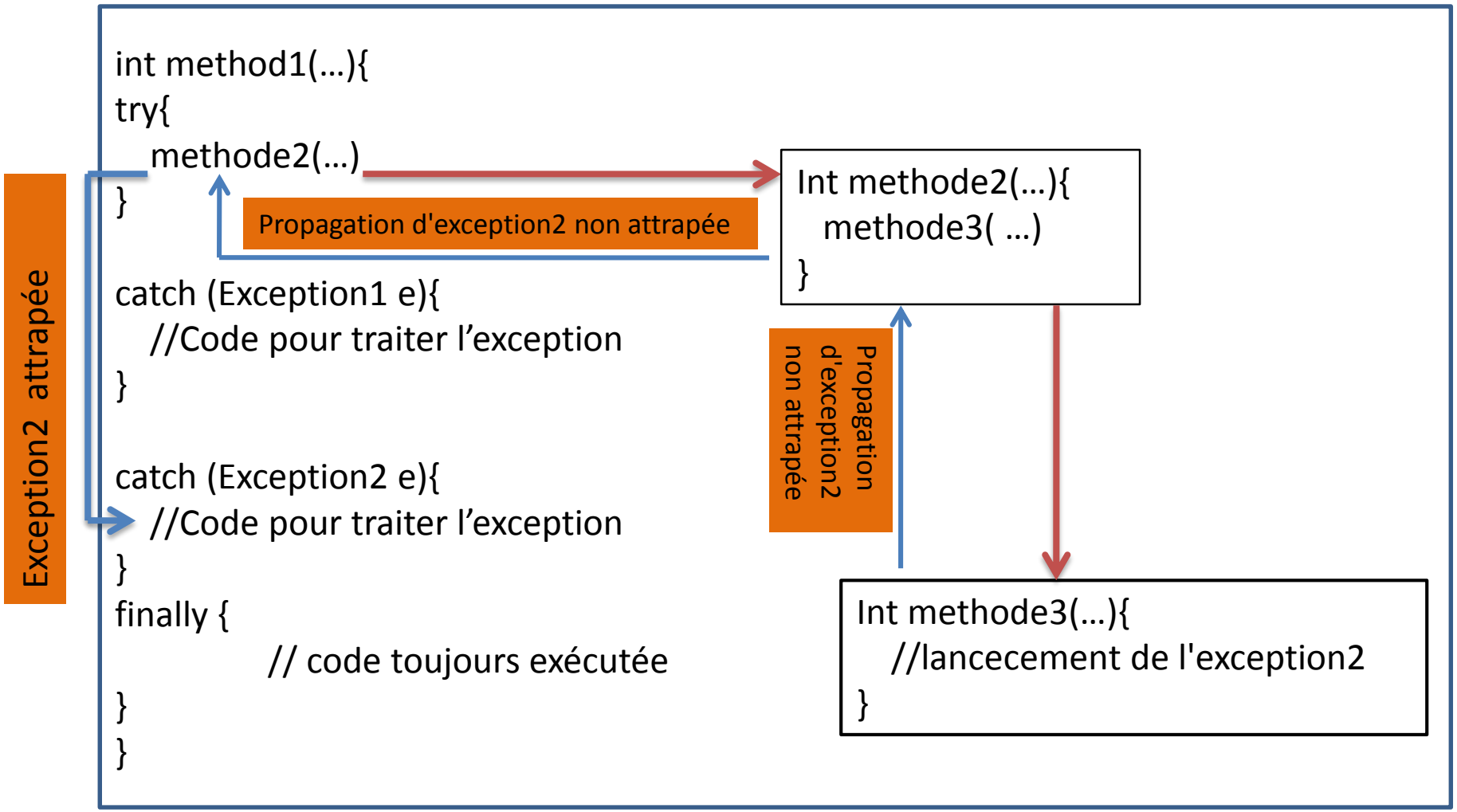
Après une sortie normale du bloc try, **le bloc finally est exécuté, avant toute instruction return dans le bloc try**

Si aucune instruction return ne se trouve ni dans try, ni dans finally, l'exécution continue à l'instruction qui suit finally.

Exécution lorsqu'une exception est lancée et attrapée



Exécution lorsqu'une exception est lancée mais pas attrapée. Propagation d'une exception



Une exception lancée mais non attrapée se propage d'un niveau à l'autre jusqu'à ce qu'elle soit attrapée ou jusqu'au niveau le plus élevé. Si elle n'est pas traitée dans ce dernier niveau, l'exécution du programme est interrompue

Que fait ce programme ?

```
public class TryBlockTest {
  public static void main(String[] args) {
    int[] x = {10, 5, 0};              // Array of three integers

    // This block only throws an exception if method divide() does
    try {
      System.out.println("First try block in main() entered");
      System.out.println("result = " + divide(x,0));  // No error
      x[1] = 0;                        // Will cause a divide by zero
      System.out.println("result = " + divide(x,0));  // Arithmetic error
      x[1] = 1;                        // Reset to prevent divide by zero
      System.out.println("result = " + divide(x,1));  // Index error

    } catch(ArithmeticException e) {
      System.out.println("Arithmetic exception caught in main()");
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Index-out-of-bounds exception caught in main()");
    }
    System.out.println("Outside  try block in main()");
  }
```

```
// Divide method
  public static int divide(int[] array, int index) {
    try {
      System.out.println("\nFirst try block in divide() entered");
      array[index + 2] = array[index]/array[index + 1];
      System.out.println("Code at end of first try block in divide()");
      return array[index + 2];
    } catch(ArithmeticException e) {
      System.out.println("Arithmetic exception caught in divide()");
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Index-out-of-bounds exception caught in divide()");
    } finally {
      System.out.println("finally block in divide()");
    }
    System.out.println("Executing code after try block in divide()");
    return array[index + 2];
                }
}
```

Blocs try emboités

```
try{
    try{
        //premier bloc try interne
    }
    catch(Exception1 e){
        //……
    }
  //bloc try externe
    try{
        //second bloc try interne
    }
    catch(Exception1 e){
        //……..
    }
}
Catch (Exception2 e){
    bloc  catch du bloc try externe
}
```

Le bloc catch externe peut attraper n'importe quelle exception lancée mais non attrapée à l'intérieur du bloc try , y compris dans les blocs try internes

Relance d'une exception

```
try{
    //code qui lance une ArithmeticException
}
catch(ArithmeticException e){
    traitement de l'exception
    //lancement de l'exception vers le programme appelant
     throw e;
}
```

Une exception qui a été lancée et traitée dans une méthode appelée peut être relancée vers le programme appelant en vue de terminer son traitement et en tenir compte dans la suite des opérations du programme.

Objets Exception

L'objet Exception qui est passé comme argument au block catch peut fournir des informations importantes sur la nature du problème qui a causé l'exception.

La classe de base des exceptions est la classe **Throwable**, par conséquent chaque objet Exception hérite ses membres

Classe Throwable

La classe Throwable contient deux constructeurs public:

- Le constructeur par défaut
- un constructeur qui accepte un argument de type String. Cet argument est utilisé pour fournir une description de la nature du problème qui a causé l'exception.

Un objet Throwable contient deux entités d'information:

- Un message, qui est initialisé par le constructeur
- Un enregistrement de la pile d'exécution au moment où l'objet Exception est créé, autrement dit au moment où l'exception est lancée.

La pile d'exécution garde la trace de toutes les méthodes qui sont en cours d'exécution à un moment donné. Elle fournit le moyen qui permet de retourner au programme appelant après exécution d'une instruction return de la méthode appelée.

L'enregistrement de la pile d'exécution stocké dans l'objet

Exception consiste en le numéro de la ligne du code source où l'exception s'est produite, suivie par une trace des appels de méthodes qui précèdent le point où l'exception est arrivée.

Cette trace des appels de méthodes contient le FQN de chaque méthode appelée, plus le numéro de la ligne du code source où l'appel de méthode a eu lieu.

Ces appels sont en séquence, avec l'appel de méthode le plus récent en premier. Cela aide à comprendre comment on est arrivé au point qui cause l'exception

La classe Throwable contient les méthodes public ci-dessous pour permettre l'accès au message et à l'enregistrement de la pile d'exécution:

Méthode	Description
getMessage()	Retourne le message qui décrit l'exception. Il est généralement formé par le FQN de la classe exception, et une brève description de l'exception
printStackTrace()	Affiche à l'écran le message et l'enregistrement de la pile d'exécution

Méthode	Description
printTraceStack (PrintStream s)	Identique à printStackTrace, sauf qu'elle écrit dans un fichier.

fillInStackTrace()	Modifie le numéro de la ligne où s'est produite l'exception par celui où cette méthode est appelée.
	L'intérêt principal de cette méthode est lorsqu'on veut relancer une exception, et enregistrer le numéro de la ligne où la relance a été effectuée.
	e.fillInStackTrace(); //enregistrer la ligne de relance
	throw e; //relancer l'exception.

Pourquoi des exceptions non contrôlées ?

- Les exceptions non contrôlées peuvent survenir dans toute portion de code (par exemple **`NullPointerException`** ou **`IllegalArgumentException`**)
- Si ces exceptions étaient contrôlées, toutes les méthodes auraient une clause **`throws`** ou le code serait rempli de bloc **`try-catch`**

Définir ses propres exceptions

Il ya deux raisons principales pour définir ses propres exceptions:

Vous voulez ajouter des informations lorsqu'une exception standard se produit. Vous pouvez faire cela en lançant un objet de votre propre classe exception

Vous avez une condition d'erreur dans le programme qui garantit la distinction d'une classe d'exception spéciale.

Définition d'une classe Exception

une classe exception doit toujours avoir **Throwable** comme superclasse, sinon vous ne définissez pas une exception.

Même si vous pouvez dériver votre classe exception de n'importe quelle classe d'exception standard, la meilleure politique est de la dériver à partir de la classe **Exception**.

Cela permet au compilateur de garder trace de l'endroit où l'exception s'est produite, puis de contrôler si l'exception est attrapée (caught) ou lancée (thrown) dans la méthode.

Si vous utilisez RunTimeException comme superclasse, le compilateur supprimera le contrôle de l'existence des blocs catch pour votre exception.

Exemple de définition d'une classe exception

```
Public class MyException extends Exception{
    //Constructeurs
    public MyException(){}   //Constructeur par défaut
    public MyException(String s){
            super(s);          //appel du constructeur de la superclasse
    }
}
```

Par convention, votre classe exception doit inclure un constructeur par défaut , et un constructeur qui accepte un argument de type String.

Le message stocké dans la superclasse Exception sera automatiquement initialisé par le nom de votre classe quelque soit le constructeur utilisé pour créer l'objet exception.

Si vous utilisez le deuxième constructeur, l'argument String est ajouté au nom de la classe.

Il est bien sûr possible de définir d'autres constructeurs.

Il est également possible d'ajouter à votre classe des variables membres d'instances pour stocker davantage d'information sur les causes de l'exception, et des méthodes qui vont permettre d'accéder à ces informations dans un bloc catch.

Votre classe héritera bien sûr des méthode de la classe Throwable citées précédemment.

Lancement de votre propre exception

Pour lancer votre propre exception, vous utiliserez le mot clé **throw** suivi par un objet instance de votre classe exception:

```
MyException e = new MyException()

throw e;
```

ou tout simplement:

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

Cette instruction mettra immédiatement fin à l'exécution de la méthode qui la contient à moins qu'elle ne se trouve dans un bloc try ou catch qui est associé à un bloc finally, lequel est d'abord exécuté avant que la méthode ne se termine. L'exception est lancée au programme appelant au point où la méthode est appelée.

Si vous voulez ajouter un message spécifique, vous pouvez bien sûr utiliser le 2ème constructeur.

Exemples de traitements dans un bloc catch

- Fixer le problème et réessayer le traitement qui a provoqué le passage au bloc **`catch`**
- Faire un traitement alternatif
- Retourner (**`return`**) une valeur particulière
- Sortir de l'application avec **`System.exit()`**
- Faire un traitement partiel du problème et relancer (**`throw`**) la même exception (ou une autre exception)

Souplesse dans le traitement des exceptions

- La méthode dans laquelle l'erreur a eu lieu peut
  - traiter l'anomalie
    - pour reprendre ensuite le déroulement normal du programme,
    - ou pour faire un traitement spécial, différent du traitement normal
  - ne rien faire, et laisser remonter l'exception
  - faire un traitement partiel de l'anomalie, et laisser les méthodes appelantes terminer le traitement

Pourquoi laisser remonter une exception ?

- Plus une méthode est éloignée de la méthode **`main`** dans la pile d'exécution, moins elle a une vision globale de l'application
- Une méthode peut donc laisser remonter une exception si elle ne sait pas comment la traiter, en espérant qu'une méthode appelante en saura assez pour la traiter

Cas des exceptions non traitées

- Si une exception remonte jusqu'à la méthode **`main()`** sans être traitée par cette méthode,
  - l'exécution du programme est stoppée
  - le message associé à l'exception est affiché, avec une description de la pile des méthodes traversées par l'exception
- A noter : en fait, seul le *thread* qui a généré l'exception non traitée meurt ; les autres *threads* continuent leur exécution