POO-IG

Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université Paris Diderot
cristina@irif.fr

Exemples et matériel empruntés :

* Core Java - C.Horstmann - Prentice Hall Ed.
* POO in Java - L.Nigro & C.Nigro - Pitagora Ed.

Exceptions

Gestion des erreurs et exceptions

- Tout programme doit prévoir la présence d'erreurs pendant son exécution
  - comportements imprévus du programme
  - entrées non valides
  - erreurs de communication avec les périphériques / le réseau

- Il s'agit de cas "exceptionnels" dans l'exécution d'un programme

- Vérifier avec des tests toutes les situations potentielles d'erreur interférerait avec la logique du code et rendrait le code peu lisible

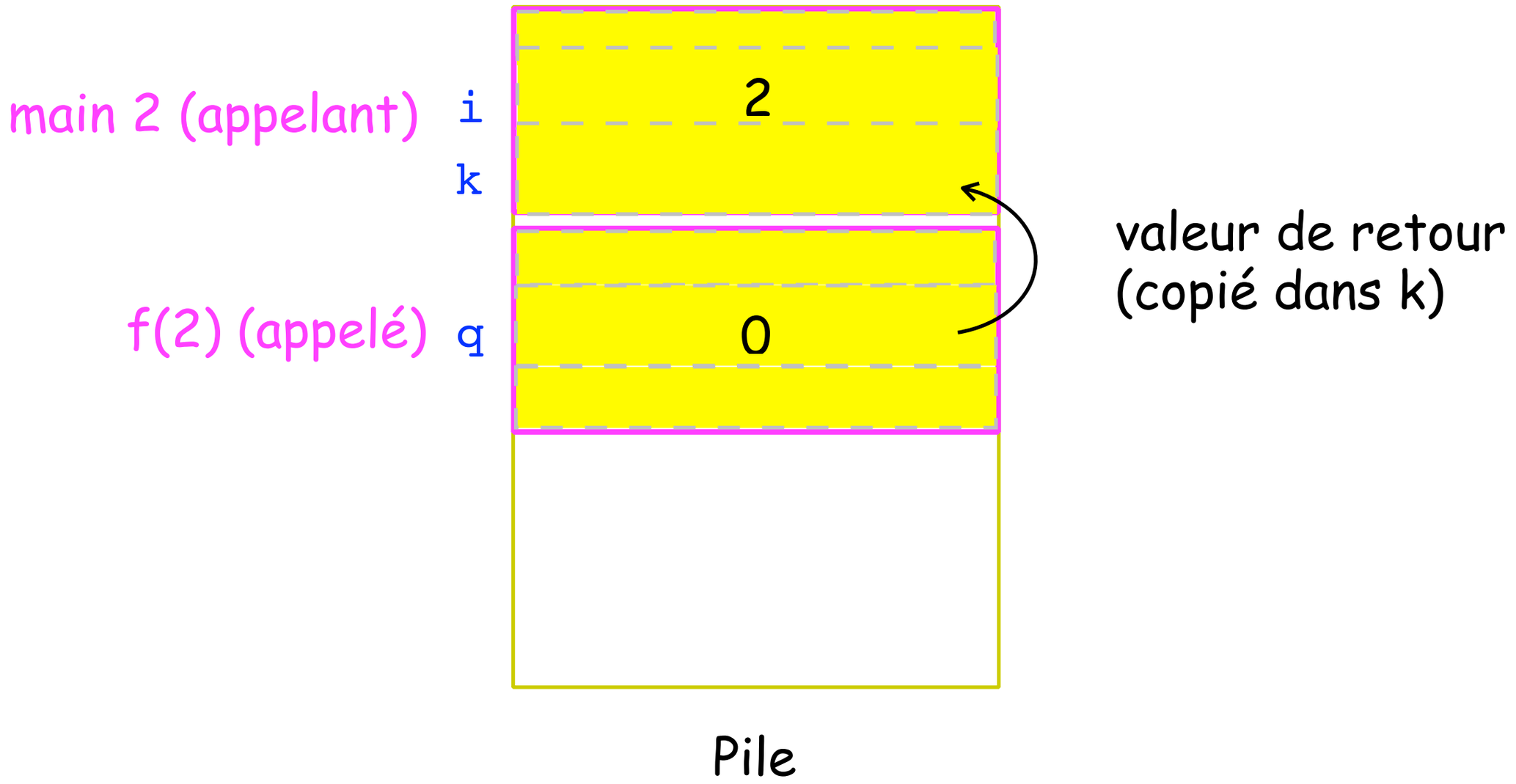
- Gestion des exceptions :
  - permet de séparer clairement le code qui décrit le comportement du programme en situation normale
  - du code qui gère les situations d'erreurs

Principe des exceptions

- Une méthode peut terminer
  - dans un état normal : l'exécution de la méthode a terminé correctement
  - dans un état d'erreur : une exception a été soulevée pendant l'exécution de la méthode, la méthode termine avant sa fin
- Dans les deux cas le contrôle revient à la méthode appelante

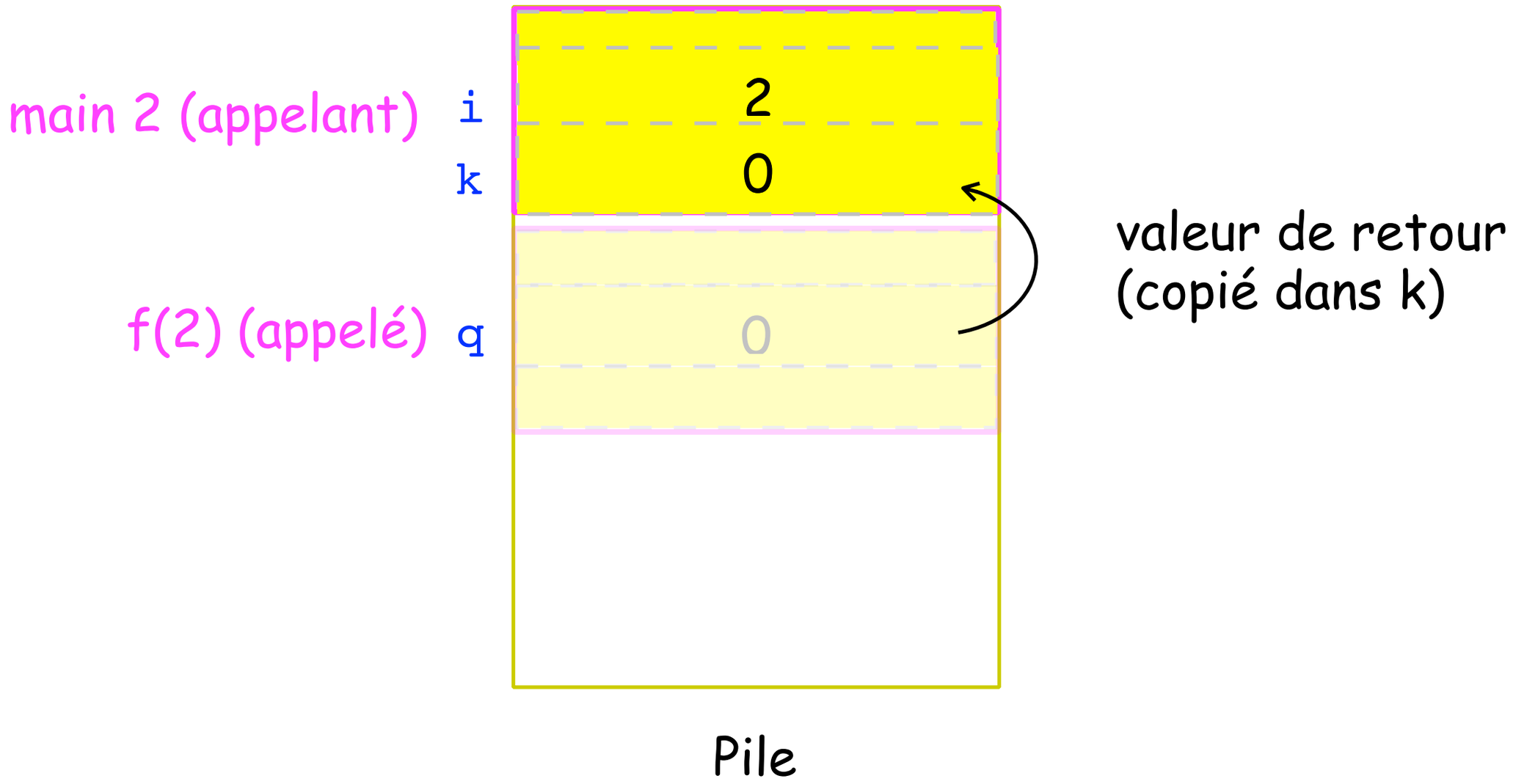
```
class ExceptionTest {
    private static int[] t = new int [5];
    public static void main (String[] args) {
        int i = Integer.valueOf (args[0]); int k = f(i);
        System.out.println (k);
        System.out.println ("Terminé.");
    }
    public static int f (int i) {
        int q = t[i] * t[i];
        return q;
    }
}
```

Méthode qui termine correctement



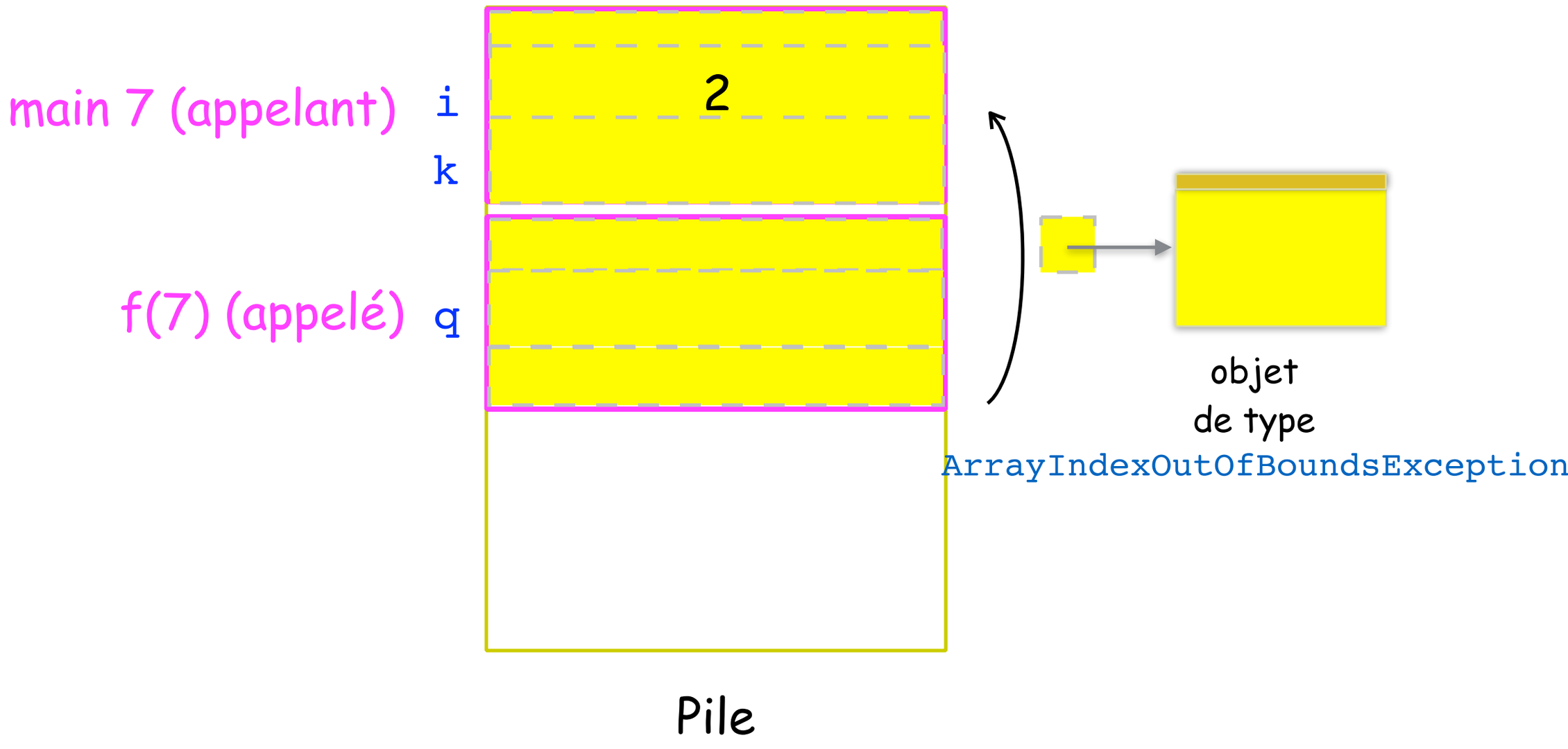
Méthode qui termine correctement

- Dans le cas de terminaison normale l'appelant reçoit la valeur de retour de la méthode



Méthode qui termine en état d'erreur

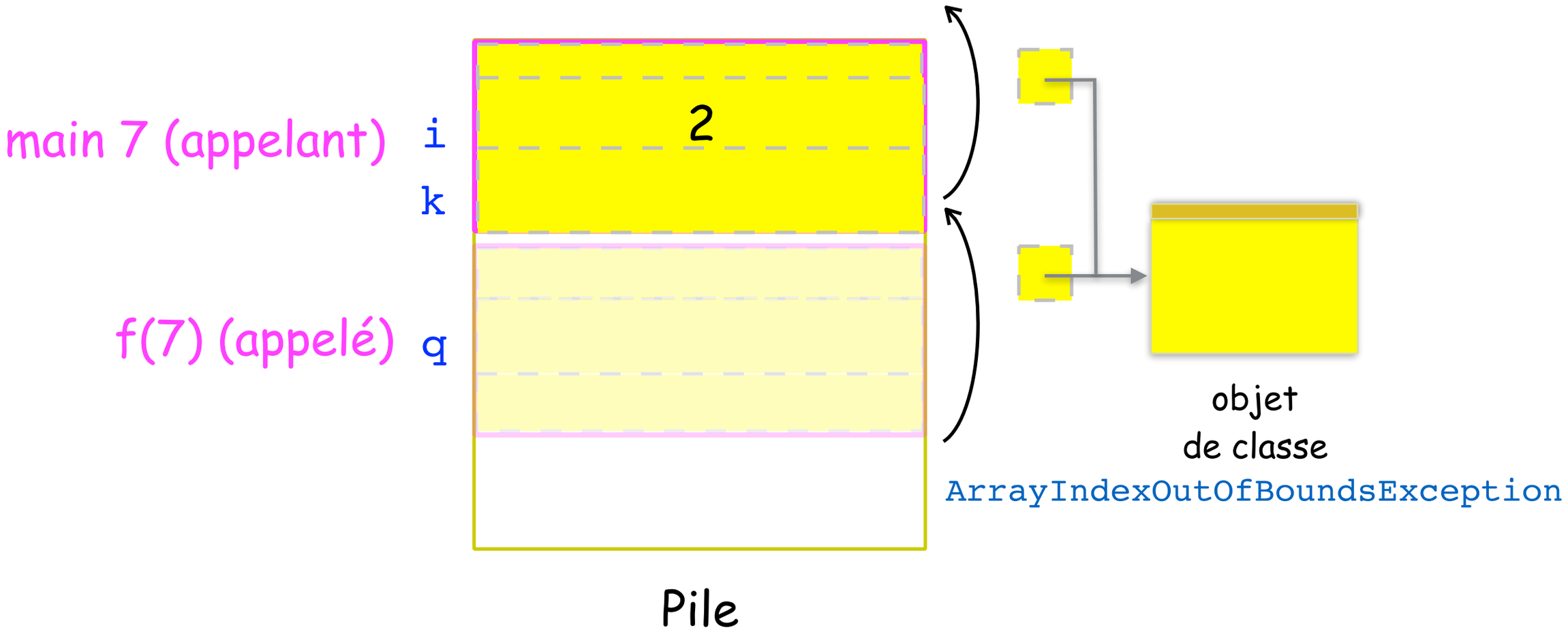
- Dans le cas de terminaison en état d'erreur, l'appelant reçoit un objet de type `Exception`, qui peut contenir des informations sur la situation d'erreur qui l'a produite



- La classe `Exception` a plusieurs sous-classes correspondantes à different types d'erreur

Méthode qui termine en état d'erreur

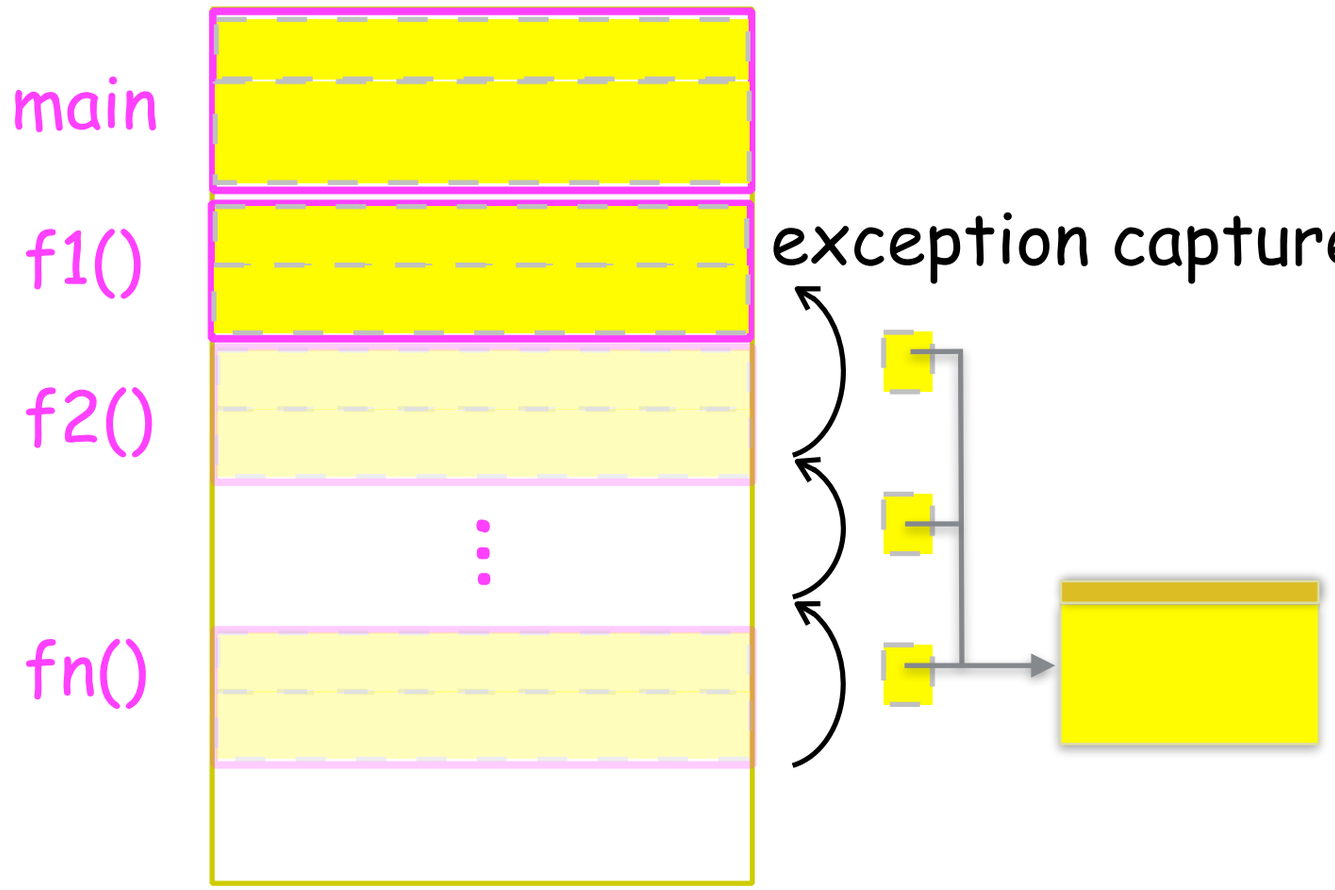
- L'appelant a deux choix :
  - gérer ("capturer") l'exception et continuer son exécution
  - ne rien faire (le cas ici): dans ce cas soulève à son tour l'exception, termine en situation d'erreur et rend le contrôle à son appelant, et ainsi de suite…



- Si le main soulève une exception le programme termine

Capturer une exception

- Une exception peut remonter toute la pile d'exécution
- Jusqu'à ce que un méthode ne la capture (ou qu'elle soit soulevée par le main)



- Si l'exception est capturée par une méthode, celle-là la gère et continue son exécution normalement

Capturer une exception : bloc try - catch

- Pour capturer une exception : bloc `try - catch`

```
class ExceptionTest1 {
    private static int[] t = new int [5];
    public static void main (String[] args) {
      int i = Integer.valueOf (args[0]);
      int k = 0;
      try {
        k = f(i);
        System.out.println (k);
      } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println
            ("Résultat pas calculé");
      }
      System.out.println ("Terminé");
    }
    public static int f (int i) {
        int q = t[i] * t[i];
        return q;
    }
}
```

Capturer une exception : bloc try - catch

```
try {
  <code qui peut soulever une exception>
} catch (<Type exception> e) {
    <code qui gère l'exception>
     //exécuté uniquement si le bloc try soulève
     // une exception de <Type exception>
}
//suite de la méthode exécutée :
// - directement après le bloc try, s'il ne soulève pas
     d'exceptions, ou,
// - après le block catch, si une exception a été
     capturée
```

Remarque : si un exception d'un type incompatible avec `<Type exception>` est soulevée pendant l'exécution du bloc `try`, la méthode termine en état d'erreur avant de compléter le bloc `try` et soulève l'exception non-capturée

Capturer une exception : bloc try - catch

```
class ExceptionTest1 {
    private static int[] t = new int [5];
    public static void main (String[] args) {
      int i = Integer.valueOf (args[0]);
      int k = 0;
      try {
        k = f(i);
        System.out.println (k);
      }
      catch (ArrayIndexOutOfBoundsException e) {
        System.out.println
             ("Résultat pas calculé");
      }
      System.out.println ("Terminé");
    }
    public static int f (int i) {
        int q = t[i] * t[i];
        return q;
    }
}
```

java ExceptionTest1 3
-> 0
-> Terminé.

Capturer une exception : bloc try - catch

```
class ExceptionTest1 {
    private static int[] t = new int [5];
    public static void main (String[] args) {
      int i = Integer.valueOf (args[0]);
      int k = 0;
      try {
        k = f(i);
        System.out.println (k);
      }
      catch (ArrayIndexOutOfBoundsException e) {
        System.out.println
            ("Résultat pas calculé");
      }
      System.out.println ("Terminé");
    }
    public static int f (int i) {
        int q = t[i] * t[i];
        return q;
    }
}
```

java ExceptionTest1 7
-> Résultat pas calculé
-> Terminé.

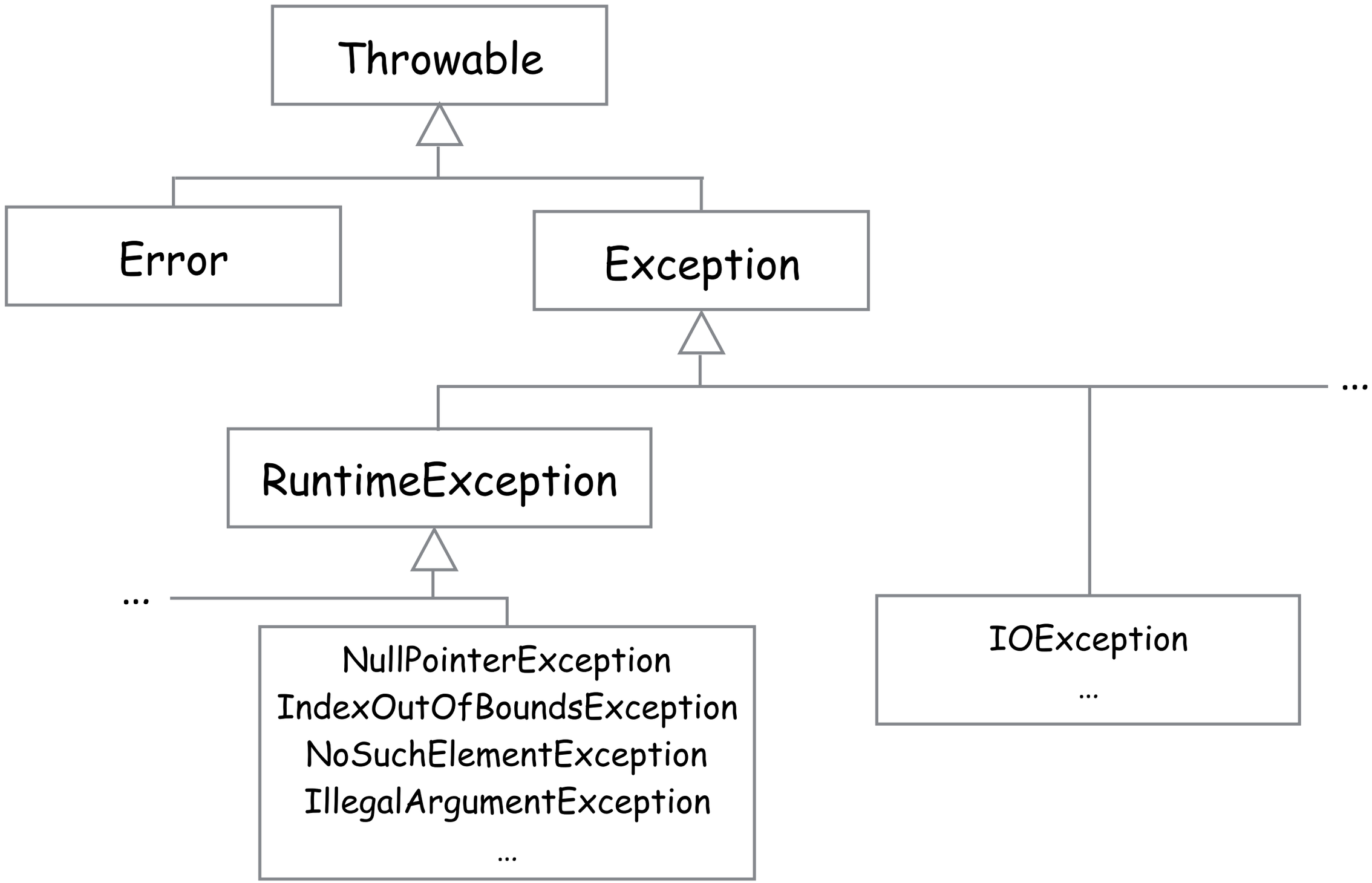
Afficher la trace de la pile

- Afin de debugger le programme le bloc `catch` peut afficher l'état de la pile d'exécution au moment de la capture de l'exception

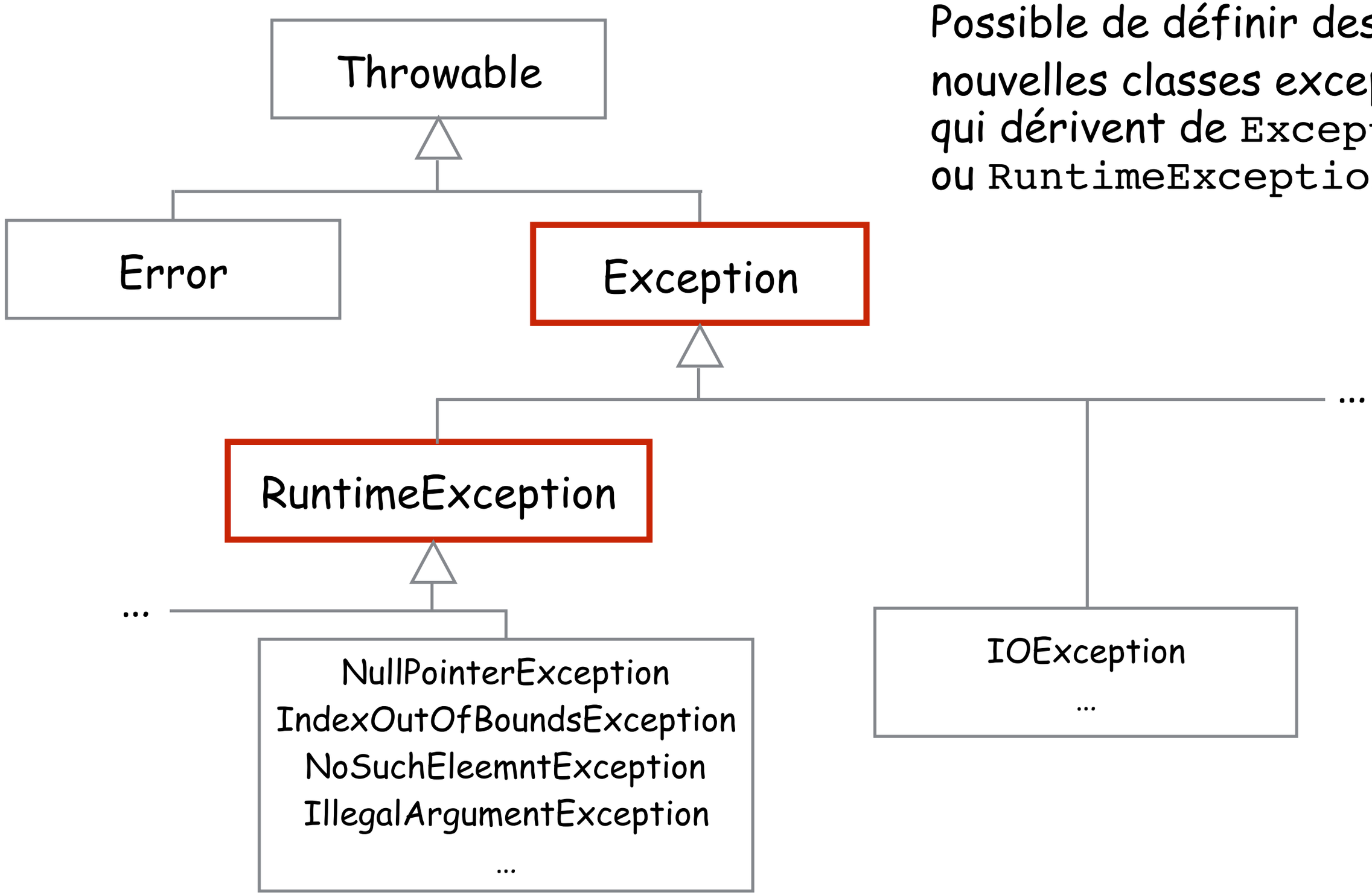
```
…
catch (ArrayIndexOutOfBoundsException e) {
 e.printStackTrace();
 System.out.println ("Résultat pas calculé");
}
…
```

```
java  ExceptionTest1  7
-> java.lang.ArrayIndexOutOfBoundsException: 7
        at ExceptionTest1.f(ExceptionTest1.java:19)
        at ExceptionTest1.main(ExceptionTest1.java:8)
-> Résultat pas calculé
-> Terminé
```

Hiérarchie de classes exception dans `java.lang`

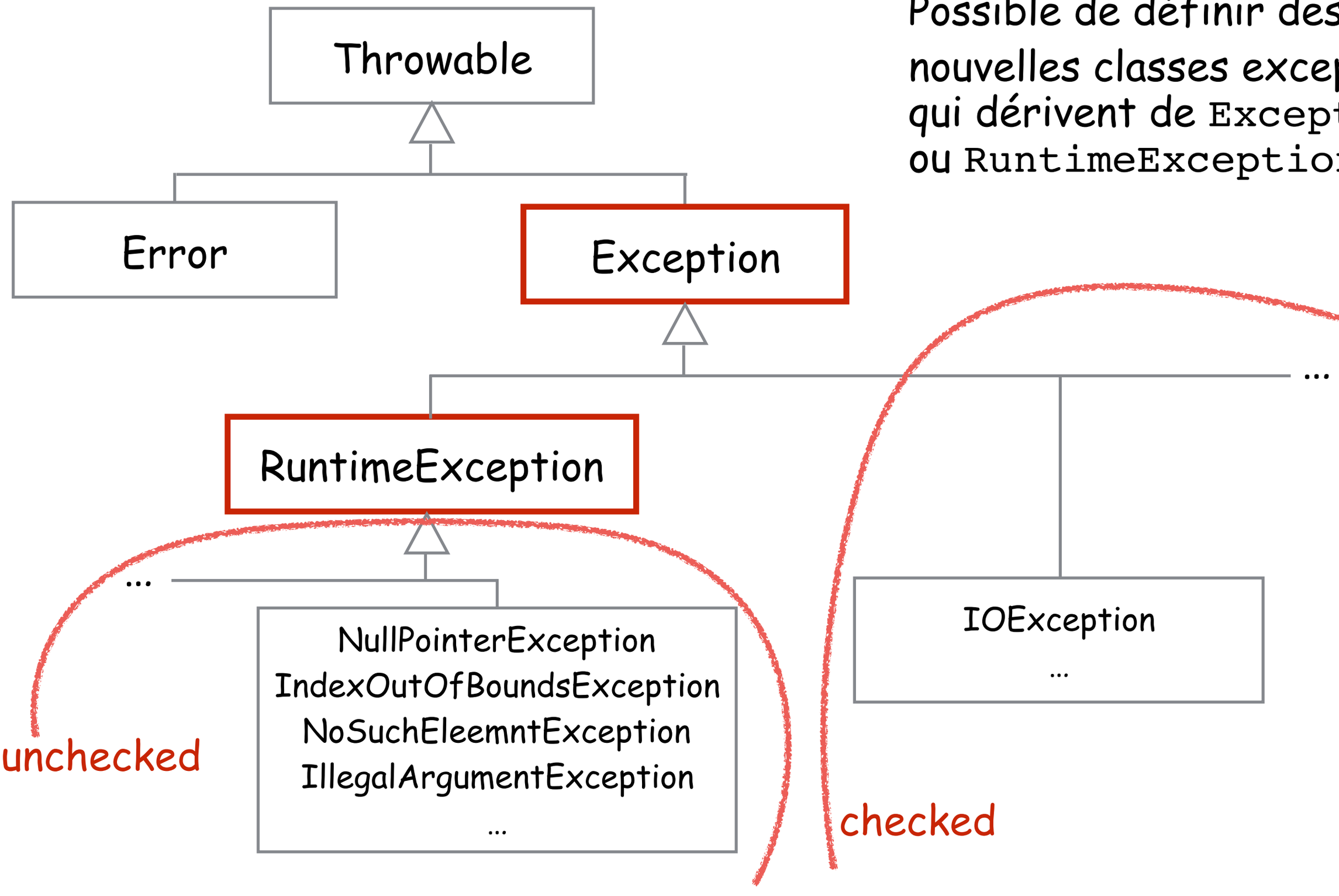


Hiérarchie de classes exception dans `java.lang`



Possible de définir des nouvelles classes exception qui dérivent de `Exception` ou `RuntimeException`

Hiérarchie de classes exception dans `java.lang`



Possible de définir des nouvelles classes exception qui dérivent de `Exception` ou `RuntimeException`

Exception "checked" et "unchecked"

- Checked : Classes qui dérivent de `Exception`, mais pas de `RuntimeException` :
  - erreurs qui dépendent de conditions externes (I/O, fichiers, réseaux,…)
- Unchecked : Classes qui dérivent de `RuntimeException`
  - Mauvais fonctionnement du programme (bugs) : situation d'erreur qui ne devrait jamais être atteinte (erreur du programmeur)
  - En général évitables avec des tests (i.e pointeur null, index supérieure à la dernière position d'un tableau, etc…)
  - Les exceptions de ce type sont souvent capturées uniquement pour garantir une sortie "gentille" du programme

Un exemple de définition d'exception unchecked

- But : définir une classe Rationnels
- Un rationnel comporte deux entiers : le numérateur et le dénominateur
- La tentative de création d'un rationnel avec dénominateur 0 soulève une exception
- On choisit de conserver l'information sur le numérateur dans l'exception

```
package poo.rationnels;
public class DenominateurNul extends RuntimeException{
 private int num;
 public DenominateurNul(int num) {this.num = num;}
 public DenominateurNul (int num, String msg)
   { super(msg); this.num = num;}
}
```

- On choisit d'étendre RuntimeException parce que quand le programme fonctionne correctement il ne devrait jamais créer un rationnel avec dénominateur 0

Un exemple de définition d'exception checked

- But : définir une classe qui représente un système d'équations linéaires
- Un système peut ne pas avoir de solutions
- Si c'est le cas on le détecte pendant la résolution et on soulève une exception
- Pour cela on définit :

```
package poo.systeme;

public class SystemeSansSolution extends Exception {
   public SystemeSansSolution() {};
   public SystemeSansSolution(String msg) { super(msg); }
}
```

- Il est courant de définir des classes exceptions vides : drapeaux

Contraintes sur les exceptions checked

- Toute méthode qui peut soulever une exception `checked` doit le déclarer :

```
public void fullRead(String filename)
                                throws FileNotFoundException {
  InputStream in = new FileInputStream(filename); int b;
  while ((b = in.read()) != -1){
    …
  }
}
```

(en effet, le constructeur `FileInputStream` peut soulever une exception de classe `FileNotFoundException` si le fichier n'existe pas)

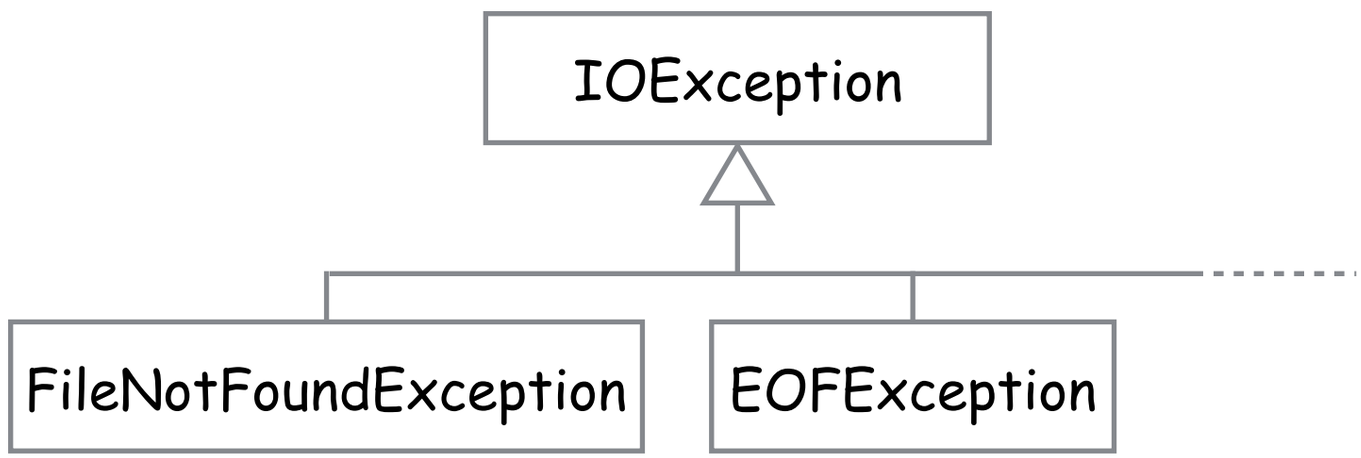
- Remarque : throws pas nécessaire pour les exceptions unchecked (toute instruction peut en soulever une…)

Contraintes sur les exceptions checked

- Remarque :

```
public void fullRead(String filename)
                                    throws IOException {
 InputStream in = new FileInputStream(filename); int b;
 while ((b = in.read()) != -1){
   …
 }
```

En effet `FileNotFoundException` potentiellement soulevée par `new FileInputStream(…)` étend `IOException`



Contraintes sur les exceptions checked

- Si la méthode capture l'exception elle ne peut plus la soulever donc pas de déclaration `throws`

```
public void fullRead(String filename) {
  InputStream in;
  try {
     in = new FileInputStream(filename);
  } catch (FileNotFoundException e) {
     e.printStackTace();
     return;
}
int b;
while ((b = in.read()) != -1){
     …
}
```

Lancer une exception

- Une exception peut être lancée automatiquement par une opération "primitive" (accès à un tableau, à un pointeur, à un fichier etc….)
- Ou par l'utilisation d'une méthode qui en soulève une
- Mais on peut également lancer une exception (checked ou unchecked) explicitement quand on détecte une situation d'erreur

```
public class Rationnel {
 private int num, den;
 public Rationnel(int n, int d) {
     if (d == 0) {
            throw new DenominateurNul();
     }
     …
 }
…
}
```

Lancer une exception : exemple

```
public class Syntax {
 public static void parse (String filename) throws IOException,
    EOFException {
   Scanner sc = new Scanner (Paths.get(filename), “UTF-8");
   String word = "";
   int count = 0; //compte les accolades ouvertes - fermées
   while (sc.hasNext()) {
       word = sc.next ();
       if (word.equals ( "{" )) count++;
       else if (word.equals( "}" )) count--;
       //sinon faire quelque chose avec le mot...
    }
   if (count != 0) throw new EOFException("Syntax error");
 }
 public static void main (String args[]) throws IOException {
   try {
       parse ("Syntax.java");
   } catch (EOFException e) {
       System.out.println (e.getMessage());
   }
}
}
```

Remarque : même effet

```
public class Syntax {
 public static void parse (String filename) throws IOException {
   Scanner sc = new Scanner (Paths.get(filename), “UTF-8");
   String word = "";
   int count = 0; //compte les accolades ouvertes
    while (sc.hasNext()) {
        word = sc.next ();
        if (word.equals ( "{" )) count++;
        else if (word.equals( "}" )) count--;
        //sinon faire quelque chose avec le mot...
    }
   if (count != 0) throw new EOFException("Syntax error");
 }
 public static void main (String args[]) throws IOException {
   try { parse ("Syntax.java");}
   catch (IOException e) {
        if (e.getClass() == EOFException.class)
           System.out.println (e.getMessage());
        else throw e;
   }
 }
}
```

Le bloc try - catch : détails

- On peut capturer plusieurs exceptions dans le même bloc

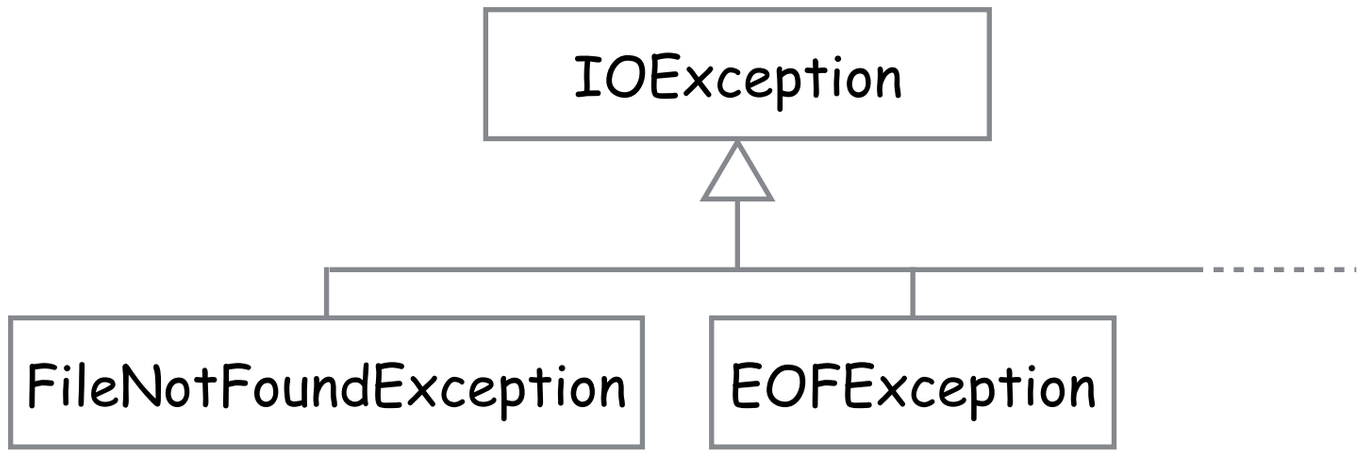
```
public static void main (String args[]) {
    try { parse ("Syntax.java");}
     catch (EOFException e) { System.out.println (e.getMessage());}
     catch (IOException e) {
         // gestion de tous les autres types de IOException
     }
     System.out.println (“parsing terminé”);
}
```

- bloc catch exécuté : le premier dont le type est compatible avec celui de l'exception soulevée dans le bloc try

Le bloc try - catch : détails

- Si le fichier n'est pas trouvé (FileNotFoundException)

```
public static void main (String args[]) {
   try { parse ("Syntax.java");}
    catch (EOFException e) { System.out.println
       (e.getMessage());}
    catch (IOException e) {
       // gestion de tous les autres types de IOException
    }
   System.out.println ("parsing terminé");
}
```



Le bloc try - catch : détails

- Si le fichier est trouvé mais il n'a pas les bonnes parenthèses (EOFException) :

```
public static void main (String args[]) {
    try { parse ("Syntax.java");}
    catch (EOFException e) { System.out.println
        (e.getMessage());}
    catch (IOException e) {
        // gestion de tous les autres types de IOException
    }
    System.out.println (“parsing terminé”);
}
```

Le bloc try - catch : détails

- Si le fichier est trouvé et il a les bonnes parenthèses (pas d'exceptions):

```
public static void main (String args[]) {
    try { parse ("Syntax.java");}
     catch (EOFException e) { System.out.println
        (e.getMessage());}
     catch (IOException e) {
        // gestion de tous les autres types de IOException
     }
    System.out.println ("parsing terminé");
}
```

Le bloc try - catch : détails

- Remarque : l'ordre inverse des blocs catch, n'est pas accepté par le compilateur :

```
public static void main (String args[]) {
    try { parse ("Syntax.java");}
     catch (IOException e) {
         // gestion de tous les autres types de IOException
    }
     catch (EOFException e) { System.out.println
        (e.getMessage());}
       System.out.println (“parsing terminé”);
}
```

- EOFException déjà capturée !

La clause `finally`

- Peut être ajoutée à un bloc try-catch
- contient du code qui sera exécuté immédiatement après le bloc try-catch, quelle que soit la façon d'en sortir.

- Façons possibles de sortir d'un bloc try-catch :
  - par l'exécution correcte du bloc try
  - par l'exécution d'un des blocs catch
  - par une exception soulevée par le bloc try ou par le bloc catch et pas capturée
  - par un return exécuté dans le bloc try ou dans le bloc catch

La clause `finally`: exemple

```
import java.io.*;
import java.nio.charset.*;

public class FinallyTest {
    static final int noEx = 1;
    static final int cc = 2;
    static final int fnf = 3;
    static final int eof = 4;
    static final int io = 5;
```

```
 public static int test (int i) throws IOException {
  if (i == fnf) throw new FileNotFoundException();
  else if (i == eof) throw new EOFException();
  else if ( i == cc) throw new  CharacterCodingException();
  else if (i == io) throw new IOException();
  else return 100;
 }
// continue…
```

La clause `finally`: exemple

```
public static void main (String args[]) throws IOException {
   int i = 0;
   try { i = test(Integer.valueOf(args[0]));}
   catch (CharacterCodingException e) {
       System.out.println(
                       "exception CharacterCoding capturée");
   }
   catch (FileNotFoundException e) {
       System.out.println("exception FileNotFound
                          capturée + exception Runtime lancée");
       throw new RuntimeException();
    }
    catch (EOFException e) {
       System.out.println("exception EOF capturée + return");
       return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
 }
} //class FinallyTest
```

La clause `finally`: exemple

`java FinallyTest 1` (noEx)

```
    try { i = test(Integer.valueOf(args[0]));}
    catch (CharacterCodingException e) {
        System.out.println(
                        "exception CharacterCoding capturée");
    }
    catch (FileNotFoundException e) {
        System.out.println("exception FileNotFound
                        capturée + exception Runtime lancée");
        throw new RuntimeException();
    }
    catch (EOFException e) {
        System.out.println("exception EOF capturée + return");
        return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
```

—> 100

—> suite du main…

La clause `finally`: exemple

`java FinallyTest 2` (cc)

```
    try { i = test(Integer.valueOf(args[0]));}
    catch (CharacterCodingException e) {
        System.out.println(
                        "exception CharacterCoding capturée");
    }
    catch (FileNotFoundException e) {
        System.out.println("exception FileNotFound
                        capturée + exception Runtime lancée");
        throw new RuntimeException();
    }
    catch (EOFException e) {
        System.out.println("exception EOF capturée + return");
        return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
```

—> exception CharacterCoding capturée

—> 0

—> suite du main…

La clause `finally`: exemple

`java FinallyTest 3` (fnf)

```
    try { i = test(Integer.valueOf(args[0]));}
    catch (CharacterCodingException e) {
        System.out.println(
                        "exception CharacterCoding capturée");
    }
    catch (FileNotFoundException e) {
        System.out.println("exception FileNotFound
                        capturée + exception Runtime lancée");
        throw new RuntimeException();
    }
    catch (EOFException e) {
        System.out.println("exception EOF capturée + return");
        return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
```

—> exception FileNotFound capturée + exception Runtime lancée

—> 0

Exception in thread "main"...

La clause `finally`: exemple

`java FinallyTest 4` (eof)

```
    try { i = test(Integer.valueOf(args[0]));}
    catch (CharacterCodingException e) {
        System.out.println(
                        "exception CharacterCoding capturée");
    }
    catch (FileNotFoundException e) {
        System.out.println("exception FileNotFound
                        capturée + exception Runtime lancée");
        throw new RuntimeException();
    }
    catch (EOFException e) {
        System.out.println("exception EOF capturée + return");
        return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
```

—> exception EOF capturée + return

—> 0

La clause `finally`: exemple

`java FinallyTest 5` (io)

```
    try { i = test(Integer.valueOf(args[0]));}
    catch (CharacterCodingException e) {
        System.out.println(
                        "exception CharacterCoding capturée");
    }
    catch (FileNotFoundException e) {
        System.out.println("exception FileNotFound
                        capturée + exception Runtime lancée");
        throw new RuntimeException();
    }
    catch (EOFException e) {
        System.out.println("exception EOF capturée + return");
        return;
    }
    finally  { System.out.println (i);}
    System.out.println("Suite du main...");
```

—> 0

Exception in thread “main”…

Travailler avec des ressources

- Quand on travaille avec des ressources qui doivent être fermées (e.g. un Scanner, une connexion à une base de données etc) l'idiome suivant est typique :

```
ouvrir la ressource
try {
   travailler avec la ressource
}
finally {
   fermer la ressource
}
```

- I.e. quelle que soit la façon de sortir du try (par exécution correcte ou par une exception) on veut fermer la ressource avant de continuer

Travailler avec des ressources

- Exemple

```
try {

    InputStream in = new FileInputStream("fichier.txt");
    try {
        //code qui peut lancer des exceptions
    }
    finally {
        in.close();
    }
}
catch (IOException e) {…}
```

try-with-resources

- Depuis Java 7 on a un raccourci pour le code :

```
TypeRessource r = new TypeRessource (…);
try {
   travailler avec la ressource r
}
finally {
   r.close()
}
```

- Il s'agit du bloc try-with-resources :

```
try (TypeRessource r = new TypeRessource (…)) {
   travailler avec la ressource
}
// r.close() appelé automatiquement
// quelle que soit la façon de sortir du bloc try
```

- Possible uniquement si `TypeRessource` étend l'interface `AutoClosable` (qui fournit la méthode `close()`)

try-with-resources

Exemple :

```
try (var in = new Scanner(
      new FileInputStream("fichier.txt"))) {

       while (in.hasNext())
              System.out.println(in.next());
}
//in.close() appelé quelle que soit la façon de sortir du
   bloc try
```

Remarque :

Des éventuelles exceptions générées par la création de la ressource (`var in =`…) doivent être capturées (ou déclarées)

Cela peut être fait par exemple en ajoutant un block `catch` (cf. slide suivant)

try-with-resources

Un bloc try-with-resources peut avoir un block catch, ainsi qu'un bloc finally

```
try (var in = new Scanner(
      new FileInputStream("fichier.txt"))) {

       while (in.hasNext())
              System.out.println(in.next());
}
catch (FileNotFoundException e){
    System.out.println("fichier non trouvé");
}
```

les deux, si présents, sont exécutés après la clôture de la ressource

try-with-resources

- Depuis Java 9 la ressource ne doit pas être crée nécessairement dans le bloc try-with-resources
- Elle peut être une variable existante, à condition qu'elle soit effective final

```
public static void scanAll(Scanner in) {
   try (in) { // effectively final variable in
       while (in.hasNext())
           System.out.println(in.next());
   }
}
```

Cas d'étude :

la résolution d'un système d'équations linéaires

Exemple :

$$x_1 - 3x_2 + 3x_3 = 5$$
$$2x_1 - 6x_2 + 2x_3 = 2$$
$$-x_1 + x_2 + x_3 = -3$$

Représentation matricielle

$$\begin{bmatrix} 1 & -3 & 3 \\ 2 & -6 & 2 \\ -1 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 5 \\ 2 \\ -3 \end{bmatrix}$$

a b

Pendant la résolution d'un tel système on peut se rendre compte qu'il n'admet pas exactement une solution : on gère cela avec une exception.

poo/systeme/Systeme.java
poo/systeme/SystemeSingulier.java

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

1	-3	3	5	
2	-6	2	2	⟵ -= 2*a[0]
-1	1	1	-3	

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{matrix} \boxed{1} & -3 & 3 & 5 \\ 0 & 0 & -4 & -8 \\ -1 & 1 & 1 & -3 \end{matrix}$$

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

1	-3	3	5	
0	0	-4	-8	
-1	1	1	-3	⟵ += a[0]

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{matrix} \boxed{1} & -3 & 3 & 5 \\ 0 & 0 & -4 & -8 \\ 0 & -2 & 4 & 2 \end{matrix}$$

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{matrix} 1 & -3 & 3 & 5 \\ 0 & \boxed{0} & -4 & -8 \\ 0 & -2 & 4 & 2 \end{matrix}$$

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{array}{cccc} 1 & -3 & 3 & 5 \\ 0 & \boxed{0} & -4 & -8 \\ 0 & -2 & 4 & 2 \end{array} \longleftarrow \text{échange de lignes}$$

si un tel échange n'était pas possible le système n'admettrait pas de solutions

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{matrix} 1 & -3 & 3 & 5 \\ 0 & \boxed{-2} & 4 & 2 \\ 0 & 0 & -4 & -8 \end{matrix}$$

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{array}{cccc} 1 & -3 & 3 & 5 \\ 0 & -2 & 4 & 2 \\ 0 & 0 & -4 & -8 \end{array}$$

forme triangulaire obtenue

$$\begin{aligned} x_1 - 3x_2 + 3x_3 &= 5 \\ -2x_2 + 4x_3 &= 2 \\ -4x_3 &= -8 \end{aligned}$$

système équivalent

Cas d'étude :

la résolution d'un système d'équations linéaires

- Solution d'un système d'équations de ce type : plusieurs méthodes
- La méthode d'élimination de Gauss
  - 1ere étape : triangulation du système

$$\begin{aligned} x_1 - 3x_2 + 3x_3 &= 5 \\ -2x_2 + 4x_3 &= 2 \\ -4x_3 &= -8 \end{aligned}$$

système équivalent

  - 2ème étape : résolution du système triangulaire (par substitution à partir de la dernière équation)

$$x_3 = -8 / -4 \Rightarrow x_3 = 2$$
$$-2x_2 + 4 * 2 = 2 \Rightarrow x_2 = 3$$
$$x_1 - 3 * 3 + 3 * 2 = 5 \Rightarrow x_1 = 8$$

Cas d'étude :
la résolution d'un système d'équations linéaires

- Un système qui n'admet pas de solutions

$$x_1 - x_2 = 0$$
$$x_1 - x_2 = 1$$

$$\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

poo/systeme/Gauss.java
poo/systeme/SEL.java

java SEL

Cas d'étude :

la résolution d'un système d'équations linéaires

- On peut avoir une sous-classe de Système pour chaque méthode de résolution

```
public Class Cramer extends Systeme{…}
//utilise la méthode de Cramer
```

```
public class GaussDiagonale extends Gauss {…}
// diagonalisation à la place de triangulation
// utilise la triangulation pour passer en forme
//diagonale
```