

Le mécanisme des exceptions en Java

HLIN505 – L3 – 2014

Qu'est-ce qu'une exception?
Comment définir et signaler des exceptions?
Comment récupérer des exceptions?

Qu'est-ce qu'une exception?

Un objet qui représente une *erreur à l'exécution*

due à

- une faute de saisie
- un problème matériel
- une faute de programmation

}

**Causes externes
au programme**

Bugs

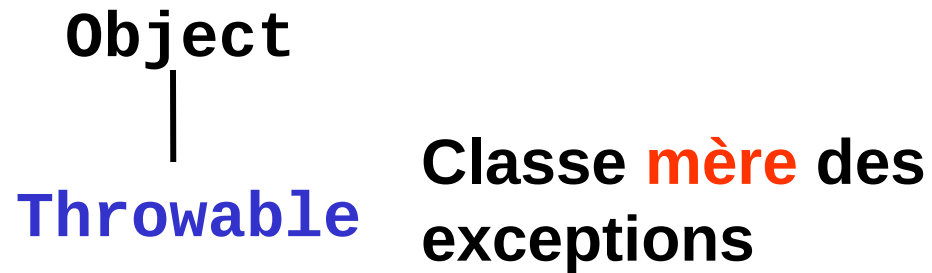
`T[i]` avec `i = T.length`

`ArrayIndexOutOfBoundsException`

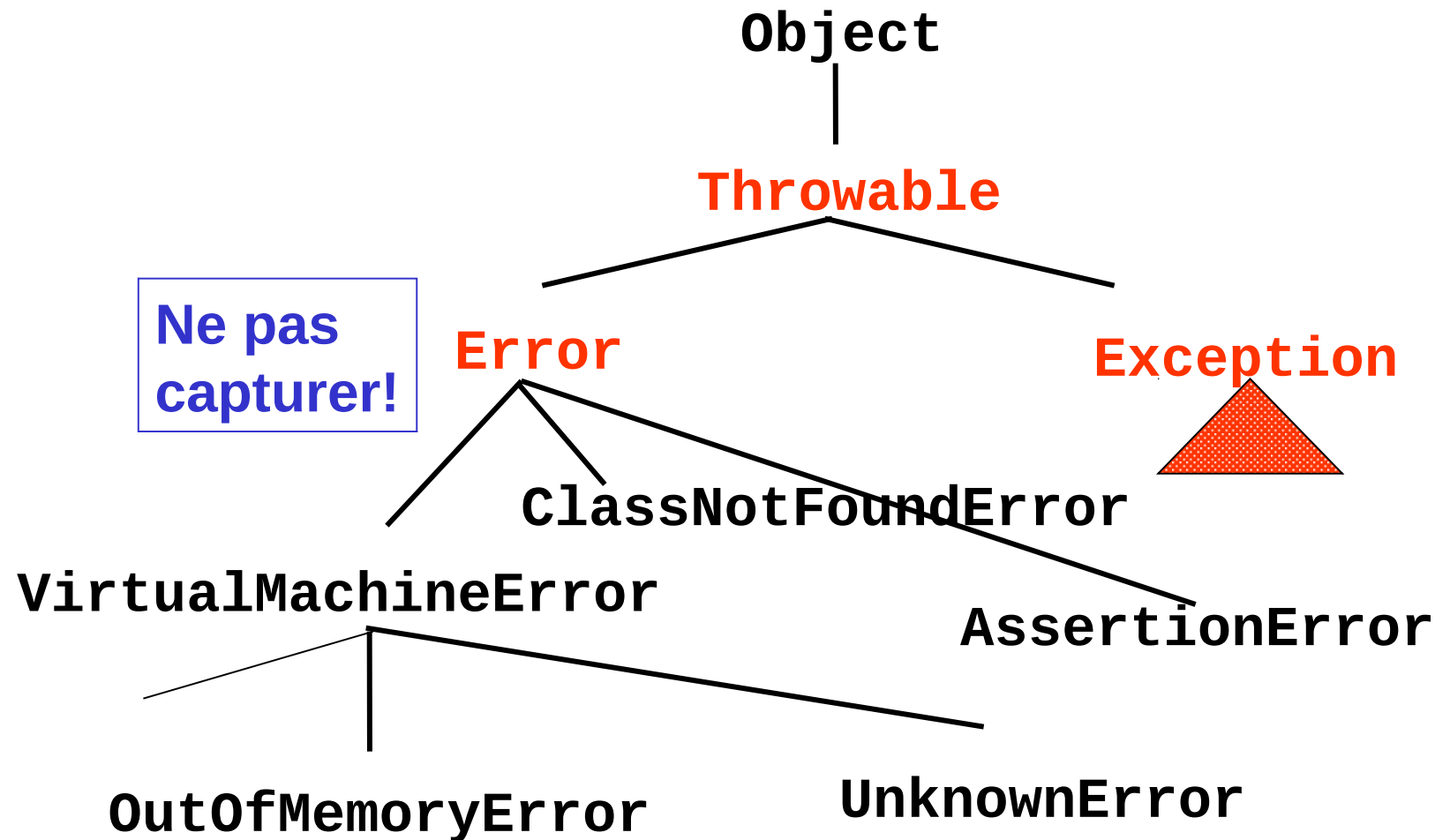
`o.f()` avec `o = null`

`NullPointerException`

Hiérarchie des exceptions en Java

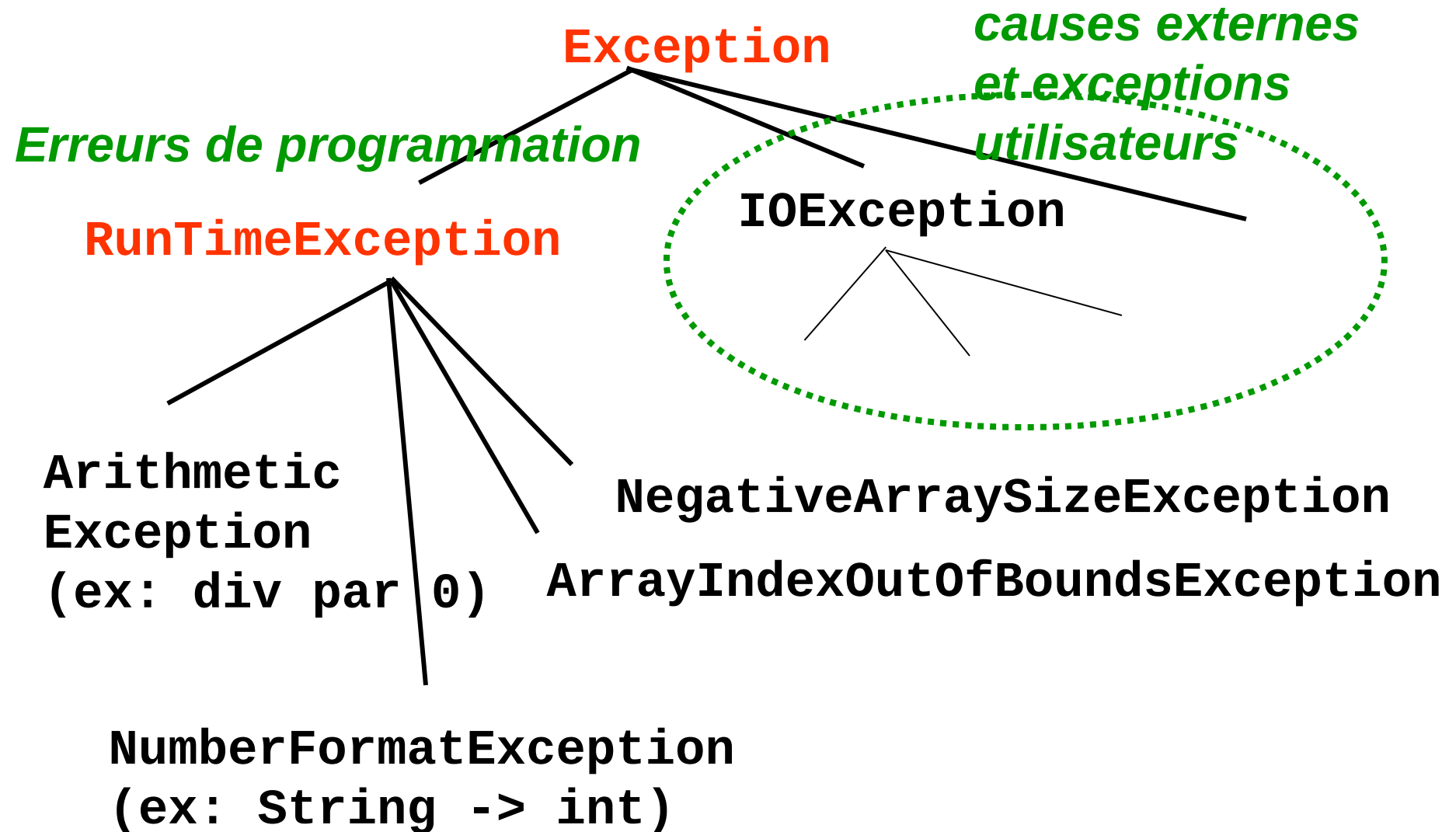


- **Attributs** :
message d'erreur (une String)
état de la pile des appels
- **Méthodes** :
public Throwable()
public Throwable(String message)
public String getMessage()
public void printStackTrace()



Error = problème de la machine virtuelle :

- Erreurs internes
- manque de ressources



Méthodes génératrices d'exceptions

- Toute méthode doit **déclarer** les exceptions qu'elle est susceptible de lancer/transmettre

classe java.io.BufferedReader

```
public final String readLine()  
                                throws IOException  
{.....}
```

- ... **sauf** si ce sont des **RuntimeException** ou **Error**

classe java.lang.Integer

```
public static int parseInt(String s)  
                    throws NumberFormatException  
                    Pas obligatoire  
{.....}
```

Philosophie générale

Exceptions hors contrôle

- les **Error** car leur traitement ne nous est pas accessible
- les **Runtime** car on n'aurait pas dû les laisser survenir

Exceptions sous contrôle


- Toutes les autres !

Méthodes génératrices d'exceptions

Signalement

Exemple

```
public final String readLine()  
                    throws IOException  
{  
    if (.....) throw new IOException();  
}
```



- crée un objet d'une certaine classe d'exception (ici IOException)
- signale (lève) cette exception : **throw**

Exceptions dans une classe utilisateur : Point

```
public class Point
{ private int x, y; //coordonnées

  public Point(int x, int y)
  {.....}
  public String toString()
  { return x + " " + y; }
  public void deplace (int dx, int dy)
  // ajoute dx et dy à x et y
  {.....}
} // fin classe
```

On ajoute la contrainte : un Point doit avoir des coordonnées **positives** ou **nulles**.

Comment assurer le respect de cette contrainte?

```
public Point(int x, int y)
{
    // si x < 0 ou y < 0, que faire?

    this.x = x;
    this.y = y;
}
```

```
public void deplace (int dx, int dy)
{
    // si (x + dx) < 0 ou (y + dy) < 0,
    // que faire?

    x += dx;
    y += dy;
}
```

Une classe d'exception pour Point

```
public class PointCoordException
    extends Exception
{
    public PointCoordException()
    { super() ; }

    public PointCoordException(String s)
    { super(s); }
}
```

On pourrait aussi créer une hiérarchie de classes d'exception pour Point

```
public Point(int x, int y)
{
    // si x < 0 ou y < 0,
    // générer une PointCoordException

    this.x = x;
    this.y = y;
}
```

```
public Point(int x, int y)
    throws PointCoordException
{
    if ((x < 0) || (y < 0))
        throw new PointCoordException
            ("création pt invalide "+ x + ' ' + y);

    this.x = x;
    this.y = y;
}
```

```
public void deplace (int dx, int dy)
{
    // si (x + dx) < 0 ou (y + dy) < 0,
    // générer une PointCoordException

    x += dx;
    y += dy;
}
```

```
public void deplace (int dx, int dy)
                                throws PointCoordException
{
    if ((x + dx < 0) || (y + dy < 0))
        throw new PointCoordException
            ("déplacement invalide "+dx+' '+dy);

    x += dx;
    y += dy;
}
```

Capture versus transmission d'exception

```
public static int lireEntier()  
{  
    BufferedReader clavier = new BufferedReader  
        (new InputStreamReader(System.in));  
  
    String s = clavier.readLine();  
    int ilu = Integer.parseInt(s);  
    return ilu;  
}
```

Erreur de compilation : lireEntier() doit *capturer* l'exception susceptible d'être transmise par readLine() *ou déclarer* qu'elle peut transmettre une exception (la laisser passer)

Solution 1 : la laisser passer

```
public static int lireEntier() *  
                                ajout à l'entête  
{.....}
```

* throws IOException

* throws IOException, NumberFormatException

* throws Exception *pas très informant !*

La clause throws doit "**englober**" tous les types d'exception à déclaration obligatoire susceptibles d'être transmis de la manière la plus spécifique possible

Solution 2 : la capturer (et la traiter)

- 1) surveiller l'exécution d'un bloc d'instructions : **try**
- 2) capturer *des* exceptions survenues dans ce bloc : **catch**

```
public static int lireEntier()  
{ BufferedReader clavier = .....;  
  try  
  { String s = clavier.readLine();  
    int ilu = Integer.parseInt(s);  
  }  
  
  catch (IOException e) {.....}  
  return ilu; /*  
  }  
  * Erreur de compilation : ilu  
  inconnu
```

```
public static int lireEntier()  
{ BufferedReader clavier = .....;  
  int ilu = 0;  
  
  try  
  { String s = clavier.readLine();  
    ilu = Integer.parseInt(s);  
  }  
  catch (IOException e) {}  
  // on capture e mais traitement = rien  
  
  return ilu;  
}
```

Que se passe-t-il si :

- une exception est générée par **readLine** ? (IOException)
- par **parseInt** ? (NumberFormatException)

```
public static int lireEntier()  
{ BufferedReader clavier = .....;  
  int ilu = 0;  
  
  try  
  { String s = clavier.readLine();    // 1  
    ilu = Integer.parseInt(s);    }    // 2  
  catch (IOException e) {}  
  // on capture e mais traitement = rien  
  
  return ilu;    // 3  
}
```

Si une exception est générée par **readLine** :

2 n'est pas exécuté ;

clause **catch** capture l'exception ;

l'exécution continue en **3** : *ilu retourné (avec valeur 0)*²⁰

```
public static int lireEntier()  
{ BufferedReader clavier = .....;  
  int ilu = 0;  
  
  try  
  { String s = clavier.readLine();    // 1  
    ilu = Integer.parseInt(s);    }    // 2  
  catch (IOException e) {}  
  // on capture e mais traitement = rien  
  
  return ilu;    // 3  
}
```

Si une exception est générée par **parseInt**
(NumberFormatException) :
aucune clause **catch** ne capture l'exception ;
elle est donc transmise à l'**appelant** ;
3 n'est pas exécuté

try + une (ou plusieurs) clause(s) **catch**

Si une exception est générée dans un bloc **try**,

- l'exécution **s'interrompt**
- les clauses **catch** sont examinées dans l'ordre, jusqu'à en trouver une qui "englobe" la classe de l'exception
- s'il en existe une : le bloc du catch est exécuté, et l'exécution **reprend** juste après les clauses catch
- sinon : l'exception n'est pas capturée ; elle est donc **transmise** à l'appelant, et le reste de la méthode n'est pas exécuté

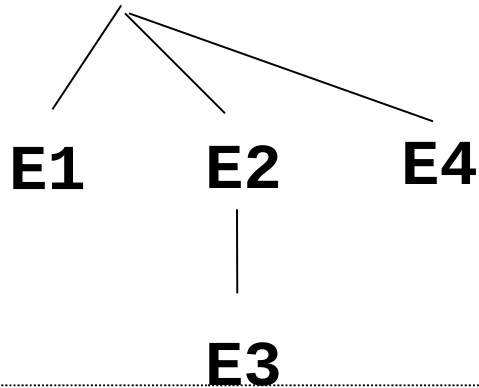
[On peut ajouter une clause **finally**

par laquelle on passe toujours]

Nom :

Prénom :

Exception



Quelles instructions sont exécutées si la partie (1) signale une exception de type :

-E1 :

-E2 :

-E3 :

-E4 :

méthode f(...) **throws E4**

{

try

{

(1) pouvant générer
des E1, E2, E3 et E4

}

catch(E1)

{ **(2)** }

catch(E2)

{ **(3)** }

finally

{ **(4)** }

(5)

}

Try avec ressource (depuis Java 7)

- Constat avant Java 7 :
 - Certaines ressources (fichiers, flux, etc) doivent être fermées explicitement pour les libérer
- Parade avant Java 7 :
 - utilisation du bloc finally pour fermer le flux même si une exception est levée
- Ce qui impliquait :
 - Déclaration de la ressource en dehors du bloc try
 - La méthode close() de la ressource peut lever une IOException à gérer (autre bloc try/catch ou propagation)

Try avec ressource (depuis Java 7)

- Depuis Java 7, définition possible d'une ressource avec l'instruction try
- La ressource sera automatiquement fermée à la fin de l'exécution du bloc try.

Try avec ressource (depuis Java 7)

```
try {  
    try (BufferedReader bufferedReader = new  
        BufferedReader(new FileReader("myFile.txt"))) {  
        String line=null;  
        while ((line = bufferedReader.readLine()) != null) {  
            System.out.println(line);  
        }  
    }  
} catch (IOException ioe) {  
    ioe.printStackTrace();  
}
```

bufferedReader sera
fermé proprement à la
fin normale ou anormale
des traitements → appel
automatique à close.

Try avec ressource (depuis Java 7)

- Ce qu'on peut mettre comme ressource dans le try :
 - Un `java.lang.AutoCloseable`
- `java.lang.AutoCloseable` définit une seule méthode : `close()` qui lève une exception de type `Exception`
- `java.io.Closable` extends `AutoCloseable`
- Le `close` de `Closable` lève une `IOException`

Try avec ressource (depuis Java 7)

- Il est possible de créer de nouveaux types de ressources autoclosable → implémenter `AutoClosable`
- La ressource doit être déclarée et définie dans le try
- Possibilité de spécifier plusieurs ressources : `try (Ressource a= ... ; Ressource b= ... ; Ressource c=...) →` fermeture automatique de c, puis b, puis a

Attraper plusieurs types d'exceptions (depuis Java 7)

```
try {  
    ...  
}  
catch(IOException | SQLException ex) {  
    ex.printStackTrace();  
}
```

Retour à lireEntier()

Essayons de trouver une bonne façon de gérer les erreurs

```
public static int lireEntier()  
{ BufferedReader clavier = .....;  
  int ilu = 0;  
  
  try  
  { String s = clavier.readLine();  
    ilu = Integer.parseInt(s);  
  }  
  catch (Exception e) {}  
  // on capture e mais traitement = RIEN  
  
  return ilu;  
}
```

Qu'en penser...?

Problème...

L'erreur n'est pas **vraiment** réparée:

si 0 est retourné,

l'appelant ne peut pas savoir que ça ne correspond pas *forcément* à une **valeur saisie**

**Si on ne sait pas comment traiter
une exception, il vaut mieux ne pas
l'intercepter**

**Trouvons un traitement plus
approprié ...**

```

public static int lireEntier()throws IOException
{
    BufferedReader clavier = .....;
    int ilu = 0;
    boolean succes = false ;
    while (! succes)
    {
        try
        {
            String s = clavier.readLine();
            ilu = Integer.parseInt(s);
            succes = true;
        }

        catch (NumberFormatException e)
        {
            System.out.println("Erreur : " + e.getMessage());
            System.out.println("Veuillez recommencer ... ");
        }
    } // end while

    return ilu;
}

```


lireEntier()

Changer de niveau d'abstraction

```
public static int lireEntier()  
  
throws IOException, MauvaisFormatEntierException  
{  
    BufferedReader clavier = .....;  
    int ilu = 0;  
  
    try  
    {  
        String s = clavier.readLine();  
        ilu = Integer.parseInt(s);  
    }  
    catch (NumberFormatException e) {  
        throw new MauvaisFormatEntierException();  
    }  
    return ilu;  
}
```

L'erreur de bas-niveau retournée est interceptée et transformée en erreur du niveau de lireEntier()

A ne pas faire

Remplacer un test par la génération d'une exception

tableau d'entiers **Tab** de taille **t**

Problème : calculer l'indice **i** du premier 0 de **Tab**
s'il existe (sinon **i** est affecté de -1)

NON !

```
int i = 0;
try
{ while (Tab[i] != 0) i++; }
catch(ArrayIndexOutOfBoundsException e)
{ i = -1; }
```

A ne pas faire

Cacher les exceptions pour éviter des erreurs de compilation

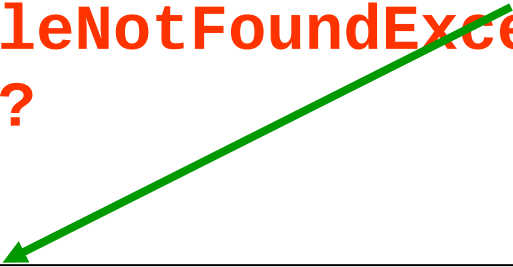
```
public void f()  
{  
    try  
    { ..... // ici le corps normal de f  
    }  
    catch (Exception e) {}  
}
```

NON !

A ne pas faire

Chercher à traiter des exceptions à tout prix

```
public void f (String nomFichier)
{
    On essaye d'ouvrir le fichier dont le
    nom est passé en paramètre
    Si une FileNotFoundException surgit,
    que faire?
}
```



A faire : transmettre l'exception à l'appelant, jusqu'à arriver à la méthode qui a *décidé* du nom de fichier

Exercice : soit une méthode qui utilise la classe Point

```
public Rectangle CreerRect(int x1, int y1,  
                           int x2, int y2)  
{  
    Point p1 = new Point(x1, y1);  
    Point p2 = new Point(x2, y2);  
    Rectangle r = new Rectangle(p1, p2);  
    return r;  
}
```

Quelle(s) attitude(s) cette méthode peut-elle adopter face aux PointCoordException susceptibles d'être générées?

Retour sur la conception

Comment déterminer les exceptions/assertions :

- invariants de classe
 - l'âge d'une personne est compris entre 0 et 140
 - une personne mariée est majeure
- préconditions
 - dépiler() seulement si pile non vide
- postcondition
 - après empiler(a), l'élément est dans la pile
- abstraction/encapsulation des exceptions des parties ou des éléments de l'implémentation
 - Point mal formé --> Rectangle mal formé
 - tableau interne de pile plein --> impossible d'empiler

Retour sur la conception

Assertions

Erreurs de logique du programme

Pour la mise au point du programme

Exceptions

Erreurs imprévisibles,

Hors du contrôle du programmeur

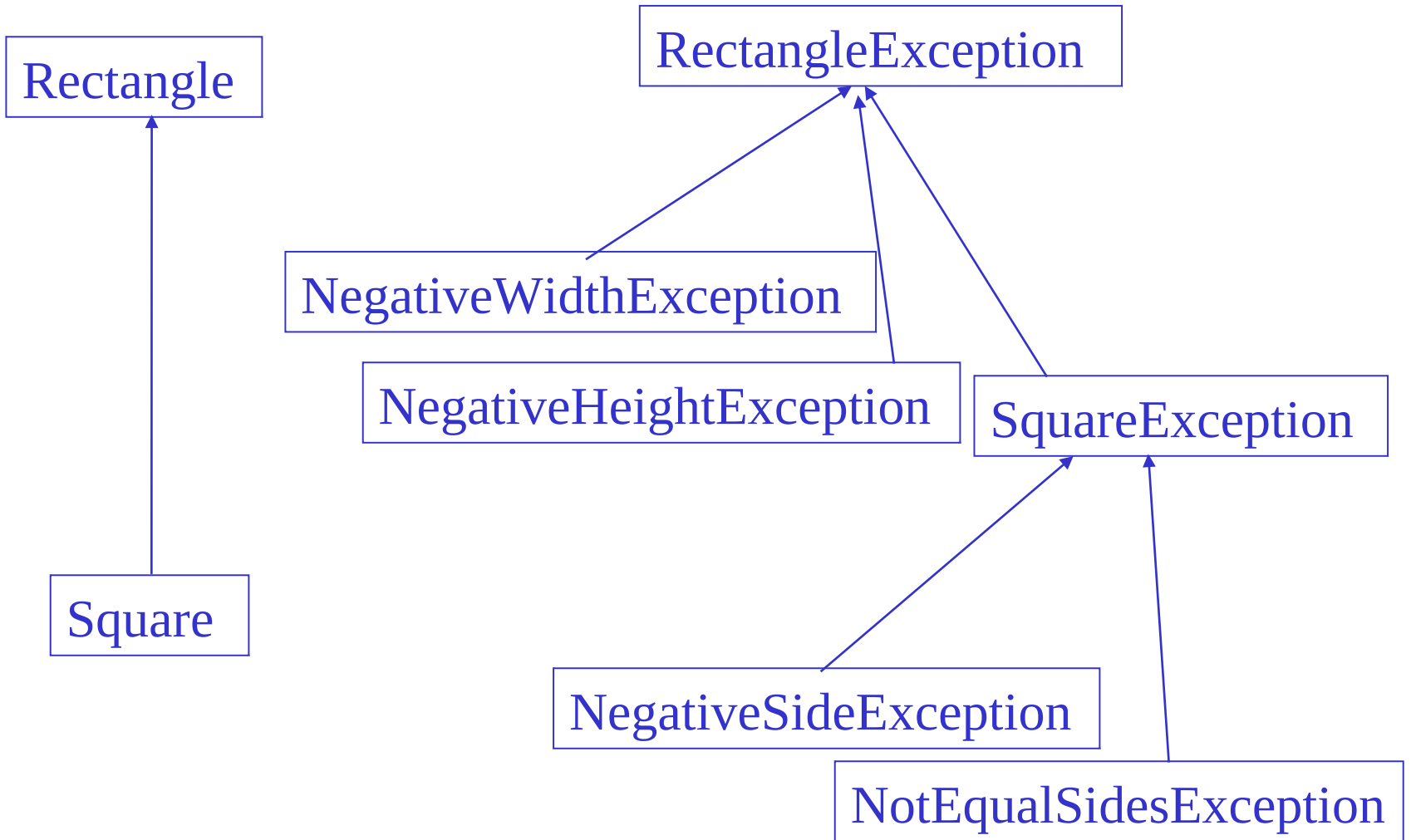
que l'on doit donc accepter

et gérer à l'exécution

Retour sur la conception

Comment organiser les exceptions :

- une racine pour les exceptions associées à une classe
 - `RectangleException`, `SquareException`
- sous la racine, exceptions pour
 - les invariants de classes
 - les pré- et post-conditions
- les exceptions des sous-classes s'organisent sous les exceptions de la classe



Redéfinition de méthodes

(règles assurant la substituabilité)

Dans la redéfinition:

- Ajouter une déclaration n'est pas possible
- Retirer une exception est possible
- Spécialiser une exception est possible

Dessiner() throws RectangleException, IOException

Peut être redéfinie en

Dessiner () throws NegativeWidthException