

Les exceptions

Maria Virginia Aponte

CNAM-Paris

26 février 2016

1. Échecs, erreurs, exceptions.

Sources d'échec dun sous-programme

- Un sous-programme peut requérir certaines pré-conditions sur ses entrées :
 - tableau non vide, objet non nul, etc.
- son interface peut être imprécise (comportement non spécifié) ;
- les appels externes peuvent échouer (ouvrir un fichier qui n'existe pas) ;
- ses ressources (ex : mémoire) épuisés ;

⇒ **Ce sont toutes des *circonstances exceptionnelles***

- **comment les traiter ?**

Manières de traiter l'échec

- Retourner une valeur "éronnée" (ou par défaut)
 - beaucoup de bibliothèques retournent null ;
 - lecture de fichiers retourne -1 si plus de données ;
 - problème : **l'appelant doit tester la valeur de retour**
⇒ utiliser avec précaution (difficile trouver bugs) ;
- Utiliser un résultat avec "informations"
 - un peu mieux qu'avant, mais
 - *responsabilité reste du côté de l'appelant* qui **doit** faire les bons tests ;
- Utiliser les exceptions
 - tout appelant peut les traiter,
 - si non traités, le programme termine avec un **échec**

Les exceptions

- Une exception est un *objet* qui représente des conditions anormales
 - son état interne peut décrire les données de l'échec,
 - exceptions prédéfinies (NullPointerException, IllegalArgumentException, IOException) ;
 - on peut définir ses propres exceptions.
- *Lever* une exception \Rightarrow *sortie d'urgence* de la méthode courante ;
 - l'exception se propage à travers la pile d'appels jusqu'à trouver un *traitement* ;
 - si non traitée, **tout** le programme échoue ;
- *Traiter* une exception \Rightarrow permet de prendre les mesures adéquates pour tenter de poursuivre l'exécution.

Un exemple realiste

```
void loadImage (String fileName) {
    try {
        Picture p = new Picture(fileName); // peut echouer
        // ... code qui dessine la figure
        // s'execute seulement si la figure a pu etre cree

    } catch (IOException ex) {
        // Utilise l'interface graphique pour montrer un
        // message d'erreur via une boite de dialogue.
        JOptionPane.showMessageDialog(
            frame, // boite de dialogue
            //
            "Cannot_load_file\n" + ex.getMessage(),
            "Alert", // titre boite
            JOptionPane.ERROR_MESSAGE // type de dialogue
        );
    }
}
```

Exemple simplifié : programme qui échoue

```
public static void main (String [] args){  
    Terminal.ecrireString("Entrez_un_nombre:_");  
    int x = Terminal.lireInt();  
    Terminal.ecrireStringln("Double_du_nombre:_"+ (x*2));  
}
```

Si la valeur entrée ne correspond pas à un nombre, quelle exécution ?

Exemple simplifié : programme qui échoue (2)

```
public static void main (String [] args){  
    Terminal.ecrireString("Entrez_un_nombre:_");  
    int x = Terminal.lireInt();  
    Terminal.ecrireStringln("Double_du_nombre:_"+ (x*2));  
}
```

```
> java Arret  
Entrez un nombre: 5f6  
Exception in thread "main" TerminalException  
    at Arret.main(Arret.java:4)
```


Exemple simplifié : programme qui échoue (3)

```
public static void main (String [] args){
    Terminal.ecrireString("Entrez_un_nombre:_");
    int x = Terminal.lireInt(); // <-- leve TerminalException
    Terminal.ecrireStringln("Double_du_nombre:_"+ (x*2));
}
```

La méthode `Terminal.lireInt()` lève l'exception `TerminalException`.

- (dans contexte courant = main) \Rightarrow code à exécuter au retour de p1
(..."fin p1...")
- contexte pour exécuter p1 \Rightarrow x = 3

2. Effets d'une levée d'exceptions

Exemple simplifié (méthodes statiques)

```
public static void p1(int x) {  
    p2(x+1);  
    System.out.println("fin_p1:_" + x);  
}  
public static void p2(int y) {  
    p3();  
    System.out.println("fin_p2:_" + y);  
}  
public static void p3() { throw new RuntimeException(); }  
  
public static void main (String [] args) {  
    p1(3);  
    System.out.println("fin_main_");  
}
```

Quelle exécution pour la méthode main ?

Utilisation de la pile d'exécution

Pour exécuter ce programme :

- 1 Empiler le contexte d'exécution du main (contient `args`),
- 2 puis il y a 3 appels imbriqués :
 - 1 premier appel $\Rightarrow p1(3)$,
 - 2 qui appelle `p2`,
 - 3 qui appelle `p3`,
- 3 empilement successifs de leurs 3 contextes d'exécution ;
- 4 + le code qui restera à exécuter à chaque retour ;

Appel imbriqué

Le code à exécuter au retour d'un appel imbriqué est sauvegardé dans la pile (avec le contexte courant), avant d'empiler le contexte « suivant ».

Utilisation de la pile d'exécution (2)

- 1 **avant** exécution `p1 (3)` empiler :
 - (dans contexte courant = `main`) \Rightarrow code à exécuter au retour de `p1` (`... "fin p1..."`)
 - contexte pour exécuter `p1` \Rightarrow `x = 3`
- 2 **avant** `p2 (x+1)` empiler :
 - (dans contexte courant = `p1`) \Rightarrow code à exécuter au retour de `p3` (`... "fin p2 ..."`)
 - contexte `p2` \Rightarrow `y=4`
- 3 **avant** exécution `p3 ()`, empiler contexte vide.

Au retour

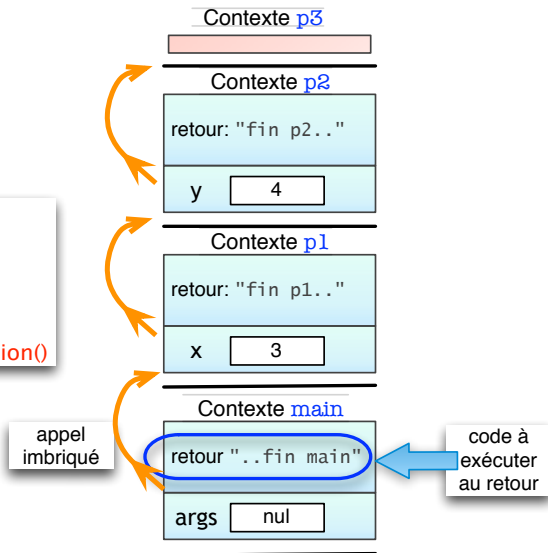
chaque méthode terminée **dépile** son contexte + **reprend** le code restant à exécuter empilée dans le contexte précédent.

Si tout va bien, après `p3`, on reprendra le code de `p2`, puis celui de `p1`.

Pile d'exécution : début appel p3

Code exécuté:

```
main →  
p1(3) →  
p2(4) →  
p3() →  
  throw  
  new RuntimeException()
```



Effet d'une levée d'exception

```
throw new RuntimeException();
```

Lève exception prédéfinie `RuntimeException`. Cela a pour effet :

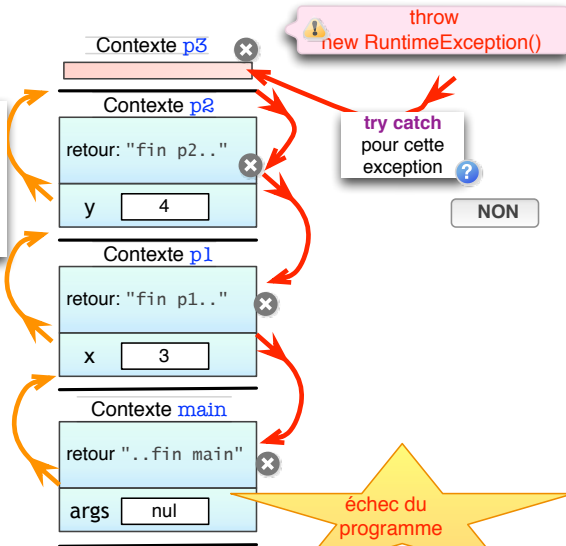
- *interruption exécution* méthode courante,
- recherche dans contexte courant `try ... catch` *entourant le code ayant levé l'exception*,
- si non trouvé, le *sous-programme actif échoue* ⇒ dépiler son contexte + instructions en attente *non exécutées* ;
- continuer recherche `try` dans la pile (du sommet vers le bas) ;
- si aucune instruction `try`, le *programme entier échoue*.

Pas d'instruction `try` dans notre exemple ! ⇒ Notre programme échoue.

Pile d'exécution : exception non rattrapée

Code exécuté:

```
main →  
p1(3) →  
p2(4) →  
p3() →  
  throw new  
  RuntimeException()
```



3. Traiter les exceptions

Traiter une exception

Idee : *entourer* du code pouvant lever une exception par une construction de *de rattrapage* qui prévoit instructions à exécuter, selon l'exception levée.

```
try { <code-pouvant-echouer>
    <code-suite-si-tout-va-bien>
} catch (UneException e) {
    <code-traitement>
}
<code-hors-try>
```

Si le code entouré par le try (*code-pouvant-échouer*) ne lève aucune exception, on continue *son exécution normale* avec :

- *<code-suite-si-tout-va-bien>*,
- puis avec *<code-hors-try>*.

Traiter une exception (2)

```
try { <code-pouvant-echouer>
    <code-suite-si-tout-va-bien>
} catch (UneException e) {
    <code-traitement>
}
<code-hors-try>
```

Si *<code-pouvant-echouer>* lève une exception :

- si elle est de type `UneException` :
 - exécuter *<code-traitement>*,
 - puis continuer la suite du programme : *<code-hors-try>*.

L'exception a été traitée ⇒ **le sous-programme se poursuit normalement.**

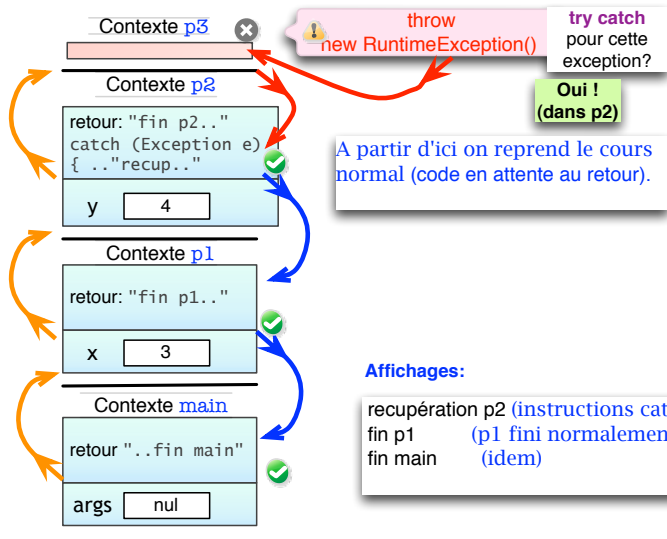
- si elle n'est pas de type `UneException`, **le programme en cours échoue.** On saute *<code-traitement>*, *<code-hors-try>*, on dépile et on continue à chercher un traitement en suivant la pile.

Traitement dans p2

```
public static void p1(int x) {
    p2(x+1); System.out.println("fin_p1:_" + x);
}
public static void p2(int y) {
    try {
        p3(); // <-- leve exception
        System.out.println("fin p2: " + y);
    } catch (RuntimeException e) {
        System.out.println("recuperation p2: ");
    }
}
public static void p3() { throw new RuntimeException(); }

public static void main (String [] args) {
    p1(3); System.out.println("fin_main_");
}
```

Poursuite exécution après rattrapage



Traitement dans main

Quels affichages pour cette version ?

```
public static void p1(int x) {
    p2(x+1); System.out.println("fin_p1:_" + x);
}
public static void p2(int y) {
    p3(); System.out.println("fin_p2:_" + y);
}
public static void p3() { throw new RuntimeException(); }

public static void main (String [] args) {
    try { p1(3); System.out.println("fin_main_");
    } catch (Exception e) {
        System.out.println("fin_avec_recuperation_main_");
    }
}
```

Autre exemple de levée

```
public class Arret {
    public static void main(String [] args) {
        Terminal.ecrireString("Un_entier?_");
        int x = Terminal.lireInt();
        Terminal.ecrireStringln("Coucou_1");
        if (x >0){
            throw new Stop();
        }
        Terminal.ecrireStringln("Coucou_2");
        Terminal.ecrireStringln("Coucou_3");
        Terminal.ecrireStringln("Coucou_4");
    }
}
class Stop extends RuntimeException {}
```

```
> java Arret
Un entier? 5
```

Exemple de levée (suite)

```
> java Arret
Un entier? 5
Coucou 1
Exception in thread "main" Stop
    at Arret.main(Arret.java:7)
```

Les lignes 9, 10, 11, n'ont pas été exécutées.

Le programme se termine en indiquant que l'exception `Stop` lancée dans `main`, ligne 7 du fichier `Arret.java` n'a pas été rattrapée.

Exemple de récupération

```
public class Arret2 {
    public static void P () {
        int x = Terminal.lireInt();
        if (x >0){ throw new Stop();
        }
    }
    public static void main(String [] args) {
        Terminal.ecrireStringln("Coucou_1");           // 1
        try { P ();
            Terminal.ecrireStringln("Coucou_2");       // 2
        } catch (Stop e){
            Terminal.ecrireStringln("Coucou_3");       // 3
        }
        Terminal.ecrireStringln("Coucou_4");           // 4
    }
}
class Stop extends RuntimeException {}
```

Exemple de récupération

```
public static void main(String [] args) {  
    Terminal.ecrireStringln("Coucou_1"); // 1  
    try {  
        P ();  
        Terminal.ecrireStringln("Coucou_2"); // 2  
    } catch (Stop e){  
        Terminal.ecrireStringln("Coucou_3"); // 3  
    }  
    Terminal.ecrireStringln("Coucou_4"); // 4  
}}  
  
class Stop extends RuntimeException {}
```

Affiche, si l'on saisit une valeur positive :

```
Coucou 1  
Coucou 3  
Coucou 4
```

L'instruction 2 n'est pas exécuté. Pourquoi ?

Exemple d'exécution "normale"

```
public static void main(String [] args) {  
    Terminal.ecrireStringln("Coucou_1"); // 1  
    try {  
        P ();  
        Terminal.ecrireStringln("Coucou_2"); // 2  
    } catch (Stop e){  
        Terminal.ecrireStringln("Coucou_3"); // 3  
    }  
    Terminal.ecrireStringln("Coucou_4"); // 4  
}}  
  
class Stop extends RuntimeException {}
```

Ce programme affiche, si l'on saisit une valeur négative :

```
Coucou 1  
Coucou 2  
Coucou 4
```

car l'exception n'est pas levée.

Exemple de non récupération

```
public class Arret3 {
    public static void P () {
        int x = Terminal.lireInt();
        if (x >0){ throw new Stop2();
        }
    }
    public static void main(String [] args) {
        Terminal.ecrireStringln("Coucou_1"); // 1
        try {
            P ();
            Terminal.ecrireStringln("Coucou_2"); // 2
        } catch (Stop e){
            Terminal.ecrireStringln("Coucou_3"); // 3
        }
        Terminal.ecrireStringln("Coucou_4"); // 4
    }
}
class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

Exemple de non récupération

```
public static void main(String [] args) {
    Terminal.ecrireStringln("Coucou_1");    // 1
    try { P ();
        Terminal.ecrireStringln("Coucou_2"); // 2
    } catch (Stop e){
        Terminal.ecrireStringln("Coucou_3"); // 3
    }
    Terminal.ecrireStringln("Coucou_4");    // 4
}
}
class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

Stop2 n'est pas récupérée. Si l'on saisit une valeur positive :

Coucou 1

```
Exception in thread "main" Stop2
    at Arret3.P(Arret3.java:7)
    at Arret3.main(Arret3.java:15)
```

Rattraper plusieurs exceptions

```
public static void P () {
    int x = Terminal.lireInt();
    if (x >0){ throw new Stop2();}
}

public static void main(String [] args) {
    Terminal.ecrireStringln("Coucou_1");           // 1
    try { P ();
        Terminal.ecrireStringln("Coucou_2"); // 2
    }catch (Stop e){Terminal.ecrireStringln("Coucou_3");//3
    }catch (Stop2 e){Terminal.ecrireStringln("Coucou_3bis");//4
    }
    Terminal.ecrireStringln("Coucou_4"); // 4
}

class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

3.1 Récupérer plusieurs exceptions

Rattraper plusieurs exceptions

```
public static void main(String [] args) {
    Terminal.ecrireStringln("Coucou_1");           // 1
    try { P ();
        Terminal.ecrireStringln("Coucou_2");     // 2
    } catch (Stop e){ Terminal.ecrireStringln("Coucou_3"); //3
    } catch (Stop2 e){Terminal.ecrireStringln("Coucou_3bis");/
    }
    Terminal.ecrireStringln("Coucou_4");        // 4
}
```

Après la saisie d'une valeur positive :

```
Coucou 1
Coucou 3 bis
Coucou 4
```


3.2 Exemple : récupérer les erreurs de saisie

Exemple : récupérer les erreurs de saisie

```
public static void main(String [] args) {
    int x; boolean ok=false;
    while (!ok) {
        Terminal.ecrireStringln("Entrez_un_entier");
        try {
            x = Terminal.lireInt();
            ok=true;
        } catch (TerminalException e){
            Terminal.ecrireStringln("Erreur_de_saisie._Recomenc");
        }
    }
    Terminal.ecrireStringln("Valeur_de_x=_"+x);
}
```

Comment s'exécute ce programme ?

Méthode de saisie avec récupération

```
static int saisieInt (String message) {
    int res; boolean ok=false;
    while (!ok) {
        Terminal.ecrireStringln(message);
        try { res = Terminal.lireInt();
            ok=true;
        } catch (TerminalException e){
            Terminal.ecrireStringln
                ("Erreur_de_saisie._Recomencez");
        }
    }
    return res;
}
```

La méthode renvoie l'entier saisi en résultat.

Méthode de saisie avec récupération (2)

```
public static void main(String [] args) {
    int taille = saisieInt("Taille_du_tableau?");
    int [] t = new int [taille];
    for (int i=0; i<t.length; i++){
        t[i] = saisieInt("Element_" + (i+1) + "?_");
    }
}
```

On peut employer la méthode plusieurs fois, par exemple, pour saisir de manière sécurisée, la taille d'un tableau et ensuite, les éléments du tableau.

Comment s'exécute ce programme ?

4. Définir ses exceptions

Définir ses propres exceptions

```
class NouvelleException extends ExceptionDejaDefinie {}
```

- Une exception est *un objet* : crée via une *classe*.
- mot clé **extends** ⇒ notion d'*héritage* (vue plus tard).
- `NouvelleException` : nom de l'exception que lon défini.
- `ExceptionDejaDefinie` est une exception définie auparavant.

Exemple

L'exception `Error` est prédéfinie en Java

```
class PasDefini extends Error {}
```

Exceptions prédéfinies en Java

Trois catégories :

- Dérivées de `Error` : erreurs critiques.
- Dérivées de `Exception` : erreurs à gérer “obligatoirement”.
- Dérivées de la classe `RuntimeException` : erreurs pouvant ou non être gérées.

Quelques exceptions prédéfinies

- `NullPointerException` : accès à un champ ou appel de méthode non statique sur un objet valant `null`. Utilisation de `length` ou accès à un case d'un tableau valant `null`.
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau, création d'un tableau de taille négative.
- `StringIndexOutOfBoundsException` : accès au i^{eme} caractère d'un chaîne de caractères de taille inférieure à i .
- `NumberFormatException` : erreur lors de la conversion d'un chaîne de caractères en nombre.

La classe `Terminal` utilise également l'exception `TerminalException` pour signaler des erreurs.

- Représentent des erreurs critiques : lorsque cela arrive, le programme ne peut plus continuer son exécution.
- Ne sont pas censées être gérées dans le programme.

Exemple : `OutOfMemoryError` est levée lorsqu'il n'y a plus de mémoire disponible dans le système.

Erreur critique : plus de mémoire, plus d'exécution possible.

Dérivées de `Exception`

- représentent les erreurs non critiques,
- qui doivent normalement être gérées par le programme.

Exemple : une exception de type `IOException` est levée en cas d'erreur lors d'une entrée sortie.

Le programme doit normalement prévoir un mécanisme de gestion : message d'erreur, nouvelle saisie, etc., mais **pas s'interrompre** définitivement.

Dérivées de `RuntimeException`

- elles représentent des erreurs pouvant ou non être gérées par le programme.
- Exemple typique : `NullPointerException`, est levée si l'on tente d'accéder au contenu d'un tableau ou d'un objet qui vaut `null`.

5. Déclarer une méthode qui lève des exceptions.

Déclaration `throws`

- Si une méthode **se termine anormalement** en *propageant* une exception dérivée de `Exception`,
- on doit préciser dans son entête qu'elle est *susceptible d'échouer* avec une telle exception.

`throws` (entête de méthode)

```
int m(...) throws Exc1, Exc2, ...{  
    ...  
}
```

On déclare `Exc1, Exc2, ...` comme pouvant **faire échouer** la méthode `m`.

Déclaration **obligatoire uniquement** pour la catégorie `Exception`.

Déclaration throws

```
static int factorielle(int n) throws PasDefini {
    int res = 1;
    if (n<0){ throw new PasDefini(); }
    for(int i = 1; i <= n; i++) {
        res = res * i;
    } return res;
}

public static void main (String [] args) {
    int x; Terminal.ecrireString("Un_nombre?_");
    x = Terminal.lireInt();
    try { Terminal.ecrireIntln(factorielle(x));
    } catch (PasDefini e) {
        Terminal.ecrireStringln("La_factorielle_de_"
                                +x+"_n'est_pas_définie!"); }
}

class PasDefini extends Exception {}
```

Comment s'exécute ce programme selon la valeur saisie pour x ?

Déclaration throws

```
public static void main (String [] args) {
    int x;
    Terminal.ecrireString("Entrez_un_nombre_(petit):");
    x = Terminal.lireInt();
    try { Terminal.ecrireIntln(factorielle(x));
    } catch (PasDefini e) {
        Terminal.ecrireStringln("La_factorielle_de_"
                                +x+"_n'est_pas_définie!"); }
}
```

Comment s'exécute ce programme selon la valeur saisie pour x ?

```
> java Factorielle
Entrez un nombre (petit):4
24
> java Factorielle
Entrez un nombre (petit):-3
La factorielle de -3 n'est pas définie !
```

6. Les exception sont des objets !

Informations récupérées dans un objet « exception »

- Une exception est un **objet**,
- la variable **e** récupérée dans un `catch` est un « objet exception » *créé et lancé* « quelque part ailleurs » par un `throw new CalculErr(..)` :

```
try {  
    System.out.println(factorielle(x));  
  
} catch (CalculErr e){  
    e.afficheErreur();  
}
```

On invoque la méthode `afficheErr()` interne à l'objet `e`.

Dans nos propres exceptions, nous pouvons inclure :

- 1 variables d'instance, constructeurs, méthodes ;
- 2 puis, transmettre des informations dans cet objet lors de sa creation/lancement ;
- 3 et quelque part ailleurs dans le programme, utiliser ces informations lors du rattrapage de ces mêmes objets.

Informations + méthodes d'un objet « exception »

```
public class CalculErr extends Exception
    private int val;
    private String nomOp;
    private String diagnostique;

    public CalculErr(int v, String o, String d){
        this.val=v; this.op=o; this.diagnostique= d;
    }
    public afficheErreur(){
        System.out.println("Calcul_de_" + op + "_impossible_:_" +
            val + "_est_" + diagnostique);
    }
}
```

```
public static int factorielle (int n) throws CalculErr {  
    int res = 1;  
    if (n<0)  
        throw new CalculErr(n,"factorielle","negatif");  
    if (n>10000)  
        throw new CalculErr(n,"factorielle","trop_grand");  
    for(int i = 1; i <= n; i++) {  
        res = res * i;  
    } return res;  
}
```

Exemple d'exécution

```
try {  
    System.out.println(factorielle(x));  
  
} catch (CalculErr e){  
    e.afficheErreur();  
}
```

Comment s'exécute ce programme selon la valeur de x ?

- si $x=-5 \Rightarrow$

Calcul de factorielle impossible: -5 est negatif

- si $x= 30\ 000 \Rightarrow$

Calcul de factorielle impossible: 30000 est trop grand

Exceptions : règles importantes

Le mécanisme d'exécution et le raisonnement sur les programmes avec exceptions est compliqué :

- N'utiliser les exceptions qu'en des circonstances exceptionnelles et seulement si les solutions avec tests ne sont pas viables ;
- Ne jamais lever et rattraper une exception dans la **même** méthode : on aurait pu faire autrement avec un test ! inexistante dans un tableau, création d'un tableau de taille négative.
- Définir ses propres exceptions lorsque cela facilite la lecture du programme ;
- Favoriser les solutions simples et homogènes.