

# Les exceptions

Maria Virginia Aponte

CNAM-Paris

10 janvier 2019

# 1. Échecs, erreurs et exceptions.

# Erreurs en cours d'exécution

L'exécution d'un programme peut requérir des actions « impossibles à exécuter » au moment où elles sont invoquées. Elles vont provoquer des **erreurs à l'exécution**.

Exemples :

- modifier la case 5 d'un tableau qui n'en contient que 3,
- ouvrir en lecture un fichier qui n'existe pas,
- créer un nouveau tableau alors que le quota mémoire est épuisé,
- lire un entier au clavier alors que l'on a tapé un texte dont la syntaxe n'est pas celle d'un entier.

# Exceptions en Java

Afin de traiter les erreurs à l'exécution, Java leur associe des objets nommées **exceptions**.

Divers types d'exceptions correspondent à diverses sortes d'erreur.

Beaucoup d'exceptions sont prédéfinies :

- `ArrayIndexOutOfBoundsException`
- `OutOfMemoryException`, `RuntimeException`
- `FileNotFoundException`

On peut définir ses propres exceptions :

- `TerminalException` définie par nous, est lancée par les méthodes de `Terminal`.

# Exemple 1 (TerminalException)

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

Quelle exécution si l'on saisit « 5f6 » ?

```
> java Arret1  
Entrez un nombre: 5f6  
Exception in thread "main" TerminalException  
    at Arret.main(Arret.java:4)
```

## Exemple 1 (2)

---

```
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` :

- cette méthode revient (ligne 3) en mode échec ;
- les intructions suivantes sont « abandonnées » (la 3ème instruction)
- le programme entier échoue.

# Échec d'un sous-programme P

Suite à une erreur pendant l'exécution de P, arrêt de P avant terme. Peut mener (transitivement) à la propagation de l'échec au programme entier.

## Échec du sous-programme P

Suite à une erreur d'exécution, interruption du « cours séquentiel » d'exécution des instructions de P :

- 1 interruption de l'instruction en cours d'exécution au moment de l'erreur,
- 2 si aucun rattrapage-traitement de l'erreur n'est prévu :
  - abandon des instructions de P restant à exécuter ⇒ P échoue
  - propagation de l'erreur vers le sous-programme appelant.

# A quoi servent les exceptions ?

Les erreurs à l'exécution sont inévitables. Grosso modo, les exceptions servent à :

- 1 **les identifier** (quelle type d'erreur, dans quelles circonstances ?)
- 2 **les rattrapper-traiter** pour reprendre le contrôle de l'exécution (on la rattrape, on répare, on continue)
- 3 **les déclencher** (je devrais retourner un résultat, mais dans ce cas précis, il est impossible à calculer : je lance une erreur à la place)

**But : améliorer la robustesse**

*In fine*, on souhaite écrire des programmes plus robustes : moins susceptibles de planter face aux cas « non conventionnels ».



## Exemple d'utilisation avec `Terminal.lireInt()`

`Terminal.lireInt()` lit un texte et retourne l'entier correspondant. Sauf si ce texte n'a pas la syntaxe d'un entier, auquel cas la méthode lance `TerminalException`

2 points de vue : celui d'un utilisateur de la méthode ; celui de son codeur.

- **identification** : la lecture d'un entier syntaxiquement invalide est identifié par un type d'exception (déclaré par nous) : `TerminalException`
- **rattrapage-traitement** : un appel à `Terminal.lireInt()` peut lancer et propager `TerminalException` jusqu'à dans mon programme. Si je veux éviter son arrêt total, je dois rattrapper et traiter cet erreur (et pas une autre, donc l'identification précise est importante).
- **déclenchement** : le codeur de `Terminal.lireInt()` doit retourner l'entier résultat de la conversion du texte lu. Or, si sa syntaxe est invalide, on ne peut pas « inventer » un entier. Seule issue : lancer une exception permettant d'identifier précisément l'erreur là où elle sera propagée.

# Les exceptions en Java

Une exception est un *objet* qui possède un *type (ou nom d'exception)*. Ce type ou nom d'exception représente une erreur spécifique ou une famille d'erreurs à l'exécution.

- nombreuses exceptions prédéfinies
- on peut définir ses propres exceptions
- on peut déclencher une erreur à l'exécution

Exemple :

- `TerminalException`, `RuntimeException` sont des noms ou types d'exceptions.
- `new TerminalException()` est une exception (objet de type exception).
- `throw new TerminalException()` : création d'un objet exception (new), puis lancement de celui-ci (throw).

# Ce que l'on peut faire avec les exceptions

- 1 *Déclarer un nouveau type* d'exception
- 2 *Créer un objet exception* (via new)
- 3 *Lever* ou lancer un objet exception
  - correspond au déclenchement d'une erreur d'exécution  $\Rightarrow$  arrêt de l'instruction en cours d'exécution ;
  - l'objet lancé doit être créé au préalable,
  - l'exception se propage à travers la pile d'appels jusqu'à trouver du code de *traitement* ;
  - si non traitée, **tout** le programme échoue ;
- 4 *Traiter* une exception  $\Rightarrow$  permet de stopper la propagation de l'échec et d'exécuter des actions « correctives » puis de reprendre l'exécution.

# Un exemple realiste

---

```
void loadImage (String fileName) {
    try {
        Picture p = new Picture(fileName); // peut echouer
        // si le fichier 'filename' n'existe pas
        .... // instructions pour dessiner la figure

    } catch (IOException ex) {
        // montre un message d'erreur (boite de dialogue)
        JOptionPane.showMessageDialog(
            frame, // boite de dialogue
                //
            "Cannot_load_file\n" + ex.getMessage(),
            "Alert", // titre boite
            JOptionPane.ERROR_MESSAGE // type de dialogue
        );
    }
}
```

---

## 2. Lever, déclencher (throw)

# Comment lever (déclencher) une exception ?

On doit :

- 1 *Créer un objet* (via `new`) du type exception pertinent pour le cas d'erreur
- 2 *Lever* ou lancer cet objet (via `throw`), si les conditions de l'erreur sont réunies (suite à un test, donc)

---

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

---

# Quand lever (déclencher) une exception ?

Si une méthode (fonction) ne peut retourner un résultat correct, on peut lever à la place, une exception correspondant à l'erreur.

`plusGrandTabInt` doit retourner l'élément le plus grand du tableau.

- cet élément n'existe pas si `t` est nul ou s'il ne contient aucune case (possible en Java).

---

```
public static int plusGrandTabInt(int [] t) {
    if (t==null || t.length==0)
        throw new IllegalArgumentException();
    int max=t[0];
    for (int i=0; i<t.length; i++) {
        if (t[i]>max)
            max=t[i];
    }
    return max;
}
```

## Quand lever une exception ? (2)

---

```
public static void main(String[] args) {  
    int[] t = new int [0]; // tableau avec 0 cases  
    System.out.println(plusGrandTabInt(t));  
}
```

---

Exception in thread "main"

```
java.lang.IllegalArgumentException  
at demoChapExceptions.ExempleLevee1.plusGrandTabInt(ExempleLevee1.java:10)  
at demoChapExceptions.ExempleLevee1.main(ExempleLevee1.java:5)
```



## 2.1 Propagation 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

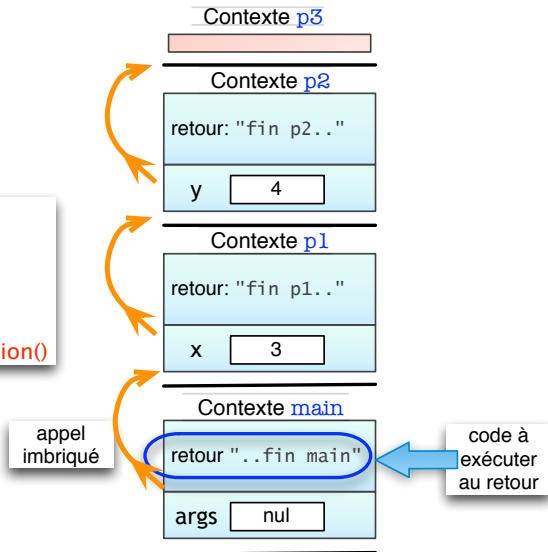
chaque méthode terminée **dépile** son contexte + **reprend** le code restant à exécuter empilée dans le contexte précédent.

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()
```



# Propagation d'une levée d'exception

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

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

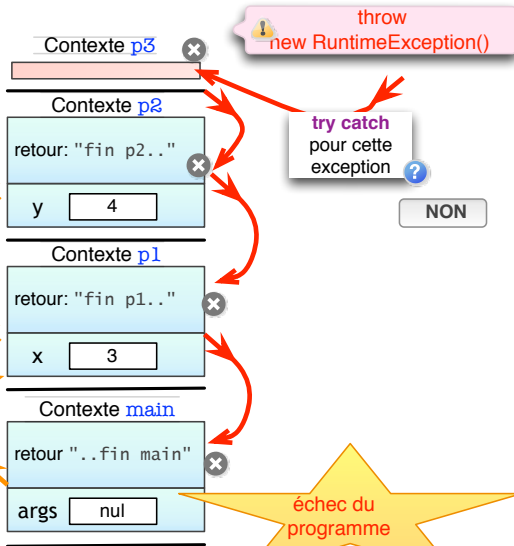
- *interruption exécution* méthode courante,
- recherche dans contexte courant `try ... catch` *entourant le code ayant levé l'exception*,
- si non trouvé, le *sous-programme actif échoue* ⇒ dépiler son contexte + instructions en attente *non exécutées* ;
- continuer recherche `try` dans la pile (du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

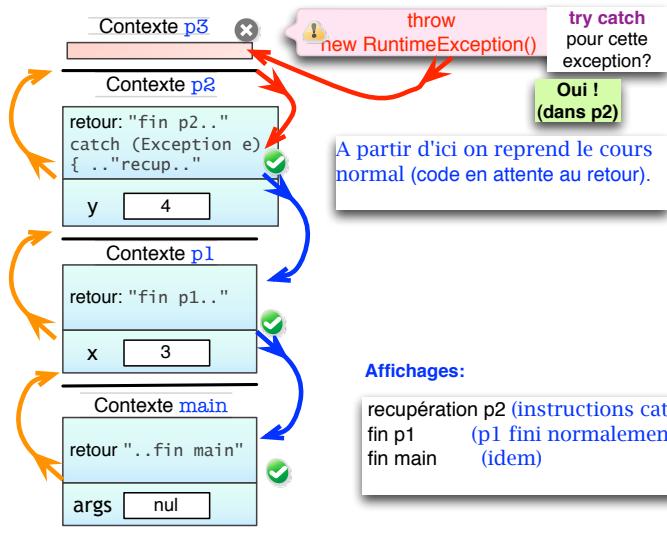
---

```
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.afficheStringln("Entrez_un_entier");
        try {
            x = Terminal.lireInt();
            ok=true;
        } catch (TerminalException e){
            Terminal.afficheStringln("Erreur_de_saisie._Recomenc");
        }
    }
    Terminal.afficheStringln("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 ?

# 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.