M2103 (POO) Bases de la programmation orientée objet

Responsable : Cristina Onete

cristina.onete@gmail.com

https://www.onete.net/teaching.html

Les interfaces

Les interfaces en Java

- L'heritage "factorise" la similarité physique de divers types d'objets
  - Un loup, un humain, un ours sont tous des mammifères
- Une interface "factorise" la similarité fonctionnelle
  - On **conduit** des véhicules de tout type
  - On **joue** aux instruments, mais aussi aux jeux
  - On **achète** des divers types d'objets
- Une interface en Java : un ensemble de caractéristiques liées décrites par des méthodes abstraites
- Attention : **une interface n'est pas une classe**!
  - Une interface est implémentée par des classes, qui concrétisent ses méthodes

Exemple

- On part sur une classe **Personne**
  - Une personne aura un nom, un prenom, un budget

- On aura également les classes : **Burger**, **Instrument**, **Piano**, **JeuVideo**
  - Tous ces objets sont ***achetables***
  - Les jeux video sont ***achetables*** et ***jouables***
  - Les instruments sont ***achetables***, et ***jouables***
  - Un piano est un instrument qui de plus est ***accordable***

- Autrement dit, on classifie ces objets en termes de comment une personne peut interagir avec eux
  - Une personne achète des achetables (de tout type)
  - Une personne joue à un jouable (de tout type)

Nos interfaces

```
interface Achetable {
    double getPrix();
}
```

```
interface Accordable {
    void accorder();
}
```

```
interface Jouable {
    void jouer(Personne[] joueurs);
}
```

mot dédié : interface

méthodes abstraites

même sans mention public, toute méthode est publique

Implémenter une interface

- Un burger est achetable

Mot dédié : implements

Burger est une classe **concrète**
Elle doit implémenter la méthode
double getPrix() de l'interface
Achetable

```
public class Burger implements Achetable{
	private String type;
	private double prix;

	public Pizza(String nom, double prix){
		this.nom = nom;
		this.prix = prix;
	}

	public double getPrix(){
		return this.prix;
	}
}
```

Implémenter deux interfaces

- Un jeu video est achetable et jouable

Virgule entre les deux interfaces

Obligatoire : JeuVideo implémente Achetable

Obligatoire : JeuVideo implémente Jouable

```
public class JeuVideo implements Achetable, Jouable{
	private String nom;
	private double prix;

	public JeuVideo(String nom, double prix){
		this.nom = nom;
		this.prix = prix;

	public double getPrix(){
		return this.prix;
	}

	public double jouer(Personne[] joueurs){
		// code de la méthode
		//...
	}
}
```

Classe Instrument (abstraite)

- Classe abstraite
- Peut implémenter des interfaces
- Doit obligatoirement contenir les méthodes des interfaces
  - Mais ce n'est pas obligatoire de les détailler
  - Toute méthode implémentée est soit abstraite soit détaillée

```
public abstract class Instrument implements Achetable, Jouable{
	protected String nom;
	protected String marque;
	protected double prix;

	public Instrument(String nom, String marque, double prix){
		this.nom = nom;
		this.marque = marque;
		this.prix = prix;
	}

	public double getPrix(){
		return this.prix;
	}

	public abstract double jouer(Personne[] joueurs);
}
```

Héritage et interfaces

- Les pianos sont des instruments qui sont de plus accordables

extends, puis **implements**

Classe concrète
Doit détailler cette méthode héritée de la classe Instrument

Piano implémente Accordable

```
public class Piano extends Instrument implements Accordable {
    private int nombreDePedales;

    // ... constructeur

    public double jouer(Personne[] joueurs){
        // code de la méthode jouer
    }

    public double accorder(){
        // code de la méthode accorder
    }
}
```

Classes et interfaces

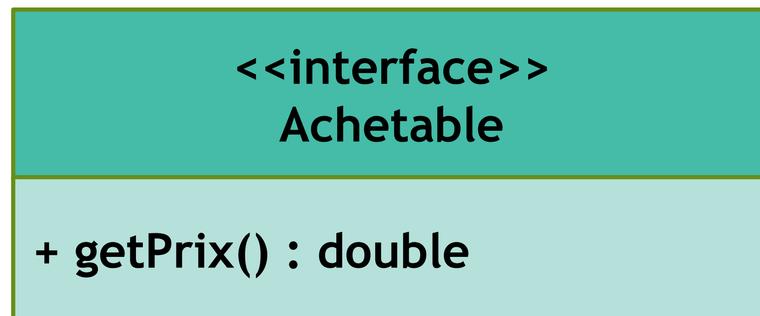
- Une classe concrete qui implémente une interface détaille **toutes ses méthodes**
- Une interface peut être implémentée par plusieurs classes
  - Et une classe peut implementer plusieurs interfaces

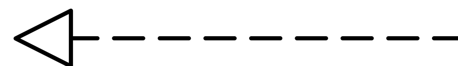
```
class NomClasse implements Interface1,Interface2
```

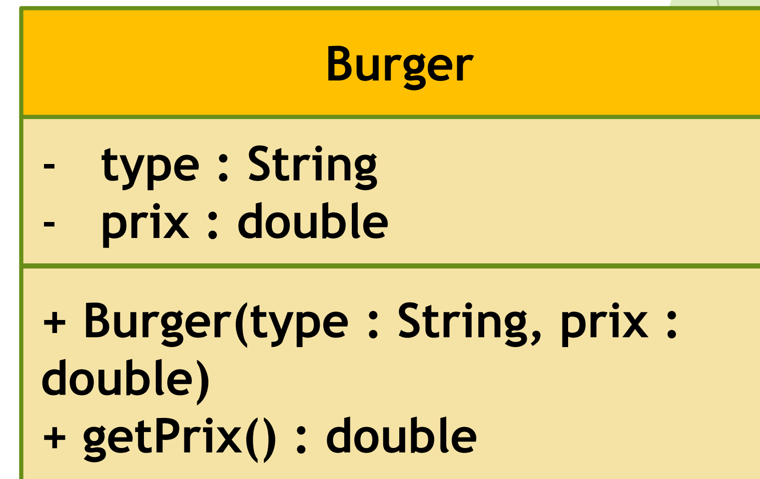
  - Une classe peut hériter d'une superclasse et implémenter plusieurs interfaces

```
class NomClasse extends Superclasse implements Interface1,Interface2
```

- Diagrammes de classe







L'utilité du polymorphisme : sans

- Une personne doit pouvoir acheter: des burgers, des pianos, des jeux
- Sans interfaces (classe Personne) :
  - void acheter(Burger burger)
  - void acheter(Instrument instrument)
  - void acheter(JeuVideo jeuVideo)
- Même chose pour jouer (Instrument, JeuVideo), accorder(Piano)

Personne
- nom : String - budget : double - budgetActuel : double

// constructeurs
+ acheter(burger : Burger): void
+ acheter(instrument : Instrument) : void
+ acheter(jeuVideo : JeuVideo) : void
+ jouer(instrument : Instrument) : void
+ jouer(jeuVideo : JeuVideo) : void
+ accorder(piano : Piano) : void
// d'autres méthodes

L'utilité du polymorphisme : avec

- Une personne doit pouvoir acheter: des burgers, des pianos, des jeux
- Utilisons les interfaces créées !
  - **Une seule méthode** `void acheter(Achetable)` applicable à tout ce qui est "achetable" : un burger, un instrument, un jeu vidéo
  - Nous pouvons être sûrs de l'**existence des méthodes relevantes** dans les classes respectives (`accorder` pour les accordables, `getPrix` pour les achetables…)
- Même chose pour `void jouer(Jouable)`, `void accorder(Accordable)`

Personne
- nom : String - budget : double - budgetActuel : double

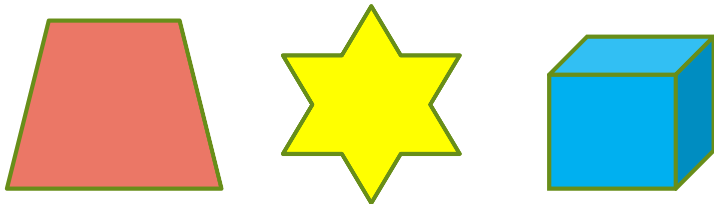
// constructeurs
+ acheter(achetable : Achetable): void
+ jouer(jouable : Jouable) : void
+ accorder(accordable : Accordable) : void
// d'autres méthodes

Interface vs. héritage

Héritage	Interface
• = "est un type de"	• = "fonctionne comme"
• Une sousclasse ne peut hériter que d'**au plus une superclasse**	• Une classe peut implémenter plusieurs interfaces
• Mot clé **`extends`**	• Mot clé **`implements`**
• Peut contenir des variables et méthodes concrètes	• Ne contient jamais de variables • Que de méthodes abstraites
• Une sous-classe **peut** redéfinir les méthodes de la superclasse • Mot clé **`@Override`**	• Une classe concrète **doit** détailler toutes les méthodes de l'interface

Exemple : interface et héritage

- On se rappelle de nos blocs :
  - Chaque bloc émet un son différent
  - Ils peuvent également être faits tourner



Bloc

couleur : String
nom : String

+ Bloc(couleur : String, nom : String)
+ faireTourner() : void

Cube

+ Cube()
+ emettreSon() : void

Etoile

+ Etoile()
+ emettreSon() : void

Trapeze

+ Trapeze()
+ emettreSon() : void

JouetSonore

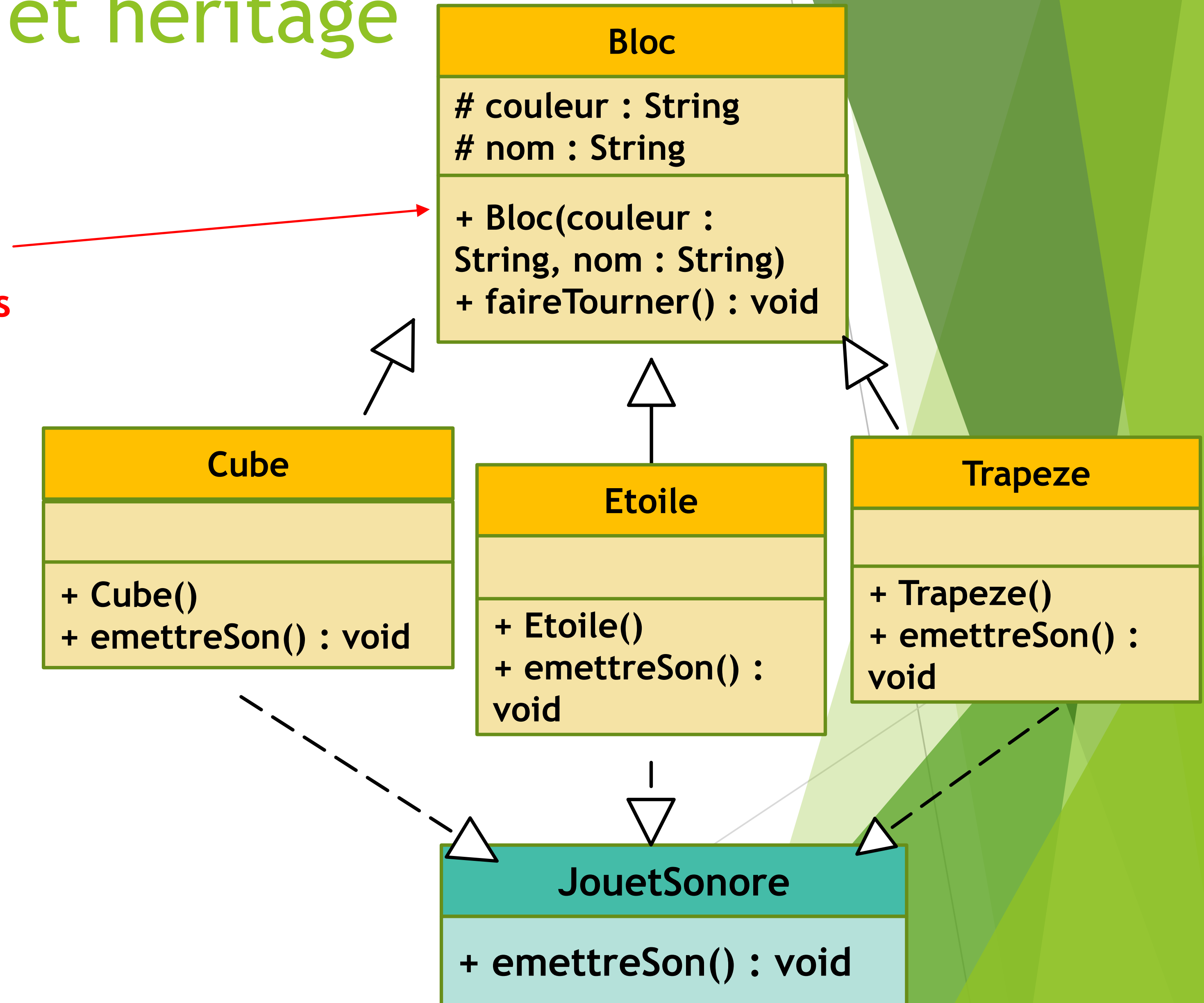
+ emettreSon() : void

Exemple : interface et héritage

Les attributs sont privés

Alors on ne peut pas y accéder des classes Cube, Etoile, Trapeze

Il nous faut des méthodes get



Les exceptions

C'est quoi, une exception ?

- Une exception est un objet

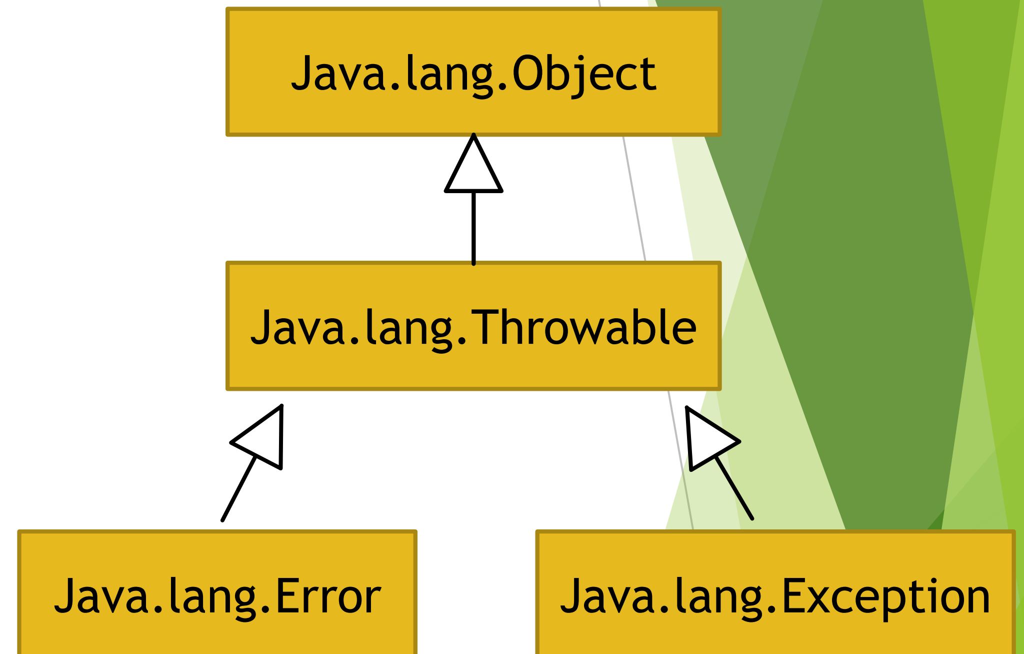
- Elle est instanciée lors d'un événement qui interrompt le flot normal d'un programme

- Une exception peut apparaître lors de l'arrêt d'un programme, ou dans un comportement aberrant

Anticiper et traiter ce comportement peut éviter que des erreurs importantes se produisent

Des divers types d'exceptions

- **Error :**
  - un type d'exception terminale : il arrête l'exécution d'un programme
  - Les erreurs signalent un défaut grave et ne doivent pas être traitées (nous devons savoir si elles se produisent)
  - VirtualMachineError, OutOfMemory, ...
- **Exception :**
  - Moins grave qu'une erreur, peuvent toutefois créer des problèmes
  - Deux types : exceptions à compilation, exceptions à l'execution (runtime)
  - IOException, SQLException, NullPointerException



Les exceptions se lèvent …

- Si une faille technique interrompt le fonctionnement du programme
  - Par exemple si un index déborde (par rapport à la taille d'un tableau)

    IndexOutOfBoundsException
  - Ou si la Java Virtual Machine rencontre un problème :

    OutOfMemoryError
- Si une faille applicative interrompt le fonctionnement du programme
  - Par exemple si un fichier donné en entrée n'est pas trouvé :

    FileNotFoundException

Comment lever des exceptions

- Mod dédié : throw
- **Lever** l'exception dans une clause if :

```
if (monFichier == null ){
	throw new FileNotFoundException("Ce fichier n'existe pas.");
}
```

- **Capturer** l'exception dans un bloc try-catch (mots dédiés)

```
try{
	// code a executer
}
catch (FileNotFoundException e) {
	// code a executer si l'exception est levee
}
```

Contrôler les exceptions

- Une exception peut être contrôlée ou non (checked/unchecked)
- Une méthode peut contrôler elle même les exceptions :

```
public void lire() {
    try{
        FileReader lecteur = new FileReader(monFichier);
    } catch (FileNotFoundException e){  //traitement erreur
    }
}
```

- … ou non

```
public void lire() throws FileNotFoundException{
    … // autre code
    FileReader lecteur = new FileReader(monFichier);
}
```

A savoir

- Les blocs try-catch peuvent avoir multiples blocs catch
  - Mais il n'y a qu'un bloc try, qui contient un code commun
  - Si blocs catch multiples : exceptions en hiérarchie inversée
    - Sous-classes avant superclasses : NullPointerException avant RuntimeException
- Les blocs try peuvent avoir un bloc finally après 0, 1, 2... blocs catch
  - Il s'exécute obligatoirement
  - Il permet de bien clôturer des processus ouverts (e.g. un fichier ouvert)
- Deux instructions utiles :
  - System.err.println(String) : une méthode qui permet d'afficher des textes lors du lever d'une exception
  - printStackTrace() : méthode applicable à tout objet qui implémente l'interface Throwable; elle trace la source de l'erreur ou exception

Un exemple : index out of Bounds

Regardons ce code

MonException.java

```java
public class MonException {
    public static void main(String[] args) {
        MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
        int taille = 3;
        int[] monTableau = new int[taille];
        int monEntree = 10;
        monTableau[0] = monEntree;
        System.out.println("La premiere composante du tableau est " + monTableau[0]);
        objet.doubler(monEntree);
        monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
        System.out.println("La deuxieme composante du tableau est " + monTableau[1]);

    }

    public int augmenter (int index) {
        System.out.println("La valeur initiale de l'index est " + index);
        index = index + 1;
        System.out.println("L'index a ete augmente a " + index);
        return index;
    }

    public int doubler (int valeur) {
        System.out.println("Nous allons doubler la valeur " +valeur);
        valeur = valeur * valeur;
        System.out.println("Valeur doublee " + valeur);
        return valeur;
    }
}
```

Lors d'une execution normale

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
La deuxieme composante du tableau est 100
```

Avec une erreur

ErrorProgram/src/MonException.java

```java
1
2  public class MonException {
3      public static void main(String[] args) {
4          MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5          int taille = 1;
6          int[] monTableau = new int[taille];
7          int monEntree = 10;
8          monTableau[0] = monEntree;
9          System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
12         System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
13
14     }
15
```

taille changée

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
Exception in thread "main" L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
java.lang.ArrayIndexOutOfBoundsException: 1
	at MonException.main(MonException.java:11)
```

Lever l'exception

```java
1
2 public class MonException {
3     public static void main(String[] args) {
4         MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
5         int taille = 1;
6         int[] monTableau = new int[taille];
7         int monEntree = 10;
8         monTableau[0] = monEntree;
9         System.out.println("La premiere composante du tableau est " + monTableau[0]);
10        objet.doubler(monEntree);
11        try {
12            monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13            System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14        }
15        catch(IndexOutOfBoundsException e) {
16            System.out.println("Taille du tableau depassee !");
17        }
18    }
19
20    public int augmenter (int index) {
21        System.out.println("La valeur initiale de l'index est " + index);
22        index = index + 1;
```

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Jav
La premiere composante du tableau est 10
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
```

try-catch

cherche une exception du type IndexOutOfBounds

exception levée

Tracer la source de l'erreur

```
 2 public class MonException {
 3     public static void main(String[] args) {
 4         MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
 5         int taille = 1;
 6         int[] monTableau = new int[taille];
 7         int monEntree = 10;
 8         monTableau[0] = monEntree;
 9         System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         try {
12             monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13             System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14         }
15         catch(IndexOutOfBoundsException e) {
16             System.err.println("Taille du tableau depassee !");
17             e.printStackTrace();
18         }
19         finally {
20             System.out.println("Dans le bloc finally");
21         }
22     }
```

Problems | Javadoc | Declaration | Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_141\bin\javaw.exe (Jan 17, 2018, 1:43:26 PM)
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
java.lang.ArrayIndexOutOfBoundsException: 1
        at MonException.main(MonException.java:12)
Dans le bloc finally
```

System.out.println remplacé par System.err.println

tracer l'erreur

Tracer la source de l'erreur

```
 2 public class MonException {
 3     public static void main(String[] args) {
 4         MonException objet = new MonException(); // nous avons besoin d'un objet pour pouvoir utiliser les methodes en bas
 5         int taille = 1;
 6         int[] monTableau = new int[taille];
 7         int monEntree = 10;
 8         monTableau[0] = monEntree;
 9         System.out.println("La premiere composante du tableau est " + monTableau[0]);
10         objet.doubler(monEntree);
11         try {
12             monTableau[objet.augmenter(0)] = objet.doubler(monEntree);
13             System.out.println("La deuxieme composante du tableau est " + monTableau[1]);
14         }
15         catch(IndexOutOfBoundsException e) {
16             System.err.println("Taille du tableau depassee !");
17             e.printStackTrace();
18         }
19         finally {
20             System.out.println("Dans le bloc finally");
21         }
22     }
```

Problems Javadoc Declaration Console

```
<terminated> MonException [Java Application] C:\Program Files\Java\jre1.8.0_141\bin\javaw.exe (Jan 17, 2018, 1:43:26 PM)
Nous allons doubler la valeur 10
Valeur doublee 100
La valeur initiale de l'index est 0
L'index a ete augmente a 1
Nous allons doubler la valeur 10
Valeur doublee 100
Taille du tableau depassee !
java.lang.ArrayIndexOutOfBoundsException: 1
        at MonException.main(MonException.java:12)
Dans le bloc finally
```

Un bloc finally

Nouvelles exceptions & bonnes pratiques

Créer une nouvelle exception

- Nous pouvons créer de nouvelles exceptions en Java

BP1 : Toujours d'abord utiliser les exceptions qui existent déjà en Java !

- Si contrôlée, l'exception hérite impérativement de la classe Exception

- Si non-contrôlées, l'exception hérite de la classe RuntimeException

- La nouvelle exception formera une nouvelle classe, avec des attributs et des méthodes.
  - N'oubliez pas d'utiliser l'héritage d'Exception/RuntimeException !

Exemple : Etudiant non-trouvé

```
public class EtudiantNonTrouve extends RuntimeException{
        // constructeur personalisé
        public EtudiantNonTrouve(String m){
                super(m); // nous utilisons le constructeur d'Exception
        }
}
```

```
public class ClassePrincipale{
        public static void main(String[] args) {
                Etudiant[] mesEtudiants = new Etudiant[5];
                for (int i = 0; i < mesEtudiants.length; i++){
                        if (mesEtudiants[i].getNom().equals("Dupont")){
                                ...
                        }
                        else
                                throw new EtudiantNonTrouve("Aucun Dupont dans la classe !");
                }
        } }
```

D'autres bonnes pratiques

- Important de capturer toute exception impactant notre code

BP2 : Utiliser les sous-classes spécifiques des exceptions, plutôt que les superclasses (Exception/RuntimeException)

- Les blocs try-catch-finally ont des structures spéciales

BP3 : Ne jamais utiliser les blocs catch{} comme branche alternative d'execution (c'est pourquoi on a if-then-else !)

BP4 : Ne jamais utiliser `return` lorsqu'une exception est levée