

# Les Exceptions en Java



Cours de Programmation Orientée Objet Avancée

ISSET Bizerte

CHALOUAH Anissa

~~EXCEPTIONS~~

# Introduction aux exceptions

- 1) Introduction aux exceptions
- 2) Exception : Définition
- 3) Exception : Mécanisme

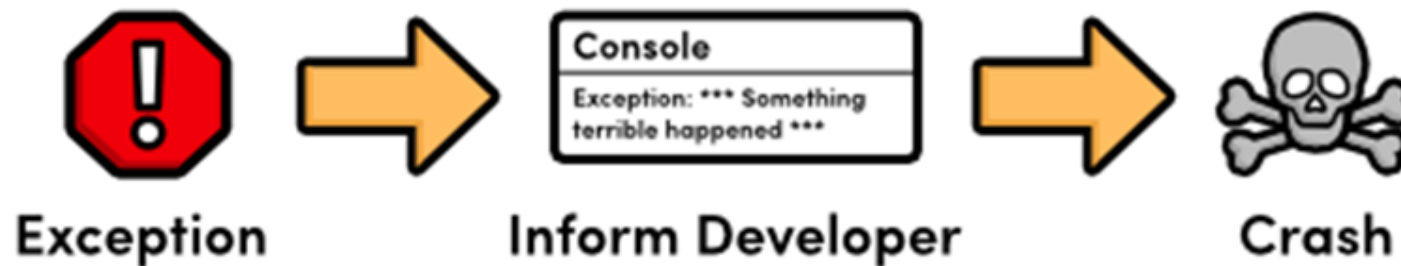


# Introduction aux Exceptions

- ▶ Avoir un programme qui compile bien, n'est en aucun cas une garantie d'un programme qui fonctionne correctement.
- ▶ un programme peut être rendue instable par toute une série de facteurs :
  - ▷ Des **problèmes** liés au **matériel** : perte subite d'une connexion à un port, un disque défectueux...
  - ▷ Des **actions imprévues de l'utilisateur**, entraînant par exemple une division par zéro...
  - ▷ Des **débordements de stockage** dans les structures de données...
- ▶ Java prend en charge la gestion de telles erreurs qui peuvent conduire à l'arrêt brutale de l'exécution du programme

# Exception : Définition

- ▶ Une exception est un **signal** qui
  - ▶ indique qu'un **événement anormal** (une **erreur**) est survenu dans un programme,
  - ▶ **interrompt** le flot d'exécution du programme.



- ▶ La récupération (le traitement) de l'exception permet au programme de continuer son exécution.



# Programme sans gestion de l'exception

```
class Action1 {  
    public void meth() {  
        int x;  
        System.out.println(" ...Avant incident");  
        x=1/0;  
        System.out.println(" ...Après incident");  
    }  
}
```

---- java UseAction1

Début du programme

...Avant incident

java.lang.ArithmeticException : / by zero

---- : Exception in thread "main"

sortir du bloc

```
class UseAction1 {  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        Obj.meth();  
        System.out.println("Fin du programme.");  
    }  
}
```

sortir du bloc

# Gestions des exceptions : Principe

- ▶ Lorsque qu'une erreur a été détectée à un endroit, on la **signale** en **lançant** (**throw**) un objet contenant toutes les informations que l'on souhaite donner sur l'erreur (« lancer » = créer un objet disponible pour le reste du programme)
- ▶ à l'endroit où l'on souhaite gérer l'erreur, on peut **attraper** (**catch**) l'objet **lancé**
- ▶ si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.
- ▶ Un tel mécanisme s'appelle gestion des exceptions.

# Gestions des exceptions : Principe

- ▶ Une exception est un signal déclenché par une instruction et traité par une autre
  - ▷ Il faut marquer les instructions susceptible de générer l'erreur par un try
  - ▷ Il faut qu'un objet soit capable de **signaler** ou **lever** (**throw**) une exception à un autre objet
  - ▷ Il faut que l'autre objet puisse **saisir** (**catch**) une exception afin de la **traiter**
- ▶ Lorsque l'exception se produit le contrôle est transféré à un *gestionnaire d'exceptions*:
  - ▷ Séparation de l'exécution normale de l'exécution en cas de condition anormale

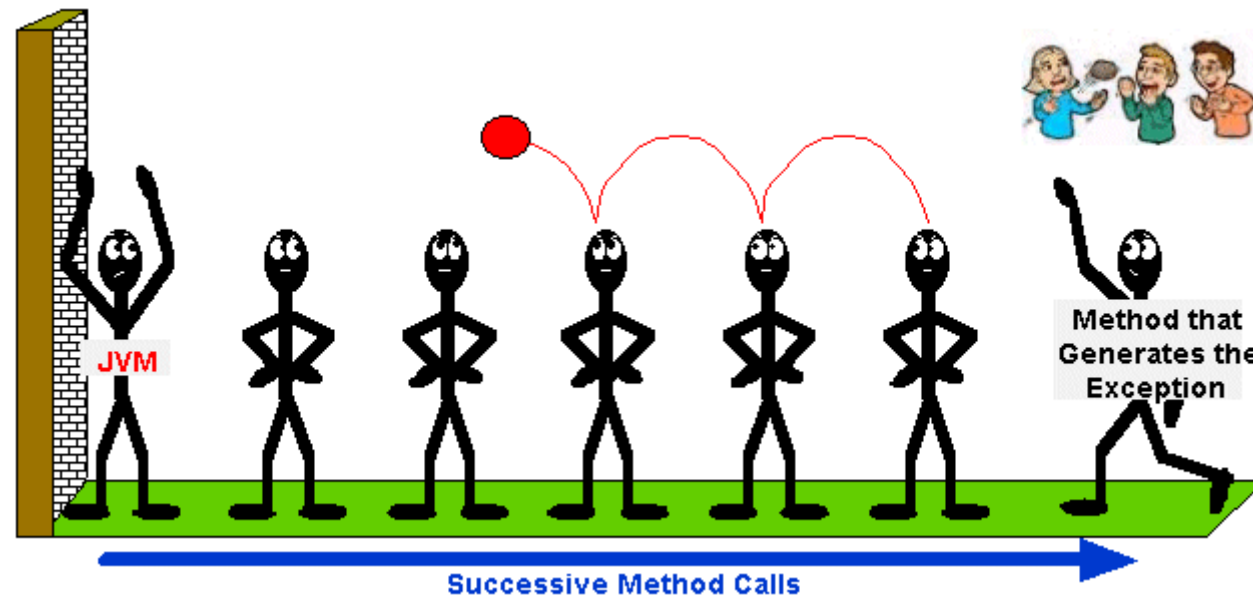
# Gestions des exceptions : Mécanisme(1)

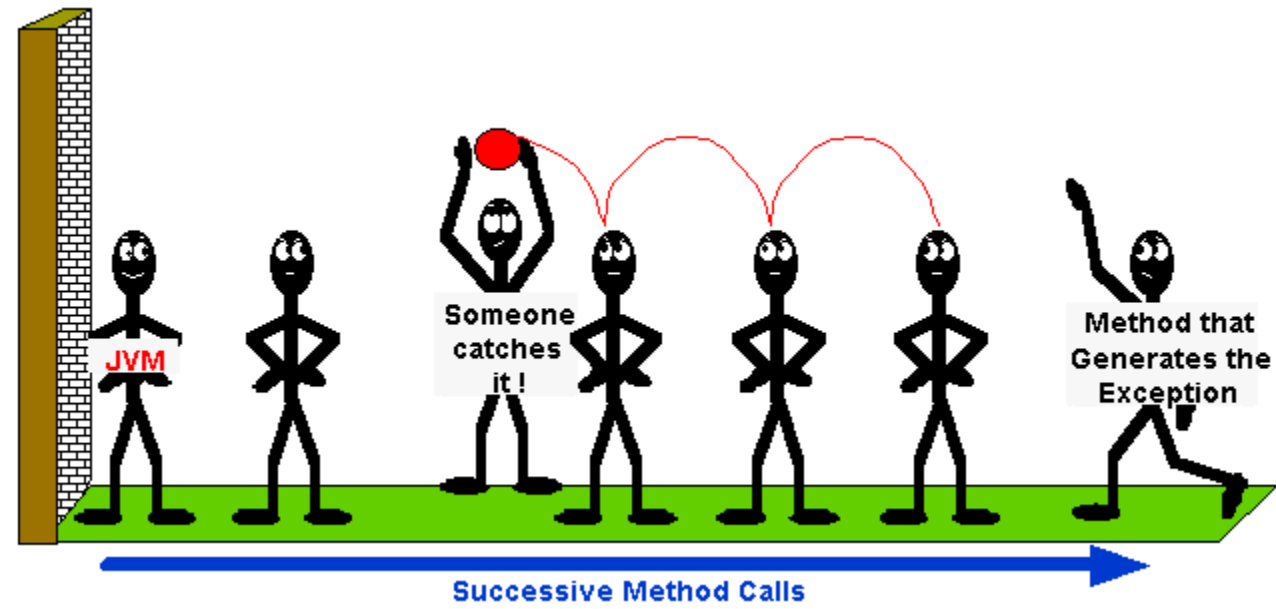
- ▶ Lorsqu'une exception est levée, le programme arrête son exécution et ne reprendra que dans un bloc *catch()* adéquat de l'exception.
- ▶ La méthode qui a levé l'exception va alors :
  - ▷ la traiter **immédiatement**
  - ▷ ou va la **propager** à la **méthode appelante** qui à son tour va traiter l'exception ou la propager.
- ▶ La seule façon de stopper ce parcours sera de **traiter l'exception** (*catch*).



# Gestions des exceptions : Mécanisme(2)

- Il faut voir ce phénomène comme des passes au rugby: l'exception est le ballon et il est transmis de joueurs en joueurs jusqu'à celui qui va 'traiter l'exception' et marquer l'essai.





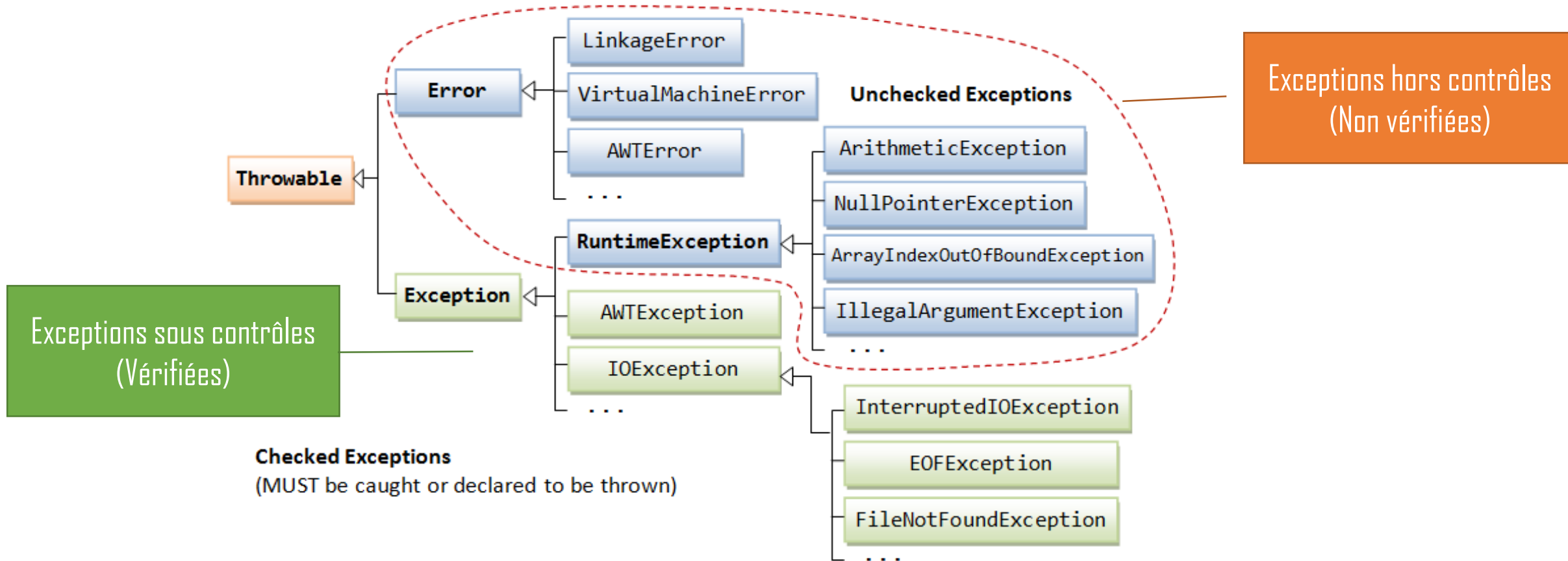
# Gestions des exceptions : Avantages

- ▶ Les exceptions permettent :
  - ▷ de transférer le flot de contrôle de l'instruction qui lève l'exception (qui détecte l'anomalie) vers la partie du programme capable de la traiter ;
- ▶ d'éviter de surcharger le code d'une méthode avec de nombreux tests concernant ces cas anormaux ;
- ▶ de regrouper le traitement des cas anormaux et erreurs ;
- ▶ de classer les anomalies (différents types d'exceptions).

# Hiérarchie des exceptions



# Hiérarchie des exceptions



# La classe throwable

- ▶ Cette classe descend directement de la classe **Object**.
- ▶ La classe *Throwable* est la classe mère de toutes les exceptions et erreurs : seules des instances de *Throwable* ou de ses classes dérivées peuvent être levée par l'instruction *throw* ou être argument d'un *catch*.
- ▶ La classe *Throwable* a un constructeur par défaut et un constructeur qui a en argument une chaîne de caractères : le « message » de l'exception.

Throwable()	Le constructeur par défaut.
Throwable(String message)	Le constructeur avec un paramètre, le message de l'exception.

# La classe throwable (2)

► Les principales méthodes de la classe **Throwable** sont :

Méthodes	Rôle
String getMessage( )	lecture du message
void printStackTrace( )	affiche l'exception et l'état de la pile d'exécution au moment de son appel
String toString()	

# Exceptions hors contrôle (Unchecked exceptions)

## ▶ classes dérivées de `java.lang.Error` :

- ▷ ce sont des erreurs qui ne peuvent pas être récupérées (plus de mémoire, classe non trouvée etc.) ;
- ▷ Vous n'avez pas à traiter les instances de la classe **Error**

## ▶ classes dérivées de `java.lang.RuntimeException` :

- ▷ ce sont des erreurs de programmation, elles ne devraient pas se produire(`NullPointerException`, `IndexOutOfBoundsException`).



**Les exceptions hors contrôle n'ont pas besoin d'être spécifié dans la déclaration d'une méthode (throws)**



# Unchecked Exceptions : RuntimeException examples

- ▷ `ArithmeticException` (e.g., bad computation such as divide by 0)
- ▷ `ArrayStoreException` (e.g., storing wrong type of object in array)
- ▷ `ClassCastException` (e.g., cannot typecast one class to another)
- ▷ `IndexOutOfBoundsException` (e.g., gone outside array bounds)
- ▷ `NoSuchElementException` (e.g., cannot find any more elements)
- ▷ `NullPointerException` (e.g., attempt to send message to null)
- ▷ `NumberFormatException` (e.g., trouble converting to a number)

# Exceptions sous contrôle (checked exceptions)

- ▶ Ce sont les classes dérivées de `Exception` (mais pas de **`RuntimeException`**).
- ▶ Elles peuvent (**doivent** !) être récupérées et traitées.
- ▶ Elles correspondent à la notion de robustesse.



Si on écrit une méthode qui lève une exception contrôlée, on doit utiliser une clause **throws** afin de déclarer l'exception au sein de la signature de la méthode.

# Checked Exceptions : Examples

- ▶ **ClassNotFoundException** (e.g., tried to load an undefined class)
- ▶ **CloneNotSupportedException** (e.g., cannot make copy of object)
- ▶ **DataFormatException** (e.g., bad data conversion)
- ▶ **IllegalAccessException** (e.g., access modifiers prevent access)
- ▶ **InstantiationException** (e.g., problem creating an object)
- ▶ **IOException**
  - ▷ **EOFException** (e.g., end of file exception)
  - ▷ **FileNotFoundException** (e.g., cannot find a specified file)

# Gestions des Exceptions



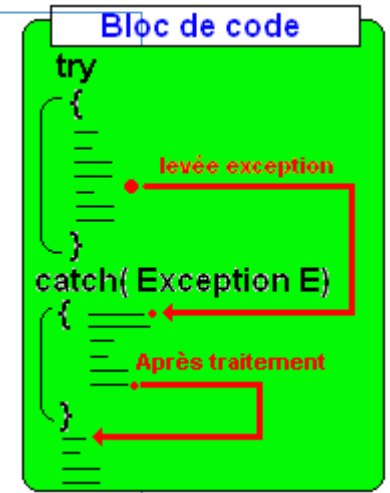
# Traitement d'une exception : mots clés

On utilise trois types de blocs d'instructions :

- ▶ **try { ... }** : permet de spécifier une section de code sur laquelle on s'attend qu'une exception (une erreur) soit levée.
- ▶ **catch(exception e) { ... }** : sert à spécifier le code à exécuter si une exception du type de celle passée en argument se produit. On peut avoir plusieurs blocs **catch** à la suite d'un bloc **try** si nécessaire.
- ▶ **finally { ... }** : introduit un code de traitement d'exception, qui sera toujours exécuté qu'une exception est été levé ou non. Ce bloc est facultatif.

# Traitement d'une exception : syntaxe

```
try {  
    /* <lignes de code à protéger> */  
} catch ( UneException E ) {  
    /* <lignes de code réagissant à l'exception UneException > */  
}
```



Le type **UneException** est obligatoirement une classe qui **hérite de la classe Exception**.

# Traitement d'une exception : Exemple

```
class Action1 {  
    public void meth(){  
        int x;  
        System.out.println(" ...Avant incident");  
        x=1/0; ← engendre une exception  
        System.out.println(" ...Après incident");  
    }  
}
```

```
class UseAction1{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth(); ← levée d'ArithmeticException  
        }  
        catch(ArithmeticException E){ ← traitement, puis poursuite de  
            System.out.println("Interception exception"); ← l'exécution  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```

---- *java UseAction1*  
Début du programme.  
...Avant incident  
Interception exception  
Fin du programme.  
---- : *operation complete.*

# Le bloc finally

- ▶ les clauses **catch** sont suivies de manière optionnelle par un bloc **finally** qui contient du code qui sera exécuté quelle que soit la manière dont le bloc **try** a été quitté.
- ▶ le bloc **finally** permet de spécifier du **code dont l'exécution est garantie** quoi qu'il arrive (qu'une exception ait été lancée ou pas par le bloc try)
- ▶ **But: faire le ménage** (fermer des fichiers, des connexions, etc..)



# Le bloc finally : Syntaxe

```
try {  
    <code à protéger>  
}  
catch (exception1 e ) {  
    <traitement de l'exception1>  
}  
catch (exception2 e ) {  
    <traitement de l'exception2>  
}  
...  
finally {  
    <action toujours effectuée>  
}
```

# Exceptions multiples

- 1) Interceptions de plusieurs exceptions
- 2) Ordre d'exécution de plusieurs exceptions hiérarchisées



# Interceptions de plusieurs exceptions

- ▶ Dans un gestionnaire try...catch, il est en fait possible d'intercepter plusieurs types d'exceptions différentes et de les traiter.
- ▶ la syntaxe d'un tel gestionnaire fonctionne comme un sélecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.
- ▶ Lorsqu'une erreur survient dans le bloc try,
  - ▷ la suite des instructions du bloc est abandonnée
  - ▷ les clauses catch sont testées **séquentiellement**
  - ▷ le premier bloc catch correspondant à l'erreur est exécuté.


# Interceptions de plusieurs exceptions : Syntaxe

```
try {  
    < bloc de code à protéger >  
    }  
    catch ( TypeException1 E ) { <Traitement TypeException1 > }  
    catch ( TypeException2 E ) { <Traitement TypeException2 > }  
    .....  
    catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

- Où TypeException1, TypeException2, ... , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes**.
- Seule une seule clause **catch** ( TypeException E ) {...} est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

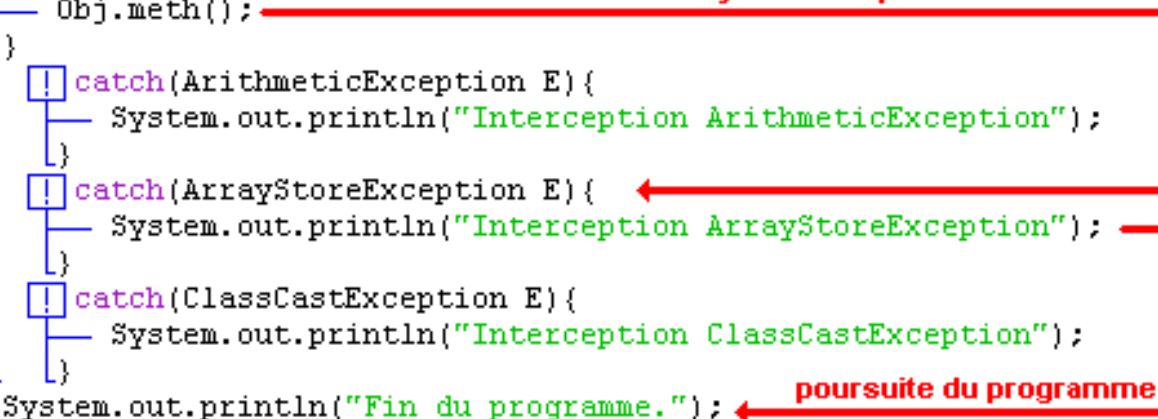
# Interceptions de plusieurs exceptions : Exemple (1)

```
class Action2 {  
    public void meth(){  
        // une exception est levée ...  
    }  
}
```



**ArrayStoreException**


```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception ArithmeticException");  
        }  
        catch(ArrayStoreException E){  
            System.out.println("Interception ArrayStoreException");  
        }  
        catch(ClassCastException E){  
            System.out.println("Interception ClassCastException");  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```


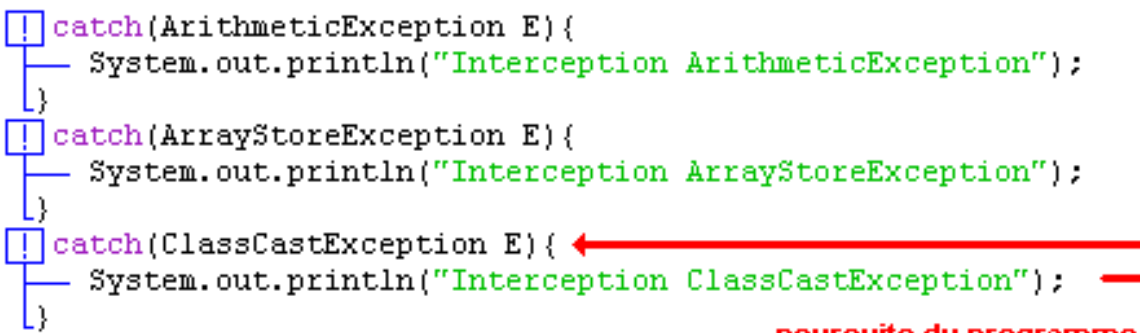



**ArrayStoreException**

**poursuite du programme**


# Interceptions de plusieurs exceptions : Exemple (2)

```
class Action2 {  
    public void meth(){  
        // une exception est levée ...  
          
        ClassCastException  
    }  
}
```

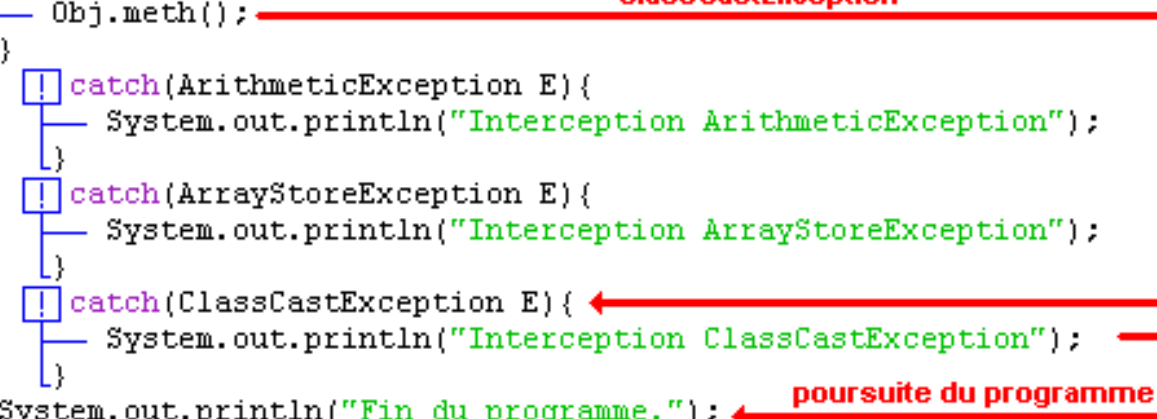
```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  ClassCastException  
        }  
          
         poursuite du programme  
        System.out.println("Fin du programme.");  
    }  
}
```

# Interceptions de plusieurs exceptions : Exemple (3)

```
class Action2 {  
    public void meth(){  
        // une exception est levée ...  
    }  
}
```



```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception ArithmeticException");  
        }  
        catch(ArrayStoreException E){  
            System.out.println("Interception ArrayStoreException");  
        }  
        catch(ClassCastException E){  
            System.out.println("Interception ClassCastException");  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```



# Ordre d'interception d'exceptions hiérarchisées

- ▶ l'ordre des blocs catch doit se faire **de l'erreur la plus spécifique à la plus générale !**
  - ▷ On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans **l'ordre inverse de la hiérarchie.**



# Ordre d'interception d'exceptions hiérarchisées : exemple

Soit une hiérarchie d'exceptions dans java

java.lang.Exception

|--java.lang.RuntimeException

|--java.lang.ArithmeticException

|--java.lang.ArrayStoreException

|--java.lang.ClassCastException

```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(RuntimeException E){  
            System.out.println("Interception RuntimeException");  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception ArithmeticException");  
        }  
        catch(ArrayStoreException E){  
            System.out.println("Interception ArrayStoreException");  
        }  
        catch(ClassCastException E){  
            System.out.println("Interception ClassCastException");  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```

# Ordre d'interception d'exceptions hiérarchisées : exemple

## Résultats de l'exécution :

---- java UseAction2

```
UseAction2.java:19: exception java.lang.ArithmeticException has already been caught  
    catch (ArithmeticException E) {  
      ^
```

```
UseAction2.java:22: exception java.lang.ArrayStoreException has already been caught  
    catch (ArrayStoreException E) {  
      ^
```

```
UseAction2.java:25: exception java.lang.ClassCastException has already been caught  
    catch (ClassCastException E) {  
      ^
```

**3 errors**

Le compilateur proteste à partir de la clause **catch** ( ArithmeticException E ) en nous indiquant que l'exception est déjà interceptée et ceci trois fois de suite.

# Ordre d'interception d'exceptions hiérarchisées : exemple

```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception ArithmeticException");  
        }  
        catch(ArrayStoreException E){  
            System.out.println("Interception ArrayStoreException");  
        }  
        catch(ClassCastException E){  
            System.out.println("Interception ClassCastException");  
        }  
        catch(RuntimeException E){  
            System.out.println("Interception RuntimeException");  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```

La classe parent doit être placée après ses classes filles

# Throw, throws



# Lever une exception (throw)

- ▶ La Java machine peut déclencher une exception automatiquement comme dans l'exemple de la levée d'une `ArithmeticException` lors de l'exécution de l'instruction `"x = 1/0 ;"`.
- ▶ La Java machine peut aussi lever (déclencher) une exception à votre demande suite à la rencontre d'une instruction `throw`.

# Lever une exception (throw)

► L'opérateur **throw** permet de lever une exception

```
if (<condition anormale>) {  
    throw new TypeException(<parametres effectifs>);  
}
```

## Fraction.java

```
public Fraction(int num, int den) {  
    if(den == 0) {  
        throw new ArithmeticException("Division par zero");  
    }  
    ...  
}
```

# Lever une exception (throw) : Exemple

```
class Action3 {  
    public void meth(){  
        int x=0;  
        System.out.println(" ...Avant incident");  
        if (x==0)  
            throw new ArithmeticException("Mauvais calcul !");  
        System.out.println(" ...Après incident");  
    }  
}
```

```
class UseAction3{  
    public static void main(String[] Args) {  
        Action3 Obj = new Action3();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception exception : "+E.getMessage());  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```

Résultats de l'exécution :

---- *java UseAction3*

Début du programme.

...Avant incident

Interception exception : Mauvais calcul !

Fin du programme.

---- : *operation complete.*

# Propagation d'une exception (throws)

- ▶ Lorsque les méthodes utilisées génèrent des exceptions, autres que des `RuntimeException`, on a **le devoir** de capter ces exceptions.
- ▶ Mais parfois il n'est pas possible ou pas souhaité de traiter l'exception immédiatement.
  - ▷ Dans ce cas on doit déclarer celle-ci en utilisant le mot-clé **throws** dans l'en-tête de la méthode susceptible de générer l'exception en question.
  - ▷ On **propage** alors l'exception à la méthode appelante, c'est à dire à la méthode qui fait appel à la méthode à partir de laquelle l'exception est générée.



# Propagation d'une exception (throws)

Si une méthode est susceptible de lever une exception qu'on ne veut pas gérer au sein de la méthode, elle se doit de prévenir les appelants avec le mot clé **throws**.

# Spécification d'une exception : Syntaxe

```
<modifieurs> Type maMethode(<parametres>) throws TypeExc1, TypeExc2, ...
```

## Exemple

```
protected static void meth ( int x, char c )  
    throws IOException, ArithmeticException {  
    .....  
}
```

# Spécification d'une exception : Exemple

```
import java.io.*;
```

```
class Action4 {  
    public void meth()  
    {  
        int x=0;  
        System.out.println(" ...Avant incident");  
        if (x==0)  
        {  
            throw new IOException("Problème d'E/S !");  
        }  
        System.out.println(" ...Après incident");  
    }  
}
```

Résultats de l'exécution :

---- java UseAction4

UseAction4.java:8: unreported exception java.io.IOException; must be caught or declared to be thrown

```
        throw new IOException("Problème d'E/S !");  
            ^
```

1 error

```
class UseAction4{  
    public static void main(String[] Args) {  
        Action4 Obj = new Action4();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(IOException E){  
            System.out.println("Interception exception : "+E.getMessage());  
        }  
        System.out.println("Fin du programme.");  
    }  
}
```

# Spécification d'une exception : Exemple

```
import java.io.*;

class Action4 {
    public void meth() throws IOException {
        int x=0;
        System.out.println(" ...Avant incident");
        if (x==0)
            throw new IOException("Problème d'E/S !");
        System.out.println(" ...Après incident");
    }
}

class UseAction4{
    public static void main(String[] Args) {
        Action4 Obj = new Action4();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
        catch(IOException E){
            System.out.println("Interception exception : "+E.getMessage());
        }
        System.out.println("Fin du programme.");
    }
}
```

signaler l'exception susceptible d'être propagée

Interception de l'exception dans le bloc englobant

Résultats de l'exécution :

---- java UseAction4

Début du programme.

...Avant incident

Interception exception : Problème d'E/S !

Fin du programme.

# Exceptions personnalisées



- ▶ Il est possible de créer dans vos applications **vos propres classes d'exception**.
- ▶ La nouvelle classe d'exception créée doit obligatoirement **hériter** de la classe **Exception** de Java ou **d'une de ses sous-classes** plus spécifiques.

```
class MaClasseException extends Exception{  
  
}
```

► La structure de votre nouvelle classe d'exception doit au moins posséder un des deux constructeurs:

- un sans paramètre
- un avec un paramètre de type String correspondant au message associé à l'erreur générée.

```
class MaClasseException extends Exception{  
    MaClasseException() {  
        super();  
    }  
  
    MaClasseException(String monMessage) {  
        super(monMessage);  
    }  
}
```

Le constructeur prenant un objet de type **String** en paramètre permet de spécifier un message pour l'exception. Ce message est récupérable par **getMessage()**.

- ▶ Les méthodes de la classe `Exception` seront également disponibles pour votre nouvelle classe. Vous pourrez donc:
  - ▷ afficher la valeur du message éventuellement transmis en paramètre au constructeur de la classe grâce à la méthode **`getMessage()`**
  - ▷ afficher, sur la sortie d'erreur, la liste des appels ayant conduit à la génération de l'exception grâce à la méthode **`printStackTrace()`**



# Lever une exception personnalisée

- Pour une exception personnalisée, le mode d'action est strictement identique
  - ▷ Il faut spécifier les exceptions propagées dans la signature de la méthode (**throws**)
  - ▷ Les exceptions sont levées ou envoyées par le mot-clé **throw**.

```
maMethode throws MaClasseException {  
    if (condition_particuliere)  
        throw new MaClasseException("Message d'erreur");  
}
```

# Exception personnalisée : Exemple

```
class AEgalBException extends Exception {  
    public String message () {  
        return "A égal à B !";  
    }  
}
```

On définit une classe dérivée de la classe `Exception` et une méthode retournant un message.

On indique au compilateur que la *ma\_méthode* est susceptible de **lever** une exception *AEgalBException*.

```
public class EssaiException {  
    static void ma_méthode (int a, int b) throws AEgalBException {  
        if (a == b)  
            throw new AEgalBException ();  
        else  
            System.out.println (a+" et "+b+" OK !");  
    }  
}
```

On **lève** effectivement une instance (**new**) de l'exception.

```
public static void main (String args []) {  
    try {  
        ma_méthode (2,2);  
        System.out.println ("Pas d'erreur");  
    }  
    catch (AEgalBException e) {  
        System.out.println ("Erreur "+e.message());  
    }  
}
```

Ici les deux nombres sont égaux, donc l'exception est levée dans *ma\_méthode* et interceptée par le bloc **catch** du `main`.

La méthode *message* de la classe est appelée et "Erreur A égal à B!" apparaît à l'écran.