



Ministère de l'Enseignement Supérieur et de la  
Recherche Scientifique et de la Technologie  
Institut Supérieur des  
Études Technologiques Bizerte



## Les exceptions en java

# I. Introduction



- Avoir un programme qui compile bien n'est en aucun cas une garantie d'un programme qui fonctionne correctement .
- Différentes causes sont à l'origine de ce problème :
  - ✓ **des problèmes liés au matériel** :perte subite d'une connexion à un port ,un disque défectueux...
  - ✓ **des actions imprévus de l'utilisateur** :entrant 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 .=>**gestion des exceptions**

# Un premier exemple



- Considérons une application qui permet de :
  - ✓ Saisir au clavier deux entiers a et b
  - ✓ Faire appel à une fonction qui permet de calculer et de retourner a divisé par b.
  - ✓ Affiche le résultat

```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=calcul(a, b);
        System.out.println("Resultat="+resultat);
    }
}
```

# Un premier exemple:Exécution



- Nous constatons que le compilateur ne signale pas le cas où b est égal à zero.
- Ce qui constitue un cas fatal pour l'application.
- Voyons ce qui se passe au moment de l'exécution

## Scénario 1 : Le cas normal

```
Donner a:12
Donner b:6
Resultat=2
```

## Scénario 2: cas où b=0

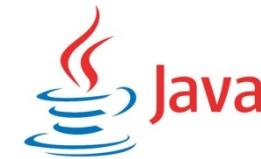
```
Donner a:12
Donner b:0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Appl.calcul(Appl.java:4)
at Appl.main(Appl.java:11)
```

## Définition



- La notion d'exception est offerte aux programmeurs Java pour résoudre de manière efficace et simple le problème de la gestion des erreurs émises lors de l'exécution d'un programme.
- Une exception est un signal qui indique qu'un **événement anormal** est survenu dans un programme
- Le traitement de l'exception permet au programme **de continuer son exécution**.

## Définition



- Une exception est un signal **déclenché** par une instruction et **traité** par une autre
  - 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

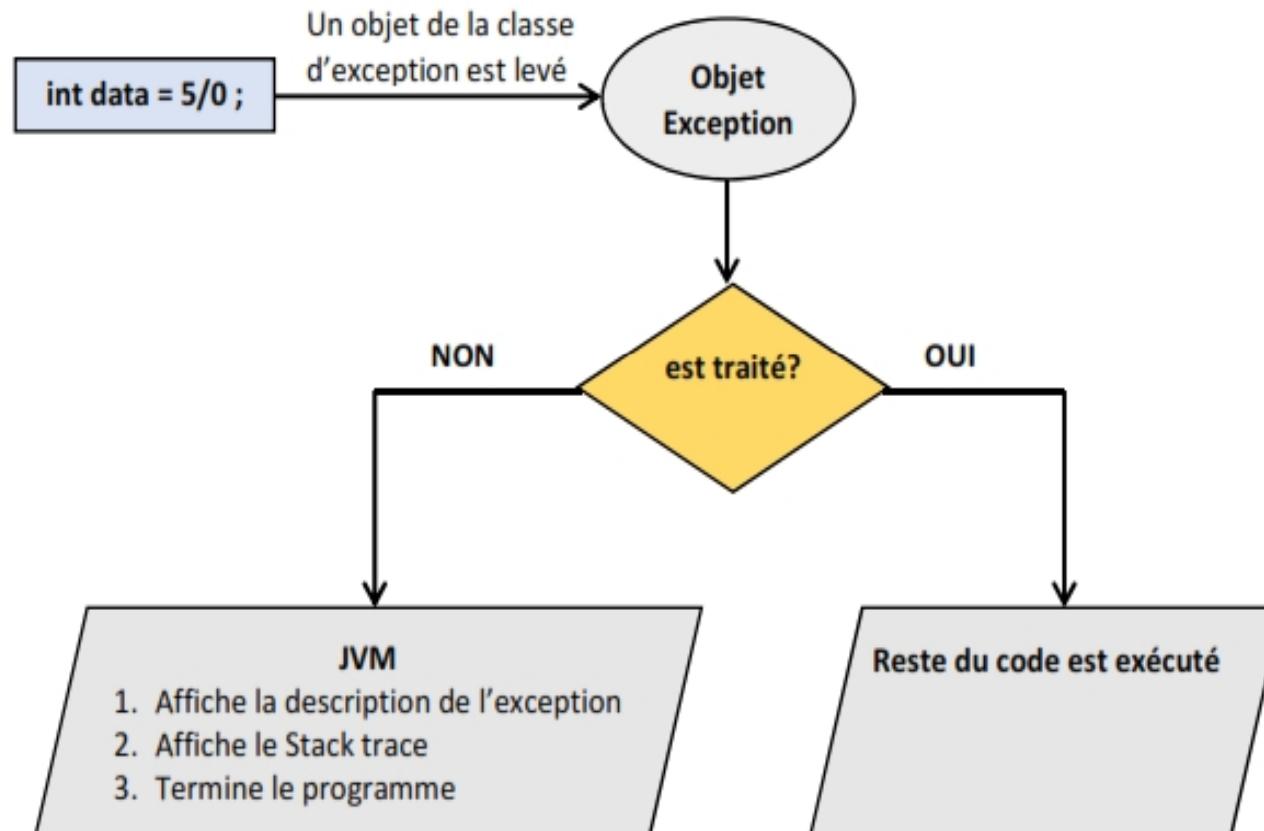
## Avantages



Eviter de surcharger le code d'une méthode avec de nombreux tests concernant ces cas anormaux :

- ✓ regrouper le traitement des cas anormaux et erreurs
- ✓ de classifier les anomalies (différents types d'exceptions)

# Principe du traitement des exceptions en java



## II Les exceptions

### Hiérarchie



En Java, les exceptions sont de véritables objets.

Ce sont des instances de classes qui héritent de la classe **Throwable**.

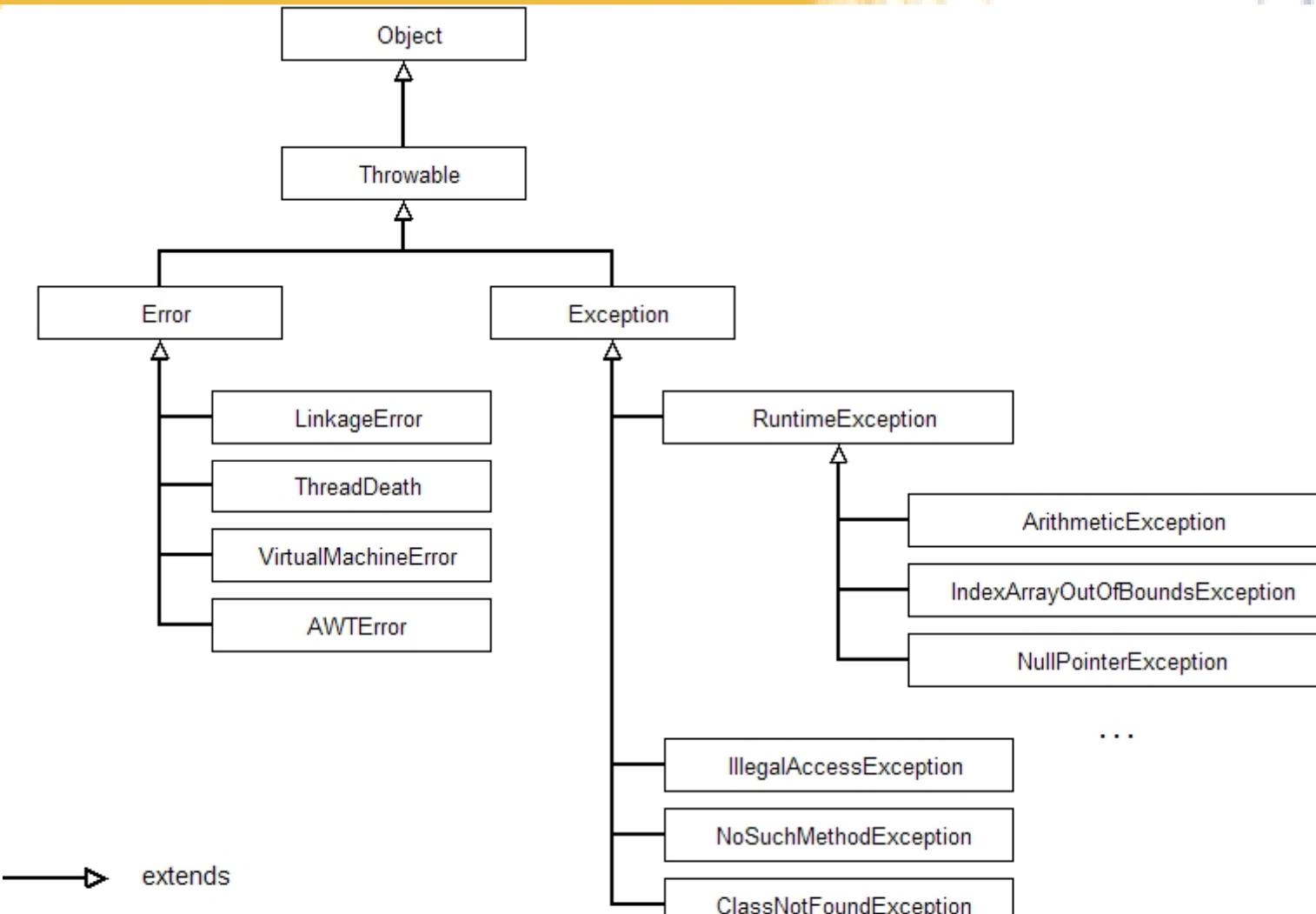
Lorsqu'une exception est levée, une instance de la classe **Throwable** est créée.

Cette classe descend directement de `Object` : c'est la classe de base pour le traitements des erreurs.

Cette classe possède deux constructeurs :

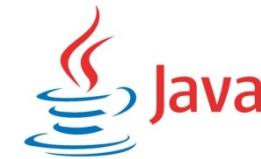
Méthode	Rôle
<code>Throwable()</code>	
<code>Throwable(String)</code>	La chaîne en paramètre permet de définir un message qui décrit l'exception et qui pourra être consultée dans un bloc catch.

# Hiéarchie



### III. La classe throwable

#### Méthodes



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'execution au moment de son appel

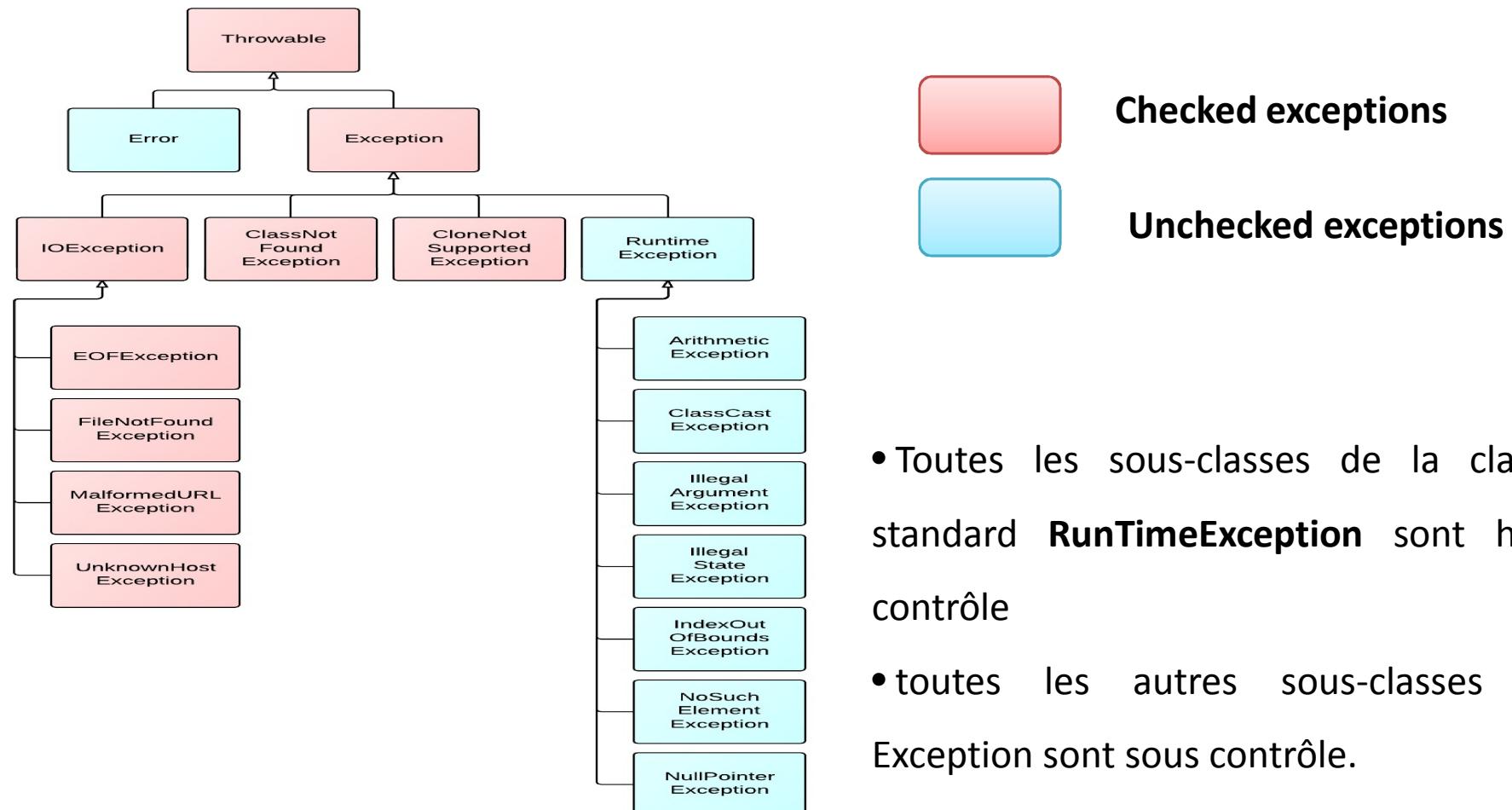
```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (ArithmeticException e) {
            System.out.println("getmessage");
            System.out.println(e.getMessage());
            System.out.println(" ");
            System.out.println("toString");
            System.out.println(e.toString());
            System.out.println(" ");
            System.out.println("printStackTrace");
            e.printStackTrace();
        }
    }
}
```

C:>java TestException  
getmessage  
/ by zero  
  
toString  
java.lang.ArithmeticException: / by zero  
  
printStackTrace  
java.lang.ArithmeticException: / by zero  
at tests.TestException.main(TestException.java:24)

# La classe throwable



Java divise les classes d'exception en deux catégories : les exceptions **sous contrôle** (« checked exceptions ») et les exceptions **hors contrôle** (« unchecked exceptions »).



# Exceptions hors Contrôle:unchecked exceptions



- Ces exceptions servent à gérer les erreurs internes à la MV ou les fautes de programmation (Division par zéro, dépassement des limites d'un tableau, etc.)
- le compilateur n'oblige pas le développeur de les traiter.

Deux ensembles distincts

- ▶ Fautes de programmation gérées par **RuntimeException** et toutes ses sous-classes (NullPointerException, IndexOutOfBoundsException..).
- ▶ Erreurs internes à JAVA gérées par **Error**(p.ex. VirtualMachineError)
  - ★ des erreurs qui ne peuvent pas être récupérés (plus de mémoire...)
  - ★ Ne peuvent pas être déclenchées par le programmeur

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

# Exceptions sous Contrôle:Checked exceptions



- Ce sont les classes dérivées de Exception (mais pas de **RuntimeException**).
- Elles peuvent **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.**

## VI. Gestion des exceptions

### Traitement d'une exception :mots clés



On utilise trois types de blocs d'instructions:

#### **Le bloc d'instructions try {...}**

Le bloc try permet de définir un ensemble d'instructions à surveiller . Ce bloc est obligatoire. Il ne peut pas y avoir de bloc catch (ou finally) sans bloc try associé et les accolades y sont obligatoires.

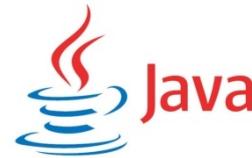
#### **Le bloc d'instructions catch(..){..}**

Il peut y avoir un ou plusieurs blocs catch associé à un bloc try. Chaque bloc catch est associé à un type d'erreur (une classe d'exception) à traiter

#### **Le bloc finally{...}.**

le bloc finally. Ce bloc est facultatif, mais si vous le fournissez, il devra obligatoirement être positionné en dernier. Il permet de dire ce que vous souhaitez faire en fin d'instruction try et ce quelle que soit la manière dont il se termine (en succès ou en échec).

# Traitement d'une exception :Syntaxe



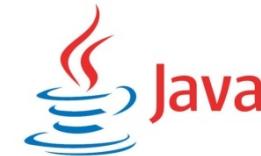
```
try
{
    //Partie de code susceptible de générer une erreur
}
catch (ClasseException1 E)
{
    //Action à réaliser si le cas appartient à ClasseException1
}
catch (ClasseException2 E)

{
    //Action à réaliser si le cas appartient à ClasseException2
...
}
finally
{//Action toujours effectués
...
}

}
```

La ClasseException est obligatoirement une classe qui hérite de la classe **Exception**.

# Traitement d'une exception



```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=0;
        try{
            resultat=calcul(a, b);
        }
        catch (ArithmeticException e) {
            System.out.println("Divisio par zero");
        }
        System.out.println("Resultat="+resultat);
    }
}
```

## Scénario 1

Donner a:12  
Donner b:6  
Resultat=2

## Scénario 2

Donner a:12  
Donner b:0  
Divisio par zero  
Resultat=0

# Traitement d'une exception



Tous les types d'exceptions possèdent les méthodes suivantes :

- **getMessage()** : retourne le message de l'exception

- ✓ `System.out.println(e.getMessage());`
  - ✓ *Résultat affiché : / by zero*

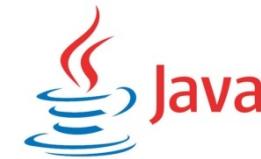
- **toString()** : retourne une chaîne qui contient le type de l'exception et le message de l'exception.

- ✓ `System.out.println(e.toString());`
  - ✓ *Résultat affiché : java.lang.ArithmeticException: / by zero*

- **printStackTrace**: affiche la trace de l'exception

- ✓ `e.printStackTrace();`
  - ✓ *Résultat affiché :*
  - ✓ *java.lang.ArithmeticException: / by zero*
  - ✓ *at App1.calcul(App1.java:4)*
  - ✓ *at App1.main(App1.java:13)*

# 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.
- Dès qu'une exception intervient dans le < bloc de code à protéger>, la Java machine scrute séquentiellement toutes les clauses catch de la première jusqu'à la dernière.
- Si l'exception actuellement levée est d'un des types présents dans la liste des clauses de traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

# 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, TypeException12, ..., 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)



Exemple théorique, supposons que la méthode meth() de la classe Action2 puisse lever trois types différents d'exceptions: ArithmeticException, ArrayStoreException , ClassCastException.

Notre gestionnaire d'exceptions est programmé pour intercepter l'une de ces 3 catégories. Nous figurons ci-dessous les trois schémas d'exécution correspondant chacun à la levée (l'instanciation d'un objet) d'une exception de l'un des trois types et son interception :

# Exemple: Interception d'une ArrayStoreException



```
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(); → ArrayStoreException
        } catch(ArithmetricException E){
            System.out.println("Interception ArithmetricException");
        } catch(ArrayStoreException E){ ←
            System.out.println("Interception ArrayStoreException");
        } catch(ClassCastException E){
            System.out.println("Interception ClassCastException");
        }
        System.out.println("Fin du programme."); ← poursuite du programme
    }
}
```

# Exemple: Interception d'une ClassCastException



```
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(); // ClassCastException
        } catch(ArithmetricException E){
            System.out.println("Interception ArithmetricException");
        } catch(ArrayStoreException E){
            System.out.println("Interception ArrayStoreException");
        } catch(ClassCastException E){ // ClassCastException
            System.out.println("Interception ClassCastException");
        }
        System.out.println("Fin du programme."); // poursuite du programme
    }
}
```

# Exemple: Interception d'une ArithmeticException



```
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(); -> ArithmeticException
        } catch(ArithmetiException 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."); -> poursuite du programme
    }
}
```

# Ordre d'interception d'exceptions hiérarchisées



Dans un gestionnaire try...catch comprenant plusieurs clauses, la recherche de la clause catch contenant le traitement de la classe d'exception appropriée, s'effectue séquentiellement dans l'ordre d'écriture des lignes de code.

# Ordre d'interception d'exceptions hiérarchisées



Exemple : Soit une hiérarchie d'exceptions dans java

```
java.lang.Exception
  |
  +--java.lang.RuntimeException
    |
    +--java.lang.ArithmeticsException
    |
    +--java.lang.ArrayStoreException
    |
    +--java.lang.ClassCastException
```

- Supposons que nous souhaitions intercepter une quatrième classe d'exception, par exemple une **RuntimeException**, nous devons rajouter une clause :  
**catch ( RuntimeException E ) { <Traitement RuntimeException> }**
- Insérons cette clause en premier dans la liste des clauses d'interception :

# 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(RuntimeException E){  
            System.out.println("Interception RuntimeException");  
        }  
        catch(ArithmaticException E){  
            System.out.println("Interception ArithmaticException");  
        }  
        catch(ArrayStoreException E){  
            System.out.println("Interception ArrayStoreException");  
        }  
        catch(ClassCastException E){  
            System.out.println("Interception ClassCastException");  
        }  
        System.out.println("Fin du programme.");  
    }  
}  
  
Résultats de l'exécution :  
---java UseAction2  
  
UseAction2.java:19: exception java.lang.ArithmaticException has already been caught  
        catch(ArithmaticException 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
```

# Ordre d'interception d'exceptions hiérarchisées:Exemple

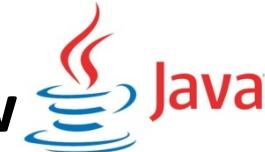


```
class UseAction2{  
    public static void main(String[] Args) {  
        Action1 Obj = new Action1();  
        System.out.println("Début du programme.");  
        try{  
            Obj.meth();  
        }  
        catch(ArithmeticException E){  
            System.out.println("Interception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

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs clauses dans l'ordre inverse de la hiérarchie.

## V. déclenchement d'une exception : throw

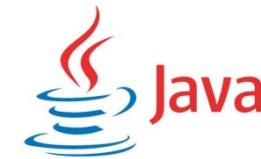


Il est possible de déclencher soi-même des exceptions en utilisant l'instruction **throw**, même de déclencher des exceptions personnalisées .

**Exemple:** Considérons le cas d'un compte qui est défini par un code et un solde et sur lequel, on peut verser un montant, retirer un montant et consulter le solde.

```
public class Compte {  
    private int code;  
    private float solde;  
    public void verser(float mt){  
        solde=solde+mt;  
    }  
    public void retirer(float mt){  
        if(mt>solde) throw new RuntimeException("Solde Insuffisant");  
        solde=solde-mt;  
    }  
    public float getSolde(){  
        return solde;  
    }  
}
```

# Throw:Utilisation de la classe compte



En faisant appel à la méthode retirer, le compilateur ne signale rien.

C'est une Exception non surveillée. On est pas obligé de la traiter pour que le programme soit compilé .

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2); // Le compilateur ne signale rien
}}
```

# Exemple : Générer, Relancer ou Jeter une exception surveillée de type Exception



```
public class Compte {  
    private int code;  
    private float solde;  
    public void verser(float mt){  
        solde=solde+mt;  
    }  
    public void retirer(float mt) {  
        if(mt>solde) throw new Exception("Solde Insuffisant");  
        solde=solde-mt;  
    }  
    public float getSolde(){  
        return solde;  
    }  
}
```

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

# Exemple : Générer, Relancer ou Jeter une exception surveillée de type Exception



```
public class Application {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Compte cp=new Compte();  
        Scanner clavier=new Scanner(System.in);  
        System.out.print("Montant à verser:");  
        float mt1=clavier.nextFloat();  
        cp.verser(mt1);  
        System.out.println("Solde Actuel:"+cp.getSolde());  
        System.out.print("Montant à retirer:");  
        float mt2=clavier.nextFloat();  
        try {  
            cp.retirer(mt2);  
        } catch (Exception e) {  
            System.out.println("erreur"+e.getMessage());  
        }  
    }  
}  
Montant à verser:200  
Solde Actuel:400.0  
Montant à retirer:500  
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
      Unhandled exception type Exception  
  
      at Compte.Compte.retirer(Compte.java:11)  
      at Compte.Application.main(Application.java:18)
```

# Redéclenchement d'une exception : throw: traiter l'exception



Soit nous interceptons et traitons l'exception Exception à l'intérieur du bloc englobant la méthode retirer() qui a lancé l'exception, auquel cas il est obligatoire de signaler au compilateur que cette méthode retirer() lance et propage une Exception non traitée.

```
public class Compte {  
    private int code;  
    private float solde;  
    public void verser(float mt){  
        solde=solde+mt;  
    }  
    public void retirer(float mt) throws Exception{  
        if(mt>solde) throw new Exception("Solde Insuffisant");  
        solde=solde-mt;  
    }  
    public float getSolde(){  
        return solde;  
    }  
}
```

signaler l'exception susceptible d'être propagée

# Exécution de l'exemple



## Scénario 1

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    try {
        cp.retirer(mt2);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("Solde Final="+cp.getSolde());
}}
```

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:2000  
Solde Final=3000.0

## Scénario 2

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:7000  
Solde Insuffisant  
Solde Final=5000.0

→ Interception de l'exception dans le bloc englobant

# Exécution de l'exemple



```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
    public static void main(String[] args) {
        Compte cp=new Compte();
        Scanner clavier=new Scanner(System.in);
        System.out.print("Montant à verser:");
        float mt1=clavier.nextFloat();
        cp.verser(mt1);
        System.out.println("Solde Actuel:"+cp.getSolde());
        System.out.print("Montant à retirer:");
        float mt2=clavier.nextFloat();
        try {
            cp.retirer(mt2);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Solde Final="+cp.getSolde());
    }
}
```

## Scénario 1

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:2000  
Solde Final=3000.0

## Scénario 2

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:7000  
Solde Insuffisant  
Solde Final=5000.0

## VI Exception personnalisée



Pour une exception personnalisée, le mode d'action est strictement identique, il vous faut seulement auparavant créer une nouvelle classe **héritant obligatoirement de la classe Exception** ou de n'importe laquelle de ses sous-classes.

Reprenons le programme précédent et créons une classe d'exception que nous nommerons **SoldeInsuffisantException** héritant de la classe Exception puis exécutons ce programme :

```
public class SoldeInsuffisantException extends Exception {  
    public SoldeInsuffisantException(String message) {  
        super(message);  
    }  
}
```

# Exception personnalisée



```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt) throws SoldeInsuffisantException{
        if(mt>solde) throw new SoldeInsuffisantException("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

```
import java.util.Scanner;
import metier.Compte;
import metier.SoldeInsuffisantException;
public class Application {
    public static void main(String[] args) {
        Compte cp=new Compte();
        Scanner clavier=new Scanner(System.in);
        System.out.print("Montant à verser:");
        float mt1=clavier.nextFloat();
        cp.verser(mt1);
        System.out.println("Solde Actuel:"+cp.getSolde());
        System.out.print("Montant à retirer:");
        float mt2=clavier.nextFloat();
        try {
            cp.retirer(mt2);
        } catch (SoldeInsuffisantException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Solde Final="+cp.getSolde());
    }
}
```

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```