



Les interfaces

Héritage simple ou multiple



- z Java n'autorise que l'héritage simple de classes (pas d'héritage multiple)
- z L'héritage multiple est remplacé par la notion d'interface
- z La notion d'interface permet de définir des comportements abstraits ou génériques aux classes

Les interfaces



- z Si le programmeur veut s'assurer qu'une certaine catégorie de classes (pas forcément reliées par des relations de généralisation / spécialisation) implémente un ensemble de méthodes, il peut regrouper les déclarations de ces méthodes dans une interface.
- z De telles classes pourront ainsi être manipulées de manière identique.

Les interfaces



- z Une interface correspond à une classe abstraite où toutes les méthodes sont abstraites
 - y on ne peut pas instancier une interface (pas de new)
- z Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une seule classe
 - y elle s'engage alors à fournir une implémentation pour toutes les méthodes définies dans l'interface
- z Une interface peut hériter (extends) d'une autre interface
 - y les interfaces forment une hiérarchie séparée de celle des classes : hiérarchie de comportements

Les interfaces



- z Une interface peut-être vue comme une classe abstraite dont toutes les méthodes sont abstraites, mais le mot clé `abstract` n'apparaît pas.
- z Dans la pratique, dans le code, une interface ne peut contenir que des attributs constants (`static final`) et des méthodes publiques non implémentées.

Les interfaces : exemple

```
interface Conduisible
{
    void demarrerMoteur();
    void couperMoteur();
    void tourner(float angle);
}

class Voiture implements Conduisible
{
    // ...
    void demarrerMoteur(); {...}
    void couperMoteur() {...}
    void tourner(float angle){...}
}

class TondeuseGazon implements Conduisible
{
    // ...
    void demarrerMoteur(); {...}
    void couperMoteur() {...}
    void tourner(float angle){...}
}
```

Les interfaces : exemple

```
// ...
```

```
Voiture maVoiture = new Voiture();  
TondeuseGazon maTondeuse = new TondeuseGazon();  
Conduisible vehicule;  
Boolean weekEnd;
```

```
// ...
```

```
if(weekEnd == true)  
{  
    vehicule = maTondeuse;  
}  
else  
{  
    vehicule = maVoiture;  
}  
vehicule.demarrerMoteur();  
vehicule.tourner(90.0F);  
vehicule.couperMoteur();
```

```
// ...
```



Les exceptions

Introduction



- z A la compilation, le compilateur cherche à éliminer le plus d 'erreurs possible, en particulier les erreurs de syntaxe
- z Des erreurs peuvent néanmoins survenir lors de l 'exécution du programme :
 - y elles dépendent du « contexte » d 'exécution (état des variables, accessibilité des périphériques, etc.)
 - y elles ne sont pas identifiées par le compilateur (en amont du processus d 'exécution)
 - y tout bon programmeur doit envisager et gérer les cas (valeurs des variables incohérentes, etc.) qui sont susceptibles de créer une erreur d 'exécution

Principe



- z Les exceptions sont un moyen pratique, fourni par Java, de faire le contrôle d'erreurs pouvant surgir lors de l'exécution.
- z Une exception est un signal qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution
- z Deux solutions alors :
 - y laisser le programme se terminer avec une erreur
 - y essayer, malgré l'exception, de continuer l'exécution normale
- z Lancer ou lever une exception consiste à signaler quelque chose d'exceptionnel
- z Capturer l'exception consiste à signaler qu'on va la traiter

Quelques exceptions préexistant dans Java



- z Division par zéro pour les entiers :
ArithmeticException
- z Déréférencement d'une référence nulle :
NullPointerException
- z Tentative de forçage de type illégale :
ClassCastException
- z Tentative de création d'un tableau de taille négative : *NegativeArraySizeException*
- z Dépassement de limite d'un tableau :
ArrayIndexOutOfBoundsException

Des exceptions pour écrire du code fiable



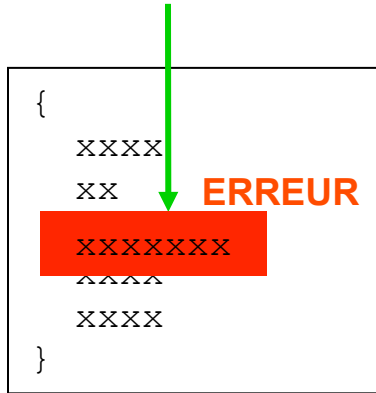
- z Java exige qu'une méthode susceptible de lever une exception (hormis les **Error** et les **RuntimeException**) indique quelle doit être l'action à réaliser.
 - y Sinon, il y a erreur de compilation.
- z Le programmeur a le choix entre :
 - y écrire un bloc **try** / **catch** pour traiter l'exception,
 - y laisser remonter l'exception au bloc appelant grâce à un **throws**.
- z C'est ce qu'on appelle : "Déclarer ou traiter".

Capter les exceptions



```
{  
    xxxx  
    xx  
    xxxxxxxx  
    xxxx  
    xxxx  
}
```

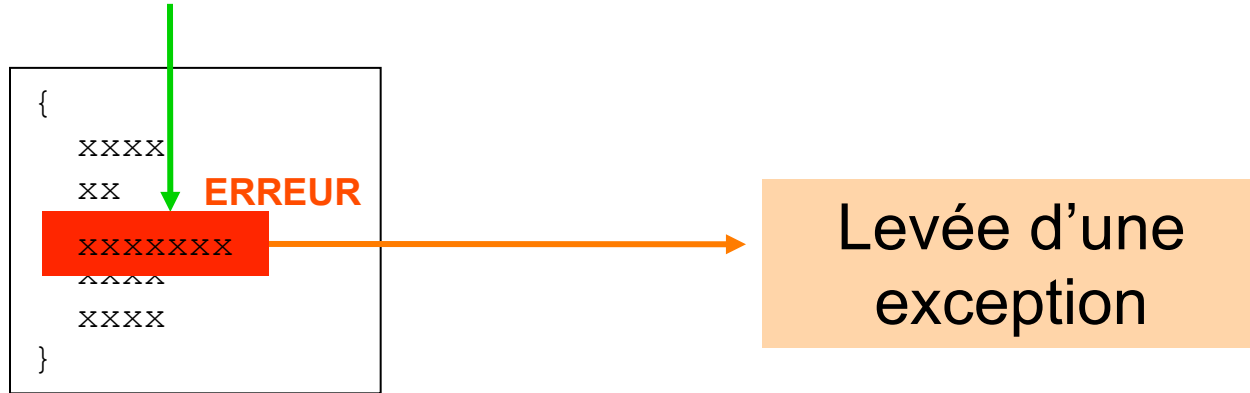
Capter les exceptions



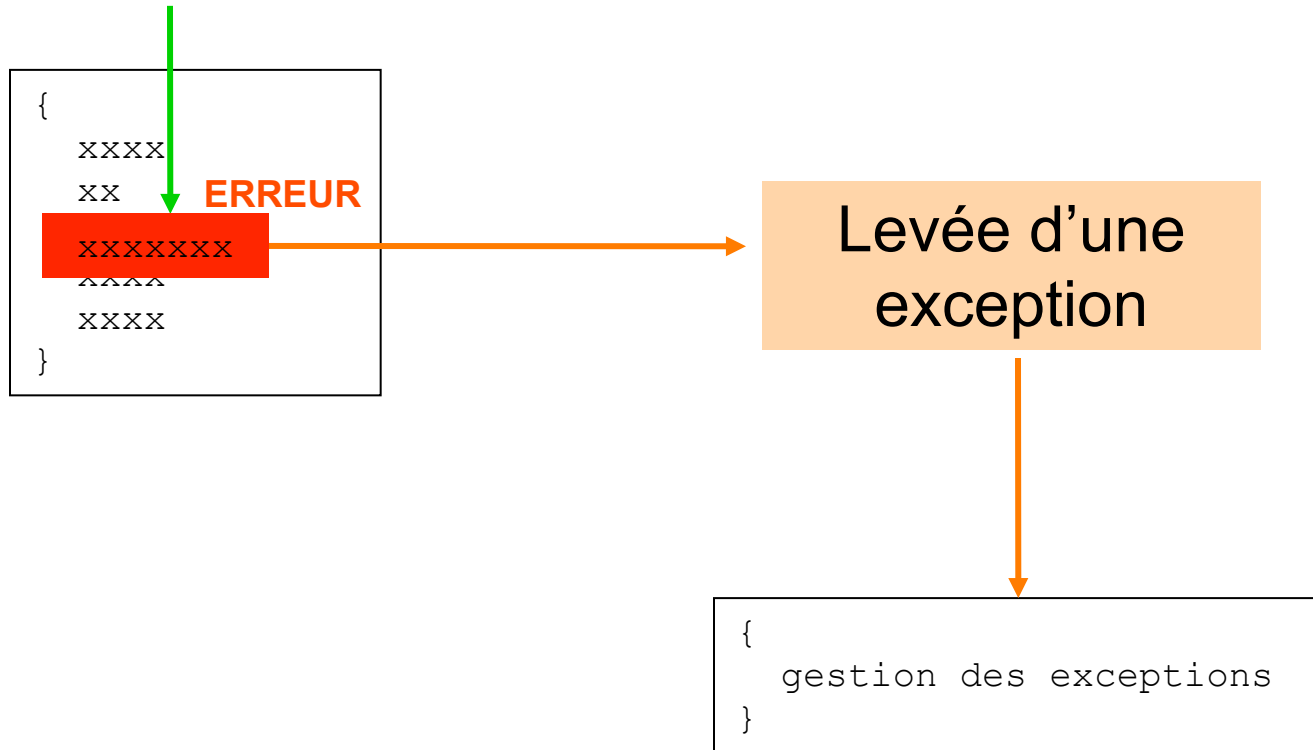
The diagram illustrates a code block with a try-catch structure. A green arrow points from the top to the start of the code block. The code block is enclosed in curly braces and contains several lines of placeholder text (xxxx, xx, xxxxxxxx, xxxx, xxxx). The line containing 'xxxxxxx' is highlighted with a red background, and the word 'ERREUR' is written in red next to it, indicating an exception has occurred.

```
{  
  xxxx  
  xx  
  xxxxxxxx  
  xxxx  
  xxxx  
}
```

Capter les exceptions



Capter les exceptions



Générer (lever) une exception

- Pour générer explicitement une exception, on utilise l'instruction **throw**:

```
throw exception_object ;
```

- L'expression qui suit l'instruction **throw** doit être un objet qui représente une exception (un objet de type **Throwable**).
- Lors de la création de l'objet exception, on peut généralement lui associer un **message** (**String**) qui décrit l'événement.

```
public static long factorial(int x) throws Exception {  
    long result = 1;  
    if (x < 0)  
        throw new Exception("x doit être >= 0");  
    while (x > 1) {  
        result *= x;  
        x--;  
    }  
    return result;  
}
```

Capter les exceptions

Z Pour capter une exception, il faut placer les instructions à surveiller dans un bloc `try`

Z Syntaxe :

```
try
{
    bloc d'instructions
}
catch(exception-type identifiant)
{
    bloc d'instructions (gestion de l'erreur)
}
catch(exception-type identifiant)
{
    bloc d'instructions (gestion de l'erreur)
}
finally
{
    bloc d'instructions
}
```

Capter les exceptions



- z Le bloc `try` est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- z Dans ce dernier cas, les clauses `catch` sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exceptions (ou une superclasse).
- z Les clauses `catch` doivent donc traiter les exceptions de la plus spécifique à la plus générale.
 - y La présence d'une clause `catch` qui intercepte une classe d'exceptions avant une clause qui intercepte une sous-classe d'exceptions déclenche une erreur de compilation.
- z Si une clause `catch` convenant à cette exception a été trouvée et le bloc exécuté, l'exécution du programme reprend son cours.

Capter les exceptions



- z Si elles ne sont pas immédiatement capturées par un bloc `catch`, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- z Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode `main()`, ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- z L'interpréteur Java affiche un message identifiant :
 - y l'exception,
 - y la méthode qui l'a causée,
 - y la ligne correspondante dans le fichier.

Capter les exceptions



Z Exemple :

```
try
{
    instructions...
}
catch(java.io.IOException ioe)
{
    System.err.println(« Probleme d 'e/s : « +ioe);
}
catch(Exception e)
{
    System.err.println(« Probleme : « +e);
}
```

La clause finally

- z L'écriture d'un bloc `finally` permet au programmeur de définir un ensemble d'instructions qui est toujours exécuté, que l'exception soit levée ou non, capturée ou non.
- z Si une clause `finally` est présente dans une instruction `try`, son code est exécuté après les instructions contenues dans le bloc `try`, quelle que soit la façon dont le traitement s'est achevé, que ce soit normalement, par une exception, ou par une instruction de contrôle comme `return` ou `break`.
- z La tâche habituelle assignée à une clause `finally` est de réinitialiser l'état interne de la mémoire, ou de libérer les ressources qui ne sont pas des objets comme, par exemple, les fichiers ouverts qui sont stockés dans des variables locales

throws



- z Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur ajoute le mot réservé `throws` à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

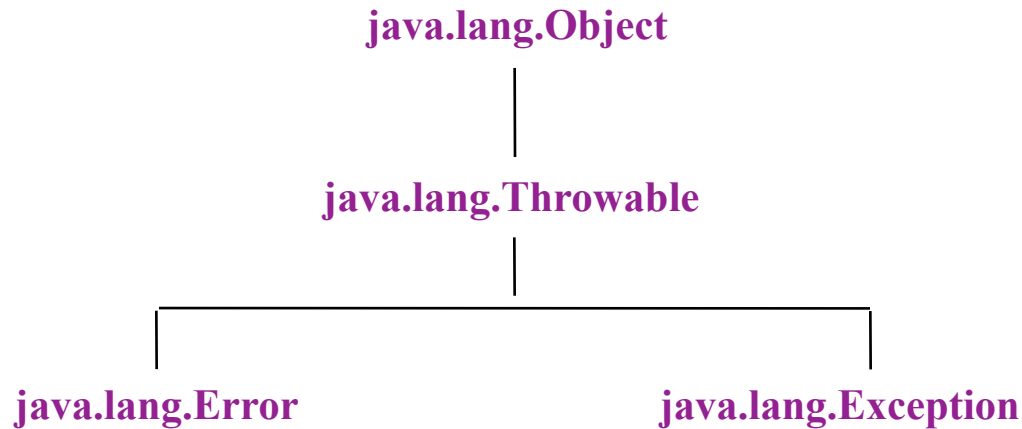
```
public void uneMethode() throws IOException
{
    // ne traite pas l'exception
    IOException
    // mais est susceptible de la générer
}
```

throws

- z Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.
- z La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée.
- z Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,).
- z Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans les méthodes qu'elle appelle (et ceci récursivement).

Les objets Exception

- Z** Dans Java, il existe deux types d'exceptions, représentées par deux classes de l'API Java.



Les erreurs sont graves et Java recommande de ne pas les corriger.

Les exceptions contiennent celles prédéfinies par Java ... et les vôtres.

Les objets `Exception`



- z La classe `Throwable` définit un message de type `String` qui est hérité par toutes les classes d'exception.
- z Ce champ est utilisé pour stocker le message décrivant l'exception.
- z Il est positionné en passant un argument au constructeur.
- z Ce message peut être récupéré par la méthode `getMessage()`.

Les objets Exception



z Exemple :

```
public class MonException extends Exception
{
    public MonException()
    {
        super();
    }
    public MonException(String s)
    {
        super(s);
    }
}
```

Levée d'exceptions



- z Le programmeur peut lever ses propres exceptions à l'aide du mot réservé `throw`.
- z `throw` prend en paramètre un objet instance de `Throwable` ou d'une de ses sous-classes.
- z Les objets exception sont souvent alloués dans l'instruction même qui assure leur lancement.

```
throw new MonException("Mon exception s'est produite !!!");
```

Exemple

```
public class PropEx {
    public static void main(String[] args) {
        System.out.println("main starts");
        b(5);
        b(0);
        System.out.println("main ends");
    }
    //----- // b //-----
    public static void b(int i) {
        System.out.println("b starts");
        try {
            System.out.println("b step 1");
            c(i);
            System.out.println("b step 2");
        }
        catch (Exception e) {
            System.out.println("b catches " + e);
            System.out.println("b ends");
        }
    }
}
```

```
//-----// c //-----
public static void c(int i) throws Exception
{
    System.out.println("c starts");
    d(i);
    System.out.println("c ends");
}

//---- //d //-----
public static void d(int i) throws Exception
{
    System.out.println("d starts");
    int a=10/i; // Cette instruction peut générer une
               exception
    System.out.println("d ends");
}
}
```