Classes internes ,exception , threads

Le contenu de ce thème :

Les classes internes

Les exceptions

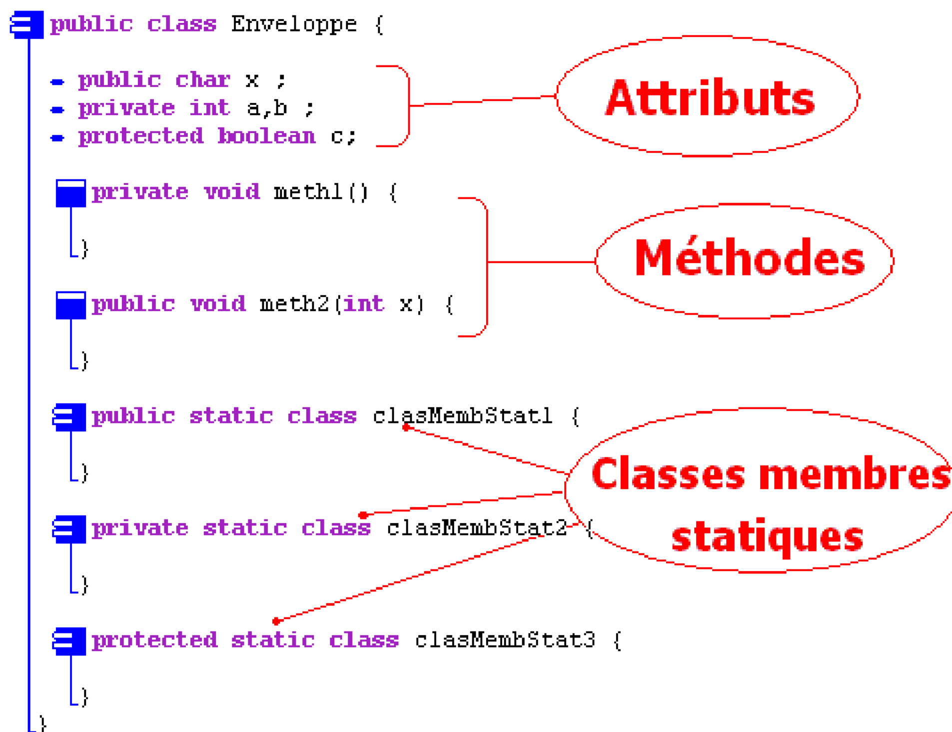
Le multi-threading

Les classes internes Java2

Depuis la version Java 2, le langage java possède quatre variétés supplémentaires de classes, que nous regroupons sous le vocable général de **classes internes**.

Les classes membres statiques

Exemple de classes membres statiques

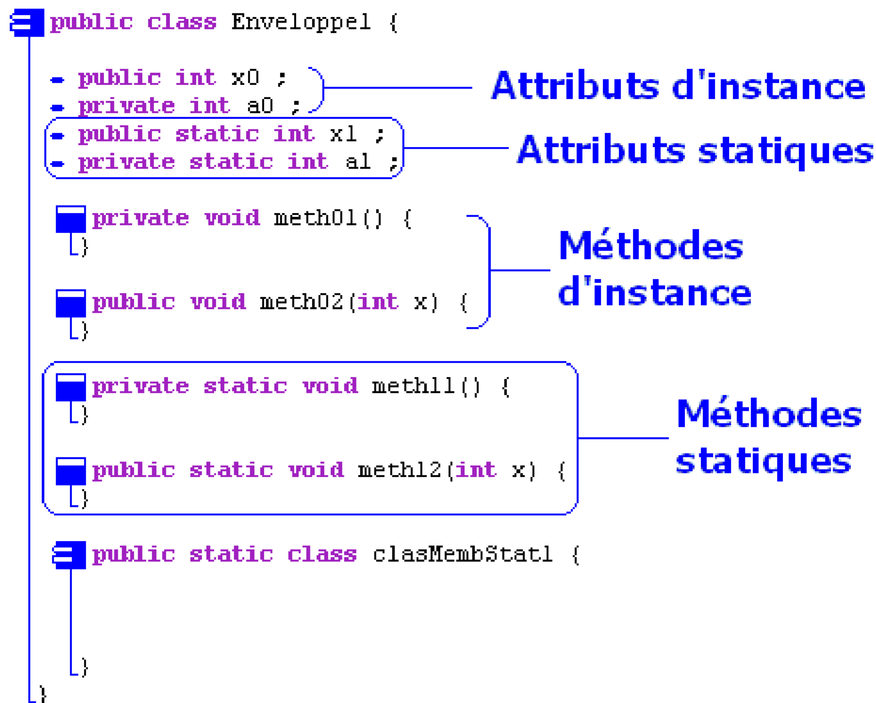


Définition des classes membres statiques

<table>
<tr>
<td>Une classe membre statique est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante), puis qualifiée par le modificateur static.
❑ Une classe membre statique est instanciable.
❑ Une classe membre statique ne peut pas être associée à un objet instancié de la classe englobante .</td>
<td>Syntaxe :

public class Enveloppe {
< membres genre attributs >
< membres genre méthodes >
< membres genre classes membres statiques >
}</td>
</tr>
</table>

Une classe membre statique est analogue aux autres membres statiques de la classe dans laquelle elle est déclarée (notée Enveloppe ici) :



On peut dans une autre classe, instancier un objet de classe statique, à condition d'utiliser le qualificateur de visibilité qu'est le nom de la classe englobante afin d'accéder à la classe interne. Une classe membre statique se comporte en fait comme une classe ordinaire relativement à l'instanciation :

Soit par exemple, la classe **Autre** souhaite déclarer et instancier un objet Obj0 de classe public clasMembStat1 :

```
class Autre {
        Enveloppe1.clasMembStat1 Obj0 = new Enveloppe1.clasMembStat1( ) ;
        .......
}
```

Soit en plus à instancier un objet local Obj1 dans une méthode de la classe **Autre** :

```
class Autre{
      Enveloppe1.clasMembStat1 Obj0 = new Enveloppe1.clasMembStat1() ;
       void meth( ){
        Enveloppe1.clasMembStat1 Obj1 = new Enveloppe1.clasMembStat1() ;
      }
}
```

Caractérisation d'une classe membre statique

Une classe membre statique accède à **tous les membres statiques** de sa classe englobante qu'ils soient **publics** ou **privés**, sans nécessiter d'utiliser le nom de la classe englobante pour accéder aux membres (raccourcis d'écriture) :

```
public class Enveloppe1 {

   public int x0 ;
   private int a0 ;
   public static int x1 ;
   private static int a1 ;

   private void meth01() {
   }

   public void meth02(int x) {
   }

   private static void meth11() {
   }

   public static void meth12(int x) {
   }

   public static class clasMembStat1 {

   }
}
```

accède à tous les membres statiques

Une classe membre statique

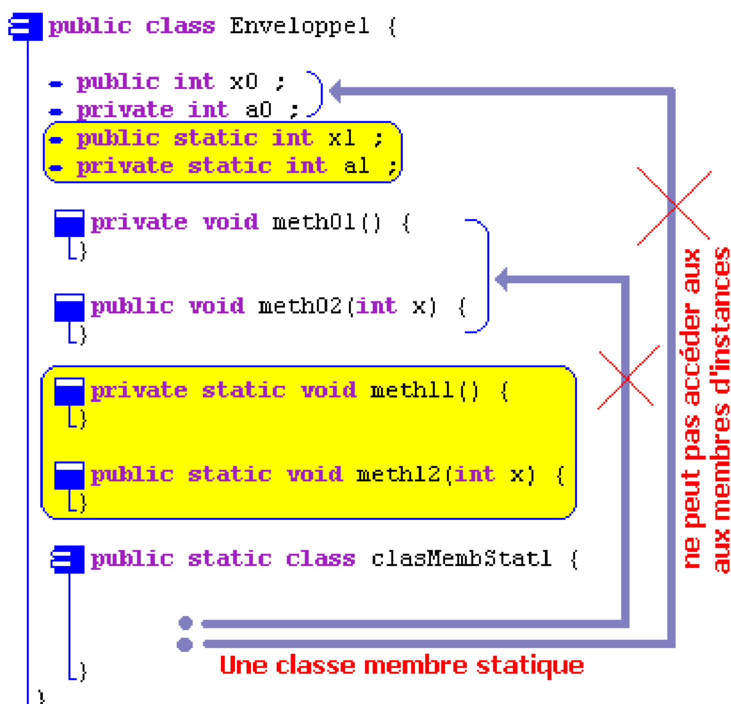
Exemple :
Ci-dessous la classe membre statique clasMembStat1 contient une méthode "**public void** meth( )", qui invoque la méthode de classe(statique) **void** meth12 de la classe englobante Enveloppe1, en lui passant un paramètre effectif statique **int** a1. Nous écrivons 4 appels de la méthode meth12( ) :

```
public static class clasMembStat1 {

        public void meth( ){
         meth12(a1);
          Enveloppe1.meth12(a1);
          meth12(Enveloppe1.a1);
         Enveloppe1.meth12(Enveloppe1.a1);
        }
}
```

Nous notons que les 4 appels de méthodes sont strictement équivalents, en effet ils différent uniquement par l'utilisation ou la non utilisation de raccourcis d'écriture.

- Le quatrième appel : "Enveloppe1.meth12(Enveloppe1.a1)" est celui qui utilise le nom de la classe englobante pour accéder aux membres statiques comme le prévoit la syntaxe générale.
- Le premier appel : "meth12(a1)" est celui qui utilise la syntaxe particulière aux classes membres statiques qui autorise des raccourcis d'écriture.

Une classe membre statique n'a pas accès aux membres d'instance de la classe englobante :



Remarque importante :

Un objet de **classe interne statique** ne comporte pas de référence à l'objet de classe externe qui l'a créé.

Remarque Interface statique :

Une interface peut être déclarée en interface interne statique comme une classe membre statique en utilisant comme pour une classe membre statique le mot clef **static**.

Syntaxe :

public class Enveloppe **{**
< membres genre attributs >
< membres genre méthodes >
< membres genre classes membres statiques >
< membres genre interfaces statiques >
}

***Exemple* :**

public class Enveloppe {
public static Interface Interfstat {
....
}
}

Les classes membres

Définition des classes membres

Une classe membre est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante).

- Une classe membre est instanciable.
- Une classe membre est associée à un objet instancié de la classe englobante .

Syntaxe :

public class Enveloppe **{**
< membres genre attributs >
< membres genre méthodes >
< membres genre classes membres statiques >
< membres genre classes membres >
}

Une classe membre se déclare **d'**une manière identique **à une classe membre statique** et aux autres membres de la classe englobante (notée Enveloppe2 ici) :

```
public class Enveloppe2 {

  public int x0 ;
  private int a0 ;
  public static int x1 ;
  private static int a1 ;

  private void meth01() {
  }

  public void meth02(int x) {
  }

  private static void meth11() {
  }

  public static void meth12(int x) {
  }

  public class classeMembre{


  }
}
```

Attributs d'instance

Attributs statiques

Méthodes d'instance

Méthodes statiques

On peut dans une autre classe, instancier un objet de classe membre, à condition d'utiliser le qualificateur de visibilité qu'est le nom de la classe englobante afin d'accéder à la classe interne. Une classe membre se comporte en fait comme une classe ordinaire relativement à l'instanciation :

Soit par exemple, la classe **Autre** souhaite déclarer et instancier un objet Obj0 de classe public classeMembre :

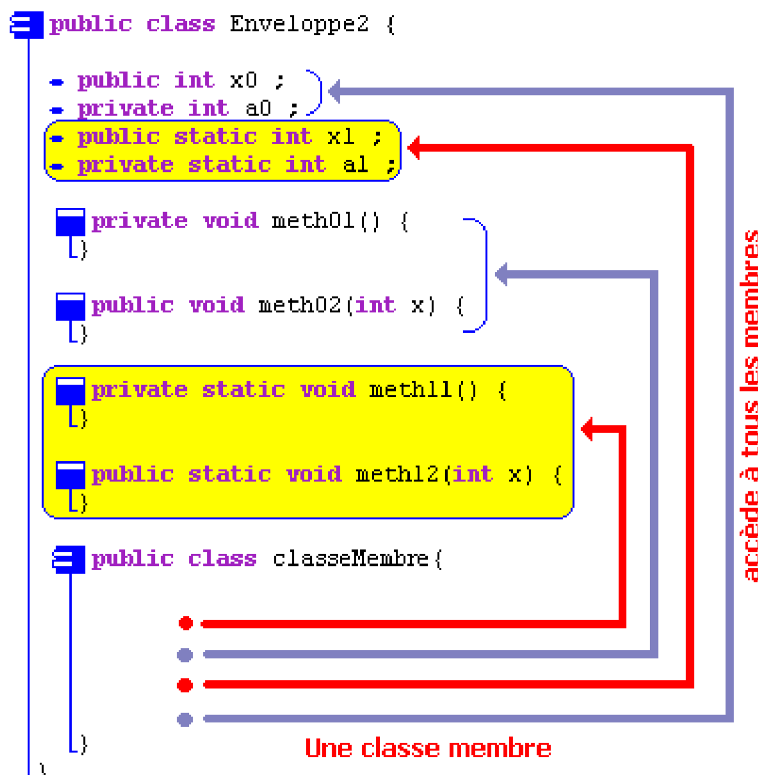
class Autre **{**
Enveloppe1.classeMembre Obj0 = **new** Enveloppe1.classeMembre( ) ;
....

```
}
```

Soit en plus à instancier un objet local Obj1 dans une méthode de la classe **Autre** :

```
class Autre{
    Enveloppe1.classeMembre Obj0 = new Enveloppe1.classeMembre() ;
     void meth( ){
       Enveloppe1.classeMembre Obj1 = new Enveloppe1.classeMembre() ;
    }
}
```

Caractérisation d'une classe membre



Remarque importante :

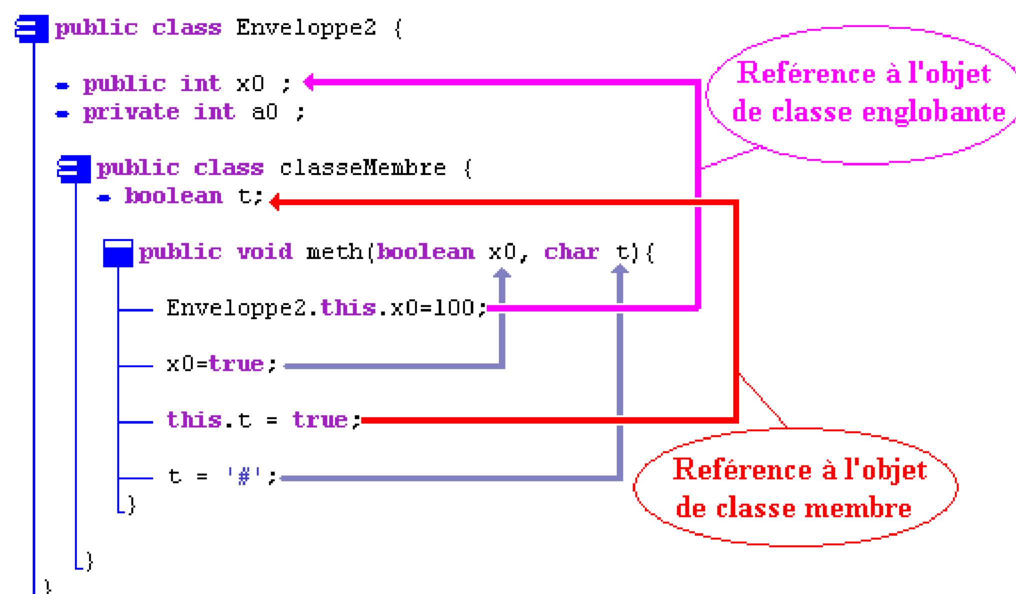
Les classes membres ont les mêmes fonctionalités syntaxiques que les classes membres statiques. La différence notable entre ces deux genres de classes se situe dans l'accès à l'instance d'objet de classe englobante : Un objet de **classe interne non statique comporte une référence à l'objet de classe externe** qui l'a créé, alors qu'il n'en est rien pour une classe membre statique. L'accès a lieu à travers le mot clef **this** qualifié par le nom de la classe englobante : "ClasseEnveloppe.**this**" .

Exemple d'accès à la référence de l'objet externe :

Question : comment, dans la classe interne membre classeMembre, ci-dessous accéder aux différents champs masqués x0 et t :

```
public class Enveloppe2 {

    public int x0 ;
    private int a0 ;
     ........
     public class classeMembre {
      boolean t ;
      void meth( boolean x0, char t ) {
       /*
          accéder au membre x0 de Enveloppe2 ;
          accéder au  paramètre formel x0 de meth ;
          accéder au membre t de classeMembre ;
          accéder au  paramètre formel t de meth ;
          */
      }
     ........
    }
 }
```

Réponse : en utilisant le mot clef **this** à la fois pour obtenir la référence de l'objet de classe classeMembre et Enveloppe2.**this** pour obtenir la reférence sur l'objet externe de classe englobante :



Remarque Interface non déclarée static :

Une interface ne peut pas être déclarée en interface interne non statique comme une classe membre ordinaire car on ne peut pas instancier d'objetà partir d'une interface. Si vous oubliez le mot clef **static**, le compilateur java le "rajoutera" automatiquement et l'interface sera considérée comme statique !

Les classes locales

Définition des classes locales

Une classe locale est déclarée au sein d'un **bloc de code** Java.généralement dans le corps d'une méthode d'une autre classe qui la contient (nommée classe englobante). Elle adopte un schéma de visibilité gross-mode semblable à celui d'une variable locale. ❑ Une classe locale est instanciable comme une classe membre. ❑ Une classe locale est associée à un objet instancié de la classe englobante.	***Syntaxe*** : **public class** Enveloppe **{** < membres genre attributs > < membres genre méthodes > < membres genre classes membres statiques > < membres genre classes membres > < membres genre interfaces statiques > < membres genre classes locales > **}**

Une classe locale peut être déclarée au sein d'un quelconque bloc de code Java. Ci-dessous 3 classes locales déclarées chacune dans 3 blocs de code différents :

```
public class Enveloppe3 {

  public int x0 ;
  private int a0 ;

  public void meth(char x0 ) {

    class classeLocale1 {
      //......
    }

    for(int i=0;i<10;i++) {
      class classeLocale2 {
        //......
      }

      classeLocale2 Obj2;

      if (i==5) {

        Obj2 = new classeLocale2();

        class classeLocale3 {
          //......
        }
      }
    }
  }
}
```

Caractérisation d'une classe locale

Une classe locale n'est visible et utilisable qu'au sein d'un **bloc de code** Java.où

elle a été déclarée.

> Une classe locale peut utiliser des variables (locales, d'instances ou paramètres) visibles dans le bloc ou elle est déclarée à la condition impérative que ces variables aient été déclarées en mode **final**.
>
> Une classe locale peut utiliser des variables d'instances comme une classe membre.

A la lumière de la définition précédente corrigeons le code source de la classe ci-dessous, dans laquelle le compilateur Java signale 3 erreurs :

```
public class Enveloppe3 {
    public int x0 = 5 ;
    private int a0 ;

     public void meth(char x0 ) {
       int x1=100;
       class classeLocale1 {
         int j = x1; // le int x1 local à meth( )
        int c =  x0;  // le int x0 de Enveloppe3
       }
       for ( int i = 0;  i<10; i++) {
        int k = i;
        class classeLocale2 {
          int j = k + x1 + x0; // le k local au for
        }
       }
    }
  }
```

Après compilation, le compilateur java signale des accès érronés à des variables non final :

C:\j2sdk1.4.2\bin\javac - 3 errors :

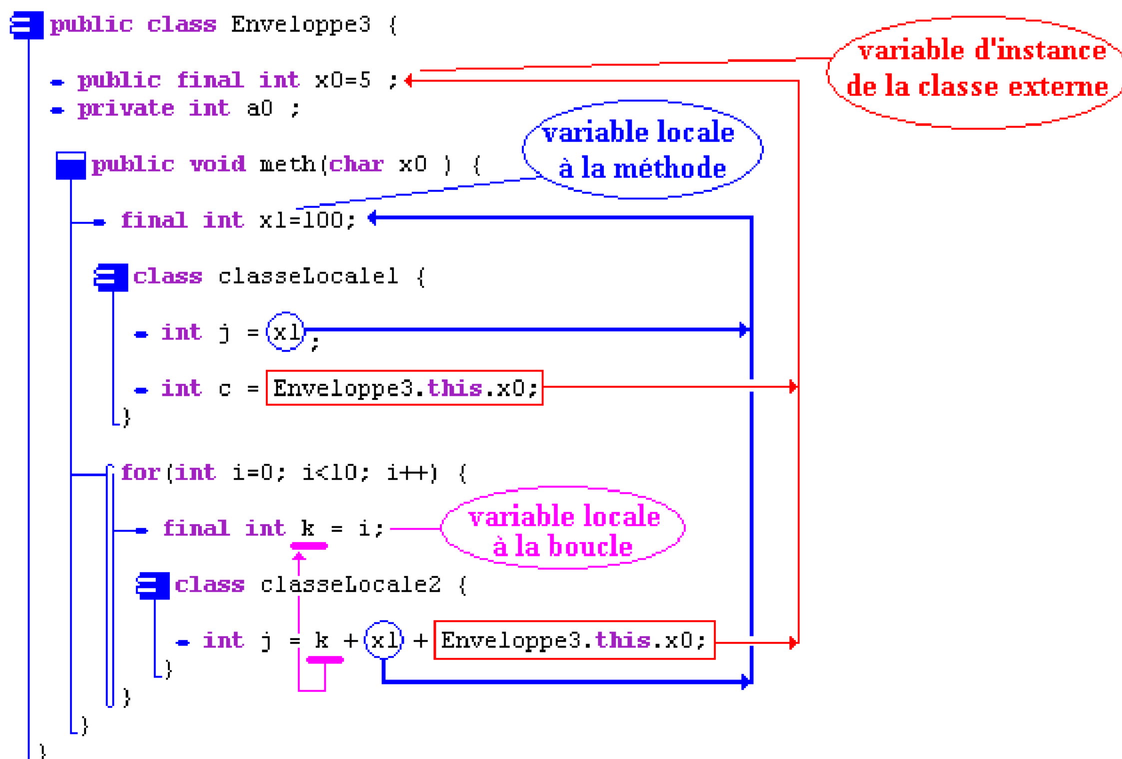
```
local variable x1 is accessed from within inner class; needs to be declared final
int j = x1;

local variable x0 is accessed from within inner class; needs to be declared final
int c =  x0;

local variable k is accessed from within inner class; needs to be declared final
int j = k + x1 + x0;
```

On propose une première correction :

```
public class Enveloppe3 {
    public int x0=5 ;
    private int a0 ;

     public void meth ( final char x0 ) {
      final int x1=100;
       class classeLocale1 {
        int j = x1;
        int c =  x0;
```

```
      }
      for ( int i = 0;  i<10; i++) {
        final int k = i;
         class classeLocale2 {
           int j = k + x1 + x0;
         }
      }
    }
  }
```

Cette fois-ci le compilateur accepte le code source comme correct. Toutefois cette solution ne correspond pas à la définition de l'instruction "**int** j = k + x1 + x0; " dans laquelle la variable x0 doit être la variable d'instance de la classe externe Enveloppe3. Ici c'est le paramètre formel "**final** **char** x0" qui est utilisé. Si l'on veut accéder à la variable d'instance, il faut la mettre en mode final et qualifier l'identificateur x0 par le nom de la future instance de classe Enveloppe3, soit "Enveloppe3.**this**" :



Remarque :

Une classe locale **ne peut pas** être qualifiée en **public**, **private**, **protected** ou **static**.

Les classes anonymes

Définition des classes anonymes

Une classe **anonyme** est une **classe locale qui ne porte pas de nom**.

Une classe anonyme possède toutesles propriétés d'une classe locale.

Comme une classe locale n'a pas de nom vous ne pouvez pas définir un constructeur.

Syntaxe :

public class Enveloppe **{**
< membres genre attributs >
< membres genre méthodes >
< membres genre classes membres statiques >
< membres genre classes membres >
< membres genre interfaces statiques >
< membres genre classes locales >
< membres genre classes anonymes >
}

Une classe anonyme est instanciée immédiatement dans sa déclaration selon une syntaxe spécifique :

new <identif de classe> **(** <liste de paramètes de constructions>**) {** <corps de la classe>**}**

Soit l'exemple ci-dessous , comportant 3 classes Anonyme, Enveloppe4, Utilise :

```
class Anonyme{
  public int a;

  Anonyme(int x){
    a=x*10;
    methA(x);
    methB();
  }

  public void methA(int x){
    a+=2*x;
  }

  public void methB(){}
}

public class Enveloppe4 {

  public void meth(Anonyme x){
  }
}
```

```
class Utilise{
  Enveloppe4 Obj = new Enveloppe4() ;

  void meth(){

    Obj.meth(
         new Anonyme(8){        });

    Obj.meth(
         new Anonyme(12){
           public void methA(int x){
             super.methA(x);
             a -=10;
           }
           public void methB(int x){
             System.out.println("a = "+a);
           }
         });

    Obj.meth(
         new Anonyme(-54){
           public void methB(int x){
             System.out.println("a = "+a);
           }
         });
  }
}
```

La classe Enveloppe4 déclare une méthode public possédant un paramètre formel de type Anonyme :

```
public class Enveloppe4 {

   public void meth(Anonyme x){
   }
}
```

La classe Anonyme déclare un champ entier public a, un constructeur et deux méthodes public :

```
class Anonyme {
    public int a;
  // constructeur:
     Anonyme (int x){
       a = x*10;
       methA(x);
       methB();
     }
     public void methA(int x){
       a += 2*x ;    }

     public void methB( ){ }
  }
```

La classe Utilise déclare un champ objet de type Enveloppe4 et une méthode qui permet la création de 3 classes internes anonymes dérivant chacune de la classe Anonyme :

```
class Utilise{
    Enveloppe4 Obj = new Enveloppe4() ;

     void meth(){
      /* création d'une première instance anonyme de classe dérivant de Anonyme
         avec comme valeur de construction 8.
      */
       Obj.meth( new Anonyme (8){ });

      /* création d'une seconde instance anonyme de classe dérivant de Anonyme
         avec comme valeur de construction 12, redéfinition de la méthode methA et
         redéfinition de la méthodee methB :
      */
       Obj.meth(new Anonyme(12){
              public void methA(int x){
                super.methA(x);
                 a -= 10;
              }
              public void methB(int x){
                 System.out.println("a = "+a);
              }
            });
      /* création d'une troisième instance anonyme de classe dérivant de Anonyme
         avec comme valeur de construction -54 et redéfinition de la seule méthode methA :
      */
       Obj.meth(new Anonyme(-54){
              public void methB(int x){
                 System.out.println("a = "+a);
              }
            });
     }
  }
```

Caractérisation d'une classe anonyme

- Une classe anonyme étend concrètement une classe déjà existante abstraite ou non, ou bien implémente concrètement une interface.
- Une classe anonyme sert lorsque l'on a besoin d'une classe pour un seul usage unique, elle est définie et instanciée là où elle doit être utilisée.
- L'exemple le plus marquant d'utilisation de classe anonyme est l'instanciation d'un écouteur.

Voyons la différence d'écriture entre la version précédente de la classe Utilise avec 3 objets de classes anonymes et la réécriture avec des classes ordinaires :

On doit d'abord créer 3 classes dérivant de Anonyme :

```
class Anonyme1 extends Anonyme {
    Anonyme1(int x){
      super(x);
    }
}

class Anonyme2 extends Anonyme {
    Anonyme2(int x){
      super(x);
    }
    public void methA(int x){
      super.methA(x);
      a -= 10;
    }
    public void methB(int x){
      System.out.println("a = "+a);
    }
}

class Anonyme3 extends Anonyme {
    Anonyme3(int x){
      super(x);
    }
    public void methB(int x){
      System.out.println("a = "+a);
    }
}
```

Puis enfin définir la classe Utilise, par exemple comme suit :

```
class Utilise{
    Enveloppe4 Obj = new Enveloppe4() ;

void meth() {
        Anonyme1 Obj1 = new Anonyme1(8);
        Anonyme2 Obj2 = new Anonyme2(12);
        Anonyme3 Obj3 = new Anonyme3(-54);
        Obj.meth( Obj1);
        Obj.meth( Obj2 );
        Obj.meth( Obj3 );
    }
  }
```

Vous pourrez comparer l'élégance et la briéveté du code utilisant les classes anonymes par rapport au code classique :

```
class Utilise{
    Enveloppe4 Obj = new Enveloppe4() ;

    void meth(){
        Obj.meth( new Anonyme(8){ });
        Obj.meth(new Anonyme(12){
                public void methA(int x){
                  super.methA(x);
                   a -= 10;
                }
                public void methB(int x){
                   System.out.println("a = "+a);
                }
              });
        Obj.meth(new Anonyme(-54){
                public void methB(int x){
                   System.out.println("a = "+a);
                }
              });
    }
  }
```

Les exceptions

Java2

Les exceptions : syntaxe, rôle, classes

Rappelons au lecteur que la sécurité d'une application peut être rendue instable par toute une série de facteurs :

Des problèmes liés au matériel : par exemple la perte subite d'une connection à un port, un disque défectueux... Des actions imprévues de l'utilisateur, entrainant par exemple une division par zéro... Des débordements de stockage dans les structures de données...

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue, lors d'actions sur les données. Le logiciel doit donc se " ***défendre*** " contre de tels incidents potentiels, nous nommerons cette démarche la programmation défensive !

Progammation défensive

La ***programmation défensive*** est une attitude de pensée consistant à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

En programmation défensive il est possible de protéger directement le code à l'aide de la notion d'exception. L'objectif principal est d'améliorer la qualité de " ***robustesse*** " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et efficace de logiciels robustes.

Rôle d'une exception

Une exception est chargée de signaler un comportement ***exceptionnel*** (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même.C'est le cas de Java qui intègre les exceptions comme une classe particulière: la classe **Exception**.Cette classe contient un nombre important de classes dérivées.

Comment agit une exception

Dès qu'une erreur se produit comme un manque de mémoire , un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " ***déclenche une exception*** ".

Comment gérer une exception dans un programme

Programme sans gestion de l'exception

Soit un programme Java contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;) dans la méthode meth() de la classe Action1, cette méthode est appelée dans la classe UseAction1 à travers un objet de classe Action1 :

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

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.");
  }
}
```

Lors de l'exécution, après avoir affiché les chaînes "**Début du programme**" et " **...Avant incident**", le programme s'arrête et la java machine signale une erreur. Voici ci-dessous l'affichage obtenu sur la console lors de l'exécution :

---- java UseAction1
Début du programme
  ...Avant incident
java.lang.ArithmeticException : / by zero
---- : Exception in thread "main"

Que s'est-il passé ?

La méthode main :
- a instancié un objet Obj de classe Action1,;
- a affiché sur la console la phrase "Début du programme",
- a invoqué la méthode meth() de l'objet Obj,
- a affiché sur la console la phrase " ...Avant incident",
- a exécuté l'instruction "x = 1/0;"

Dès que l'instruction "x = 1/0;" a été exécutée celle-ci a provoqué un incident. **En fait une exception de la classe ArithmeticException a été "levée" (un objet de cette classe a été instancié) par la Java machine**.

La Java machine a arrêté le programme immédiatement à cet endroit parce qu'elle n'a pas trouvé de code d'interception de cette exception qui a été automatiquement levée :

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

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

Nous allons voir comment intercepter (on dit aussi "attraper" - to catch) cette exception afin de faire réagir notre programme afin qu'il ne s'arrête pas brutalement.

Programme avec gestion de l'exception

Java possède une instruction de gestion des exceptions, qui permet d'intercepter des exceptions dérivant de la classe Exception :

try ... catch

Syntaxe minimale d'un tel gestionnaire :

try {

<lignes de code à protéger>

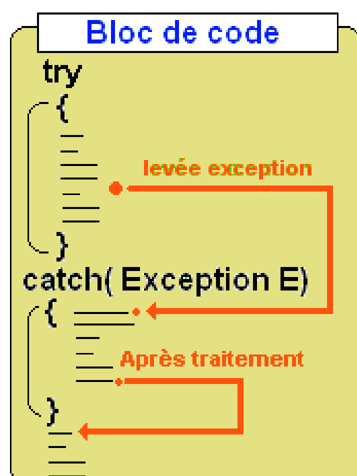
}catch (UneException E) **{**

<lignes de code réagissant à l'exception UneException >

}

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

Schéma du fonctionnement d'un tel gestionnaire :



Le gestionnaire d'exception "déroute" l'exécution du programme vers le bloc d'interception catch qui traite l'exception (exécute le code contenu dans le bloc catch), puis renvoie et continue l'exécution du programme vers le code situé après le gestionnaire lui-même.

Principe de fonctionnement de l'interception

Dès qu'une **exception est levée** (instanciée), la Java machine **stoppe immédiatement** l'exécution normale du programme à la **recherche d'un gestionnaire** d'exception susceptible d'intercepter (saisir) et traiter cette exception. Cette recherche s'effectue à partir du **bloc englobant** et se poursuit sur les blocs plus englobants si aucun gestionnaire de **cette exception** n'a été trouvé.

Soit le même programme Java que précédemment, contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;). Cette fois nous allons gérer l'incident grâce à un gestionnaire d'exception **try..catch** dans le bloc englobant immédiatement supérieur.

Programme sans traitement de l'incident :

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

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.");
  }
}
```

Programme avec traitement de l'incident par try...catch :

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

engendre une exception

```
class UseAction1{
   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 exception");
      }
      System.out.println("Fin du programme.");
   }
}
```

levée d'ArithmeticException

traitement, puis poursuite de l'exécution

Ci-dessous l'affichage obtenu sur la console lors de l'exécution de ce programme :

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

- Nous remarquons donc que la Java machine a donc bien exécuté le code d'interception situé dans le corps du "**catch** (ArithmeticException E){...}" et a poursuivi l'exécution normale après le gestionnaire.
- Le gestionnaire d'exception se situe dans la méthode main (code englobant) qui appelle la méthode meth() qui lève l'exception.

Interception d'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.

Ci-après nous montrons la syntaxe d'un tel gestionnaire qui fonctionne comme un selecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.

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 nième. Si l'exception

actuellement levée est d'un des types présents dans la liste des clauses le traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

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).

Exemple théorique :

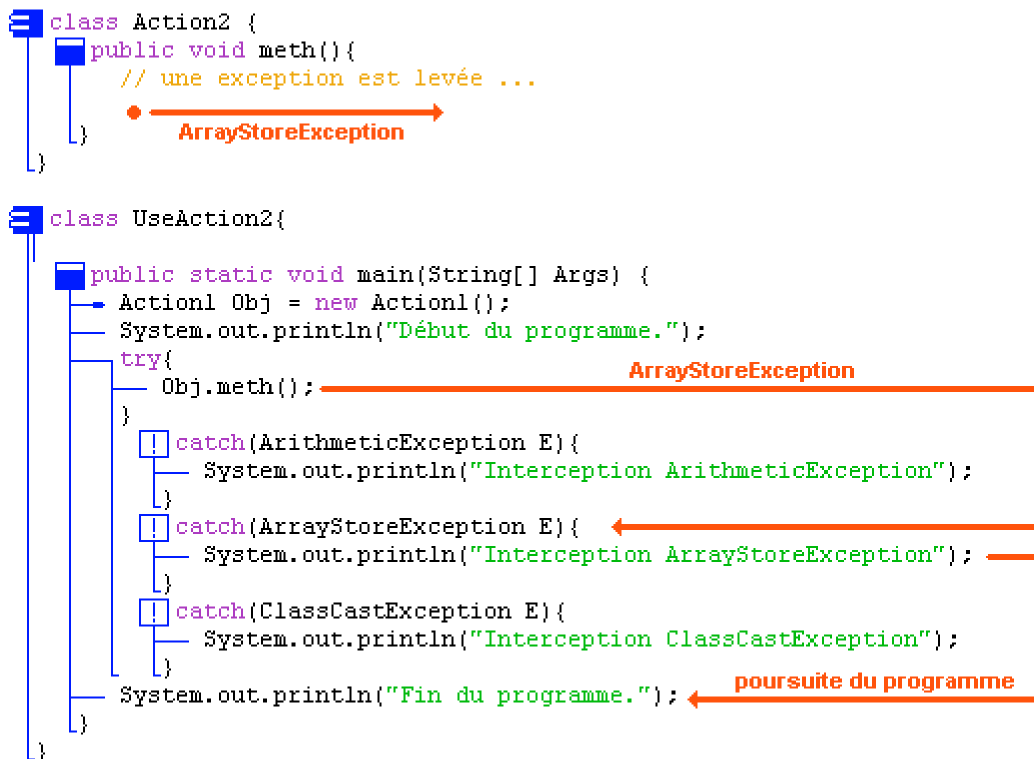
Supposons que la méthode meth() de la classe Action2 puisse lever trois types différents d'exceptions: ArithmeticException, ArrayStoreException, ClassCastException.

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 :

Interception d'une ArrayStoreException :

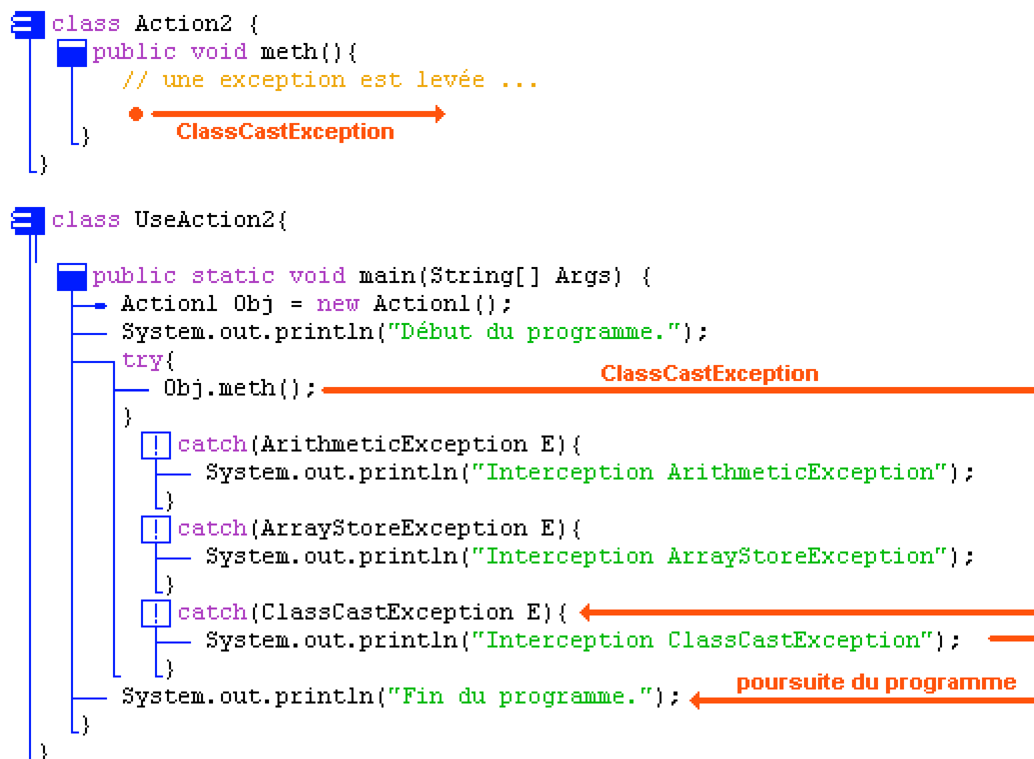
```
class Action2 {
  public void meth(){

  }      ArrayStoreException
}

class UseAction2{
  public static void main(String[] Args) {
      interception de l'ArrayStoreException
  }
}
```

source java :

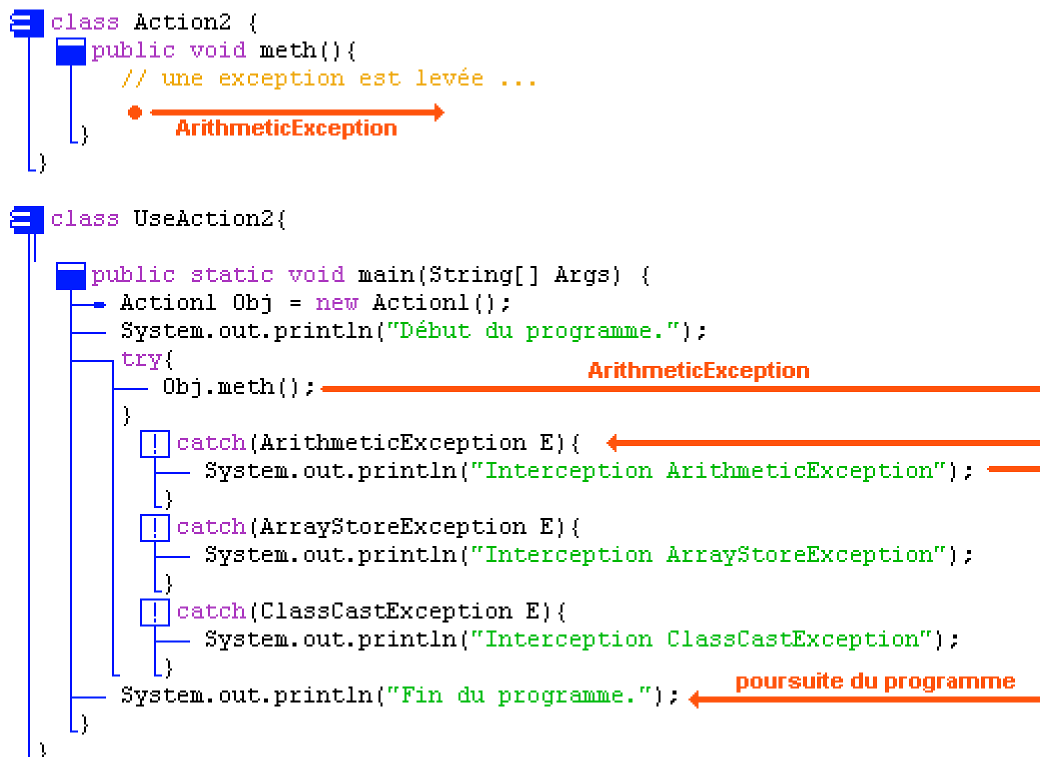


Il en est de même pour les deux autres types d'exception.

Interception d'une ClassCastException :



Interception d'une ArithmeticException :



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.

Soit le pseudo-code java suivant :

try {
< bloc de code à protéger générant un objet exception>
}
catch (TypeException1 E) { <Traitement TypeException1 > }
catch (TypeException2 E) { <Traitement TypeException2 > }
.....
catch (TypeExceptionk E) { <Traitement TypeExceptionk > }

La recherche va s'effectuer comme si le programme contenait des **if...else if...** imbriqués :

if (<Objet exception> **instanceof** TypeException1) { <Traitement TypeException1 > }
else if (<Objet exception> **instanceof** TypeException2) { <Traitement TypeException2 > }
...
else if (<Objet exception> **instanceof** TypeExceptionk) { <Traitement TypeExceptionk > }

Les tests sont effectués sur l'appartenance de l'objet d'exception à une classe à l'aide de l'opérateur **instanceof**.

Signalons que l'opérateur **instanceof** agit sur une classe et ses classes filles (sur une hérarchie de classes), c'est à dire que tout objet de classe TypeExceptionX est aussi considéré comme un objet de classe parent au sens du test d'appartenance en particulier cet objet de classe TypeExceptionX est aussi considéré objet de classe Exception qui est la classe mère de toutes les exceptions Java.

> Le test d'appartenance de classe dans la recherche d'une clause **catch** fonctionne d'une façon identique à l'opérateur **instanceof** dans les if...else

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.

Exemple : Soit une hiérarchie d'exceptions dans java

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

Soit le modèle de gestionnaire d'interception déjà fourni plus haut :

```
try {
  < bloc de code à protéger générant un objet exception>
}
catch ( ArithmeticException E ) { <Traitement ArithmeticException> }
catch ( ArrayStoreException E ) { <Traitement ArrayStoreException> }
catch ( ClassCastException E ) { <Traitement 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 :

```
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.");
  }
}
```

Nous lançons ensuite la compilation de cette classe et nous obtenons un message d'erreur :

```
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.");
  }
}
```

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.

Que s'est-il passé ?

Le fait de placer en premier la clause **catch** (RuntimeException E) chargée d'intercepter les exceptions de classe RuntimeException implique que n'importe quelle exception héritant de RuntimeException comme par exemple ArithmeticException, est considérée comme une RuntimeException. Dans un tel cas cette ArithmeticException est interceptée par la clause **catch** (RuntimeException E) mais elle **n**'est **jamais** interceptée par la clause **catch** (ArithmeticException E).

Le seul endroit où le compilateur Java acceptera l'écriture de la clause **catch** (RuntimeException E) se situe **à la fin de la liste des clauses** catch. Ci-dessous l'écriture d'un programme correct :

```
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

Dans ce cas la recherche séquentielle dans les clauses permettra le filtrage correct des classes filles puis ensuite le filtrage des classes mères.

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.

Redéclenchement d'une exception : throw

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

Déclenchement manuel d'une exception existante

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**. Le programme qui suit lance une ArithmeticException avec le message "Mauvais calcul !" dans la méthode **meth()** et intercepte cette exception dans le bloc englobant **main**. Le traitement de cette exception consiste à afficher le contenu du champ message de l'exception grâce à la méthode **getMessage()** :

```
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.

Déclenchement manuel d'une 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 ArithmeticExceptionPerso héritant de la classe des ArithmeticException puis exécutons ce programme :

```
class ArithmeticExceptionPerso extends ArithmeticException{
  ArithmeticExceptionPerso(String s){
    super(s);
  }
}

class Action3 {
  public void meth(){
    int x=0;
    System.out.println("  ...Avant incident");
    if (x==0)
      throw new ArithmeticExceptionPerso("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(ArithmeticExceptionPerso 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.

L'exécution de ce programme est identique à celle du programme précédent, notre exception fonctionne bien comme celle de Java.

Exception vérifiée ou non

La majorité des exceptions de Java font partie du package java.lang. Certaines exceptions sont tellement courantes qu'elles ont été rangées par Sun (concepteur de Java) dans une catégorie dénommée la catégorie des exceptions **implicites** ou **non vérifiées** (unchecked)**,** les autres sont dénommées exceptions **explicit**es ou **vérifiées** (checked) selon les auteurs.

Une exception **non vérifiée** (implicite) est une classe dérivant de l'une des deux classes **Error** ou **RuntimeException** :

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
```

Toutes les exceptions vérifiées ou non fonctionnent de la même manière, la différence se localise dans la syntaxe à adopter dans une méthode propageant l'un ou l'autre genre d'exception.

Nous pouvons déjà dire que pour les exceptions non vérifiées, il n'y a aucune contrainte syntaxique pour propager une exception d'une méthode vers un futur bloc englobant. La meilleure preuve de notre affirmation est qu'en fait nous n'avons utilisé jusqu'ici que des exceptions non vérifiées, plus précisément des exceptions dérivant de RuntimeException et que nous n'avons utilisé aucune syntaxe spéciale dans la méthode meth() pour indiquer qu'elle était susceptible de lever une exception :

```
+--java.lang.RuntimeException
        |
        +--java.lang.ArithmeticException
        |
        +--java.lang.ArrayStoreException
        |
        +--java.lang.ClassCastException
```

Il n'en est pas de même lorsque l'exception lancée (levée) dans la méthode meth() est une exception vérifiée. Le paragraphe suivant vous explique comment agir dans ce cas.

Méthode propageant une exception vérifiée : throw

Une méthode dans laquelle est levée une ou plusieurs exceptions vérifiées doit obligatoirement signaler au compilateur quelles sont les classes d'exceptions qu'elle laisse se propager sans traitement par un gestionnaire (propagation vers un bloc englobant).

Java dispose d'un spécificateur pour ce signalement : le mot clef **throws** suivi de la liste des noms des classes d'exceptions qui sont propagées.

Signature générale d'une méthode propageant des exceptions vérifiées

<**modificateurs**> <type> < identificateur> (<liste param formels>)
throws < **liste d'exceptions**> {
.....
}

Exemple :

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

Nous allons choisir une classe parmi les très nombreuses d'exceptions vérifiées, notre choix se porte sur une classe d'exception très utilisée, la classe IOException qui traite de toutes les exceptions d'entrée-sortie. Le J2SE 1.4.2 donne la liste des classes dérivant de la classe IOException :

ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, IIOException, InterruptedIOException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedEncodingException, UTFDataFormatException, ZipException.

Reprenons le programme écrit au paragraphe précédent concernant le déclenchement manuel d'une exception existante en l'appliquant à la classe d'exception existante IOException, cette classe étant incluse dans le package **java.io** et non dans le package **java.lang** importé implicitement, on doit ajouter une instruction d'importation du package java.io, puis exécutons le programme tel quel. Voici ce que nous obtenons :

```
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");
  }
}
```

```java
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.");
  }
}
```

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

Que s'est-il passé ?

Le compilateur Java attendait de notre part l'une des deux seules attitudes suivantes :

Soit nous interceptons et traitons l'exception IOException à l'intérieur du corps de la méthode où elle a été lancée avec pour conséquence que le bloc englobant n'aura pas à traiter une telle exception puisque l'objet d'exception est automatiquement détruit dès qu'il a été traité. Le code ci-après implante cette première attitude :

```java
import java.io.*;

class Action4 {
  public void meth(){
    int x=0;
    System.out.println("   ...Avant incident");
    try{
      if (x==0)
        throw new IOException("Problème d'E/S !");}
      catch(IOException E){
        System.out.println("Interception exception : "+E.getMessage());
      }

    System.out.println("   ...Après incident");
  }
}

class UseAction4{
  public static void main(String[] Args) {
    Action4 Obj = new Action4();
    System.out.println("Début du programme.");
    Obj.meth();
    System.out.println("Fin du programme.");
  }
}
```

Interception de l'exception dans la méthode

Appel ordinaire de la méthode dans le bloc englobant

Résultats de l'exécution :
---- java UseAction4
Début du programme.
...Avant incident
Interception exception : Problème d'E/S !
...Après incident
Fin du programme.

Soit nous interceptons et traitons l'exception IOException à l'intérieur du bloc englobant la méthode meth() qui a lancé l'exception, auquel cas il est obligatoire de signaler au compilateur que cette méthode meth() lance et propage une IOException non traitée. Le code ci-après implante la seconde attitude possible :

```
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.

Redéfinition d'une méthode propageant des exceptions vérifiées

Principe de base : la partie **throws** < **liste d'exceptions**> de la signature de la méthode qui redéfinit une méthode de la super-classe peut comporter moins de types d'exception. Elle ne peut pas propager plus de types ou des types différents de ceux de la méthode de la super-classe.

Ci-après un programme avec une super-classe **Action4** et une méthode **meth()**, qui est redéfinie

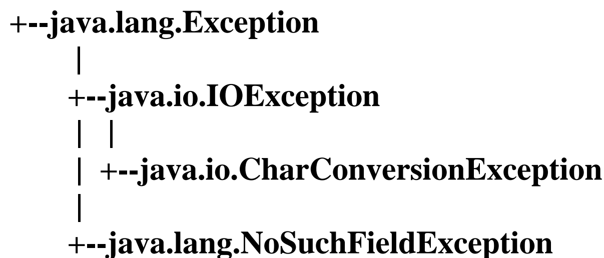
dans une classe fille nommée **Action5** :

Propagations initiales

Redéfinition de la méthode meth(int x)

```
import java.io.*;

class Action4 {
  protected void meth(int x)throws IOException,NoSuchFieldException {
    System.out.println("  ...Avant incident-mère");
    if (x==0)
      throw new CharConversionException("Problème conversion !-mère");
    System.out.println("  ...Après incident-mère");
  }
}

class Action5 extends Action4 {
  public void meth(int x)throws CharConversionException,
    IOException, NoSuchFieldException {
    System.out.println("Appel meth("+x+")");
    System.out.println("  ...Avant incident-fille");
    if (x==0)
      try{
        super.meth(x);}
        catch(CharConversionException e){
          throw e;
        }
    if (x==1)
      throw new IOException("Problème d'E/S !-fille");
    if (x==2)
      throw new NoSuchFieldException("Problème de champ !-fille");
    System.out.println("  ...Après incident-fille");
  }
}
```

Voici la hiérarchie des classes utilisées :



```
+--java.lang.Exception
      |
      +--java.io.IOException
      |   |
      |   +--java.io.CharConversionException
      |
      +--java.lang.NoSuchFieldException
```

Notez que la méthode meth de la super-classe propage IOException et NoSuchFieldException bien qu'elle ne lance qu'une exception de type IOException; ceci permettra la redéfinition dans la classe fille.

Voici pour terminer, un code source de classe utilisant la classe Action5 précédemment définie et engendrant aléatoirement l'un des 3 types d"exception :

```
class UseAction5{
  public static void main(String[] Args) {
    Action5 Obj = new Action5();
    System.out.println("Début du programme.");
    try{
      Obj.meth((int)(Math.random()*10)%3);
    }
    catch(CharConversionException E){
      System.out.println("Interception exception_0 : "+E.getMessage());
    }
    catch(IOException E){
      System.out.println("Interception exception_1 : "+E.getMessage());
    }
    catch(NoSuchFieldException E){
      System.out.println("Interception exception_2 : "+E.getMessage());
    }
    System.out.println("Fin du programme.");
  }
}
```

Analysez et comprenez bien le fonctionnement de ce petit programme.

Listing des 3 exécutions dans chacun des cas d'appel de la méthode meth :

Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(0)
...Avant incident-fille
...Avant incident-mère
Interception exception_0 : Problème conversion de caractère !-mère
Fin du programme.

Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(2)
...Avant incident-fille
Interception exception_2 : Problème de champ !-fille
Fin du programme.

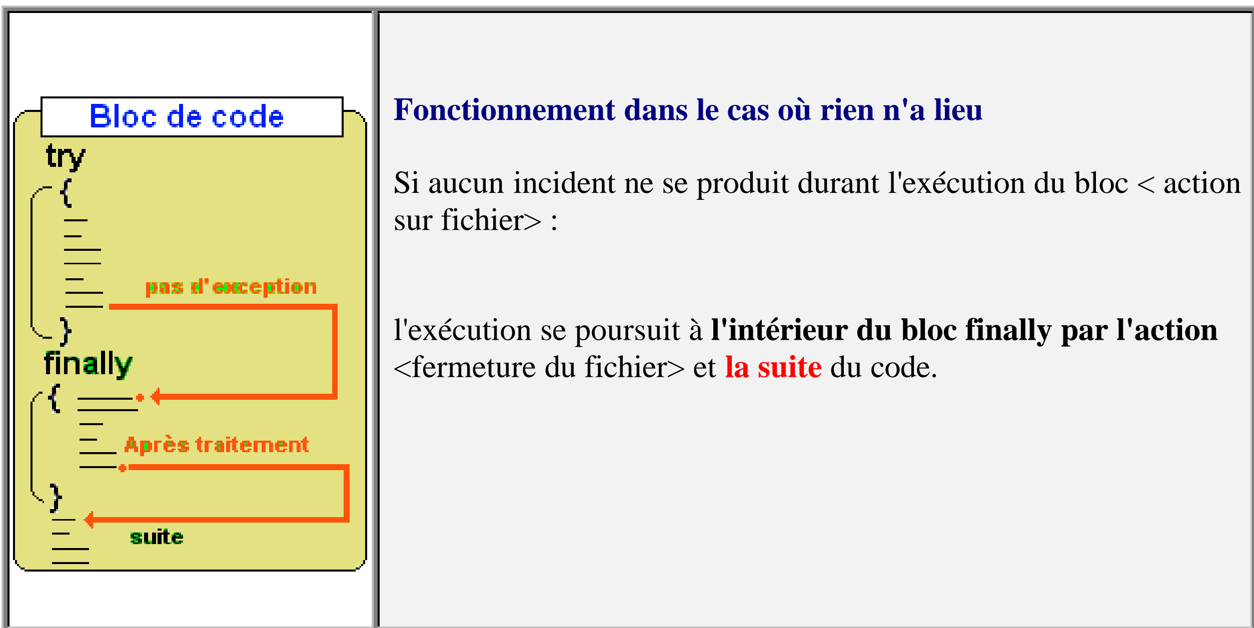
Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(1)
...Avant incident-fille
Interception exception_1 : Problème d'E/S !-fille
Fin du programme.

Clause finally

Supposons que nous soyons en présence d'un code contenant une éventuelle levée d'exception, mais supposons que quoiqu'il se passe nous désirions qu'un certain type d'action ait toujours lieu

(comme par exemple fermer un fichier qui a été ouvert auparavant). Il existe en Java une **clause spécifique optionnelle** dans la syntaxe des gestionnaires d'exception permettant ce type de réaction du programme, c'est la clause **finally.** Voici en pseudo Java une syntaxe de cette clause :

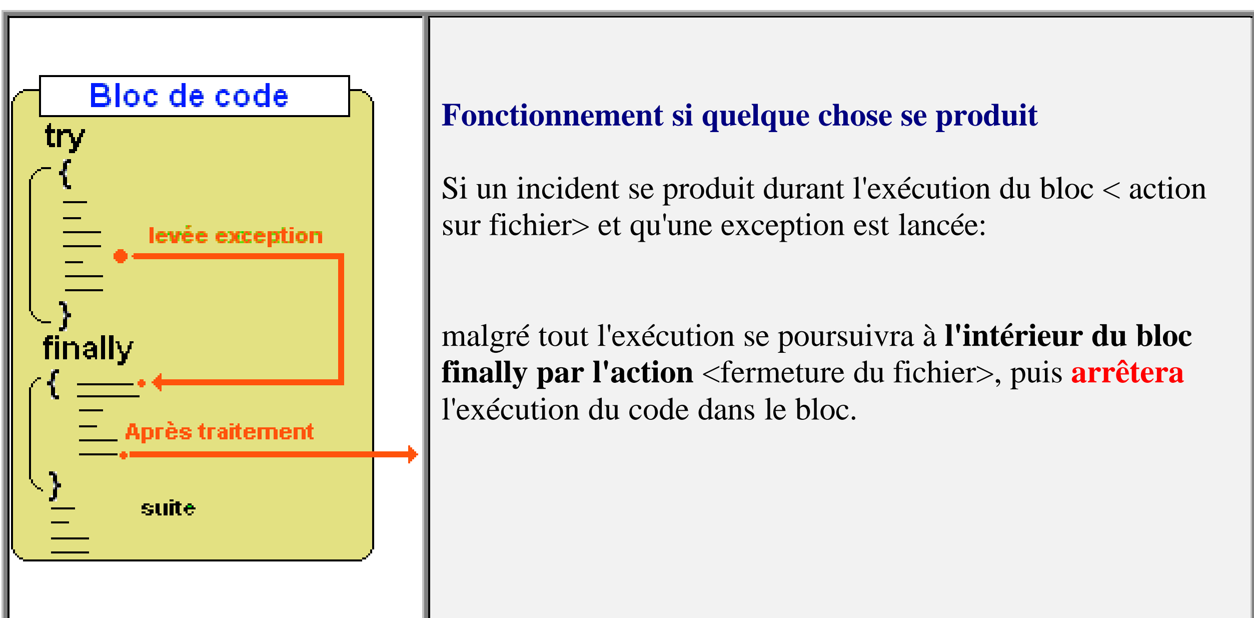
```
<Ouverture du fichier>
try {
< action sur fichier>
}
finally {
 <fermeture du fichier>
}
.... suite
```



Fonctionnement dans le cas où rien n'a lieu

Si aucun incident ne se produit durant l'exécution du bloc < action sur fichier> :

l'exécution se poursuit à **l'intérieur du bloc finally par l'action** <fermeture du fichier> et **la suite** du code.



Fonctionnement si quelque chose se produit

Si un incident se produit durant l'exécution du bloc < action sur fichier> et qu'une exception est lancée:

malgré tout l'exécution se poursuivra à **l'intérieur du bloc finally par l'action** <fermeture du fichier>, puis **arrêtera** l'exécution du code dans le bloc.

La syntaxe Java autorise l'écriture d'une clause **finally** associée à plusieurs clauses **catch** :

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

Remarque :

Si le code du bloc à protéger dans try...finally contient une instruction de rupture de séquence comme break, return ou continue, le code de la clause finally{...} est malgré tout exécuté avant la rupture de séquence.

Nous avons vu lors des définitions des itérations while, for et de l'instruction continue, que l'équivalence suivante entre un **for** et un **while** valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue** (instruction forçant l'arrêt d'un tours de boucle et relançant l'itération suivante) :

*Equivalence incorrecte si Instr contient un **continue** :*

for (Expr1 ; Expr2 ; Expr3) Instr	Expr1 ;
	while (Expr2)
	{ Instr ;
	  Expr3
	}

E*quivalence correcte même si Instr contient un **continue** :*

for (Expr1 ; Expr2 ; Expr3) Instr	Expr1 ;
	while (Expr2)
	{
	 try {
	  Instr ;
	 } finally { Expr3
	  }
	}

Le multi-threading

Java2

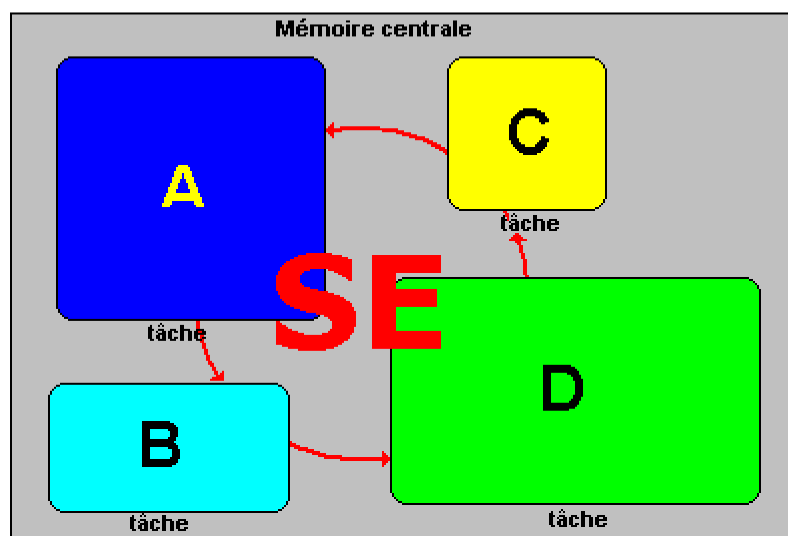
Le Multithreading

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multi-programmation ou multi-tâches, permettant l'exécution "simultanée" de plusieurs tâches.

Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'**un seul processeur**, en apparence **en même temps** ou encore en **parallèle,** en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

Multitâche et thread

Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un **processus** :

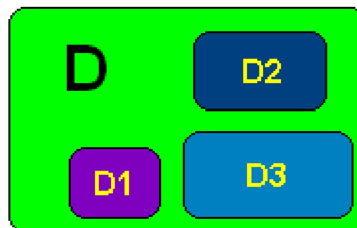


Rappelons la définition des **processus** donnée par A.Tanenbaum: un programme qui s'exécute et qui possède **son propre espace mémoire** : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multi-programmation par la commutation entre processus effectuée par le processeur unique).

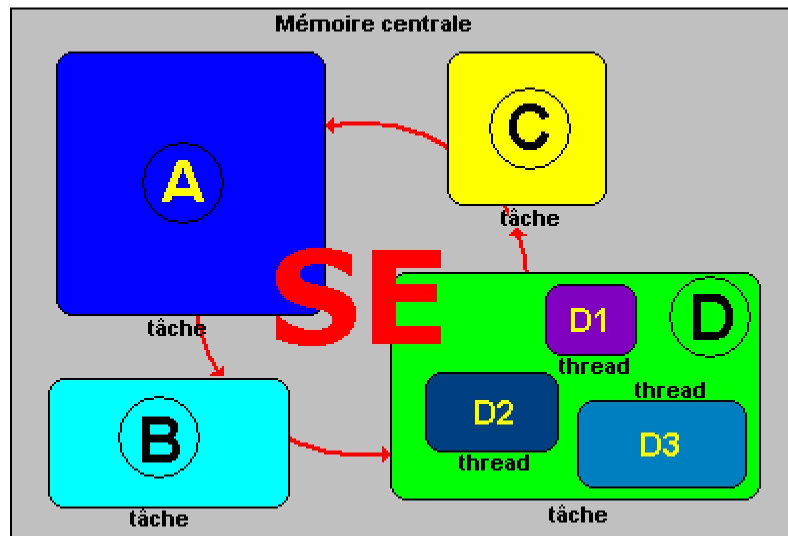
Thread

En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multi-programmation. Ces sous-tâches sont nommées "flux d'exécution" ou **Threads.**

Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :

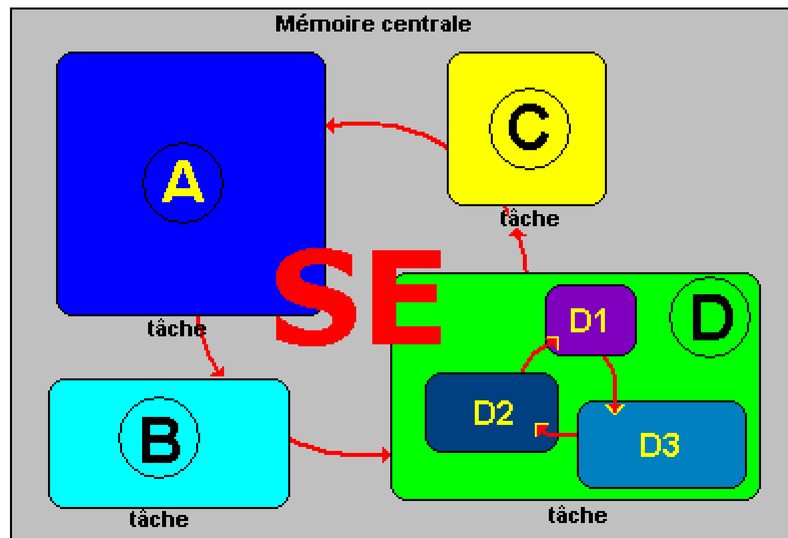


Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads) :



La commutation entre les threads d'un processus fonctionne de la même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps.

Le partage et la répartition du temps sont effectués **uniquement** par le système d'exploitation :



Multithreading et processus

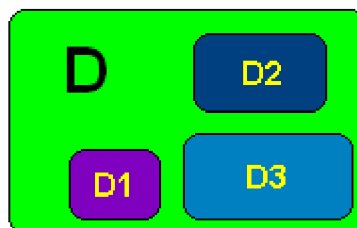
Définition :

La majorité des systèmes d'exploitation (Windows, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de **Multithreading.**

Différences entre threads et processus :

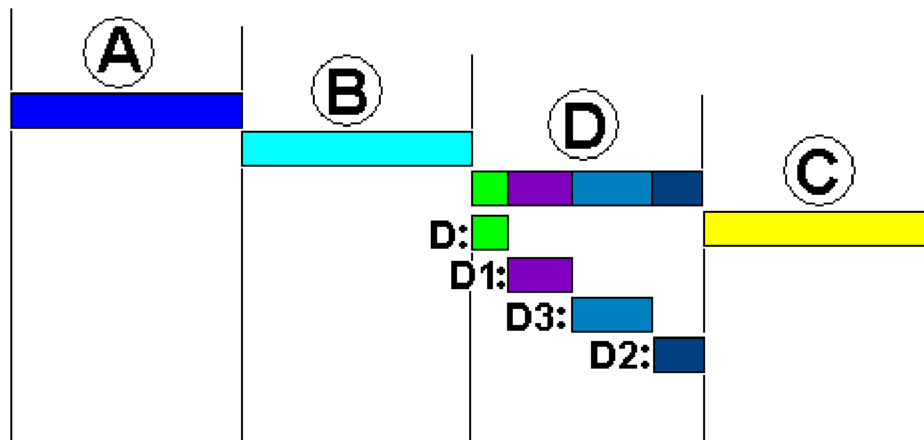
- Communication entre threads **plus rapide** que la communication entre processus,
- Les threads partagent un **même espace de mémoire** (de travail) entre eux,
- Les processus ont chacun un **espace mémoire personnel**.

Dans l'exemple précédent, figurons les processus A, B, C et le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.



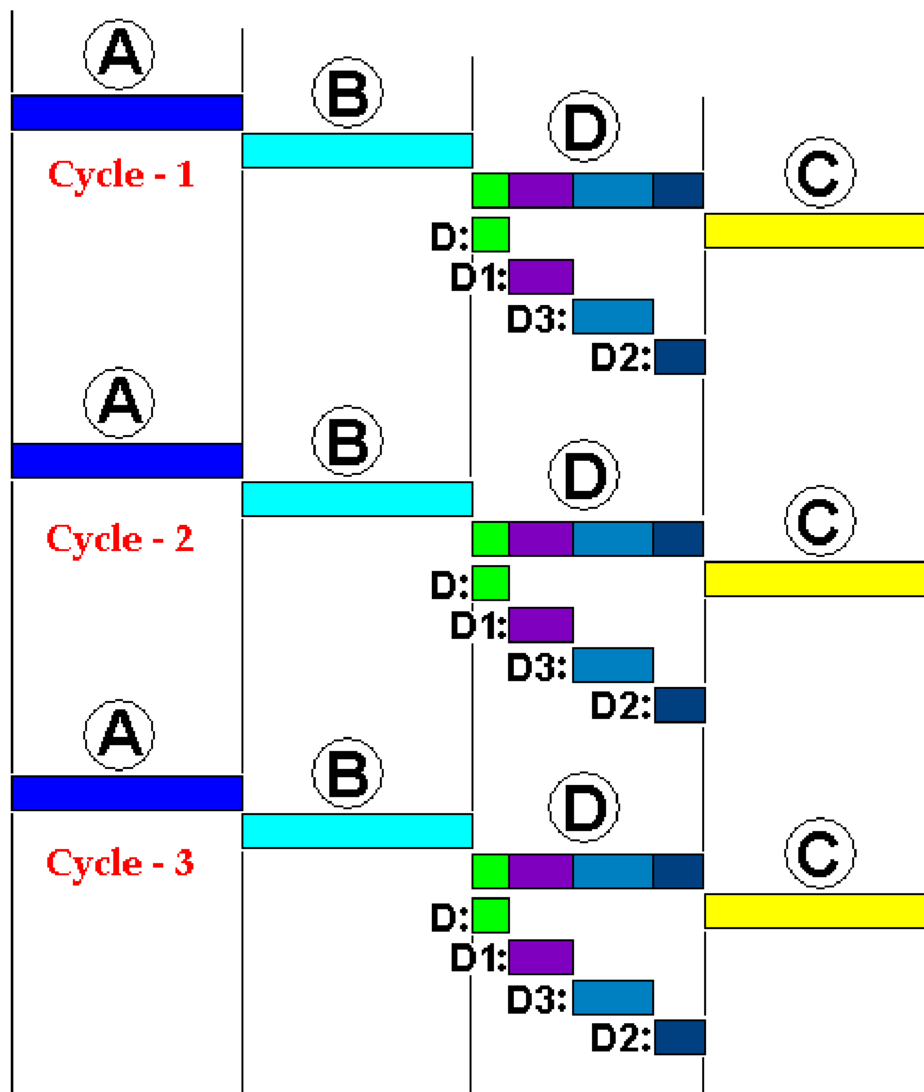
Le système ayant alloué le même temps d''exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une

petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :



Tranches de temps allouées pendant l'exécution

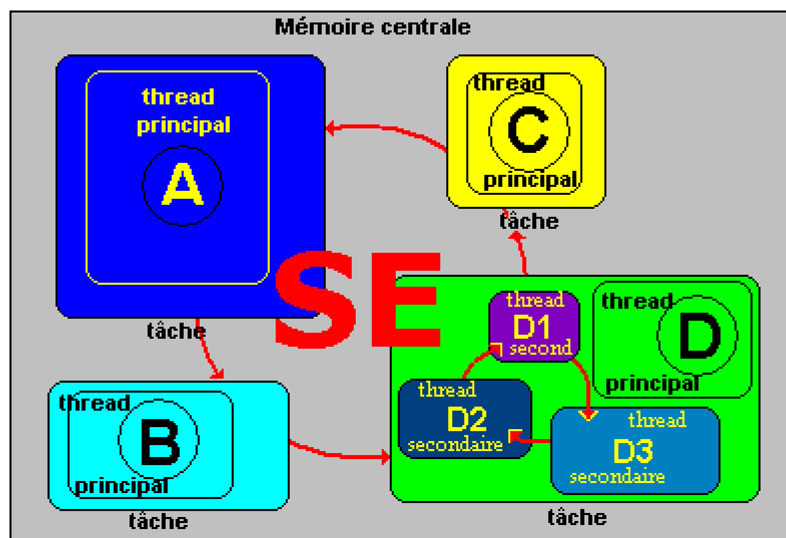
Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



Les langages comme Delphi, Java et C# disposent chacun de classes permettant d'écrire et d'utiliser des threads dans vos applications.

Java autorise l'utilisation des threads

Lorsqu'un programme Java s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé **thread principal**. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et D soient toutes des applications Java, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



En Java c'est l'interface **Runnable** qui permet l'utilisation des threads dans la Java machine. Voici une déclaration de cette interface :

```
public interface Runnable {
  public void run( ) ;
}
```

Cette interface Runnable doit obligatoirement être implémentée par toute classe dont les instances seront exécutées par un thread. Une telle classe implémentant l'interface Runnable doit alors implémenter la méthode abstraite sans paramètre **public void** run( ) . Cette interface est conçue pour mettre en place un protocole commun à tous les objets qui souhaitent exécuter du code pendant que eux-mêmes sont actifs.

L'interface Runnable est essentiellement implantée par la classe **Thread** du package java.lang :

```
java.lang.Object
  |
  +--java.lang.Thread
```

Nous voyons maintenant comment concevoir une classe personnalisée dénommée **MaClasse** qui permettre l'exécution de ses instances dans des threads machines. Java vous offre deux façons différentes de décrire la classe **MaClasse** utilisant des threads :

- Soit par implémentation de l'interface **Runnable**.
- Soit par héritage de la classe **java.lang.Thread**

Interface Runnable
On effectue ce choix habituellement lorsque l'on veut utiliser une classe **ClassB** héritant d'une classe **ClassA** et que cette **ClassB** fonctionne comme un thread : il faudrait donc pouvoir hériter à la fois de la classe **ClassA** et de la classe **Thread,** ce qui est impossible, java ne connaît pas l'héritage multiple. En implémentant l'interface **Runnable** dans la classe **ClassB**, on rajoute à notre classe **ClassB** une nouvelle méthode **run ( )** (qui est en fait la seule méthode de l'interface Runnable).

Classe Thread
Dans le second cas l'héritage à partir de la classe Thread qui implémente l'interface **Runnable** et qui permet l'utilisation de méthodes de manipulation d'un thread (mise en attente, reprise,...).

Dans les deux cas, il vous faudra instancier un objet de classe Thread et vous devrez mettre ou invoquer le code à "exécuter en parallèle" dans le corps de la méthode run ( ) de votre classe .

L'interface Runnable

Première version Maclasse implémente l'interface Runnable :
vous héritez d'une seule méthode run ( ) que vous devez implémenter

Une classe dénommée MaClasse :

```
public interface Runnable {
  public void run( ) ;
}

class Thread implements Runnable{
  public void run( ){
  } ;
  //....
}

class MaClasse implements Runnable{
  public void run( ){
  } ;
  //....
}
```

Exemple de modèle :
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres de 1 à 100

```
public class MaClasse implements Runnable {
  Thread  UnThread ;
```

```
    MaClasse ( ) {
    //....initialisations éventuelles du constructeur de MaClasse
    UnThread = new Thread ( this , "thread secondaire" );
    UnThread.start( ) ;
    }

    public void run ( ) {
    //....actions du thread secondaire ici
    for ( int i1=1; i1<100; i1++)
            System.out.println(">>> i1 = "+i1);
    }
  }
```

Pour utiliser cette classe contenant un thread secondaire, il est necessaire d'instancier un objet de cette classe :

```
class UtiliseMaclasse {
  public static void main(String[] x) {
    MaClasse tache1 = new MaClasse (  );
    /*.......le thread secondaire a été créé et activé
    le reste du code concerne le thread principal */
    for ( int i2 =1; i2<100;i2++)
           System.out.println(" i2 = "+i2);
  }
}
```

La classe java.lang.Thread

Deuxième version Maclasse dérive de la classe Thread :

vous héritez de toutes les méthodes de la classe Thread dont la méthode run( ) que vous devez redéfinir

```
public interface Runnable {
  public void run( ) ;
}

class Thread implements Runnable{
  public void run( ){
  } ;
  //....
}

class MaClasse extends Thread{
  public void run( ){
  } ;
  //....
}
```

Exemple de modèle :
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres

de 1 à 100

```
public class MaClasse extends Thread {

  MaClasse ( ) {
    //....initialisations éventuelles du constructeur de MaClasse
    this .start( ) ;
  }

  public void run ( ) {
  //....actions du thread secondaire ici
  for ( int i1=1; i1<100; i1++)
        System.out.println(">>> i1 = "+i1);
  }
}
```

Lorsque vous voulez utiliser cette classe contenant un thread secondaire, il vous faudra instancier un objet de cette classe de la même manière que pour la première version :

```
class UtiliseMaclasse {
  public static void main(String[] x) {
    MaClasse tache1 = new MaClasse ( );
    /*.......le thread secondaire a été créé et activé
    le reste du code concerne le thread principal */
    for ( int i2 =1; i2<100;i2++)
        System.out.println(" i2 = "+i2);
  }
}
```

Les deux versions par interface **Runnable** ou classe **Thread**, produisent le même résultat d'exécution. Chaque thread affiche séquentiellement sur la console ses données lorsque le système lui donne la main, ce qui donne un "mélange des deux affichages" :

Résultats d'exécution :

i2 = 1	i2 = 50	i2 = 95	>>> i1 = 50
i2 = 2	i2 = 51	i2 = 96	>>> i1 = 51
i2 = 3	i2 = 52	i2 = 97	>>> i1 = 52
i2 = 4	i2 = 53	i2 = 98	>>> i1 = 53
i2 = 5	i2 = 54	i2 = 99	>>> i1 = 54
i2 = 6	i2 = 55	>>> i1 = 5	>>> i1 = 55
i2 = 7	i2 = 56	>>> i1 = 6	>>> i1 = 56
i2 = 8	i2 = 57	>>> i1 = 7	>>> i1 = 57
i2 = 9	i2 = 58	>>> i1 = 8	>>> i1 = 58
i2 = 10	>>> i1 = 1	>>> i1 = 9	>>> i1 = 59
i2 = 11	i2 = 59	>>> i1 = 10	>>> i1 = 60
i2 = 12	>>> i1 = 2	>>> i1 = 11	>>> i1 = 61
i2 = 13	i2 = 60	>>> i1 = 12	>>> i1 = 62
i2 = 14	>>> i1 = 3	>>> i1 = 13	>>> i1 = 63
i2 = 15	i2 = 61	>>> i1 = 14	>>> i1 = 64
i2 = 16	>>> i1 = 4	>>> i1 = 15	>>> i1 = 65
i2 = 17	i2 = 62	>>> i1 = 16	>>> i1 = 66
i2 = 18	i2 = 63	>>> i1 = 17	>>> i1 = 67
i2 = 19	i2 = 64	>>> i1 = 18	>>> i1 = 68
i2 = 20	i2 = 65	>>> i1 = 19	>>> i1 = 69
i2 = 21	i2 = 66	>>> i1 = 20	>>> i1 = 70
i2 = 22	i2 = 67	>>> i1 = 21	>>> i1 = 71
i2 = 23	i2 = 68	>>> i1 = 22	>>> i1 = 72
i2 = 24	i2 = 69	>>> i1 = 23	>>> i1 = 73
i2 = 25	i2 = 70	>>> i1 = 24	>>> i1 = 74

i2 = 26
i2 = 27
i2 = 28
i2 = 29
i2 = 30
i2 = 31
i2 = 32
i2 = 33
i2 = 34
i2 = 35
i2 = 36
i2 = 37
i2 = 38
i2 = 39
i2 = 40
i2 = 41
i2 = 42
i2 = 43
i2 = 44
i2 = 45
i2 = 46
i2 = 47
i2 = 48
i2 = 49
	i2 = 71
	i2 = 72
	i2 = 73
	i2 = 74
	i2 = 75
	i2 = 76
	i2 = 77
	i2 = 78
	i2 = 79
	i2 = 80
	i2 = 81
	i2 = 82
	i2 = 83
	i2 = 84
	i2 = 85
	i2 = 86
	i2 = 87
	i2 = 88
	i2 = 89
	i2 = 90
	i2 = 91
	i2 = 92
	i2 = 93
	i2 = 94
		>>> i1 = 25
		>>> i1 = 26
		>>> i1 = 27
		>>> i1 = 28
		>>> i1 = 29
		>>> i1 = 30
		>>> i1 = 31
		>>> i1 = 32
		>>> i1 = 33
		>>> i1 = 34
		>>> i1 = 35
		>>> i1 = 36
		>>> i1 = 37
		>>> i1 = 38
		>>> i1 = 39
		>>> i1 = 40
		>>> i1 = 41
		>>> i1 = 42
		>>> i1 = 43
		>>> i1 = 44
		>>> i1 = 45
		>>> i1 = 46
		>>> i1 = 47
		>>> i1 = 48
		>>> i1 = 49
			>>> i1 = 75
			>>> i1 = 76
			>>> i1 = 77
			>>> i1 = 78
			>>> i1 = 79
			>>> i1 = 80
			>>> i1 = 81
			>>> i1 = 82
			>>> i1 = 83
			>>> i1 = 84
			>>> i1 = 85
			>>> i1 = 86
			>>> i1 = 87
			>>> i1 = 88
			>>> i1 = 89
			>>> i1 = 90
			>>> i1 = 91
			>>> i1 = 92
			>>> i1 = 93
			>>> i1 = 94
			>>> i1 = 95
			>>> i1 = 96
			>>> i1 = 97
			>>> i1 = 98
			>>> i1 = 99

---- operation complete.

Méthodes de base sur les objets de la classe Thread

Les méthodes static de la classe Thread travaillent sur le thread courant ***(en l'occurrence le thread qui invoque la méthode static)*** **:**

static int activeCount( )	renvoie le nombre total de threads actifs actullement.
static Thread currentThread( )	renvoie la référence de l'objet thread actuellement exécuté (thread courant)
static void dumpStack( )	Pour le debugging : une trace de la pile du thread courant
static int enumerate(Thread [] tarray)	Renvoie dans le tableau tarray[] les reférences de tous les threads actifs actullement.
static boolean interrupted( )	Teste l'indicateur d'interruption du thread courant, mais change cet interrupteur (utiliser plutôt la méthode d'instance **boolean** isInterrupted( )).
static void sleep(**long** millis, **int** nanos)	Provoque l'arrêt de l'exécution du thread courant pendant milis milisecondes et nanos nanosecondes.

static void sleep(**long** millis)	Provoque l'arrêt de l'exécution du thread courant pendant milis milisecondes.

Reprenons l'exemple ci-haut de l'affichage entre-mêlé de deux listes d'entiers de 1 à 100 :

```
class MaClasse2 extends Thread {

   MaClasse2 ( ) {
     this .start( ) ;
   }

   public void run ( ) {
   for ( int i1=1; i1<100; i1++)
           System.out.println(">>> i1 = "+i1);
   }
}

class UtiliseMaclasse2 {
  public static void main(String[] x) {
    // corps du thread principal
    MaClasse2 tache1 = new MaClasse (  ); // le thread secondaire
    for ( int i2 =1; i2<100;i2++)
           System.out.println(" i2 = "+i2);
  }
}
```

Arrêtons pendant une milliseconde l'exécution du thread principal : **Thread.sleep(1);** comme la méthode de classe sleep( ) propage une InterruptedException nous devons l'intercepter :

```
class UtiliseMaclasse2 {
  public static void main(String[] x) {
    MaClasse2 tache1 = new MaClasse2 (  );
  //.......le thread secondaire a été créé et activé le reste du
    try {
      Thread.sleep(1);
    }
    catch(InterruptedException e){
      System.out.println(e);
    }
    for(int i2=1;i2<100;i2++)
      System.out.println("i2 = "+i2);
  }
}
```

Ralenti de 1ms le thread principal

En **arrêtant pendant 1ms** le thread principal, nous laissons donc plus de temps au thread secondaire pour s'exécuter, les affichages ci-dessous le montrent :

Résultats d'exécution :

>>> i1 = 1
>>> i1 = 2
>>> i1 = 3
	>>> i1 = 50
	>>> i1 = 51
	>>> i1 = 52
		i2 = 13
		i2 = 14
		i2 = 15
			>>> i1 = 95
			>>> i1 = 96
			>>> i1 = 97

>>> i1 = 4	>>> i1 = 53	i2 = 16	>>> i1 = 98
>>> i1 = 5	>>> i1 = 54	i2 = 17	>>> i1 = 99
>>> i1 = 6	>>> i1 = 55	i2 = 18	i2 = 55
>>> i1 = 7	>>> i1 = 56	i2 = 19	i2 = 56
>>> i1 = 8	>>> i1 = 57	i2 = 20	i2 = 57
>>> i1 = 9	>>> i1 = 58	i2 = 21	i2 = 58
>>> i1 = 10	>>> i1 = 59	i2 = 22	i2 = 59
>>> i1 = 11	>>> i1 = 60	i2 = 23	i2 = 60
>>> i1 = 12	>>> i1 = 61	i2 = 24	i2 = 61
>>> i1 = 13	>>> i1 = 62	i2 = 25	i2 = 62
>>> i1 = 14	>>> i1 = 63	i2 = 26	i2 = 63
>>> i1 = 15	>>> i1 = 64	i2 = 27	i2 = 64
>>> i1 = 16	>>> i1 = 65	i2 = 28	i2 = 65
>>> i1 = 17	>>> i1 = 66	i2 = 29	i2 = 66
>>> i1 = 18	>>> i1 = 67	i2 = 30	i2 = 67
>>> i1 = 19	>>> i1 = 68	i2 = 31	i2 = 68
>>> i1 = 20	>>> i1 = 69	i2 = 32	i2 = 69
>>> i1 = 21	>>> i1 = 70	i2 = 33	i2 = 70
>>> i1 = 22	>>> i1 = 71	i2 = 34	i2 = 71
>>> i1 = 23	>>> i1 = 72	i2 = 35	i2 = 72
>>> i1 = 24	>>> i1 = 73	i2 = 36	i2 = 73
>>> i1 = 25	>>> i1 = 74	i2 = 37	i2 = 74
>>> i1 = 26	>>> i1 = 75	i2 = 38	i2 = 75
>>> i1 = 27	>>> i1 = 76	i2 = 39	i2 = 76
>>> i1 = 28	>>> i1 = 77	i2 = 40	i2 = 77
>>> i1 = 29	>>> i1 = 78	i2 = 41	i2 = 78
>>> i1 = 30	>>> i1 = 79	i2 = 42	i2 = 79
>>> i1 = 31	>>> i1 = 80	i2 = 43	i2 = 80
>>> i1 = 32	>>> i1 = 81	i2 = 44	i2 = 81
>>> i1 = 33	i2 = 1	i2 = 45	i2 = 82
>>> i1 = 34	>>> i1 = 82	i2 = 46	i2 = 83
>>> i1 = 35	i2 = 2	i2 = 47	i2 = 84
>>> i1 = 36	>>> i1 = 83	i2 = 48	i2 = 85
>>> i1 = 37	i2 = 3	i2 = 49	i2 = 86
>>> i1 = 38	>>> i1 = 84	i2 = 50	i2 = 87
>>> i1 = 39	i2 = 4	i2 = 51	i2 = 88
>>> i1 = 40	>>> i1 = 85	>>> i1 = 87	i2 = 89
>>> i1 = 41	i2 = 5	i2 = 52	i2 = 90
>>> i1 = 42	>>> i1 = 86	>>> i1 = 88	i2 = 91
>>> i1 = 43	i2 = 6	i2 = 53	i2 = 92
>>> i1 = 44	i2 = 7	>>> i1 = 89	i2 = 93
>>> i1 = 45	i2 = 8	i2 = 54	i2 = 94
>>> i1 = 46	i2 = 9	>>> i1 = 90	i2 = 95
>>> i1 = 47	i2 = 10	>>> i1 = 91	i2 = 96
>>> i1 = 48	i2 = 11	>>> i1 = 92	i2 = 97
>>> i1 = 49	i2 = 12	>>> i1 = 93	i2 = 98
		>>> i1 = 94	i2 = 99

---- operation complete.

Les méthodes d'instances de la classe Thread

Java permet d'arrêter (**stop**), de faire attendre (**sleep, yield**), d'interrompre (**interrupt**), de changer la priorité(**setPriority**), de détruire(**destroy**) des threads à travers les méthodes de la classe Thread.

Java permet à des threads de "communiquer" entre eux sur leur état (utiliser alors les méthodes : **join, notify, wait**).