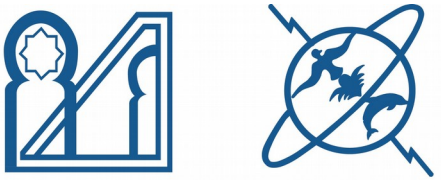


Chapitre I : Introduction à la programmation en Java

Pr. Younès EL AMRANI
Faculté des Sciences de Rabat
Université Mohamed V-Agdal
2015-2019

Introduction du chapitre 1

dans ce chapitre nous effectuerons un tour d'horizon de JAVA en prélude d'un exposé détaillé qui suivra.

Premier programme en java

```
Class HelloWorld {
 public static void main(String[] args) {
    System.out.println ( ” Hello, World ” ) ; }
}
```

Les types primitifs de java

boolean	true ou false : booléen
char	Unicode de 16-bit
byte	entier de 8-bit signé
short	entier 16-bit signé
int	entier 32-bit signé
long	entier 64-bit signé
float	décimal 32-bit (IEEE 754-1985)
double	décimal 64-bit (IEEE 754-1985)

Les commentaires en java

- Les lignes qui commencent par // sont considérées comme des commentaires
- Tout texte compris entre /* et */ est considéré comme un commentaire

Les constantes

- Les constantes sont les valeurs telles que:

  12,
  17.9
  ainsi ***"que les chaînes de caractères comme celle-ci"***

-

```
class Cercle {
   static final double pi = 3.1416 ;
}
```

Les constantes en java

- Pour les variables statiques le nom de la classe suffit pour accéder à la constante.

  Cercle.pi

La gestion des constantes

```
class LengthMonth {
   static final int january 31 ;
   static final int february 28 ;
   ...
   static final int december 31 ;
}
```

Les classes et les objets

- Les objets en java ont un type. Ce type est la classe de l'objet.

L'instanciation des objets d'une classe

```
class Point {   public double x, y ; }
  /* La classe Point possède deux attributs et aucune méthode. */
```

-
```
Point lowerLeft = new Point() ;
lowerLeft.x = 0.0 ;
lowerLeft.y = 0.0 ;
```

Les champs de classe versus les champs statiques.

- Les champs qui ne sont pas statiques sont des champs par-objet

- Les champs statiques sont partagés

Exemple de variable statique

- La classe Point ci-dessous possède trois attributs et aucune méthode.

```
class Point {
   public static Point origin = new Point () ;
   public double x, y ;
}
```

Initialisations par défaut

- Java initialise par défaut les champs d ’une classe, en voici les valeurs:
  - boolean = false
  - floating point = `\u0000`
  - référence a un objet = null
  - integer (byte, short, int, long) 0
- Java n’initialise pas par défaut les:
  - Les variable locales d’une méthode
  - Les variable locales d’un constructeur
  - Les variable locales d’un bloc d ’initialisation statique.

La création d'objets

- Les objets sont créés dans le Heap

La création et destruction d'objets en mémoire

- La création d'objets : la primitive *new ( )*

```
Point lowerLeft = new Point ( ) ;
lowerLeft.x = 0.0 ;
lowerLeft.y = 0.0 ;
```

- La destruction des objets en mémoire : le *Garbage Collector*.

Méthodes et paramètres

- Les opérations en java sont définies dans les méthodes.

La signature d'une méthode

- La signature d'une méthode est son nom ainsi que le type et le nombre de ses paramètres.

L’auto-référence *this*

- L ’auto-référence *this* est disponible dans les méthodes et sert d ’auto-référence pour l ’objet en cours de définition.

Les méthodes statiques

- Les méthodes statiques sont appelées les méthodes de classe, par exemple toutes les méthodes de la classe java.lang.Math sont statiques

Les tableaux

- Déclaration d ’un tableau de Point:

```
class tabDePoint {
   final int N = 100 ;
   Point[ ]    centPoint = new Point[100] ;
   public void print ( ) {
        for ( i := 0 ; i < centPoint.length; i++)
                System.out.println ( "("+ x + "," + y + ")" );
        }
  }
```

Perspectives du chapitre 1

- En perspective de ce tour d'horizon, il convient de rappeler que JAVA possède un double héritage : celui de la syntaxe du langage C, dont il est naît comme un programme en 1995 (à l'origine JAVA était écrit entièrement en JAVA) d'ailleurs la machine virtuelle de JAVA, la JVM , est entièrement écrite en C.

- Le second héritage de JAVA provient de OAK Lisp : un langage fonctionnel qui s'est en partie ré-incarné en JAVA sous la forme des lambda-expressions, présentées dans la suite de ce cours.

Conclusion du chapitre 1

- Nous avons fais un tour d'horizon du langage JAVA, avec en particulier un parcours rapide des éléments de l'héritage ainsi que les constructions essentielle du langage.
- Le langage JAVA comporte de nombreuses autres constructions syntaxiques ainsi que d'autres mécanismes liés ou non à l'héritage que nous allons présenter dans la suite.

Chapitre II : PRINCIPES DE LA PROGRAMMATION ORIENTE OBJET EN JAVA

Pr. Younès EL AMRANI.
youneselamrani@gmail.com

Introduction à la programmation orientée Objet

- Depuis le milieu des années 80, la programmation OO elle est utilisée dans le but de la ré-utilisation de programmes.

Définition de la programmation orientée Objet

- La programmation orientée objet consiste à associer des données (attributs) a des fonctions (méthodes) qui agissent sur ces attributs au sein d'une même entité appelée la classe.

Définition de la programmation orientée Objet

- Une classe est une généralisation de la notion de type

Définition de la référence this

- Toutes les méthodes d'une classe utilisent un paramètre supplémentaire, qui n'apparaît pas dans la liste des paramètre de la méthode et qui est noté this**.**

L'auto-référence this

- This est une auto-référence de la classe à elle-même:

```
class A {
Integer a, b;
Void set( int a, int b)
      {
            This.a = a ;
            This.b = b ;
      }
}
```

L’auto-référence this et les variables locales de même nom

- This permet de lever l’ambiguité lorsque des variables locales ont le même nom que des attributs de classe:

```
class A {
Integer a = 1, b = 2, c = 3, d =4;
Integer F1( Integer a) {
      return a*(this.a + b + c + d) ;
}
}
```

L’impression en JAVA: les méthodes System.out.print et System.out.printf

- La méthode System.out.printf possède la même syntaxe que le printf de C

```
class A {
Integer a = 1, b = 2, c = 3, d =4;
void affiche( ) {
     System.out.printf("%d, %d, %d, %d" ,  a, b, c,
d) ;
}
public String toString( ){
  return a + ", " + b + ", " + c ", " + ", " + d ;
}
public static void main(String[] args){
      A v1 = new A( ) ;
      v1.affiche () ;                //1, 2, 3, 4
      System.out.print(v1);// 1, 2, 3, 4
 }
}
```

L'initialisation des classes en JAVA: les constructeurs.

- Les constructeurs des classes, sont des méthodes spéciales qui n'ont pas de type de retour et qui ont exactement le même nom que la classe

```
class A {
// Définition de 4 attributs
Integer a = 1, b = 2, c = 3, d =4;
// Définition de deux constructeurs
A(){ a = 10; b = 20; c = 30; d = 40 ;}
A(int v0, int v1, int v2, int v3)
{a = v0; b = v1; c = v2; d = v3; }
// String toString() est utilisée par print
public String toString()
{return A + " " + B + " " + C + " " + D ;}
// Programme main de la classe A.
public static void main(String[] args) {
      A v1 = new A();
      A v2 = new A(11, 12, 13, 14);
      System.out.print(v1);// 10 20 30 40
      System.out.print(v2);// 11 12 13 14
}
}
```

Héritage: un mécanisme de réutilisation des classes entre elles 1/2

- L'héritage est un mécanisme de la programmation orientée-objet qui permet de réutiliser les attributs et les méthodes d'une classe lors de la définition d'une nouvelles classe:

```
class A {
Integer a = 1, b = 2, c = 3, d =4;
Integer F1( Integer a) {
        return a*(this.a + b + c + d) ;
}
}
Classe B extends A {
        // e et r sont deux nouveaux attributs de B
        Integer e = 5, r= 6;

        // F2 est une nouvelle méthode de B
        Integer  F2(Integer b) {
            // Réutilisation de a de b et de F1
            return b * (F1(a) + a*5 + a*r);
         }

    }
}
```

Héritage: la notion de réutilisation des classes entre elles 2/2

- L'héritage est un mécanisme de la programmation orientée-objet qui permet de réutiliser les attributs et les méthodes d'une classe lors de la définition d'une nouvelles classe:

```
class A {
Integer a = 1, b = 2, c = 3, d =4;
Integer F1( Integer a) {
        return a*(this.a + b + c + d) ;
}
}
// B utilise l'héritage
Classe B extends A {
        Integer e = 5, r= 6;
        Integer  F2(Integer b) {
           return b * (F1(a) + a*5 + a*r);
         }
}
// La classe C fait comme A SANS héritage
class C {
        A v; // v de type A
      Integer e = 5, r= 6;
         Integer  F2(Integer b) {
             return ( v.b * ( v.F1(v.a) + v.a * 5 + v.a*r));
        }
}
```

Héritage : Classe Générique qui étend une classe qui est elle-même extension d'une autre classe

- class CClass<U extends AClass>{
- U a1;
- U geta1(){return a1;}
- }

Héritage : Exclusivement du MONO-HERITAGE !

- Une classe en JAVA hérite au plus/au maximum d'une seule classe : on appelle cela du monohéritage (une classe parente au plus/au maximum)

```
class B{...}
class C{...}
class D{...}
class A extends B, C, D {...}
}
```

Héritage : on peut implémenter plusieurs interface MAIS hériter d'une classe au plus!

```
1)/* On peut exiger qu'un type générique implémente plusieurs interfaces.
2) * Le mot clef utilisé est extends et NON PAS implements
3) * On peut AU PLUS mettre une classe dans la restriction du type générique!
4) * Toutes les autres devront être des interfaces!
5) * La classe devra être donnée en premier si il y'a une classe. */
6)              interface I{}
7)              interface J{}
8)              interface K{}
9)
10)             class E{ }
11)             class EClass<T extends E & I & J & K>
12)            {
13)              T a;
14)              T b;
15)            }
```

Définition incomplète de classes en JAVA: les classes abstraites.

- Les classes abstraites sont des classes dont certaines méthodes n'ont pas encore reçues de définition.

```
abstract class A {
Integer a = 1, b = 2, c = 3, d =4;
public String toString()
{return A + " " + B + " " + C + " " + D ;}
public abstract Integer m1(int p);
public static void main(String[] args) {
// new INTERDIT POUR UNE CLASSE ABSTRAITE
!!!!
      A   v = new A();
      }
}
class B extends A{
      // Définition de m1
      public Integer m1(int p){return p + 1;}
public static void main(String[] args) {
// new OK MAINTENANT !!!!
      B b = new B() ;
  }
}
```

L’héritage multiple est interdit en JAVA… les interfaces sont alors utilisées!

```
Interface I {…}
Interface J {…}
Interface K{…}

class A implements I, J, K {…}
```

- Les interfaces sont des classes abstraites

Les interfaces: classes abstraites

- Les attributs sont finaux et statiques

```
interface K { }

// La classe S est définie à la suite ; elle est utilisée pour public toString()
class A extends S implements K { }
class B extends S implements K { }

// La classe S est utilisée pour imprimer correctement chaque type d'instance.
class S{
    public String toString(){

            if (this instanceof A) return "A";

            else if (this instanceof B) return "B";

            else return "S";
    }
}
```

```
class Coli {

    public static void main(String[] args){
       K [ ] ai = new I[7];
       // ON MET DANS LE TABLEAU DES OBJETS DIFFERENTS
       ai[0]=new A();
       ai[1]=new B();

       // toString( ) les imprime correctement voir class S
       for(int i=0;i<ai.length;i++)
           System.out.println(ai[i]);

       // On fait pareil à l'aide d'une liste chainée
       LinkedList<K> li = new LinkedList<K>();
       li.add(new A());
       li.add(new B());
       System.out.println(li);
    }
```

Que permettent les interfaces de plus que les classes

1. Les interfaces permettent un MULTI-HERITAGE limité aux constantes et aux déclarations de fonctions

Déclaration d'une classe interne

```
1.  class A{  }
2. class B extends A{  }
3.  class C extends B{
4.     // Declaration D dans C (interne)
5.     class D{  }
6.     // i déclarer de type D dans C
7.     D i; // La classe D n'est visible dans C
```

Utiliser static avec une classe

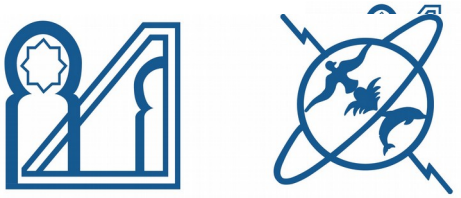
```
1. class A{  }
2. class B extends A{  }
3. class C extends B{
4.    // Declaration D dans C (interne)
5.    static class D{  }
6.    // le type C.D au top-level
7.    class Main { C.D d=new C.D();}
```

Conclusion

- Les classes peuvent hériter entres-elles les attributs et les méthodes

Perspectives

- La programmation orienté-objet peut se greffer au-dessus de la programmation procédurale ou même fonctionnelle.



Chapitre III : Les entrées-sorties en JAVA

Younès EL AMRANI

2015

Introduction

Le langage JAVA est un langage très évolué, comme tous les langages de haut niveau, son rapport avec les périphériques d'entrée-sorties s'avère plus complexe à mettre en place que pour un langage considéré comme bas-niveau comme le C.

E/S en JAVA

Les entrées-sorties en JAVA peuvent se faire à partir de plusieurs entrée possibles :

1. à partir de l'entrée standard et/ou
2. à partir d'un fichier FILE en utilisant un Scanner (java.util.Scanner) et/ou
3. à partir des paramètres en ligne en utilisant le paramètre du main(String[] args) ou partir d'un Stream sur le réseau.

E/S en JAVA à partir d'un fichier en utilisant un Scanner

1. Les entrées-sorties en JAVA peuvent se faire à partir de l'entrée standard ou d'un fichier FILE en utilisant un Scanner (java.util.Scanner)

```
File file = new File("ScanFile.java");
```

```
import java.util.Scanner
```

```
Scanner scan = new Scanner(file);
```

Lecture au clavier avec Scanner

1.Pour lire a clavier, il faut initialiser le Scanner avec l'entrée standard :

```
Scanner scan = new Scanner(System.in) ;
```

Les constructeurs de Scanner prennent : un Stream, un File, un Canal, une chaîne...

1.public java.util.Scanner(java.lang.Readable);

2. public java.util.Scanner(java.io.InputStream);

3. public java.util.Scanner(java.io.InputStream, java.lang.String);

4. public java.util.Scanner(java.io.File) throws java.io.FileNotFoundException;

5. public java.util.Scanner(java.io.File, java.lang.String) throws java.io.FileNotFoundException;

6. public java.util.Scanner(java.nio.file.Path) throws java.io.IOException;

7. public java.util.Scanner(java.nio.file.Path, java.lang.String) throws java.io.IOException;

8. public java.util.Scanner(java.lang.String);

9. public java.util.Scanner(java.nio.channels.ReadableByteChannel);

10.public java.util.Scanner(java.nio.channels.ReadableByteChannel, java.lang.String);

Méthodes de la classe Scanner qui sont utilisées pour la lecture

La classe Scanner offre plusieurs méthodes d'entrées-sorties prêtent à l'emploi :

1. public boolean hasNext();
2. public java.lang.Object next()
3. public java.lang.String nextLine();
4. public int nextInt();
5. public double nextDouble();

Des méthodes Booléennes de la classe Scanner pour tester l'entrée.

1. public boolean hasNext();
2. public boolean hasNextBoolean(); <=>public boolean nextBoolean() ;
3. public boolean hasNextByte(); <=>public Byte nextByte() ;
4. public boolean hasNextFloat(); <=>public float nextFloat();
5. public boolean hasNextDouble(); <=>public double nextDouble()

Exemple pour compter les caractères d'un fichier

```
1. import java.util.Scanner
2. class Test {
3. public static void main(String[ ] args)
4.   throws FileNotFoundException
5. {
6. File file = new File("unfichier.java");
7. Scanner scan = new Scanner(file) ;
8. int count = 0;
9.     while (scan.hasNext())
10.      {
11.         count++;
12.         scan.next();
13.      }
14.   System.out.println("le nombre de caractères est :" + count) ;
15.}
16.
```

Exemple en dix lignes pour compter les caractères de plusieurs fichiers

```
1.  class CountChar{
2.      public static void main(String[ ] args)
3.          throws FileNotFoundException { // Fichier non trouvé
4.  int count=0; // On initialise le compteur à zéro
5.  for( String s: args) { // pour chaque nom de fichier
6.      Scanner scan =
7.          new Scanner(new File(s));  //File file = new File(s);
8.      while (scan.hasNext()){count++;scan.next();    } // tout lire
9.  } // fin du for : tous les fichiers ont été lus
10. System.out.println("LE NOMBRE TOTAL DE CARACTERES EST
 EGAL A:  "+count);    }  }
```

Exemple en 10 lignes pour compter le nombre de mots (word) avec un StreamTokenizer

```
1.  class CountWord{
2.      public static void main(String[ ] args)
3.   throws FileNotFoundException, IOException { // pas de fichier ou entrée incorrecte
4.  int count=0, token=0;
5.  for( String s: args) { // Le StreamTokenizer « sait » distinguer les mots des nombres
6.      StreamTokenizer streamTok = new StreamTokenizer(new FileReader(new
    File(s)));
7.   Do { // On test si on a lu un mot : le « token » lu est un TT_WORD
8.      if( StreamTokenizer.TT_WORD == (token=streamTok.nextToken()))
9.          Count++;  // On incrémente le compteur à chaque mot lu
10.    } while( StreamTokenizer.TT_EOF != token ); // fin du do-while à la fin du fichier
11. }
12. System.out.println("LE NOMBRE TOTAL DE MOTS EST EGAL A:  "+count);    } }
```

Sortie vers un fichier en utilisant FileOutputStream avec print

Il est possible d'écrire dans un fichier directement en utilisant FileOutputStream pour ouvrir un fichier et la méthode print :

```
1.  class PiF{
2.      public static void main(String[] args)
3.                          throws FileNotFoundException {
4.  PrintStream print_out
5.    = new PrintStream(new FileOutputStream("output.text"));
6.  print_out.println("Hello, World!");
7.  System.out.println("Le msg Hello, World! est dans
 output.text");
8.      }
9.  }
```

Les arguments en Ligne

La fonction main fournit un tableau de chaînes de caractères qui encodent les arguments en ligne,

Exemples :

```
double d = Double.valueOf(args[i]) ; // double
Integer   i = Integer.getInteger(args[ i ] ) ; // entier
```

Lecture d'un Double à partir d'une chaîne de caractères

1.public static java.lang.Double valueOf(java.lang.String) throws java.lang.NumberFormatException;

2. public static java.lang.Double valueOf(double);

Plusieurs arguments en ligne

Université Mohammed V
Faculté des Sciences
Rabat

```
1.  class MinMax{
2.      public static void main( String[ ] args ){
3.  Integer minimum, maximum ; // Compute the maximum and the minimum
4.  if( args.length < 5 ) {
5.          System.out.print( "usage: provide five integers:" ) ;
6.          System.exit( 0 ) ;   }
7.  Else{  Integer x = Integer.valueOf( args[ 0 ] ) ;
8.              minimum = x ;   maximum = x ;
9.  // Compute the maximum and the minimum
10. for ( int i = 1 ; i < args.length ; i = i + 1 ){
11.   x = Integer.valueOf( args[ i ] ) ;
12.   if( x < minimum )minimum = x ;else if(x > maximum) maximum = x;}
13. System.out.println( "Minimum = " + minimum ) ;
14. System.out.println( "Maximum = " + maximum ) ;    }    } }
```

E/S interactives avec JOptionPane

```
1.import javax.swing.JOptionPane;
2.class Dialog{
3.    public static int getInt(String msg){
4.return Integer.valueOf( JOptionPane.showInputDialog(msg));
5.    }
6.    public static double getDouble(String msg){
7.return Double.valueOf( JOptionPane.showInputDialog(msg));
8.    }
9.    public static void affiche(String msg){
10JOptionPane.showMessageDialog(null,msg,"
",JOptionPane.PLAIN_MESSAGE );     }    }
```

En conclusion

1.Les entrées-sorties en JAVA se font de manière relativement uniforme quelque soir type du flôt en entréé (Stream) que ce soit un fichier, ou le clavier ou le réseau.

Perspectives

1.Les entrées-sorties futuristes incluent la reconnaissance vocale, la reconnaissance gestuel de l'utilisateur, les interactions (déjà bien éprouvées) tactiles avec le clavier ou l'écran !

Université Mohammed V
Faculté des Sciences
Rabat

Chapitre 4 : Les Exceptions

Exceptions in JAVA
Les Exceptions en JAVA
The Path to Robust programming
Le chemin vers une programmation robuste

Pr. Younès EL AMRANI

Introduction

- Les exceptions en JAVA sont une élégante machinerie, une furtive machination pour séparer le code utile du traitement des erreurs et situations exceptionnelles. Il est bien entendu que le traitement des erreurs et au moins aussi utile que nécessaire à la fiabilité du code et à sa robustesse.
- Robuste dans le sens résistant aux erreurs mais aussi résistant aux situations exceptionnelles susceptibles de briser la session d'exécution courante d'un programme JAVA.

Définition d'une exception.

- Une Exception est plus générale qu'une erreur : c'est une situation prévisible mais qui ne survient pas systématiquement, on identifie ce type de situation et on lui donne un nom : c'est la définition de l'exception proprement dite.

```
class E1 extends Exception{…}
class E2 extends Exception{…}
```

Les conséquences à une Exception

- Lorsque survient une situation exceptionnelle, y compris un cas d'erreur, il y-a trois réactions possibles du programme :
- 1) le programme s'arrête ou est arrêté par le système d'exploitation car une ressource manque ou le programme a violé le mémoire vive ou autre limite à son exécution
- 2) le programme renvoie une valeur qui traduit l'erreur ou la situation exceptionnelle à la fonction appelante
- 3) le programme va détecter l'erreur et continuer sur un programme élaboré en vue de récupérer la situation et éviter l'arrêt brutal de l'exécution.

Prévoir une Exception

- Lorsque le programme détecte une erreur ou une situation exceptionnelle et qu'il est dans une des deux configurations suivantes soit de retourner un code d'erreur ou de traiter sur place la situation exceptionnelle, alors une Exception peut-être rajoutée dans cette situation.

Le bloc try-catch

- La structure du code autour duquel s'articule la levée (l'apparition, l'occurrence) des Exceptions et le traitement qui s'ensuivra, s'appelle un bloc **TRY-CATCH**

Exemples de situations Exceptionnelles

Les situations exceptionnelles sont liées généralement soit à une erreur d'entrée sortie, soit à une ressource périphérique manquante, soit à une opération de calcul erronée du programme.

Implémentation d'une Exception

Les Exceptions, telles que nous les avons vu, sont implémentées comme des instances de la classe Exception, quelle que soit leur nature (erreur de calcul, erreur d'entrée-sortie, ressource défectueuse, etc.)

class E1 extends Exception {…}

class E2 extends Exception {...}

Cependant, il est possible de définir une relation d'héritage entre les Exceptions définies par l'utilisateur.

class E1 extends Exception {…}

class E2 extends E1 {…} // E2 est une spécialisation de E1

Il faudra bien garder en vue que les blocs catch devront commencer par l'exception la plus spécialisée en allant (vers le bas du code) en direction de l'exception la plus générale, car autrement, les exceptions sous-classées ne seraient pas visibles et ne seraient jamais atteintes.

catch(E2 e){ … Traitement du cas E2, plus **spécial** que E1, …}

catch(E1 e){ … Traitement du cas E1, plus **général** que E1, ...}

Traitement d'une Exception

L'introduction des Exceptions ne dédouane par le programmeur du traitement de l'erreur ou de la situation exceptionnelle :

try { … **throw** new E1( ) ; …}

catch (E1 e) { … Traitement de l'Exception E1 … }

Il faut traiter les Exceptions

Si aucune portion de code pour traiter une exception n'existe, alors le programme s'arrête en affichant la pile d'exécution.

Cependant, cet arrêt du programme intempestif doit être évité à tout prix, c'est pourquoi, une la portion de code est toujours prévu pour pallier au problème de plusieurs manières généralement

Pourquoi parle t-on de try-throw-catch ?

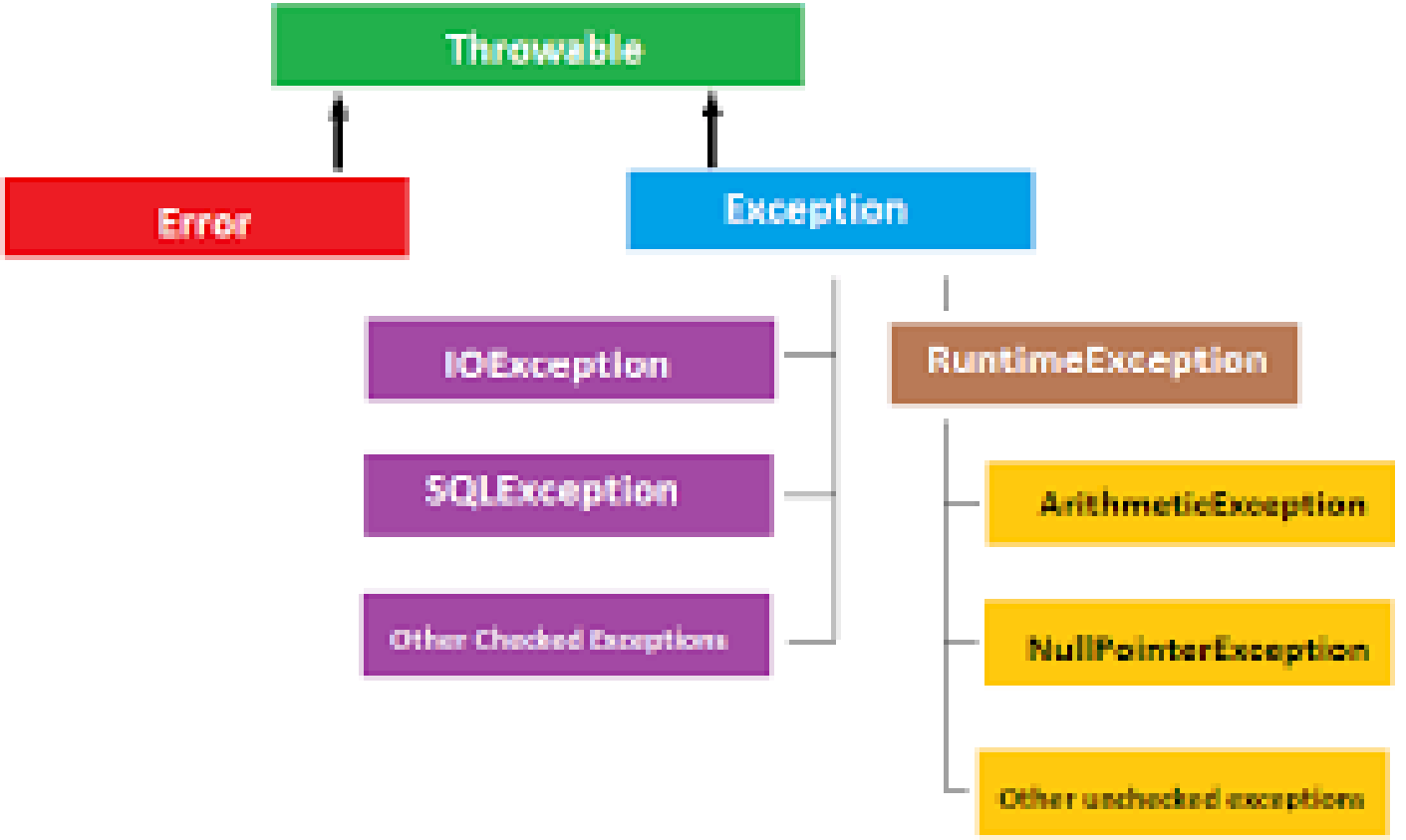
La notion d'Exception est emprunté au langage populaire du base-ball, ainsi l'erreur ou la situation exceptionnelle est vue comme une balle qui est lancée par le joueur, on parle de **throw,** tandis que le traitement de l'exception est perçu comme le joueur qui réussi à intercepter la balle, on parle de la situation de **catch** ;

Le bloc dans lequel se trouve le code utile qui pourrait comporter des déclenchement ou des levées d'Exception est appelé le bloc **try** ; c'est exactement la notion d'un essai au sens footballistique dans le jargon des jeu de balles.

Cette terminologie du jeu le plus populaire des état-unis après le Basket a donc pénétré le langage JAVA pour le rendre encore plus familier aux programmeurs anglophones, très friands de matchs **try-throw-catch**

Cependant, le mécanisme d'Exception n'est pas né avec JAVA, c'est un mécanisme bien plus ancien, apparu dans les premiers langages à Objets des années 80.

Hiérarchie des Exceptions en couleurs



Exception : c'est un Throwable + 5 constructeurs

```
public class java.lang.Exception extends java.lang.Throwable {
  static final long serialVersionUID;
  public java.lang.Exception();
  public java.lang.Exception(java.lang.String);//Constructeur
  public java.lang.Exception(java.lang.String, java.lang.Throwable);
  public java.lang.Exception(java.lang.Throwable);
  protected java.lang.Exception(java.lang.String,
java.lang.Throwable, boolean, boolean);
}
```

Throwable : 5 constructeurs + getMessage() + printStackTrace() ...

public class java.lang.Throwable implements java.io.Serializable {

public java.lang.Throwable();

public java.lang.Throwable(java.lang.String);

public java.lang.Throwable(java.lang.String, java.lang.Throwable);

public java.lang.Throwable(java.lang.Throwable);

protected java.lang.Throwable(java.lang.String, java.lang.Throwable, boolean, boolean);

public java.lang.String getMessage();

…

public java.lang.String toString();

public void printStackTrace();

…

Plusieurs Exceptions sont prédéfinies en JAVA

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

Exemple avec l'exception prédéfinie ArrayOutOfBoundsException

```
int[ ] tabint={1,2,3,4,5,6,7,8,9,10};
try {
   for (int i=0;i<12;i++){
      System.out.println(tabint[i]) ;
 } catch ( ArrayOutOfBoundsException e){
      System.out.println( ¨NO TENTH ELEMENT !¨) ;
}
```

L'exception se produit généralement dans le bloc try.

Les exceptions sont toujours traitées dans un bloc catch. Locale ou dans les couches supérieurs

Structure Générale d'un bloc try-catch

```
Instruction[ ] instructions={declare(),compute(),...,return};
try {
   for (int i=0;i<instructions.length;i++){
      Execute(instructions[ i ] ) ;
 } catch ( Exception1   e) { Process Exception 1 }
 } catch ( Exception2   e) { Process Exception 2 }
 } catch ( Exception3   e) { Process Exception 3 }
 } catch ( Exception4   e) { Process Exception 4 }
 } finally {  Bloc finally is always executed, whatever happens in the try }
}
```

L'exception devra se produire dans le bloc try.

Les exceptions sont toujours traitées dans un bloc catch. Locale ou dans les couches supérieurs

Que se passe t-il dans un bloc try-catch-finally

- Le bloc try est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- 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).
- Les clauses catch doivent donc traiter les exceptions de la plus spécifique à la plus générale.
- Si une clause catch convenant à cette exception a été trouvée et le bloc exécuté, l'exécution du programme reprend son cours

Si aucun catch correspondant n'a été trouvé, l'Exception devra être reportée **thrown** par la fonction contenant le bloc **try-catch-finally**

Remontée d'une Exception jusqu'au main

```
1)  class E1 extends Exception{} class E2 extends Exception{}
2)  class E3 extends Exception{}class E4 extends Exception{}
3)  class ExpMeth{
4)     void m1() throws E1,E2{ int flip=1; Random ran=new Random(); flip=1+ran.nextInt(4);
5)  try{
6)       if (flip==1)     throw new E1();
7)       else if(flip==2) throw new E2();
8)       else if(flip==3) throw new E3();
9)      else if(flip==4) throw new E4();
10)     else{System.out.println("NO EXCEPTION");}}//end try
11) catch(E3 e){System.out.println("exception E3");}
12)  catch(E4 e){System.out.println("exception E4");}     }
13)  public static void main(String[] args){
14) ExpMeth em=new ExpMeth();
15) for(int i=0;i<10;i++)
16)    try{em.m1();}
17)    catch(E1 e){System.out.println("Exception E1 dans main");}
18)    catch(E2 e){System.out.println("Exception E2 dans main");}    }  }
```

Interception versus Propagation

Si une méthode émet une exception, alors, il y-a deux cas possibles :

- soit **propager** l'exception : dans ce cas, lla méthode doit la déclarer;
- soit faire un **catch** localement de l'exception et la traiter dans le bloc catch.

Les messages de la classe Exception qui hérite de Throwable

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

Récupération du message passé au constructeur à l'instant du throw

```
public class MonException extends Exception   {
  public MonException( ) {   // Constructeur sans passer de message
      super( ); // ici aucun message n'est passé
    }
  public MonException(String s)  {  // constructeur avec message
      super(s); // ici on passe un message dans le constructeur
                // la classe super est is la classe Throwable
    }
}
```

Utilisation de la clause throws pour la propagation des exceptions vers le haut

```
class E1 extends Exception{  }
public void m1 (  )   throws  E1     {
     // ici m1 ne traite pas l'exception E1
     // mais m1 peut génére E1 avec throw
  }
```

Pourquoi une méthode DOIT indiquer les Exception qu'elle pourrait générer sans les traiter

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.

Conclusion

- Grâce aux exceptions, on peut écrire du code « robuste » qui « résiste » aux erreurs et aux situations exceptionnelles qui apparaissent sur la plate-forme d'exécution.

Perspectives

- La robustesse du code est d'abord et avant tout, la capacité du programmeur à prévoir les situations exceptionnelles ainsi que les cas d'erreurs

Chapitre 5: La Généricité en JAVA

La généricité en JAVA permet
d'écrire une classe avec un (ou plusieurs) type(s)
en paramètre

La Généricité en JAVA : Les Classes Génériques avec des Types en Paramètres !

Introduction du chapitre 5

La généricité est l'appellation du paramétrage des types dans un langage de programmation.

Classe générique avec un paramètre

```
1.class ZClass<T> {
2.    private T a1, a2, a3;     //Définition des attributs qui utilisent le type générique
3.    public ZClass(T a){     //Définition d'un constructeur de la classe, avec un paramètre de type g'en'erique
4.           a1 = a2 = a3 = a;
5.    }
6.    public String toString(){  //Définition de la méthode public toString() qui facilite l'affichage
7.         return "("+a1.toString() + ", " + a2.toString() + ", " + a3.toString() + ")";
8.    }
9.                  // Les méthodes de lecture: les "get" méthodes
10.  public T geta1(){return a1;}
11.  public T geta2(){return a2;}
12.  public T geta3(){return a3;}
13.
14.                  // Les méthodes d'écriture, les "set" méthodes
15.  public void seta1(T a1){this.a1=a1;}
16.  public void seta2(T a2){this.a2=a2;}
17.  public void seta3(T a3){this.a3=a3;}
18.
```

Instanciation d'une classe générique

Exemple :

```
HashMap<String, Integer> = new
HashMap<String,Integer>( ) ;
```

Exemple de classe générique

```
class D<T1>{
    T1 a,b,c,d,e,f;
}
 // Utilisation de D dans une autre classe :
 class Main{
    public static void main(String[] args)
    {
        D<String> obj4 = new D<String>();
        // Tous les attributs a,b,c,d,e,f sont de type String
    }
 }
```

Exemple 2 d'une classe générique

```
1.  class AClass<T> {
2.      private T a1, a2;
3.      public String toString()
4.          {return "("+a1.toString() + ", " + a2.toString() + ", " + ")";    }
5.      public T geta1(){return a1;}    public T geta2(){return a2;}
6.      public void seta1(T a1){this.a1=a1;}
7.      public void seta2(T a2){this.a2=a2;}
8.      public static void main(String[] args)    {
9.          AClass<Integer> o1 = new AClass<Integer>();
10.         o1.seta1(10);o1.seta2(11);o1.seta3(12);
11.         AClass<String> o2 = new AClass<String>();
12.          o2.seta1("BONJOUR");o2.seta2("BONSOIR");
13.         System.out.println("a3= " + o2.geta2( ));    } }
```

Exemple de classe générique avec deux paramètres

```
class C2<T1,T2> {
    T1 a1;
    T2 a2;
} // Illustration de C2 dans une autre class appelée Main
class Main{
    public static void main(String[ ] args)
    {
        C2<Double, Integer> obj =
                new C2<Double, Integer>();
        C2<String, Double> obj2 =
                new C2<String, Double>();
    }
```

Généricité : getters and setters génériques


```
class ZClass<T> {
    private T a1, a2, a3;   //Définition des attributs qui utilisent le type générique T
   public ZClass(T a){              a1 = a2 = a3 = a;   } // Le constructeur utilise T
   public String toString(){  //Définition de la méthode public toString()
            return "("+a1.toString() + ", " + a2.toString() + ", " + a3.toString() + ")";
    }
    // Les méthodes de lecture: les "get" m'ethodes
    public T geta1(){return a1;}
    public T geta2(){return a2;}
    public T geta3(){return a3;}
    // Les méthodes d’écriture, les "set" methodes
    public void seta1(T a1){this.a1=a1;}
    public void seta2(T a2){this.a2=a2;}
    public void seta3(T a3){this.a3=a3;}
```

Utilisation de Zclass avec T=Integer et T=String

```
public static void main(String[] args)
{
  ZClass<Integer> o1 = new ZClass<Integer>(10);
  o1.seta2(11);
  o1.seta3(12);
  System.out.println(o1);

  ZClass<String> o2 = new ZClass<String>("Hello");
  o2.seta2("ASSALAMOU-ALAIKOUM");
  o2.seta3("WA-ALAIKOUMOU-SALAM-WA-RAHMATOU-ALLAH");
  System.out.println(o2);
}
```

Classe Générique qui étend une classe Générique

- Une classe qui étend une autre classe générique DOIT contenir les mêmes paramètres génériques et, éventuellement, rajouter de nouveaux paramètres génériques.

Héritage et Généricité : Règles à respecter

- **Règle 1** :
- Une classe contient au moins les mêmes paramêtres de type
- Que la classe qu'elle étend mais elle peut en rajouter (cas de U,V,W)
- **Règle 2** : on ne peut pas diminuer la restriction sur un type générique

Définition d'une classe Générique avec deux paramètres

Université Mohammed V
Faculté des Sciences
Rabat

```
1.class C2<T1,T2> // On donne entre crochets les types en paramètres
2.{
3.    T1 a1;
4.    T2 a2
5.    public static void main(String[] args)
6.    {
7.      C2<Double, Integer> obj = new C2<Double, Integer>();
8.      C2<String, Double> obj2 = new C2<String, Double>();
9.      C2<C,C> obj3 = new C2<C,C>();
10.     D<String> obj4 = new D<String>();
11.   }
12.  }
```

Les classes génériques de java .util ; très pratiques !!

- Collection<E>
- Comparator<T>
- Enumeration<E>
- Iterator<E>
- List<E>
- ListIterator<E>
- Map<K,V>
- Map.Entry<K,V>
- Queue<E>
- Set<E>
- SortedMap<K,V>
- SortedSet<E>

Exemple d'interface générique pour introduire une relation d'ordre : Comparable

```
public interface Comparable<T>{
   int compareTo(T o1);
}
```

Utilisation de Comparable<T>


```
1.class Point implements Comparable<Point>
2.{
3.    Double x,y,z;
4.
5.    Point(Double x,Double y,Double z)
6.    {
7.      this.x = x; this.y = y; this.z = z;
8.    }
9.
10.     public int compareTo(Point p)
11.    {
12.    if(this.x < p.x )
13.        return -1;
14.    else if (this.x == p.x )
15.        return 0;
16.    else
17.        return 1;
18.    }
```

Utilisation de Comparable avec une LinkedList


```
1.      public static void main(String[] args){
2.        Point p1 = new Point(1.0, 6.0, 16.0),
3.              p2 = new Point(2.0, 5.0, 34.0),
4.              p3 = new Point(3.0, 4.0, 25.0);
5.        // Ici on utilise une LinkedList formée des trois Points p1,p2,p3
6.         LinkedList<Point> LP = new LinkedList<Point>(  );
7.         LP.add(p1); LP.add(p2); LP.add(p3);
8.        // On utilise la fonction prédéfinie dans Collections pour trouver le max
9.         Point pmaxSelonX = Collections.max(LP); // Ordre de Comparable
10.    System.out.println("Voici le point maximum selon les X: " + pmaxSelonX);
```

Méthodes disponible de Collections grâces à Comparable: sort ; min; max

1. public **static** <T extends java/lang/Comparable<? super T>>
   void **sort**(java.util.List<T>);

2. public **static** <T extends java/lang/Object>
   int **binarySearch**(java.util.List<? extends java.lang.Comparable<? super T>>, T);

3. public **static**
   void **reverse**(java.util.List<?>);

4. public **static** <T extends java/lang/Object & java/lang/Comparable<? super T>>
   T **min**(java.util.Collection<? extends T>);

5. public **static** <T extends java/lang/Object & java/lang/Comparable<? super T>>
   T **max**(java.util.Collection<? extends T>);

Utilisation de Comparable<T>, LinkedList<T> et Collections.max

```
class Main
{
    public static void main(String[] args)
    {
       Point
          p1 = new Point(1.0, 6.0, 16.0),
          p2 = new Point(2.0, 5.0, 34.0),
          p3 = new Point(3.0, 4.0, 25.0);

       LinkedList<Point> LP = new LinkedList<Point>();
       LP.add(p1); LP.add(p2); LP.add(p3);

       Point pmaxSelonX = Collections.max(LP);

       System.out.println("Voici le point maximum selon les X: " +
pmaxSelonX);
    }
}
```

```
class Point implements
Comparable<Point>
{
    Double x,y,z;

    public int
compareTo(Point p)
    {
        if( this.x < p.x )
            return -1;
        else if (this.x == p.x )
            return 0;
        else
            return 1;
    }
}
```

Pour définir plusieurs relations d'ordre : utiliser Comparator

```
public interface java.util.Comparator<T> {
  public abstract int compare(T t1, T t2);
}
```

Définition d'un Comparator selon l'axe des X

```
1.class OrdreX implements Comparator<Point>{
2.
3.  public int compare(Point p1, Point p2){
4.    if( p1.X < p2.X )
5.        return -1;
6.    else if( p1.X == p2.X )
7.        return 0;
8.    else
9.        return 1;    }
10.  }
```

Définition d'un Comparator selon l'axe des X

A l'appel on écrira :

```
Collections.max( LP , new CompX( ) ) ;
```

On pourra écrire aussi la même chose comme suit :

```
CompX comp = new CompX( ) ;

Collections.max( LP , comp ) ;
```

Avec Comparable on s'est contenté d'écrire

```
Collections.max(LP) ;
```

Définition de compare dans la liste des arguments

```
1.      Point pmaxSelonY = Collections.max(LP ,
2.         new Comparator<Point>()
3.         {
4.           public int compare( Point obj1, Point obj2)
5.           {
6.              if( obj1.y < obj2.y)
7.                return -1;
8.              else if( obj1.y == obj2.y )
9.                return 0;
10.             else
11.               return 1;
12.          } // Fin du corps de compare
13.        } // Fin du new Comparator
14.                                  ) ; // Fin d'appel Collections.max
```

Utilisation des lambdas pour définir un Comparator

Université Mohammed V
Faculté des Sciences
Rabat

```
1. //////////
2. // Utilisation des Lambdas Expressions pour Comparator
3. ////
4. Point pmaxX = Collections.max(LP ,
                                  (p1,p2)->{if(p1.x < p2.x)return -1;else if(p1.x == p2.x )return 0;else return 1;});
5. Point pmaxY = Collections.max(LP ,
                                  (p1,p2)->{if(p1.y < p2.y)return -1;else if(p1.y == p2.y )return 0;else return 1;});
6. Point pmaxZ = Collections.max(LP ,
                                  (p1,p2)->{if(p1.z < p2.z)return -1;else if(p1.z == p2.z )return 0; else return 1;});
7. System.out.println("Max/X="+pmaxX+"\nMax/Y="+pmaxY+"\nMax/Z="+pmaxZ);
```

Amélioration du style avec des Lambdas initialisées avant l'appel

Université Mohammed V
Faculté des Sciences
Rabat

1) Comparator<Point> copX = (p1,p2)->{if(p1.x < p2.x)return -1;else if(p1.x == p2.x)return 0;else return 1;} ;

2) Comparator<Point> copY = (p1,p2)->{if(p1.y < p2.y)return -1;else if(p1.y == p2.y)return 0;else return 1;} ;

3) Comparator<Point> copZ = (p1,p2)->{if(p1.z < p2.z)return -1;else if(p1.z == p2.z)return 0; else return 1;};

4) pmaxX = Collections.max(LP, copX);

5) pmaxY = Collections.max(LP, copY);

6) pmaxZ = Collections.max(LP, copZ);

7) System.out.println("Max/X="+pmaxX+"\nMax/Y="+pmaxY+"\nMax/Z="+pmaxZ);

Définition de plusieurs distances et ordonnancement

```
1)  class Point implements Comparable<Point> { double x,y,z,t,u;
2)      public double norme2(){ // Norme euclidienne utilisée par CompE
3)          return Math.sqrt(Math.pow(x,2.0)+Math.pow(y,2.0)
4)              +Math.pow(z,2.0)+Math.pow(t,2.0)+Math.pow(u,2.0));}
5)      public double norme1(){ // Norme Manhattan, utilisée par CompM
6)          return Math.abs(x)+ Math.abs(y)+ Math.abs(z)+ Math.abs(t)+ Math.abs(u);}
7)      public double norme8( ){ // Norme infinie, utilisée par Comp8
8)          return Math.max(Math.max(Math.max(Math.max
(Math.abs(x),Math.abs(y)),Math.abs(z)),Math.abs(t)),Math.abs(u));}
9)      public double m1( ){return x*y*(z+t+u);}
10)     public double m2( ){return Math.pow(x+y,z+t+u);}
11)     public Point( ){x=Math.random()*10;y=Math.random()*10;z=Math.random()*10;
12)         t=Math.random()*10;u=Math.random()*10;}
13)     @Override // Pour l'affichage d'un Point
14)     public String toString(  ){String s="";s+="\n("+x+","+y+","+z+","+t+","+u+")";return s;}
```

Définition d'un ordre naturel avec Comparable<Point>

```
1.  //////////
2.  //      La méthode compareTo, déclarée par l'interface Comparable<Point>
3.  //      est utilisée par tous les méthodes sort, min et max de Collections
4.  //       qui utilisent un seul paramêtre:
5.  //       les autres versions avec Comparator<Point> utilisent deux paramètres
6.  /////
7.  @Override
8.  public int compareTo(Point p){
9.        double nm1 = this.norme2(), nm2 = p.norme2();
10.       if(nm1>nm2) return 1;else if(nm1<nm2) return -1;else return 0;
11. }
```

On peut aussi définir un Comparator pour la norme euclidienne

Université Mohammed V
Faculté des Sciences
Rabat

```
1.  //////////
2.  // Comparator pour la norme Euclidienne
3.  /////
4.  class CompE implements Comparator<Point>{
5.      public int compare(Point p1, Point p2){
6.  double nm1=p1.norme2(), nm2=p2.norme2();
7.  if(nm1>nm2) return 1;else if(nm1<nm2) return -1;else return 0;
8.      }
9.  }
```

Définition d'un Comparator pour la norme 1, dite norme de Manhattan


```
1.  //////////
2.  // Comparator pour la norme Manhattan
3.  /////
4.  class CompM implements Comparator<Point>{
5.      public int compare(Point p1, Point p2){
6.  double nm1=p1.norme1(), nm2=p2.norme1();
7.  if(nm1>nm2) return 1;else if(nm1<nm2) return -1;else return
 0;
8.      }
9.  }
```

Définition d'un Comparator pour la norme ∞, dite norme infinie

```
1.  //////////
2.  // Comparator pour la norme infinie
3.  /////
4.  class Comp8 implements Comparator<Point>{
5.      public int compare(Point p1, Point p2){
6.  double nm1=p1.norme8(), nm2=p2.norme8();
7.  if(nm1>nm2) return 1;else if(nm1<nm2) return -1;else return
 0;
8.      }
9.  }
```

Utiliser un Tableau et/ou une liste

```
1.  class Test{
2.       Point[ ]   tabPoint;
3.       List<Point> listePoint;
4.      public Test( ) { // CONSTRUCTEUR
5.      tabPoint   = new Point[20];
6.      listePoint = new LinkedList<Point>();
7.      for(int i=0;i<tabPoint.length;i++) {
8.                tabPoint[i]=new Point();       // Initialise un Point
9.                listePoint.add(new Point()); // Rajoute le Point dans la liste
10.     } // Fin de la boucle for d'initialisation
11.     // On tri le tableau dans le constructeur avec l'ordre naturel
12.         java.util.Arrays.sort(tabPoint);
13.         java.util.Collections.sort(listePoint) ;
```

Tri d'une liste avec Comparable<Point>

```
1.      public static void main(String[] args){
2.          Test t = new Test();
3.           // On tri la liste avec l'ordre naturel
4.           java.util.Collections.sort(t.listePoint);
5.          // System.out.println (sait) trier les
 Lists
6.          System.out.println(t.listePoint);
```

Tri avec Comparator<Point> avec la distance de Manhattan

1. Comparator<Point> **Comp1** = new CompM(); // Le Comparator de la norme 1 (de Manhattan)
2. Collections.sort(t.listePoint,Comp1);
3. System.out.println(t.listePoint);

Tri avec Comparator<Point> avec la norme infinie

1. Comparator<Point> Comp8 = new Comp8();
2. Collections.sort(t.listePoint,Comp8);
3. System.out.println(t.listePoint);

Utilisation de Lambda pour définir le Comparator<Point> de norme infinie

```
1.      Comparator<Point> Comp8bis =
2.          (p1,p2)->{double nm1=p1.norme8(),
3.                                  nm2=p2.norme8();
4.                      if(nm1>nm2) return 1;
5.                      else if(nm1<nm2) return -1;
6.                      else return 0;}; // Fin de la Lambda
7.      Collections.sort(t.listePoint,Comp8bis);
8.      System.out.println(t.listePoint);
```

Perspectives du chapitre 5

1.La généricité est un puissance outil pour la factorisation des classes par rapport à plusieurs types, elle est très utilisée dans les package de JAVA mis à disposition des programmeurs, donc, la connaissance de son mécanisme est nécessaire ;

Conclusion du chapitre 5

1.Les deux interfaces Comparable<T> et Comparator<T> permettent de définir plusieurs ordres différents sur les objets/variables d'une classe.

Chapitre VI : Les Collections en JAVA : Arrays,LinkedList, Map, Sets, Trees...

Auteur : Younès EL AMRANI
Initialement : Automne 2017

Introduction

- La notion de Collections en JAVA est une généralisation des tableaux, listes, ensembles et de toute structure représentant une collection d'élément. Cette généralisation permet de factoriser plusieurs mécanismes pour en permettre un usage uniforme et spontané par le programmeur. La boucle for suivante illustre parfaitement cette idée :
- for (TypeElement elem : CollectionDeElement) {Traitement(elem) ;}
- Les Collections fournissent aussi plusieurs interfaces intuitives qui facilitent une programmation spontanée et intuitive

Définition d'une Collection

- On appelle Collection en JAVA, tout ensemble d'instances d'une classe regroupées au sein d'un tableau, d'une liste, d'un set ou de toute autre structure pouvant rassembler plusieurs instances sous un même nom.

Exemple d’une Collection: le tableau

- L’accès aux éléments d’un tableau est un accès indexé

Exemple avec un tableau d'Élément

```
1.  class Elem{public String toString(){return "Elem";};}
2.
3.  class LesTableaux{
4.      Elem[] tab4={new Elem(),new Elem(),new Elem(),new
    Elem(), new Elem()};
5.  public void test( ) {
6.  System.out.println("\n///  Impression d'un tableau d'Elem:  ///");
7.  for(Elem elem: tab4){
8.      System.out.println(elem);
9.  }
```

Exemple avec un tableau de Lambdas

```
1.  class LesTableaux{
2.    interface SayHello {String say( );} // interface fonctionnelle
3.      SayHello[] tab6={( )->"Salam",
4.                       ( )->"Hello",
5.                       ( )->"Bonjour",
6.                       ( )->"Salut"};
7.      public void test( ){
8.          System.out.println("/// Impression via les Lambdas:///");
9.          for(SayHello sayhello: tab6){
10.             System.out.print(sayhello.say()+" ");  } } }
```

La Collection de base de JAVA: la collection *List*

```
1.  public interface java.util.List<E> extends java.util.Collection<E> {
2.    public abstract int size();
3.    public abstract boolean isEmpty();
4.    public abstract boolean contains(java.lang.Object);
5.    public abstract java.lang.Object[] toArray();
6.    public abstract <T> T[] toArray(T[]);
7.    public abstract boolean add(E);
8.    public abstract boolean remove(java.lang.Object);
9.    public void sort(java.util.Comparator<? super E>);
10.   public abstract void clear();
11.   public abstract void add(int, E);
12.   public abstract int indexOf(java.lang.Object);
13.   public abstract int lastIndexOf(java.lang.Object);
14. } // La déclaration ci-dessus n'est pas complète : c'est un extrait
```

Les Listes Chaînées sont des List

```
1.  public class java.util.LinkedList<E>  implements java.util.List<E> …  {
2.    public java.util.LinkedList(java.util.Collection<? extends E>);
3.    public E getFirst();
4.    public E getLast();
5.    public E removeFirst();
6.    public E removeLast();
7.    public void addFirst(E);
8.    public void addLast(E);
9.    public boolean contains(java.lang.Object);
10.   public int size();
11.   public void clear();
12.   public java.lang.Object[] toArray();
13.   public <T> T[] toArray(T[]);
14. } // La déclaration ci-dessus n'est pas complète : c'est un extrait
```

Utilisation d'une LinkedList : avec « instantiation » d'une interface!

```
1.  import java.util.LinkedList;
2.  interface I{    public String mI(String msg);     }
3.  class Main {
4.      public static void main(String[] args) {
5.  LinkedList<I> listDeI = new LinkedList<I>();
6.  listDeI.add(new I(){public String mI(String msg){return "Salam:   " + msg;}});
7.  listDeI.add(new I(){public String mI(String msg){return "Bonjour: " + msg;}});
8.  listDeI.add(new I(){public String mI(String msg){return "Salut:   " + msg;}});
9.  for( int i = 0 ; i < listDeI.size() ; i++ )
10.     System.out.println( listDeI.get(i).mI("C'est le bonheur Numéro " + i) );   } }
```

Collection utilisant une LinkedList composée de Lambdas

```
1.  interface I{    public String mI(String msg);     }
2.  class Main{
3.     public static void main(String[] args) {
4.      // On instancie une Collection qui est une LinkedList
5.      LinkedList<I> listOfI = new LinkedList<I>();
6.      ListOfI.add(msg->msg+": "+"Salam!"); // On y met des Lambdas
7.      listOfI.add(msg->msg+": "+"Bonjour!");
8.      listOfI.add(msg->msg+": "+"Salut!");
9.      // Maintenant on parcours la Linked List et on fabrique une chaîne s
10.     String s="\n";
11.     for(I i: listOfI){s+=i.say("takalam: ")+"\n";}
12.     System.out.println(s);
```

Map : une Collection qui incarne les applications : Exemple avec HashMap


```
1.  class Main {
2.  public static void main(String[] args) {
3.      System.out.println("usage: java Main mot-1 mot-2 mot-3 ... mot-n");
4.
5.  // On instancie une map qui contiendra un nombre de paires égal à args.length
6.
7.          Map<String, Integer> hashmap =
8.              new HashMap<String, Integer>(args.length);
9.
10. // On rajoute des pairs : on initialise la fréquence d'un mot à zéro
11.
12.         for (String arg : args){hashmap.put(arg,0);}
13. // A chaque fois qu'on rencontre un mot, on rajoute un à sa fréquence (le nombre d'occurrences)
14.         for (String arg : args){
15.                 Integer value = hashmap.get(arg);
16.                 hashmap.put(arg, (value == null) ? 1 : value + 1);
17. };
```

La Collection List, ainsi que Map ont pour particularité de disposer d'un format par défaut pour l'impression avec System.out.print


```
1.  Voici l'appel :
2.   java Main ceci cela ceci cela apres quoi bonjour bonjour bonjour
    // après exécution du code précédent, voici l'impression :
3.  System.out.println("HASPMAP SIZE: "
4.                  +hashmap.size( )
5.                  + " distinct words:");
6.      System.out.println(hashmap);
7.  // Voici le résultat de l'impression :
8.  HASPMAP SIZE: 5 distinct words:
9.  {ceci=2, cela=2, apres=1, quoi=1, bonjour=3}
```

Implémentation/Réalisation de l'interface Map avec un TreeMap

```
1.  // L'accès à un tree est en O(log(N)) et pour une liste : O(N)
2.  Map<String, Integer>
3.      treemap = new TreeMap<String, Integer>();
4.  // On initialise la fréquence de chaque mot rencontré à zéro
5.          for (String arg : args) treemap.put(arg,0);
6.  // On parcourt la liste de nouveau pour calcul des fréquences
7.          for (String arg : args){
8.              Integer value = treemap.get(arg);
9.              treemap.put(arg, (value == null) ? 1 : value + 1);
10. };
```

Impression d'un TreeMap après calcul des fréquences des mots

```
1.  // appel : C'est le même appel que précédent, mais l'accès à l'arbre
2.  //   est plus efficace : on remarque l'ordre alphabétique des mots
3.  java Main ceci cela ceci cela apres quoi bonjour bonjour bonjour
4.  System.out.println("TREE MAP SIZE: "
5.          +treemap.size()
6.          + " distinct words:");
7.        System.out.println(treemap);
8.  // Affichage : les fréquences sont les mêmes mais l'ordre diffère
9.  TREE MAP SIZE: 5 distinct words:
10. {apres=1, bonjour=3, ceci=2, cela=2, quoi=1}
```

Réalisation/Implémentation d'une Map à l'aide d'une Table de Hashage, plus efficace que l'ordre alphabétique !

```
1.  // L'accès à une table de Hashage se fait en temps constant (meilleur que du
    O(log(N))
2.  Map<String, Integer>
3.      linkedhashmap = new LinkedHashMap<String, Integer>( );
4.  // On rajoute des pairs et on initialise la fréquence à zéro pour
5.  // chaque mot
6.      for (String arg : args) linkedhashmap.put(arg,0);
7.  // Puis on calcule pour chaque mot sa fréquence :
8.      for (String arg : args){
9.          Integer value = linkedhashmap.get(arg);
10.         linkedhashmap.put(arg, (value == null) ? 1 : value + 1);
11. };
```

Impression d'une LinkedHashMap

1. Appel utilisé :
2. java Main ceci cela ceci cela apres quoi bonjour bonjour bonjour
3. System.out.println("LINKED HASH MAP SIZE: "
4. +linkedhashmap.size()
5. + " distinct words:");
6. System.out.println(linkedhashmap);
7. // Résultat de l'impression (on remarque le même ordre que
8. // pour la LinkedList
9. LINKED HASH MAP SIZE: 5 distinct words:
10. {ceci=2, cela=2, apres=1, quoi=1, bonjour=3}

Perspectives

- Les Collections sont un des packages les plus à même de fournir une traduction/compilation des mathématiques ensemblistes vers les langages de programmation : c'est aussi, avec l'aide des Collection qu'il faut tenter de comprendre à quoi correspondent les mécanismes de l'héritage en général mais surtout des modificateurs de ce mécanisme notamment induits par les mots-clefs : private,package,protected,etc.

Conclusion

- Les Collections couvrent plusieurs classes qui permettent de faire le lien directement avec plusieurs concepts bien assimilés en mathématiques comme par exemple les ensembles, les applications, les tableaux et les matrices.
- L'utilisation des Collections en JAVA permettent aussi de factoriser de nombreuses fonctions utilitaires comme par exemple le tri ou la recherche du minimum/maximum dans un ensemble : la maîtrise de ce package est donc incontournable pour une bonne modélisation des problèmes courants et quotidien.

Conclusion

- Les Collections couvrent plusieurs classes qui permettent de faire le lien directement avec plusieurs concepts bien assimilés en mathématiques comme par exemple les ensembles, les applications, les tableaux et les matrices.
- L'utilisation des Collections en JAVA permettent aussi de factoriser de nombreuses fonctions utilitaires comme par exemple le tri ou la recherche du minimum/maximum dans un ensemble : la maîtrise de ce package est donc incontournable pour une bonne modélisation des problèmes courants et quotidien.

Chapitre VII : LA PROGRAMMATION FONCTIONNELLE EN JAVA

Université Mohammed V
Faculté des Sciences
Rabat

LES LAMBDA-EXPRESSIONS

Younès EL AMRANI

Introduction

- La programmation fonctionnelle en JAVA a fait initialement son apparition en JAVA en 2015. Elle apporte avec elle d'immenses possibilités en termes de flexibilité du langage et de modélisation des programmes. Cependant, il faut noter que JAVA est né dedans : c'est à l'origine une implémentation de OAK Lisp, bien qu'ayant relégué son patrimoine génétique aux oubliettes ; force est de constater qu'après 20 ans (de 1995 à 2015) les traits caractéristiques de la programmation fonctionnelle sont revenues tracer sur la face de JAVA les stigmates d'un héritage codé au plus profond de sa machine virtuelle.

Définition de la programmation fonctionnelle

- La programmation fonctionnelle dénote les langages capables de traiter les fonctions comme des citoyennes de première zone.
- La définition d'un élément du langage comme citoyen de première zone signifie qu'il peut se trouver et se retrouver dans n'importe qu'elle structure du langage :
  - Dans une affectation : aussi bien à gauche qu'à droite
  - Dans une liste de paramêtres
  - Dans un retour de fonction : comme valeur de retour

Comment faire de la programmation fonctionnelle

Université Mohammed V
Faculté des Sciences
Rabat

Pour faire de la programmation fonctionnelle, il faut être capable d'utiliser les fonctions comme des paramêtres, ou des valeurs de retour d'une autre fonction : des langages insoupconnés sont fonctionnels : cas de C !

// Illustration en C d'une fonction en right-hand side :

```
typedef int (*Tii)(int) ;
int f(int x){…}
int g(int x){ Tii F=f ;  F(10) ;} // f est en rhs
```

// Illustration en C d'une en paramêtres

```
int h(Tii F, int n){return F(n) ;}    // pour appel dans le main on écrit h(f,10)
int main( ) { h(f,10) ; }             // h est ici une fonction d'ordre 2 ! (en C!)
```

Quels types de fonctions en JAVA ?

- Depuis 1995, nous avons disposé en JAVA UNIQUEMENT des méthodes dans les classes pour réaliser des opérations.
- Les méthodes ne peuvent pas voir leur valeur (en tant que fonction) modifiée.
- Une méthode est définie par une séquence d'opérations qui ne peut être changée.
- Si on considère la classe comme une produit cartésien de valeurs scalaires et fonctionnelles, la méthode est une valeur fonctionnelle constante !

Que nous apportent les lambdas de java ?

Les lambdas en JAVA SONT DES FONCTIONS inspirées des langages fonctionnelles et introduisent au sein de JAVA, plusieurs possibilités :

- Affecter une lambda en partie droite de l'affectation
- Passer en paramêtre une lambda
- Retourner une lambda en valeur d'une autre fonction/méthode/lambda-expression
- Modifier la valeur d'une lambda après sa définition
- Passer en paramêtre une lambda à une autre lambdas : ce sont des fonctions d'ordre supérieur !

Pour définir une lambda expression, on utilise une interface fonctionnelle

- Les interfaces fonctionnelles sont des interfaces de JAVA qui donne la signature, prototype de la lambda expression : UNE ET UNE SEULE signature doit être donnée dans une interface fonctionnelle, exemple :

```
interface iddrd {double call(double x, double y) ;}
```

Pour définir une lambda expression, on peut utiliser une interface fonctionnelle

interface iddrd {double call(double x, double y) ;}

Avec l'interface fonctionnelle ci-dessus on peut définir des lambdas comme suit :

Iddrd f1=(x1,x2)-> x1+x2 ;

Iddrd f2=(x1,x2)-> Math.pow(x1,x2) ;

Utilisation d'une lambda en paramètre

On commence par déclarer une interface fonctionnelle :

interface Lambda_DrD{ double call(double d);}

Voici alors comment passer une lambda en paramêtre:

double F1(Lambda_DrD f, double x1, double x2)

{ return(f.call(x1)+f.call(x2)); }

Dans un second exemple, ci-dessous, on illustre à la fois le passage dans la liste des paramêtres, mais aussi comme valeur de retour de la méthode nommée setf3 :

Lambda_DrD setf3(Lambda_DrD f3)

{ return this.f3=f3; }

Utilisation d'une lambda en valeur de retour d'une fonction

On utiliser l'interface fonctionnelle suivante:

interface Lambda_DrD{ double call(double d);}

on illustre ci-dessous à la fois le passage dans la liste des paramêtres, mais aussi comme valeur de retour de la méthode nommée setf3 :

Lambda_DrD setf3(Lambda_DrD f3)

{ return this.f3=f3; }

Les lambda-expressions pour le style fonctionnel

- Les lambda-expressions permettent au programmeur un nouveau style de programmation, directement inspiré des langages fonctionnels, plus souple, plus intuitif
- Les lambdas expressions permettent d'obtenir des classes où les méthodes elles-mêmes peuvent céder la place aux lambda-expressions et devenir modifiables comme tout autre attribut

La lambda remplacerait la méthode et serait plus flexible

- Les lambdas expressions permettent d'obtenir des classes où les méthodes elles-mêmes peuvent céder la place aux lambdas
- Les lambdas sont modifiables comme tout autre attribut, pourraient être passer comme paramêtre et retournées comme résultat : sont-elles de super-méthodes ?
- Vers une autre manière de penser la méthode dans la classe : une manière plus souple et plus ouverte aux modifications, aux changements !

Initialisation des lambdas dans un constructeur

- Les lambdas expressions peuvent être initialisées dans le constructeur de la classe comme n'importe quelle autre valeur d'attribut :

```
1. interface Lambda_DrD{     double call(double d); }
2. class C1{
3.   Lambda_DrD f1,f2,f3;
4.   public C1 (  ){
5.            f1=x1->x1+1.0;
6.            f2=x1->2.0*x1+3.0;
7.            f3=x1->Math.pow(x1,2.0)+1.0;
8.   }
9. Lambda_DrD setf1(Lambda_DrD f1)
10. {          return this.f1=f1;     }
```

Comment modifier une lambda

1. *interface Lambda_DrD{ double call(double d);}*
2. class C1{
3. Lambda_DrD f1 ; // Définition d'une lambda
4. public C1( ) { f1=x1→x1+1.0;} // constructeur

Illustration de Modifications dans le main :

1. public static void main(String[ ] args){
2. C1 v1=new C1();
3. v1.setf1(x→x+100.0);***// ici on modifie f1***
4. d1=v1.f1.call(2.0) ;

Lambdas sans paramètres

```
class C1{
    double a=2,b=3,c=5;
    Lambda_voidrD g1,g2,g3;
    public C1(){
        g1=()->Math.pow(a,2.0);
        g2=()->Math.pow(b,2.0);
        g3=()->Math.pow(c,2.0);
    }
```

Tableaux de Lambdas

Il est possible de définir des tableaux de Lambdas, exemple de définition lors de la déclaration :

```
Lambda_DrD[ ] tablam=
    {x1->x1+1,
     x1->x1+10,
     x1->x1+100,
     x1->Math.pow(x1,2.0),
     x1->Math.pow(x1,2.0)+x1+1};
```

Utiliser un tableau de Lambda sur une valeur

```
System.out.print("provide xx=");
double xx=scan.nextDouble();
// Ttableau de double où stocker le résultat
Double[ ] res=new double[tablam.length];
// Appel des lambdas et stockage du résultat
for(int i=0;i<tablam.length;i++){
    res[i]=tablam[i].call(xx);
}
```

Affichage du résultat de l'application d'un tableau de lambdas

On utilise la classe java.util.Arrays

Qui contient plusieurs fonctions prédéfinies pour

Afficher, convertir en chaînes de caractère, trier, etc.

System.out.print(java.util.Arrays.toString(res));

Fonctions d'ordre 2

Les fonctions d'ordre 2 sont celles qui prennent en paramètre une autre Lambda : en particulier pour l'appliquer sur une valeur ou sur un ensemble de valeurs.

Définition d'une Lambda d'ordre 2

```
double F1(Lambda_DrD f, double x1, double x2)
{
    return(f.call(x1)+f.call(x2));
}
```

La lambda f est appliquée à x1 et x2 : elle est elle-même un paramêtre, donc F1 est d'ordre 2

Utilisation d'une fonction d'ordre 2

```
////////
// - Test de F1, fonction de second ordre
System.out.println("\n// - Test de fonction de second ordre\n");
////

double d0=v1.F1(x1->x1+2,3.0,2.0);

System.out.println("v1.F1(x1->x1+2,3.0,2.0) == "+d0);
```

Utilisation d'une fonction d'ordre 2 : changement du paramètre

```
double d0=v1.F1(x1->x1+2,3.0,2.0);
System.out.println("v1.F1(x1->x1+2,3.0,2.0) == "+d0);
    double d01=v1.F1(x1->Math.pow(x1,2.0),2.0, 3.0);
    System.out.println("F1(x1->Math.pow(x1,2.0),2.0, 3.0) == "+d01)
```

Modification de la valeur d'une Lambda par affectation

```
Lambda_DrD setf1(Lambda_DrD f1){
        return this.f1=f1;
  }
```

```
v1.setf1(x->x+10.0);
d1=v1.f1.call(2.0);
System.out.println("(x->x+10.0) (2.0) == "+d1);
```

Interfaces Fonctionnelles versus les autres Interfaces de JAVA

1. Pour utiliser les lambdas avec la déclaration dans une interface alors il faut se restreindre à une seule déclaration par interface car seules les interfaces avec une seule déclaration fonctionnent: si on met deux déclarations dans une interface, les lambdas ne fonctionneront plus avec cette interface.

2. On peut considérer qu'une interface avec une seule fonction constitue la défintion d'un ensemble de fonctions et on ne peut pas définir deux ensembles de fonctions différents dans la même déclaration d'interface: d'un point de vue mathématique une interface fonctionnelle définit le domaine et l'image d'un ensemble de fonctions.

3. Si on souhaite définir N ensembles de fonctions, il faudra utiliser N définition d'interfaces fonctionnelles

Définition d'Interfaces Fonctionnelles de |R⁴→ |R

```
interface DomainDoubleRangeDouble{    double f(double x);}

interface DomainDoubleDoubleRangeDouble

{    double f(double x, double y);}

interface DomainDoubleDoubleDoubleRangeDouble

{    double f(double x, double y);}

interface dddrd extends DomainDoubleDoubleDoubleRangeDouble{}//
Renommage

interface DomainDoubleDoubleDoubleDoubleRangeDouble

{        double f(double x, double y);}

interface ddddrd extends
DomainDoubleDoubleDoubleDoubleRangeDouble{}//Rename
```

Renommage d'une Interface Fonctionnelle par héritage

```
interface DomainDoubleDoubleDoubleRangeDouble
{    double f(double x, double y);}

// Renommage de l'interface fonctionnelle par voie
// d'héritage (s'inspirer du typedef de C)
interface ddddrd extends
DomainDoubleDoubleDoubleDoubleRangeDouble
{ /*Aucune extension proposée*/      }
```

Utilisation de tableaux de Lambdas

- La lambda, citoyenne de première zone, apparaissent aussi dans des tableaux !

```
1. interface Lambda_printText {String say();}
2. class Al{
3.    Lambda_printText[ ]    tab6=
4.    {()->"text1",()->"text2",()->"text3",()->"text4"};
5.   public void test() {
6.          for(int i=0;i<4;i++)
7.          System.out.println(tab6[i].say()); }
```

implémentation de contraintes avec des lambda-expressions

- Objectif: Définir plusieurs interfaces fonctionnelles pour définir des contraintes
- Double -> Boolean,
- Double²->Boolean,
- Double⁴->Boolean,
- DoubleX...->Boolean

Exemple d'Interface fonctionnelle Pour Implémenter des contraintes

- interface IDrB{Boolean call(Double x1);}
- interface IDDrB
- {Boolean call(Double x1, Double x2);}
- interface IDDDrB
- {Boolean call(Double x1, Double x2, Double x3);}
- interface IDDDDrB
- {Boolean call(Double x1, Double x2, Double x3, Double x4);}
- interface IDxNrB{Boolean call(Double[ ] x);} // sur $|R^{100}$

Contraintes sur un espace de Dimension N

- Pour un espace de dimension N, on utilisera un tableau de Double
-
-
- interface IDNrB{ Boolean call(Double[] X); }
-

Le type de Lambda «apply » qui appliquent un paramètre aux autres

```
////////
//  Problème : On souhaite Définir une lambda d'ordre 2
//     qui prend en paramètre une contrainte
//     puis l'applique à son second argument :
//   Réponse :   ce sont des Fonctions d'ordre 2 qui
//     prennent leur valeurs dans Boolean
////


interface ApplyIDrB { Boolean call(IDrB  f, Double x1);}

interface ApplyIDDrB{ Boolean call(IDDrB f, Double x1, Double x2);}
```

Définition d'un ensemble de contraintes sur un point

```
////////
// Question : définir uneinterface fonctionnelle pour
// une lambda qui applique N contraintes sur un point
// dans |R¹ :
////


interface ISysConstDrB{
    Boolean call(IDrB[  ] sysc, double x1);
}
```

Lambda qui applique N contraintes sur |R³

- ////////
- // Question : définir une interface fonctionnelle d'une lambda
- // qui applique N contraintes sur un espace de 3 dimensions
- ////
- interface ISysConstDDDrB{
- Boolean call(IDDDrB[ ] sysc,
- double x1, double x2, double x3);
- }
-

Définition de Map : qui applique une contrainte à un sous-ensemble de|R

- ////////////
- // Question: définir La fonction Map qui applique une fonction
- // sur un ensemble ordonné de valeurs et retourne comme résultat
- // un ensemble ordonné de résultats selon le même ordre!
- ////
- .
- ////////
- // Pour un sous-ensemble de N valeurs réelles, on utilise un tableau.
- interface MapIDrB{Boolean[ ] call(IDrB f, Double[ ] x);}
- ////
- .
- // le Map va appliquer la même contrainte IdrB sur un ensemble x ordonnée de points,
- // puis le Map va retourner un ensemble ordonné de Booléens qui sont le résultat
- // de la contraintes, sur chacun des points en entrée

Application d'un ensemble d'une contrainte à 100 points de |R³

- ////////
- // Question : définir un Map Pour un ensemble de Points
- // sur un espace de dimension N
- // Réponse : utiliser un tableau à deux dimensions:
- // la première dimension sera utilisée pour déterminer le nombre de point
- // La seconde dimension du tableau sera utilisée pour les coordonnées
- // Exemple avec |R³ :
-
- interface MapIDDrB{Boolean[ ] call(IDDrB f, Double[ ][ ] x);}
- On aura à l'utilisation Double[ ][ ] x=new Double[100][3]
- pour représenter 100 points de |R³
-

Exemples de contraintes

- ////////
- // Définitions de contraintes sur IR, IR², IR³, IR⁴
- ////
- IDrB c1=x1 -> x1>4.0;
- IDDrB c2=(x1,x2) -> (3.0*x1+5.0*x2)>11;
- IDDDrB c3=(x1,x2,x3) -> (5.0*x1+7.0*x2+11.0*x3)>100.0;
- IDDDDrB c4=(x1,x2,x3,x4)-> (5.0*x1+7.0*x2+11.0*x3-100.0*x4)<1000.0;
-

Définition de 2 lambdas d'ordre 2

- ////////
- // Définition de deux interfaces fonctionnelles
- // d'application d'une contrainte d'ordre 2
- ////
- ApplyIDrB apply=(c,x1)->c.call(x1);
- ApplyIDDrB apply2=(c,x1,x2)->c.call(x1,x2);
-

Définition de la fonction Map |P(|R)

```
////////
// Utilisation de la fonction Map
// sur un ensemble ordonné de valeurs scalaires
////
MapIDrB map=(c,x)->{
 Boolean[] r= new Boolean[x.length];
 for(int i=0;i<x.length;i++){
     r[i]=c.call(x[i]);
 }
 return r;
};
```

Définition de Map sur |P(|R²)

```
////////
// Utilisation de la fonction Map sur une ensemble ordonné de Point dans IRxIR
////
MapIDDrB map2=(c,x)->{
Boolean[] r=new Boolean[x.length];
for(int i=0;i<x.length;i++){
  // Le second indice est utilisé pour l'oordonnée
  r[i]=c.call(x[i][0],x[i][1]);
} // end for
return r;
};  // end MapIDDrB
```

Définition d’un sous-ensemble de |R ordonné

- ////////
- // Initialisation d'un tableau de valeurs rééelles
- // à la ligne de la déclaration d'un tableau de |R
- ////
- Double[ ] tabxx={2.0,3.0,4.0,5.0};
-

Définition d'un élément de |P(|R²)

- ////////
- // Voici l'initialisation d'un tableau de dimension [5][2] lors de la déclaration
- // Nous avons dans ce cas 5 points de |R²
- // p1=(11,12) ; p2=(21,22) ; p3=(31,32) ; p4=(41,42) ; p5=(51,52)
- ////
- Double[ ][ ] **tabyy**={{11.0,12.0},{21.0,22.0},{31.0,32.0},{41.0,42.0},
- {51.0,52.0}
- };
-

Initialisation d'un élément de $|R^3$

- ////////
- // Initialisation d'un tableau de dimension [5][3]
- // La seconde dimension du tableau est utilisée pour les coordonnées (x1,x2,x3)
- // La première dimension du tableau est utilisée pour ordonner les points
- // et déterminer leur nombre: p1, p2, p3, p4, p5 (pour un ensemble de 5 Points)
- ////
- Double[ ][ ] **tabzz**={{11.0,12.0,13.0}, {21.0,22.0,13.0}, {31.0,32.0,13.0},
- {41.0,42.0,13.0}, {51.0,52.0,13.0}
- };
-

Exemple d'une classe avec plusieurs contraintes

- class C5{
- ////////
- // Définitions de contraintes sur IR, IR^2, IR^3, IR^4
- ////
- IDrB c1=x1 -> x1>4.0;
- IDDrB c2=(x1,x2) -> (3.0*x1+5.0*x2)>11;
- IDDDrB c3=(x1,x2,x3) -> (5.0*x1+7.0*x2+11.0*x3)>100.0;
- IDDDDrB c4=(x1,x2,x3,x4)-> (5.0*x1+7.0*x2+11.0*x3-100.0*x4)<1000.0;
- ////////
- // Définition de deux interfaces fonctionnelles d'application d'une contrainte d'ordre 2
- ////
- ApplyIDrB apply=(c,x1)->c.call(x1);
- ApplyIDDrB apply2=(c,x1,x2)->c.call(x1,x2);
-

Exemple d'utilisation de apply

- C5 v5=new C5();
- Scanner scan=new Scanner(System.in);
- System.out.print("Donner l'argument=");
- Double xx=scan.nextDouble();
- Boolean b1=v5.apply.call(**x->x+1>2**, xx);
- // Affichage du résultat qui rappelle l'instruction
- System.out.println("(x->x+1>2)("+xx+")="+b1);
-
-

Modification du paramètre de apply

-
- Boolean b2=v5.apply.call(
- **x->Math.sqrt(x)*Math.pow(x,2.5)+3.0>178.0**,
- xx);
- // Affichage du résultat qui affiche l’instruction
- System.out.println("(x->Math.sqrt(x)*Math.pow(x,2.5)+3.0>10)("+xx+")="
- +b2);
-

Exemple d'utilisation de Map

```
//////////
// Fonction d'ordre #2
////
System.out.println("-EXEMPLE AVEC FONCTION D'ORDRE 2-");
Double[ ] tabxx={2.0,3.0,4.0,5.0};
Boolean[  ] tabb1=v5.map.call(x->x%2==0, tabxx);
// Affichage avec rappel de l'instruction
System.out.println("(x->x%2==0)("
        +java.util.Arrays.toString(tabxx)+")="
        +java.util.Arrays.toString(tabb1));
```

Prérequis : COURS THREADS Utilisation Runnable avec Lambdas

- // Question : comment utiliser une Lambda
- // pour initialiser un Runnable ?
- // Réponse : Initialiser un Runnable avec une Lambda
- // comme suit :
- Runnable r1=()->{for(int i=0;i<N;i++) System.out.println("Salam");};
- // Lancement du Runnable avec un bloc try-catch
- Thread t1 = new Thread(r1) ;
- t1.start( ) ;

Prérequis : COURS COMPARATOR
Comparator avec Lambda selon X

- import java.util.Comparator;
- import java.util.Arrays;
- // Note : Comparable est déjà dans java.lang
- **class Point {double X, Y, Z;}**
- **Comparator<Point>** selonX = // On donne une lambda comme valeur
- (p1,p2) -> // p1 et p2 sont deux Points (X,Y,Z)
- {double **x1**=p1.X, **x2**=p2.X;
- if(x1>x2) return 1;
- else {if (x1==x2) return 0;
- else return -1;
- }
- };

Prérequis : COURS COMPARATOR
Comparator avec Lambda selon Y

- import java.util.Comparator;
- import java.util.Arrays;
- // Note : Comparable est déjà dans java.lang
- **class Point {double X, Y, Z;}**
- **Comparator<Point>** selonY = // On donne une lambda comme valeur
- (p1,p2) -> // p1 et p2 sont deux Point
- {double y**1**=p1.Y, y**2**=p2.Y;
- if(y1>y2) return 1;
- else {if (y1==y2) return 0;
- else return -1;
- }
- };
-

Prérequis : COURS COMPARATOR
Comparator avec Lambda selon Z

- import java.util.Comparator;
- import java.util.Arrays;
- // Note : Comparable est déjà dans java.lang
- **class Point {double X, Y, Z;}**
- **Comparator<Point>** selonZ = // On donne une lambda comme valeur
- (p1,p2) -> // p1 et p2 sont deux Points (X,Y,Z)
- {double z**1**=p1.Z, z**2**=p2.Z;
- if(z1>z2) return 1;
- else {if (z1==z2) return 0;
- else return -1;
- } };

Prérequis : COURS COMPARATOR Utilisation du Comparator selonY

- // Tri d'un tableau tabPoint avec le
- // Comparator selonY défini comme une Lamda
- java.util.Arrays.sort(tabPoint, selonY);
- for(int i=0;i<tabPoint.length;i++) System.out.println(tabPoint[i]);

Prérequis : COURS COMPARATOR Utilisation du Comparator selonX

- // Tri d'un tableau tabPoint avec le
- // Comparator selonY défini comme une Lamda
- java.util.Arrays.sort(tabPoint, selonX);
- for(int i=0;i<tabPoint.length;i++) System.out.println(tabPoint[i]);

Prérequis : COURS COMPARATOR
Utilisation d'un Comparator dans l'instruction d'appel

- //Maintenant, il est possible de définir le Comparator, dans l'appel même de Arrays.sort, sous forme de lambda
- java.util.Arrays.sort(tabPoint, (p1,p2)-> {double z1=p1.Z,z2=p2.Z;if(z1>z2) return 1; else {if(z1==z2) return 0; else return -1;}});
- for(int i=0;i<tabPoint.length;i++) System.out.println(tabPoint[i]);

Conclusion

1. L'introduction des lambda-expressions en JAVA et le renforcement de la programmation fonctionnelle ouvre de multiples possibilités comme nous l'avons vu : mais ce n'est pas tout, c'est le style de programmation entier qui est invité à revisiter son architecture et à promouvoir la fonction au premier rang et à l'introduire dans les Collections pour une utilisation nouvelle, flexible, imaginative et dynamique à la recherche d'algorithmes encore à découvrir.

Perspectives

1. La programmation fonctionnelle en JAVA : sa découverte permet de se convaincre que les paradigmes de la programmation son perméables et évolutifs : les langages, plus que jamais, semblent capables de se convertir, s'étendre, se diversifier en termes de constructions, locutions, mots-clefs, mécanismes d'héritage de prototypage et d'autres encore inconnus. Finalement, le langage doit permettre au programmeur de s'exprimer comme il le pense sans que jamais la syntaxe ou les constructions de base ne deviennent des obstacles entre l'algorithme imaginé par le programmeur et la machine sur lequel il va prendre vie.

Chapitre VIII : Le package swing et les interfaces graphique

La programmation d'interfaces graphiques en JAVA

Introduction à SWING

- Le package swing et ses classes permettent l'interaction graphique
- L'interaction graphique se compose de fenêtres, boutons, menu, barre de défilement, etc.
- Le programmeur a un large choix de classe pour implémenter son interaction graphique.
- Toute interaction est fondée sur deux directions fondamentales : les entrées de l'utilisateur vers le programme, et les sorties du programme vers l'écran.
- Un excellent exemple de classe qui permet ces entrées et sorties est la classe JOptionPane

Introduction

- La programmation des interfaces graphiques est certainement un des types de programmation les plus valorisantes pour le programmeur : en effet, elle permet de donner une bien meilleure visibilité aux fonctions d'un programmeur, une vision globale de l'ensemble de ses fonctionnalités sous une forme agréable, interactive et intuitive.

Fondements des interfaces graphiques

- Les interfaces graphiques sont fondamentalement articulées autour de la notion d'événement et de réactions par rapport à un événement donné.
- Comme elles s'articulent autour de concepts bien connu dans le quotidien qui sont : les pressoirs-boutons, les menus et menus déroulants, les ascenseurs, les fenêtres, les cadres : autant d'objet empruntés à notre environnement de travail vont se retrouver sur l'écran pour nous faciliter l'interaction avec les programmes !
- L'utilisation des interfaces graphiques permettent un accès beaucoup plus intuitif et facile aux utilisateurs des programmes qui sont développés par les spécialistes.

Les classes parmi les plus pratiques de swing

- Les classes misent à la disposition des programmeurs sont très nombreuses, parmi les plus pratiques on peut citer les classes suivantes, remarquez les noms self-documentés :

1. Container c1;
2. JPanel panel1;
3. JTextArea text1;
4. JMenuBar menubar1;
5. JMenu jmenuadd, jmenufile,jmenusayhello;
6. JmenuItem jmenuitemreadfile, jmenuitemaddtext, jmenusayhelloselectlanguage;

Les éléments de base de l'interaction graphique

- L'interaction graphique est fondée essentiellement sur la génération d'évènements et sur la réponse à ces évènements.
- Les objets interactifs à l'écran sont devenus familiers des interfaces graphiques : essentiellement des menus et des boutons !
- La qualité de l'interface, se mesure au degré de son utilisation intuitive par un utilisateur novice

L’initialisation d’un FRAME : première étape d’une GUI

- GUI == Graphical User Interface
- La première étape est constituée de l’initialisation d’un JFrame

Conclusion

- La programmation des interfaces graphiques en SWING se fait à l'aide des menus, bouton, événements, Frame/cadre-d'application et autres classes de base en SWING. Cependant, il existe des logiciels qui permettent la génération automatique de ce code, sans pour autant que le programmeur aille dans le détail des implémentations de ces divers composants : cependant, cette connaissance lui permettra d'affiner la représentation de son interface et de mieux en assujettir le code et les détails à sa propre convenance.

Perspectives

- La programmation interactive à beaucoup évolué, les écrans tactiles et la reconnaissance vocale et gestuelle sont aujourd'hui déjà au rendez-vous, il convient d'approfondir ses connaissances au-delà des outils fondamentaux pour enrichir l'interaction avec les programmeurs de nombreuses fonctionnalités intuitives et plus proches de l'interaction homme-homme que de l'interaction homme-machine !

Université Mohammed V
Faculté des Sciences
Rabat

Chapitre 9 : l'ajout de fonctions utilisation de types externes à JAVA

La Foreign Function Interface : FFI
La JAVA Native Interface : JNI

Introduction

- La JNI c'est une ouverture de JAVA vers la programmation mixée, elle permet à JAVA de s'ouvrir à son propre langage d'implémentation : le langage C, mais aussi vers l'ensemble de tous les langages qui possèdent eux-mêmes une interface avec C.
- La JNI permet l'extension de la JVM par le programmeur lui-même, cependant, le prix à payer est élevé car en activant la JNI on perd la portabilité de JAVA.

Pourquoi la JAVA-NATIVE-INTERFACE, la JNI

- L’ouverture de JAVA à d’autres langages de programmation, permet d’augmenter les possibilités de JAVA en matière de gestion des périphériques, par exemple, plus facile en C.
- La JNI rend possible la réutilisation de bibliothèques écrites dans un tiers langage.
- Le mot natif est utilisé ici car JAVA a été à l’origine développé en C.

Les étapes de l'interfaçage d'une fonction C en JAVA en Cinq étapes

1. Déclarer la fonction native en JAVA dans un fichier java : exemple « dec.java »
2. Disposer du code de la fonction native puis le compiler avec un compilateur pour obtenir une dll : dynamic loadable library.
3. Écrire un wrapper en C de la fonction native : l'entête du wrapper s'obtient avec le binaire javah -jni decl.java Il convient de préciser que la déclaration est généré dans le code native, mais le corps du wrapper reste à compléter !
4. Dans la classe ou sera utilisée la fonction native faire System.load("decl.dll") dans un bloc static

Etape 1 : déclaration des fonctions natives dans un fichier JAVA

- /////
- // Fichier appelé GENJNativeDeclOfPrototypes.java
- /////
- class GENJNativeDeclOfPrototypes {
- static { System.loadLibrary ("Prototypes"); }
- public native int factoriel(int n);
- public native int f(int x);
- public native double g(char c); }

Étape 2 : compiler le code natif obtenir une dll

- int f (int n){
- if (n <= 0) return 1 ;
- else return (n * n) ;}
- int factoriel (int n){
- if (n <= 1) return (1) ;
- else return (n * (factoriel (n - 1)));
- }
- double g(char c) {
- double x
- ; … ;
- return (x*5.3) ; }

Étape-2 (suite) compilation vers une DLL

1. CC=gcc
2. CFLAGS=-c -I.
3. JAVA=java
4. JAVAC=javac
5. JAVAH=javah
6. JAVAHFLAG=-jni
7. JAVACFLAGS= -cp .;..
8. JAVAFLAGS= -cp .;..
9. LDLIBRARYPATH=-Djava.library.path=.
10. DLLFLAG=-DBUILD_DLL
11. **lib: Prototypes.o GENWrappersFromPrototypes.o GENWrappersFromMath.o**

    $(CC) $(INCLUDES) -shared -o prototypes.dll Prototypes.o GENWrappersFromMath.o -lgcc

Étape 3 : Écriture du Wrapper en C, une affaire d'expert !

```
1.  JNIEXPORT jint JNICALL _Java_GENJNativeDeclOfPrototypes_factoriel (JNIEnv *env, jobject obj, jint J_arg_1){
2.    jint result;// Declaration used in macros
3.    return CINT2JINT(factoriel(JINT2CINT(J_arg_1)));
4.  }
5.  JNIEXPORT jint JNICALL _Java_GENJNativeDeclOfPrototypes_f (JNIEnv *env, jobject obj, jint J_arg_1){
6.    jint result;// Declaration used in macros
7.    return CINT2JINT(f(JINT2CINT(J_arg_1)));
8.  }
9.  JNIEXPORT jdouble JNICALL _Java_GENJNativeDeclOfPrototypes_g (JNIEnv *env, jobject obj, jchar J_arg_1){
10.   jdouble result;// Declaration used in macros
11.   return CDOUBLE2JDOUBLE(g(JCHAR2CCHAR(J_arg_1)));
12. } // Les prototypes peuvent être générés à l'aide de javah -jni à partir des déclarations JAVA
```

Étape 4 : charger la DLL dans la classe d'utilisation dans un bloc static

- class Test {
- static { System.loadLibrary ("jnilib"); }
- public static void main(String[] args) {
- int[] array_int = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} ;
- GENJNativeDeclOfPrototypes
- **prototypes = new GENJNativeDeclOfPrototypes() ;**
- GENJNativeDeclOfMath
- mat = new GENJNativeDeclOfMath() ;
- System.out.println(// Test des fonctions natives de la dll
- "factoriel(3) = "+ **prototypes.factoriel(3)**
- +"\ncube(5) = "+ **prototypes.cube(5))**;

Intérêt de l’interfaçage aux autres langages : la programmation mixée

- La possibilité de s’interfacer à d’autres langages permet la programmation mixée et l’ouverture à d’autres communautés de programmeurs autre que celle de JAVA
- La programmation mixée permet la réutilisation de programmes, mais aussi elle permet de bénéficier des points forts des autres langages de programmation ; par exemple "C" pour la gestion des périphériques !

Désavantage de la Foreign Function Interface ou la JNI

- Le principale désavantage de la JNI/FFI c’est la perte de la portabilité, en effet, la "dynamic loadable library" est spécifique à la machine cible et donc, la "dll" générée n’est pas portable
- Il y-a une perte de la sécurité garantie à la base par la machine virtuelle : les appels systèmes passent par la Java Virtual Machine qui permet un contrôle de la sécurité : les fonctions natives sont au dehors de ce périmètres de sécurité.

Automatisation de la JNI

- Il convient de préciser ici qu'il est tout à fait possible de compléter l'automatisation de la JNI, pour l'heure, plusieurs choses sont faciles à obtenir :
  - Les déclarations des WRAPPERS grâces à l'utilitaire javah -jni decl.java
  - La génération de la dll se fait aussi grâces au compilateur du code natif
- Cependant, le code du WRAPPER est à la charge du programmeur de la JNI

Automatisation de l'écriture des WRAPPERS

- Les WRAPPERS sont des enveloppes de fonctions permettant le passage d'un type d'un langage L1 vers L2 et vice-versa
- Il est possible d'automatiser ce passage par le biais d'écriture de macros prédéfinies permettant cette traduction.
- En identifiant les types de L1 et leur correspondants dans L2, on peut définir des macros pour une traduction automatisée.

Macros pour générer le code des WRAPPERS pour les nombres

- #define JBOOLEAN2CBOOLEAN(arg) ((unsigned char) (arg))
- #define CBOOLEAN2JBOOLEAN(arg) ((jboolean) (arg))
- #define JCHAR2CCHAR(arg) ((unsigned short) (arg))
- #define CCHAR2JCHAR(arg) ((jchar) (arg))
- #define JSHORT2CSHORT(arg) ((short) (arg))
- #define CSHORT2JSHORT(arg) ((jshort) (arg))
- #define JFLOAT2CFLOAT(arg) ((float) (arg))
- #define CFLOAT2JFLOAT(arg) ((jfloat) (arg))
- #define JDOUBLE2CDOUBLE(arg) ((double) (arg))
- #define CDOUBLE2JDOUBLE(arg) ((jdouble) (arg))
- #define JINT2CINT(arg) ((int) (arg))
- #define CINT2JINT(arg) ((jint) (arg))
- #define JLONG2CLONG(arg) ((long) (arg))
- #define CLONG2JLONG(arg) ((jlong) (arg))
- #define JBYTE2CBYTE(arg) ((signed char) (arg))
- #define CBYTE2JBYTE(arg) ((jbyte) (arg))

Macros pour générer les WRAPPERS pour les chaînes

- #define JSTRING2CSTRING(jstring_arg) ((char *)((*env)->GetStringUTFChars(env, jstring_arg, NULL)))
- #define CSTRING2JSTRING(cstring_arg) ((jstring)((*env)->NewStringUTF(env,(const char *)cstring_arg)))

Utilisation de JAVA à partir de C

- Il est tout aussi intéressant d'avoir accès au code JAVA à partir d'un autre langage de programmation et notamment à partir de C.
- La possibilité d'exécuter des scripts JAVA à partir de C, permet une intégration complémentaire des deux langages et le traitement dans chaque langage des programmes les plus adaptés à chacun d'eux.

La Java Native Interface versus les autres Langages Orientés Objet

- L’extension de la JNI à d’autre langages orientés objet est aussi possible, notamment vers le C++ qui possède lui même une ouverture sur le C.
- L’extension de la JNI pose cependant des questions sur l’intégration des modèles objets : notons qu’en C++ le multi-héritage existe, tandis que le modèle OO de JAVA est mono-héritage, donc, il y-a des limites à l’intégration JAVA-C-C++ qui sont liées à la nature des modèles qui les sous-tendent.

La JNI versus l'extension fonctionnelle avec les Lambdas

- Aujourd'hui JAVA se dirige vers une extension fonctionnelle de lui-même par l'intégration des Lambdas depuis 2015.
- La question de l'utilité de la JNI fasse au développement et à l'évolution du langage lui-même est posée :
  - Est-il plus utile de faire évoluer un langage en lui intégrant de nouveaux paradigmes de programmation ou doit-on augmenter la mixité du langage en renforçant la Java Native Interface et toutes les Foreign Function Interface ?

Conclusion

- La JNI fournit un extraordinaire outil d'ouverture de JAVA vers ses principales conccurents : C et C++, cependant, la perte de la portabilité est un prix trop cher à payer dans beaucoup de cas
- La JNI permet à la fois l'intégration de C à JAVA mais aussi l'intégration de JAVA à C, cette double ouverture demeure cependant très technique et nécessite une connaissance fine de la JVM pour la complétion des WRAPPERS : tant qu'elle n'est pas entièrement automatisée elle reste limitée à une très petite communauté d'experts

Perspectives

- L’inter-opérabilité versus l’évolutivité sont deux aspects d’une même problématique : la recherche d’un langage universel de programmation.
- Si L’inter-opérabilité permet à chaque communauté de rester sur son langage, elle exige cependant une forte dose d’expertise pour être mise en œuvre
- L’évolutivité des langages semble très prometteuse, car en dépit des idées préconçues il semblerait que la programmation fonctionnelle, la programmation impérative et la programmation en logique du premier ordre sont des vues de l’esprit qui peu à peu vont en s’intégrant et en s’unifiant au sein du même langages qui deviennent, jour après jour, multi-paradigmes !

Chapitre 10 : les Threads

chapitre 10 :
Les Threads
ou la programmation parallèle,
Younès EL AMRANI.

Introduction

- La programmation parallèle en JAVA s'effectue à l'aide de la classe Thread mais aussi Runnable et Future.
- Les Systèmes du futur possèdent déjà des micro processeurs avec plusieurs centaines de « core »
- Les « core » appelés cœurs en français sont des sortes de micro processeurs dans le micro-processeur qui sont capables d'effectuer des calculs en parallèle les uns des autres.
- Le langage JAVA procure le package Thread pour tirer partie de ces micro processeurs du futur dès aujourd'hui

Définition d'un Thread

- Les Threads *sont des processus légers*
- *que signifie légers ? « léger » signifie que Les Threads ne nécessitent pas un changement de contexte (avec le rétablissement d'une pile d'exécution : c'est pourquoi on les appelle « processus légers ». légers aussi car les Threads partagent entre eux du code, des données et des ressources. Donc légèreté grâce au partage des données. En particulier, les threads partagent les ressources système : très pratiques pour les Entrées-Sorties.*

Les Threads s’exécutent en parallèle du programme principal main

Les Threads s’exécutent en "parallèle", y compris en parallèle du programme principal main. Autrement dit, le main lui-même s'exécute dans un Thread à part. D'où de meilleures performances au lancement et en exécution, surtout s'il existe plusieurs processeurs disponibles. D'où le gain de temps et de performance.

Pourquoi une modélisation Multi Threads ?

Le monde réel est par nature Multi-Thread, donc, la modélisation utilisant des Threads est plus proche du problème dans le monde réel.

La classe Thread et Runnable sont toutes deux dans le langage de base de JAVA : leur utilisation est particulièrement aisée.

Comment utiliser la classe Thread

Pour utiliser la classe Thread, il suffit d'en hériter et d'implémenter la méthode **void run(){…}**

La méthode **run** contient exactement le code qui va s'exécuter en parallèle.

Pour exécuter le code run en parallèle, il suffira d'appeler la méthode **start( )** sur l'instance d'un **Thread**

```
class T1 extends Thread{
    void run( ){   for(;;) System.out.println("T1") ;  }    }
// Pour lancer le Thread T1, on fait comme suit :
new T1( ).start( ) ;
```

Comment utiliser l'interface Runnable

Pour utiliser la classe Thread, une seconde méthode consiste à passer par l'interface Runnable.

Pour cela il faudra implémenter la méthode **void run(){…}**

La méthode **run** contient exactement le code qui va s'exécuter en parallèle, comme pour une sous classe de Thread.

Pour exécuter le code run en parallèle, il suffira d'appeler la méthode **start( )** sur l'instance d'un **Thread**

```
class R1 implements Runnable {
    void run( ){   for(;;) System.out.println("R1") ;  }    }
// Pour transformer le Runnable R1 en Thread, on fait:
new Thread ( new R1( ) ).start( ) ;
```

Définition de la classe Thread


```
1.  public class java.lang.Thread implements java.lang.Runnable {
2.    public static final int MIN_PRIORITY;
3.    public static final int NORM_PRIORITY;
4.    public static final int MAX_PRIORITY;
5.  public synchronized void start();
6.    public void run();
7.    public final void stop();
8.    public final synchronized void stop(java.lang.Throwable);
9.    public void interrupt();
10.   public static boolean interrupted();
11.   public boolean isInterrupted();
12.   public void destroy();
13.   public final native boolean isAlive();
14.   public final void suspend();
15.   public final void resume();
16.   public final void setPriority(int);
17.   public final int getPriority();
18.   public final synchronized void setName(java.lang.String);
19.   public final java.lang.String getName();
```

Définition de l'interface Runnable

L'interface Runnable est extrêmement simple : elle se compose de la seule déclaration de la méthode **void run( ) ;**

```
public interface java.lang.Runnable {
    public abstract void run();
}
```

Définition de l'interface Runnable

L'interface Runnable est extrêmement simple : elle se compose de la seule déclaration de la méthode **void run( ) ;**

```
public interface java.lang.Runnable {
    public abstract void run();
}
```

Méthode 1 : spécialiser Thread

```
1.  class T1 extends Thread {
2.     public T1() {...} // Le constructeur
3.     public void run() { for(;;) System.out.println("T1") ; }
4.     public static void main(String[ ] args){
5.       T1 p1 = new T1(); // Création du processus p1
6.       p1.start(); // Démarre le processus et exécute p1.run()
7.  }
```

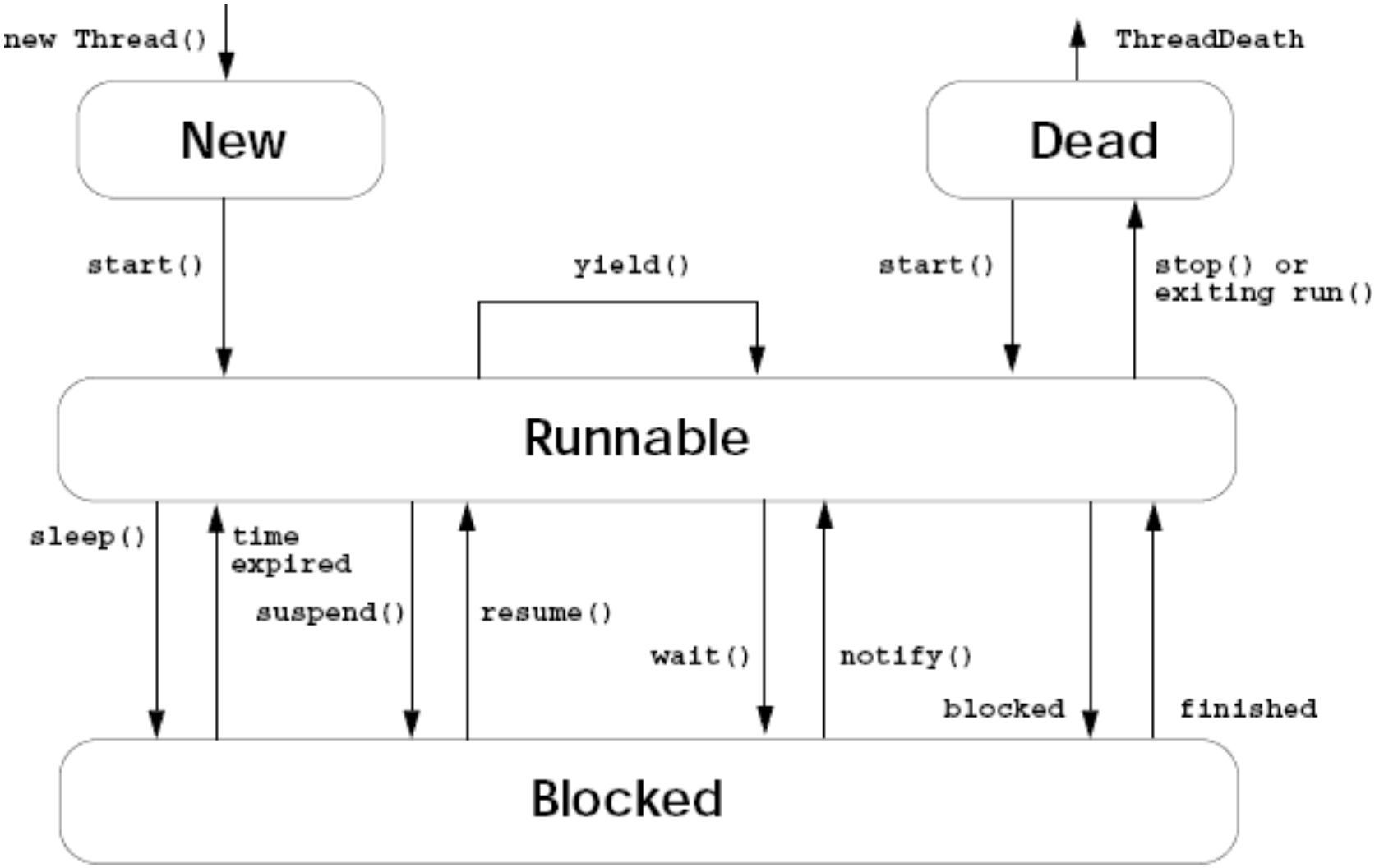
Méthode 2 : implémenter Runnable

```
1.  class R1 implements Runnable {
2.     public R1() {...} // Le constructeur
3.     public void run() { for(;;) System.out.println("R1") ; }
4.     public static void main(String[ ] args){
5.       R1 p1 = new R1();//Création du processus p1
6.       // Construit un Thread puis exécute p1.run()
7.      new Thread(p1).start();
8.  }//IL FAUDRA toujours passer par Thread !
    On a utilisé le constructeur de Thread qui prend en entrée
      un Runnable !
```

Comment Choisir ? Thread ou Runnable

1. Il faudra étendre Thread si on n'hérite pas préalablement d'une autre classe : en effet, en JAVA seul le mono-héritage est disponible, donc Thread s'utilisera si on n'a pas d'autres classes à hériter
2. Il faudra implémenter **Runnable** et le donner en entrée à un constructeur de Thread dans le cas où notre classe est déjà une héritière. C'est par exemple le cas des **Applet** , ou des **Frame** de l'interface graphique : dans les Interfaces Graphiques, très souvent l'héritage est utilisé, donc, dans ce cas, on choisira d'implémenter **Runnable**

Cycle de vie d'un Thread



Les états d'un Thread

(a) Un Thread est d'abord déclaré **Thread t ;**

(b) puis instancié avec new Thread(…)

(c) puis rendu actif avec **.start( )**

(d) Il peut mourir (terminé) avec **.stop( )** ou si le

(e) void run() ; a terminé son exécution complètement.

(f) Il pourra être endormi ou suspendu ou en attente d'un autre Thread avec **.sleep( )** ou **.suspend( )** ou **.wait( )**

(g) puis de nouveau rendu actif avec **.resume( )**

Quand est ce qu'un Thread est inactif ?

- Un Threard est inactif s'il est Endormi ou bloqué :
  1) Endormi après **.sleep(time en ms)**
  2) Suspendu avec un **.suspend( )**
  3) suspend le Thread **resume( )** le réactive
  4) une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread indépendamment du programmeur
- Un Thread devient aussi inactif s'il Meurt :
  1) Un Thread meurt avec **.stop( )** appelé explicitement
  2) Un Thread meurt si il achevé les instructions de son **.run( )**

Exemple avec **.sleep( )** avec InterruptedException

```
1.  class TClass extends Thread {
2.      public static final int N = 3;
3.      public static final int T = 10;
4.      public String msg;
5.      public TClass(String msg){this.msg = msg;}
6.      public void run() {
7.          try {
8.             for(int i=0;i<N;i++) {
9.                 System.out.println("TClass1: " + msg);
10.                java.lang.Thread.sleep(T);
11.            }
12.         }
13.        // L'appel à .sleep( ) est susceptible de voir lever l'exception InterruptedException
14.        catch(java.lang.InterruptedException e){}
15.    }
16. }
```

Priorités des Threads

- Java permet de modifier les priorités desThreads avec la méthode setPriority()
- Par défaut, chaque nouveau Thread a la même priorité que le Thread qui l'a crée
- La JVM choisit d'exécuter le Thread actif qui a la plus haute priorité.
- si plusieurs Threads ont la même priorité, la JVM répartit équitablement le temps CPU (time slicing) entre tous.

Comment modifier les priorités des Threads ?

setPriority(int) : fixe la priorité du receveur ; le paramètre doit être dans l'intervalle :

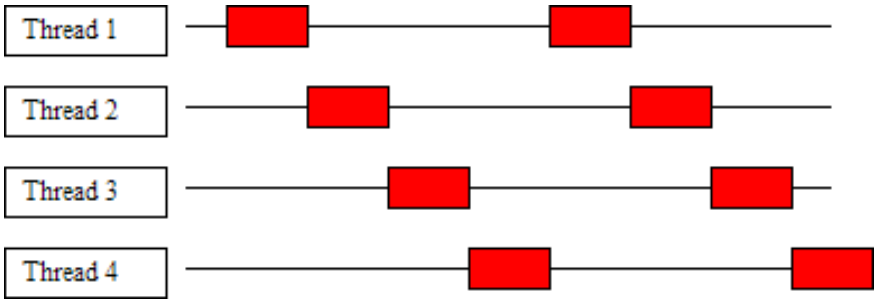
[MIN_PRIORITY, MAX_PRIORITY]

car autrement **IllegalArgumentException** est levée

La méthode **int getPriority( )** permet de connaître la priorité d'un Thread. La valeur final static appelée

Thread.NORM_PRIORITY : donne le niveau de priorité par défaut d'un Thread.

Comment se passe le scheduling en JAVA ?



- Time-slicing (ou round-robin scheduling) :
  - La JVM répartit de manière équitable le CPU entre tous les threads de même priorité. Ils s'exécutent en "parallèle".
- Préemption (ou priority-based scheduling) :
  - Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il peut le céder :
    - involontairement : sur entrée/sortie
    - volontairement : appel à la méthode statique **yield( )**

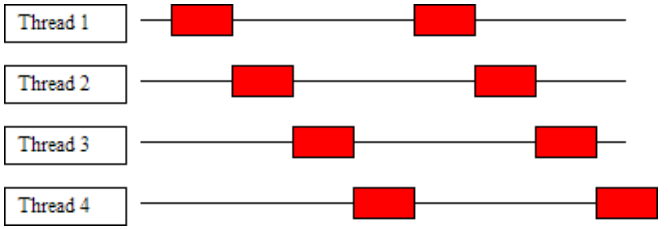
      Attention : ne permet pas à un thread de priorité inférieure de s'exécuter (seulement de priorité égale)
    - implicitement en passant à l'état endormi avec

      wait(),

      sleep() ou

      suspend()

Comment accéder à la mémoire commune : la **synchronisation**



- **Entre Thread, l'espace de travail est commun, il n'y a une "mémoire commune" pour tous les threads d'une même classe**
  - En cas d'accès simultané à une même ressource
    Il faut souvent l'accès exclusif à un objet pendant l'exécution d'une ou plusieurs instructions d'écriture ou de modification pour conserver la cohérence des données : on utilise le mot-clef **synchronized** pour cela !
- Le mot-clé **synchronized** permet de gérer les concurrence d'accès dans les cas suivants :
  - plusieurs appels / accès simultanés à une méthode
  - plusieurs accès simultanés en écriture à un objet
  - plusieurs accès simultanés en écriture à un bloc

Comment fonctionne la synchronisation ?

- **Basée sur la technique de l'exclusion mutuelle :**
  - à chaque objet Java est associé un « verrou » géré par le thread quand une méthode (ou un objet) **synchronized** est accédée.
  - Le verrou garantit l'accès exclusif à une ressource (la section critique) pendant l'exécution d'une portion de code.
- Une section critique peut-être définie dans trois situations :
  - une section critique définie par toute une méthode : déclaration précédée de synchronized
  - une section critique définie par une instruction (ou un bloc) elle sera précédée du mot-clef : **synchronized**
  - une instance, un objet à usage exclusif : dans ce cas le déclarer **synchronized**
- Attention L'UTILISATION DE SYNCHRONIZED PEUT PROVOQUER DES CAS D'INTER-BLOCAGES !

Déclaration de méthodes synchronized

- Pour gérer la concurrence d'accès à une méthode on la déclare **synchronized ;** si un thread exécute cette méthode sur un objet, un autre thread ne peut l'exécuter pour le même objet :

```
public synchronized void m1 ( ) {...}
```

- synchronisation d'un :

```
public void m1() { ...
      synchronized(objet) {objet.m1();}}
```

l'accès à l'objet passé en paramètre de **synchronized(Object)** est réservé à un unique thread.

Des Threads qui sont de véritables démons !

- Un thread peut être déclarer comme daemon :
  - comme le "garbage collector",
  - un Daemon s’arrête quand le programme se termine, sinon il s’exécute sans arrêt !
- Comment rendre un Thread démoniaque ? :-)
  - **setDaemon( )** : transforme un thread en daemon
  - **isDaemon( )** : vérifie si un thread est un daemon ?

Regroupement de plusieurs Thread en Groupe : les ThreadGroup !

Pour contrôler plusieurs threads

- Plusieurs processus (Thread) peuvent s'exécuter en même temps, il serait utile de pouvoir les manipuler comme une seule entité
  - pour les suspendre
  - pour les arrêter, ...

Java offre cette possibilité via l'utilisation des groupes de threads : `java.lang.ThreadGroup`

- on groupe un ensemble nommé de threads
- ils sont contrôlés comme une seule unité

Les groupes de Threads

- Une arborescence :
  - la classe `ThreadGroup` permet de constituer une arborescence de Threads et de ThreadGroups
  - elle donne des méthodes classiques de manipulation récursives d'un ensemble de threads : `suspend(), stop(), resume()`, ...
  - et des méthodes spécifiques : `setMaxPriority()`, ...
- Fonctionnement :
  - la JVM crée au minimum un groupe de threads nommé main
  - par défaut, un thread appartient au même groupe que celui qui l'a crée (son père)
  - `getThreadGroup()` : pour connaître son groupe

Création d'un groupe de threads

- Pour créer un groupe de processus :

```
ThreadGroup p1p2p3 = new ThreadGroup("P1P2P3");
Thread p1 = new Thread(p1p2p3, "P1");
Thread p2 = new Thread(p1p2p3, "P2");
Thread p3 = new Thread(p1p2p3, "P3");
```

- On peut créer des sous-groupes de threads pour la création d'arbres sophistiqués de processus
  - des ThreadGroup contiennent des ThreadGroup
  - des threads peuvent être au même niveau que des ThreadGroup

Les méthodes sur les groupes de Threads : les mêmes que pour un !

- `resume(), suspend(), stop(), s'appliquent aussi sur un groupe de Threads :`
  - Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode

Les méthodes sur les groupes de Threads : les mêmes que pour un !

- `resume(), suspend(), stop(), s'appliquent aussi sur un groupe de Threads :`
  - Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode

Conclusion

Les Threads permettent la programmation parallèle.

Les Threads permettent d'améliorer les performances en optimisant l'utilisation des ressources.

Les Threads introduisent un type de programmation très difficile à corriger en cas d'erreur car la séquence des appels introduit un nombre factoriel / exponentiel de séquence d'exécution possible

Perspectives

De nouveaux paquetages et classes permettent de renvoyer une valeur pour les exécutions parallèles.

La classe Future introduit de nouvelles possibilités à explorer dans une extension de ce cours.