Les nouveautés de java 5, 6 ...

contenu de la section

La nouvelle boucle for

Quelques généralités

- construction introduite en java5
- simplifier l'écriture des boucles ➔ offrir une instruction "foreach"
  - un même traitement appliqué à TOUS LES ELEMENTS
    impossibilité de distinguer des itérations particulières

```
package java.lang;
import java.util.Iterator;

public interface Iterable<E> {
    public Iterator<E> iterator() ;
}
```

```
package java.util;

public interface Iterator<E> {
    public boolean hasNext();
    public E next() ;
}
```

l'aspect syntaxique

- prise en compte directe des tableaux et des classes implémentant l'interface Iterable
- la syntaxe d'utilisation est : **for (ELEMTYPE var : expression) { ..... }**

```
int[ ] table = new int[10] ;
.......
for (int elem :  table ) { System.out.println(elem); }
List list = new ArrayList() ;
list.add(....);
list.add(....);
…….
for (Object elem : list ) {
   System.out.println("l'element est  " + elem);
}
```

table est un tableau

list est un **Iterable**

remarques

- impossibilité d'itérer en sens inverse
- impossibilité d'exprimer l'accès simultané à plusieurs tableaux ou plusieurs Iterable

Les types énumérés (1)

quelques généralités

- construction introduite en java5
- un type énuméré = un type possédant un ensemble fini et prédéfini de valeurs
  - Couleur :: **ROUGE, ORANGE, JAUNE, VERT, CYAN, BLEU, INDIGO, VIOLET, BLANC, NOIR**
  - type Pays :: **Belgique, Espagne, France, Italie**

la création d'un type énuméré simple

- mot clé : enum suivi du nom du type énuméré
- valeurs définies dans un bloc

```
public enum Couleur { ROUGE, ORANGE, BLEU, BLANC, NOIR }
```

```
public class Peinture {
   private Couleur couleur ;

   public Peinture(Couleur col) { couleur = col ; }
   public Couleur getCouleur() { return couleur ; }
   ……
}
```

quelques caractéristiques

- les enums sont des classes *final*
- les enums étendent la classe java.lang.Enum
  - quelques méthodes génériques :
    static <TYPE-ENUM> valueOf(String vString)
    int ordinal()
  - les enums implémentent l'interface java.lang.Comparable

- les valeurs ne sont PAS des entiers !! ce sont des références public static et final sur une instance de la classe

```
public class TestPeinture {

   public static void main(String[] args) {
         Peinture pRef = new Peinture(Couleur.BLEU) ;
         System.out.println(pRef.getCouleur());
         ……
   }
}
```

Les types énumérés (2)

l'utilisation des valeurs d'un enum comme case d'un switch

```
public enum Couleur { ROUGE, ORANGE, BLEU, BLANC, NOIR, VERT }
```

```
public class Peinture {
   private Couleur couleur ;

   public Peinture(Couleur col) { couleur = col ; }
   public Couleur getCouleur() { return couleur ; }
   ……
}
```

```
switch(peinture.getCouleur()) {
  case BLEU : …..
  case VERT : …..
                ……
  default :
}
```

les valeurs ne sont pas précédées par le nom de la classe

l'utilisation des valeurs d'un enum comme indice d'un tableau

- possibilité d'utiliser un type énuméré comme un indice de tableau via la méthode ordinal()

```
enum TestCouleur {
   public static final void main(String[ ] args) {
      boolean[] isPrimaryColor = { true, false, true, …… };
      System.out.println("la couleur " + Couleur.VERT + "est primaire : " + isPrimarycolor[Couleur.VERT.ordinal()]);
   }
}
```

Les types énumérés (3)

la définition d'un enum complexe

- les enums sont des classes à part entière
  - ils peuvent disposer de constructeurs, de méthodes, de champs, ...
  - ils peuvent implémenter des interfaces

```
public interface RGB {
   public int composanteRouge();
   public int composanteVert();
   public int composanteBleu();
   public int RGB();
}

public enum Couleur implements RGB {
   ROUGE(700), ORANGE(600), JAUNE(580), VERT(550), CYAN(510), BLEU(480),
   INDIGO(440), VIOLET(400), BLANC(-1), NOIR(0);
   static final private boolean[] primaires = { true, false, false, true, false, true, false, false, false, false, false };
   private float longeurOnde;

   private Couleur(float v) {  longeurOnde = v ; }                    //ctor obligatoirement private
   public float longueurOnde() { return longeurOnde; }
   public float energieMoyenne() { ..... }
   public boolean couleurPrimaire() { return primaires[ordinal()]; }
   public int composanteRouge() { ..... }
   public int composanteVert() { ..... }
   public int composanteBleu() { ..... }
   public int RGB() { ..... }
}
```

Quelques compléments sur les types énumérés (1)

la construction d'un masque de bits associé à un type énuméré

- La classe java.util.EnumSet permet de construire des masques à partir de types énumérés
- quelques méthodes
  static EnumSet allOf(Class eType)
  static EnumSet complementOf(EnumSet e)
  static boolean contains(Enum e)

```
EnumSet colorSet1 = EnumSet.allOf(Couleur);
EnumSet colorSet2 = EnumSet.of(Couleur.BLEU, Couleur.VERT, Couleur.NOIR);
boolean cont = colorSet2.contains(Couleur.BLEU) ;
```

l'équivalent java 1.4 d'un enum

```
public class Couleur {
    public static final Couleur BLEU = new Couleur(0) ;
    public static final Couleur VERT = new Couleur(1) ;
    public static final Couleur ROUGE = new Couleur(2) ;

    private static final String[ ] SVALUES = { "BLEU", "VERT", "ROUGE", ..... }
    private static final Couleur[ ] couleurs = { BLEU, VERT, ROUGE, ..... } ;
    private int ordinal;

    private Couleur(int ord) {    ordinal = ord ; }
    public int ordinal() { return ordinal; }
    public Couleur[ ] values()  { return couleurs; }
    public String toString() { return  SVALUES[ordinal]; }
}
```

Le nombre variable d'arguments de méthodes

quelques généralités

- construction introduite en java5
- permettre l'utilisation d'un nombre d'arguments inconnu a priori

les aspects syntaxiques

- nombre variable d'arguments exprimé via l'ellipse : …
- l'ellipse ne porte que sur un type
- l'ellipse porte nécessairement sur le dernier argument de la méthode
- le compilateur traduit l'ellipse par un tableau

```
public interface Ellipse1 {
    public void m1(int a0, float... a1);
    public  void m2(String... a0);
}
```

```
public class Ellipse2 {
    private int[] args;
    public void m1() {
        System.out.println("appel de m1()");
    }
    public void m1(int x) {
        System.out.println("appel de m1(int)");
    }
    public void m1(int...is) {
        System.out.println("appel de m1(int...)");
        if (is[0] == 0) args = is;
    }

    //public void m1(int[ ] is) { System.out.println("appel de m1(int[ ])"); }
}
```

interdit : ambigü

```
public class TEllipse2 {

    public static void main(String[] args) {
        Ellipse2 a = new Ellipse2();
        a.m1();
        a.m1(2);
        a.m1(0,1);
    }
}
```

```
appel de m1()
appel de m1(int)
appel de m1(int...)
```

résultat de l'exécution

L'autoboxing-unboxing des types primitifs (1)

la conversion des types primitifs et des classes wrapper

- mécanisme introduit en java5
- conversion transparente entre types primitifs et classes de wrapper correspondante

```
Integer integ0 = new Integer(1) ;
int i1 = integ0;                    //unboxing → affectation
int i2 = 2;
integ0 = integ0 + 2;                //unboxing → évaluation → boxing
Number n = 0.5f;                    //boxing → affectation
```

Conversion automatique et transparente

Conversion implicite de Float vers Number

quelques précautions .....

```
Integer i0 = 1;             //Integer i0 = Integer.valueOf(1)
Integer i1 = 1;             //Integer i1 = Integer.valueOf(1)
int i2 = 1;
if (i0 == i1)  System.out.println("i0 == i1");
else  System.out.println("i0 != i1");
if (i0 == 1)  System.out.println("i0 == 1");
else  System.out.println("i0 != 1");
if (i0 == i2)  System.out.println("i0 == i2");
else System.out.println("i0 != i2");
Integer j0 = 130;
Integer j1 =  130;
if (j0 == j1)  System.out.println("j0 == j1");
else System.out.println("j0 != j1");
if (j0 == 130)  System.out.println("j0 == 130");
else System.out.println("j0 != 130");
```

```
i0 == i1
i0 == 1
i0 == i2
j0 != j1
j0 == 130
```

```
Integer i0 = new Integer(1);
Integer i1 = new Integer(1);
int i2 = 1;
if (i0 == i1)  System.out.println("i0 == i1");
else  System.out.println("i0 != i1");
if (i0 == 1)  System.out.println("i0 == 1");
else  System.out.println("i0 != 1");
if (i0 == i2)  System.out.println("i0 == i2");
else System.out.println("i0 != i2");
```

```
i0 != i1
i0 == 1
i0 == i2
```

L'autoboxing-unboxing des types primitifs (2)

l'algorithme de résolution de surcharge des méthodes

- algorithme en 3 phases :
  - résolution sans prise en compte des mécanismes de boxing/unboxing
  - résolution avec prise en compte des mécanismes de boxing/unboxing sans prise en compte des varargs
  - résolution avec prise en compte des mécanismes de boxing/unboxing avec prise en compte des varargs

```
public class Exemple {
        .....

        public void m1(long arg0) { ..... }
        public void m1(double arg0) { ..... }
        public void m1(Integer arg0) { ..... }
        .....
}
```

```
public class TestExemple {
  public static void main(String[] args) {
        Exemple refEx = new Exemple();
        refEx.m1(4);
        Integer i0 = 4;
        refEx.m1(i0);
        refEx.m1(new Float(3.14f));
  }
}
```

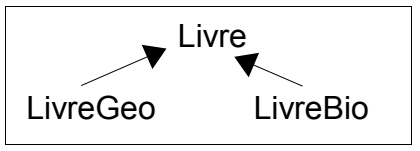
- mais ….

```
public class AmbiguousOverload1 {
   private void m0(long a0, Double a1) { ... }
   private void m0(Long a0, Double a1) { … }
   private void m0(Long a0, double a1) {… }
}
```

ambiguité multiple pour : ref.m0(1L, 3.14) ;

La redéfinition des méthodes

l'assouplissement de la contrainte sur le type de retour



- comportement introduit en java5
- possibilité de réduire le type de retour en cas de redéfinition de méthode

```
public interface ILibrairie {
   public Livre chercher(String titre);
}
```

```
public class Librairie implements ILibrairie {
   public Livre chercher(String titre) { ..... }
}
```

OK

```
public class LibrairieGeo extends Librairie  {
   public LivreGeo chercher(String titre) { ..... }
}
```

OK

```
public interface ILibrairie {
   public Livre chercher(String titre);
}
```

```
public class LibrairieBio implements ILibrairie {
   public LivreBio chercher(String titre) { ..... }
}
```

OK

```
public class LibrairieBioMoleculaire extends LibrairieBio {
   public Livre chercher(String titre) { ..... }
}
```

Erreur

L'import static

quelques généralités

- directive introduite en java5
- simplifier le code en permettant l'import :
  - des membres static d'une classe (champs et méthodes)
  - des constantes d'une interface

```
import static java.lang.System.out ;
import static java.lang.Math.PI;

public class StaticImportExemple {
   public static void main(String[] args) {
      out.println("exemple d'import static : " + PI);
   }
}
```

les aspects syntaxiques

- la syntaxe : import static .....
- possibilité d'utiliser * pour importer tous les membres static d'une classe
- possibilité d'importer les valeurs des types énumérés

```
package exemples;

public enum Couleur { BLEU, BLANC, VERT, JAUNE }
```

```
import static java.lang.System.out ;
import static java.lang.Math.*;
import static exemples.Couleur.*;

public class StaticImportExemple {
    public static void main(String[] args) {
       out.println("resultat de : " +  abs(sin(PI / 7)));
       out.println("couleur : " + BLEU);
    }
}
```

- possibilité de masquer les imports static par des variables !!!
- prise en compte de la surcharge pour les imports static de méthodes

La manipulation des chaines de caractères

la prise en compte de la nouvelle norme UNICODE

- prise en compte de la norme UNICODE 4.0
- surcharge de méthodes pour accepter des int

introduction de caractères nécessitant plus de 16 bits

quelques nouvelles fonctionnalités de lecture/écriture

- La classe Scanner permet lire les types de base en parsant le stream en entrée
  boolean hasNextLong(), booleab hasNextShort(), hasNextBoolean(), ...
  long nextLong(), int nextInt(), boolean nextBoolean(), ...
  findInline(String pattern),

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
...
Scanner sc2 = new Scanner(new
File("nombres.txt"));
while (sc2.hasNextLong()) {
long l = sc2.nextLong();
...
}
```

- nouvelle méthode printf(String format, Object... args) dans les classes PrintStream et PrintWriter
- le format suit les conventions de format généralisant ceux de la méthode printf du langage C
  System.out.printf("article : %s quantité : %d \n", libelle, nbr);

la manipulation des tableaux

- la classe **java.util.Arrays** offre de nouvelles méthodes utilitaires
  static boolean equals(TYPE[ ], TYPE[ ]),
  static int hashCode(TYPE[ ]),

  static boolean deepEquals(TYPE[ ], TYPE[ ]), static int deepHashCode(TYPE[ ]), static void deepToString()
  static void sort(TYPE[ ])

les nouveautés de java 6 (1)

les principales nouveautés

- introduction d'un framework d'intégration de scripting. L'API permet de :
  - connaître l'ensemble des interpréteurs intégrés dans la JVM,
  - d'exécuter un script (en récupérant éventuellement les résultats et/ou passant des objets java)

```
int intValue = 10;
ScriptEngineManager manager = new ScriptEngineManager();
// On récupère l'interpréteur de script JavaScript
ScriptEngine engine = manager.getEngineByName("JavaScript");
// On crée une association pour la variable Java 'intValue'
engine.getBindings(ScriptContext.ENGINE_SCOPE).put("intValue", intValue);
// Evaluation d'un script :
Object result = engine.eval(" ( 10 + intValue ) * 4 ");
// Affichage du résultat :
System.out.println("Résultat : " + result);
```

la liste n'est pas imposée :JavaScript (rhino), Python, Ruby, BeanShell, Groovy, ...

- version JDBC 4.0 (XML accepté comme un type SQL, meilleure intégration des BLOBs et CLOBs, annotations pour le mapping objet/relationnel)

- nouvelles classes dans Swing (classe SwingWorker), amélioration du drag and drop
- nouvelles interfaces Collection et nouvelles implémentations plus performantes

- API d'accès au compilateur. Il est possible de :
  - appeler le compilateur dans une application et de récupérer toutes les informations générées (messages d'erreur, warning, code généré, ...)
  - interagir avec le compilateur pour prendre en compte les annotations (API intégrant les fonctionnalités de *APT* dans java5)

les nouveautés de java 6 (2)

les principales nouveautés (suite)

- amélioration des performances de la JVM
- amélioration du format des .class (réduction importante de l'espace mémoire et accélération du chargement)

- nouvelles annotations prédéfinies : @Generated, @PostConstruct, @PreDestroy, @Resource

- nouvelles fonctionnalités d'administration et de supervision de la JVM (intégration de Jhat)

- locale pluggable

- développement de services Web interopérables avec *.NET* (via d'annotations)
- introduction dans le J2SE de nouveaux parsers XML (disponibles uniquement dans J2EE ou le pack Java Web Service)

les nouveautés de java 7

d'après le blog d'Alex Miller : http://tech.puredanger.com/java7/

les principales nouveautés

quelques nouveautés non intégrées :
les types réïfiés,
les clotures,
le support natif de XML, ...

Modularity

- Project Jigsaw YES
- JSR 294 Superpackages YES
- ~~Java Kernel~~ Java 6u10

Libraries

- JSR 203 NIO2 YES
- JSR 275 Units and Quantities HMM
- JSR 310 Date and Time API HMM
- JSR 166 Concurrency Utilities YES
- JSR 225 XQuery API for Java YES?
- Miscellaneous Library Changes

Swing

- JSR 296 Swing Application Framework YES
- JSR 303 Beans Validation HMM
- Java Media Components YES?

JMX

- JSR 255 JMX 2.0 YES
- JSR 262 Web Services Connector for JMX YES

Tools

Types and Generics

- JSR 308 Annotations on Java Types YES
- Type Inference YES

Language Proposals

Miscellaneous Language

- Strings in switch statements HMM
- Comparisons for Enums HMM
- Improved catch YES
- Null-safe handling YES
- Suggestions

JVM

- JSR 292 Dynamic language support YES
- Tiered compilation HMM
- G1 garbage collector YES

More script engines YES