Quelques nouveautés de Java 2v5

Bruno Mermet
Université du Havre
Masters 1 et 2
octobre 2004

Plan

- Modifications du langage
  - Boucle *for each*
  - Types énumérés
  - Import statique
  - Conversion automatique des types de base
  - Méthodes à nombre variable de paramètres
  - Généricité
- Modification dans les bibliothèques
  - Classe Scanner
  - *Classe Formatter : A FAIRE*
  - *Collections génériques : A FAIRE*

Boucle for each (1)

- But

  Parcourir simplement tous les éléments

  - D'un tableau
  - D'une collection

- Exemple sur un tableau

```
int somme(int [] tab) {
    int resultat = 0;
    for (int i : tab) {
        resultat += i;
    }
    return resultat;
}
```

Boucle for each (2)

- Exemples sur une collection

```
void affichage(Collection<String>  col) {
    for (String i : col) {
        System.out.println(i);
    }
}

void affichage(List  lis) {
    for (Object i : col) {
        System.out.println(i);
    }
}
```

Boucle for each (3)

- Contrainte d'utilisation

  Pas de possibilité de modifier le tableau ou la collection (rajouter, supprimer, déplacer un élément)

  Les méthodes des objets du tableau ou de la collection sont appelables ; les objets ou données sont utilisables en arguments de méthodes ou opérateurs

- Remarque

  aussi utilisable pour les types énumérés

Types énumérés
syntaxe de base

```
public class Meteo {
    public enum Saison {printemps, été, automne, hiver}

    private Saison saisonCourante;

    public Meteo() {
        saisonCourante = Saison.printemps;
        System.out.println(saisonCourante);
    }
}
```

Types énumérés
boucle *for each*

```
Import java.util.*;
public class Meteo {
    public enum Saison {printemps, été, automne, hiver}
    public static void listeSaison1() {
        for (Saison s : Saison.values()) {
            System.out.println(s);
        }
    }
    public static void listeSaison2() {
        for (Saison s : EnumSet.range(Saison.printemps,
          Saison.automne)) {
            System.out.println(s);
        }
    }
}
```

Types énumérés
nouvelles classes

- Classe java.lang.Enum<E extends Enum <E>>

- Classe java.util.EnumSet<E extends Enum <E>>

- Classe java.util.EnumMap<K extends Enum<K>, V>

Types énumérés
ajout d'un comportement global

```
public class Geometrie {
    public enum Point {
        A(0,0), B(1,0), C(1,1), D(0,1);
        private final int abscisse;
        private final int ordonnee;
        Point(int x, int y) {abscisse = x; ordonnee = y;}
        double distOrigine() {
            return Math.sqrt(abscisse*abscisse+ordonnee*ordonnee);
        }
    }
    public static void main(String [] args) {
        for (Point p : Point.values()) {
            System.out.println(p + «  :  » +p.distOrigine());
    }}}
```

Liste des constantes avec paramètres à passer au constructeur

Comportement

Utilisation

Types énumérés
comportement spécifique

```
public class Calcul {
    public enum Operation {
        PLUS {int calc(int a, int b) {return a+b;}},
        MOINS {int calc(int a, int b) {return a-b;}},
        MULT {int calc(int a, int b) {return a*b;}},
        DIV {int calc(int a, int b) {return a/b;}};
        abstract int calc(int a, int b);
    }
    public static void main(String [] args) {
        int a = 8, b = 2;
        System.out.println(Operation.PLUS.calc(a,b));
        for (Operation op : Operation.values()) {
            System.out.println(a +«  » +op «  »+b+ «=» + op.calc(a,b));
}}}
```

Import statique

- But

  Alléger l'écriture lors de l'utilisation de variables ou de méthodes de classe

- Avant

```
import java.lang.*;

double r = Math.cos(Math.PI * theta);
```

- Après, 1er exemple

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

double r = Math.cos(PI * theta);
```

- Après, 2ème exemple

```
import static java.lang.Math.*;

double r = cos(PI*theta);
```

Attention : uniquement raccourci syntaxique : PI ne devient pas une variable de classe de la classe dans laquelle on réalise l'import statique

Converstion automatique des types de base (*autoboxing*)

- Principe
  - Lorqu'un objet est attendu, il est possible de passer directement une donnée de type simple
  - Lorsqu'un objet d'une classe type Integer, Double, etc. est renvoyé, il est possible de l'utiliser comme une donnée du type simple associé
- Remarque
  - La différence entre les deux notions reste !

Converstion automatique des types de base
Exemples

- **Premier exemple : int → Integer (génère un *warning*)**

```
import java.util.*;

public class Conversion {
    public static void main(String [] args) {
Vector v;

v = new Vector();
for (int i = 0 ; i < 10 ; i++) {
    v.add(i);
}
System.out.println(v);
    }
}
```

- **Deuxième exemple : Integer → int**

```
public class Conversion2 {
        public static void main(String [] args) {
            Integer i1, i2;
            int result;
            i1 = new Integer(1);
            i2 = 2;
            result = i1 + i2;
            System.out.println(result);
        }
}
```

Méthodes à nombre variable de paramètres

- **But**

  Définir des méthodes pouvant prendre un nombre variable de paramètres

- **Notation**

  TypeResult nomMéthode(param1, param2, typeDuReste... params)

- **Principe**

  Dans la méthode, le dernier paramètre (param) est considéré comme un tableau dont les objets sont du type qui précède les « ... » (typeDuReste)

- **Utilisation**

  Passer les paramètres en nombre variables soit comme une liste de paramètres classiques, soit comme un tableau

Méthodes à nombre variable de paramètres
Premier exemple

```
public class NombreParametres {
   public static void affiche(int... liste) {
       for (int i = 0 ; i < liste.length ; i++) {
           System.out.println(liste[i]);
       }
       System.out.println("----");
   }


   public static void main(String [] args) {
      int[] tab = {1, 2, 3};
      affiche(tab);
      affiche(1, 2, 3);
      affiche(1, 2, 3, 4, 5);
   }
}
```

Méthodes à nombre variable de paramètres
Deuxième exemple

```
public class NombreParametres {
    public enum OPER {
        addition {int calc(int a, int b) { return a + b;} int init() {return 0;}},
        soustraction  {int calc(int a, int b) { return a - b;} int init() {return 0;}},
        multiplication  {int calc(int a, int b) { return a * b;} int init() {return 1;}},
        division  {int calc(int a, int b) { return a / b;} int init() {return 1;}};
        abstract int calc(int a, int b);
        abstract int init();
    }
    public static int operation(OPER op, int... liste) {
        int resultat;
        resultat = op.init();
        for (int i : liste) {
            resultat = op.calc(resultat, i);
        }
        return resultat;
    }

    public static void main(String [] args) {
        int resultat;
        resultat = NombreParametres.operation(OPER.addition, 1, 2, 3, 4, 5);
        System.out.println(resultat);
    }
}
```

Généricité : introduction

- Principe de base

  Définir des classes ou des méthodes paramétrées par un type

- Utilisation : exemple
  - Déclaration des variables

```
List<Integer> maListe;
Vector<Point> vec;
```

  - Création des objets

```
maListe = new List<Integer>();
vec = new Vector<Point>();
```

  - Récupération des données

```
Point p;
p = vec.elementAt(0);
```

Attention !
Si B hérite de A, C<B> n'hérite pas de C<A>

Généricité : déclaration de méthodes

- Exemple 1

```
Void afficherVecteur(Vector<Object> v) {
    For (Object o : v)  {
        System.out.println(o);
    }
}
```

⇒ Problème : applicable uniquement à un Vector<Object>

- Exemple 2

```
Void afficherVecteur(Vector<?> v) {
    For (Object o : v)  {
        System.out.println(o);
    }
}
```

⇒ Problème : on perd en partie le gain en typage lié à la généricité

Méthodes à généricité contrainte

- Syntaxe
  - Utilisation du joker « ? » avec le mot-clef « extends »
  - Si la méthode prend un paramètre p de type Vector<?extends A>, elle pourra accepter en paramètre tout vecteur d'objets A ou d'objets héritant de A
- Exemple

```
public class Dessinable {
    void dessiner() {...}
}
public static void dessinerVecteur(Vector<? Extends Dessinable> v) {
    For (Dessinable d : v)  {
        d.dessiner();
    }
}
```

- Si Rectangle hérite de Dessinable, on pourra faire :

```
Vector<Dessinable> vd = new Vector<Dessinable>,
Vector<Rectangle> vr = new Vector<Rectangle>;
Dessinable.dessinerVecteur(vd);
Dessinable.dessinerVecteur(vr);
```

Lien entre les types

- Problème

  Une méthode doit prendre en paramètre deux types, le deuxième devant être un sous-type du second

- Solution

  Utiliser des variables de type

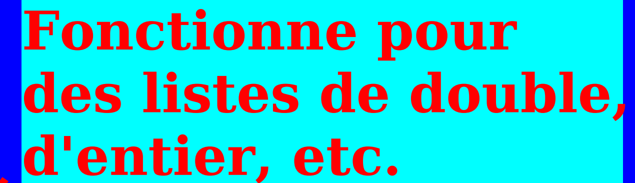
- Convention

  Variables d'une seule lettre, majuscule

Exemple

```
public static <S, T extends S> S affectation(S dest, T src) {
    dest = src;
    return dest;
    }
```

Classe générique : exemple complet




```
import java.util.*;
public class PileGenerique<T> {
    private Vector<T> vecteur;
    public PileGenerique() {vecteur = new Vector<T>();}
    public boolean estVide() {return (vecteur.size() == 0);}
    public void empiler(T donnee) {vecteur.add(donnee);}
    public T depiler() {
        T sommet;
        sommet = vecteur.elementAt(vecteur.size()-1);
        vecteur.removeElement(sommet);
        return sommet;
    }

    public static double sommer(PileGenerique<? extends Number> p) {
        double somme = 0;
        for (Number n : p.vecteur) {somme += n.doubleValue();}
        return somme;
    }

    public static void main(String [] args) {
        PileGenerique<Integer> pile = new PileGenerique<Integer>();
        double somme;
        int val;
        pile.empiler(1);pile.empiler(2);pile.empiler(3);
        val = pile.depiler();System.out.println(val);
        somme = sommer(pile);System.out.println(somme);
    }
}
```

Classe Scanner : bases

- But : faciliter la lecture
  - De fichiers
  - De l'entrée standard
- Constructeurs principaux
  - Scanner(File source)
  - Scanner(InputStream source)
  - Scanner(String source)
- Application des constructeurs
  - Scanner s = new Scanner(new File(«monFichier.txt»));
  - Scanner s = new Scanner(System.in);
  - String ch = «Ma chaine»; Scanner s = new Scanner(ch);
- Ne pas oublier d'inclure java.util.*

Classe Scanner : principes généraux

- Lecture (adaptée à la langue)
  - Lire un entier : nextInt()
  - Lire un double : nextDouble()
  - ...
  - Lire une chaine : nextLine()
  - ...
- Tester sans consommer
  - hasNextInt()
  - ...
- Gestion des erreurs
  - Utiliser l'exception InputMismatchException (qui hérite de RuntimeException)
- Permet aussi de faire de la recherche d'expression régulière