# JAVA ET JDBC : UNE MINI-INTRODUCTION


## MARCIN SKUBISZEWSKI

***skubi@skubi.net***
***http://www.skubi.net***


### Comment nous procédons

- Etude de quelques points essentiels et difficiles
- Etude d'exemples de code

### Manuels

- Tous les documents sont sur `www.javasoft.com`

# JAVA

- Un langage à objets
- Dérivant de C, quelques similitudes avec C++
- Priorités
  - Portabilité
  - Concepts du langage peu nombreux, clairs et simples
  - La programmation doit être facile
  - On réduit la chance laissée au programmeur de faire des erreurs
    - pas de destruction manuelle d'objets (ramasse-miettes)
- Priorités moindres
  - Flexibilité
  - Compatibilité C
- Priorités très faibles
  - Efficacité
  - Compatibilité C++
  - Elégance apparente
    - pas de fonctions, seulement des méthodes
      - on écrit `Math.max(a,b)` et non `max(a,b)`

# PORTABILITE

- **En théorie**, tout le monde fait de la portabilité
  - C, C++ sont normalisés
  - POSIX est une normalisation de Unix
  - On peut compiler et exécuter sur MS Windows du code Unix non-graphique
- **En pratique**, c'est très imparfait
  - Chaque compilateur C++ a des particularités et des bugs différents
    - On peut écrire du code qui passe pour tous les compilateurs, mais c'est restrictif
    - En pratique, on peut écrire du code portable, moyennant **un effort spécifique**
  - Chaque SGBD apporte des restrictions différentes à SQL
    - même conséquence
  - Chaque système de famille Unix met en oeuvre POSIX à sa façon.

**La portabilité de programmes Java**

- On veut que tous les programmes soient portables, sans effort spécifique
- Le même exécutable doit fonctionner sur toutes les plateformes
  - même s'il est écrit sans aucun effort spécifique de portabilité
  - sans recompilation

# JDBC

- Officiellement n’est pas un acronyme
- En fait signifie ***J****ava* ***D****ata****b****ase* ***C****onnectivity*
- Bibliothèque pour exécuter du SQL à partir de Java
  - client-serveur
  - bas niveau
    - différent des systèmes à objets persistants
    - on manipule explicitement les objets de la base de données, qui sont distincts des objets Java
- Fait partie de JDK (*Java Development Kit*), bibliothèque fondamentale Java
- Mêmes objectifs de portabilité que Java
  - le code doit fonctionner quelque soit la machine qui l’exécute
  - et aussi, quelque soit le SGBD auquel on se connecte

# LES POINTEURS EXPLICITES

## En C, C++, Pascal

Pour un type `A`, on peut avoir

```
A a1, a2;
A *p1, *p2;
```

Copier un pointeur (le nombre et le contenu d'objets de type `A` ne change pas)

```
p1 = p2;
```

Copier un objet (le contenu d'un objet de type `A` change)

```
a1 = a2;
*p1 = *p2;
```

Possible

```
class A {
   ...
   A *next;
   B plusPetitObjet;
}
```

# LES POINTEURS EXPLICITES, suite

Impossible

```
class A {
   ...
   A next;
}
```

Appel de méthode

```
a1.m();
(*p1).m();
p1->m();
```

Les objets et les pointeurs (en fait, toutes les données) peuvent être mémorisés dans la pile et dans le tas

# LES POINTEURS IMPLICITES

**En Java, O2C, Lisp**

On distingue entre

- les *objets*, entités sur lesquels un pointeur (implicite) peut pointer
- les autres entites, sur lesquelles on ne peut pas pointer

Pour un type `A`, on peut avoir seulement la déclaration suivante

```
A a1, a2;
```

- si `A` n'est pas un type d'objet, la signification de cette déclaration est ordinaire :
  - `a1`, `a2` contiennent chacune une donnée de type `A`
  - il est impossible de déclarer un pointeur sur un `A`
- si `A` est un type d'objet, alors
  - `a1`, `a2` contiennent chacune un pointeur sur un objet de type `A`
  - syntaxiquement, on écrit tout comme si `a1`, `a2` contenaient des objets de type `A`
  - il est impossible de déclarer une variable susceptible de contenir un objet de type `A`

# LES POINTEURS IMPLICITES, suite

Copier un pointeur (le nombre et le contenu d'objets de type `A` ne change pas)

```
a1 = a2;
```

Copier un objet (le contenu d'un objet de type `A` change): il faut une syntaxe spécifique

```
a1 = a2.clone();
```

Possible

```
class A {
   ...
   A next;
}
```

Impossible : avoir un objet à l'intérieur d'un autre

# LES POINTEURS IMPLICITES, suite

Les objets peuvent être mémorisés dans le tas seulement

- le système gère cela
- impossible de déclarer une variable susceptible de contenir un objet

Les entités qui ne sont pas des objets peuvent être mémorisés dans la pile seulement

- car une donnée du tas ne peut être accédée qu'à travers un pointeur

Exemple de conséquence : en Java, il y a

- des entiers `int` (non-objets)
- des entiers `Int` (objets)

# CLASSES, INTERFACES, HERITAGE MULTIPLE

## Interface

- une classe où on déclare seulement des méthodes, pas de champs
- les méthodes ne sont pas définies
  - une classe abstraite

## Restriction à l'héritage multiple

- une classe hérite
  - d'un nombre illimité d'interfaces (héritage multiple d'interfaces)
  - d'au plus une classe non-interface
- une interface hérite
  - d'un nombre illimité d'interfaces, mais pas de classes (c'est logique)

# INTERFACES, suite

## Raison d'être des interfaces

- un véritable héritage multiple rend le langage très compliqué
  - C++ met en oeuvre un véritable héritage multiple
  - Java ne le fait pas
- l'héritage multiple est réellement utile
- une solution intermédiaire est donc utile

## Utilité des interfaces

- en programmation propre, généralement seules des méthodes sont publiques, les champs sont tous privés
  - donc, avec des méthodes seules, on peut définir le comportement d'une catégorie d'objets (même si on ne peut pas mettre en oeuvre ces objets)

# EXEMPLE D'INTERFACE

- on notifiera à un objet le fait qu'une fenêtre à subi un évènement
- chaque notification sera faite par un appel de méthode
- l'objet devra donc avoir certaines méthodes, il devra hériter de l'interface suivante

```
public interface netscape.application.WindowOwner {


   public abstract void windowDidBecomeMain(Window);
   public abstract void windowDidHide(Window);
   public abstract void windowDidResignMain(Window);
   public abstract void windowDidShow(Window);
   public abstract boolean windowWillHide(Window);
   public abstract boolean windowWillShow(Window);
   public abstract void windowWillSizeBy(Window, Size);
}
```

# EXEMPLE DE CODE UTILISANT `WindowOwner`

```
public class AuditView extends ContainerView
implements WindowOwner {
   ImageFinder _imageFinder;
   PackLayout _layout;
   int _maxWidth = 0;
   ...
   public static Font nameFont =
       Font.fontNamed("Helvetica:Plain:10");
   // variable globale, s'accede par
   // AuditView.nameFont
   ...
```

# EXEMPLE DE CODE, suite

```
public AuditView(Window w, ScrollGroup a, View [] but)
   {
       super();
       int i;
       ancestor = a;
       _window = w;
       _buttons = but;
       w.setOwner(this);
       setTransparent(true);
       setBorder(new EmptyBorder());
       setDirty(true);
       w.setBounds(0, 0, 400, 400);
   }
   ...
   String drawRelatedClasses(SaverStringTokenizer st,
   Vector text, int indexForMore) {
       _headerView = new HeaderView(st, text,
           indexForMore, this);
       return _headerView.name;
   }
}
```