

PPOOGL

Florent de Dinechin

**Java pour les nuls
(et les physiciens)**

- ① Introduction
- ② Grands principes
- ③ Pour en finir avec l'organisation
- ④ Types de base et objets
- ⑤ Le reste est dans le manuel

Introduction

- 1 Introduction
- 2 Grands principes
- 3 Pour en finir avec l'organisation
- 4 Types de base et objets
- 5 Le reste est dans le manuel

La première fois que vous mettez votre main dans le java

- Fichier source `Toto.java`, fichier objet `Toto.class`
- `javac Toto.java` crée `Toto.class` (le c c'est pour *compilo*)
- `java Toto` lance la méthode main de `Toto.class` dans la machine virtuelle.
- Si on faisait une applet, c'est pas une méthode main qu'il faudrait.
- Et voici le code source :

```
class Hello {  
    // pas d'attributs ni de méthodes, sauf  
    // la méthode main, statique.  
    // cela veut dire quoi, statique, déjà ?  
    public static void main(String[] args) {  
        System.out.println("Hello, java ?");  
    }  
}
```

Grands principes

- 1 Introduction
- 2 Grands principes**
- 3 Pour en finir avec l'organisation
- 4 Types de base et objets
- 5 Le reste est dans le manuel

- *Une classe, un fichier*

- ⊕ Pas de fichiers `.h`
- ⊕ La doc du code n'est plus le fichier `.h`, c'est une vraie doc créée par `javadoc`
- ⊕ Pas besoin de `Makefile` ni d'`autoconf`
- ⊕ Pas de bug dû au préprocesseur
- ⊖ Si cela se trouve, c'est même mieux que Caml

- *Édition de lien dynamique*

- Lorsque la machine virtuelle a besoin de créer un objet de classe `Toto`, elle charge `Toto.class`
- Il faut qu'elle sache où le trouver (on verra plus loin)
- ⊕ Pas d'éditeur de lien
- ⊕ Pas de `Makefile` (je l'ai déjà dit ?)
- ⊕ Les chemins pour la compilation, pour l'édition de liens et pour l'exécution sont les mêmes.
- ⊖ Performance : des accès disques (et même réseau) cachés au milieu de votre programme

applet (en français *appliquette* ou *programmouillette*) :

- destiné à tourner dans une fenêtre de navigateur chez des gens qu'on ne connaît même pas
- dérive d'une classe qui en principe protège la machine hôte (*sandbox*)
 - pas d'écriture ni de lecture du disque local
 - par contre accès en lecture à tous les fichiers du Ternet par leur *url*

application (en français *application*)

- pas les restriction précédentes
- un navigateur n'a pas plus le droit de la lancer qu'un autre exécutable

Applications et appliquettes partagent le gros de la bibliothèque standard, notamment l'*awt* (*abstract windowing toolkit*).

JavaScript n'a rien à voir avec Java à part la syntaxe superficielle : c'est un langage interprété, plein de trous de sécurité, pas OO pour un sou, et qu'on laissera volontiers aux authentiques kakous.

- Un outil qui prend du code bien documenté, et construit une doc html toute jolie
- Les commentaires pour Javadoc sont entre `/** ... */`
- Un commentaire de ce type par méthode, attribut, classe, etc.
- Ligne de commande : `javadoc Toto.java`
- Les docs de référence des classes standard du langage sont construites comme cela

Exemple de commentaire de l'an dernier :

```
/**  
 * classe Cartes  
 */  
public class Cartes extends Remote ...
```

Pour en finir avec l'organisation

- 1 Introduction
- 2 Grands principes
- 3 Pour en finir avec l'organisation**
- 4 Types de base et objets
- 5 Le reste est dans le manuel

- Un paquetage c'est comme un module en Caml.
- Les paquetages sont organisés hiérarchiquement en répertoires.
Rien à voir avec la hiérarchie des classes.

- Exemples tirés des bibliothèques standard :

<code>java.applet</code>	<code>java.math</code>
<code>java.io</code>	<code>java.awt</code>
<code>java.awt.image</code>	<code>java.awt.event</code>

- On déclare qu'une classe `Toto` fait partie d'un paquetage `projetLala.tata` en mettant tout au début de `Toto.java` :

```
package projetLala.tata ;
```

- En l'absence d'une telle ligne, la classe fait partie du paquetage par défaut, constitué de
 - l'ensemble des classes dans le répertoire courant
 - l'ensemble des classes accessibles par la variable `CLASSPATH`
- On peut ranger un paquetage dans une archive zip, qui s'appellera d'ailleurs `jar`, mais il faut lire le manuel.

- Le nom complet d'une classe est `paquetage.Classe`
- Unicité planétaire des noms par l'URL renversée :
`com.projetMIM2002.www.lala.test`
- Le nom complet d'un membre est `paquetage.Classe.membre`
 - la fonction sinus : `java.lang.Math.sin(x)`
 - la variable à l'unicité planétaire :
`com.projetMIM2002.www.lala.test.RandomTest.nombreErreurs`
- Si on a la flemme de taper tout cela tout le temps dans `Toto.java`, on peut *importer* une fois pour toutes, au début de ce fichier,
 - une classe : `import java.lang.Math ;`
 - ou bien toutes les classes d'un paquetage :
`import java.lang.* ;`
- On n'est pas dispensé de taper `Classe.membre` ou `objet.membre...` Exemple `Math.sin(x)`.

Quelques conventions

- Tout le monde appelle ses classes avec une majuscule. Du coup les fichiers aussi.
- Les membres de la plupart des programmeurs sont minuscules.
- Vous faites bien ce que vous voudrez.

- Dans le corps d'une méthode, c'est comme en C.
- Dans le corps d'une classe, un membre peut être
 - `public` : tout le monde le voit même en dehors de la classe
 - `private` : visible uniquement à l'intérieur de la classe
 - `protected` : visible dans la classe, ses sous-classes, et le paquetage
 - rien du tout : visible dans le paquetage
- Dans un paquetage, une classe peut-être
 - `public` : tout le monde le voit même en dehors du paquetage
 - rien du tout : visible dans le paquetage seulement
- Vous trouverez plein d'exemples sur le Ternet.

Types de base et objets

- 1 Introduction
- 2 Grands principes
- 3 Pour en finir avec l'organisation
- 4 Types de base et objets**
- 5 Le reste est dans le manuel

- Entiers *signés* de différentes tailles : byte (8 bits), short (16), int (32), long (64)
- Nombres en virgule flottante : float et double
- boolean qui vaut true ou false
- char est un caractère Unicode (sur 16 bits !)

- Tous ces types ont une valeur initiale spécifiée par le langage, mais javac fait des warning si on ne les initialise pas.

```
class HelloHello {
    public static void main(String[] args) {
        int i;
        for(i=0; i<10; i++) {
            System.out.println("Hello, java, " + i + " fois?");
        }
    }
}
```

```
import java.awt.*;

class HelloLaFenetre {
    public static void main(String[] args) {
        Frame objet_fenetre;

        objet_fenetre = new Frame("Hello, java ?") ;
        objet_fenetre.setSize(300,100);
        objet_fenetre.setVisible(true);
    }
}
```

La vraie classe

```
import java.awt.*;

public class ObjetHello {

    private Frame fenetre;

    public void construit(String titre){
        fenetre = new Frame(titre) ;
        fenetre.setSize(300,100);
        fenetre.setVisible(true);
    }

    //constructeurs
    public ObjetHello(){
        construit("Hello, java ?");
    }

    public ObjetHello(String titre){
        construit(titre);
    }

    // Pas besoin de main() ici
}
```

```
//import java.awt.*; // plus besoin ici

public class ProjetHello {

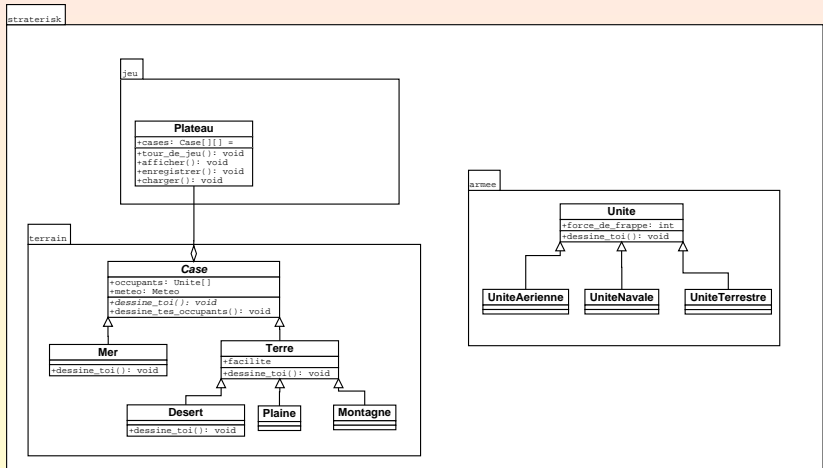
    public static void main(String[] args) {
        // un objet que c'est moi qui l'ai fait
        ObjetHello uoqcmqlaf;

        uoqcmqlaf = new ObjetHello();

        uoqcmqlaf = new ObjetHello("Youpi!");

    }
}
```

Reprenons cet UML bâclé :



et passons-le à dia2code

```
package straterisk.terrain;

import straterisk.jeu.Plateau;

public abstract class Case {
    /** Attributes */
    public Unite[] occupants;
    public Meteo meteo;
    /** Associations */
    private Plateau;
    /**
     * Operation
     *
     */
    abstract public void dessine_toi ( );
    /**
     * Operation
     *
     */
    public void dessine_tes_occupants ( ){
    }
}
```

```
package straterisk.terrain;

import straterisk.terrain.Case;

public class Terre extends Case {
    /** Attributes */
    public facilite;
    /**
     * Operation
     *
     */
    public void dessine_toi ( ){
    }
}
```

```
package straterisk.terrain;

import straterisk.terrain.Terre;

public class Desert extends Terre {
    /**
     * Operation
     *
     */
    public void dessine_toi ( ){
    }
}
```

dia2code c'est pas encore cela

- Il gère bien les `import` et `package` dans le code source, mais ne construit pas la hiérarchie des répertoires
- La version du CRI est un peu vieille
- Utilisable uniquement en phase initiale

Mais bon, cela vous oblige à bien réfléchir votre modèle objet avant de coder...

- String est une classe d'objets (majuscule...) mais avec du sucre syntaxique dans le langage pour le constructeur, la concaténation (+), ...

```
int i,j;  
String errorMessage;  
(...)  
errorMessage="Feature not implemented, because we  
started the project two days before the deadline";
```

Méthodes d'une chaîne : `errorMessage.length()`, et plein d'autres.

- Il y a aussi des chaînes de taille variable (`StringBuffer`), etc

Les tableaux sont aussi des objets avec du sucre syntaxique et sémantique (classe paramétrée).

```
public class SpaceHello {  
  
    public static void main(String[] args) {  
        ObjetHello[] plein_de_fenetres;  
  
        if(args.length == 0)  
            System.err.println("Usage: java SpaceHello text ");  
        else  
            {  
                int i;  
                plein_de_fenetres = new ObjetHello[args.length];  
                for(i=0; i<args.length; i++)  
                    plein_de_fenetres[i] = new ObjetHello(args[i]);  
            }  
    }  
}
```

Le reste est dans le manuel

- 1 Introduction
- 2 Grands principes
- 3 Pour en finir avec l'organisation
- 4 Types de base et objets
- 5 Le reste est dans le manuel**

- Le site de Sun
 - Des tutoriels
 - La doc de référence sur le langage (on s'en passe bien)
 - La doc de références sur les classes standard (à parcourir absolument !)
 - le tout téléchargeable pour pouvoir partir en vacances avec
- Plein de bouquins à la bibliothèque
- Celui de Brondeau en français
 - ⊕ court
 - ⊕ en français
 - ⊖ un peu lège sur l'OO