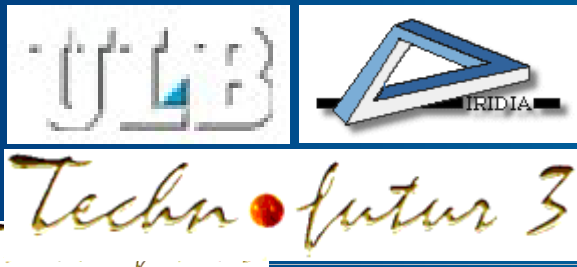


# Introduction à Java

Hugues Bersini

Code/IRIDIA – Université Libre de Bruxelles



# Objectifs du cours (1/2)

- Décrire les éléments-clé de la plate-forme Java
- Compiler et exécuter une application Java
- Prendre en mains l'environnement de développement Eclipse
- Comprendre et utiliser la documentation en ligne de Java
- Décrire la syntaxe du langage
- Comprendre le paradigme OO et utiliser Java pour le mettre en œuvre
- Comprendre et utiliser les exceptions

# Objectifs du cours (2/2)

- Etre capable, au terme de la formation, de développer de petites applications OO comprenant une dizaine de classes et mettant en œuvre les principaux concepts OO et structures Java.
- Le cours couvre l'essentiel de la matière des examens « OO for Java – Basic », « Java SE – Basic » et « Java SE Core – Intermed » de Java BlackBelt ([www.javablackbelt.com](http://www.javablackbelt.com))

# Plan du cours (1/4)

## 1. Introduction générale et historique

- Le langage de programmation Java
- La plateforme Java
- Les versions de Java

## 2. Première application en Java

- Ecriture du code, compilation et exécution
- Application v/s Applet
- Utilitaires Java

## 3. Syntaxe et sémantique de Java

- Identificateurs
- Types primitifs et types de référence
- Tableaux et chaînes de caractères
- Arithmétique et opérateurs
- Instructions de contrôle

# Plan du cours (2/4)

## 4. Programmation orientée objets en Java

- Programmation procédurale v/s Programmation OO
- Concepts de l'OO
- La création d'objets: Constructeurs et mot-clé « new »
- Les variables: Déclaration et portée
- Les méthodes: Déclaration, interface et surcharge
- L'encapsulation: « public », « private » et « protected »
- Les membres d'instance et de classe: « static »
- Utilisation de l'héritage: « this » et « super »
- Conversion de types
- Polymorphisme
- Classes abstraites
- Interfaces

# Plan du cours (3/4)

## 5. Structure des API de Java

## 6. Les collections

- Aperçu du Java Collections Framework
- La classe ArrayList

## 7. La gestion des Exceptions

- Principes et hiérarchie des classes d'exceptions
- Interception par bloc *try – catch – finally*
- Lancement par mots-clés *throws* et *throw*

# Quelques sujets non couverts

- Développement des applets
- Interfaces graphiques (Swing et AWT)
- Développement d'application clients/serveur
  - TCP/IP
  - UDP
- Enterprise Java Beans (EJB)
- Servlets et Java Server Pages (JSP)
- Connections à des bases de données (JDBC)

# Références Web

- The Java Tutorial from Sun

<http://java.sun.com/docs/books/tutorial/>

- Thinking in Java

<http://www.thinkinginjava.org/>

<http://penserinja.free.fr/>

- The Java Developer Connection

<http://developer.java.sun.com/developer/index.html>

- Gamelan

<http://www.gamelan.com>

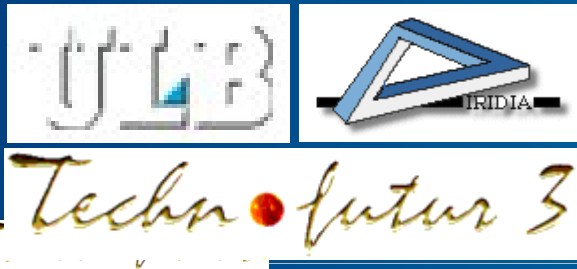
- Java Applet Rating Services

<http://www.jars.com>



# Introduction à Java

## I. Introduction et historique



# Survol du chapitre

- **Qu'est-ce que Java ?**
- **Java comme langage de programmation**
- **La plateforme Java**
  - La Java Virtual Machine
  - Les interfaces de programmation d'application (API)
- **Déploiement d'un programme**
- **Les versions de Java**
- **Quelques notions historiques**

# Qu'est-ce que Java ?

- **Java est un langage de programmation**

- Voir le « white paper » de J.Gosling
- Un programme Java est compilé et interprété

- **Java est une plateforme**

- La plateforme Java, uniquement software, est exécutée sur la plateforme du système d'exploitation
- La « Java Platform » est constituée de :
  - La « Java Virtual Machine » (JVM)
  - Des interfaces de programmation d'application (Java API)

# Java comme langage de programmation

**Java est un langage de programmation particulier qui possède des caractéristiques avantageuses:**

- Simplicité et productivité:
  - Intégration complète de l'OO
  - Gestion mémoire (« Garbage collector »)
- Robustesse, fiabilité et sécurité
- Indépendance par rapport aux plateformes
- Ouverture:
  - Support intégré d'Internet
  - Connexion intégrée aux bases de données (JDBC)
  - Support des caractères internationaux
- Distribution et aspects dynamiques
- Performance

# Java comme langage de programmation

## Simple et orienté objet

- **Java est un langage de programmation simple**
  - Langage de programmation au même titre que C/C++/Perl/Smalltalk/Fortran mais plus simple
  - Les aspects fondamentaux du langage sont rapidement assimilés
- **Java est orienté objet :**
  - La technologie OO après un moment de gestation est maintenant complètement intégrée
  - En java, tout est un objet (à la différence du C++ par ex.)
- **Simple aussi parce qu'il comporte un grand nombre d'objets prédéfinis pour l'utilisateur**
- **Java est familier pour les programmeurs C++**

# Java comme langage de programmation

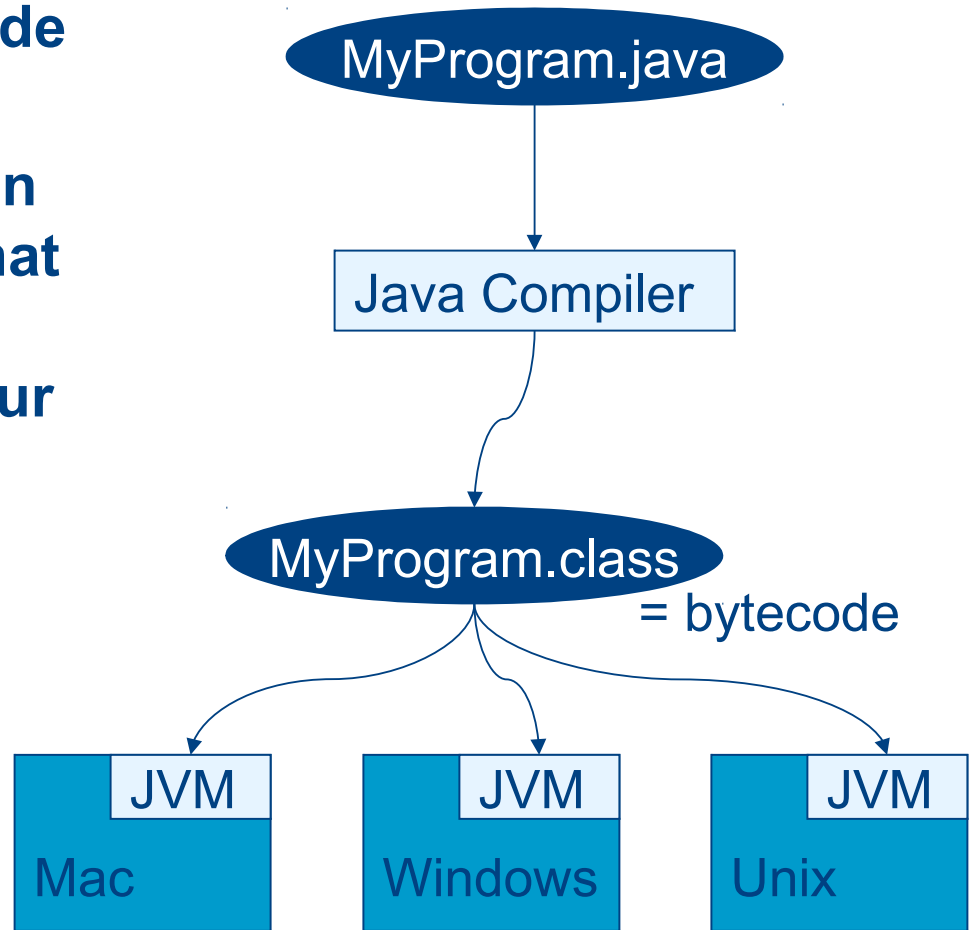
## Robuste et sécurisé

- Conçu pour créer des logiciels hautement fiables
- Oblige le programmeur à garder à l'esprit les erreurs hardware et software
- Vérifications complètes à l'exécution et à la compilation
- Existence d'un « garbage collector » qui permet d'éviter les erreurs de gestion de la mémoire

# Java comme langage de programmation

## Neutre architecturalement

- Il existe une grande diversité de systèmes d'exploitation
- Le compilateur Java génère un bytecode, c'est à dire un format intermédiaire, neutre architecturalement, conçu pour faire transiter efficacement le code vers des hardware différents et/ou plateformes différentes
- Le bytecode ne peut-être interprété que par le processeur de la JVM



# Java comme langage de programmation

## Ouvert et distribué

- **Support intégré d'Internet**

- La Class URL
- Communication réseaux TCP et UDP
- RMI, CORBA, Servlets
- JINI, le « must » pour construire des applications complexes distribuées.....

- **Connectivité aux bases de données**

- JDBC: Java DataBase Connectivity
- Offre des facilités de connexions à la plupart des BD du marché
- Offre un pont vers ODBC

- **Support des caractères internationaux**

- Java utilise le jeu de caractères UNICODE
- JVM équipée de tables de conversion pour la plupart des caractères
- JVM adapte automatiquement les paramètres régionaux en fonction de ceux de la machine sur laquelle elle tourne



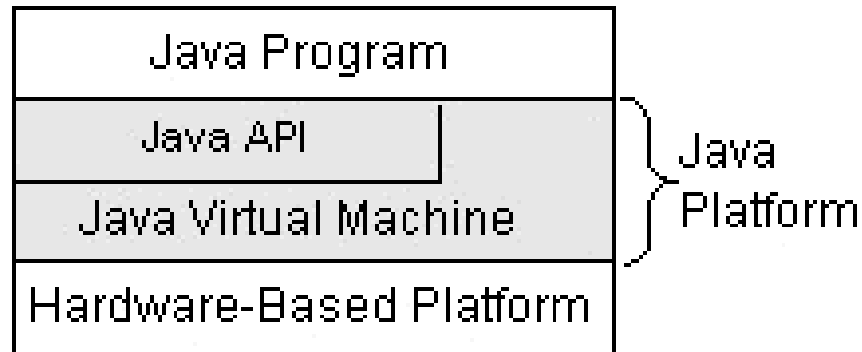
# Java comme langage de programmation

## Performant

- **Considération basique**
- **Exécution ralentie à cause de l'interpréteur ?**
- **Le code natif généré par l'interpréteur est-il aussi rapide que celui réalisé par un compilateur classique (par ex C)?**
- **Plusieurs processus peuvent être exécutés en même temps**
  - Comparable au multitâche d'un OS
  - Le temps du CPU est divisé (sliced)
  - Toutes les bibliothèques Java
- **Edition de lien effectuée à l'exécution du programme**
- **Codes exécutables chargés depuis un serveur distant permet la mise à jour transparente des applications**

# Java comme Plateforme

- **Plateforme = environnement hardware ou software sur lequel le programme est exécuté.**
- **La Java « Platform » se compose de:**
  - la Java Virtual Machine (Java VM)
  - la Java Application Programming Interface (Java API)



# Java comme Plateforme

## Java Application Programming Interface (API)

- **L'API Java est structuré en libraires (packages). Les packages comprennent des ensembles fonctionnels de composants (classes)..**
- **Le noyau (core) de l'API Java (incluse dans toute implémentation complète de la plateforme Java) comprend notamment :**
  - Essentials (data types, objects, string, array, vector, I/O,date,...)
  - Applet
  - Abstract Windowing Toolkit (AWT)
  - Basic Networking (URL, Socket –TCP or UDP-,IP)
  - Evolved Networking (Remote Method Invocation)
  - Internationalization
  - Security
  - .....

# Java comme Plateforme

## Java Virtual Machine (1/2)

- « An imaginery machine that is implemented by emulating it in software on a real machine. Code for the JVM is stored in .class files, each of which contains code for at most one public class »
- Définit les spécifications hardware de la plateforme
- Lit le bytecode compilé (indépendant de la plateforme)
- Implémentée en software ou hardware
- Implémentée dans des environnements de développement ou dans les navigateurs Web

# Java comme Plateforme

## Java Virtual Machine (2/2)

### La JVM définit :

- Les instructions de la CPU
- Les différents registres
- Le format des fichiers .class
- Le « stack »
- Le tas (« Heap ») des objets « garbage-collectés »
- L'espace mémoire

# Java comme Plateforme

## Java Runtime Environment

### Trois tâches principales :

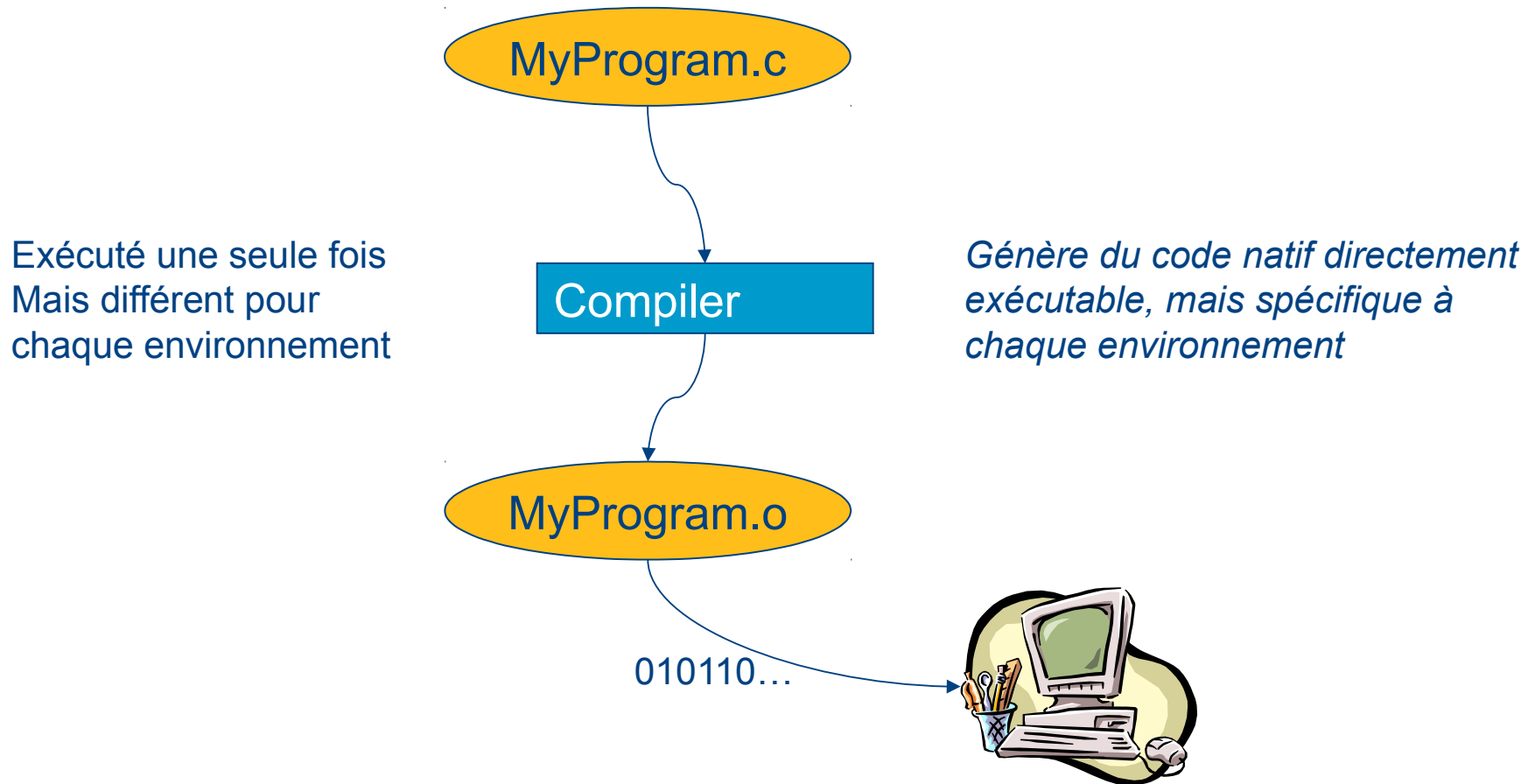
- Charger le code (class loader)
- Vérifier le code (bytecode verifier)
- Exécuter le code (runtime interpreter)

### D'autres THREAD s'exécutent :

- Garbage collector
- (JIT compiler)

# Déploiement d'un programme (1/2)

## Paradigme classique de la compilation



# Déploiement d'un programme (2/2)

## Changement de la vision traditionnelle de la compilation

- Chaque programme est compilé et interprété
- « Write once run everywhere »

MyProgram.java

Exécuté une seule fois

Compiler

*Traduit le programme en un code intermédiaire  
Appelé bytecode – indépendant de la machine*

MyProgram.class

Chaque fois que le  
programme est exécuté

Interpreter

*Lit le bytecode et exécute sur la machine*

010110...





# Les versions de Java

- SE v/s EE, annexes
- SDK v/s JRE
- 1.1 v/s 1.2 et suivantes
- Télécharger l'environnement Java et Eclipse

# Bref Historique

- **1991: Développement de OAK**
  - langage simple, portable et orienté objets
  - pour la programmation d'appareils électroniques ménagers
  - emprunte la portabilité du Pascal (VM) et la syntaxe de C++
- **1994: Abandon du projet OAK**
  - Peu d'enthousiasme pour l'idée
- **1995: Intégration de la JVM dans Netscape**
  - Apparition des Applets
  - Explosion d'Internet → attrait grandissant pour Java
- **1999: Apparition de JINI**
  - Nouvelle technologie basée sur Java
  - Reprend l'ambition de départ d'un plug and play universel
  - Distribué sur tous les appareils munis d'un processeur
- **2006: Java devient Open Source**
  - Les sources de la plateformes Java sont désormais libres sous licence GNU

# Introduction à Java

## II. Première application en Java



# Comment développer une application?

## Deux façons d'écrire des programmes Java:

- **En écrivant le code dans un simple éditeur de texte**
  - Compilation et exécution du code en ligne de commande (DOS)
- **En utilisant un environnement de développement (IDE)**
  - Eclipse (<http://www.eclipse.org>)
  - Netbeans (<http://www.netbeans.com>)
  - Borland JBuilder (<http://www.borland.com/jbuilder>)
  - IBM WebSphere Studio (<http://www.ibm.com/software/awdtools>)
  - Sun ONE Studio (<http://www.sun.com/software/sundev>)
  - Microsoft .Net Studio (<http://msdn.microsoft.com/vstudio>)

# Une première application

## Application versus Applet

- Une application Java

- est composée d'une classe possédant une méthode main() :

```
public static void main (String[] args){
```

```
    //code à exécuter pour initialiser l'application
```

```
}
```

- L'environnement d'exécution dépend de l'OS de la machine
  - Pas de restriction au niveau des API

- Une applet Java

- Comprend une classe publique dérivant de java.applet.Applet
  - L'environnement d'exécution dépend du browser Web
  - Restrictions au niveau des API
    - Généralement pas autorisée à lire ou écrire sur des fichiers locaux.
    - Interdiction d'ouvrir des connections réseaux vers d'autres systèmes que la machine hôte qui a chargé l'applet
    - ...

# Une première application

## Application HelloWorld

- Créer un fichier texte : HelloWorld.java
- Règle de bonne pratique : 1 classe par fichier et 1 fichier par classe

```
public class HelloWorld
{
    public static void main (String[]args)
    {
        System.out.println("Hello the World");
    }
}
```

La première ligne du programme doit être la déclaration de la classe

Tout programme doit contenir une méthode **main** qui porte la signature ci-contre

Écrire à l'écran "Hello the World"

Fermer les accolades

- Compiler le programme : javac HelloWorld.java
- Le compilateur génère le bytecode dans le fichier : HelloWorld.class
- Exécuter l'application : java HelloWorld
- « Hello the World » s'affiche à l'écran

# Une première application

## Applet HelloWorldApplet (1/2)

- Créer un fichier texte : HelloWorldApplet.java
- Règle de bonne pratique: 1 classe par fichier et 1 fichier par classe

```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HelloWorldApplet extends Applet  
{  
    public void paint(Graphics g)  
    {  
        g.drawString("Hello the World", 50, 25);  
    }  
}
```

Importation des classes externes  
nécessaires (issues des API Java)

Déclaration de la classe qui hérite de la  
classe prédéfinie « Applet »

La méthode paint détermine l'affichage dans  
la fenêtre de l'Applet

Écrire à l'écran "Hello the World"

Fermer les accolades

- Compiler le programme : `javac HelloWorldApplet.java`
- Le compilateur génère le bytecode dans le fichier :  
`HelloWorldApplet.class`

# Une première application

## Applet HelloWorldApplet (2/2)

- Les applets s'exécutent dans une page HTML
- Pour intégrer l'applet dans une page HTML, il suffit d'utiliser la balise <APPLET>
- Le paramètre « CODE » de la balise <APPLET> indique le nom de la classe principale de l'applet

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY> Here is the output of my program:
<APPLET CODE="HelloWorldApplet.class" WIDTH=150 HEIGHT=75>
</APPLET>
</BODY>
</HTML>
```

- Ouvrir la page HTML dans un navigateur, l'applet se lance automatiquement au sein de la page



# Les utilitaires de Java

- **javac**
  - Compilateur, traduit fichier source .java en fichier bytecode .class
- **java**
  - Interpréteur java, lance des programmes
- **javadoc**
  - Générateur de documentation d'API
- **jar**
  - Utilitaire d'archivage et de compression

# Les utilitaires de Java

## Javac et Java

- **Javac**

- Compile un fichier source .java ou un package entier
- Exemples:
  - `javac MyBankAccount.java`  
compile le fichier mentionné, qui doit se trouver dans le package par défaut
  - `javac be\newco\*.java -d c:\classes`  
compile tout le package `be.newco` et génère du code compilé dans `c:\classes`, qui doit exister

- **Java**

- Lance un programme principal
- Exemples:
  - `java bankStream.MyProgram`  
Lance le programme spécifié par la méthode `public static void main(String[] args)` dans la classe `MyProgram` qui se trouve dans le package `bankStream`.
- Possibilité de spécifier un `classpath`: un chemin de recherche où `java` est censé de trouver ses classes

# Les utilitaires de Java

## Javadoc – Générateur de documents

- **Nécessité d'une documentation suffisante**
  - pour aider les membres de l'équipe
  - pour s'aider soi-même
  - javadoc: une partie de votre documentation
- **Intégrer code et documentation**
  - résoudre problème de maintenance de la documentation
  - informations dans le code lui-même
- **Lancer Javadoc**
  - Se mettre dans le répertoire parent de vos packages
  - Pour créer la javadoc, taper `javadoc -d c:\mydir\html demo`
  - Conditions
    - javadoc dans le PATH
    - répertoire destination (-d) doit exister
    - « demo » est le nom d'un package
    - Commentaires délimités par `/**` et `*/`

# Les utilitaires de Java

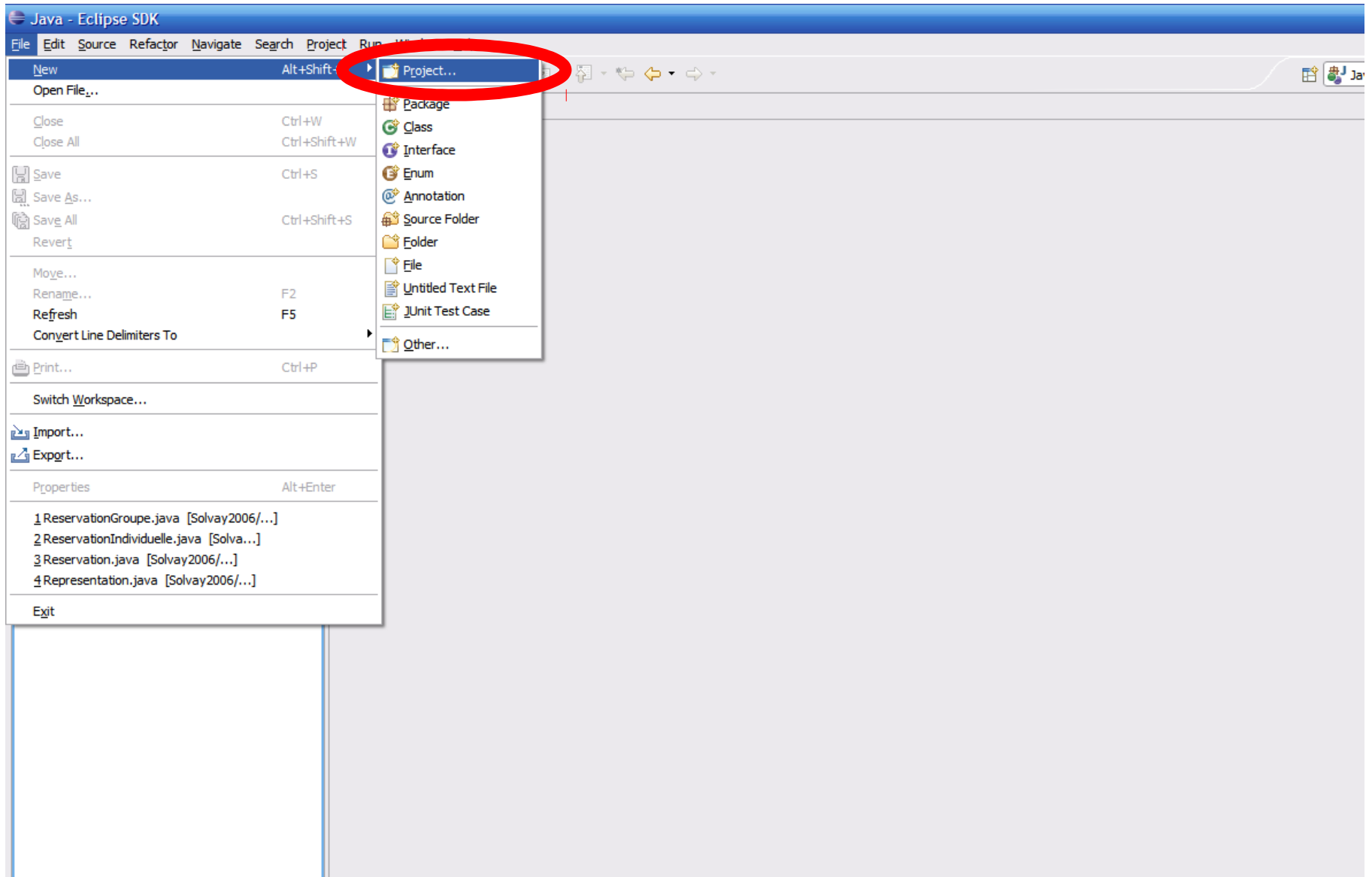
## Jar – Utilitaire d'archivage

- Permet de grouper et compresser des fichiers utilisés par un programme Java
- **Syntaxe d'utilisation similaire à tar**
  - `jar cf myjarfile.jar *.class`  
archivage de tout fichier .class, trouvé dans le répertoire courant et tout sous-répertoire, dans le fichier myjarfile.jar
  - `jar xf myjarfile.jar`  
Extrait tout fichier contenu dans myjarfile.jar vers une structure de répertoires
- **l'interpréteur java reconnaît les fichiers .jar et peut les traiter comme un répertoire.**
  - `java -cp myarchive.jar be.newco.MyMain`  
Lance le `main()` contenu dans `be.newco.MyMain`, tout en ajoutant les fichiers de `myarchive.jar` dans le `classpath`

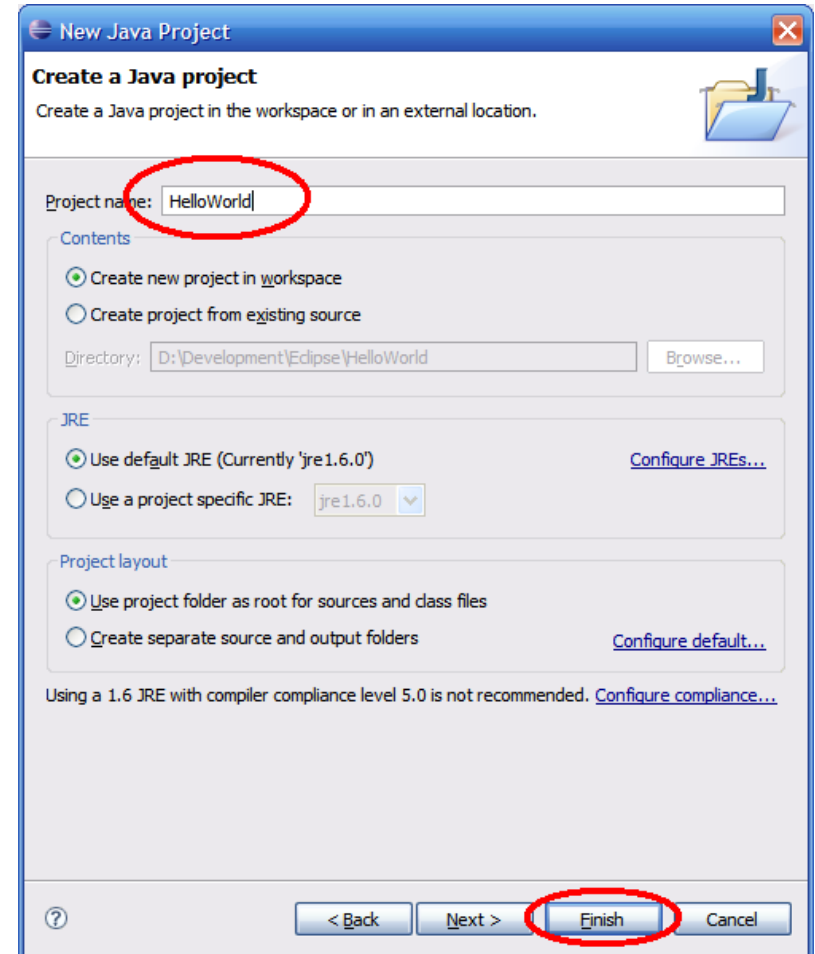
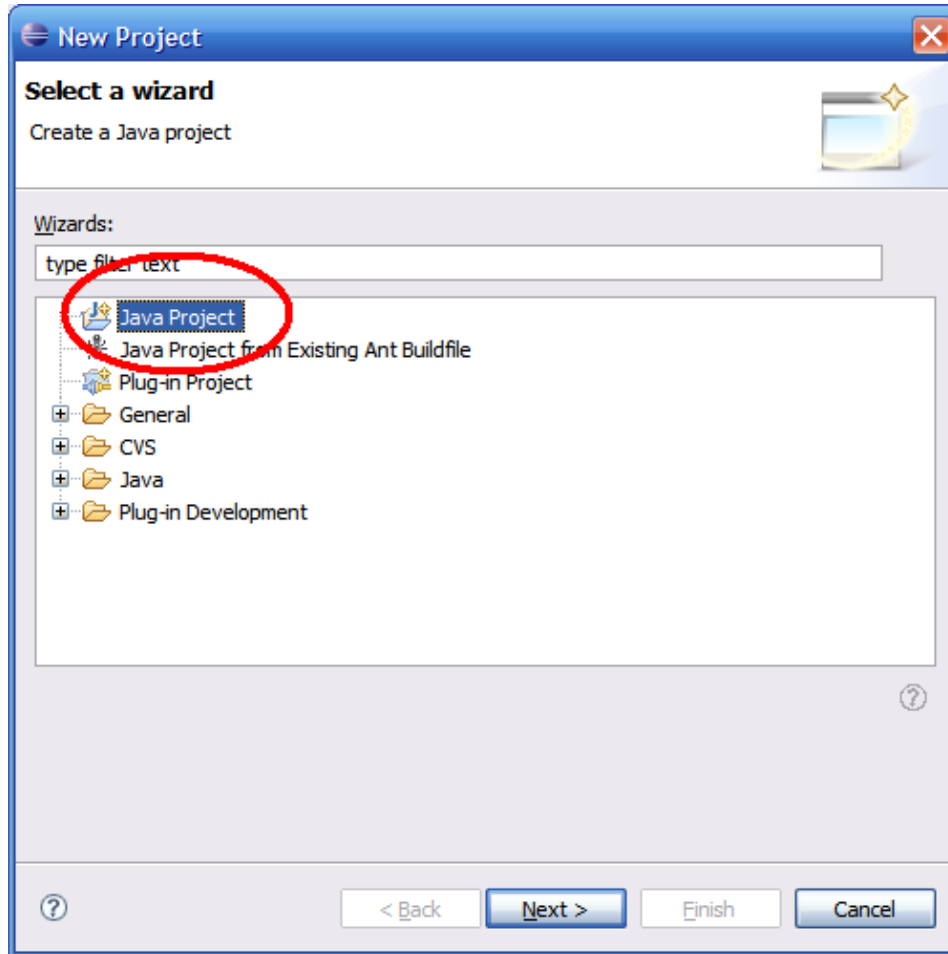
# L'environnement Eclipse

- **Eclipse est un Environnement de Développement Intégré (IDE)**
- **Spécialement conçu pour le développement en Java**
- **Créé à l'origine par IBM**
- **Puis cédé à la communauté Open Source**
- **Caractéristiques principales**
  - Notion de « projet » (1 programme → 1 projet)
  - Colore le code en fonction de la signification des mots utilisés
  - Force l'indentation du code
  - Compile le code en temps réel  
→ Identifie les erreurs en cours de frappe
  - Peut générer des bouts de code automatiquement
  - Permet de gérer le lancement des applications

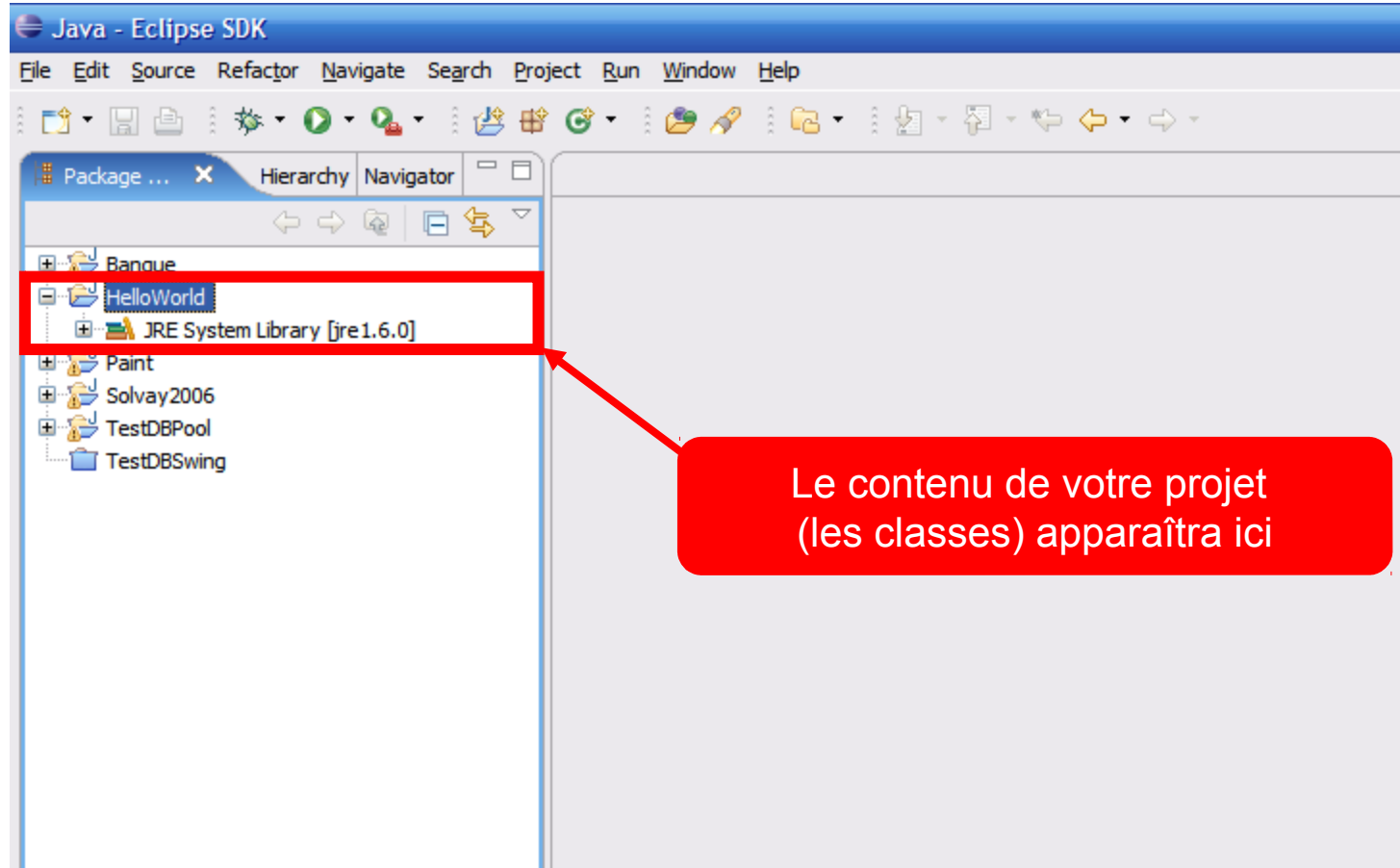
# Créer un projet Eclipse – Etape 1



# Créer un projet Eclipse – Etape 2



# Créer un projet Eclipse – Etape 3

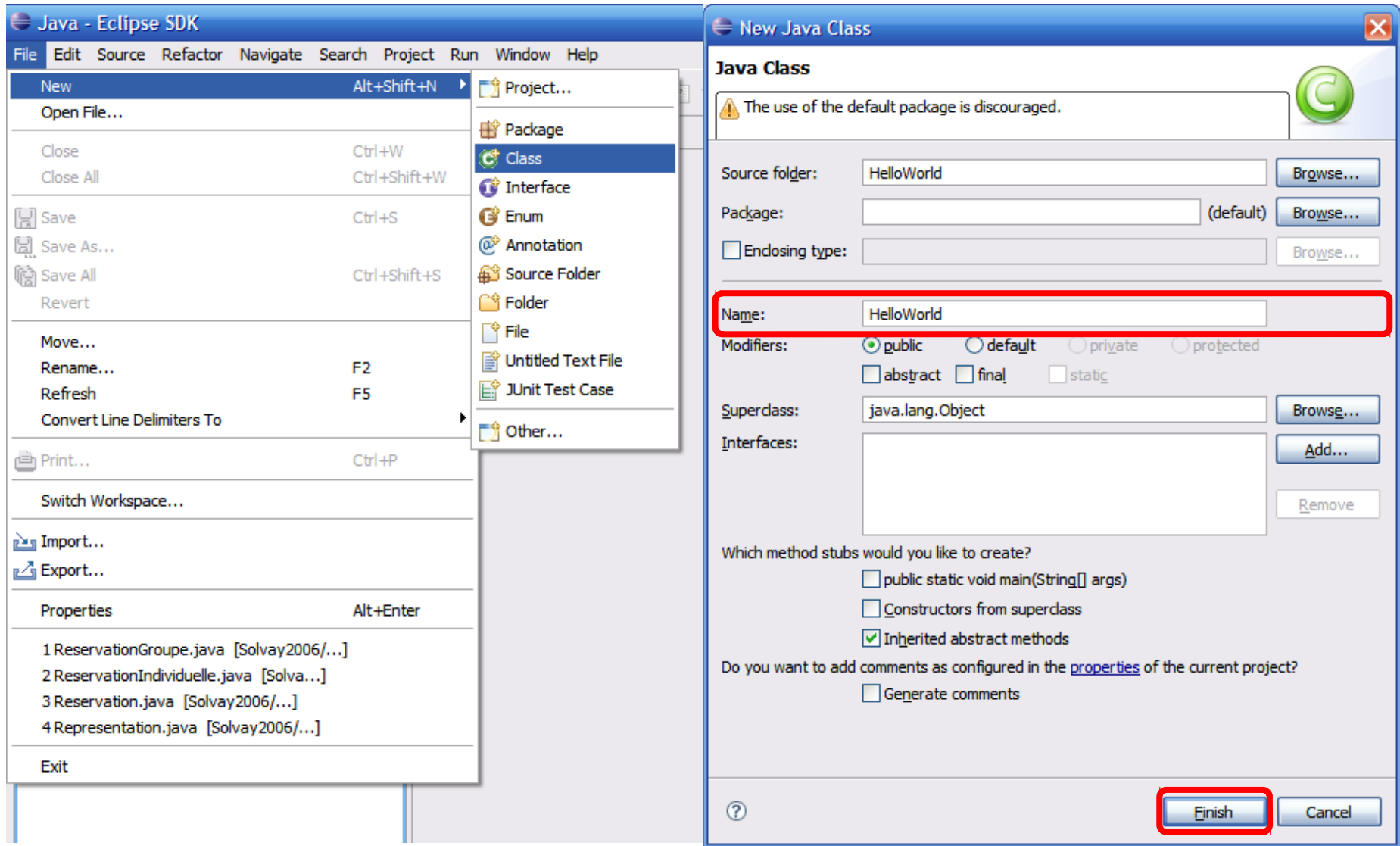




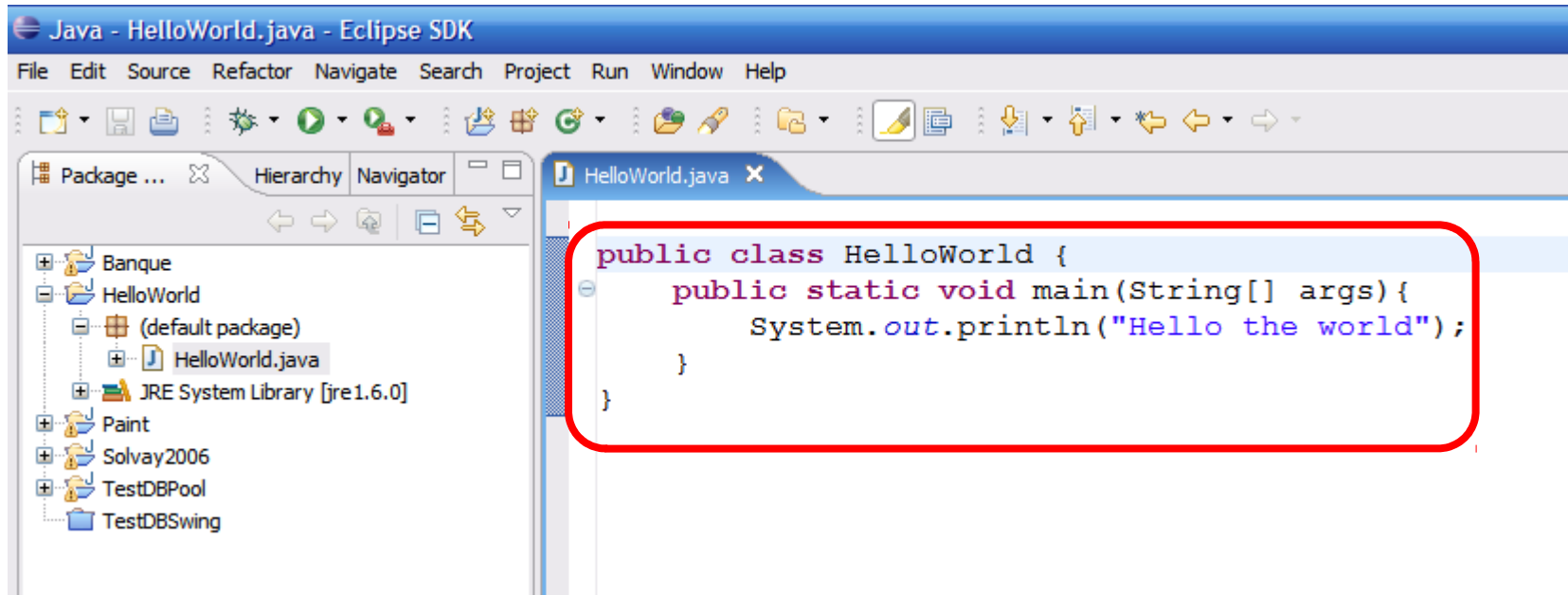
# Une première application en Java

- **Maintenant que notre projet a été créé, nous pouvons commencer à développer une application**
- **Une application Java est composée de « Classes »**
  - En règle générale, chaque classe correspond à un fichier
  - Chaque fichier « source » (le code de chaque classe) est sauvé avec un nom de fichier correspondant au nom de la classe et l'extension « .java »
  - Java est dit « case-sensitive » → **Distingue majuscules et minuscules!!!**
- **Notre première application sera composée d'une seule classe**
  - Le nom de cette classe sera « HelloWorld »
  - Elle sera donc enregistrée dans un fichier nommé « HelloWorld.java »
  - Le code de cette classe (fourni plus loin) doit être recopié tel quel
  - **ATTENTION**
    - Chaque symbole importe
    - Une majuscule n'est pas une minuscule

# Une première application en Java



# Une première application en Java



```
public class HelloWorld  
{  
    public static void main (String[]args)  
    {  
        System.out.println("Hello the World");  
    }  
}
```

La première ligne du programme doit être la déclaration de la classe

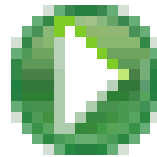
Tout programme doit contenir une méthode **main** qui porte la signature ci-contre

Écrire à l'écran "Hello the World"

Fermer les accolades

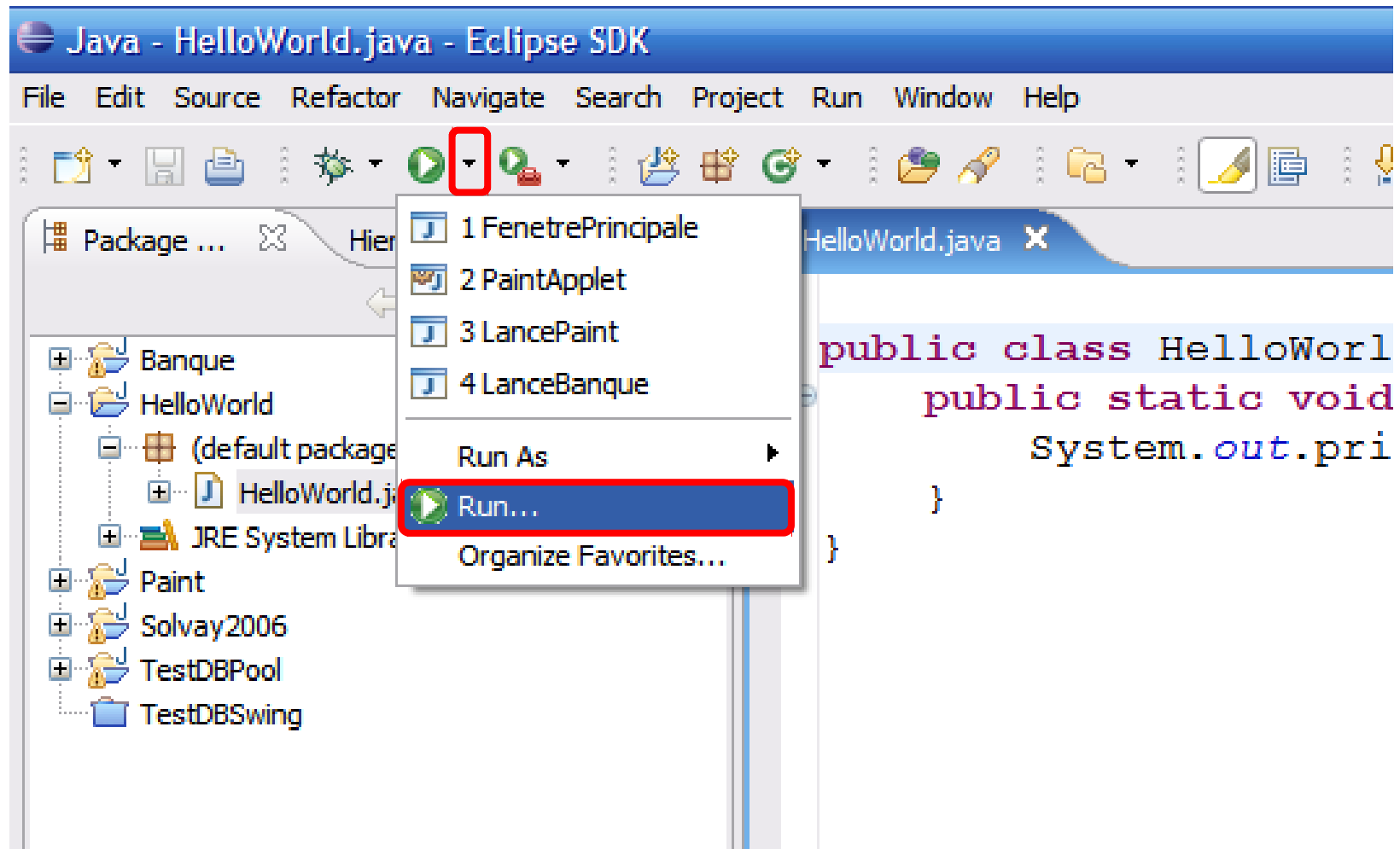
# Une première application en Java

- Une fois le programme écrit (ici l'unique classe), il reste à le lancer
- Pour lancer une application Java, il faut
  - La compiler (*fait automatiquement par Eclipse*)
  - Lancer la machine virtuelle (JVM) (*fait automatiquement par Eclipse*)
  - Ordonner à la JVM d'appeler la méthode « main » de la classe principale
    - ➔ Créer une « configuration de lancement » dans Eclipse
    - ➔ Pour « apprendre » à Eclipse comment lancer notre programme
    - ➔ Une fois cette configuration créée, on pourra relancer le programme en cliquant simplement sur le bouton



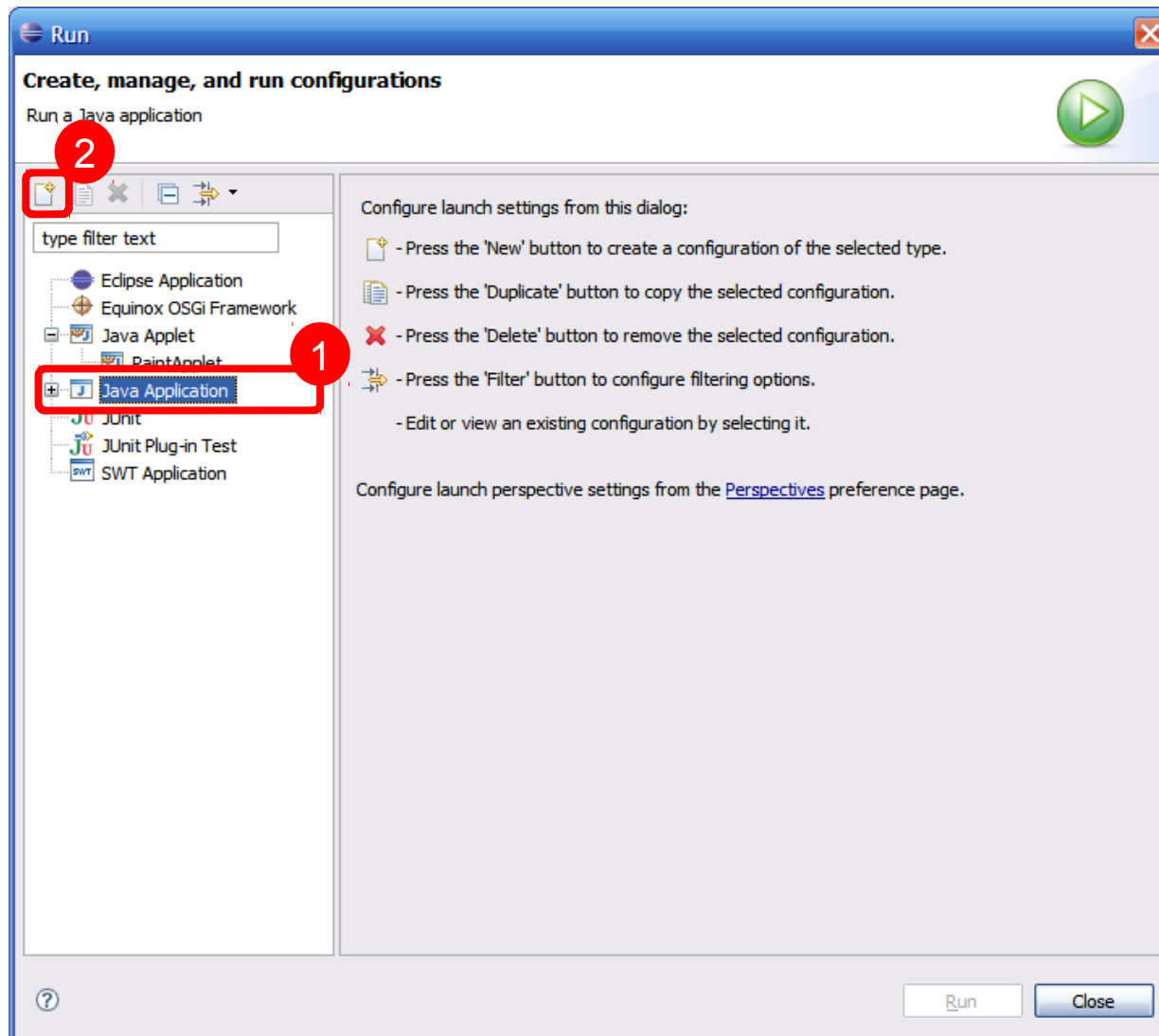
# Une première application en Java

## Créer une configuration de lancement Eclipse



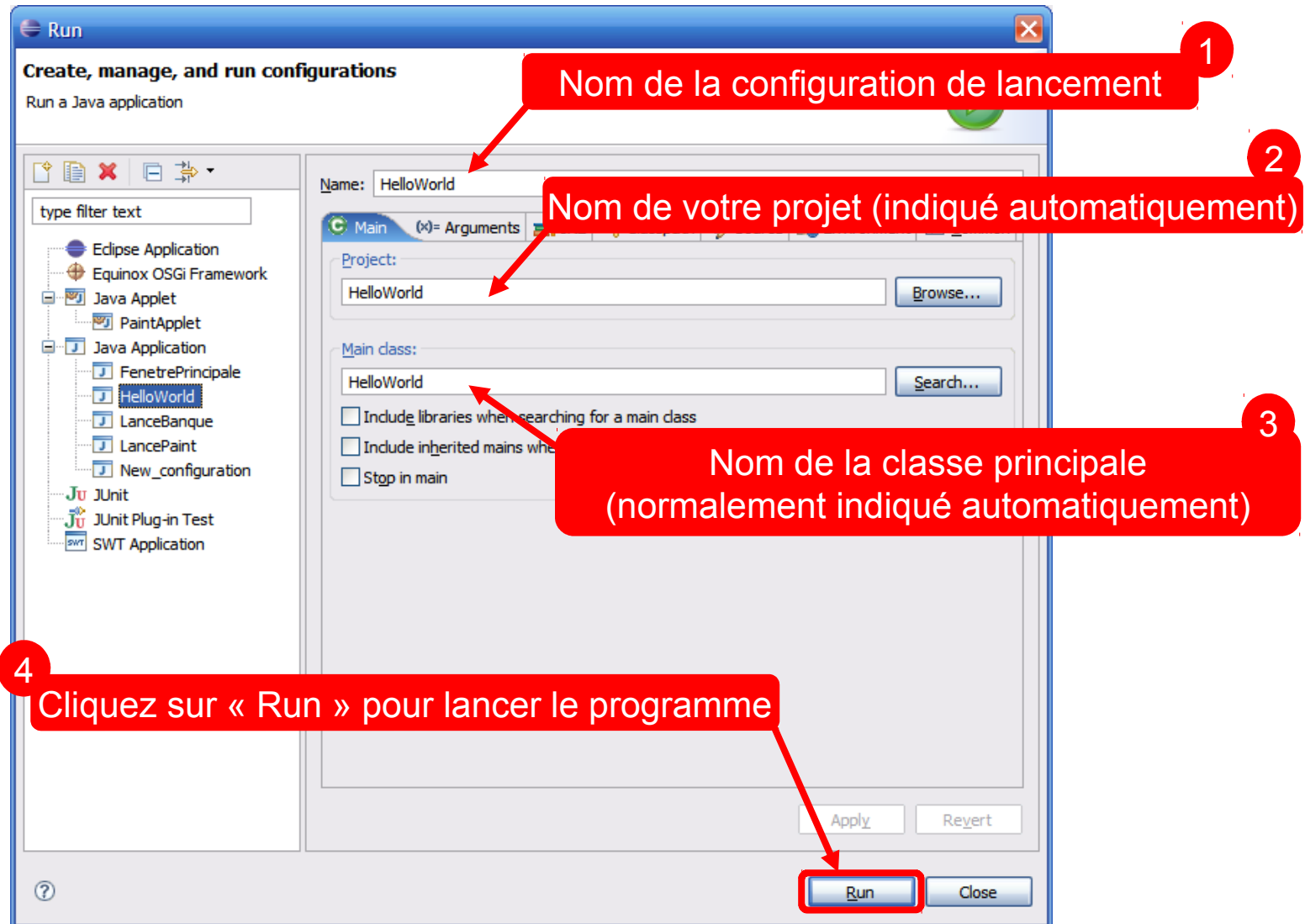
# Une première application en Java

## Créer une configuration de lancement Eclipse



# Une première application en Java

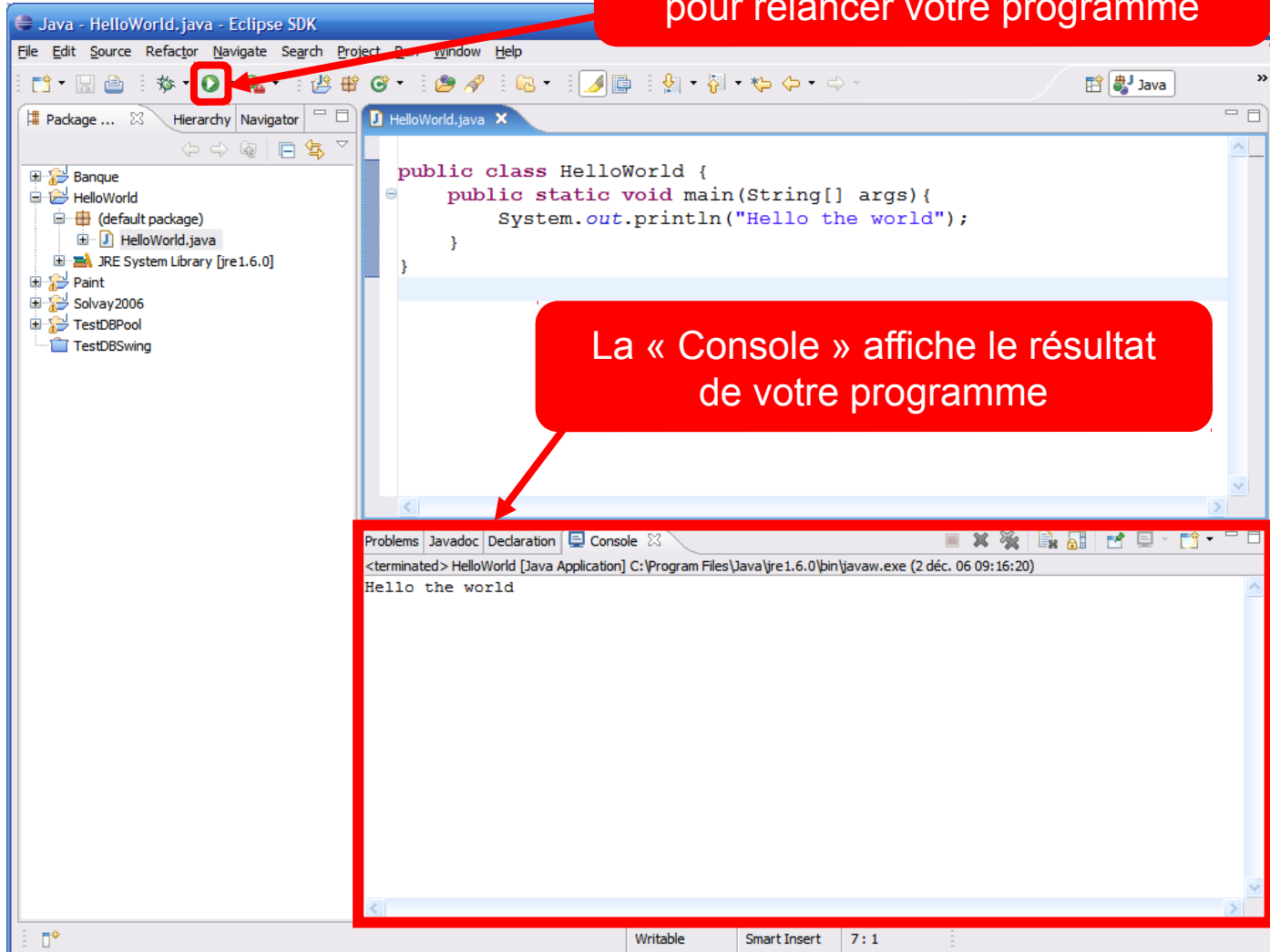
## Créer une configuration de lancement Eclipse



# Une première application en Java

## Le résultat de votre application

Cliquez sur le bouton « Run »  
pour relancer votre programme



La « Console » affiche le résultat  
de votre programme



# Introduction à Java

## III. Syntaxe du langage Java



# Survol du chapitre

- **Le « vocabulaire » de Java: Mots-clés et variables**
  - Mots-clé
  - Identificateurs: Quel nom donner à ses variables, méthodes et classes?
  - Les variables
    - Variables primitives
    - Création et utilisation
    - Chaînes de caractères
- **La « grammaire » de Java: Structures et règles**
  - Arithmétique et opérateurs
  - Instructions et blocs d'instructions
  - Structures de contrôle
    - If, then, else
    - For
    - While et Do... While
    - Break et Continue
- **Conventions**
  - Commentaires dans le code source
  - Conventions d'écriture
- **Trucs et astuces de base**
  - Méthodes essentielles
  - Les tableaux (« Array »)

# Syntaxe Java

- **Un langage informatique est composé de**
  - **Mots-clés**
    - ➔ Constituent le vocabulaire du langage
  - **Structures et règles**
    - ➔ La « grammaire » du langage (=la forme requise des instructions)
  - **Conventions**
    - ➔ En général des règles de notations adoptées par tous les programmeurs

# Vocabulaire Java

## Mots-clé: Le « vocabulaire » de Java

<code>abstract</code>	<code>else</code>	<code>interface</code>	<code>switch</code>
<code>assert</code>	<code>enum</code>	<code>long</code>	<code>synchronized</code>
<code>boolean</code>	<code>extends</code>	<code>native</code>	<code>this</code>
<code>break</code>	<code>false</code>	<code>new</code>	<code>throw</code>
<code>byte</code>	<code>final</code>	<code>null</code>	<code>throws</code>
<code>case</code>	<code>finally</code>	<code>package</code>	<code>transient</code>
<code>catch</code>	<code>float</code>	<code>private</code>	<code>true</code>
<code>char</code>	<code>for</code>	<code>protected</code>	<code>try</code>
<code>class</code>	<code>goto</code>	<code>public</code>	<code>void</code>
<code>const</code>	<code>if</code>	<code>return</code>	<code>volatile</code>
<code>continue</code>	<code>implements</code>	<code>short</code>	<code>while</code>
<code>default</code>	<code>import</code>	<code>static</code>	
<code>do</code>	<code>instanceof</code>	<code>strictfp</code>	
<code>double</code>	<code>int</code>	<code>super</code>	

# Vocabulaire Java

## Identificateurs

- En informatique, on définit des variables, des classes et des fonctions (ou méthodes)
- Un identificateur (*identifier*) permet de désigner une classe, une méthode, une variable, c'est le nom que vous choisissiez de leur donner
- On ne peut choisir n'importe quel nom. En Java en particulier:
  - Interdiction d'utiliser les mots-clés
  - Ne peuvent pas contenir:
    - D'espaces
    - De caractères internationaux (accents, etc.) (techniquement possible mais déconseillé)
  - Commencent par:
    - Une lettre
    - Un « \$ »
    - Un « \_ » (underscore)
  - Ne commencent pas par:
    - Un chiffre
    - Un signe de ponctuation autre que « \$ » ou « \_ »

# Vocabulaire Java

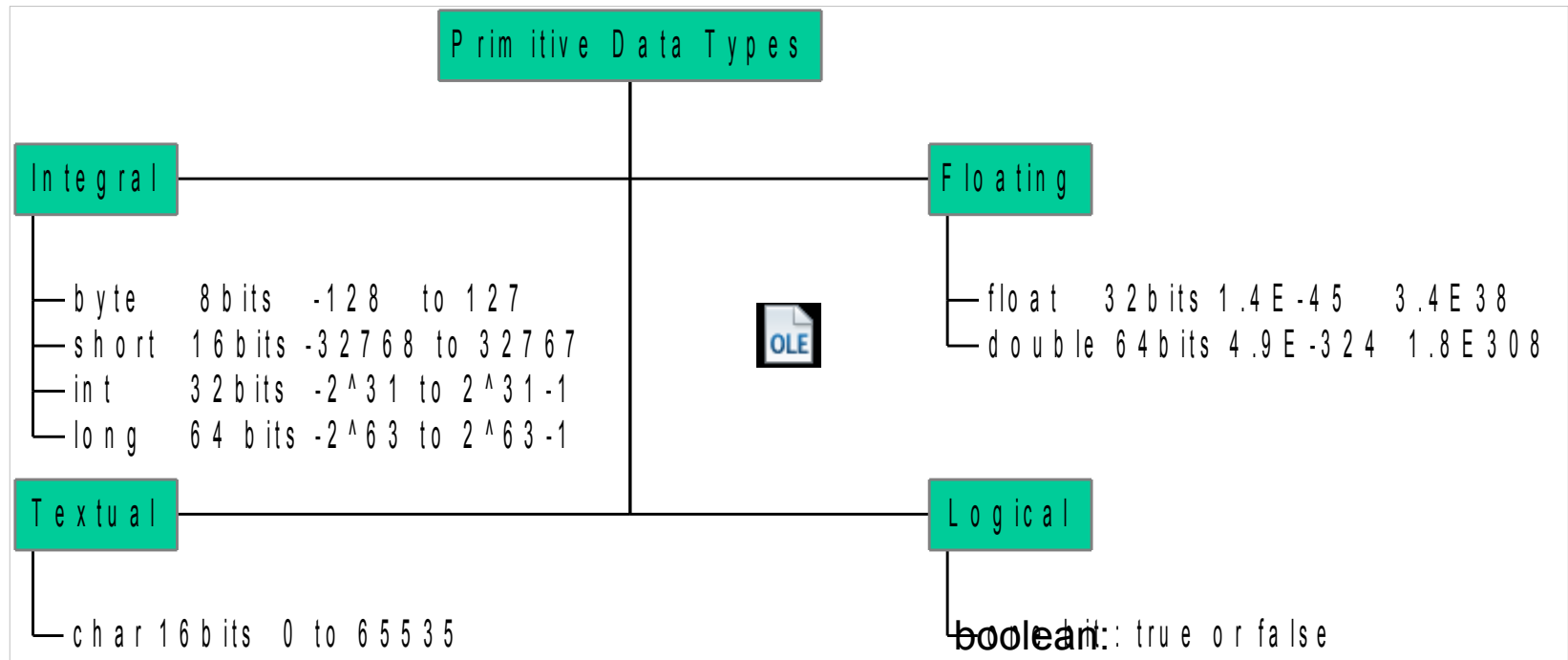
## Les variables

- Une variable est un endroit de la mémoire à laquelle on a donné un nom de sorte que l'on puisse y faire facilement référence dans le programme
- Une variable a une valeur, correspondant à un certain type
- La valeur d'une variable peut changer au cours de l'exécution du programme
- Une variable Java est conçue pour un type particulier de donnée

# Vocabulaire Java

## Types de variables en Java: Types primitifs (1/4)

- Les types de données primitifs en Java sont prédéfinis et en nombre limité:



# Vocabulaire Java

## Types de variables en Java: Types primitifs (2/4)

- **Explication:**

- byte : codé sur 8 bits → 28 valeurs →  $(-2^7)$  to  $(2^7-1)$  = -128 à 127
- int : codé sur 32 bits → 232 valeurs →  $(-2^{31})$  to  $(2^{31}-1)$

- **Déclaration et initialisation :**

- `int int x=12;`
- `short short x= 32; (short x=33000; // Hors limite)`
- `long long x= 200L; // Nombre accolé à un L`
- `byte byte x=012; // Nombre commençant avec un 0`
- `double double x=23.2323;`
- `float float x= 23.233F; // Nombre accolé à un F`
- `char char c='a'; char c='\u0061'; char c=(char) 97;`
- `boolean boolean b=true;`



# Vocabulaire Java

## Types de variables en Java: Types primitifs (3/4)

- **Pour pouvoir être utilisée, une variable en Java doit être**
  - Déclarée (définir son nom et son type)
  - Initialisée (lui donner une valeur initiale)
    - ➔ Peut se faire en même temps que la déclaration
  - Assignée (modifier sa valeur au cours de son cycle de vie)
- **Syntaxe:**
  - `int t;` Déclaration d'un entier t (t est l'identificateur)
  - `int u = 3;` Déclaration et initialisation d'un entier u
  - `t=7;` Initialisation de t à la valeur 7
  - `u=t;` Assignment (affectation) de la valeur de t à u
  - `m=9;` **ERREUR**: « m » n'a pas été déclaré
  - `char c;` Déclaration
  - `c='a';` Initialisation

# Vocabulaire Java

## Types de variables en Java: Types primitifs (4/4)

### Fonctionnement:

```
int a = 5;
```

```
int b = 8;
```

Déclaration et initialisation de 2 entiers: a et b

```
a=b;
```

Affectation de la valeur de b à a



Désormais, il existe deux variables en mémoire qui ont la même valeur

# Vocabulaire Java

## Les chaînes de caractères

- Les chaînes de caractères (« String ») sont essentielles et omniprésentes dans les programmes informatiques
- Or il n'existe pas de type primitif « string » en Java
- String n'est en fait pas un type primitif, c'est une classe (cf. plus loin)
- Leur utilisation ressemble néanmoins très fort à celle des autres types:
  - Déclaration de deux String:  

```
String s1, s2; // On peut toujours déclarer plusieurs variables de même type  
              // simultanément en les séparant par des virgules
```
  - Initialisation :  

```
s1 = "Hello";  
  
s2 = "le monde";
```
  - Déclaration et initialisation :  

```
String s3 = "Hello";
```
  - Concaténation :  

```
String s4 = s1 + " " + s2;
```

# Grammaire Java

## Arithmétique et opérateurs: opérateurs arithmétiques

- Quelle est la valeur de :  $5+3*4+(12/4+1)$
- Règles de précédences sur les opérateurs:

Niveau	Symbole	Signification
1	()	Parenthèse
	*	Produit
2	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

# Grammaire Java

## Arithmétique et opérateurs – Opérateurs de comparaison et logiques

- Pour comparer deux valeurs:

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

- Opérateurs logiques:

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
	expr1    expr2	Expr1 ou expr2, ou les deux sont vraies
	expr1   expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse
!=	expr1 != expr2	si expr1 est différent de expr2

# Grammaire Java

## Arithmétique et opérateurs – Opérateurs d'assignation

- L'opérateur de base est '='
  - Exemple: `int a = 5;`
- Il existe des opérateurs d'assignation qui réalisent à la fois
  - une opération arithmétique, logique, ou bit à bit
  - et l'assignation proprement dite

Opérateur	Exemple	Équivalent à
+=	<code>expr1 += expr2</code>	<code>expr1 = expr1 + expr2</code>
-=	<code>expr1 -= expr2</code>	<code>expr1 = expr1 – expr2</code>
*=	<code>expr1 *= expr2</code>	<code>expr1 = expr1 * expr2</code>
/=	<code>expr1 /= expr2</code>	<code>expr1 = expr1 / expr2</code>
%=	<code>expr1 %= expr2</code>	<code>expr1 = expr1 % expr2</code>
++	<code>expr1++</code>	<code>expr1 = expr1 + 1</code>
--	<code>expr1--</code>	<code>expr1 = expr1 - 1</code>

# Grammaire Java

## Instructions et blocs d'instruction

- **Une instruction**

- Réalise un traitement particulier:
- Renvoie éventuellement le résultat du calcul
- Est comparable à une phrase du langage naturel
- Constitue l'unité d'exécution
- Est (presque) toujours suivie de « ; »
- Il en existe plusieurs types: déclaration, assignation, envoi de message, etc.

- **Un bloc**

- Est une suite d'instructions entre accolades « { » et « } »
- Délimite la portée des variables qui y sont déclarées
- Suit (presque) toujours la déclaration des classes et méthodes
- Définit également le contenu des boucles et structures conditionnelles
- Doit toujours être refermé (autant de « { » que de « } »)

# Grammaire Java

## Instructions et blocs d'instruction

- **Types d'instructions:**

- **Déclaration**

- Définit un élément (variable, méthode ou classe)
    - Constitue la signature de cet élément
    - S'il s'agit d'une déclaration de variable:
      - Est normalement toujours suivie d'un « ; »
      - Ex: `int unNombreEntier;`
    - S'il s'agit d'une déclaration de classe ou de méthode:
      - Est normalement toujours suivie d'un bloc d'instructions
      - Ex: `int uneMethodeQuiRenvoieUnEntier() {...}`
      - On définit alors dans le bloc les traitements (instructions) à réaliser par la méthode

- **Assignation**

- Sert à modifier la valeur d'une variable
    - Est toujours suivie d'un « ; »
    - Ex: `a = 57;`

- **Envoi de message**

- Sert à appeler une méthode (lancer un traitement)
    - Est toujours suivie d'un « ; »
    - Ex: `monChien.vaChercher(leBaton);`



# Grammaire Java

## Structures de contrôle

- Les structures de contrôles permettent d'arrêter l'exécution linéaire des instructions (de bas en haut et de gauche à droite)
- Il en existe 4 grands types:
  - Les conditionnelles (« SI ... ALORS ... SINON ... »)
  - Les boucles (« TANT QUE ... JE REFAIS ... »)
  - Les branchements (« JE SORS DE LA BOUCLE ET JE VAIS A ... »)
  - Le traitement d'exceptions (« J'ESSAIE ... MAIS SI CA PLANTE ... »)

Type d'instruction	Mots clés associés
Conditionnelle	if() else – switch() case
Boucle	for( ; ; ) – while () – do while()
Branchement	label : -- break – continue -- return
Traitement d'exceptions	try catch finally – throw

# Grammaire Java

## Structures de contrôle

- Les conditionnelles (traditionnellement « IF – THEN »)

```
if (LA_CONDITION_A_EVALUER)
{
    //instructions à réaliser si la condition est rencontrée
}
```

- Les conditionnelles (traditionnellement « IF – THEN – ELSE »)

```
if (LA_CONDITION_A_EVALUER)
{
    //instructions à réaliser si la condition est rencontrée
}
else
{
    //instructions à réaliser si elle n'est pas rencontrée
    //Peut contenir elle-même d'autres sous-conditions
}
```

# Grammaire Java

## Structures de contrôle

- Une autre forme de conditionnelle: **SWITCH – CASE**
  - Permet d'évaluer une variable numérique entière
  - Et de provoquer des traitements différents selon sa valeur

```
switch (UNE_VARIABLE_NUMERIQUE_ENTIERE)
{
    case 1 : instructions; // A réaliser si elle vaut 1
    case 2 : instructions; break; // Si elle vaut 2
    default : instructions; // Dans tous les autres cas
}
```

# Grammaire Java

## Structures de contrôle

- Les boucles FOR (boucles traditionnelles)

```
for (initialisation; condition; mise à jour de valeurs) {  
    // instructions  
}
```

- **Initialisation**: à exécuter lorsque le programme rentre pour la première fois dans la boucle → Sert à définir le compteur de la boucle
- **Condition**: à remplir pour recommencer le bloc d'instructions
- **Mise à jour**: instruction exécutée chaque fois que la boucle est terminée

### Exemple:

```
for (int i=0 ; i<10 ; i++) {  
    System.out.println("The value of i is : " + i);  
}
```

# Grammaire Java

## Structures de contrôle

- Un autre forme de boucle: WHILE – DO WHILE

```
while (CONDITION_A_RENCONTRER_POUR_CONTINUER) {  
    // Instructions à réaliser tant que la condition est  
    // rencontrée. A la fin de chaque itération, on  
    // réévalue la condition et on décide soit de rester  
    // dans la boucle avec une nouvelle itération, soit de sortir  
}
```

### Exemple:

```
int i = 0;  
while(i<10) {  
    System.out.println("The value of i is : " + i);  
    i++;  
}
```

# Grammaire Java

## Structures de contrôle

- **Interrompre une boucle une fois lancée: BREAK / CONTINUE**

- BREAK: achève immédiatement la boucle ou la conditionnelle
- CONTINUE: ignore le reste des instructions et passe tout de suite à l'itération suivante (au début de la boucle)

```
for (int i=0; i<10 ;i++){  
    if (i==5) continue; // Si i=5, on passe à 6 sans afficher  
    if (i==7) break;    // Si i=7, on sort de la boucle  
    System.out.println("The value of i is : " + i);  
}
```

# Instructions et structures de contrôle

## Structures de contrôle

**BREAK [LABEL]**

**CONTINUE [LABEL]**

```
outer:
for (int i=0 ; i<10 ; i++) {
    for(int j=20;j>4;j--){
        if (i==5) continue; // if i=5, you jump to the beginning of the loop
        if (i==7) break outer; // if i=7, you jump outside the loop and continue
        System.out.println("The value of i,j is :"+i+", "+j);
    }
}
```

# Grammaire Java

## Structures de contrôle

- **Intercepter les exceptions (« plantages »): TRY – CATCH**

- Certains traitements sont parfois « à risque », c'est-à-dire que le programmeur ne peut savoir avec certitude à l'avance si l'instruction qu'il écrit fonctionnera correctement ou non au moment de l'exécution car cela dépendra de paramètres dont il n'a pas le contrôle
- Les blocs TRY – CATCH (littéralement « ESSAIE [ceci] ET SI CELA PROVOQUE UNE ERREUR FAIS [cela] ») permettent de « prévoir le coup » en interceptant l'erreur qui pourrait se produire et en prévoyant d'emblée une action correctrice

```
try{  
    // Une ou plusieurs instructions potentiellement « à risque »  
} catch(ExceptionXYZ unProblemeDeTypeXYZ) {  
    // Ce qu'il faut faire si un problème de type XYZ se produit  
}
```



# Grammaire Java

## Portée des variables

- **Portée d'une variable et des attributs**

- Portée = Section du programme dans laquelle une variable existe
- La variable ne peut donc pas être utilisée en dehors de cette section
- La portée est définie par les accolades qui l'entourent directement
- Exemple:

```
if(solde < 0){  
    String avertissement = "Attention, solde négatif !";  
}  
else{  
    String avertissement = "Tutto va bene !";  
}  
System.out.println(avertissement);  
  
// Une erreur apparaîtra dès la compilation, car la variable  
// « avertissement » n'existe pas en dehors du bloc IF
```

- **Avantages**

- Rend les programmes plus faciles à corriger
- Limite le risque d'erreurs liées au réemploi d'un nom pour différentes variables

# Grammaire Java

## Structure d'un programme

- Un programme simple en Java doit avoir la structure suivante:

```
public class NomDeLaClasse {  
    public static void main(String[] args) {  
        // Le programme commence ici  
        // Tout se passe à l'intérieur de ces accolades  
    }  
}
```

# Conventions

## Règles de notation pour les identificateurs

- Java est « case sensitive », il est donc crucial d'adopter des règles de notation claires et définitives en termes de majuscules et minuscules.
- Les règles ci-dessous sont universellement appliquées par les développeurs Java et sont en particulier d'application dans toutes les classes prédéfinies

- Classes

- class BankAccount
- class RacingBike

- Interfaces

- interface Account

- Méthodes

- deposit()
- getName()

- Packages

- package coursSolvay2006;

- Variables

- int accountNumber

- Constantes

- MAXIMUM\_SIZE

# Conventions

## Commentaires dans le code source

- Il est hautement conseillé de « documenter » son code
- Cela se fait en ajoutant des commentaires précisant le rôle des instructions
- Les commentaires sont ignorés dans le code à la compilation
- Cela permet donc aussi de « neutraliser » une instruction sans pour autant l'effacer
- Pour inclure des commentaires dans le code:
  - Pour une seule ligne de commentaires:
    - 2 barres obliques « // » neutralisent tout le code à droite sur la même ligne
  - Pour annuler tout un bloc de lignes: Tout le texte entre « /\* » et « \*/ » est neutralisé

```
System.out.println("Hello"); // Affiche Hello à l'écran
```

```
public static void main(String[] args) {
```

```
/* La ligne qui suit sert à afficher un message à l'écran
```

```
Le message en question est « Hello »
```

```
Mais il peut être remplacé par n'importe quel autre */
```

```
System.out.println("Hello");
```

# Trucs et astuces de base

## Méthodes essentielles

- **Afficher quelque chose à l'écran:**

- `System.out.println("Texte ou données à afficher");`

- **Choisir un nombre aléatoire**

- `Math.random();` // Renvoie un **double** entre 0 et 1

- **Convertir un double en entier**

- `int i = (int) d;` // A supposer que **d** est un **double**

- **Convertir un entier en chaîne de caractères**

- `String s = ""+i;` // A supposer que **i** est un **int**

- **Obtenir le nombre entier contenu dans une chaîne de caractère:**

- `int i = Integer.parseInt(s);` // Si **s** est une chaîne de  
// caractères faite d'1 entier

- **Vérifier si une chaîne de caractères est égale à une autre**

- `s1.equals(s2)` // Renvoie **true** si **s1=s2** et **false** sinon
  - `s1.equalsIgnoreCase(s2)` // Idem sans prêter attention aux  
// majuscules/minuscules

# Trucs et astuces de base

## Méthodes essentielles

- Manipuler les dates

- La classe java.util.Date

- Contient en fait un entier long qui correspond à un moment dans le temps exprimé en millisecondes depuis le 1/1/1970 à 0:00:00-000

- Pour créer une date:

```
- Date d = new Date();  
  DateFormat df=DateFormat.getDateInstance(DateFormat.SHORT, Locale.FRANCE);  
  try {  
      d = df.parse("13/11/2005");  
  } catch (ParseException e) {e.printStackTrace();}
```

- Pour obtenir la date et l'heure du système:

```
- Date d = new Date(System.currentTimeMillis());
```

- Pour formater une date:

```
- String s = DateFormat.getDateInstance(DateFormat.SHORT).format(d);  
- String s = DateFormat.getDateInstance(DateFormat.LONG).format(d);  
- String s = DateFormat.getDateInstance(DateFormat.MEDIUM).format(d);
```

- Il faut pour cela importer:

```
- java.util.Date, java.util.Locale, java.text.DateFormat
```

# Trucs et astuces de base

## Méthodes essentielles

- Récupérer les entrées au clavier de l'utilisateur:

- Au début du programme (avant la déclaration de la classe):

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;
```

- Au moment d'inviter l'utilisateur à introduire quelque chose au clavier:

```
BufferedReader b = new BufferedReader(new InputStreamReader(System.in));  
String s = "";  
try {  
    s = b.readLine();  
} catch (Exception e) {System.exit(1);}  
n = Integer.parseInt(s);
```

# Exercices

- **Ding Ding Bottle**

- Ecrire un programme qui compte de 1 à 100
- Affiche ces nombres à l'écran
- Remplace les multiples de 5 par « Bottle »
- Remplace les multiples de 7 par « Ding Ding »
- Remplace les multiples de 5 et 7 par « Ding Ding Bottle »

- **Calcul de factorielle**

- Ecrire un programme qui calcule la factorielle de 15
- Et affiche le résultat à l'écran ainsi que tous les résultats intermédiaires

- **Trouve le nombre**

- Ecrire un programme qui choisit un nombre entier au hasard entre 1 et 10
- Laisse 3 chances à l'utilisateur pour le trouver (en l'indiquant au clavier)
- Lui demande ensuite s'il souhaite rejouer
- Eviter au maximum les redites (ne pas répéter le code)



# Exercices Bonus

- Ecrire un programme qui affiche sur la console un triangle de la forme suivante :

\*

\*\*\*

\*\*\*\*\*

- En paramètre, fournissez à l'application le nombre de lignes du triangle

# Exercices

- **Comment procéder? Pour chaque programme:**

- Créez une nouvelle classe dans votre projet « HelloWorld »
- Le code de base de la classe est le suivant:

```
public class NomDeLaClasse{  
  
    public static void main(String[] args){  
  
        // C'est ici que vous écrirez votre programme  
  
    }  
  
}
```

- Une fois votre programme écrit, créez une nouvelle configuration de lancement dans Eclipse et lancez-le pour en tester le fonctionnement

# Trucs et astuces de base

## Les tableaux ("Array") (1/3)

- Il arrive d'avoir à gérer un ensemble de variables qui forment un tout
- Un tableau est utilisé pour stocker une collection de variables de même type
- Il faut définir le type de variables que le tableau va contenir
- Les tableaux sont identifiés par des crochets [ ]
- Les « cases » du tableau sont identifiées par des indices dans les [ ]
- L'indexation des tableaux commence toujours à 0
- Un tableau doit être déclaré, initialisé, et rempli

```
int[] nombres;           // déclaration
nombres = new int[10];    // création
int[] nombres = new int[10]; // déclaration et création
nombres[0] = 28;          // Le 1er élément est 28
```

- La capacité du tableau est fournie par `nomDuTableau.length`

# Trucs et astuces de base

## Les tableaux (“Array”) (2/3)

- On peut construire des tableaux à plusieurs dimensions
- Des tableaux à plusieurs dimensions sont en fait des tableaux de tableaux

```
int[][] matrice = new int[3][];
```

```
matrice[0] = new int[4];
```

```
matrice[1] = new int[5];
```

```
matrice[2] = new int[3];
```

```
matrice[0][0] = 25;
```

« matrice » est une référence vers un tableau contenant lui-même 3 tableaux de taille non définie

Le premier élément de la matrice est une référence vers un tableau de 4 entiers,...

Le premier élément du premier tableau de la matrice est un entier de valeur 25

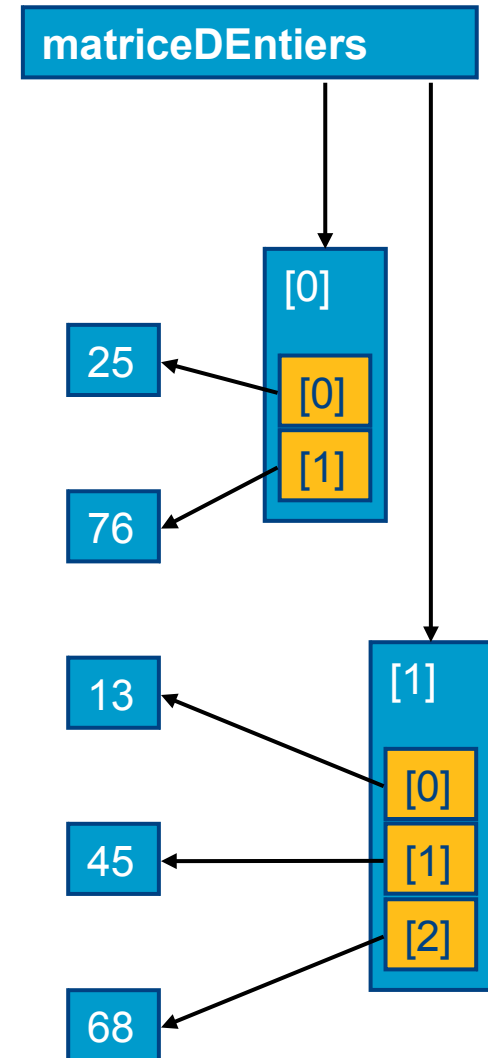
### Exemple:

- Créer et initialiser une matrice contenant deux tableaux de 2 et 3 entiers respectivement
- Remplir les 5 « cases » du tableau avec des valeurs entières

# Trucs et astuces de base

## Les tableaux (“Array”) (3/3)

```
int[][] matriceDEntiers;  
matriceDEntiers = new int[2][];  
matriceDEntiers[0] = new int[2];  
matriceDEntiers[1] = new int[3];  
matriceDEntiers[0][0] = 25;  
matriceDEntiers[0][1] = 76;  
matriceDEntiers[1][0] = 13;  
matriceDEntiers[1][1] = 45;  
matriceDEntiers[1][2] = 68;
```



# Exercices

- **Etudiants**

- Ecrire un programme permettant d'encoder les cotes d'étudiants à un examen
- Et fournissant quelques statistiques de base sur ces cotes
- Le programme demande d'abord à l'utilisateur combien de cotes il souhaite encoder (soit N)
- Ensuite il demande à l'utilisateur d'introduire une à une les notes des N étudiants (/20)
- Votre programme mémorise ces résultats
- Les affiche une fois la dernière cote encodée
- Et affiche quelques statistiques de base:
  - le nombre de réussites et d'échecs
  - la moyenne totale
  - la cote moyenne des échecs
  - la cote moyenne des succès

# Utilisation des tableaux de taille dynamique

- Les arrays ont pour défaut d'avoir une taille fixée à priori. Java fournit des collections plus souples et simples d'utilisation. La gestion de la taille est automatisée et allège la tâche du programmeur.
- Pour déclarer une variable de type tableau d'entiers :
- `ArrayList<Integer> monTableau = new ArrayList<Integer>();`
- On peut remplacer Integer par n'importe quelle "class"
- Pour ajouter une donnée au tableau :
  - `monTableau.add(3);`
- Pour récupérer un élément en position k :
  - `int elem = monTableau.get(k);`

# Utilisation des tableaux de taille dynamique

- **Parcours d'un tableau**

- **Approche classique :**

```
for(int i=0; i<monTableau.size(); i++)
```

```
{  
    int elemti = monTableau.get(i);  
    System.out.println(elemti);  
}
```

- **Approche “for each”**

```
for(int elemti : monTableau)
```

```
{  
    System.out.println(elemti);  
}
```

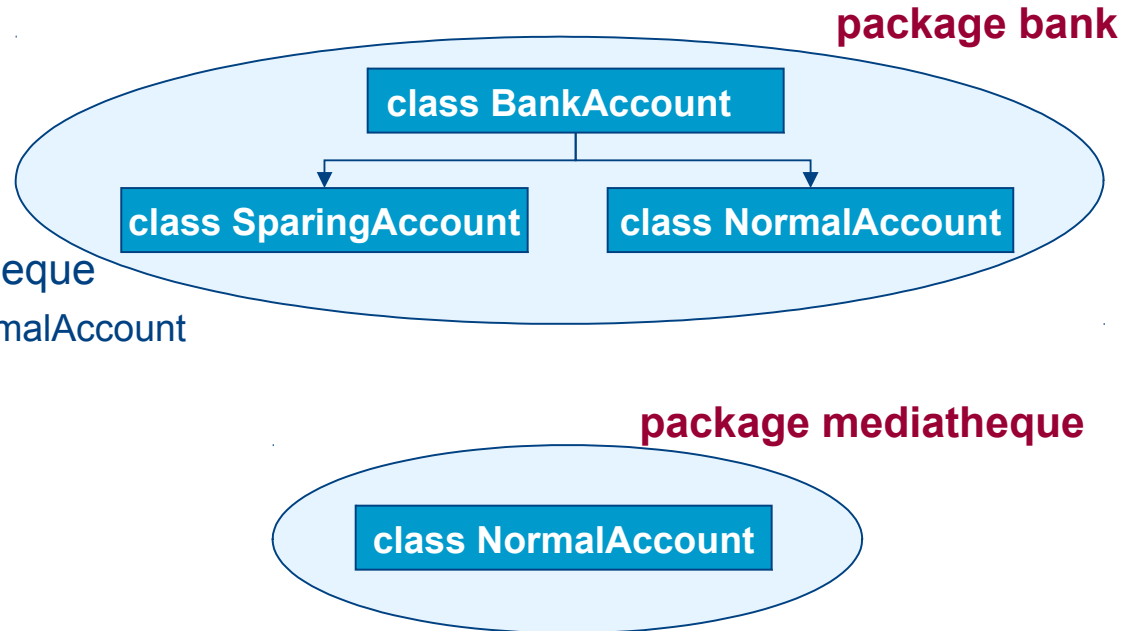


# Les packages et les importations

- Les packages offrent une organisation structurée des classes
- La répartition des packages correspond à l'organisation physique
- Les packages conditionnent les importations de classes
- La variable CLASSPATH indique le chemin d'accès aux packages
- Les packages permettent la coexistence de classes de même nom
- Les mots-clé associés sont « package » et « import »

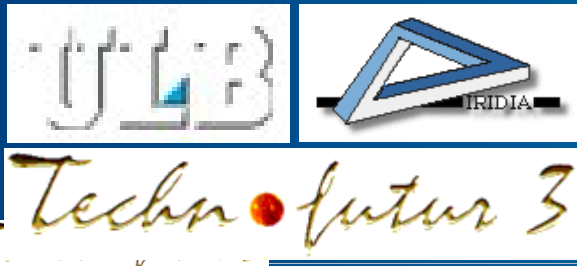
- **Exemple:**

- package technofutur3.bank
  - class NormalAccount
  - class SparingAccount
- package technofutur3.mediatheque
  - import technofutur3.bank.NormalAccount



# Introduction à Java

## IV. Programmation orientée objet en Java



# Les concepts de l'OO

- **Introduction à l'orienté objet (OO)**
  - Pourquoi l'OO?
  - Qu'est-ce que l'OO?
  - Qu'est-ce qu'un « objet »?
- **Les principes de l'orienté objet**
  - L'encapsulation
    - Pourquoi l'objet est-il une boîte noire?
  - La classe
    - Comment définir un objet?
    - Comment définir les états? → Déclaration des variables membres
    - Comment initialiser les états? → Le constructeur
    - Comment définir les comportements? → Déclaration des méthodes
  - Interaction entre objets
    - Envoi de messages
    - Permettre aux objets de s'envoyer des messages → Association
    - Création d'objets → Appeler le constructeur
  - Héritage
  - Modéliser les classes avec des diagrammes UML

# Les concepts de l'OO

## Pourquoi l'OO?

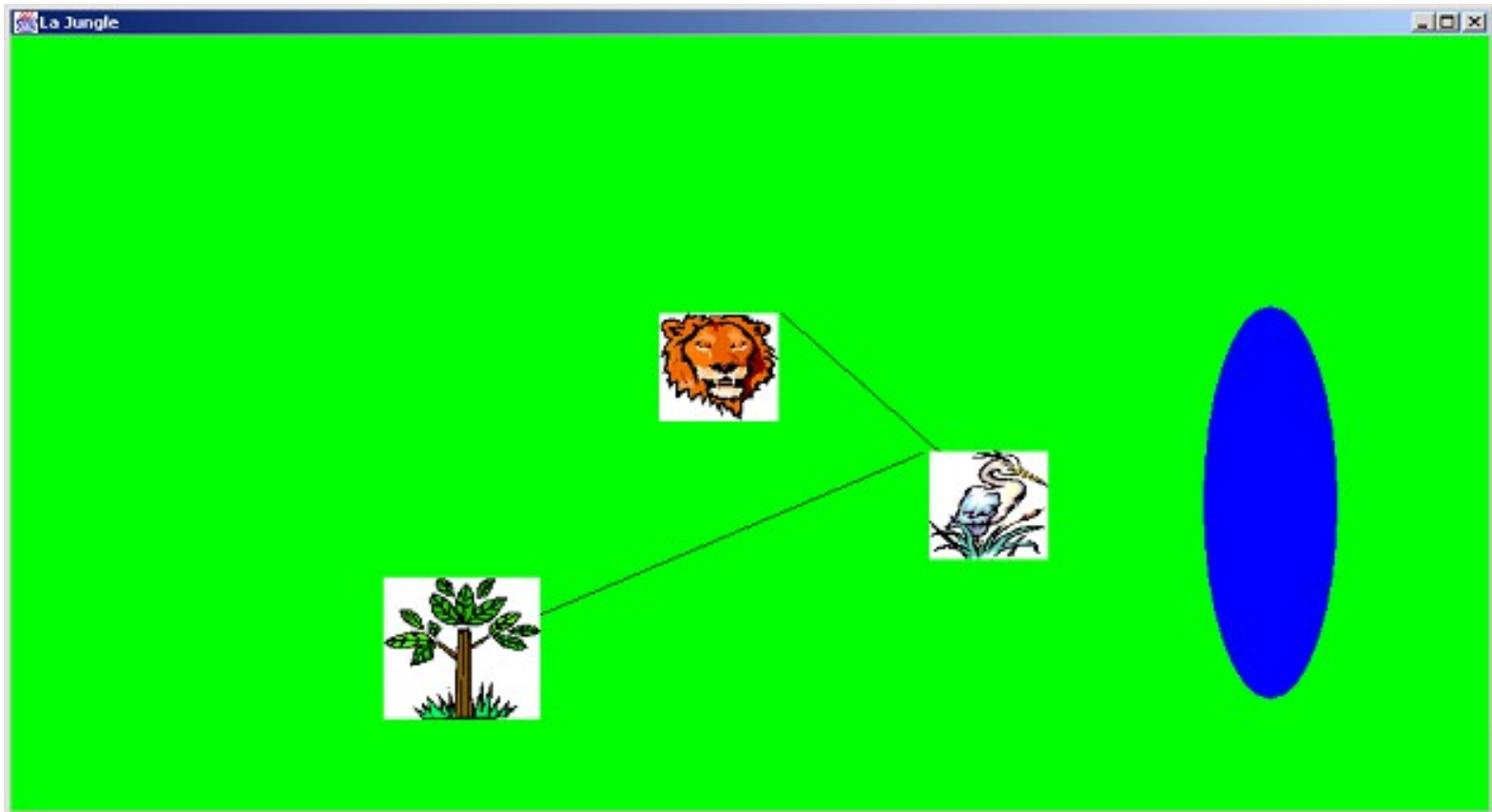
- Des décennies durant, on a programmé en informatique en se concentrant prioritairement sur ce qui « devait se passer » dans le programme.
- On a ainsi structuré les programmes en ensembles de « traitements », définis sous forme des fonctions ou procédures, que l'on pouvait appeler à tout moment et qui pouvaient manipuler toutes les données du programme.
- Mais les problèmes que doivent résoudre les programmes informatiques sont devenus de plus en plus complexes et ce découpage en fonctions ou traitements est devenu sous-optimal.
- De fait, il présente 3 inconvénients majeurs:
  - Il rend difficile la réutilisation d'un bout de programme hors de son contexte
  - Il force l'information (autrement dit les données, càd la matière première de l'informatique) à se plier aux traitements (on les structure en fonction des traitements à réaliser) alors qu'elles devraient être le point de départ
  - Il est inadapté à la façon dont l'homme décrit les problèmes

# Les concepts de l'OO

## Un petit écosystème

- But du jeu?

→ Simuler la vie dans une petite jungle composée de prédateurs, de proies, de plantes et de points d'eau



# Les concepts de l'OO

## Un petit écosystème

- **Règles du jeu?**

- Les proies se déplacent soit vers l'eau, soit vers une plante, soit pour fuir un prédateur en fonction du premier de ces objets qu'elles repèrent
- Les prédateurs se déplacent soit vers l'eau, soit vers une proie en fonction du premier de ces objets qu'ils rencontrent
- Les plantes poussent lentement au cours du temps, tandis que l'eau s'évapore peu à peu
- Les animaux épuisent leurs ressources énergétiques peu à peu, au fur et à mesure de leurs déplacements
- Lorsque les animaux rencontrent de l'eau, ils s'abreuvent et récupèrent de l'énergie tandis que le niveau de l'eau diminue en conséquence
- Lorsqu'un prédateur rencontre une proie, il la dévore et récupère de l'énergie
- Lorsqu'une proie rencontre une plante, il la dévore et récupère de l'énergie

# Les concepts de l'OO

## Un petit écosystème

- **Comment procéder?**

→ En découpant le problèmes en quelques grand traitements:

1. La proie et le prédateur doivent se déplacer

→ On pense/code le déplacement de la proie et du prédateur ensemble

→ Chacun des animaux doit pour cela repérer une cible

→ On conçoit une fonctionnalité de repérage commune à tous les acteurs

→ Les animaux cherchent l'eau

→ Le prédateur cherche la proie

→ La proie cherche la plante et à éviter le prédateur

2. Les animaux se ressourcent

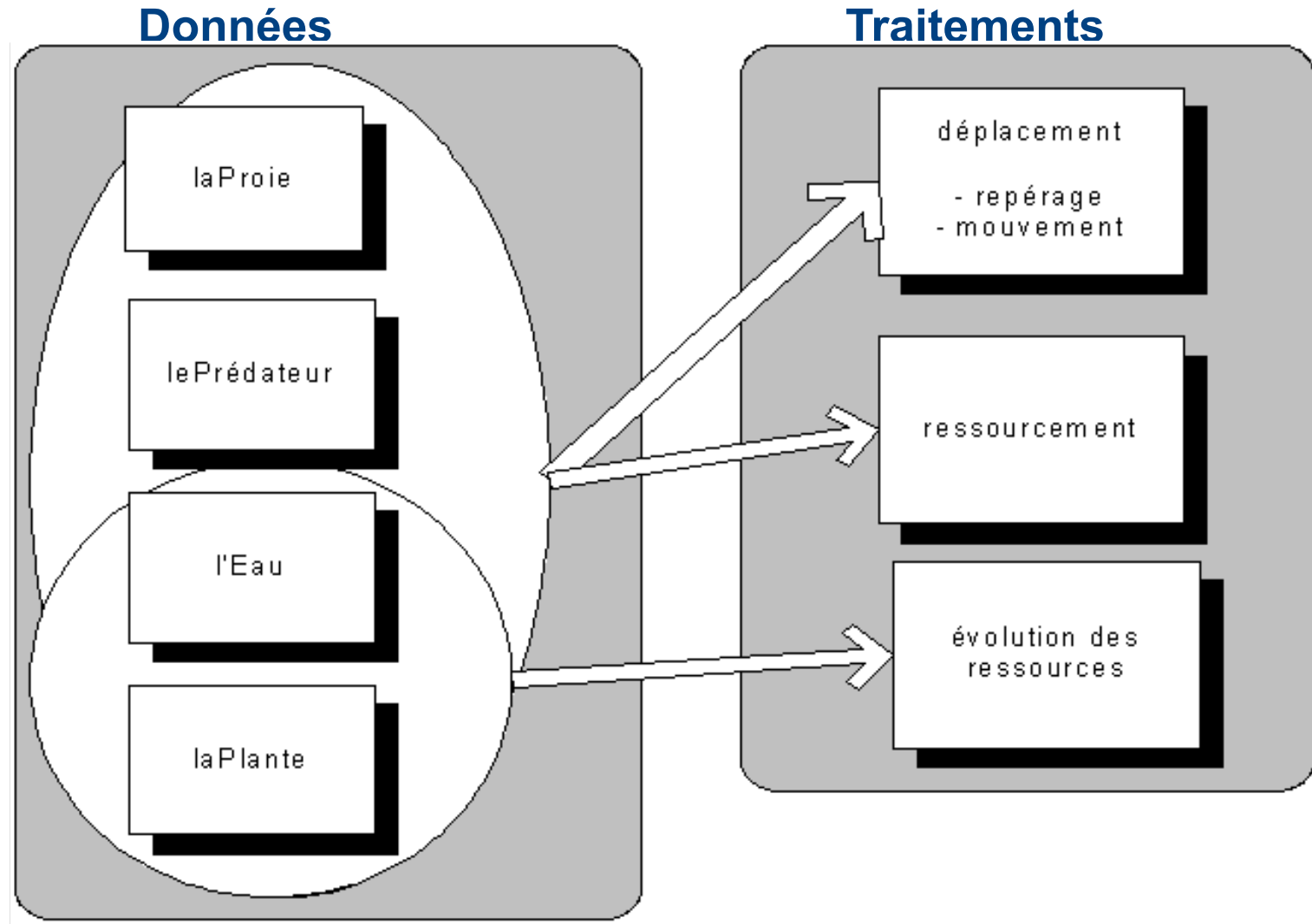
→ On conçoit une fonctionnalité qui concerne tous les acteurs

3. Les ressources de la plante et de l'eau évoluent

→ On pense / code une fonction d'évolution des ressources

# Les concepts de l'OO

## Un petit écosystème





# Les concepts de l'OO

## Un petit écosystème

- Enseignements?

- Les données constituent des ensembles globaux traités globalement
- Les traitements sont regroupés dans quelques grandes fonctions qui peuvent être imbriquées et faire appel les unes aux autres
- Les données sont donc partagées collectivement et les fonctions sont interpénétrées
- On a donc pensé aux traitements et à leur enchaînement AVANT de penser aux données
  - ➔ Approche « TOP-DOWN »
- Modus operandi de plus en plus complexe à mesure que la complexité et interdépendances des traitements s'intensifient
- Difficulté de réutiliser les fonctions dans un autre contexte
- Difficulté de maintenance: toutes les activités impliquant tous les objets
  - ➔ Si on modifie une entité, il faut vérifier l'entièreté du code



# Les concepts de l'OO

## Flashback au temps de l'assembleur

- **Aux origines de l'informatique**

- Programmation des ordinateurs dictée par le fonctionnement des processeurs
- Programme = Succession d'instructions
- Organisation du programme et nature des instructions doit être le plus proche possible de la façon dont le processeur les exécute
  - Modification des données mémorisées
  - Déplacement des données d'un emplacement à un autre
  - Opérations arithmétiques et de logique élémentaire
- Programmation en langage « machine »
- Exemple: «  $c = a + b$  » se programme
  - LOAD a, REG1
  - LOAD b, REG2
  - ADD REG3, REG1, REG2
  - MOVE c, REG3

# Les concepts de l'OO

## Flashback au temps de l'assembleur

- **Langages de programmation procéduraux**

- Mise au point d'algorithmes plus complexes
- Nécessité de simplifier la programmation
  - ➔ Distance par rapport au fonctionnement des processeurs
- Apparition de langages procéduraux
  - ➔ Cobol, Basic, Pascal, C, etc.
- S'intercalent entre langage ou raisonnement naturel et instructions élémentaires
- Permettant d'écrire les programmes en langage plus verbal
  - ➔ Ex: IF a>b THEN a = a + 5 GOTO 25...
- Nécessité donc d'une traduction des programmes en instructions élémentaires ➔ Compilation
- Raisonnement reste néanmoins conditionné par la conception des traitements et leur succession
  - ➔ Eloigné de la manière humaine de poser et résoudre les problèmes
  - ➔ Mal adapté à des problèmes de plus en plus complexes

# Les concepts de l'OO

## Pourquoi l'OO?

- L'avènement de l'objet, il y a 40 ans...
  - Afin de
    - Libérer la programmation des contraintes liées au fonctionnement des processeurs
    - Rapprocher la programmation du mode cognitif de résolution des problèmes
  - Mise au point d'un nouveau style de langages de programmation
    - ➔ Simula, Smalltalk, C++, Eiffel, Java, C#, Delphi, PowerBuilder, Python...
  - Idée?
    - ➔ Placer les entités, objets ou acteurs du problème à la base de la conception
    - ➔ Etudier les traitements comme des interactions entre les différentes entités
    - ➔ **Penser aux données AVANT de penser aux traitements**
    - ➔ Penser un programme en modélisant le monde tel qu'il nous apparaît
    - ➔ Définir des objets et les faire interagir

# Les concepts de l'OO

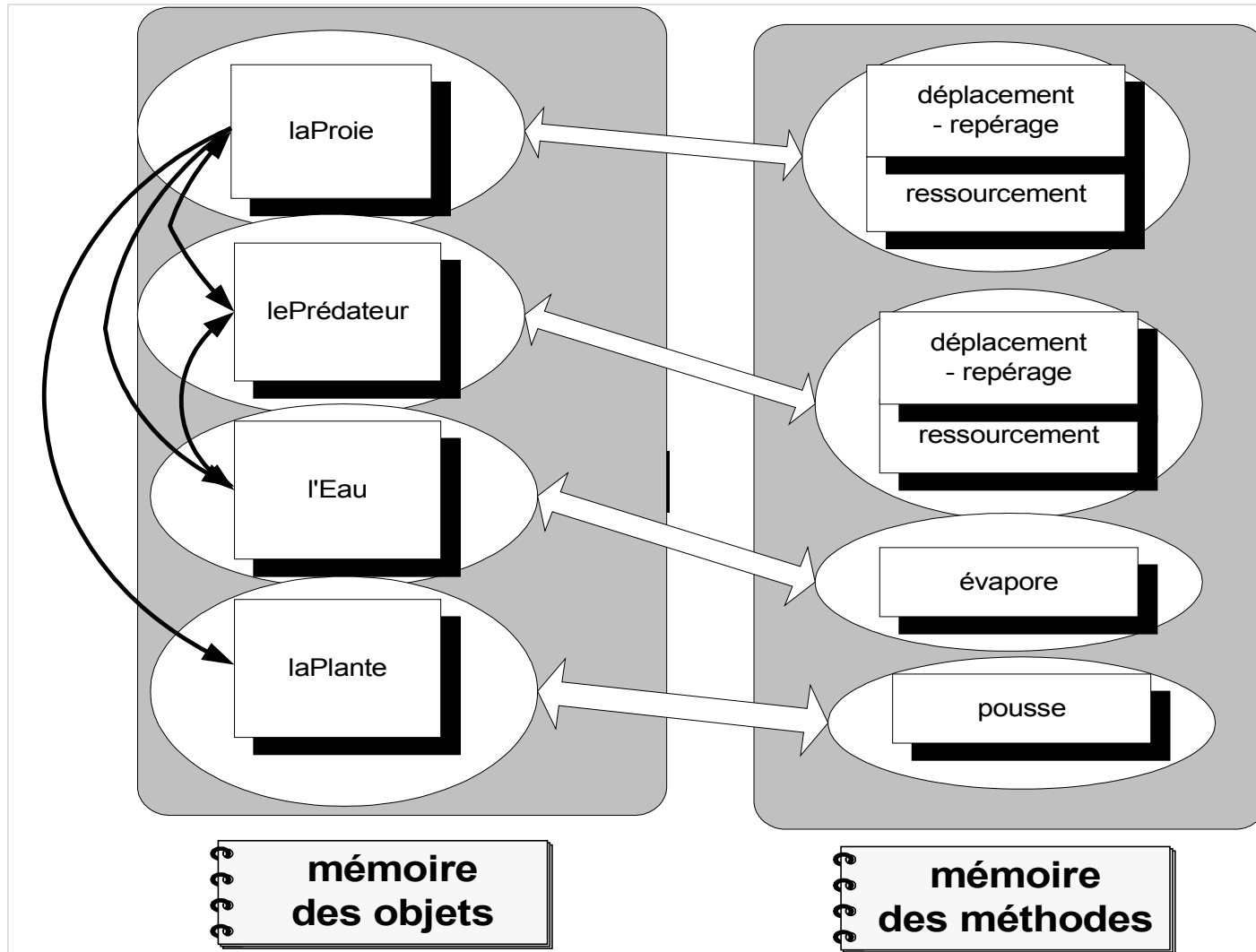
## Qu'est-ce que l'OO?

- **Comment procéder?**

- Identifier les acteurs du problème: Proie/Prédateur/Plante/Eau
- Identifier les actions ou comportements de chaque acteur
  - Le prédateur:
    - se déplace en fonction des cibles, peut boire l'eau et manger la proie
  - La proie:
    - se déplace en fonction des cibles, peut boire l'eau et manger la plante
  - La plante:
    - pousse et peut diminuer sa quantité
  - L'eau:
    - s'évapore et peut diminuer sa quantité

# Les concepts de l'OO

## Qu'est-ce que l'OO?



# Les concepts de l'OO

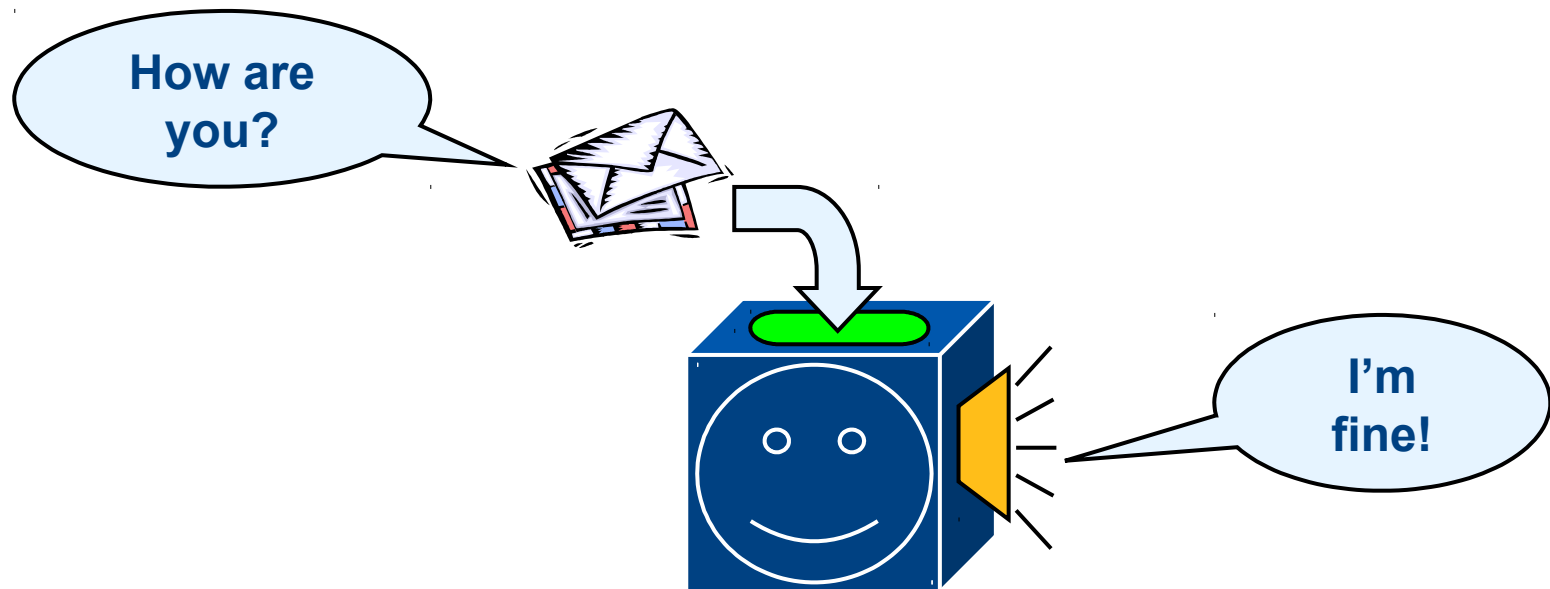
## Qu'est-ce que l'OO?

- La programmation consistera donc à définir les différents acteurs (objets), leurs caractéristiques et comportements et à les faire interagir.
- La programmation OO permet donc de découper les problèmes informatiques non en termes de fonctions (que se passe-t-il?) mais en termes de données (qui intervient dans le problème?) et des services que celles-ci peuvent rendre au reste du programme (les comportements).
- Programmer en OO, c'est donc définir des objets et les faire interagir.

# Les concepts de l'OO

## Qu'est-ce qu'un objet?

- La programmation OO consiste à définir des objets et à les faire interagir
- Qu'est-ce qu'un objet?
  - ➔ Une boîte noire qui reçoit et envoie des messages





# Les concepts de l'OO

## Qu'est-ce qu'un objet?

- **Que contient cette boîte?**

- Du code → Traitements composés d'instructions  
→ Comportements ou Méthodes
- Des données → L'information traitée par les instructions et qui caractérise l'objet → Etats ou Attributs
- Données et traitements sont donc indissociables
- Les traitements sont conçus pour une boîte en particulier et ne peuvent pas s'appliquer à d'autres boîtes

# Les concepts de l'OO

## Qu'est-ce qu'un objet?

- Quelques exemples?

Objet	Attributs	Méthodes
Chien	Nom, race, âge, couleur	Aboier, chercher le baton, mordre, faire le beau
Téléphone	N°, marque, sonnerie, répertoire, opérateur	Appeler, Prendre un appel, Envoyer SMS, Charger
Voiture	Plaque, marque, couleur, vitesse	Tourner, accélérer, s'arrêter, faire le plein, klaxonner

# Exercices

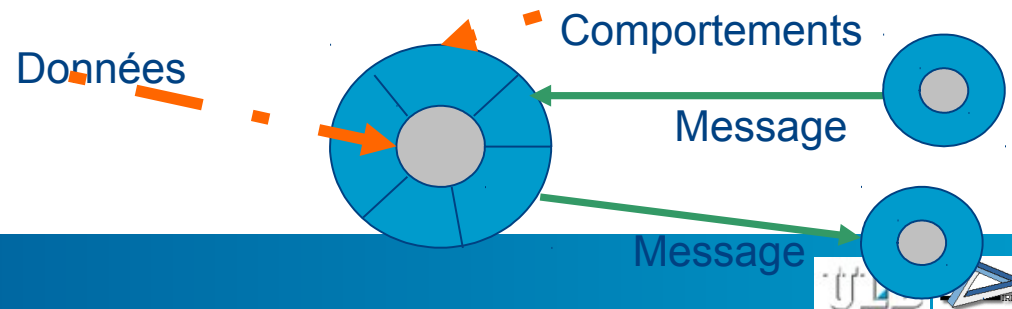
- **Comptes en banque**

- Quelles sont les variables d'états caractéristiques d'un compte en banque?
- Devraient-elles être accessibles ou non de l'extérieur?
- Quels sont les comportements possibles d'un compte en banque?
- Devraient-elles être accessibles ou non de l'extérieur?

# L'encapsulation

- Pourquoi une boîte noire?

- ➔ L'utilisateur d'un objet ne devrait jamais avoir à plonger à l'intérieur de la boîte
- ➔ Toute l'utilisation et l'interaction avec l'objet s'opère par messages
- ➔ Les messages définissent l'interface de l'objet donc la façon d'interagir avec eux
- ➔ Il faut et il suffit de connaître l'interface des messages pour pouvoir exploiter l'objet à 100%, sans jamais avoir à connaître le contenu exact de la boîte ni les traitements qui l'animent
- ➔ L'intégrité de la boîte et la sécurité de ses données peut être assurée
- ➔ Les utilisateurs d'un objet ne sont pas menacés si le concepteur de l'objet en change les détails ou la mécanique interne
- ➔ **ENCAPSULATION**



# La classe

## Un objet sans classe n'a pas de classe

- **Comment les objets sont-ils définis?**

- Par leur classe, qui détermine toutes leurs caractéristiques
  - Nature des attributs et comportements possibles
- La classe détermine tout ce que peut contenir un objet et tout ce qu'on peut faire de cet objet
- Classe = Moule, Définition, ou Structure d'un objet
- Objet = Instance d'une classe
- Une classe peut-être composite  $\Leftrightarrow$  peut contenir elle-même des objets d'autres classes
  - Ex: Une voiture contient un moteur qui contient des cylindres...
  - Ex: Un chien possède des pattes...

# La classe

## Un objet sans classe n'a pas de classe

- **Quelques exemples?**

- La classe « chien » définit:
  - Les attributs d'un chien (nom, race, couleur, âge...)
  - Les comportements d'un chien (Aboier, chercher le bâton, mordre...)
- Il peut exister dans le monde plusieurs objets (ou instances) de chien

Classe	Objets
Chien	Mon chien: Bill, Teckel, Brun, 1 an Le chien de mon voisin: Hector, Labrador, Noir, 3 ans
Compte	Mon compte à vue: N° 210-1234567-89, Courant, 1.734 €, 1250 € Mon compte épargne: N° 083-9876543-21, Epargne, 27.000 €, 0 €
Voiture	Ma voiture: ABC-123, VW Polo, grise, 0 km/h La voiture que je viens de croiser: ZYX-987, Porsche, noire, 170 km/h

# La classe

## Un objet sans classe n'existe pas

Déclaration  
de la classe

Variables d'instance  
ou « champs »

Définition du  
constructeur

Méthodes d'accès

Définition  
des  
méthodes

```
import java.lang.*;
public class Voiture {
    private String marque;
    private int vitesse;
    private double reserveEssence;
    public Voiture(String m,int v,double e){
        marque=m ;
        vitesse=v;
        reserveEssence=e;
    }
    public String getMarque(){return marque;}
    public void setMarque(String m){marque=m;}
    ...
    public void accelere(int acceleration) {
        vitesse += acceleration;
    }
    public void ravitaille(double quantite){
        reserveEssence+=quantite;
    }
}
```

Class Body

# La classe

## Déclaration des attributs

- Les attributs d'un objet sont définis par ses variables membres
- Pour définir les attributs d'un objet, il faut donc définir les variables membres de sa classe
- Les principes suivants gouvernent toutes les variables en Java:
  - Une variable est un endroit de la mémoire à laquelle on a donné un nom de sorte que l'on puisse y faire facilement référence dans le programme
  - Une variable a une valeur, correspondant à un certain type
  - La valeur d'une variable peut changer au cours de l'exécution du programme
  - Une variable Java est conçue pour un type particulier de donnée
  - Une déclaration typique de variable en Java a la forme suivante:
    - `int unNombre;`
    - `String uneChaineDeCaracteres;`



# La classe

## Déclaration des attributs

- **Rappel:** toute variable doit être déclarée et initialisée
- **Les variables membres** sont des variables déclarées à l'intérieur du corps de la classe mais à l'extérieur d'une méthode particulière, c'est ce qui les rend accessibles depuis n'importe où dans la classe.
- **La signature de la variable :**
  - Les modificateurs d'accès: indiquent le niveau d'accessibilité de la variable
  - optionnels** {
    - [static]: permet la déclaration d'une variable de classe
    - [final]: empêche la modification de la variable (→ Crée une constante)
    - [transient]: on ne tient pas compte de la variable en sérialisant l'objet
    - [volatile]: pour le multithreading
  - Le type de la variable (ex: int, String, double, RacingBike,...)
  - Le nom de la variable (identificateur)
- **Exemples:**
  - `private int maVitesse; // Je possède un entier « maVitesse »`
  - `private final String maPlaque; // « maPlaque » est constante`

# La classe

## Déclaration des attributs

**Les modificateurs d'accès qui caractérisent l'encapsulation sont justifiées par différents éléments:**

- **Préservation de la sécurité des données**
  - Les données privées sont simplement inaccessibles de l'extérieur
  - Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques
- **Préservation de l'intégrité des données**
  - La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable
- **Cohérence des systèmes développés en équipes**
  - Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

# La classe

## Déclaration des attributs

- En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des attributs et des méthodes.
- Les accès sont contrôlés en respectant le tableau suivant:

Mot-clé	classe	package	sous classe	world
<code>private</code>	Y			
<code>protected</code>	Y	Y	Y	
<code>public</code>	Y	Y	Y	Y
[aucun]	Y	Y		

Seul les membres publics sont visibles depuis le monde extérieur.

Une classe a toujours accès à ses membres.

Les classes d'un même package protègent uniquement leurs membres privés (à l'intérieur du package)

Une classe fille (ou dérivée) n'a accès qu'aux membres publics et `protected` de la classe mère.

# Bien utiliser les accesseurs et comprendre l'encapsulation

- De manière générale, on évitera de déclarer un attribut “public”
- Ou alors, n'importe qui pourra modifier la variable.
- Il faudra paramétrer les accès en utilisant les packages : seules les classes du même package (que vous contrôlez vous même!!!) pourront modifier les variables packagées.
- Pour packager une variable, une méthode ou une classe. ne mettez aucun accesseur devant celle-ci. Créez juste un package (càd. un répertoire dans le système de fichiers) et déclarez que votre classe appartient au package au tout début du fichier via :

```
package nomdepackage;
```

# Exercices

- **Création d'une classe compte en banque packagée**
  - Créez un nouveau projet Eclipse
  - Créez un nouveau package « banksys »
  - Créez-y une nouvelle classe « CompteEnBanque »
  - Déclarez les attributs de la classe
- **Orienté objet et bases de données**
  - Quelle analogie peut-on établir entre orienté objet et bases de données?
  - Quelles sont les limites de cette analogie?

# Exercices

- **Similitudes avec les bases de données?**

- Classe → Table
- Attribut → Champ / Colonne
- Objet → Tuple (enregistrement) de la table
- Valeur → Valeur du champ ou de la colonne

Marque	Modele	Serie	Numero
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

- **Mais il y a des limites à cette analogie (qui apparaîtront plus tard)**

# La classe

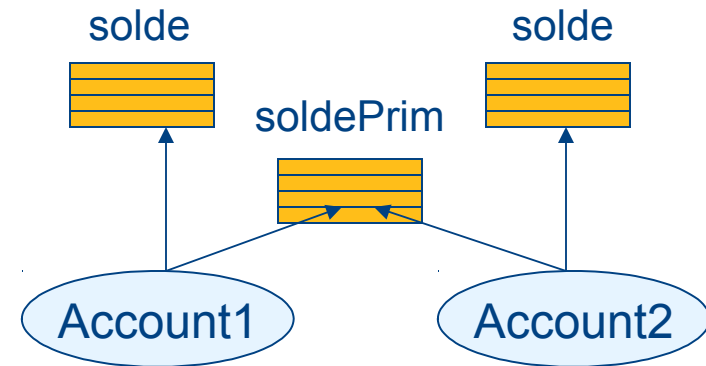
## Déclaration des attributs

- Chaque objet a sa propre “mémoire” de ses variables d’instance
- Le système alloue de la mémoire aux variables de classe dès qu’il rencontre la classe. Chaque instance possède la même valeur d’une variable de classe.

variable d'instance

```
class BankAccount {  
    int solde;  
    static int soldePrim;  
    void deposit(int amount) {  
        solde+=amount;  
        soldePrim+=amount;  
    }  
}
```

variable de classe



# La classe

## Déclaration des attributs

- **Variables et méthodes statiques**

- Initialisées dès que la classe est chargée en mémoire
- Pas besoin de créer un objet (instance de classe)

- **Méthodes statiques**

- Fournissent une fonctionnalité à une classe entière
- Cas des méthodes non destinées à accomplir une action sur un objet individuel de la classe
- **Exemples:** `Math.random()`, `Integer.parseInt(String s)`, `main(String[] args)`
- Les méthodes statiques ne peuvent pas accéder aux variables d'instances (elles sont "au-dessus" des variables d'instances)

```
class AnIntegerNamedX {  
    int x;  
    static public int x() { return x; }  
    static public void setX(int newX) { this.x = newX; }  
}
```

**x est une variable d'instance, donc inaccessible pour la méthode static setX**



# La classe

## Le constructeur

- Comme on l'a vu, les variables doivent être initialisées
- Or la déclaration des attributs n'inclut généralement pas d'initialisation
- De plus, comment pourrait-on connaître à l'avance la valeur des différentes variables membres, puisqu'elle sera propre à chaque instance (objet) de la classe? (Chaque voiture a sa propre marque, sa propre couleur et sa propre vitesse)
- Il faut donc définir dans la classe un mécanisme permettant de donner une valeur aux variables membres (aux états) de l'objet
- Ce mécanisme s'appelle le « constructeur »
- Une classe doit toujours avoir au minimum un constructeur
- Le constructeur recevra les valeurs à attribuer aux variables et les y affectera
- Le constructeur servira ainsi à « matérialiser » la classe et sera donc appelé chaque fois qu'un objet de la classe doit être créé
- Le constructeur est donc la « Recette » pour fabriquer un objet

# La classe

## Le constructeur

- A le même nom que la classe
- Utilise comme arguments des variables initialisant son état interne
- On peut surcharger les constructeurs, i.e définir de multiples constructeurs (plusieurs recettes existent pour fabriquer un même objet avec des ingrédients différents)
- Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
- Signature d'un constructeur:
  - Modificateur d'accès (en général public)
  - Pas de type de retour
  - Le même nom que la classe
  - Les arguments sont utilisés pour initialiser les variables de la classe

```
public Voiture(String m,int
v,double e)
{
    marque=m ;
    vitesse=v;
    reserveEssence=e;
}

public Voiture(String m)
{
    marque=m ;
    vitesse=0;
    reserveEssence=0;
}
```

# Exercices

- **Comptes en banque**
  - Créer deux constructeurs différents dans la classe CompteEnBanque

# Interaction entre objets

## Création d'objets

- Notons toutefois qu'il ne « se passe » toujours rien dans notre programme à ce stade
- En effet, nous avons conçu deux classes, dont nous avons défini les caractéristiques, les constructeurs et les comportements possibles, mais nous n'avons toujours créé aucun objet
- Nous n'avons en fait encore que défini ce que seraient un compte en banque ou une personne si nous en créions dans notre programme
- Pour qu'il « se passe » quelque chose, il faudrait donc que nous matérialisions nos classes, autrement dit que nous lesinstancions, ce qui signifie que nous « créions des objets »
- Pour ce faire, il faut appeler le constructeur de la classe dont nous souhaitons créer une instance

# Interaction entre objets

## Création d'objets

- **L'appel au constructeur**

- Se fait pour initialiser un objet
  - Provoque la création réelle de l'objet en mémoire
  - Par l'initialisation de ses variables internes propres
- Se fait par l'emploi du mot clé « **new** »

```
Voiture v1, v2;
```

```
v1 = new Voiture("Peugeot", 120, 35.3);
```

```
v2 = new Voiture("Ferrari");
```

# Interaction entre objets

## Création d'objets

### Déclaration et création d'objets

- Déclaration : `Point p;`
- Création : `p = new Point(2,3);`

p 

????
------

1. Recherche une place

p 

????
------

x	0
y	0

2. Assignment d'une valeur

p 

????
------

x	7
y	10

3. Exécution du constructeur

p 

????
------

x	2
y	3

4. Création du pointeur

p 

0123abcd
----------



x	2
y	3

## • Comptes en banque

- Création d'une classe Principale
  - Créer une classe « Principale » représentant un scénario possible d'utilisation de vos classes.
  - Mettez cette classe en dehors du package.
  - Y définir une méthode « main », dans laquelle déclarer quelques objets de type « CompteEnBanque »
  - Instancier arbitrairement ces quelques comptes
  - Réaliser quelques opérations sur les comptes (retraits, dépôts, virements, calcul d'intérêts, etc.)
  - Afficher tous les comptes à l'écran
  - Jouez avec les accesseurs et constatez les conséquences au niveau de la méthode main.

# La classe

## Déclaration des méthodes

- **Quid des comportements?**

- La classe définit aussi les comportements des objets
- Les comportements sont les messages que les objets de la classe peuvent comprendre → « Aboie », « Va chercher », « Fais le beau »
- Les traitements ou instructions à réaliser lorsqu'un message en particulier est envoyé à un objet sont définis dans la classe mais restent cachés (cf. boîte noire)
- Certains messages requièrent des renseignements complémentaires → « Va chercher... le bâton », « Aboie... 3 fois très fort »
- Une méthode peut renvoyer une valeur ou un objet à son expéditeur



# La classe

## Déclaration des méthodes

- **Quid des comportements?**

- Pour déclencher un traitement, il faut donc envoyer un message à l'objet concerné → On n'invoque jamais une fonction dans le vide
- L'opérateur d'envoi de messages est le point « . »
- Pour envoyer un message, il faut toujours préciser à quel objet en particulier on le destine
  - Ex: `monChien.vaChercher(leBaton)`
- Si on ne précise pas le destinataire, Java considère qu'on est son propre destinataire
  - Ex: `vaChercher(leBaton)`
    - C'est celui qui envoie l'ordre qui le reçoit
    - Ce qui peut aussi s'écrire: `this.vaChercher(leBaton)`
- « this » signifie en effet « l'objet présent »

# La classe

## Déclaration des méthodes

- Une méthode est composée de:

```
public void deposit (int amount) {  
    solde+=amount ;  
}
```

Sa déclaration  
Son corps

- Signature d'une méthode:

Signature

- Modificateurs d'accès : public, protected, private, aucun
- [modificateurs optionnels] : static, native, synchronized, final, abstract
- Type de retour : type de la valeur retournée
- Nom de la méthode (identificateur)
- Listes de paramètres entre parenthèses (peut être vide mais les parenthèses sont indispensables)
- [exception] (throws Exception)

- Au minimum:

- La méthode possède un identificateur et un type de retour
- Si la méthode ne renvoie rien → le type de retour est void

- Les paramètres d'une méthode fournissent une information depuis l'extérieur du "scope" de la méthode (idem que pour le constructeur)

# La classe

## Déclaration des méthodes

- **Qu'est-ce que « l'interface » d'une méthode**

- L'interface d'une méthode, c'est sa signature
- Cette signature, qui définit l'interface de la méthode, correspond en fait au message échangé quand la méthode est appelée
- Le message se limite de fait uniquement à la signature de la méthode
  - Type de retour
  - Nom
  - Arguments
- L'expéditeur du message n'a donc jamais besoin de connaître l'implémentation ou corps de la méthode
- On a donc:
  - Déclaration = Signature = Message de la méthode
  - Bloc d'instruction = Corps = Implémentation de la méthode

# Exercices

- **Comptes en banque**

- Déclarer et implémenter les méthodes (comportements) de la classe compte en banque
- Définissez en particulier une méthode « afficheToi » qui affiche les caractéristiques du comptes à l'écran
- Utilisez pour cela la méthode `System.out.println (« ... ») ;`

# Interaction entre objets

## Qu'est-ce qu'un envoi de message?

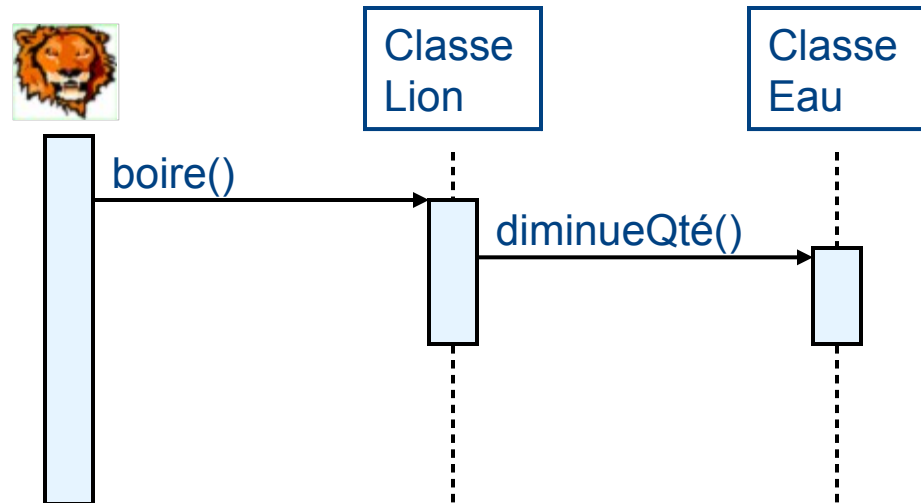
- Tout ce qui se produit dans un programme OO est le résultat d'objets qui interagissent
- L'interaction entre deux objets consiste en l'envoi d'un message de l'objet A à l'objet B
- Cet envoi de message est comme un ordre ou une question qu'un objet adresse à l'autre
- Techniquement, un envoi de message se matérialise par l'appel d'une méthode définie dans la classe de l'objet destinataire
- Exemple:
  - Soit un objet Maître et un objet Chien
  - Le maître peut envoyer un message au chien:  
→ `monChien.faisLeBeau()` ;

# Interaction entre objets

## Qu'est-ce qu'un envoi de message?

### • Exemple?

- Quand le lion s'abreuve, il provoque une diminution de la quantité d'eau
- Ce n'est pas le lion qui réduit cette quantité
- Le lion peut seulement envoyer un message à l'eau, lui indiquant la quantité d'eau qu'il a consommée
- L'eau gère seule sa quantité, le lion gère seul son énergie et ses déplacements



# Interaction entre objets

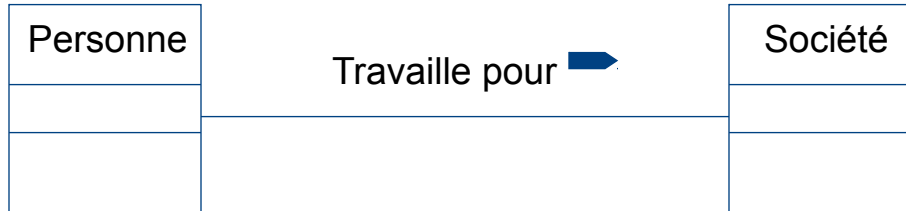
## Association

- **Comment le lion connaît-il le message à envoyer à l'eau?**
  - Pour pouvoir envoyer un message à l'eau, il faut que le lion connaisse précisément l'interface de l'eau
  - La classe lion doit pour cela « connaître » la classe eau
  - Cette connaissance s'obtient par l'établissement d'une communication entre les deux classes
  - De tels liens peuvent être
    - Circonstantiels et passagers → **Dépendance**  
(le lien ne se matérialise que dans une méthode)
    - Persistants → **Association**
  - On distingue 3 types de liens d'association, du + faible au + fort:
    - Association simple (il y a envoi de messages entre les classes)
    - Agrégation faible (une classe possède un attribut de l'autre type)
    - Agrégation forte = Composition (l'agrégé n'existe pas sans son contenant)

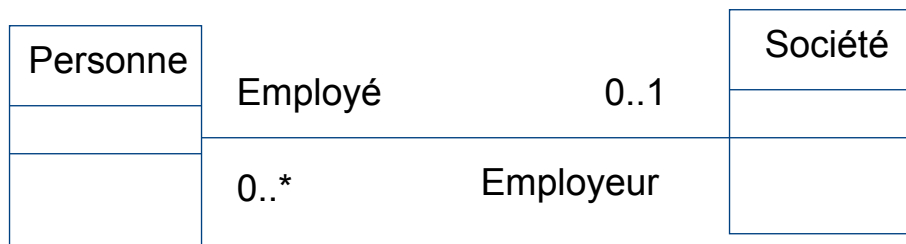
# Interaction entre objets

## Association

- L'association entre classes



- Multiplicité des associations



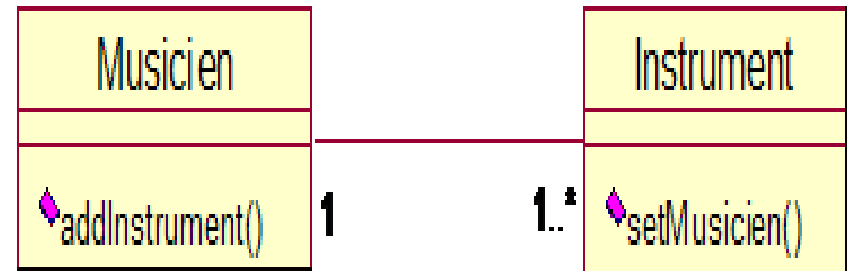


# Interaction entre objets

## Association

- **Exemple:**

```
class Musicien {  
    private List<Instrument> mesInstruments = new  
    List<Instrument>();  
    public Musicien() {}  
    public void addInstrument(Instrument unInstrument) {  
        mesInstruments.add(unInstrument);  
        unInstrument.setMusicien(this);  
    }  
}  
  
class Instrument {  
    private Musicien monMusicien;  
    public Instrument() {}  
    public void setMusicien(Musicien monMusicien) {  
        this.monMusicien = monMusicien;  
    }  
}
```

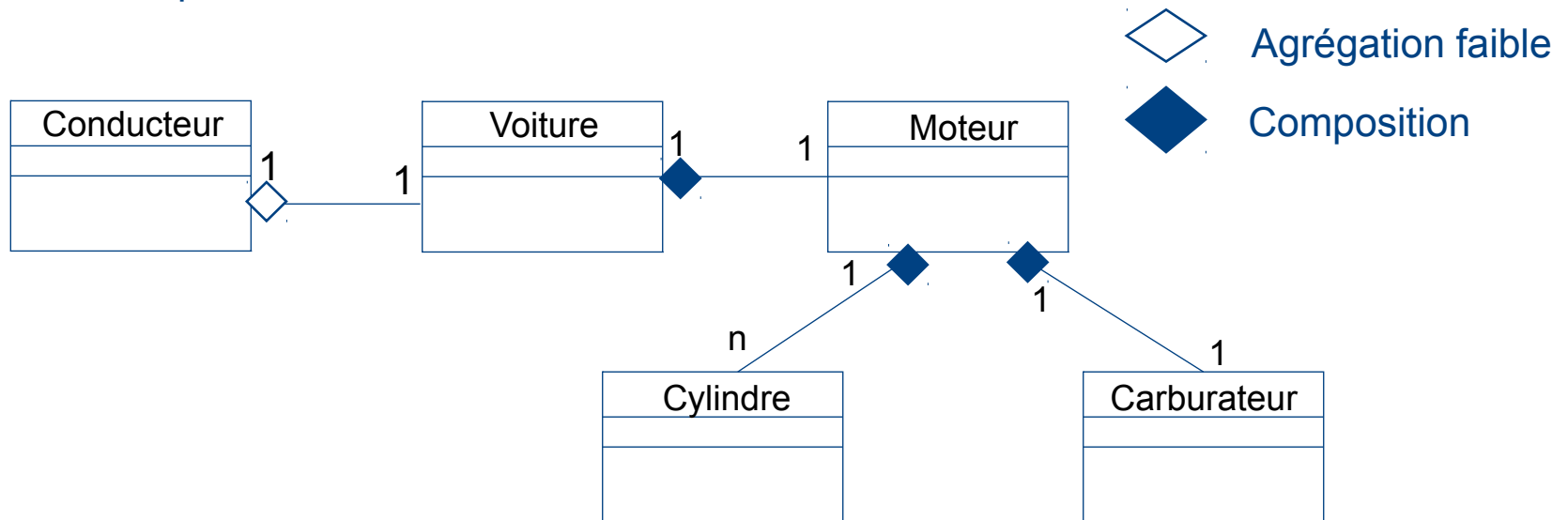


# Interaction entre objets

## Association

- Deux types d'association particuliers:

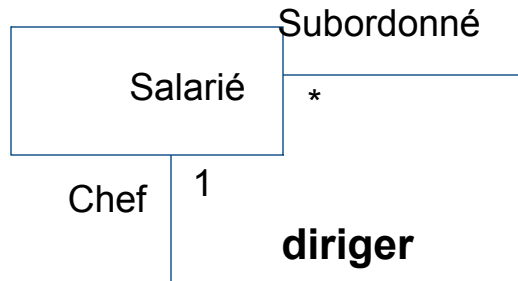
- L'agrégation faible
  - L'objet « agrégeant » contient une ou plusieurs instances de la classe « agrégée »
- L'agrégation forte ou Composition
  - L'objet « composé » est créé dans la classe « composite » et est détruit si l'objet composite est détruit



# Interaction entre objets

## Association

- Une classe peut être associée à elle-même: **Auto-association**



- L'association peut être unidirectionnelle  
(les messages ne sont envoyés que d'une classe vers l'autre)



# Modéliser les classes avec des diagrammes UML

## Diagramme de classes

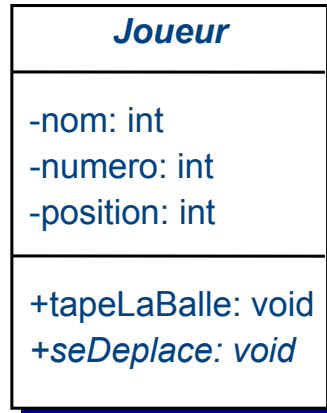
- Le but du diagramme de classes est de représenter les classes au sein d'un modèle
- Dans une application OO, les classes possèdent:
  - Des attributs (variables membres)
  - Des méthodes (fonctions membres)
  - Des relations avec d'autres classes
- C'est tout cela que le diagramme de classes représente
- L'encapsulation est représentée par:
  - (private), + (public), # (protected)
- Les attributs s'écrivent:
  - /+/# nomVariable : Type
- Les méthodes s'écrivent:
  - /+/# nomMethode(Type des arguments) : Type du retour ou « void »

Nom Classe
Attributs
Méthodes()

# Modéliser les classes avec des diagrammes UML

## Diagramme de classes

- Exemple



# Modéliser les classes avec des diagrammes UML

## Représenter les relations entre classes

- Les relations d'héritage sont représentées par:

- A  B signifie que la classe A hérite de la classe B

- L'association est représentée par:

- A  B signifie que la classe A envoie des messages à la classe B grâce à ses attributs de type B

- L'agrégation faible est représentée par:

- A  B signifie que la classe A a pour caractéristique un ou plusieurs attributs de type B

- L'agrégation forte (ou composition) est représentée par:

- A  B signifie que les objets de la classe B ne peuvent exister qu'au sein d'objets de type A où ils sont créés

# Exercices

- **Comptes en banque**

- Créer une classe « Personne » avec quelques attributs, méthodes et constructeurs
- Instancier la classe Personne en tant qu'attribut de la classe CompteEnBanque
- Faites en sorte que le CompteEnBanque, en s'affichant à l'écran, indique le nom de son titulaire
- Une personne peut posséder plusieurs compte en banque
- **N'oubliez pas Faites d'abord un diagramme de classes!**
- Définissez en particulier les 3 méthodes suivantes dans la classe Personne:
  - Une méthode « getSoldeTotal(): double » qui renverra le solde cumulé de tous les comptes de la personne
  - Une méthode « afficheToi() » qui affichera à l'écran les caractéristiques de la personne dont le nombre de compte qu'elle possède ainsi que le solde total de ses comptes et puis listera tous ses comptes
  - Ajouter une méthode « ouvrirCompte(...): void » dans la classe Personne qui attend un compte en banque en paramètre et le stocke dans son tableau

# Interaction entre objets

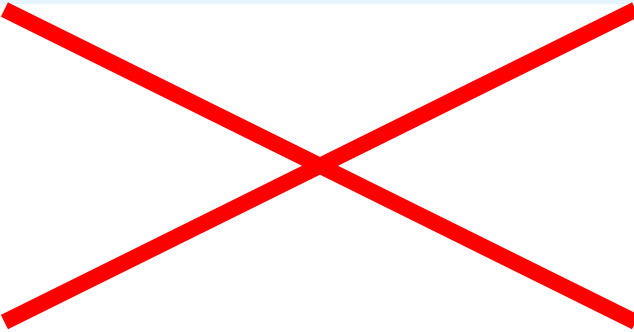



## Association

- L'association se matérialise donc par la déclaration d'objets d'un certain type en tant que variables membres d'une autre classe
- Ce faisant, il en résulte un nouveau type de variable, propre aux langages OO, et différent des variables « primitives » que l'on manipule traditionnellement en informatique
- De fait, au lieu de simples variables numérique, booléennes ou textuelles, nous pouvons aussi bien déclarer et manipuler des variables « objet » comme un chien, une voiture, un client, ou encore un compte en banque
- On appelle ces variables « variables de référence » par opposition aux « variables primitives »
- Leur caractéristique principale est qu'elles n'ont pas pour valeur l'objet qu'elle désigne, mais seulement la référence de celui-ci, c'est-à-dire l'adresse à laquelle l'objet qu'elles désignent se trouve dans la mémoire du programme ➔ On parle de « pointeurs » (implicites)



# Interaction entre objets

## Variables de référence

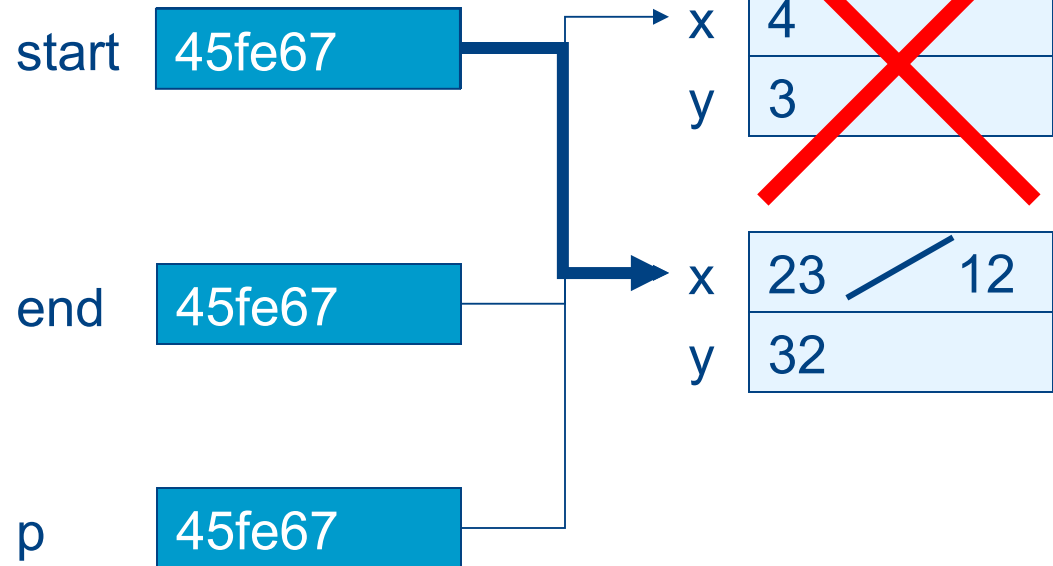
Mémoire des référents	Mémoire des valeurs	Mémoire des objets
int unNombre	10	
double unNombreDecimal	27.33	
boolean unBooléen	<i>true</i>	
Chien monChien	ab123d	
Chien leChienQueTuAsVu	ab123d	
Voiture uneVoitureDeCourse	e4f5a6	
Voiture maVoiture	d12e45	

# Interaction entre objets

## Variables de référence

### Assignation d'un type de référence

- `Point start=new Point(4,3);`
- `Point end=new Point(23,32);`
- `Point p=end;`
- `p.x=12;`
- `start=p;`



Il n'y a désormais plus de référence vers l'ancien Point « start », il sera donc détruit par le Garbage Collector

# Exercices

- **Analyser la classe Point**

- Que se passe-t-il à l'exécution du programme?
- Exécuter le programme pour vérifier

# Interaction entre objets

## Variables de référence

- **Java est un langage dit « fortement typé » :**
  - Toute variable doit être déclarée
  - Dès sa déclaration, une variable se voit attribuer un type donné
  - Une variable ne peut jamais changer de type
  - Le type de données précise
    - les valeurs que la variable peut contenir ou le type d'objet qu'elle peut désigner (une variable de type « Chat » ne peut pas désigner un objet de type « Chien »)
    - les opérations que l'on peut réaliser dessus
      - ➔ S'il s'agit d'une variable primitive, uniquement les opérations arithmétiques ou binaires correspondantes
      - ➔ S'il s'agit d'une variable de référence, uniquement les méthodes définies dans la classe correspondant au type de la variable

# Interaction entre objets

## Qu'est-ce qu'un envoi de message?

- **Les arguments d'une méthode peuvent être de deux types**
  - Variable de type primitif
  - Objet
- **Lorsque l'argument est une variable de type primitif, c'est la valeur de la variable qui est passée en paramètre**
- **Lorsque l'argument est un objet, il y a, théoriquement, deux éléments qui pourraient être passés en paramètre:**
  - La référence vers l'objet
  - L'objet lui-même
- **A la différence de C++, Java considère toujours que c'est la valeur de la référence (plus exactement une copie de cette valeur) et non l'objet qui est passée en argument**

# Exercices

- **Comptes en banque**

- Créer une Classe banque
  - Transformer la classe banque en classe OO
  - Lui associer un tableau de comptes et un tableau de clients
  - Instancier quelques comptes et clients dans son constructeur et les associer les uns aux autres
  - **N'oubliez pas Faites d'abord un diagramme de classes!**
  - Toutes les opérations sur les comptes devront se faire obligatoirement au travers de la classe publique Banque (façade) – Les compteEnBanque et les Personne sont invisible en dehors du package!
  - Implémentez les méthodes : enregsitrerUnClient, ouvririComptePourClient, depotSurCompte, retraitreDeCompte, virement
  -

# Exercices

- **Bonus**

- Que se passe-t-il dans le cas de plusieurs banques?
- Quelle conséquence cela-a-t-il sur les identifiants des comptes?
- Comment assurer de manière transparente un virement interbancaire ?
- Quelles solutions proposez-vous?

# Héritage

## En quoi consiste l'héritage?

- **Supposons qu'il existe déjà une classe qui définit un certain nombre de messages et qu'on ait besoin d'une classe identique mais pourvue de quelques messages supplémentaires**
  - ➔ Comment éviter de réécrire la classe de départ?
  - ➔ Regrouper les classes en super-classes en factorisant et spécialisant
  - ➔ La sous-classe hérite des attributs et méthodes et peut en rajouter de nouveaux

### CompteEnBanque

CompteCourant

CptProfessionnel

CompteEpargne

CptBloqué

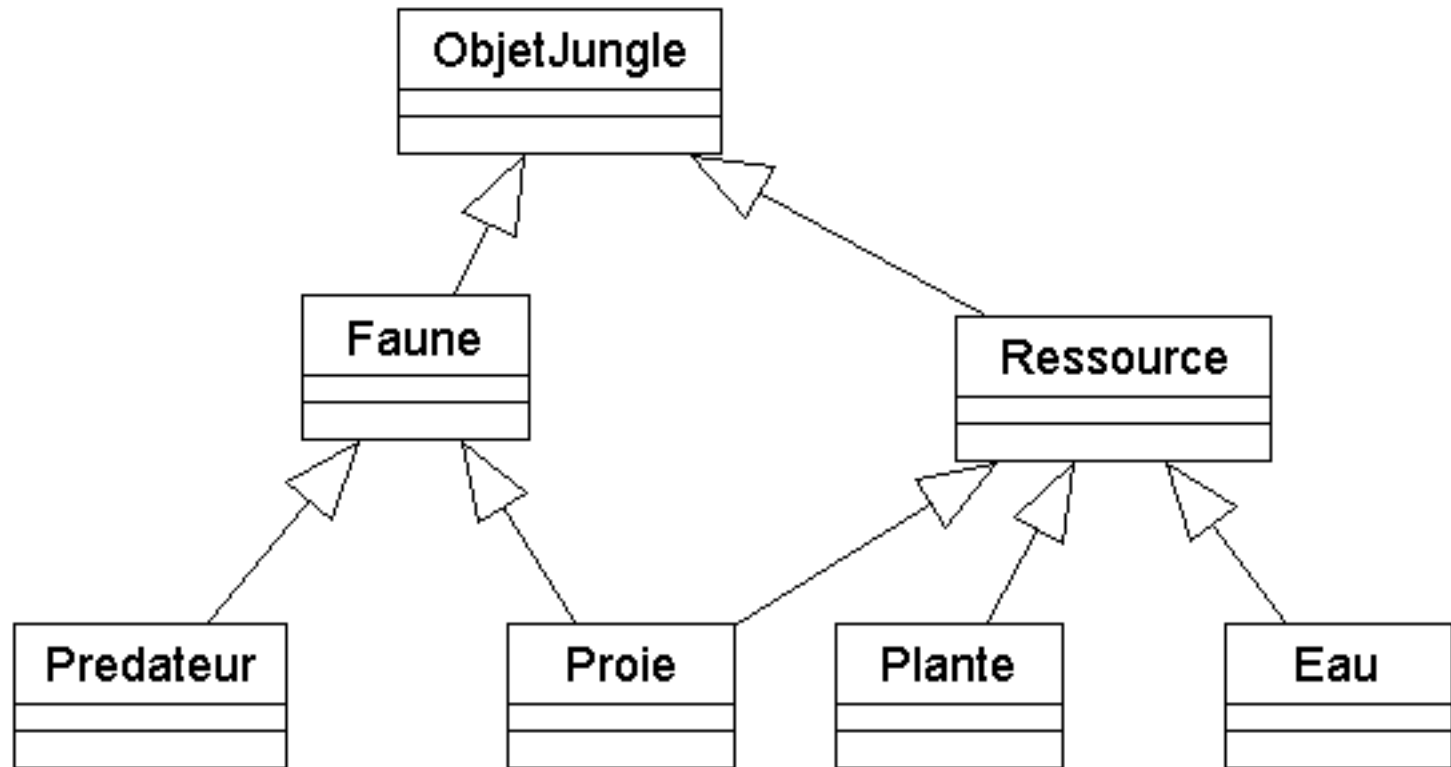


# Héritage

## En quoi consiste l'héritage?

- **Ex: Dans l'écosystème**

- La classe Faune regroupe les animaux
- La classe Ressource regroupe l'eau et la plante

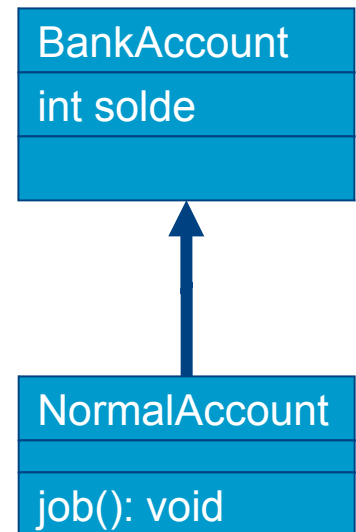


# Héritage

## Comment mettre en œuvre l'héritage?

- Pour réaliser un héritage en Java, il suffit de le déclarer dans la déclaration de la classe au moyen de la clause « *extends* »
- Une sous-classe hérite des variables et des méthodes de ses classes parentes
- Java n'offre pas la possibilité d'héritage multiple
- A la racine de l'héritage Java se trouve la classe « Object » dont toutes les classes héritent automatiquement

```
class BankAccount {  
    protected int solde;  
    ...  
}  
  
class NormalAccount extends BankAccount {  
    public void job(){solde+=1000;}  
}
```



# Héritage

## Comment mettre en œuvre l'héritage?

- Comme une sous-classe hérite des variables de la classe parent, elle doit nécessairement les initialiser, donc appeler le constructeur de la classe parent
- Cela doit se faire avant la réalisation de son propre constructeur
- Concrètement, cela implique que la première instruction dans le constructeur d'une sous-classe doit être l'appel au constructeur parent
- Cet appel se fait au moyen du mot-clé « super »
- Si la classe parent possède un constructeur vide, l'appel n'y est alors plus obligatoire mais implicite

```
class Child extends MyClass {  
    Child() {  
        super(6); // appel du constructeur parent  
    }  
}
```

# Héritage

## Comment mettre en œuvre l'héritage?

- Le mot-clé “this” désigne toujours l'objet en cours (de création)

```
class Employee {  
    String name,firstname;  
    Address a;  
    int age;  
    Employee(String name,String firstname,Address a,int age){  
        super(); // Comme le constructeur parent n'attend pas  
                // d'argument, cet appel n'est pas obligatoire  
        this.firstname=firstname;  
        this.name=name;  
        this.a=a;  
        this.age=age;  
    }  
    Employee(String name,String firstname){  
        this(name,firstname,null,-1);  
    }  
}
```

# Héritage

## Comment mettre en œuvre l'héritage?

- La variable `aNumber` du compte normal cache la variable `aNumber` de la classe générale compte en banque. Mais on peut accéder à la variable `aNumber` d'un compte en banque à partir d'un compte normal en utilisant le mot-clé `super` :

`super.aNumber`

```
class BankAccount{  
    int aNumber;  
}  
  
class NormalAccount extends BankAccount{  
    float aNumber;  
}
```

# Exercices

- **Création d'une hiérarchie de compte**

- Distinguer les comptes courants et les livrets d'épargne
- Quelle hiérarchie de classes pourrait-on proposer?
- Quelles méthodes existeraient dans une sous-classe mais pas dans l'autre?
- Implémenter cette hiérarchie dans le programme

# Héritage

## Comment mettre en œuvre l'héritage?

- **Une classe abstraite**

- Peut contenir ou hériter de méthodes abstraites (des méthodes sans corps)
- Peut contenir des attributs
- Peut avoir des méthodes normales, avec corps

- **Une classe abstraite ne peut être instanciée**

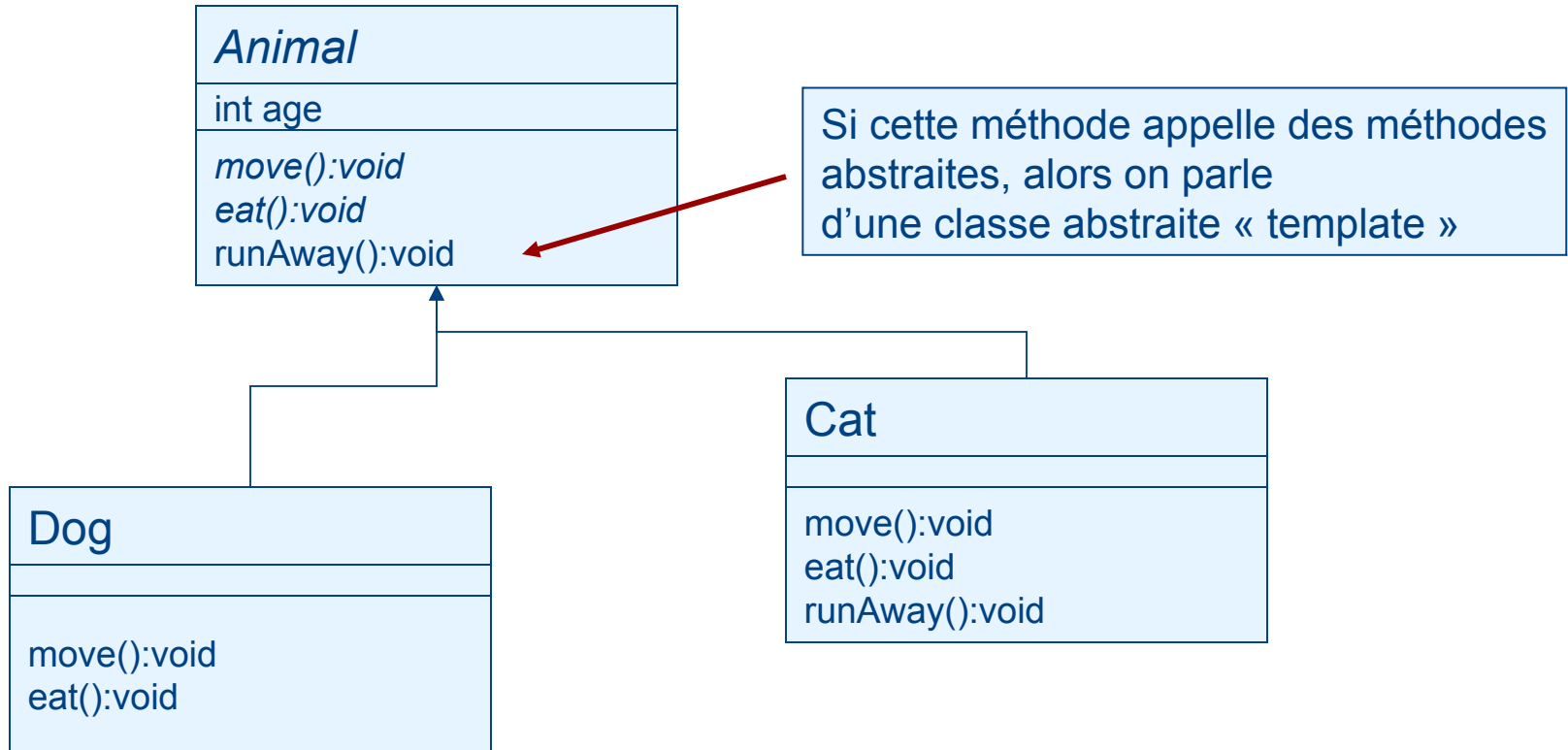
- On peut seulement instancier une sous-classe concrète
- La sous-classe concrète doit donner un corps à toute méthode abstraite
- En d'autres termes, on ne peut jamais faire un « new » sur une classe abstraite
- En revanche on peut parfaitement créer des variables de types abstraits

- **La déclaration d'une classe abstraite contenant une méthode abstraite ressemble à ceci:**

```
abstract class Animal {  
    abstract void move();  
}
```

# Héritage

## Comment mettre en œuvre l'héritage?





# Héritage

## Comment mettre en œuvre l'héritage?

**Mémoire des référents**

**Mémoire des valeurs**

**Mémoire des objets**

Chien monChien

Animal l'AnimalQueTuAsVu

Objet ceQuElleVientDeVoirPasser

ab123d

ab123d

ab123d



# Héritage

## Comment mettre en œuvre l'héritage?

- **Une méthode peut aussi être abstraite**
  - Dans ce cas, elle ne peut pas contenir de corps (elle aura un “;” à la place des accolades)
- **Si une classe comprend une méthode abstraite elle est obligatoirement abstraite elle-même**
- **Si une sous-classe hérite d'une classe abstraite, elle doit redéfinir toutes les méthodes abstraites de sa classe parent (leur donner un corps)**
- **Si elle ne le fait pas, elle sera condamnée à être abstraite elle-même**

# Exercices

- **Création d'une hiérarchie de compte**
  - Rendre la méthode afficheToi de la classe CompteEnBanque
  - Rendre la classe CompteEnBanque abstraite
  - Qu'est-ce que cela implique?

# Exercices Bonus

- Ajouter un nouveau type de compte : un super compte
- Un super compte permet de gérer un ensemble de compte par lot. Chaque super compte est associés à un ensemble de compte en banque.
- Une opération sur un super compte est automatiquement redirigée vers les comptes qu'il contient.
- Exemple : Un dépôt de 1000 euros sur un Super compte qui contient 5 compte est réalisé par 5 dépôt de 250 euros sur les comptes contenus dans le super compte.
- Proposez un diagramme de classe

# Conversion de types (1/2)

## Définition

- Java, langage fortement typé, impose le respect du type d'un objet
- Toutefois, il est possible de convertir le type d'un objet vers un type compatible
  - Un type A est compatible avec un type B si une valeur du type A peut être assignée à une variable du type B
  - Ex: Un entier et un double
- La conversion de type peut se produire
  - Implicitement (conversion automatique)
  - Explicitement (conversion forcée)
- La conversion explicite s'obtient en faisant précéder la variable du type vers lequel elle doit être convertie entre parenthèses (casting)

```
double d = 3.1416;  
int i = (int) d;
```

# Conversion de types (2/2)

## Application

- **Appliquer un opérateur de « cast » au nom d'une variable**
  - Ne modifie pas la valeur de la variable
  - Provoque le traitement du contenu de la variable en tant que variable du type indiqué, et seulement dans l'expression où l'opérateur de cast se trouve
- **S'applique aux variables de types primitifs et aux variables de types de références**
- **Types primitifs:**
  - Seulement vers un type plus large (ou risque de perte de données)
  - Interdit pour le type *boolean*
  - Ex: Short → Integer → Long
- **Types de références:**
  - Vers une classe parent ou une interface implémentée (ou risque d'erreur)
  - Dans ce cas, l'opérateur de cast n'est pas nécessaire (en fonction du contexte)
  - Peuvent toujours être castés vers OBJECT
  - Ex: Voiture → VéhiculesMotorisés → Véhicules → Object

# Héritage

## Redéfinition de méthodes

- La redéfinition de méthode consiste à fournir dans une sous-classe une nouvelle implémentation d'une classe déjà déclarée dans une classe parent
- La redéfinition n'est pas obligatoire !! Mais elle permet d'adapter un comportement et de le spécifier pour la sous-classe
- Pour la réaliser, il suffit de créer dans la sous-classe une méthode avec l'exacte même signature que la méthode originale dans la classe parent
- On peut rappeler la version d'origine (de la classe parent) avec le mot-clé *super*

```
class BankAccount {  
    public void computeInterest(){  
        solde+=300;           //annual gift  
    }  
}  
  
class NormalAccount extends BankAccount {  
    public void computeInterest(){  
        super.computeInterest(); //call the overridden method  
        solde*=1.07;           //annual increase  
    }  
}
```

- ~~Obligation de redéfinir les méthodes déclarées comme abstraites (abstract)~~
- Interdiction de redéfinir les méthode déclarées comme finales (final)

# Héritage

## Le polymorphisme

- **Qu'est-ce que le polymorphisme?**
  - Concept basé sur la notion de redéfinition de méthodes
  - Permet à une tierce classe de traiter un ensemble de classes sans connaître leur nature ultime
  - Consiste à permettre à une classe de s'adresser à une autre en sollicitant un service générique qui s'appliquera différemment au niveau de chaque sous-classe du destinataire du message
  - En d'autres termes, permet de changer le comportement d'une classe de base sans la modifier → Deux objets peuvent réagir différemment au même appel de méthode
  - Uniquement possible entre classes reliées par un lien d'héritage
- **Exemple dans l'écosystème?**
  - Demander à tous les animaux de se déplacer (selon leurs propres règles) en leur adressant un message en tant qu'objets de type "Faune"



# Héritage

## Le polymorphisme

- Utilisation du polymorphisme sur des collections hétérogènes

```
BankAccount[] ba=new BankAccount[5];
```

```
ba[0] = new NormalAccount("Joe",10000);
```

```
ba[1] = new NormalAccount("John",11000);
```

```
ba[2] = new SpecialAccount("Jef",12000);
```

```
ba[3] = new SpecialAccount("Jack",13000);
```

```
ba[4] = new SpecialAccount("Jim",14000);
```

```
for(int i=0;i<ba.length();i++)
```

```
{
```

```
    ba[i].computeInterest();
```

```
}
```

# Exercices

- **Comptes en banque**

- Redéfinir la méthode « afficheToi() » dans les classes CompteCourant et CompteEpargne pour qu'elle précise le type réel de l'objet
- Redéfinir également la méthode « calculInteret() » pour qu'elle crédite le solde d'une prime de fidélité arbitraire sur les comptes épargne mais pas sur les comptes courants.
- Modifier la méthode « main » de la classe Banque pour qu'elle crée des comptes courants et des comptes épargne et non plus de simples « CompteEnBanque »
- Ordonner dans la méthode « main » à tous les comptes de calculer leurs intérêts et de s'afficher
- Que constatez-vous?

# Exercices Super Bonus

- Peut-on éviter de modifier la classe Banque à chaque ajout d'un nouveau type de compte en Banque?
- Pourquoi cela pose-t-il problème?
- Quelle solution élégante pouvez-vous proposer?

# Les interfaces (1/3)

## Définition

- L'interface d'une classe = la liste des messages disponibles  
= signature des méthodes de la classe
- Certaines classes sont conçues pour ne contenir précisément que la signature de leurs méthodes, sans corps. Ces classes ne contiennent donc que leur interface, c'est pourquoi on les appelle elles-mêmes *interface*
- Ne contient que la déclaration de méthodes, sans définition (corps)
- Permet des constantes globales
- Une classe peut implémenter une interface, ou bien des interfaces multiples

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

```
public interface GraphicalObject {  
    public void draw(Graphics g);  
}
```

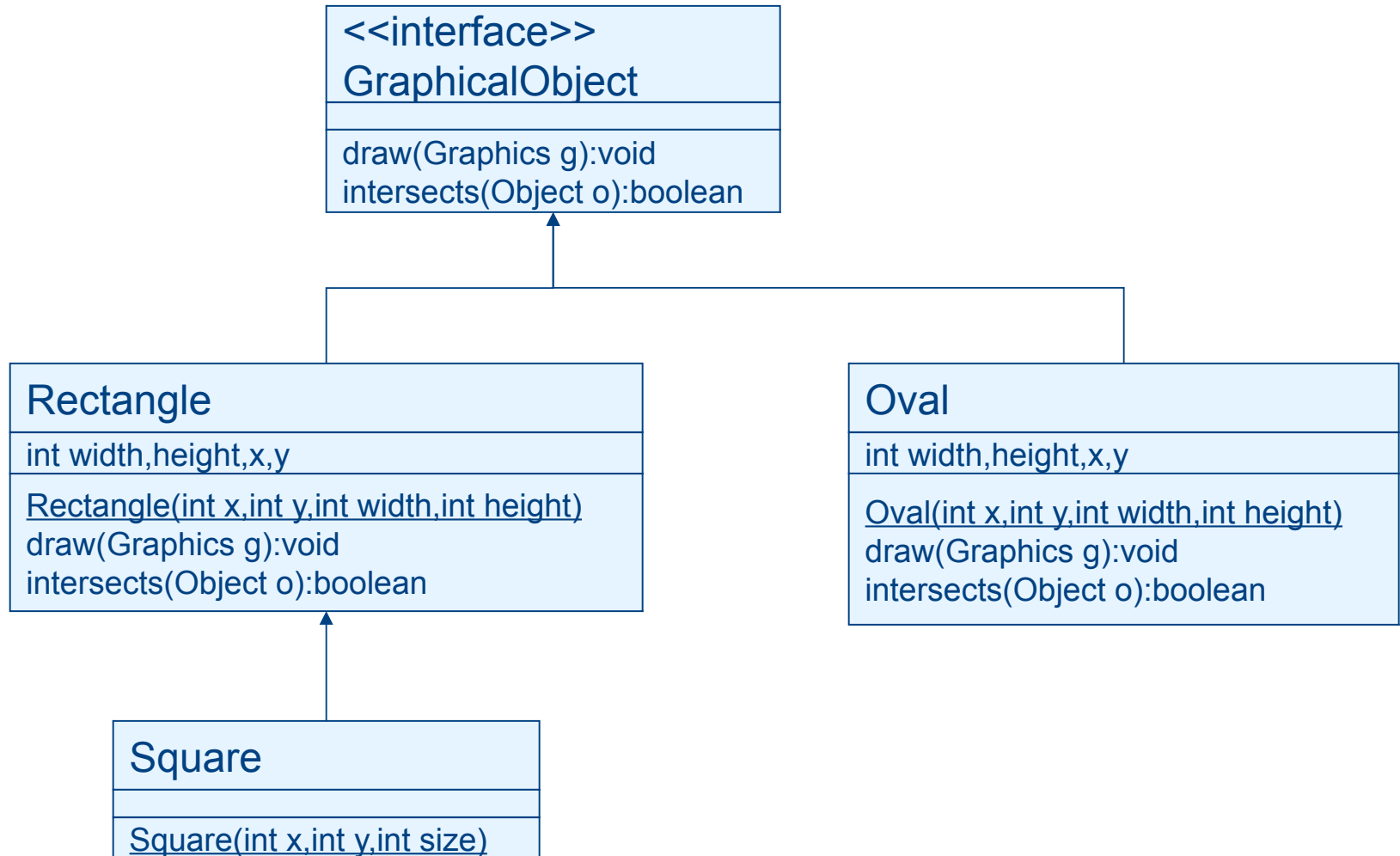
# Les interfaces (2/3)

## Raisons d'être

- Forcer la redéfinition / l'implémentation de ses méthodes
- Permettre une certaine forme de multi-héritage
- Faciliter et stabiliser la décomposition de l'application logicielle
- D'une classe qui dérive d'une interface, on dit qu'elle implémente cette interface
- Le mot clé associé est donc logiquement: `implements`
  
- Exemple:

```
public class monApplet extends Applet implements Runnable, KeyListener
```

# Exemple



# Modéliser les classes avec des diagrammes UML

- **Pourquoi la modélisation?**
  - La conception OO est entièrement bâtie sur une modélisation des objets intervenant dans le problème
  - Avant de programmer quoi que ce soit, il faut donc modéliser les classes et leurs relations au minimum
- **Comment?**
  - Sur base d'UML (Unified Modeling Language)
    - ➔ Notation standardisée pour tout le développement OO de la conception au déploiement
    - ➔ Définition de 9 diagrammes
  - 1. Identifier les classes
    - ➔ Attributs, comportements, polymorphisme
  - 2. Déterminer les relations entre les classes
    - ➔ Associations / Dépendance / Héritage
  - 3. Construire les modèles

# Modéliser les classes avec des diagrammes UML

## Qu'est-ce qu'UML?

- UML est un langage objet graphique - un formalisme orienté objet, basé sur des diagrammes (9)
- UML permet de s'abstraire du code par une représentation des interactions statiques et du déroulement dynamique de l'application.
- UML prend en compte, le cahier de charge, l'architecture statique, la dynamique et les aspects implémentation
- Facilite l'interaction, les gros projets
- Générateur de squelette de code
- UML est un langage PAS une méthodologie, aucune démarche n'est proposée juste une notation



# Modéliser les classes avec des diagrammes UML

## Représenter les relations entre classes

- Les relations d'héritage sont représentées par:

- A  B signifie que la classe A hérite de la classe B

- L'association est représentée par:

- A  B signifie que la classe A envoie des messages à la classe B grâce à ses attributs de type B

- L'agrégation faible est représentée par:

- A  B signifie que la classe A a pour caractéristique un ou plusieurs attributs de type B

- L'agrégation forte (ou composition) est représentée par:

- A  B signifie que les objets de la classe B ne peuvent exister qu'au sein d'objets de type A où ils sont créés

# Les concepts de l'OO

## Les avantages de l'OO

- Les programmes sont plus stables, plus robustes et plus faciles à maintenir car le couplage est faible entre les classes («encapsulation»)
- elle facilite grandement le ré-emploi des programmes: par petite adaptation, par agrégation ou par héritage
- émergence des «design patterns»
- il est plus facile de travailler de manière itérée et évolutive car les programmes sont facilement extensibles. On peut donc graduellement réduire le risque plutôt que de laisser la seule évaluation pour la fin.
- l'OO permet de faire une bonne analyse du problème suffisamment détachée de l'étape d'écriture du code - on peut travailler de manière très abstraite → UML
- l'OO colle beaucoup mieux à notre façon de percevoir et de découper le monde

# Les concepts de l'OO

## Les avantages de l'OO

- Tous ces avantages de l'OO se font de plus en plus évidents avec le grossissement des projets informatiques et la multiplication des acteurs. Des aspects tels l'encapsulation, l'héritage ou le polymorphisme prennent vraiment tout leur sens. On appréhende mieux les bénéfices de langage plus stable, plus facilement extensible et plus facilement ré-employable
- JAVA est un langage strictement OO qui a été propulsé sur la scène par sa complémentarité avec Internet mais cette même complémentarité (avec ce réseau complètement ouvert) rend encore plus précieux les aspects de stabilité et de sécurité

# Les concepts de l'OO

## En résumé

- Tout est un objet
- L'exécution d'un programme est réalisée par échanges de messages entre objets
- Un message est une demande d'action, caractérisée par les paramètres nécessaires à la réalisation de cette action
- Tout objet est une instance de classe, qui est le « moule » générique des objets de ce type
- Les classes définissent les comportements possibles de leurs objets
- Les classes sont organisées en une structure arborescente à racine unique : la hiérarchie d'héritage
- Tout le code des programmes se trouve entièrement et exclusivement dans le corps des classes
- A l'exception toutefois de deux instructions:
  - package → définit l'ensemble auquel la classe appartient
  - import → permet l'utilisation de classes extérieures au package
- UML permet la représentation graphique des applications

# EXERCICES

## Un jeu de rôles orienté objet

- Le programme consiste en un combat jusqu'à la mort opposant gentils et méchants dans lequel chacun des personnages se bat tour à tour.
- Les personnages – chacun caractérisé par un ennemi – sont capables de se battre au moyen de leur arme (caractérisée par un nom et une certaine force). Quand l'arme frappe un ennemi, celui-ci perd une quantité d'énergie équivalente à la force de l'arme. Quand l'énergie d'un personnage atteint zéro, il meurt.
- Mais pour pouvoir frapper leur ennemi, les personnages doivent se trouver au même endroit que lui, ce qui suppose, lorsqu'on leur demande de se battre, qu'ils commencent par s'assurer que leur ennemi est à proximité de leur arme. Dans le cas contraire, ils doivent se déplacer vers leur ennemi en avançant d'un pas dans sa direction.
- Lorsqu'ils se déplacent, les méchants n'ont qu'un objectif: rejoindre leur ennemi le plus vite possible. En revanche, les gentils – moins velléitaires – ont la sollicitude naturelle de s'efforcer plutôt de rejoindre leur ami afin de lui venir en aide (ou de le venger), à moins que leur ennemi soit plus proche et qu'il leur soit alors impossible d'éviter l'affrontement avec lui. Auquel cas, les gentils iront eux aussi au devant de leur ennemi.
- Quand un gentil parvient à rejoindre son ami (mort ou vif), suivant en cela le célèbre adage, il fait sien l'ennemi de son ami voire de l'ami de son ami (si ce dernier est déjà mort). Les gentils conservent ainsi l'obsession de secourir leur ami tant qu'ils sont menacés ou non vengés. Quant aux méchants, ils ne sont satisfaits qu'une fois leur ennemi mort.
- Notons encore que les héros (qui sont des gentils) ont – de par leur ruse – l'avantage sur tous les autres personnages de pouvoir asséner en une fois un certain nombre de coups à leur ennemi.
- A contrario, les volatiles (qui sont des méchants) ont la caractéristique d'être capable de parcourir la distance qui les sépare de leur ennemi en un battement d'ailes.

# EXERCICES

## Un jeu de rôles orienté objet

### Pour cela:

1. Structurer les classes de base et leurs relations (diagramme de classe) pour réaliser ce jeu de rôles.
2. Développer les classes.
3. Créer une classe principale qui créera quelques personnages (gentils, méchants, héros et volatiles), leur ordonnera de s'afficher en début et en fin de programme, et de se battre à tour de rôle (jusqu'à ce qu'il ne reste plus qu'une sorte de personnages (des gentils ou des méchants)).
4. La méthode maininstanciera cette classe principale et lancera les hostilités.

# EXERCICES

## Réservations de spectacles

- **Un programme de gestion des réservations d'une salle de spectacle:**
  - vend des réservations pour une représentation (un certain jour à une certaine heure)
  - d'un spectacle (caractérisé au minimum par son auteur, son titre et son type)
  - à des clients, identifiés par leurs noms, prénoms, adresses et n° de téléphone
  - qui peuvent effectuer autant de réservations qu'ils le souhaitent.
  - Si le client est abonné (auquel cas on conserve son année d'inscription):
    - il bénéficie d'une priorité sur les réservations
    - il achète ses places à crédit (et ne les paie qu'à la fin de l'année)
  - Si le client n'est pas abonné:
    - il paie ses réservations par débit de son porte-monnaie électronique
  - Une fois la réservation enregistrée, le programme permet la sélection des places désirées
  - Une fois le choix des places effectué, un ticket est généré par place

# EXERCICES

## Réservations de spectacles

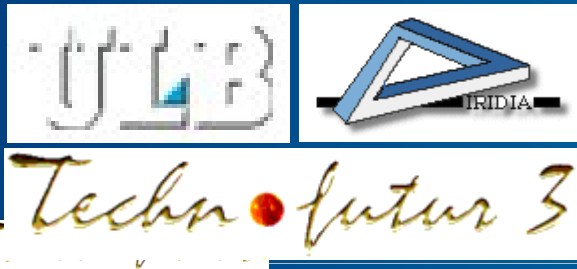
### Pour cela:

1. Structurer les classes de base et leurs relations (diagramme de classe) pour permettre la réservation de places par les clients
2. Développer les classes
3. Créer une classe principale dont la méthode « main » créera quelques clients, quelques spectacles et représentations, passera quelques réservations, en choisira les places et en effectuera le paiement.



# Introduction à Java

## V. Structure des API Java



# Survol du chapitre

- **Introduction**

- Organisation générale des API

- **Packages**

- JAVA
  - JAVAX
  - ORG

# Introduction

## Organisation générale des API Java

- Les différentes API Java sont regroupées en packages
- Ces packages sont eux-mêmes rassemblés dans trois grands groupes, JAVA, JAVAX et ORG
  - JAVA
    - Contient tous les API d'origine de Java 1.0 dont la plupart sont toujours utilisés
  - JAVAX
    - Contient des API réécrits depuis Java 2 et qui sont destinés à les remplacer
  - ORG
    - API provenant de spécifications définies par des organismes internationaux
- La documentation complète des API est toujours disponible sur le site Web de Java
  - <http://java.sun.com/>

# Packages JAVA

Package	Description
java.applet	Classes nécessaires à la création d'applets
java.awt	Abstract Windowing Toolkit → Interfaces graphiques, événements...
java.beans	Pour le développement de composants JavaBeans
java.io	Pour la gestion des IO systèmes (système de fichiers, etc.)
java.lang	Classes fondamentales du langage (toujours importées par défaut)
java.math	Pour les traitements arithmétiques demandant une grande précision
java.net	Pour les connexions et la gestion réseau
java.nio	Définit des tampons
java.rmi	Toutes les classes liées au package RMI (Remote Method Invocation)
java.security	Classes et interfaces du framework de sécurité Java
java.sql	Pour l'accès et la gestion des bases de données → JDBC
java.text	Pour la manipulation de texte, dates, nombres et messages
java.util	Collections, modèle événementiel, dates/heures, internationalisation

# Packages JAVAX

Package	Description
<code>javax.accessibility</code>	Définit un contrat entre l'U.I. et une technologie d'assistance
<code>javax.crypto</code>	Pour les opérations liées à la cryptographie
<code>javax.imageio</code>	Pour la gestion des IO liées aux images
<code>javax.naming</code>	Pour la gestion de la Java Naming and Directory Interface (JNDI)
<code>javax.net</code>	Pour les connexions et la gestion réseau
<code>javax.print</code>	Pour les services liés à l'impression
<code>javax.rmi</code>	Toutes les classes liées au package RMI (Remote Method Invokation)
<code>javax.security</code>	Classes et interfaces du framework de sécurité Java
<code>javax.sound</code>	Pour le développement d'application gérant le son (Midi / Sampled)
<code>javax.sql</code>	Pour l'accès et la gestion des bases de données → JDBC
<code>javax.swing</code>	Interfaces graphiques « légères », identiques sur toutes plateformes
<code>javax.transaction</code>	Exceptions liées à la gestion des transactions
<code>javax.xml</code>	Parseurs et autres classes liées au format XML

# Packages ORG

Package	Description
org.ietf	Framework pour le développement d'applications avec services de sécurité provenant de mécanismes comme le Kerberos
org.omg	Contient tous les packages liés aux spécifications de l'Object Management Group tels que CORBA, IDL et IOP
org.w3c	Contient un parseur DOM pour XML
org.xml	Contient des parseurs SAX pour XML

# Introduction à Java

## VI. Les collections



# Survol du chapitre

- **Introduction**

- Qu'est-ce qu'une Collection?
- Le Java Collections Framework

- **Interfaces**

- Collections → « Set » et « List »
- Maps → « Map » et « SortedMap »
- Itérateurs → « Iterator »
- Compareurs → « Comparable » et « Comparator »

- **Implémentations**

- HashSet et TreeSet
- ArrayList et LinkedList

- **Algorithmes**

- Tri
- Autres: Recherche, Mélange, etc.



# Introduction

## Qu'est-ce qu'un collection?

- **Collection**

- Un objet utilisé afin de représenter un groupe d'objets

- **Utilisé pour:**

- Stocker, retrouver et manipuler des données
- Transmettre des données d'une méthode à une autre

- **Représente des unités de données formant un groupe logique ou naturel, par exemple:**

- Une main de poker (collection de cartes)
- Un dossier d'emails (collection de messages)
- Un répertoire téléphonique (collection de correspondances NOM – N°)

# Introduction

## Java Collections Framework

- **Définit toute la structure des collections en Java**
- **Constitue une architecture unifiée pour**
  - la représentation des collections
  - la manipulation des collections
- **Contient des:**
  - Interfaces
    - Types de données abstraits représentant des collections
    - Permettent de manipuler les collections indépendamment de leur représentation
    - Organisées en une hiérarchie unique
  - Implémentations
    - Implémentations concrètes des interfaces → Structures de données réutilisables
    - Fournies pour accélérer le développement
  - Algorithmes
    - Méthodes standards fournissant des opérations comme le tri, la recherche, etc.

# Interfaces

## Structure

### Il existe 4 groupes d'interfaces liées aux collections

- Collection
    - Racine de la structure des collections
    - Représente un groupe d'objets (dits « éléments »)
    - Peut autoriser ou non les duplicats
    - Peut contenir un ordre intrinsèque ou pas
  - Map
    - Un objet qui établit des correspondances entre des clés et des valeurs
    - Ne peut en aucun cas contenir des duplicats
- Chaque clé ne peut correspondre qu'à une valeur au plus
- Interfaces de comparaison
    - Permettent le tri des objets du type implémentant l'interface
    - Deux versions: « Comparator » et « Comparable »
  - Iterator
    - Permet de gérer les éléments d'une collection

# Interfaces

## Collection (1/2)

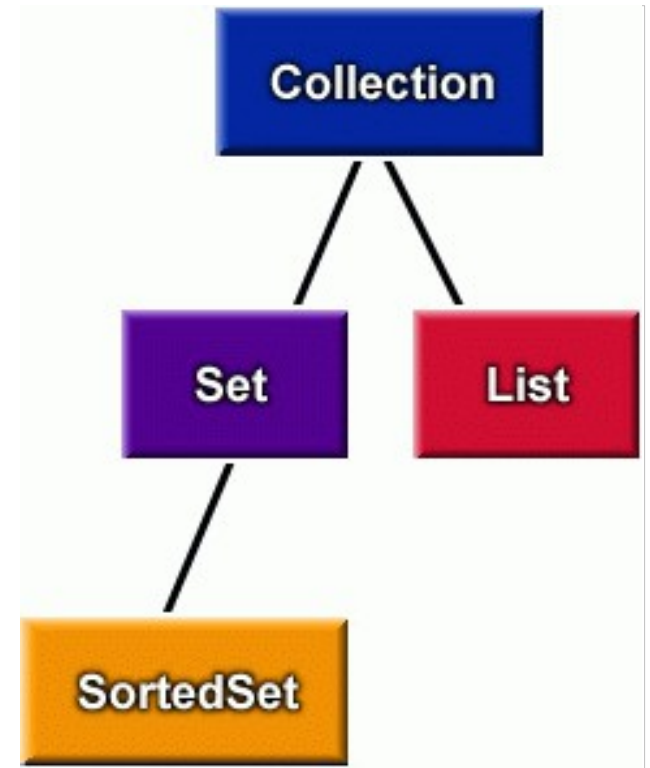
- **La racine de la structure des collections**
  - Sans ordre spécifique
  - Duplicats permis
- **Définit les comportements standards des collections**
  - Vérification du nombre d'éléments  
→ *size()*, *isEmpty()*
  - Test d'appartenance d'un objet à la collection  
→ *contains(Object)*
  - Ajout et suppression d'éléments  
→ *add(Object)*, *remove(Object)*
  - Fournir un itérateur pour la collection  
→ *iterator()*
  - Bulk Operations  
→ *addAll(Collections)*, *clear()*, *containsAll(Collections)*

# Interfaces

## Collection (2/2)

### Trois variantes principales:

- **Set**
  - Duplicats interdits
  - Sans ordre spécifique
- **List**
  - Duplicats permis
  - Contient un ordre spécifique intrinsèque
  - Parfois appelé « Séquences »
  - Permet 2 méthodes d'accès particulières:
    - Positional Access: manipulation basée sur l'index numérique des éléments
    - Search: recherche d'un objet en particulier dans la liste et renvoi de son numéro d'index (sa position dans la liste)
- **SortedSet**
  - Duplicats interdits
  - Contient un ordre spécifique intrinsèque



# Interfaces

## Map

- **Map = Objet qui contient des correspondances Clé – Valeur**
- **Ne peut contenir de doublons (clés sont primaires)**
- **Fournit des opérations de base standards:**
  - `put(key,value)`
  - `get(key)`
  - `remove(key)`
  - `containsKey(key)`
  - `containsValue(value)`
  - `size()`
  - `isEmpty()`
- **Peut générer une collection qui représente les couples Clé – Valeur**
- **Il existe aussi**
  - des « Map » permettant de faire correspondre plusieurs valeurs à chaque clé
  - des « SortedMap » fournissant des « Map » naturellement triés

# Interfaces

## Iterator

- Permet de gérer les éléments d'une collection
- Toute collection peut fournir son « Iterator »
- Permet également de supprimer des éléments d'une collection au cours d'une phase d'itération sur la collection
- Contient 3 méthodes essentielles:
  - `hasNext(): boolean` → Indique s'il reste des éléments après l'élément en cours
  - `next(): Object` → Fournit l'élément suivant de la collection
  - `Remove(): void` → Supprime l'élément en cours

- **Exemple:**

```
static void filter(Collection c) {  
  
    for (Iterator i = c.iterator(); i.hasNext(); )  
  
        if (! cond(i.next()))  
  
            i.remove();  
  
}
```

# Interfaces

## Comparaison (1/4)

### Deux interfaces

- **Comparable**

- Fournit une méthode de comparaison au sein d'une classe
- Impose de redéfinir une seule méthode `public int compareTo(Object o)` qui renvoie un entier:

1 si l'objet courant > l'objet « o » fourni dans la méthode

0 si l'objet courant = l'objet « o » fourni dans la méthode

-1 si l'objet courant < l'objet « o » fourni dans la méthode

- **Comparator**

- Permet de définir une classe servant de comparateur à une autre
- Impose de redéfinir une seule méthode `public int compare(Object o1, Object o2)` qui renvoie un entier:

1 si  $o1 > o2$

0 si  $o1 = o2$

-1 si  $o1 < o2$



# Interfaces

## Comparaison (2/4)

### Exemple d'implémentation de *Comparable*

```
import java.util.*;
public class Name implements Comparable {
    private String  firstName, lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Object o) {
        Name n = (Name) o;
        int lastCmp = lastName.compareTo(n.lastName);
        if(lastCmp==0)
            lastCmp = firstName.compareTo(n.firstName);
        return lastCmp;
    }
}
```

# Interfaces

## Comparaison (3/4)

**Depuis Java 1.5, l'interface peut être générique:**

```
import java.util.*;
public class Personne implements Comparable<Personne> {
    private String  firstName, lastName;

    public Personne(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Personne n) {
        int lastCmp = lastName.compareTo(n.lastName);
        if(lastCmp==0)
            lastCmp = firstName.compareTo(n.firstName);
        return lastCmp;
    }
}
```

# Interfaces

## Comparaison (4/4)


- Les types primitifs contiennent toujours un ordre naturel

Class	Natural Ordering
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dependent lexicographic on pathname.
String	lexicographic
Date	chronological

# Implémentations

## Structure

- Les implémentations sont les types d'objets réels utilisés pour stocker des collections
- Toutes les classes fournies implémentent une ou plusieurs des interfaces de base des collections

 JAVA		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

# Implémentations

## Sets (1/3)

### Deux principales implémentations de l'interface « Set »

- **HashSet (Set)**
  - Plus rapide
  - N'offre aucune garantie en termes de d'ordre
- **TreeSet (SortedSet)**
  - Contient une structure permettant d'ordonner les éléments
  - Nettement moins rapide
  - A n'utiliser que si la collection doit être triée ou doit pouvoir être parcourue dans un certain ordre

# Implémentations

## Sets (2/3)

### Exemple d'utilisation de « HashSet »

```
public class FindDups {  
    public static void main(String args[]) {  
        Set s = new HashSet();  
        for (int i=0; i<args.length; i++)  
            if (!s.add(args[i]))  
                System.out.println("Duplicate  
detected: "+args[i]);  
        System.out.print(s.size()+" distinct words :  
"+s);  
    }  
}
```

**Produira le résultat suivant:**

```
C:> java FindDups i came i saw i left  
Duplicate detected: i  
Duplicate detected: i  
4 distinct words detected: [came, left, saw, i]
```

# Implémentations

## Sets (3/3)

### Exemple d'utilisation de « TreeSet »

```
public class FindDups {  
    public static void main(String args[]) {  
        Set s = new TreeSet();  
        for (int i=0; i<args.length; i++)  
            if (!s.add(args[i]))  
                System.out.println("Duplicate  
detected: "+args[i]);  
        System.out.print(s.size()+" distinct words :  
"+s);  
    }  
}
```

**Produira le résultat suivant:**

```
C:> java FindDups i came i saw i left  
Duplicate detected: i  
Duplicate detected: i  
4 distinct words detected [came, i, left, saw]
```

# Implémentations

## Lists (1/2)

### Deux principales implémentations de l'interface « List »

- **ArrayList (et Vector)**

- La plus couramment utilisée
- Offre un accès positionnel à vitesse constante → Particulièrement rapide

- **LinkedList**

- A utiliser pour
  - Ajouter régulièrement des éléments au début de la liste
  - Supprimer des éléments au milieu de la liste en cours d'itération
- Mais plus lent en termes d'accès positionnel



# Implémentations

## Lists (2/2)

### Opérations spécifiques aux listes:

- **Obtenir la position d'un objet**
  - *indexOf(Object o): int*
  - *lastIndexOf(Object o): int*
- **Récupérer l'objet en position i**
  - *get(int i)* → Renvoie un objet de type « Object » → Devra être converti (cast)
- **Placer un objet à une certaine position**
  - *set(int i, Object o)*
- **Supprimer l'objet en position i**
  - *remove(int i)*
  - *removeRange(int fromIndex, int toIndex)*

# Algorithmes

## Tri

**On peut trier un tableau/collection au moyen de méthodes simples:**

- Trier un tableau → méthode « sort » de la classe « Arrays »

- `Arrays.sort(<type>[])`

- Trier une collection → méthodes « sort » de la classe « Collections »

- Ne fonctionne qu'avec les collections dérivant de « List »

- Si les éléments de la collection sont comparables (implémentent l'interface « Comparable »)

- `Collections.sort(List)`

- Si les éléments de la collection ne sont pas comparables, il faut alors indiquer quelle classe servira de comparateur

- `Collections.sort(List, Comparator)`

# Algorithmes

## Autres

- **D'autres opérations sont fournies par le Java Collections Framework:**
  - Recherche binaire
    - *Collections.binarySearch(liste, clé)*
  - Mélange
    - *Collections.shuffle(liste)*
  - Inversion de l'ordre
    - *Collections.reverse(liste)*
  - Réinitialisation des éléments (remplace tous les éléments par l'objet spécifié)
    - *Collections.fill(liste, objetParDefaut)*
  - Copie des éléments d'une liste dans une autre
    - *Collections.copy(listeSource, listeDestination)*
  - Recherche d'extrema
    - Sur base de la position des éléments *min(liste)* et *max(liste)*
    - Sur base d'un comparateur *min(liste, comparateur)* et *max(liste, comparateur)*

# Collections

## Cas pratique: les ArrayList (typées)

- **ArrayList:**

- Une classe prédéfinie représentant une collection d'objets (un peu comme les Arrays [])
- La taille de cette collection n'est pas prédéfinie et se modifie dans le temps:
  - Au départ la collection est de dimension nulle
  - Chaque fois qu'on ajoute un objet dans la collection, le vecteur s'agrandit
  - Chaque fois qu'on supprime un objet de la collection, le « trou » est supprimé et tous les autres objets sont décalés « vers la gauche »
  - ➔ La taille de la collection correspond toujours au nombre d'objet qu'elle contient
- Méthodes pour manipuler une collection:
  - Créer une collection: `ArrayList<Classe> al = new ArrayList<Classe>();`
  - Connaître sa taille: `al.size();`
  - Ajouter un objet: `al.add(new Classe(...));`
  - Récupérer le ième objet: `al.get(i);`
  - Pour supprimer le ième objet: `al.remove(i);`
  - Pour vider complètement la collection: `al.clear();`

# Exercice

## Spectacles

- Remplacez tous les « Array » par des « ArrayList »
- Implémentez l'interface Comparable au sein de la classe « Ticket » et afin de permettre le tri des objets sur base de la représentation et du numéro de la place et redéfinissez à cette fin la méthode compareTo(Object o)
- Ordonnez le tri de la collection de tickets dans la méthode main et vérifiez que l'ordre des tickets a bien été adapté en les affichant

# Introduction à Java

## VII. Gestion des exceptions



# Survol du chapitre

- Introduction
- Hiérarchie des exceptions
- Traitement des exceptions
  - Interception d'exceptions: bloc *try* – *catch* – *finally*
  - Lancement (génération) par une méthode: *throws* et *throw*

# Introduction

## La gestion des exceptions en Java

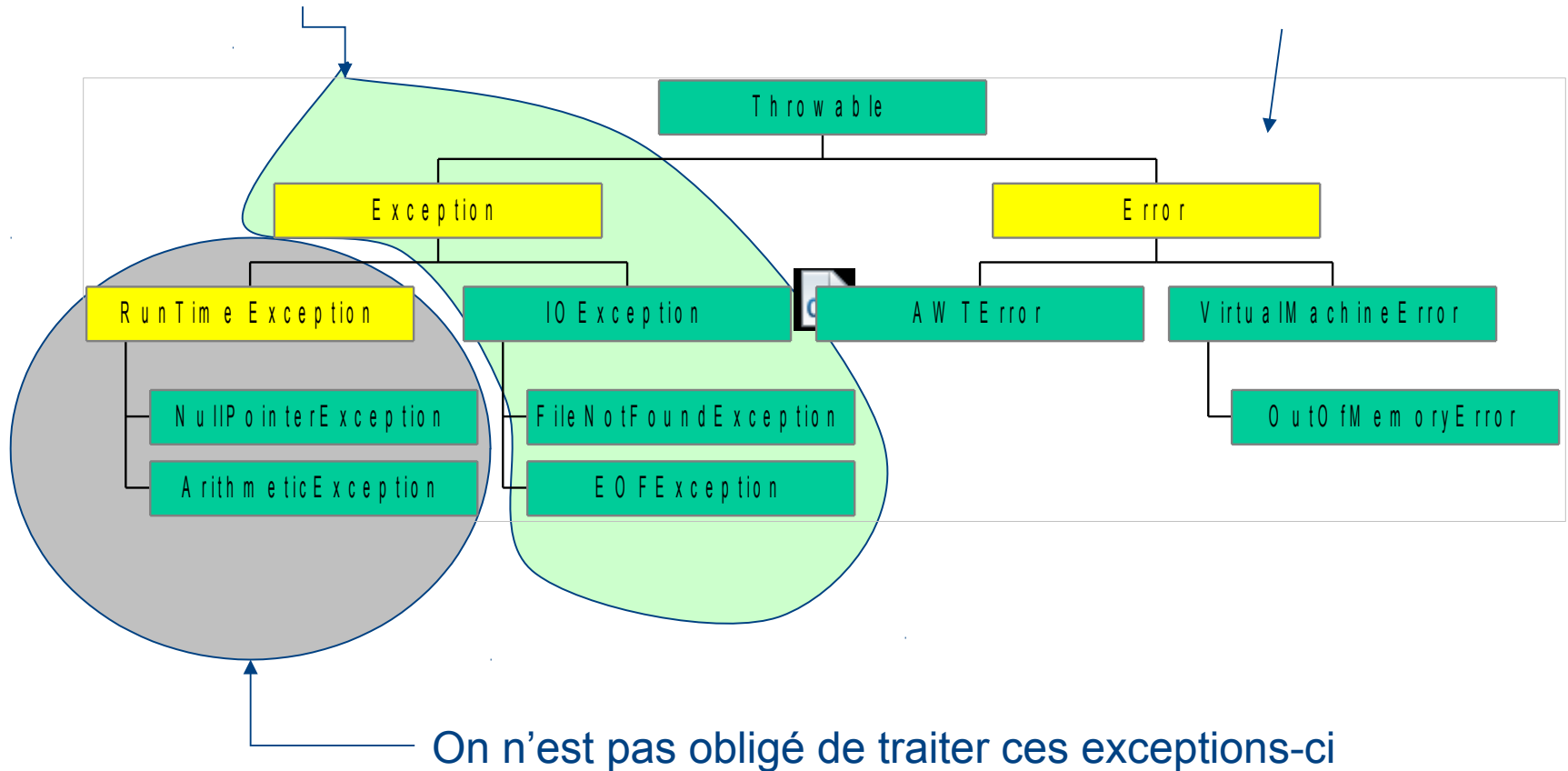
- S'approche du C++
- Des erreurs surviennent dans tout programme
- Distinction entre Exception et Error: deux classes apparentées
- La classe Exception traite les erreurs prévisibles qui apparaissent dans l'exécution d'un programme:
  - Panne du réseau
  - Fichier inexistant
  - Problème propre à la logique « business »
- La classe Error traite les conditions sérieuses que le programmeur n'est pas censé traiter



# Hiérarchie des exceptions

On doit traiter ces exceptions-ci

On ne peut pas traiter les « Error »



# Traitement des exceptions

## Principes

- **Le traitement des exceptions contient deux aspects:**
  - L'interception des exceptions
    - Utilisation du bloc *try – catch – finally* pour récupérer les exceptions
    - Et réaliser les actions nécessaires
  - Le lancement (la génération) d'exceptions
    - Automatiquement par l'environnement run-time ou la machine virtuelle pour certaines exceptions prédéfinies par Java
    - Explicitement par le développeur dans une méthode avec « throws » et « throw » (en tout cas pour les exceptions créées par le développeur)

# Traitement des exceptions

## Interception par bloc *try – catch – finally* (1/2)

```
try
{
    // quelques actions potentiellement risquées
}
catch(SomeException se)
{
    // que faire si une exception de ce type survient
}
catch(Exception e)
{
    // que faire si une exception d'un autre type survient
}
finally
{
    // toujours faire ceci, quelle que soit l'exception
}
```

# Traitement des exceptions

## Interception par bloc *try – catch – finally* (2/2)

### Implémentation

```
public ExampleException() {  
    for(int i=0;i<3;i++) {  
        test[i]=Math.log(i);  
    }  
    try {  
        for(int i=0;i<4,i++) {  
            System.out.println("log("+i+") = "+test[i]);  
        }  
    } catch (ArrayIndexOutOfBoundsException ae) {  
        System.out.println(« Arrivé à la fin du tableau »);  
    }  
    System.out.println(« Continuer le constructeur »);  
}
```

# Traitement des exceptions

## Lancement avec les mots-clés *throws* et *throw* (1/4)

- Si une exception peut survenir, mais que la méthode n'est pas censée la traiter elle-même, il faut en « lancer » une instance
- Il faut préciser que la méthode peut lancer ces exceptions  
→ ajouter une clause *throws* à la signature de la méthode

```
public void writeList() {  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
    for (int i = 0; i < vector.size(); i++)  
        out.println("Valeur = " + vector.elementAt(i));  
}
```

- Peut lancer une *IOException* → doit être attrapée
- Peut lancer une *ArrayIndexOutOfBoundsException*

```
public void writeList() throws IOException,  
    ArrayIndexOutOfBoundsException {
```

# Traitement des exceptions

## Lancement avec les mots-clés *throws* et *throw* (2/4)

- Une méthode avec une propriété *throws* peut lancer des exceptions avec *throw*
- Toute exception ou erreur lancée pendant l'exécution provient d'un *throw*
- Fonctionne avec les exceptions qui héritent de *Throwable* (la classe de base)
- Le développeur peut créer de nouvelles classes d'exceptions et les lancer avec *throw*

```
class MyException extends Exception {  
    MyException(String msg) {  
        System.out.println("MyException lancée, msg =" + msg);  
    }  
}  
  
void someMethod(boolean flag) throws MyException {  
    if(!flag) throw new MyException («someMethod»);  
    ...  
}
```

# Traitement des exceptions

## Lancement avec les mots-clés *throws* et *throw* (3/4)

- Une fois l'exception lancée avec *throw*, il faut soit l'attraper (avec *catch*), soit la re-lancer. Si vous la re-lancez, votre méthode doit le déclarer

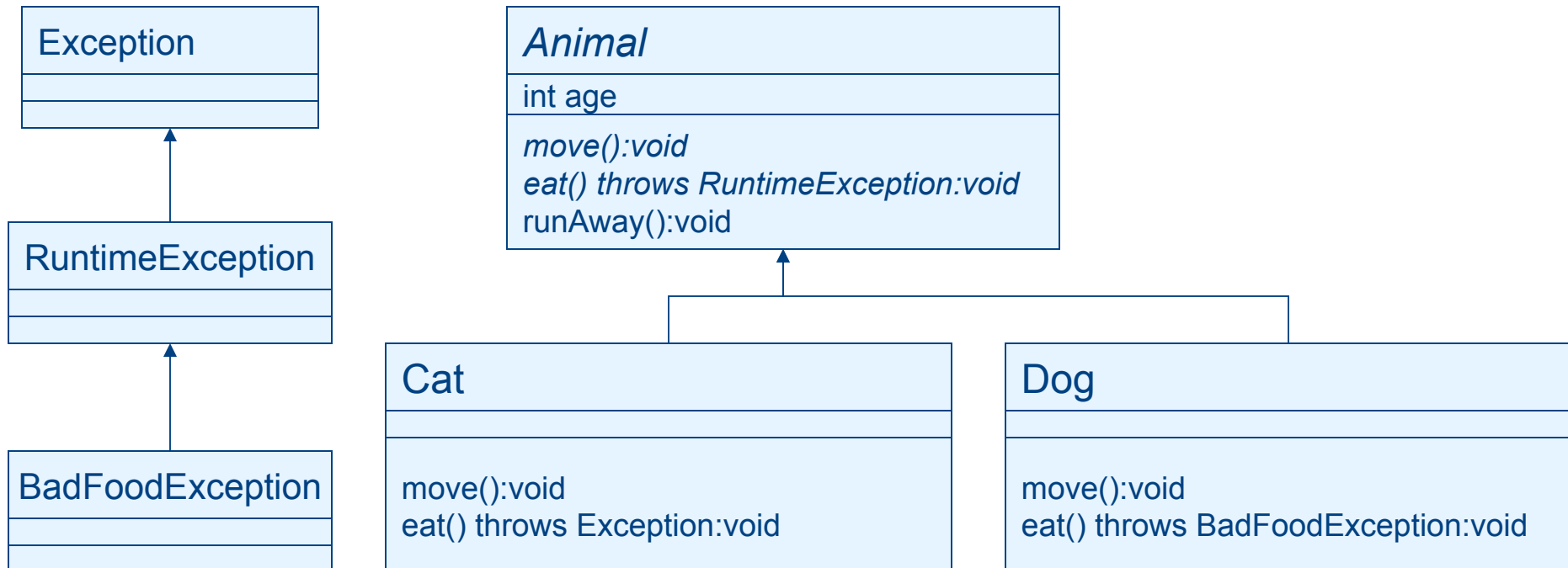
```
public void connectMe(String serverName) throws ServerTimeoutException {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        throw new ServerTimeoutException("Connection impossible", 80);
    }
}
```

```
public void connectMe(String serverName) {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        try{ throw new ServerTimeoutException("Connection impossible ", 80);
        } catch(ServerTimeoutException stoe){ }
    }
}
```

# Traitement des exceptions

## Lancement avec les mots-clés *throws* et *throw* (4/4)

- Si la méthode à redéfinir lance une exception
  - ➔ On ne peut pas lancer une « autre » exception
  - ➔ Mais on peut utiliser l'héritage





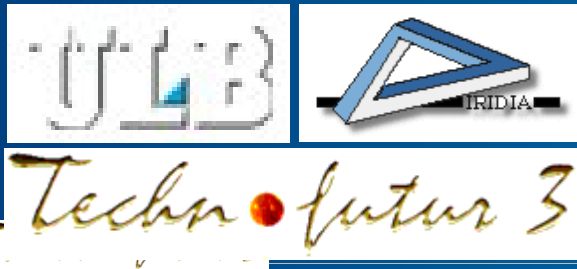
# Exercice

## Spectacles

- Générer une exception quelconque et l'intercepter dans une méthode au choix au moyen d'un bloc *try - catch*
- Créer une nouvelle exception « PlaceDejaPriseException », qui devra être lancée chaque fois que l'utilisateur tente créer un ticket alors que la place n'est plus disponible pour la représentation choisie
- Gérer l'exception par renvoi (*throws*) ou interception (*try-catch*)

# Introduction à Java

## VIII. Permanence des objets



- **Nous allons dans la suite découvrir les 3 mécanismes de permanence: Stream (fichier de bytes), Sérialisation, Base de données relationnelles.**
- **Les solutions à ce problème sont encore très controversées.**
- **Nous sauverons nos comptes en banque de ces trois manières**

# Les Stream

```
import java.io.*;

public void sauveDonnees(PrintWriter maSortie) { /* écrit les données sur le fichier */
    maSortie.println(toString());
    maSortie.println(getVitX());
    maSortie.println(getVitY());
    maSortie.println(getEnergie());
    maSortie.println();
}

public void litLesDonnees(BufferedReader monEntree) { /* lit les données du fichier */
try{
    String s=null;
    monEntree.readLine(); /* saute une ligne */
    s = monEntree.readLine(); /* lit une ligne */
    int vitx = Integer.parseInt(s); /* caste en entier */
    s = monEntree.readLine();
    int vity = Integer.parseInt(s);
    s = monEntree.readLine();
    double energie = Double.parseDouble(s); /* caste en double */
    setVitesse(vitx, vity);
    setEnergie(energie);
    monEntree.readLine(); /* saute le type */
} catch (IOException e) {System.out.println(e); }
}
}
```

```
private FileOutputStream fos; /* la connexion au fichier en écriture */
private PrintWriter maSortie; /* afin d'utiliser "println()" */
private FileReader fr; /* la connexion au fichier en lecture de texte */
private BufferedReader monEntree; /* lecture "temporisée" */

public Jungle(int largeur, int hauteur) {
    BufferedReader monEntree = null;
    try {
        fr = new FileReader("leFichierPredateur.txt"); /* connexion-fichier en lecture de texte */
        monEntree = new BufferedReader(fr); /* flux filtré pour la lecture temporisée */
    } catch (IOException e) {e.getMessage();}
    For (int i=0; i<lesLions.length; i++)
        lesLions[i].litLesDonnees(monEntree);
    try{
        monEntree.close(); /* fermeture du tube */
    } catch (Exception e) {System.out.println(e);}
    try {
        fos = new FileOutputStream("leFichierPredateur.txt"); /* connexion sur le fichier */
    } catch (IOException e) {e.getMessage();}
    maSortie = new PrintWriter(fos, true); /* flux filtré pour l'écriture de ligne de texte */
    for(int i=0; i<lesLions.length; i++)
        lesLions[i].sauveDonnees(maSortie);
    maSortie.close(); /* fermeture indispensable du tube */
}
```

- **Pratique qui sonne faux en OO**
- **L'organisation en objet est perdue**
- **Les connexions entre objets sont perdues**
- **→ Sérialisation**

# La sérialisation

```
import java.io.*;

public class Predateur extends Faune implements Serializable {
    private Proie[] lesProies;

    public void sauveDonnees(ObjectOutputStream oos) {
        try{
            oos.writeObject(this); /* on sauve l'objet */
        } catch (IOException e) { System.out.println(e); }
    }

    public void litLesDonnees(ObjectInputStream ois) {
        Predateur unPredateur;
        try{ /* on lit l'objet puis on le caste dans la classe adéquate */
            unPredateur = (Predateur)ois.readObject();
        } catch (Exception e) { System.out.println(e); }
    }
}
```

```

try {
    fis = new FileInputStream ("leFichierPredateur.ser");
    ois = new ObjectInputStream(fis);
    /* on installe un tube à lecture d'objets sur le fichier */
} catch (IOException e) {e.getMessage();}
for (int i=0; i<lesAnimaux.length; i++)
    lesLions[i].litLesDonnees(ois);
try {
    ois.close(); /* on ferme le tube */
} catch (Exception e) {}
try {
    fos = new FileOutputStream("leFichierPredateur.ser");
    oos = new ObjectOutputStream(fos);
    /* on installe un tube à écriture d'objets sur le fichier */
} catch (IOException e) {e.getMessage();}
for(int i=0; i<lesLions.length; i++)
    lesLions[i].sauveDonnees(oos);
try {
    oos.close(); /* on ferme le tube */
} catch(Exception ex) { System.out.println(ex.getMessage()); }
}

```



- C'est la version la plus simple
- C'est aussi celle qui sonne le plus OO
- A partir du premier objet, on peut sauvegarder et relire tous ceux qui lui sont connectés → sérialisation.
- Mais le fichier sérialisé est illisible en dehors de l'exécutable Java. Possibilité récente de sérialiser en soap.
- → Bases de données relationnelles.

# Connexion DB

```
import java.io.*;
import java.sql.*;
public class Predateur extends Faune {
public void litLesDonnees(Connection connexion, int i) {
try {
    Statement ordre = connexion.createStatement(); /* établit la connexion */
    String requete = "SELECT * FROM Predateur WHERE IdPredateur =" + i;
                    /* la requête SQL */
    ResultSet resultats= ordre.executeQuery(requete);
        /* envoie la requête sur la connexion */
    resultats.next(); /* itère le résultat */
    setVitX(resultats.getInt("vitx"));
    setVitY(resultats.getInt("vity"));
    setEnergie(resultats.getDouble("energie"));
    ordre.close();
} catch (Exception e) { System.out.println(e.getMessage()); }
}
```

```
public void sauveDonnees(Connection connexion, int i) {  
    try {  
        Statement ordre = connexion.createStatement();  
        /* la requête SQL */  
        String requete = "UPDATE Predateur SET vitx= " + getVitX() + " ,vity= " + getVitY()  
            + " ,energie= " + getEnergie() + " WHERE IdPredateur =" + i;  
        System.out.println("j'execute la requete");  
        int resultat = ordre.executeUpdate(requete); /* envoie la requête sur la connexion */  
        if (resultat == 1)  
            System.out.println("la base de donnees est mise a jour");  
        else  
            System.out.println("c'est rate");  
        ordre.close();  
    } catch (Exception e) { System.out.println(e.getMessage()); }  
}
```

```

import java.io.*;
import java.sql.*;
public class Jungle {
    private Connection connexion;

    public Jungle(int largeur, int hauteur) {
    try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); /* va chercher le pilote adéquat */
        /* établit la connexion sur la base */
        connexion = DriverManager.getConnection("jdbc:odbc:Ecosysteme");
    } catch (Exception se) {System.out.println("Connexion Impossible" + se.getMessage());}
    for (int i=0; i<lesAnimaux.length; i++)
        lesLions[i].litLesDonnees(connexion,i);

    for(int i=0; i<lesLions.length; i++)
        lesLions[i].sauveDonnees(connexion, i);
    }
}

```

- Solutions assez lourdes
- Basées sur ODBC
- Reste entier le problème du mapping OO/relationnel
- SQL, OQL, SQL3
- Base de données OO ??

# Introduction à Java

## IX. Interrogation de bases de données: SQL



# Interrogation de bases de données avec SQL

- Qu'est-ce que SQL?
- Les types de requêtes
- Structure des requêtes

# Interrogation de bases de données avec SQL

## Qu'est-ce que SQL?

- Structured query language (SQL), ou *langage structuré de requêtes*
- Un pseudo-langage informatique (de type requête) standard et normalisé
- Destiné à interroger ou manipuler une base de données relationnelle
- Sous réserve de quelques spécificités, toutes les bases de données relationnelles du marché peuvent exécuter du code SQL
- Les requêtes écrites en SQL sont donc normalement indépendantes de la base de données à laquelle elles s'adressent



# Interrogation de bases de données avec SQL

## Qu'est-ce que SQL?

- **SQL (Langage de requêtes structuré) est :**
  - un langage de définition de données
    - Création des tables et des relations
  - un langage de manipulation de données
    - Consultation, insertion, modification de tuples
  - un langage de protection de données
    - Définition des permissions au niveau des utilisateurs
  - pour les bases de données relationnelles.

# Interrogation de bases de données avec SQL

## Types de requêtes SQL

- **Requêtes sélection**

- SELECT → Pour extraire des données de la BD

- **Requêtes action**

- INSERT → Pour ajouter des enregistrements dans une table
- UPDATE → Pour modifier des enregistrements dans une table
- DELETE → Pour supprimer des enregistrements dans une table

- **Requêtes structurelles (sur les tables elles-mêmes)**

- INSERT TABLE → Créer une nouvelle table
- ALTER TABLE → Modifier une table existante
- DROP TABLE → Supprimer une table existante
- Etc.

# Interrogation de bases de données avec SQL

## Structure des requêtes > SELECT

- **Requêtes SELECT**

- Corps obligatoire:
  - SELECT [colonnes à extraire]
  - FROM [tables dont les colonnes seront extraites]
- Options:
  - WHERE [critères de sélection]
  - ORDER BY [champs sur lesquels trier les enregistrements]

- **Exemples:**

- SELECT \* FROM livres
- SELECT nom, prenom, adresse FROM clients WHERE ville='Paris'

# Interrogation de bases de données avec SQL

## Structure des requêtes > SELECT

Marque	Modele	Serie	Numero
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

SELECT Modele, Serie FROM VOITURES

Modele	Serie
18	RL
Kangoo	RL
Kangoo	RL
106	KID
309	chorus
Escort	Match

SELECT DISTINCT Modele, Serie FROM VOITURES

Modele	Serie
18	RL
Kangoo	RL
106	KID
309	chorus
Escort	Match

# Interrogation de bases de données avec SQL

## Structure des requêtes > SELECT

Marque	Modele	Serie	Numero	Compteur
Renault	18	RL	4698 SJ 45	123450
Renault	Kangoo	RL	4568 HD 16	56000
Renault	Kangoo	RL	6576 VE 38	12000
Peugeot	106	KID	7845 ZS 83	75600
Peugeot	309	chorus	7647 ABY 82	189500
Ford	Escort	Match	8562 EV 23	

*SELECT \* FROM OCCAZ  
WHERE (Compteur < 100000)*

Marque	Modele	Serie	Numero	Compteur
Renault	Kangoo	RL	4568 HD 16	56000
Renault	Kangoo	RL	6576 VE 38	12000
Peugeot	106	KID	7845 ZS 83	75600
Ford	Escort	Match	8562 EV 23	

Marque	Modele	Serie	Numero	Compteur
Renault	Kangoo	RL	4568 HD 16	56000
Peugeot	106	KID	7845 ZS 83	75600

*SELECT \* FROM OCCAZ  
WHERE (Compteur <= 100000) AND (Compteur  
>= 30000)*

# Interrogation de bases de données avec SQL

## Structure des requêtes > SELECT

- Lorsqu'un champ n'est pas renseigné, le SGBD lui attribue une valeur spéciale que l'on note NULL. La recherche de cette valeur ne peut pas se faire à l'aide des opérateurs standards, il faut utiliser les prédicats IS NULL ou bien IS NOT NULL.

```
SELECT * FROM OCCAZ  
WHERE Compteur IS NULL
```

Marque	Modele	Serie	Numero	Compteur
Ford	Escort	Match	8562 EV 23	

# Interrogation de bases de données avec SQL

## Structure des requêtes > SELECT

- **Trier les données:**

- La clause ORDER BY est suivie des mots clés ASC ou DESC, qui précisent respectivement si le tri se fait de manière croissante (par défaut) ou décroissante.
- Exemple:  
SELECT \* FROM VOITURE ORDER BY Marque ASC, Compteur DESC

Marque	Modele	Serie	Numero	Compteur
Ford	Escort	Match	8562 EV 23	
Peugeot	309	chorus	7647 ABY 82	189500
Peugeot	106	KID	7845 ZS 83	75600
Renault	18	RL	4698 SJ 45	123450
Renault	Kangoo	RL	4568 HD 16	56000
Renault	Kangoo	RL	6576 VE 38	12000

# Interrogation de bases de données avec SQL

## Structure des requêtes

- **Requêtes INSERT**

- Corps obligatoire:
  - INSERT INTO [table](champ1, champ2, champ3, ...)
  - VALUES(valeur1, valeur2, valeur3, etc.)

- **Exemples:**

- INSERT INTO clients(nom, prenom, adresse, ville)  
VALUES('Bersini', 'Hugues', 'Avenue Roosevelt 50', 'Bruxelles')
- INSERT INTO livres(titre, auteur, anneeDeParution)  
VALUES ("L'orienté objet", "Bersini", 2004)
- INSERT INTO livres(auteur, titre)  
VALUES ("Bersini", "L'orienté objet")



# Interrogation de bases de données avec SQL

## Structure des requêtes

- **Requêtes UPDATE**

- Corps obligatoire:
  - UPDATE [table]
  - SET champ1 = newValeur1, champ2 = newValeur2, etc.
- Options:
  - WHERE [critères de sélection]

- **Exemples:**

- UPDATE clients SET ville = 'Brussels' WHERE ville = 'Bruxelles'
- UPDATE livres SET bestSeller = 'Oui' WHERE auteur = 'Bersini'

# Interrogation de bases de données avec SQL

## Structure des requêtes

- **Requêtes DELETE**

- Corps obligatoire:
  - DELETE FROM [table]
- Options:
  - WHERE [critères de sélection]

- **Exemples:**

- DELETE FROM clients
  - ➔ Efface tous les enregistrements de la table!!!
- DELETE FROM livres WHERE auteur = 'Bersini'

# Interrogation de bases de données avec SQL

## Structure des requêtes > Opérateurs conditionnels

- **Opérateurs logiques**

- AND
- OR
- NOT

- **Opérateurs arithmétiques:**

- +
- -
- \*
- /
- %
- &
- |

- **Comparateurs arithmétiques**

- =
- !=
- >
- <
- >=
- <=
- <>
- !>
- !<

- **Comparateurs de chaîne:**

- IN
- BETWEEN
- LIKE

# Introduction à Java

## X. Connecter une application Java à une BD: JDBC



# Connexion aux bases de données en Java

- Le défi
- La solution: JDBC
- Mise en œuvre

# Connexion aux bases de données en Java

## Le défi

- Une application informatique a le plus souvent (surtout en informatique d'entreprise) besoin d'interagir avec une ou plusieurs bases de données
- Exemples:
  - L'application qui permet à un agent bancaire d'effectuer des opérations sur le compte de ses clients doit obtenir et enregistrer des données sur la base de données de la banque
  - Les terminaux Bancontact ou Visa doivent envoyer une opération à la banque de données des comptes ou cartes et obtenir une confirmation
  - Un call center dans un service à la clientèle doit pouvoir obtenir et modifier les données des clients, de leurs factures, etc.

# Connexion aux bases de données en Java

## Le défi

- **Or les bases de données sont en général centralisées et servent un grand nombre de « clients »**
  - Comment stocker l'information à un seul endroit et permettre à de multiples utilisateurs d'y accéder en même temps et de différentes façons?
- **Les bases de données se contentent donc en général de stocker l'information, éventuellement de la valider, mais pas de la traiter**
- **Et elles se mettent à la disposition des applications pour leur fournir l'information demandée ou pour stocker leurs modifications (en SQL)**

# Connexion aux bases de données en Java

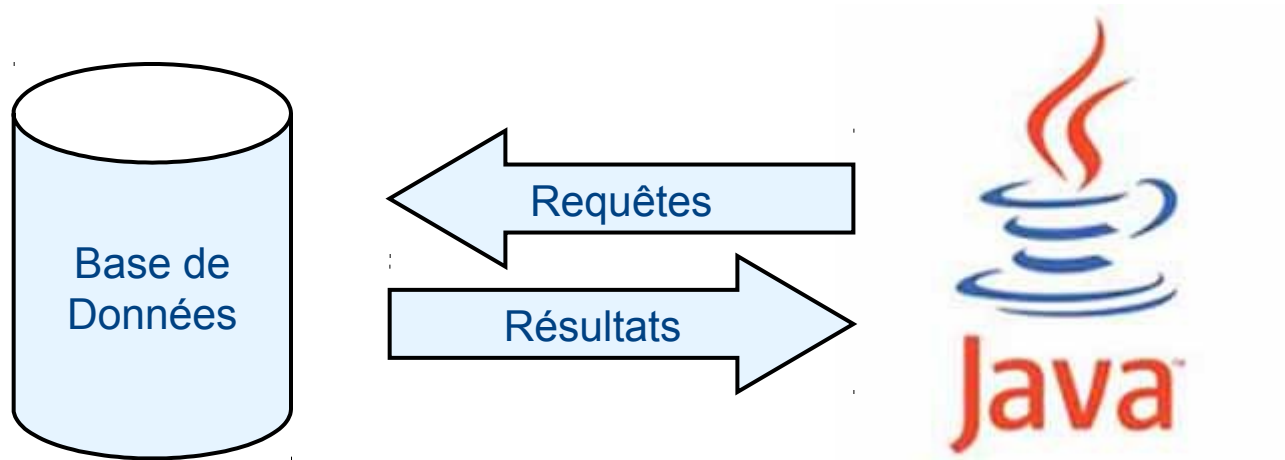
## Le défi

- **Comment une application Java peut-elle interagir avec une base de données externe (par exemple MS Access)?**
  - ➔ Logiquement, en SQL
- **Mais comment permettre à l'application de se connecter à la base de données, de lui envoyer des requêtes et d'en récupérer le résultat?**
  - ➔ En rendant la base de données Access disponible pour interrogation par des applications externes (cf. procédure ODBC)
  - ➔ Et en établissant une connexion entre l'application et cette source ODBC



# Connexion aux bases de données en Java

## Le défi



# Connexion aux bases de données en Java

## La solution: JDBC

- Dans le monde Windows, il existe une plateforme baptisée ODBC
  - ODBC = Open DataBase Connectivity
- Il s'agit d'une interface standard permettant de mettre des bases de données à disposition des applications Windows
- Chaque PC équipé de Windows possède une liste des "DataSources" ODBC qui pointent chacune vers une base de données
- ODBC "connaît" chaque type de base de données Windows et sait "dans quel langage" lui parler
- Autrement dit, il existe un "pilote ODBC" pour la plupart des bases de données qui tournent sous Windows.
  - Microsoft Access en fait partie
- Pour qu'un programme puisse accéder à une base de données, il suffit donc que celle-ci soit déclarée dans la liste des sources ODBC

# PROJET

- **Déclarez votre base de données MS Access comme une source de données ODBC**
  1. Ouvrez le menu « Démarrer » puis « Data Sources ODBC »
  2. Cliquez sur « Ajouter »
  3. Choisissez « Microsoft Access Driver (\*.mdb) » dans la liste
  4. Cliquez sur Finish
  5. Donnez un nom (simple) à votre source de données
  6. Cliquez sur « Select... »
  7. Repérez et sélectionnez votre base de données et cliquez sur OK  
(Notez que votre BD pourrait très bien se trouver sur un autre ordinateur)
  8. Cliquez sur OK pour fermer le panneau de contrôle ODBC
- **Votre base de données MS Access est maintenant reconnue comme une source accessible via ODBC**

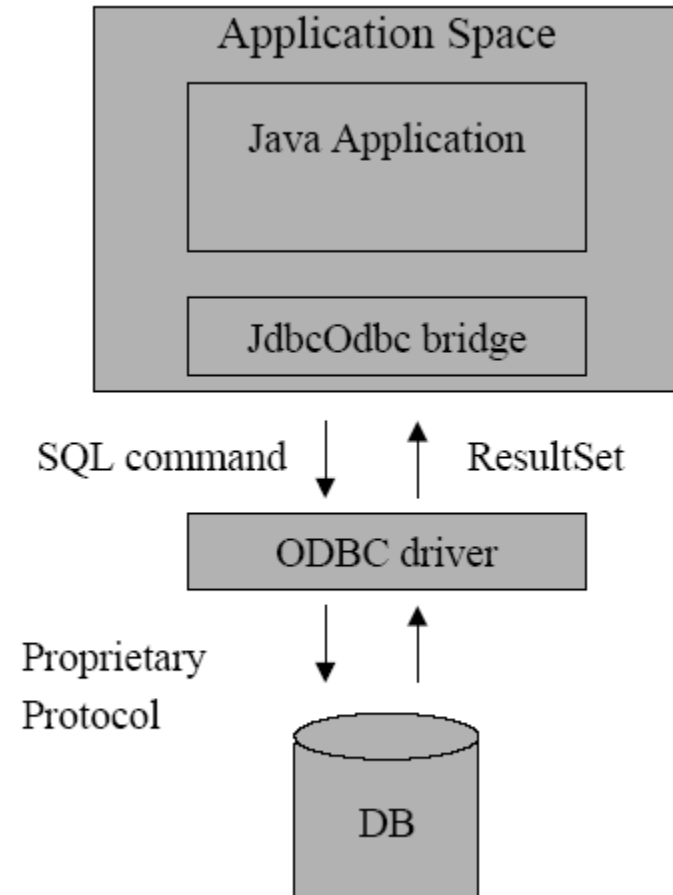
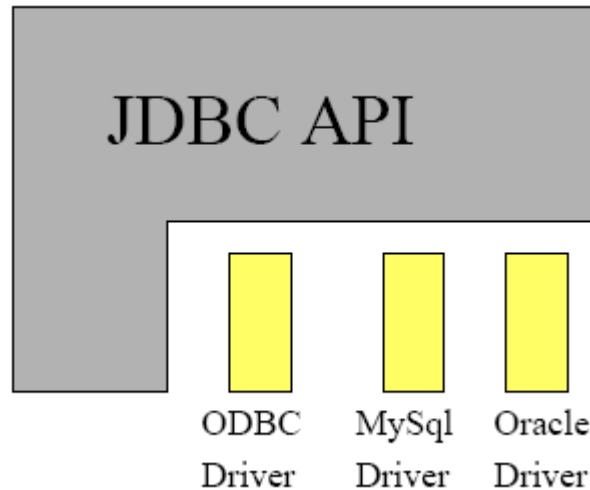
# Connexion aux bases de données en Java

## La solution: JDBC

- **Le problème: Java n'est pas une plateforme Windows**
  - Java n'est donc pas lié en soi à ODBC
  - Les applications Java doivent en effet pouvoir dialoguer avec n'importe quelle base de données
- **C'est pourquoi Java possède son propre système de communication avec les bases de données: JDBC (Java DataBase Connectivity)**
- **JDBC est un ensemble de classes prédéfinies pour chaque type de BD**
- **A l'instar d'ODBC, JDBC peut dialoguer avec un grand nombre de bases de données différentes (mais pas Access en tant que tel)**
- **En particulier, JDBC comprend un pilote qui lui permet de dialoguer avec ODBC**
- **Pour accéder à une base de données ODBC, un programme Java doit donc établir une connexion vers ODBC en précisant la DataSource dont il a besoin**
- **On parle en général d'un « Pont JDBC-ODBC »**

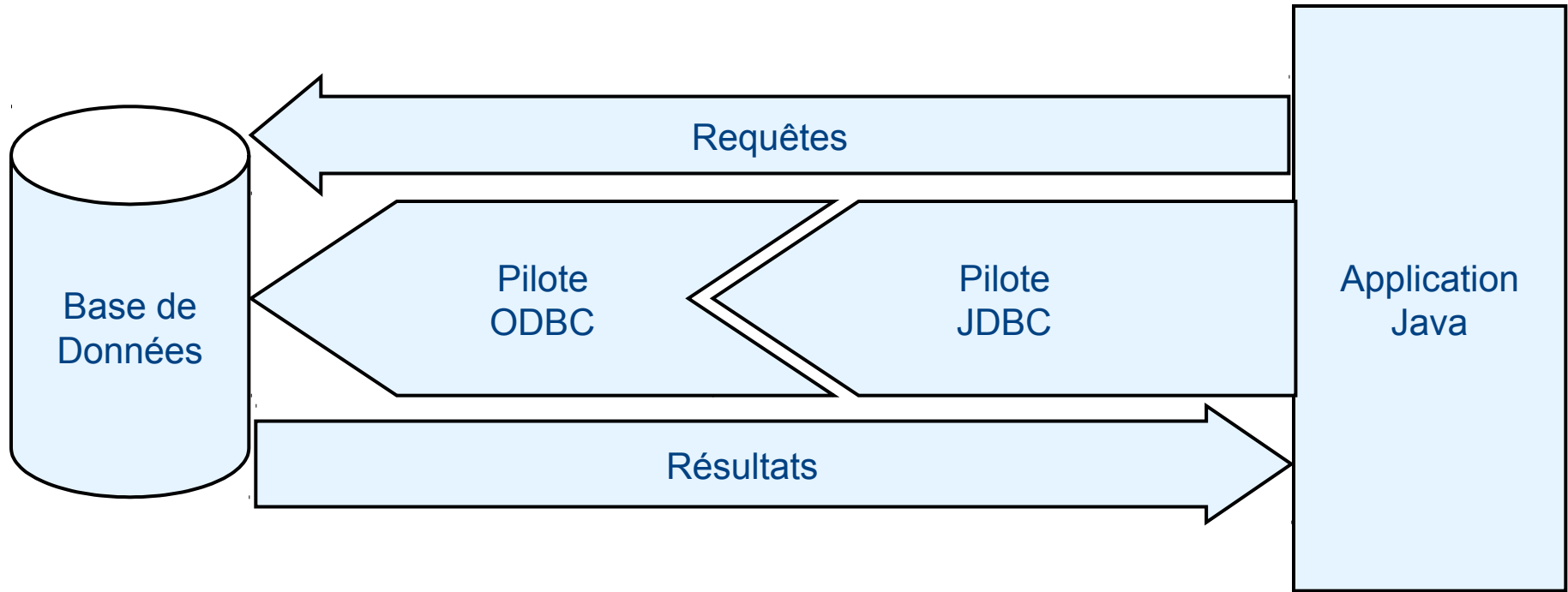
# Connexion aux bases de données en Java

## La solution: JDBC



# Connexion aux bases de données en Java

## La solution: JDBC



# Connexion aux bases de données en Java

## Mise en œuvre de JDBC

- **Pour établir une connexion JDBC, quelques lignes de code suffisent:**
  - `public static final String jdbcdriver = "sun.jdbc.odbc.JdbcOdbcDriver";`
  - `public static final String database = "jdbc:odbc:NomDeLaDataSourceODBC";`
  - `private Connection conn;`
  - **try**{  
    `Class.forName(jdbcdriver);`  
    `conn = DriverManager.getConnection(database);`  
} **catch**(Exception e) {  
    `e.printStackTrace();`  
}
- **Pour envoyer des requêtes à la BD une fois la connexion établie, il faut créer un « Statement »;**
  - `Statement stmt = conn.createStatement();`

# Connexion aux bases de données en Java

## Mise en œuvre de JDBC

- Une fois le Statement créé, on peut alors lui envoyer les requêtes SQL sous la forme de chaînes de caractères:
  - `String requete = "DELETE * FROM clients";`
  - `stmt.executeUpdate(requete);`
- Mais dans le cas de requêtes SELECT, il faut récupérer le résultat
- Les données issues de la requête sont rassemblées dans un jeu de résultats, appelé « ResultSet »:
  - `String requete = "SELECT * FROM client"`
  - `ResultSet rs = stmt.executeQuery(requete);`



# Connexion aux bases de données en Java

## Mise en œuvre de JDBC

- On peut alors accéder aux enregistrements renvoyés un à un en demandant chaque fois au **ResultSet** de nous renvoyer la ligne suivante:

```
while(rs.next()){  
  
    int c_id = rs.getInt("client_id");  
  
    String c_n = rs.getString("client_name");  
  
    String c_fn = rs.getString("client_firstname");  
  
    String c_ad = rs.getString("client_address");  
  
    String c_ph = rs.getString("client_phone");  
  
    int c_ab = rs.getInt("client_abonne");  
  
    double c_c = rs.getDouble("client_credit");  
  
    int c_am = rs.getInt("client_anneemembre");  
  
}
```

- Notez que l'on récupère la valeur de chaque champ en utilisant la méthode correspondant au type de données du champ (getInt, getString, getDate, etc.)

# Connexion aux bases de données en Java

## Mise en œuvre de JDBC

- Ce faisant, on a alors récupéré les données correspondant à chaque enregistrement
- Il nous reste donc à utiliser ces données pour créer un objet du type souhaité:

```
Client c = new Client(c_id,c_n,c_fn,c_ad,c_ph,c_c,c_am);
```

- Nous pouvons alors stocker ces clients un à un dans l'ArrayList:

```
clients.add(c); // A supposer que nous ayons une variable membre clients de  
// type ArrayList<Client> dans la classe
```

```
}
```

- Il faut enfin toujours veiller à fermer les connexions:

```
rs.close();
```

```
stmt.close();
```

```
conn.close();
```

# Introduction à Java

## XI. Multithreading



# Survol du chapitre

- **Introduction**

- Définition
- Raison d'être

- **Création de Thread**

- Par implémentation
- Par héritage

- **Gestion de Thread**

- Méthodes de gestion
- Diagrammes d'état

# Introduction

## Qu'est-ce qu'un Thread?

- **Un ordinateur qui exécute un programme :**
  - Possède un CPU
  - Stocke le programme à exécuter
  - Possède une mémoire manipulée par le programme→ Multitasking géré par l'OS
- **Un thread (« file ») a ces mêmes capacités**
  - A accès au CPU
  - Gère un processus
  - A accès à la mémoire, qu'il partage avec d'autres files→ Multithreading géré par la JVM

# Introduction

## Pourquoi le multithreading?

- **Un programme moderne est composé de**
  - Une interface graphique
  - Quelques composantes pouvant agir de manière autonome
- **Sans multithreading**
  - Les composantes ne pourraient agir que lorsque l'interface est suspendue
- **Plus généralement, le multithreading**
  - Permet de réaliser plusieurs processus indépendants en parallèle
  - Permet de gérer les files d'attente
  - Permet au besoin de synchroniser les différents processus entre eux

# Création de Thread

## Mise en œuvre (1/3)

- **Par implémentation de l'interface**

- Usage

- ➔ `public void MaClasse implements Runnable`

- Avantages et inconvénients

- ☺ Meilleur sur le plan orienté objet

- ☺ La classe peut hériter d'une autre classe

- ☺ Consistance

# Création de Thread

## Mise en œuvre (2/3)

```
public class MaFile implements Runnable {
    public void run(){
        byte[] buffer=new byte[512];
        int i=0;
        while(true){
            if(i++%10==0)System.out.println(""+i+" est divisible par 10");
            if (i>101) break;
        }
    }
}

public class LanceFile {
    public static void main(String[]arg){
        Thread t=new Thread(new MaFile());
        t.start();
    }
}
```

**Le constructeur de la classe Thread attend un objet Runnable en argument**





# Gestion des Thread

## Méthodes de gestion

- `t.start()`
  - Appeler cette méthode place le thread dans l'état "runnable"  
→ Eligible par le CPU
- `Thread.yield()` throws `InterruptedException`
  - La VM arrête la file active et la place dans un ensemble de files activables. (runnable state)
  - La VM prend une file activable et la place dans l'état actif (running state)
- `Thread.sleep(int millis)` throws `InterruptedException`
  - La VM bloque la file pour un temps spécifié (état « d'attente »)

# Exercice:

- **Faites déplacer les balles**

- Cette fois un Thread va s'en occuper
- La classe AireDeJeu contient lance un Thread
- La classe AireDeJeu implements Runnable et donc possede une methode run
- Dans la méthode run, on déplace les balles en boule
- (pas oublier d'appeler le repaint())

# Introduction à Java

## XII. Interfaces graphiques Java: SWING et AWT



# Interfaces graphiques Java: SWING et AWT

- **Qu'est-ce qu'une interface graphique (GUI)?**
- **Le paradigme « Model-View-Control » (MVC)**
- **L'affichage et ses composants (SWING)**
  - Structure de l'API SWING
  - Composants et conteneurs
  - Composants et conteneurs utiles
    - Exemples
    - Constructeurs
    - Méthodes utiles
- **La gestion des événements (AWT)**
  - Principes
  - Mise en œuvre
- **Connecter l'interface graphique à l'application OO et à la BD**

# Interfaces graphiques Java: SWING et AWT

## Qu'est-ce qu'une interface graphique?

- Une interface graphique ou GUI (*Graphical User Interface*) est:
  - Une couche applicative destinée à fournir aux utilisateurs d'ordinateurs un moyen attractif et facile pour interagir avec un programme
  - L'interface d'un programme qui profite des capacités graphiques d'un ordinateur pour le rendre plus facile d'utilisation. Des interfaces graphiques bien conçues évitent à l'utilisateur d'avoir à apprendre de complexes langages de commandes.
  - Un ensemble d'écrans de présentations qui utilisent des éléments graphiques (tels boutons et icônes) pour rendre un programme plus facile d'utilisation.

# Interfaces graphiques Java: SWING et AWT

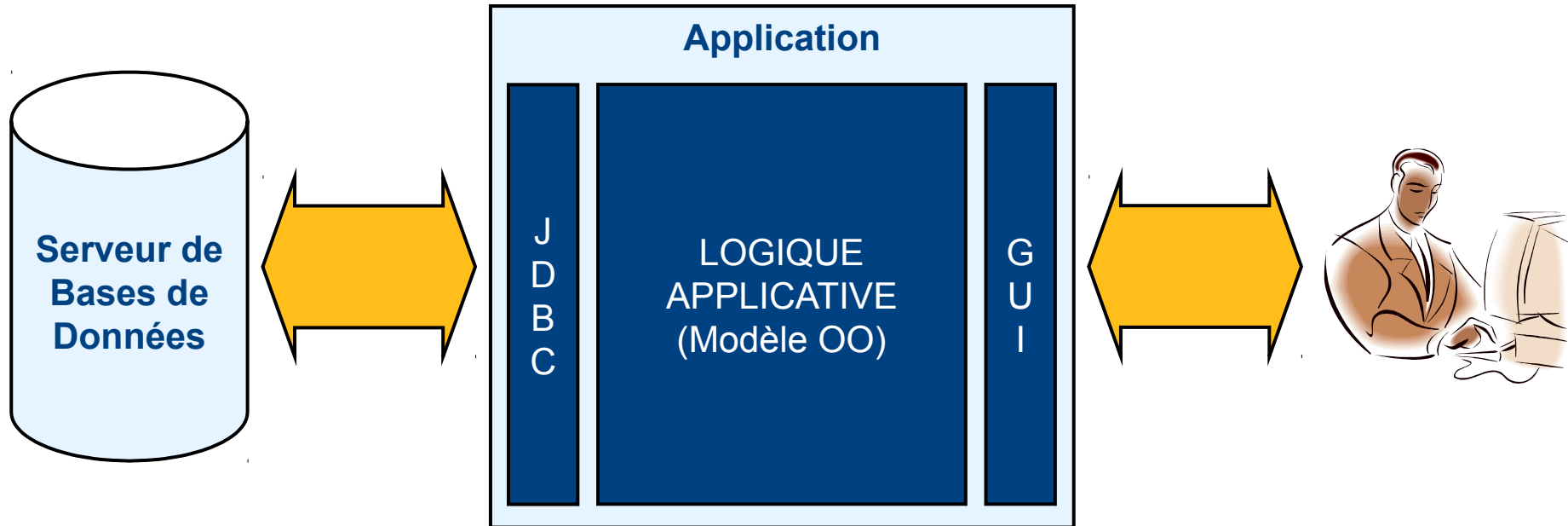
## Qu'est-ce qu'une interface graphique?

- **Une même application peut offrir différentes interfaces à l'utilisateur**
  - Une interface graphique *stricto sensu* qui s'exécute sur son ordinateur
  - Une interface web qui s'exécute dans un navigateur Internet
  - Une interface WAP qui s'exécute sur un téléphone
  - Une interface « terminal » qui s'exécute sur des appareils spécifiques
  - Des « services web » qui offrent ses fonctionnalités à d'autres applications

# Interfaces graphiques Java: SWING et AWT

## Qu'est-ce qu'une interface graphique?

- Notons que l'interface peut:
  - Etre intégrée à l'application (elle forme un tout avec l'application)  
(architecture « 2-tiers »)

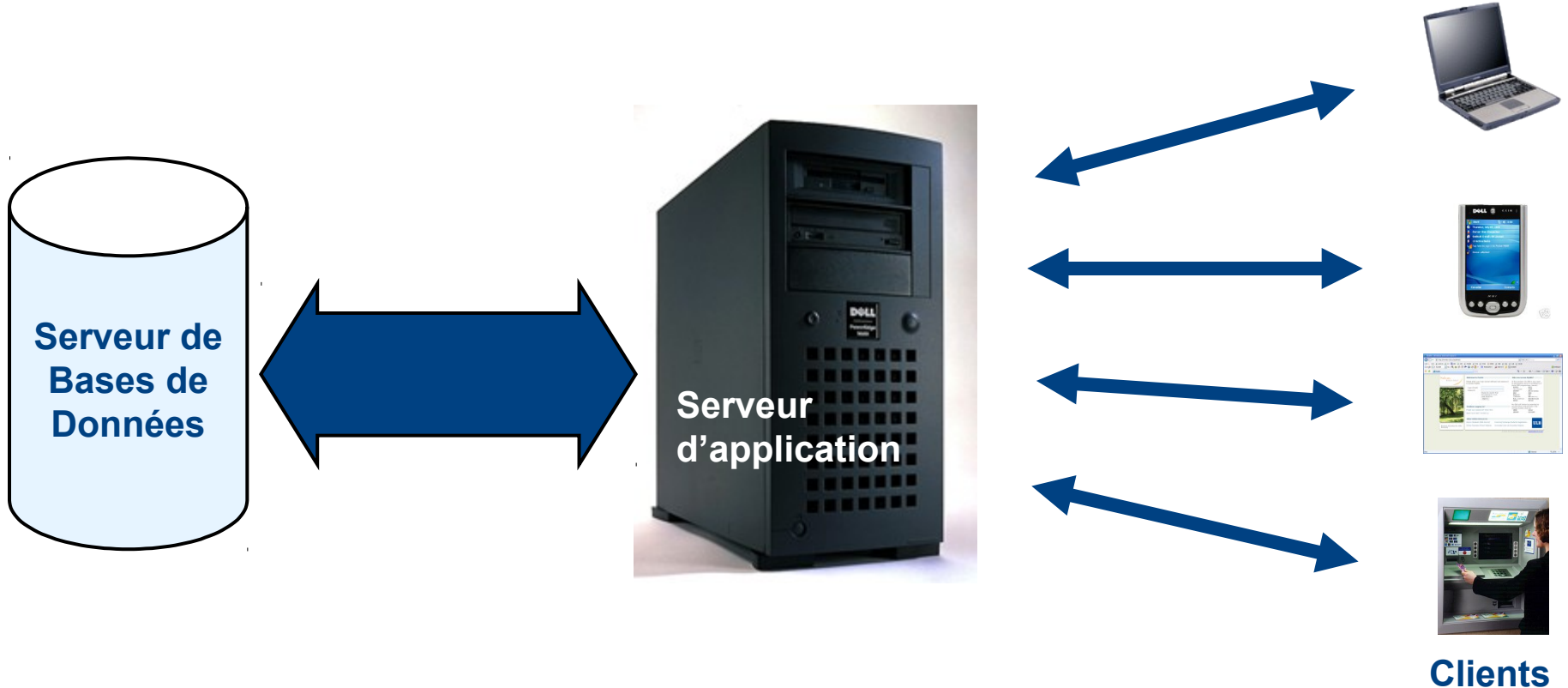


# Interfaces graphiques Java: SWING et AWT

## Qu'est-ce qu'une interface graphique?

- Notons que l'interface peut:

- Etre indépendante de l'application (différentes interfaces indépendantes utilisent la même application sous-jacente) et ne sont que la « partie visible » de l'application (architecture « 3-tiers »)





# Interfaces graphiques Java: SWING et AWT

## Model-View-Control

- **Fondement: Séparer**

- Les responsabilités relatives à la saisie des événements
- Celles relatives à l'exécution des commandes en réponse aux événements

- **Séparer au mieux**

- La gestion de l'affichage
- Le contrôle du composant
- Les informations intégrées dans le composant

- **Avantages:**

- Un même événement peut être envoyé à plusieurs objets écouteurs
  - Utile si un événement est potentiellement intéressant pour plusieurs écouteurs
- Facilite la réutilisation des composants
- Permet le développement de l'application et de l'interface séparément
- Permet d'hériter de super-classes différentes suivant les fonctionnalités
- Règle essentielle en OO: modulariser au plus ce qui est modularisable

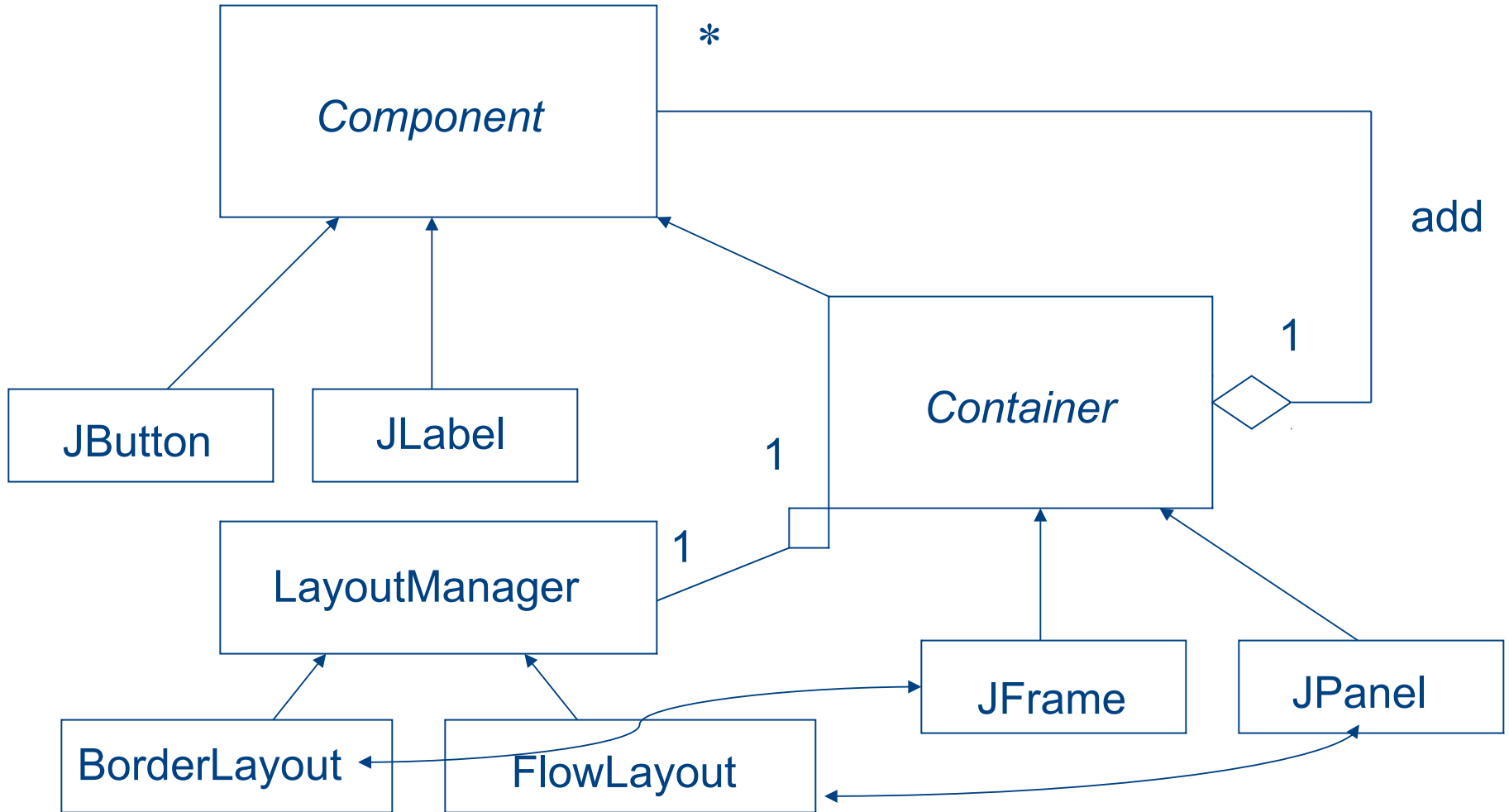
# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **SWING offre trois types d'éléments graphiques**
  - Les « Components » (composants ou contenus)
  - Les « Containers » (contenants, qui sont eux-mêmes des composants)
  - Les « LayoutManagers »
- **Les « Components »**
  - Constituent différents éléments de l'affichage (boutons, barres de menus, etc.)
  - Ex: JButton, JLabel, JTextArea, JCheckbox, etc.
- **Les « Containers »**
  - Sont destinés à accueillir des composants
  - Gèrent l'affichage des composants
  - Ex: JFrame, JPanel, JScrollPane
- **Les « LayoutManagers »**
  - Gèrent la disposition des composants au sein d'un conteneur

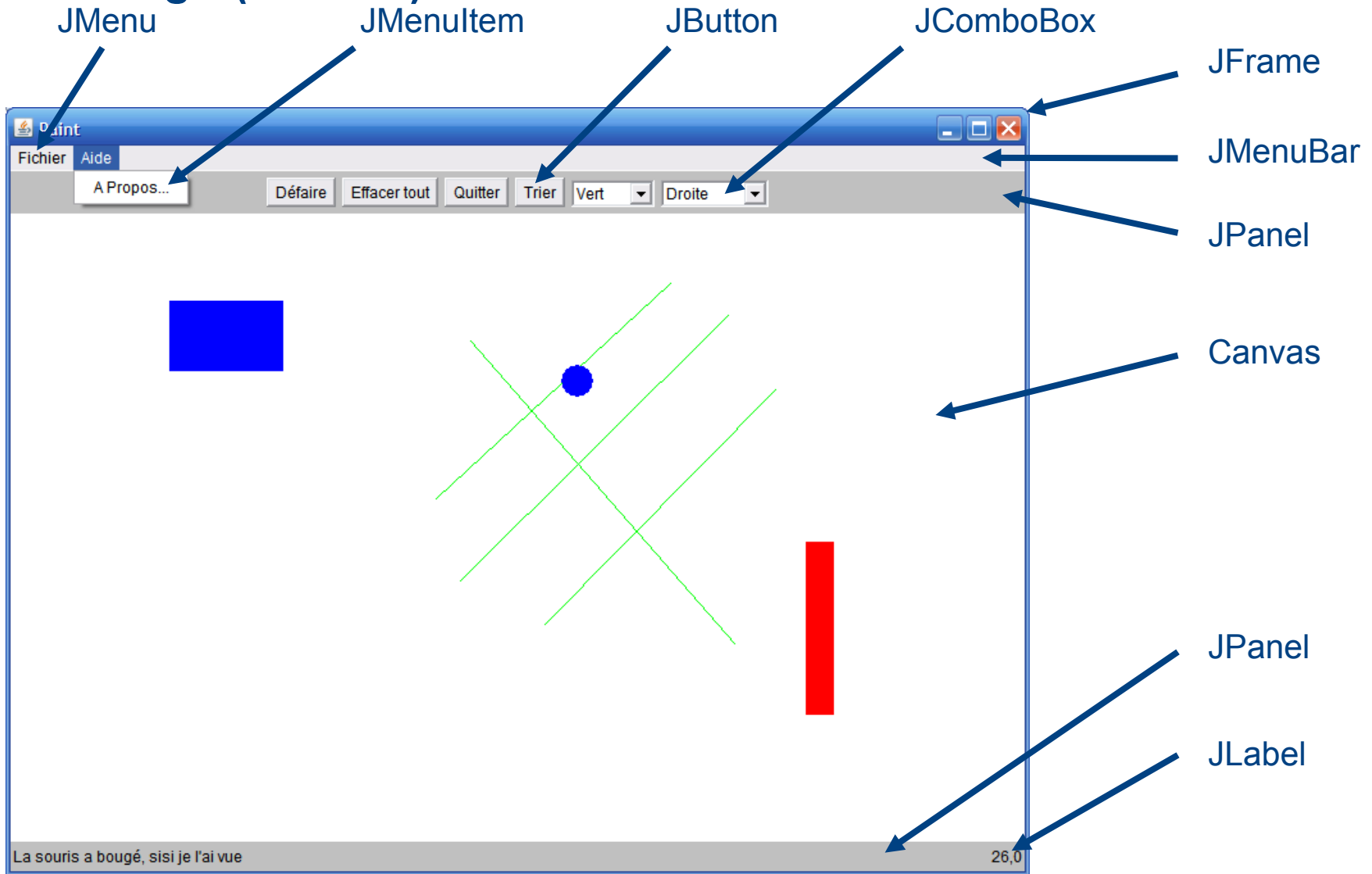
# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)



# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)



# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Exemples de composants:**

- JTextArea (zone de texte)
  - Contient une chaîne de caractères
- JTable (table de données)
  - Contient deux Arrays ([ ]):
    - Les données (tableau multidimensionnel d'objets)
    - Les intitulés des colonnes (vecteur de chaînes de caractères)
  - Gère automatiquement l'affichage des données en colonnes
  - Permet le déplacement de colonnes
  - Permet ou interdit la modification des données de la table
- JButton (bouton)
  - Contient un intitulé
- JCheckBox (case à cocher)
  - Contient un intitulé
  - Peut être sélectionnée ou non
  - Peut être activée ou désactivée (on peut ou non la cocher)
- JLabel (étiquette/intitulé)
  - Contient un intitulé
- JComboBox (liste déroulante)
  - Contient une liste d'éléments sous forme de chaînes de caractères
  - Contient un entier représentant le numéro de la ligne sélectionnée

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Exemples de conteneurs**

- JFrame

- Composant du plus haut niveau
    - La fenêtre d'une application est une instance de cette classe
    - Le Frame contient les différents composants graphiques de l'application
    - Ne peut être intégré dans un autre conteneur
    - Peut contenir une barre de menu (JMenuBar)

- JPanel

- Peut être lui-même intégré dans un autre conteneur (par ex. un JFrame)
    - Peut contenir lui-même autant de composants que nécessaire
    - Plusieurs moyens d'y disposer les composants

- JScrollPane

- Peut être lui-même intégré dans un autre conteneur (par ex. un JFrame)
    - Offre une « vue » sur un seul composant ou conteneur (JTable, JPanel, etc.)
    - Permet de changer le contenu en cours de route
    - Offre automatiquement des barres de défilement si nécessaire

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Barres et éléments de menus**

- JMenuBar

- Une barre de menu (une seule par JFrame)
    - Contient différents menus (JMenu)

- JMenu

- Représente un menu dans une barre de menu
    - Contient différents éléments (JMenuItem)
    - Contient un intitulé

- JMenuItem

- Représente un élément de menu
    - Contient un intitulé
    - Peut être associé à des raccourcis clavier

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Constructeurs des composants:**

- JTextArea (zone de texte)
  - `new JTextArea(String texteAAfficher);`
- JTable (table de données)
  - `new JTable(Object[ ][ ] donnees, String[ ] intitules);`
    - ➔ Il faut un tableau de chaînes de caractères contenant les intitulés des colonnes
    - ➔ Et une matrice d'objets dont chaque rangée est une ligne du tableau
- JButton (bouton)
  - `new JButton(String intitule);`
- JCheckBox (case à cocher)
  - `new JCheckBox(String intitule);`
- JLabel (étiquette/intitulé)
  - `new JLabel(String intitule);`
- JComboBox (liste déroulante)
  - `new JComboBox();`



# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Constructeurs des conteneurs**

- JFrame
  - new JFrame();
- JPanel
  - new JPanel();
- JScrollPane
  - new JScrollPane();

- **Constructeurs des menus**

- JMenuBar
  - new JMenuBar();
- JMenu
  - new JMenu(String intitule);
- JMenuItem
  - new JMenuItem(String intitule);

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Méthodes utiles (composants):**

- JTextArea (zone de texte)
  - setText(String) → Modifier le contenu
- JTable (table de données)
  - setSelectionMode(int) → Définir le mode de sélection (ex: une ligne à la fois)
  - getSelectionModel() → Récupérer le modèle de sélection défini
- JButton (bouton)
  - setText(String) → Modifier l'intitulé
- JCheckBox (case à cocher)
  - setText(String) → Modifier l'intitulé
  - setEnabled(boolean) → Déterminer si la case est activée ou pas (peut être cochée ou non)
  - setSelected(boolean) → Déterminer si la case est cochée ou pas
  - isEnabled() → Renvoie *true* si la case est activée et *false* si elle ne l'est pas
  - isSelected() → Renvoie *true* si la case est cochée et *false* si elle ne l'est pas
- JLabel (étiquette/intitulé)
  - setText(String) → Modifie l'intitulé
- JComboBox (liste déroulante)
  - addItem(String) → Ajouter un élément (chaîne de caractères) dans la liste
  - getSelectedIndex() → Renvoie le numéro de la ligne sélectionnée

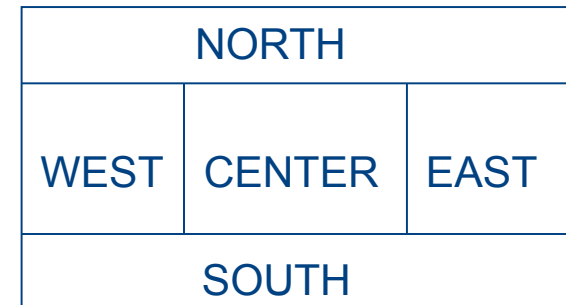
# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Méthodes utiles (conteneurs)**

- JFrame

- setSize(int width, int height) → Fixe les dimensions de la fenêtre
    - setTitle(String titre) → Définit le titre de la fenêtre
    - setJMenuBar(JMenuBar) → Installe la barre de menu
    - add(Component c) → Intègre le composant indiqué
    - setVisible(true) → Fait apparaître la fenêtre à l'écran



- JPanel

- setLayout(new BorderLayout()) → Divise le conteneur en 5 zones: N, S, E, W, C et permet d'ajouter un composant différent dans chaque zone (il n'est pas obligatoire d'utiliser les 5 zones)
    - add(Component c, "North") → Ajoute le composant « c » au « Nord »
    - setLayout(new GridLayout(x,y)) → Divise le conteneur en x lignes et y colonnes Les composants ajoutés viennent remplir la grille de gauche à droite et de ht en bas
    - add(Component c) → Ajoute le composant « c » dans la première case disponible

- JScrollPane

- setViewportView(Component c) → Affiche le composant/conteneur indiqué

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Méthodes utiles (menus)**

- JMenuBar

- add(JMenu) → Ajoute un menu dans la barre (de gauche à droite)

- JMenu

- add(JMenuItem) → Ajoute un élément dans le menu (de haut en bas)

- addSeparator() → Ajoute une ligne de séparation dans le menu

- JMenuItem

- setText(String) → Modifie l'intitulé de l'élément du menu

# Interfaces graphiques Java: SWING et AWT

## Affichage (SWING)

- **Comment utiliser les composants et conteneurs prédéfinis?**

- Par héritage

- Créer une nouvelle classe qui hérite de la classe prédéfinie qui nous intéresse
- Permet de créer « sa » propre fenêtre, « sa » propre barre de menus, etc.
- Généralement ce qu'on fait avec les conteneurs
- Plus rarement avec les composants
- Exemple:

```
- public Fenetre extends JFrame{ ... }
```

- Par association

- Consiste simplement à déclarer un objet du type du composant prédéfini comme variable membre dans une classe
- On se sert alors des méthodes du composant pour définir les propriétés du composant qui nous intéressent (intitulé, contenu, état, etc...)
- Exemple

```
- public ZonePrincipale extends JPanel{  
    JLabel l = new JLabel("Bonjour");  
}
```

# Exercice

## Une première application graphique

- Créer une classe « Fenetre » héritant de *JFrame*
- Construire la fenêtre d'application en ajoutant:
  - Au Nord: un *JPanel*
    - ➔ Contenant lui-même trois *JButton*
    - ➔ A placer dans un vecteur
  - Au Centre: un *Canvas*
    - ➔ Servira d'aire de jeu qui s'occupera de tous les objets du jeu (balle, monstre, avion, princesse,...)
  - Au Sud: un *JPanel*
    - ➔ Contenant lui-même deux *JLabel*
- Créer une classe principale contenant uniquement le main:

```
public static void main(String args[]) {  
  
    new Fenetre();  
  
}
```

# PROJET

Semaine Intensive Solvay 2006

Action Consultation

Liste des réservations

ID	Client	Spectacle	Représentation	Nb	Réservé le	Payé	Tick...
1	Bersini, Hugues	Axelle Red - Axelle Red en conc...	2007-01-21 20:00:00	1	2006-11-27	Oui	1
11	Bersini, Hugues	Peter Pan - Peter Pan on Ice	2006-12-28 19:00:00	1	2006-11-28	Oui	0
2	Bersini, Hugues	Peter Pan - Peter Pan on Ice	2006-12-30 11:00:00	2	2006-11-15	Oui	2
14	van Zeebroeck, Nic...	Peter Pan - Peter Pan on Ice	2006-12-31 15:00:00	2	2006-11-28	Oui	0
3	van Zeebroeck, Nic...	Pascal Obispo - Obispo en con...	2007-03-11 18:00:00	1	2006-11-27	Oui	0
4	Philemotte, Christo...	Disney on Ice - Les Princesses	2007-02-14 15:00:00	4	2006-11-26	Oui	4
12	Salihoglu, Utku	Axelle Red - Axelle Red en conc...	2007-01-21 20:00:00	6	2006-11-28	Oui	6
5	Salihoglu, Utku	Peter Pan - Peter Pan on Ice	2006-12-28 19:00:00	1	2006-11-27	Oui	0
6	Pignon, Françoise	Peter Pan - Peter Pan on Ice	2006-12-28 19:00:00	1	2006-11-27	Non	0
7	Brochant, Pierre	Peter Pan - Peter Pan on Ice	2006-12-28 19:00:00	1	2006-11-27	Non	0
8	Grandleur, Joséphi...	Disney on Ice - Les Princesses	2007-02-14 15:00:00	1	2006-11-27	Non	0
13	Campana, Lucienne	Peter Pan - Peter Pan on Ice	2006-12-28 19:00:00	4	2006-11-28	Non	0
15	Bonnefoi, Josiane	Peter Pan - Peter Pan on Ice	2006-12-29 19:00:00	290	2006-11-28	Non	0
10	Trembleur, Marcel	Michel Polnareff - Polnareff en c...	2007-03-29 20:30:00	1	2006-11-28	Non	0
16	Deville, Jonathan	Peter Pan - Peter Pan on Ice	2006-12-29 19:00:00	10	2006-11-28	Non	0
9	Deville, Jonathan	Johnny Hallyday - Johnny en co...	2007-02-20 20:00:00	10	2006-11-27	Non	0

Ready

Fenêtre d'application (JFrame)

Barre de menus (JMenuBar)

Titre (JLabel)

Zone de défilement (JScrollPane)  
pouvant contenir une table  
(JTable) ou un formulaire (JPanel)

Barre d'état (JLabel)

# Gestion d'événements

## Mécanismes et structure (1/4)

- **Une source d'événements**
  - Génère des objets événements
  - Les fait écouter par un ensemble d'écouteurs d'événements
  - En général: un composant ou conteneur graphique
- **Les objets événements**
  - xxxEvent
  - Contiennent la description et les caractéristiques d'un événement
- **Les objets écouteurs**
  - xxxListener ou xxxAdapter
  - Concrétisent des méthodes définies dans les Interfaces
  - Indiquent leur réaction en réponse aux événements
  - Sont des interfaces implémentables dans les classes
  - Peuvent être implémentés par les sources d'événements elles-mêmes  
(Une source d'événements peut « s'écouter » elle-même)



# Gestion d'événements

## Mécanismes et structure (2/4)

1. Un événement se produit
2. La source d'événement dans laquelle il se produit génère un objet de type événement
3. La source transmet l'événement à son (ses) écouteur(s)
4. L'écouteur appelle la méthode correspondant au type d'événement et lui passe en argument l'objet événement
5. La méthode en question spécifie les traitements à réaliser lorsqu'un événement du type correspondant se produit
6. Dans ses traitements, la méthodes peut examiner les caractéristiques de l'événement (position du curseur de la souris, code de la touche pressée au clavier...) et adapter son comportement en fonction de ces caractéristiques

# Gestion d'événements

## Mécanismes et structure (3/4)

- **Evénements**

- ActionEvent, AdjustmentEvent, ComponentEvent, ContainerEvent, FocusEvent, ItemEvent, KeyEvent, MouseEvent, TextEvent, WindowEvent

- **Les Interfaces Ecouteurs**

- ActionListener, AdjustmentListener, ComponentListener, ContainerListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, WindowListener

- **Les Adapteurs correspondants**

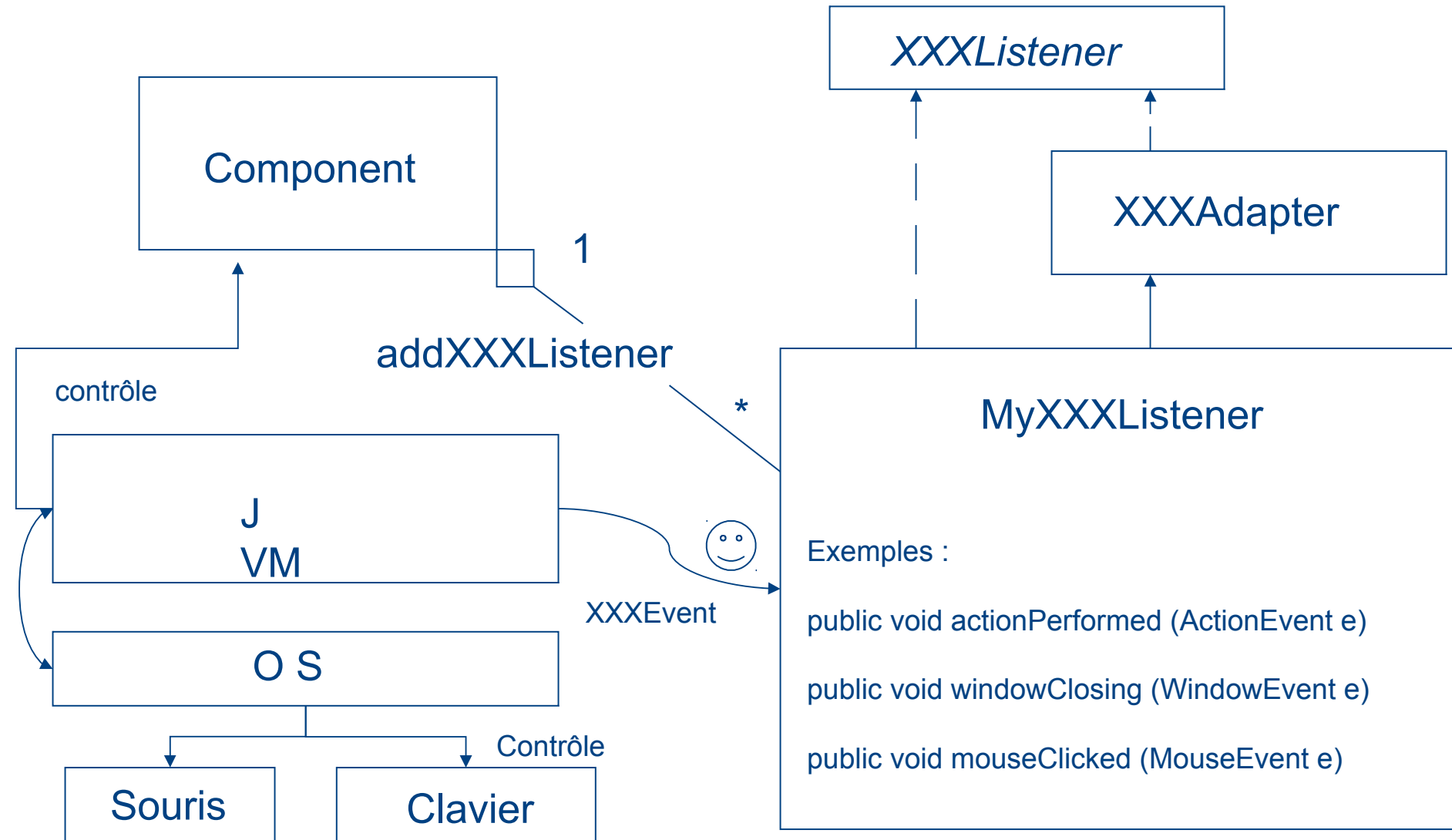
- ActionAdapter, WindowAdapter, KeyAdapter, MouseAdapter, etc.

- **Les Sources d'événements**

- Button, List, MenuItem, TextField, ScrollBar, CheckBox, Component, Container, Window

# Gestion d'événements

## Mécanismes et structure (4/4)



# Gestion d'événements

## Mise en œuvre

- **Par implémentation de l'interface**

- Usage

- ➔ `public class MaClasse implements ActionListener`

- ➔ Ensuite, il faut redéfinir la ou les méthode(s) prédéfinie(s)

- Avantages:

- ☺ Meilleur sur le plan orienté objet

- ☺ La classe peut hériter d'une autre classe

- ☺ Consistance

# Gestion d'événements

## Mise en œuvre – Événements de fenêtre

```
class Fenetre extends JFrame implements WindowListener
{
    public Fenetre()
    {
        addWindowListener(this);
    }

    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {System.exit(0);}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

# Gestion d'événements

## Mise en œuvre – Sélection de menus et boutons

```
class GereMenus implements ActionListener
{
    public void actionPerformed(ActionEvent e) {
        // Que faire quand une action est déclenchée?
    }
}
```

# Gestion d'événements

## Mise en œuvre – Sélection dans une JTable

```
class GereTables implements ListSelectionListener
{
    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting())
            return;
        ListSelectionModel lsm;
        lsm = (ListSelectionModel) e.getSource();
        if(lsm.isSelectionEmpty()){
            // Que fait si aucune ligne n'a été sélectionnée?
        } else {
            // La ligne sélectionnée est :
            int i = lsm.getMinSelectionIndex();
            // Que faire avec cette information ?
        }
    }
}
```

# Gestion d'événements

## Mise en œuvre – Associer la source et l'écouteur

- En implémentant ce qui précède dans la classe concernée, on rend alors la classe en question capable de prendre en charge les événements du type correspondant
- Il reste à associer chaque source potentielle d'événements (la fenêtre principale, chaque élément de menu, chaque bouton, chaque liste déroulante et chaque table) à son objet écouteur, instance de la classe correspondante
- Cela se fait en général pendant ou juste après la construction de l'objet source au moyen d'une méthode  
« addXXXXXXListener(objetEcouteur) »



# Gestion d'événements

## Mise en œuvre – Associer la source et l'écouteur

- Pour informer la fenêtre qu'elle est son propre écouteur:

```
this.addWindowListener(this);
```

- Pour associer un élément de menu à son écouteur :

```
clients = new JMenuItem("Liste des clients");
```

```
clients.addActionListener(new GereMenus());
```

- Pour associer un bouton à son écouteur :

```
bouton = new JButton("Sauver la réservation");
```

```
bouton.addActionListener(new GereMenus());
```

- Pour associer une table à son écouteur :

```
table = new JTable(dataTable,headersTable);
```

```
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

```
table.getSelectionModel().addListSelectionListener(new GereTables());
```

# PROJET

- **3 types d'événements (ou actions) doivent être traités :**
  - La fermeture de la fenêtre (pour qu'elle entraîne la fin du programme)
  - Les sélections dans les menus et les clics sur les boutons (pour déclencher les actions requises)
  - La sélection d'une ligne dans la table (pour savoir quelle réservation traiter)

Classe Source	Evénements	Interface	Classe Ecouteur
FenetrePrincipale	Fermeture de la fenêtre	WindowListener	FenetrePrincipale
JMenuItem	Choix dans un menu	ActionListener	GereMenus
JButton	Clic sur un bouton	ActionListener	GereMenus
JTable	Sélection d'une ligne	ListSelectionListener	GereTables

- **Il faut donc :**
  - Rendre la classe FenetrePrincipale capable de se gérer elle-même
  - Créer 2 classes d'écouteurs spécifiques : GereMenus, GereTables (code fourni)
  - Instancier ces 2 classes
  - Associer les écouteurs à la source d'événements qu'ils doivent gérer

# Connecter l'interface graphique à l'application OO

- Pour « relier » l'interface graphique à l'application OO, il suffit d'en utiliser des objets dans les classes graphiques
- Tout dépend naturellement des fonctionnalités requises dans l'interface graphique et donc des besoins de celle-ci en matière d'objets
- Par exemple, pour afficher les clients, l'interface graphique aura besoin d'une collection contenant les objets clients générés au départ de la base de données
- L'interface graphique devra donc pouvoir obtenir ces objets, ce qui suppose qu'elle puisse interagir avec la base de données...

# PROJET

- **Maintenant que la partie visible de votre application est en place et qu'elle est capable de prendre en charge les différentes actions de l'utilisateur, le moment est venu d'incorporer votre application précédente (gestion des réservations) à l'intérieur de votre interface graphique**
- **Autrement dit, il va falloir « connecter » votre interface graphique à vos objets clients, spectacles, représentations, réservations et tickets**
  - afin d'extraire les données nécessaires à l'affichage des tableaux / formulaires
  - et de pouvoir envoyer les nouvelles informations vers la base de données

# PROJET

- **Affichage des tableaux de données (clients, réservations, etc.)**

- Récupérer les données de la BD sous forme d'objets. Autrement dit, interroger l'objet de type `DataAccessObject` pour qu'il renvoie la collection des clients ou des réservations, etc :

```
ArrayList data = dao.getClients();
```

- Demander à chacun des objets (clients, réservations, etc.) de renvoyer ses informations de façon structurée, sous forme d'un vecteur à « `[]` ». Autrement dit, il faudra boucler sur l'`ArrayList` renvoyée par le `DataAccessObject` et demander à chaque élément de l'`ArrayList` de renvoyer ses données pour les stocker dans un `Array` d'Objects

```
for(int i=0;i<data.size();i++){  
    dataTable[i] = ((DataObject) data.get(i)).getDataArray();  
}
```

- Notez que cela suppose que tous vos objets (clients, spectacles, etc.) possèdent un message (méthode) de type :

```
public Object[] getDataArray(){...}
```

- Créer la table en lui envoyant ces données

```
table = new JTable(dataTable,headersTable);
```

« `headersTable` » devrait désigner un tableau de « `String` » comprenant les intitulés des différentes colonnes.

- Afficher la table à l'intérieur de la zone de défilement

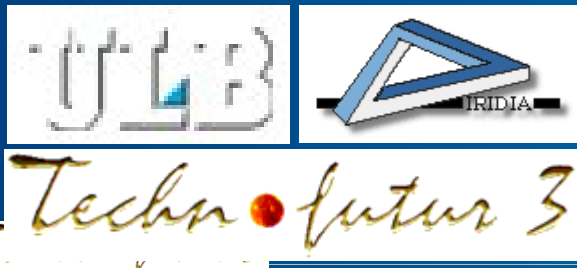
```
zoneDefilement.setViewportViewView(table)
```

# PROJET

- **Créer/modifier des données dans la BD**
  - ➔ nouvelle réservation, paiement, choix de places
  - Dans votre classe ZonePrincipale, une méthode correspondra à chaque type d'opération possible et sera appelée lorsque le bouton d'enregistrement du formulaire sera activé
  - Ces méthodes récupéreront les informations du formulaire (valeurs choisies dans les listes, cases cochées, etc.) et les vérifieront (sont-elles valables, reste-t-il suffisamment de places disponibles pour la représentation choisie, etc.)
  - Enfin, la méthode enverra un message à la classe DataAccessObjects pour appeler (comme le faisait votre « main » avant la création de l'interface graphique) la méthode correspondant à l'opération, cette dernière ayant pour mission d'effectuer les modifications dans la base de données.

# Introduction à Java

## Annexe - Une brève introduction à l'UML



# Les concepts de l'OO

## La modélisation devient la référence

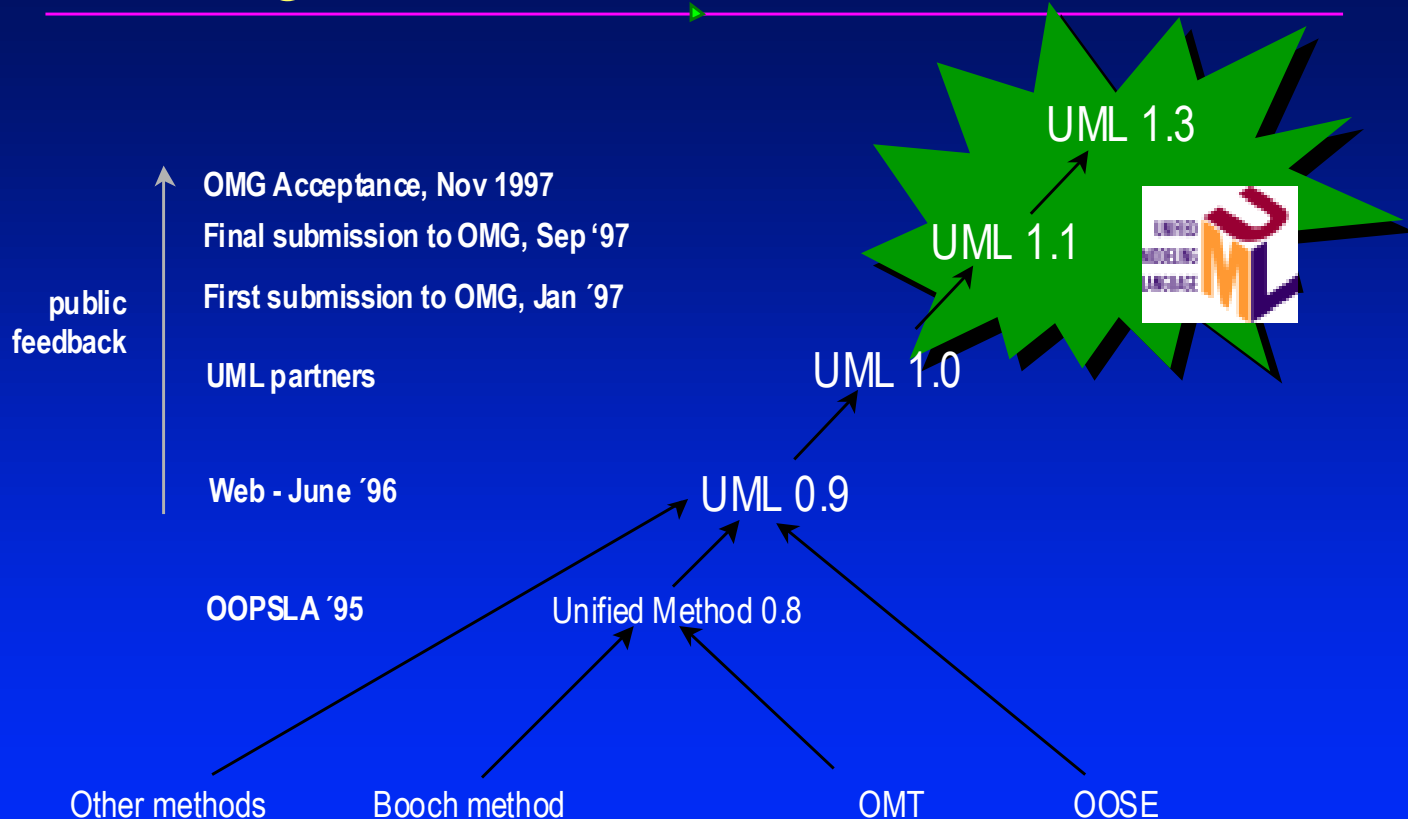
- **Pourquoi la modélisation?**
  - La conception OO est entièrement bâtie sur une modélisation des objets intervenant dans le problème
  - Avant de programmer quoi que ce soit, il faut donc modéliser les classes et leurs relations au minimum
- **Comment?**
  - Sur base d'UML (Unified Modeling Language)
    - ➔ Notation standardisée pour toute le développement OO de la conception au déploiement
    - ➔ Définition de 9 diagrammes
  - 1. Identifier les classes
    - ➔ Attributs, comportements, polymorphisme
  - 2. Déterminer les relations entre les classes
    - ➔ Associations / Dépendance / Héritage
  - 3. Construire les modèles



# Les concepts de l'OO

## La modélisation devient la référence

### Creating the UML



# Les concepts de l'OO

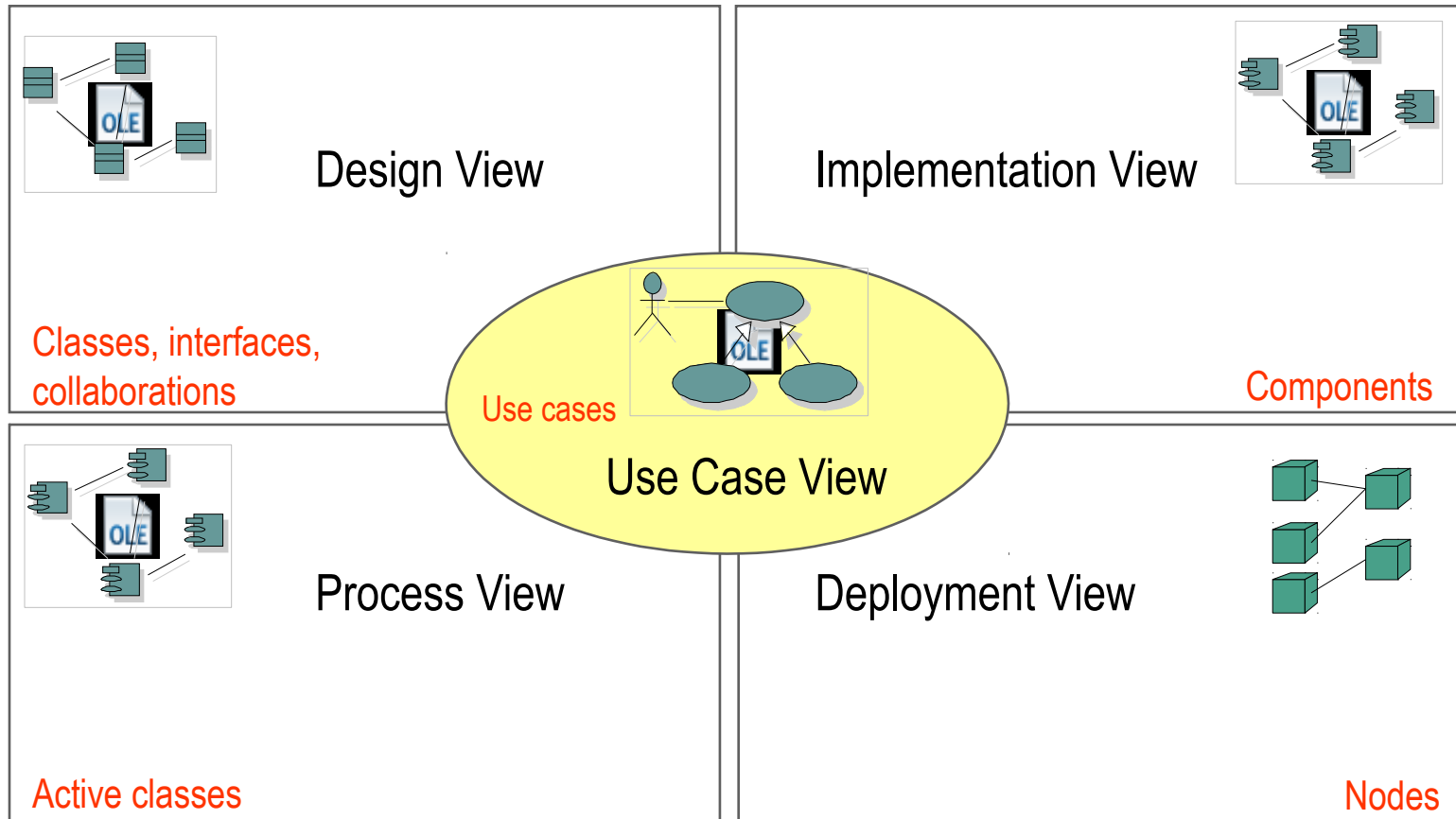
## La modélisation devient la référence

- **Qu'est-ce qu'UML?**

- UML est un langage objet graphique - un formalisme orienté objet, basé sur des diagrammes (9)
- UML permet de s'abstraire du code par une représentation des interactions statiques et du déroulement dynamique de l'application.
- UML prend en compte, le cahier de charge, l'architecture statique, la dynamique et les aspects implémentation
- Facilite l'interaction, les gros projets
- Générateur de squelette de code
- UML est un langage PAS une méthodologie, aucune démarche n'est proposée juste une notation

# Les concepts de l'OO

## La modélisation devient la référence



# Les concepts de l'OO

## La modélisation devient la référence

- **Diagrammes UML**

- Les diagrammes des cas d'utilisation: les fonctions du système, du point de vue de l'utilisateur ou d'un système extérieur - l'usage que l'on en fait
- Les diagrammes de classes: une description statique des relations entre les classes
- Les diagrammes d'objet: une description statique des objets et de leurs relations. Une version « instanciée » du précédent
- Les diagrammes de séquence: un déroulement temporel des objets et de leurs interactions
- Les diagrammes de collaboration: les objets et leurs interactions en termes d'envois de message + prise en compte de la séquentialité

# Les concepts de l'OO

## La modélisation devient la référence

- **Diagrammes UML**

- Les diagrammes d'états-transitions: scrute les cycles de vie d'une classe d'objet, la succession d'états et les transitions
- Les diagrammes d'activité: le comportement des différentes opérations en termes d'actions
- Les diagrammes de composants: représente les composants physiques d'une application
- Les diagrammes de déploiements: le déploiement des composants sur les dispositifs et les supports matériels

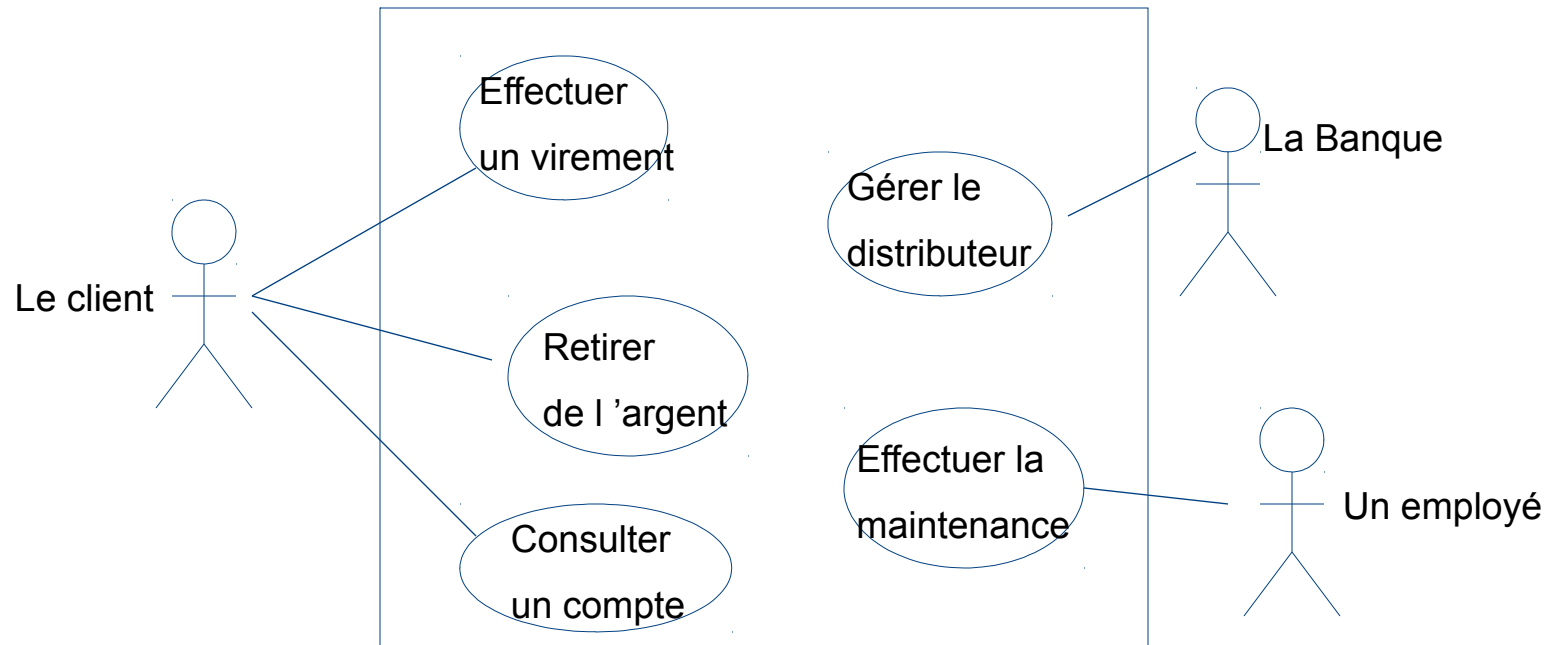
# Les concepts de l'OO

## La modélisation devient la référence

- **Diagramme de cas d'utilisation**

- Cela répond aux spécifications du système:
  - Ses fonctionnalités, son utilisation, les attentes de l'utilisateur

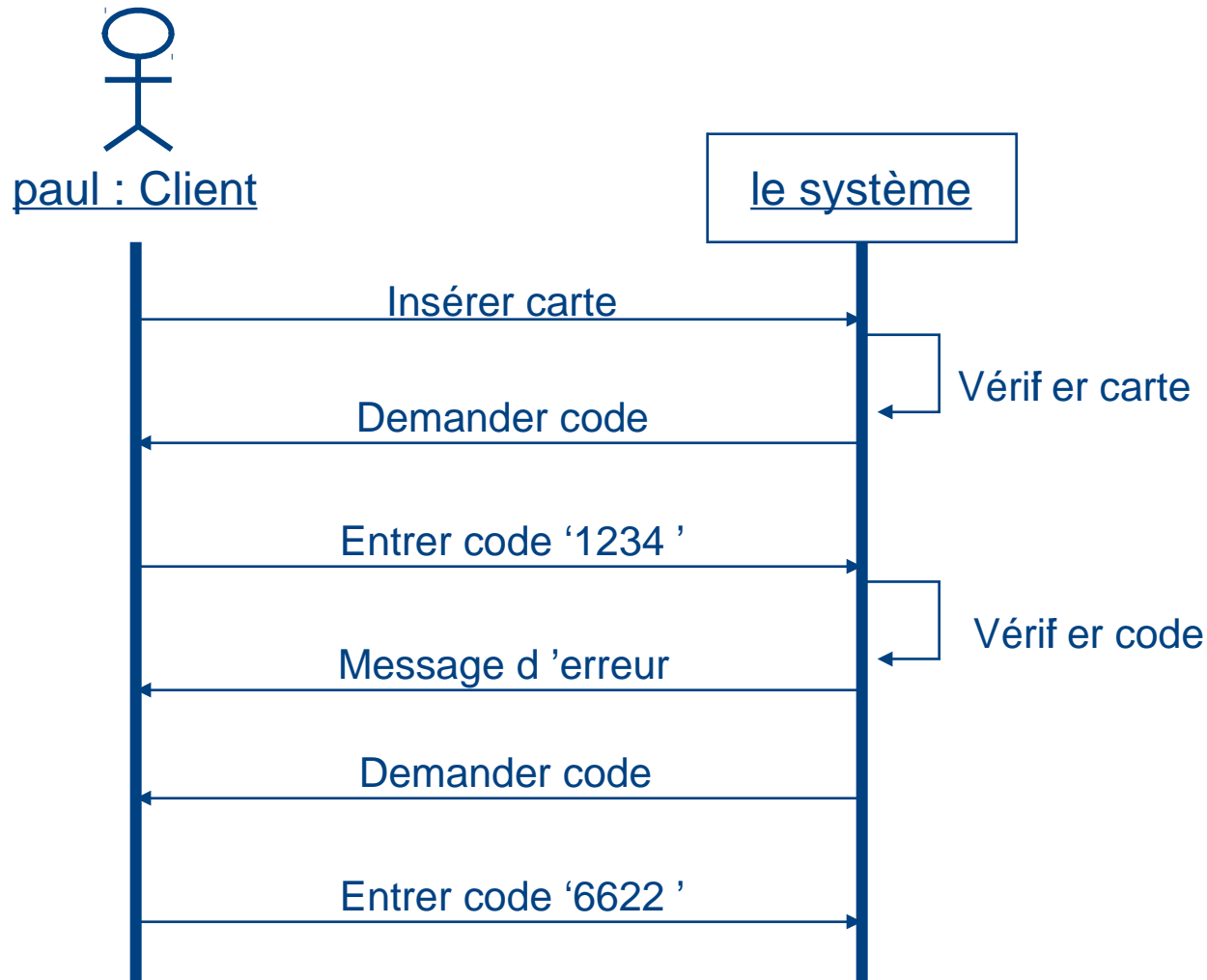
- **Ex: Distributeur MisterCash**



# Les concepts de l'OO

## La modélisation devient la référence

- Exemple de diagramme de cas d'utilisation détaillé



# Les concepts de l'OO

## La modélisation devient la référence

- **Diagramme de classes**

- Le but du diagramme de classes est de représenter les classes au sein d'un modèle
- Dans une application OO, les classes possèdent:
  - Des attributs (variables membres)
  - Des méthodes (fonctions membres)
  - Des relations avec d'autres classes
- C'est tout cela que le diagramme de classes représente
- L'encapsulation est représentée par:
  - (private), + (public), # (protected)
- Les attributs s'écrivent:  
+/-/# nomVariable : Type
- Les méthodes s'écrivent:  
+/-/# nomMethode(Type des arguments) : Type du retour ou « void »

Nom Classe
Attributs
Méthodes()



# Les concepts de l'OO

## La modélisation devient la référence

- Les relations d'héritage sont représentées par:

- A —| B signifie que la classe A hérite de la classe B

- L'agrégation faible est représentée par:

- A ◊— B signifie que la classe A possède un ou plusieurs attributs B

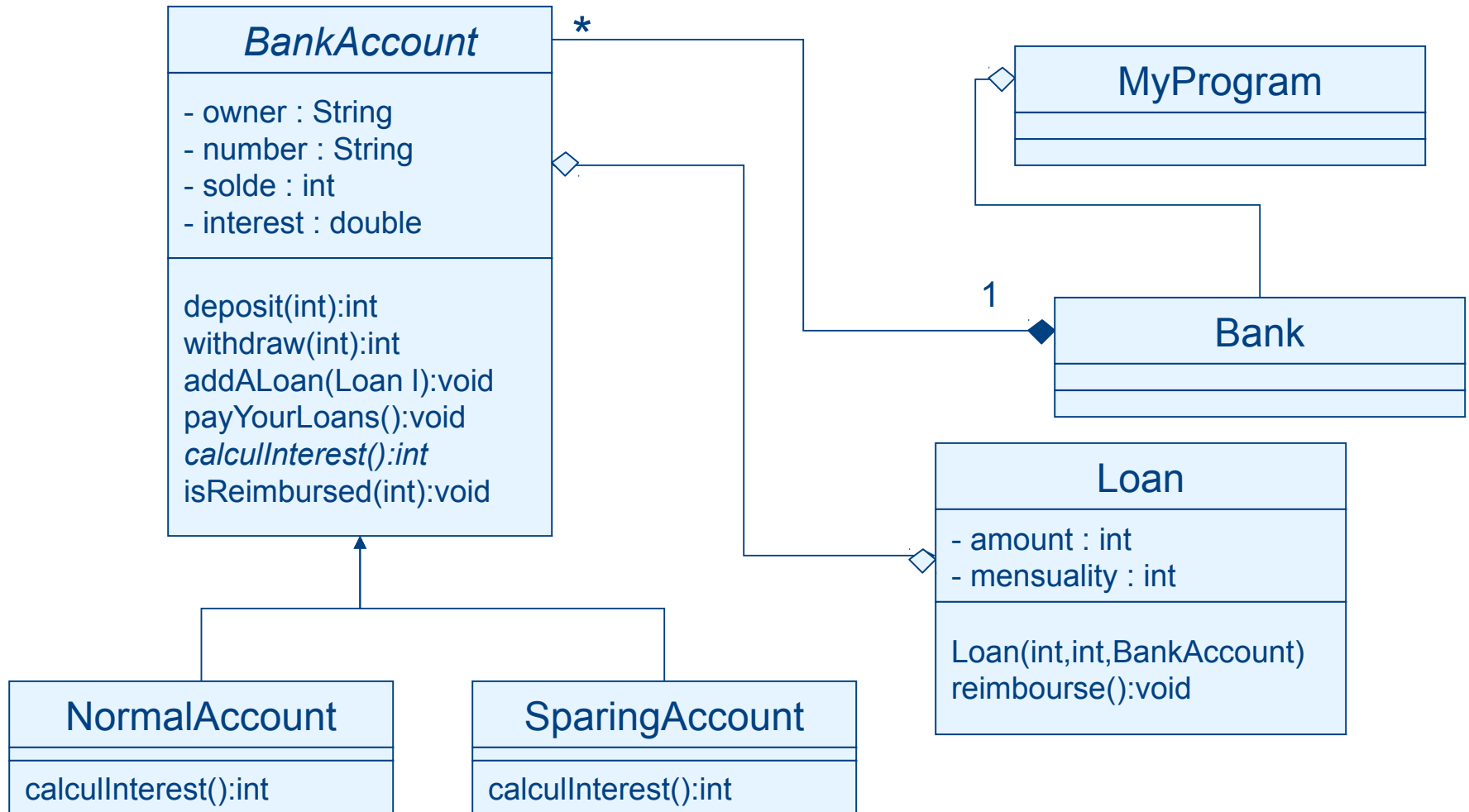
- L'agrégation forte (ou composition) est représentée par:

- A ◆— B signifie que les objets de la classe B ne peuvent exister qu'au sein d'objets de type A

# Les concepts de l'OO

## La modélisation devient la référence

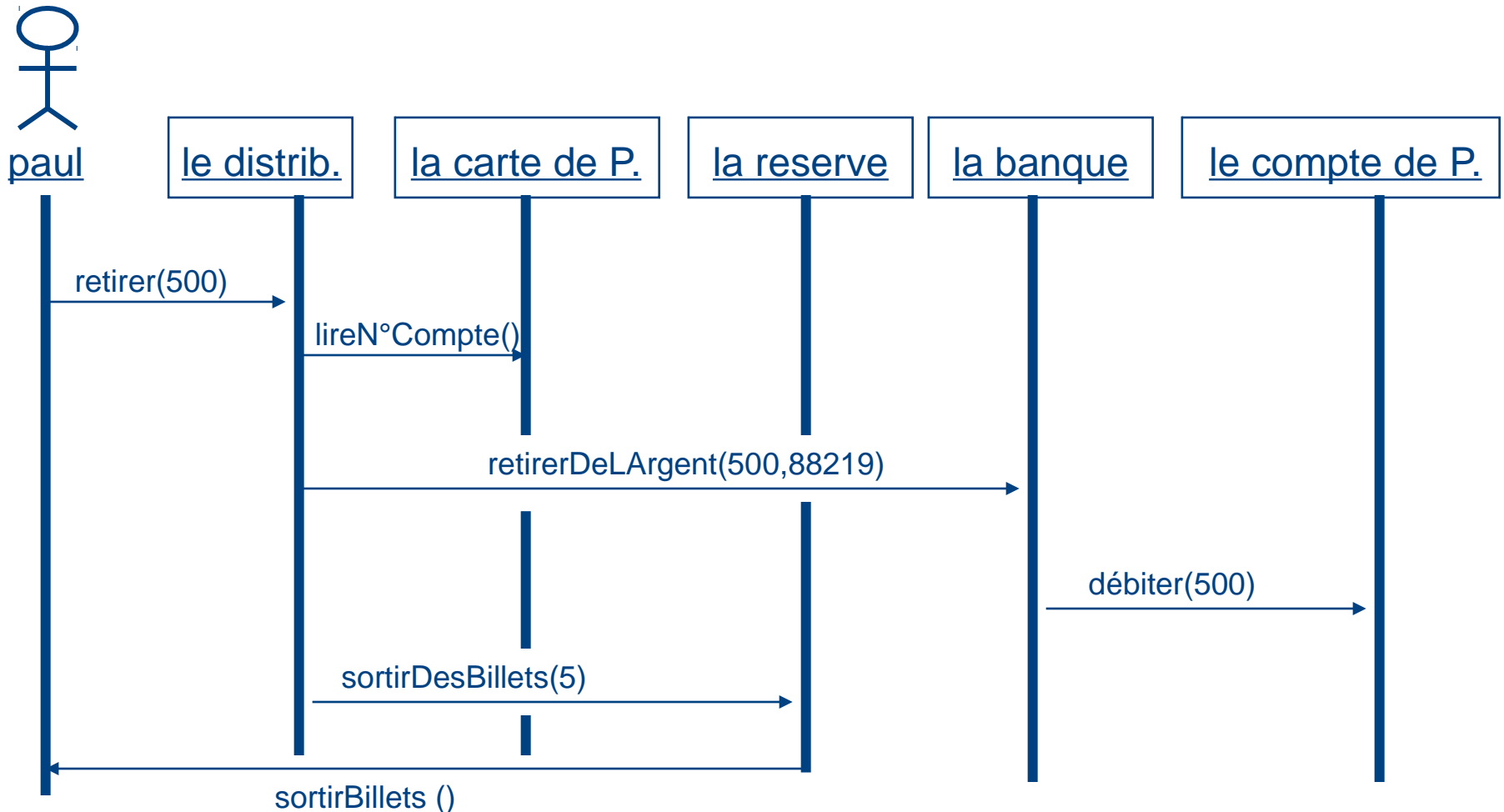
- Exemple de diagramme de classes



# Les concepts de l'OO

## La modélisation devient la référence

- Exemple de diagramme de séquence



# Les concepts de l'OO

## Les avantages de l'OO

- Les programmes sont plus stables, plus robustes et plus faciles à maintenir car le couplage est faible entre les classes («encapsulation»)
- elle facilite grandement le ré-emploi des programmes: par petite adaptation, par agrégation ou par héritage
- émergence des «design patterns»
- il est plus facile de travailler de manière itérée et évolutive car les programmes sont facilement extensibles. On peut donc graduellement réduire le risque plutôt que de laisser la seule évaluation pour la fin.
- l'OO permet de faire une bonne analyse du problème suffisamment détachée de l'étape d'écriture du code - on peut travailler de manière très abstraite → UML
- l'OO colle beaucoup mieux à notre façon de percevoir et de découper le monde

# Les concepts de l'OO

## Les avantages de l'OO

- Tous ces avantages de l'OO se font de plus en plus évidents avec le grossissement des projets informatiques et la multiplication des acteurs. Des aspects tels l'encapsulation, l'héritage ou le polymorphisme prennent vraiment tout leur sens. On appréhende mieux les bénéfices de langage plus stable, plus facilement extensible et plus facilement ré-employable
- JAVA est un langage strictement OO qui a été propulsé sur la scène par sa complémentarité avec Internet mais cette même complémentarité (avec ce réseau complètement ouvert) rend encore plus précieux les aspects de stabilité et de sécurité

# Les concepts de l'OO

## Langages et plateformes

- Quels sont les principaux langages orienté objet aujourd'hui?

- C++
  - Hybride, permet la coexistence d'OO et procédural
  - Puissant et plus complexe
  - Pas de « ramasse-miettes », multihéritage, etc.
- Java
  - Très épuré et strictement OO
  - Neutre architecturalement (Multi-plateformes)
  - Ramasse-miettes, pas de multihéritage, nombreuses librairies disponibles
- C#
  - Très proche de Java
  - Conçu par Microsoft
- Eiffel
  - Le plus pur(itain) des langages OO
  - Conçu par Bertrand Meyer
- Autres
  - PowerBuilder, Delphi, Smalltalk (I.A.), etc.

# Les concepts de l'OO

## En résumé

- Tout est un objet
- L'exécution d'un programme est réalisée par échanges de messages entre objets
- Un message est une demande d'action, caractérisée par les paramètres nécessaires à la réalisation de cette action
- Tout objet est une instance de classe, qui est le « moule » générique des objets de ce type
- Les classes définissent les comportements possibles de leurs objets
- Les classes sont organisées en une structure arborescente à racine unique : la hiérarchie d'héritage
- Tout le code des programmes se trouve entièrement et exclusivement dans le corps des classes
- A l'exception toutefois de deux instructions:
  - package → définit l'ensemble auquel la classe appartient
  - import → permet l'utilisation de classes extérieures au package
- UML permet la représentation graphique des applications