



Formation

-Programmer en Java SE 7 –

-Partie 1 -

Nantes - du 24 au 29/06/2015

- Qui êtes-vous ?
- Pourquoi avoir choisi cette formation ?
- Quels conseils, quelles informations et compétences souhaitez-vous obtenir grâce à cette formation ?
- Les objectifs cités correspondent-ils à vos attentes ?
- Quelles sont les compétences qui vous semblent nécessaires pour suivre ce cours ?
- Êtes vous venu(e) avec des questions ?

- Comprendre Java SE : historique, syntaxe, sémantique
- Mettre en oeuvre la modélisation objet avec UML
- Appliquer les concepts objet avec Java
- Connaître et utiliser les classes de base
- Se préparer à la certification Oracle Java SE 7 (1Z0 803)
- Savoir utiliser les principales nouveautés de Java 8 : lambdas et streams

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10. Les entrées / sorties (IO)**
- 11. L'accès aux bases de données**
- 12. Le mapping objet-relationnel**

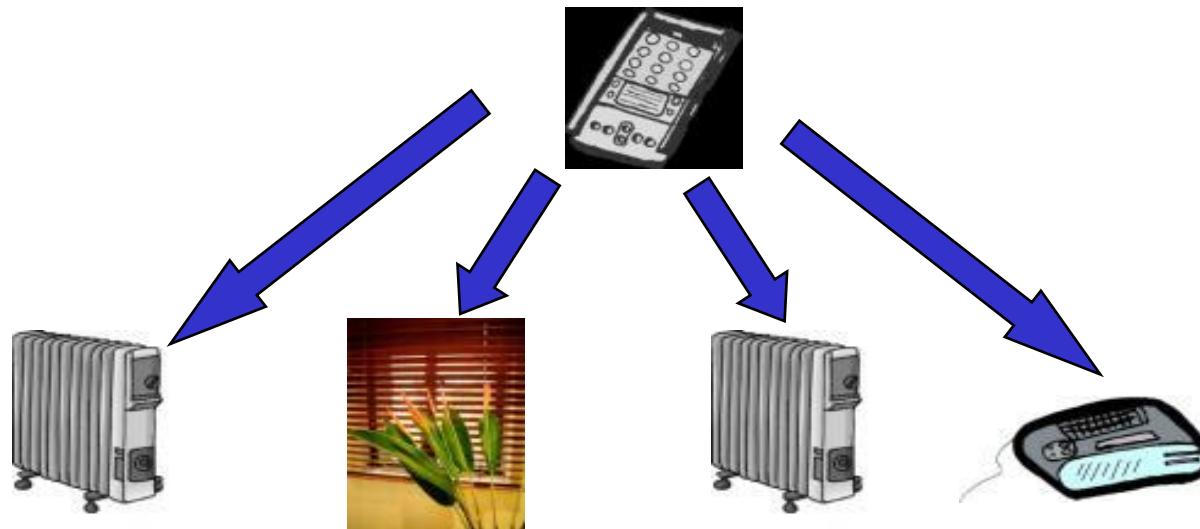
- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10. Les entrées / sorties (IO)**
- 11. L'accès aux bases de données**
- 12. Le mapping objet-relationnel**

- Historique de Java**
- Caractéristiques de Java**
- Positionnement de Java par rapport aux autres langages**



□ 1990 : Le 'Green Project'

- Une équipe d'ingénieurs de Sun conçoit un nouveau langage pour l'électronique grand public. Le langage doit être portable, simple et compact.
- Ce langage devait permettre à des appareils ménagers hétérogènes de communiquer entre eux via une plate-forme commune.

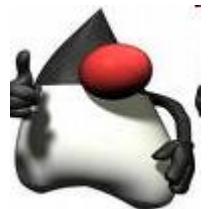


radiateur 1

Stores électriques

Radiateur 2

réveil



□ 1992 : Un début prometteur

- Création du langage Oak par Grosling
- Première démonstration (PDA Star 7)
- Duke, la mascotte de Java
- Création de la filiale FirstPerson, Inc.

□ 1994 : Le marché de la domotique ne décolle pas

- Sun doit faire un choix :
 - Stopper le projet, et attendre que le marché se développe.
 - Trouver un autre domaine dans lequel un langage portable pourrait être utile.
- Recentrage du projet sur le Web



- Début 1995 : Le marché de l'Internet est en pleine explosion

- Les sites Internet statiques fleurissent sur le Web
 - Sun travaille sur son propre navigateur Web et cherche le moyen de télécharger facilement des applications pour les exécuter dans ce navigateur
 - La 'Green team' réalise que l'Internet aura besoin d'un langage multi-plates-formes.

« WebRunner était juste une démo, mais une démo impressionnante : Il a porté à l'écran, pour la première fois, des objets animés et un contenu dynamique exécutable à l'intérieur d'un navigateur Web. »

WebRunner

WebRunner is a Rapid Web-Based Server That Runs Directly in the Browser. WebRunner enables the Internet browser to act as a server, allowing you to run your own Web-based applications without installing or running any software on your computer. WebRunner is a Java-based application that runs directly in your browser, so there's no need to download or install anything. Using WebRunner you can easily implement the most common types of client/server web applications, such as e-mail, news groups, bulletin boards, and file servers. WebRunner also provides a way to store and access local databases. WebRunner is a Java-based application that runs directly in your browser, so there's no need to download or install anything. Using WebRunner you can easily implement the most common types of client/server web applications, such as e-mail, news groups, bulletin boards, and file servers. WebRunner also provides a way to store and access local databases.

What makes the WebRunner product is how differently "Web" the web publishing environment is when it is implemented in such a way. It's a simple, powerful, and reliable way to publish and administer web sites. WebRunner is a Java-based application that runs directly in your browser, so there's no need to download or install anything. Using WebRunner you can easily implement the most common types of client/server web applications, such as e-mail, news groups, bulletin boards, and file servers. WebRunner also provides a way to store and access local databases.

WebRunner Application Examples:

- **E-mail Systems With Browser**
 - **JavaMail** - JavaMail is a Java API for the exchange of electronic mail messages between different systems.
 - **JavaMail API**
- **News Groups**
 - **JavaMail** - JavaMail is a Java API for the exchange of electronic mail messages between different systems.
 - **JavaMail API**
- **Bulletin Boards**
 - **JavaMail** - JavaMail is a Java API for the exchange of electronic mail messages between different systems.
 - **JavaMail API**
- **File Servers**
 - **JavaMail** - JavaMail is a Java API for the exchange of electronic mail messages between different systems.
 - **JavaMail API**
- **JavaMail API**

WebRunner Application Examples:

- **INTERNET BROWSER**
 - **JavaMail** - JavaMail is a Java API for the exchange of electronic mail messages between different systems.
 - **JavaMail API**
- **JavaMail API**

JavaMail API

An application on running WebRunner with JavaMail API is available at <http://www.java.com>.



□ **Mai 1995 : Sun met à disposition le code source de Java sur Internet**

- En quelques mois, Java devient célèbre. Plusieurs milliers d'utilisateurs le téléchargent chaque mois.

□ **Fin 1995 : Netscape intègre Java dans son navigateur**

- Java devient potentiellement utilisable par des millions d'Internautes.

□ **1996 : Microsoft intègre Java dans Internet Explorer**

- Sun rallie de nombreux partenaires (IBM, Oracle, Netscape...) autour de la technologie Java



- **1996 : Version 1.0 du Java Development Kit (JDK)**
 - Lancement officiel de Java par Sun et de nombreux autres partenaires (IBM, Oracle, Netscape...)
 - Première conférence JavaOne
- **1997: JDK 1.1**
 - Téléchargé plus de 220'000 fois en 3 semaines
- **1998: JDK 1.2**
 - Plus de 2 millions de téléchargements
 - Formalisation du programme Java Community Process (JCP)
 - Cartes visas basées sur Java Card

Version	Last update	Dénomination JSE/JRE	Nom de code	Spécifications	JDK	Statut Octobre 2014	Période de maintenance
1.8	0.45	Java SE 8	Kenaï	-	1.8 	Stable, actuel, version 1.8.0.45 proposée aux utilisateurs (8u60 Build b05 pour les développeurs)	
1.7	0.80	Java SE 7	Dolphin	(en) JSR 336 	1.7 	Stable, actuel, version 1.7.0.80 proposée aux développeurs et utilisateurs	2011-2015 ou +
1.6	0.45/0.51	Java SE 6	Mustang	(en) JSR 270 	1.6 	Stable, actuel, version 1.6.0.45 ³ proposée aux utilisateurs (1.6.0.51 sur MacOSX)	2005-2013 ⁴
1.5	0.22	J2SE 5.0	Tiger	(en) JSR 176 	1.5 	En fin de vie	2002-2006
1.4	2.19	J2SE 1.4	Merlin	(en) JSR 59 	1.4 	Obsolète	2000-2004
1.3	1.29	J2SE 1.3	Kestrel	(en) JSR 58 	1.3 	Obsolète	2000-2001
1.2	-	Java 1.2		(en) JSR 52 	1.2 	<i>N'est plus soutenu de façon active</i>	2000-2006

- Objet**
- Portable**
- Compact**
- Robuste**
- Orienté Réseau**
- Multi-tâches**

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

□ Java est un langage objet

- Des objets constituent une abstraction du monde réel.
- Ils communiquent entre eux par l'envoi de messages.



```
class Interrupteur {  
    ...  
    Ampoule a = new Ampoule();  
    a.allumer();  
    a.ajusterLuminosite(10);  
    ...  
}
```

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

- **Avec d'autres langages de programmation, un programme compilé sous Windows ne s'exécute pas dans un environnement Unix.**
- **Java est un langage portable : WRITE ONCE, RUN ANYWHERE**
 - Le même code Java peut s'exécuter sur toutes les plates-formes qui supportent Java, sans recompilation.
 - Les programmeurs Java n'ont pas à gérer les conversions de types primitifs d'une plate-forme à l'autre (integer, float...).
- **La portabilité Java est basée sur le concept de machine virtuelle.**

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

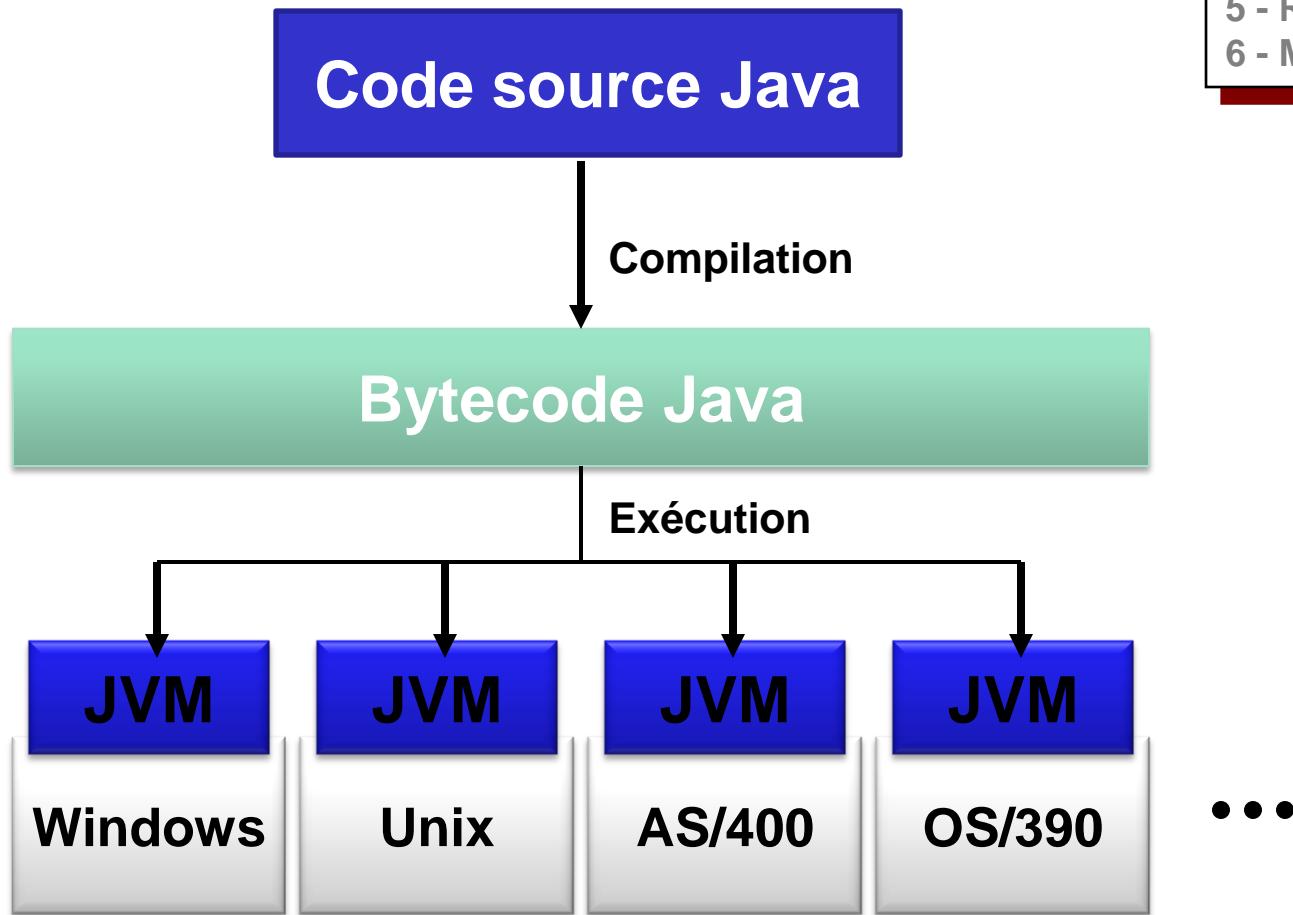
❑ Principe de la machine virtuelle

- le compilateur ne génère pas un exécutable pour un processeur particulier
- il génère du code binaire (*bytecode*) pour un processeur virtuel (la machine virtuelle)

❑ Portabilité

- la machine virtuelle est un programme disponible sur de nombreuses plates-formes
- le bytecode est traité par la machine virtuelle (tout comme les instructions d'un programme classique sont traitées par le processeur)
- ce traitement peut être une interprétation ou bien compilation dynamique.

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches



❑ Les fonctionnalités d'une JVM sont clairement spécifiées

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

- Plusieurs éditeurs peuvent implémenter la machine virtuelle
- Les principaux éditeurs sont Oracle, Sun et IBM
- La machine virtuelle est disponible pour de nombreux OS

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

□ Il existe différentes techniques d'implémentation de machines virtuelles

- **Interpréteur de bytecode**
 - ⇒ Première génération de compilateur
- **Compilateur natif** : transforme les fichiers sources ou bytecode en fichiers exécutables spécifiques au système d'exploitation.
 - ⇒ Bonnes performances au détriment de la portabilité.
- **Compilateur juste à temps** (just-in time / JIT) : compilation du code à la 1ère demande et mise en cache, puis exécution du code compilé plus rapide
 - ⇒ Performant et portable
- **Recompilateur dynamique** : la machine virtuelle analyse le comportement du programme et en recompile sélectivement les parties.
 - ⇒ Technique plus sophistiquée pouvant offrir de meilleures performances que les compilateurs JIT.

□ Java est compact

- Cet aspect a été privilégié depuis l'origine de Java
- Le bytecode dispose d'instructions de haut niveau
- La machine virtuelle fait une grande part du travail

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

□ Java est robuste

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches

- Le développeur ne manipule pas directement la mémoire
 - Pas de notion de pointeur
- La machine virtuelle gère la libération de la mémoire
 - Les objets sont détruits par le 'garbage collector' (ramasse miettes)
 - Le travail du développeur est simplifié
 - Pas de risque de libérer deux fois la même zone ou d'oublier de libérer
- Contrôle de type très strict
- Tous les attributs sont initialisés (pas de valeurs résiduelles)
- Les bornes des tableaux sont toujours contrôlées
- Les erreurs potentielles doivent être traitées par le développeur : notion d'exceptions

□ Java est orienté réseau

- Java a été adapté pour l'Internet : notion d'applet
- de nombreuses librairies permettent de simplifier le travail du développeur : client HTTP, FTP, communication entre objets Java distants

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau**
- 6 - Multi-tâches

□ Java est multi-tâches

- 1 - Objet
- 2 - Portable
- 3 - Compact
- 4 - Robuste
- 5 - Réseau
- 6 - Multi-tâches**

- Java intègre la notion de threads : la machine virtuelle peut gérer l'exécution de plusieurs « parties du programme » simultanément
- La machine virtuelle pourra profiter de plusieurs processeurs s'ils sont disponibles (vrai parallélisme)
- Les programmes Java qui utilisent le multitâches restent portables
- Cette fonctionnalité permet à Java d'être très bien adapté à l'écriture d'applications s'exécutant sur un serveur (nécessité de traiter simultanément des requêtes de plusieurs utilisateurs)

- ❑ Java est plus qu'un langage
- ❑ Le JRE (Java Runtime Environnement) contient
 - une machine virtuelle (Java Virtual Machine)
 - un ensemble de librairies (Java Application Programming Interface)
- ❑ Exemple de librairies (regroupées dans différents packages)
 - JDBC : Java DataBase Connectivity
 - AWT : Abstract Window Toolkit
 - RMI : Remote Method Invocation
 - JNI : Java Native Interface
- ❑ Java est utilisable sur différentes plates-formes
 - Linux, Windows mais aussi Java Card, serveur z/Os...

- Java est une synthèse de nombreux autres langages**

- C'est un langage objet qui peut être directement comparé à C++ et Smalltalk**

□ Java et C++

- un point commun : la syntaxe
- plus simple : élimination de certains aspects difficiles
 - pointeurs
 - héritage multiple
 - gestion de la mémoire
- plus robuste : gestion automatique de la mémoire
- plus objet : C++ autorise une programmation mixte
- moins proche de la machine
 - mais possibilité d'utiliser JNI pour appeler des librairies C
- légèrement moins performant à l'origine
 - l'écart s'est beaucoup réduit avec les JVM optimisées
 - Java peut prendre l'avantage dans certains cas

□ Java et Smalltalk

- conceptuellement très proche : machine virtuelle, garbage collector, librairies riches, approche objet obligatoire
- Java est plus standard
 - bytecode standardisé : plusieurs éditeurs de machine virtuelle
 - librairies standardisées : plusieurs fournisseurs (IBM, Microsoft...)
 - accepté par de nombreux éditeurs
- Java est plus accessible
 - Java est typé
 - syntaxe moins perturbante
 - nombreuses informations disponibles
- Java est moins adapté aux développements génériques (typage fort et absence de métaclasser)
- Java est plus verbeux (typage)

□ Historiquement les versions de Java sont déterminées par la disponibilité du JDK (Java Development Kit) sur le site Web de Oracle (ex-Sun)

- JDK 1.0.2 : 1995
- Java™ 2 SDK 1.2 à 1.2.2 : de 12/1998 à 08/1999
- Java™ 2 SDK SE 1.3 à 1.3.3 : de 05/2000 à 05/2001
- Java™ 2 SDK SE 1.4 à 1.4.2 : de 02/2002 à 05/2003
- J2SE Development Kit 5.0 : 09/2004 (Tiger)
 - Depuis Tiger chaque version a deux numéros de version : un interne (1.5.0) et un externe (5.0)
 - Le kit de développement est revenu à la convention de nommage JDK et non plus Java 2 SDK (ou J2SDK).
- Java SE 6 : 12/2006 (Mustang)
 - Depuis Mustang, l'abréviation J2SE est remplacée par Java SE

→ Rachat de Sun Par Oracle en 2009

- Java SE 7 : 2010 (Dolphin)
- Java SE 8 : 2014 (Kenaï)

1. Présentation de Java

2. Java en ligne de commande

3. Présentation d'Eclipse

4. Rappels UML

5. Les bases du langage

6. Les concepts objet avec Java

7. Les exceptions

8. Les classes de base

9. Les collections

10. Les entrées / sorties (IO)

11. L'accès aux bases de données

12. Le mapping objet-relationnel

- Présentation du JDK**
- Structure des programmes**
- Exemple de programme**
- Éléments du langage java**

□ Contenu du kit de développement

➤ Outils

- Compilateur
- Interpréteur
- Débogueur
- Générateur de documentation

➤ Bibliothèques de classes

➤ Le code source des classes

➤ Des exemples

➤ Une base de données embarquée

➤ Documentation (à télécharger en supplément)

- Des outils
- Des classes

□ Disponible sur le site <http://java.sun.com>

- **Minimum requis pour exécuter un programme java**
- **Sous-ensemble du JDK**
 - Bibliothèque des classes (bytecode seulement), sous forme de fichiers jar ('rt.jar', ...)
 - Interpréteur
- **A déployer sur toute machine où l'on souhaite simplement exécuter des applications Java**

□ Code Java

- Physiquement : des fichiers .java organisés en répertoires
- Logiquement : des classes(*) organisées en paquetages (**)
- Une classe correspond en général à un fichier de même nom
 - Source avec l'extension '.java' : **Client.java**
 - Après compilation avec l'extension '.class' : **Client.class**

□ Une application

- Regroupe un ensemble de fichiers
- Possède un « point d'entrée » (dépend du type d'application)

□ Application autonome

- Exécutée à l'aide de l'interpréteur ou bien compilée nativement
- Point d'entrée : service spécial « **main** »

□ Applet

- Application graphique embarquée dans une page HTML
- S'exécute au sein d'un navigateur qui invoque les services de l'applet, en particulier le service « **paint** » pour la dessiner
- Modèle de sécurité restreint

□ Servlet

- Application serveur qui répond aux demandes d'un client HTTP
- S'exécute au sein d'un serveur d'applications qui invoque les services du servlet, en particulier le service « **service** » pour demander de traiter une requête du client

```
public class Bonjour {  
    public static void main(String args[]) {  
        System.out.println("Bonjour tout le monde");  
    }  
}
```

- **main**
 - point d'entrée pour exécution d'un programme
- **void**
 - la méthode ne retourne aucun résultat
- **static**
 - méthode appliquée à la classe et non à une instance
- **les arguments de main sont des chaînes de caractères**
- **System.out.println("...") affiche sur la sortie standard**
- **le caractère ; est un terminator d'expression**



Les majuscules ont leur importance comme la plupart des autres signes.



Le code sera plus amplement décrit plus tard.

□ Classes de base

- Constituent un noyau pour le développement
- Regroupées en **packages** (concrètement des répertoires)
 - lang, awt, util, applet, sql, ...
- Développées en java
 - Certaines sont en fait du code natif

□ Fournies sous forme de fichier compressé

- **src.jar** ou **src.zip** : code source (fichiers .java) à la racine du JDK
- **rt.jar** : bytecode résultat de la compilation (fichiers .class) dans le répertoire **lib** du JRE

□ Packages java

- Standards
- Utilisables par le développeur

□ Packages com

- Éditeurs

□ Packages sun

- Internes

□ Packages sunw

- Internes
- Spécifiques à Windows

- **java.lang**
 - Cœur du langage
- **java.awt**
 - Interfaces graphiques
- **java.applet**
 - Applets
- **java.beans**
 - Composants
- **java.io**
 - Entrées/sorties, flux

java.net

- sockets, URL

java.rmi

- Accès distant à des objets

java.security

- Authentification, cryptage

java.util

- Collections, date

java.sql

- Java DataBase Connectivity (accès aux bases de données)

- **JDK conçu pour être utilisé dans une console ou par des scripts**
- **Nécessite de spécifier le chemin d'accès aux outils**
 - Variable d'environnement *PATH*
 - *SET PATH=%PATH%;"C:\Program Files\Java\jdk1.8.0_45\bin"*
 - Attention aux guillemets si le chemin d'accès au JDK contient des espaces
- **Nécessite de spécifier les chemins des librairies**
 - Variable d'environnement *CLASSPATH*
 - Indique les fichiers jar et les répertoires contenant des classes
 - Précise dans quel ordre la recherche s'effectue
 - *SET CLASSPATH=.;C:\Program Files\Java\jdk1.8.0_45\re\Vib*
- **Problèmes de conflits en cas d'environnements multiples**
 - Les outils disposent maintenant d'une option –*classpath*
- **Attention le séparateur d'éléments du *classpath* est ; pour Windows et : pour Unix**

- **Produit les fichiers compilés (.class) à partir des sources (.java)**
 - Sous forme de bytecode
- **Ligne de commande**
 - *javac Fichier.java*
 - Possibilité de compiler tous les fichiers du répertoire :
*javac *.java*
- **Utilise la variable CLASSPATH mais il est maintenant recommandé d'utiliser l'option -classpath à la place**
 - *javac -classpath=.;netlibs.jar;tests.jar Reseau.java*

- ❑ *-classpath*

- Surcharge la variable *CLASSPATH*

- ❑ *-d*

- Permet d'indiquer un répertoire destination pour les fichiers .class

- ❑ *-verbose*

- Affiche des informations pendant la compilation

- ❑ *-help*

- Aide sur les options
 - Fournit d'autres options non mentionnées ici

- Exécute le bytecode java

- Ligne de commande

java [-options] NomClasse [arguments]

- Utilise la variable *CLASSPATH* (option *-classpath* recommandée)
- Exécute la méthode main de la classe passée en paramètre
 - *java Fichier*
- Le nom de la classe doit être complet (packages + classe)
 - *java com.neobject.formation.BankonetTest*
- Arguments : chaînes de caractères séparées par des espaces
 - *java com.neobject.formation.BankonetTest compte1 compte2 compte3*

- ❑ **-classpath ou -cp**

- Spécifie le chemin de recherche des classes
 - Substitut recommandé à la variable CLASSPATH

- ❑ **-version**

- Indique la version du JDK utilisée

- ❑ **-Dpropriété=valeur**

- Permet de passer des propriétés (préférences) au programme

- ❑ **-verbose**

- Affiche des informations pendant l'exécution

- ❑ **-help**

- Affiche la liste des options (d'autres options sont disponibles)

❑ Format compressé (ZIP) pour déployer les applications

- Sources et/ou bytecode et ressources
- Descripteur du contenu : *META-INF/MANIFEST.MF*
- Extension : *.jar*

❑ Exécution :

- *java –jar nomArchive.jar*
- Peut être associée à l'extension *.jar* sous Windows

❑ Outil pour les manipuler : jar

- Crédit
• *jar cf monAppli.jar Exemple.class*
- Extraction
• *jar xf monAppli.jar*
- Tout autre outil capable de manipuler des zips fonctionne aussi

- ❑ Outil en ligne rudimentaire
- ❑ Utilise la variable **CLASSPATH** (option **-classpath** recommandée)
- ❑ Utilisation :
 - compiler les classes
 - lancer jdb à la ligne de commande : **jdb MaClasse**
 - mettre au moins un point d'arrêt : **stop in MaClasse.main**
 - exécuter la classe : **run MaClasse** (s'arrête au 1er point d'arrêt)
 - avancer : **step, next, cont**
 - inspecter : **dump monObjet, print monObjet, locals**
 - voir où on en est : **list**
 - pour + d'informations : **?** ou **help**

□ Commentaires sur plusieurs lignes

- Encadrés par /* et */
- Pour les détails techniques

□ Commentaires sur une seule ligne

- Commence par //
- Pour les détails au milieu du code

□ Commentaires javadoc

- Encadrés par /** et */
- Pour la documentation technique
- Syntaxe normalisée
- Automatiquement repris par les outils de production de documentation

- Reprend les commentaires encadrés par `/** et */`
- Ligne de commande
 - `javadoc Fichier.java`
 - Nombreuses options (voir `-help`)
- Documentation au format HTML
 - Possibilité d'inclure du code HTML
- Possibilité d'inclure des tags dans les commentaires (`@tag`)
 - `@author fguibert`
 - `@see com.neobject.bankonet.Client`
 - `@version 2.4, 28/05/2015`
 - ...
- Documentation API JDK elle-même générée par *javadoc*

□ Sites principaux :

<https://jdk8.java.net>

<http://www.oracle.com/technetwork/java/javase>

- API complète des classes de base
- Outils standards du JDK (javac, java, jdb,...)
- Exemples
- Tutoriaux
- Changements entre les versions
- ...

□ Réaliser les travaux pratiques suivants:

- TP 1

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

□ Eclipse est l'environnement de développement le plus utilisé en Java

- Open source et gratuit
- Concurrents principaux:
 - NetBeans (open source, sponsorisé par Oracle)
 - IntelliJ IDEA (commercial, JetBrains)
 - JBuilder (commercial, Embarcadero)

□ Eclipse couvre un large panel de fonctionnalités

- Éditeur de code Java
- Navigateur de classes
- Débogueur
- Outils de recherche
- Gestion d'un historique local
- Travail de groupe : lien et outils pour CVS, SVN, Github...
- Système d'aide
- ...

□ Architecture

- Spécifications OSGi (projet Equinox)
- Platform Runtime : démarre la plateforme et gère les plug-ins.
- Bibliothèques graphiques
 - SWT : bibliothèque de composants graphiques.
 - JFace : abstraction des widgets SWT, support de ressources ...
 - Workbench : permet de manipuler vues, éditeurs, perspectives ...

□ Projets de la fondation Eclipse

- Spring Tools Suite (STS)
- Web Tools Platform project (WTP)
- Business Intelligence and Reporting Tools Project (BIRT)
- C/C++ Development Tools Project (CDT)
- Rich Client Platform (RCP)

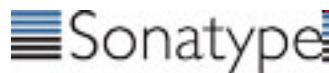
□ Pour dynamiser la création d'extensions, IBM a placé en 2001 le cœur de son environnement dans le domaine public

- Produit nommé Eclipse (Fondation Eclipse)
- www.eclipse.org
- Gratuit et open-source
- Principalement développé par la société OTI (filiale d'IBM). OTI est historiquement spécialiste des environnements Smalltalk, c'est aussi la société qui a développé VisualAge for Java
- Historique des versions :
 - Version 1.0 : novembre 2001
 - Version 2.0 : juin 2002
 - Version 3.0 : juin 2004
 - Version 3.1 : juin 2005
 - Version 3.2 : juin 2006 (Callisto)
 - Version 3.3 : juin 2007 (Europa)
 - Version 3.4: juin 2008 (Ganymede)
 - Version 3.4.2 : février 2009
 - Version 3.6 : juin 2010 (Helios)
 - Version 3.7 : mai 2011 (Indigo)
 - Version 4.2 : juin 2012 (Kepler)
 - Version 4.4 : juin 2014 (Luna)
 - **Version 4.5 : planifiée en juin 2015 (Mars)**

- **Eclipse est maintenant beaucoup plus indépendant de la société IBM**
 - IBM siège au consortium Eclipse de même que de nombreux autres membres

- **De nombreux éditeurs proposent des extensions d'Eclipse (plugins), en Open source ou dans une offre commerciale**
 - Embarcadero avec JBuilder, qui est désormais basé sur Eclipse depuis la version 2007 du produit
 - Oracle Workshop basé sur Eclipse
 - MyEclipseIDE de Genuitec
 - ...

- Le projet Eclipse est dirigé par un consortium regroupant de grands éditeurs



□ **IBM réalise son propre packaging d'Eclipse pour fournir un atelier de développement Java:**

- Rational Application Developer (RAD)
- Socle composé d'Eclipse et de plugins pour:
 - Développement Java / JEE
 - Développement Web 2.0 : HTML 5/ CSS 3, Mobile, OSGi, JavaScript, JSP, XML, XSL
 - Serveurs d'application de test : WebSphere (WAS) et Portal Server
 - Assistants bases de données
 - Assistants Services Web / SOA / REST
 - Développement d'EJB
 - Profiling
 - Travail de groupe : lien avec Rational ClearCase intégré (SCM, outil de gestion de configuration, comme CVS, Subversion, etc.)
 - Modélisation UML

- RAD a remplacé WebSphere Studio Application Developer (WSAD) en 2005.
- La version courante est la v9.1.1 datant d'Octobre 2014.
- Compatibilité Eclipse et Java :
 - RAD 6.0 : Eclipse 3.0 et JDK 1.4.2
 - RAD 7.0 : Eclipse 3.2 et JDK 5
 - RAD 7.5 : Eclipse 3.4.2 + WTP et JDK 5 + JRE 6
 - RAD 8 : Eclipse 3.6.2 + WTP et JDK 6 + JRE 6
 - RAD 8.5 : Eclipse 3.6.2.3 + WTP et JDK 7 + JRE 7
 - RAD 9 : Eclipse 4.2.2 + WTP et JDK 8 + JRE 8
- Très gourmand en ressources

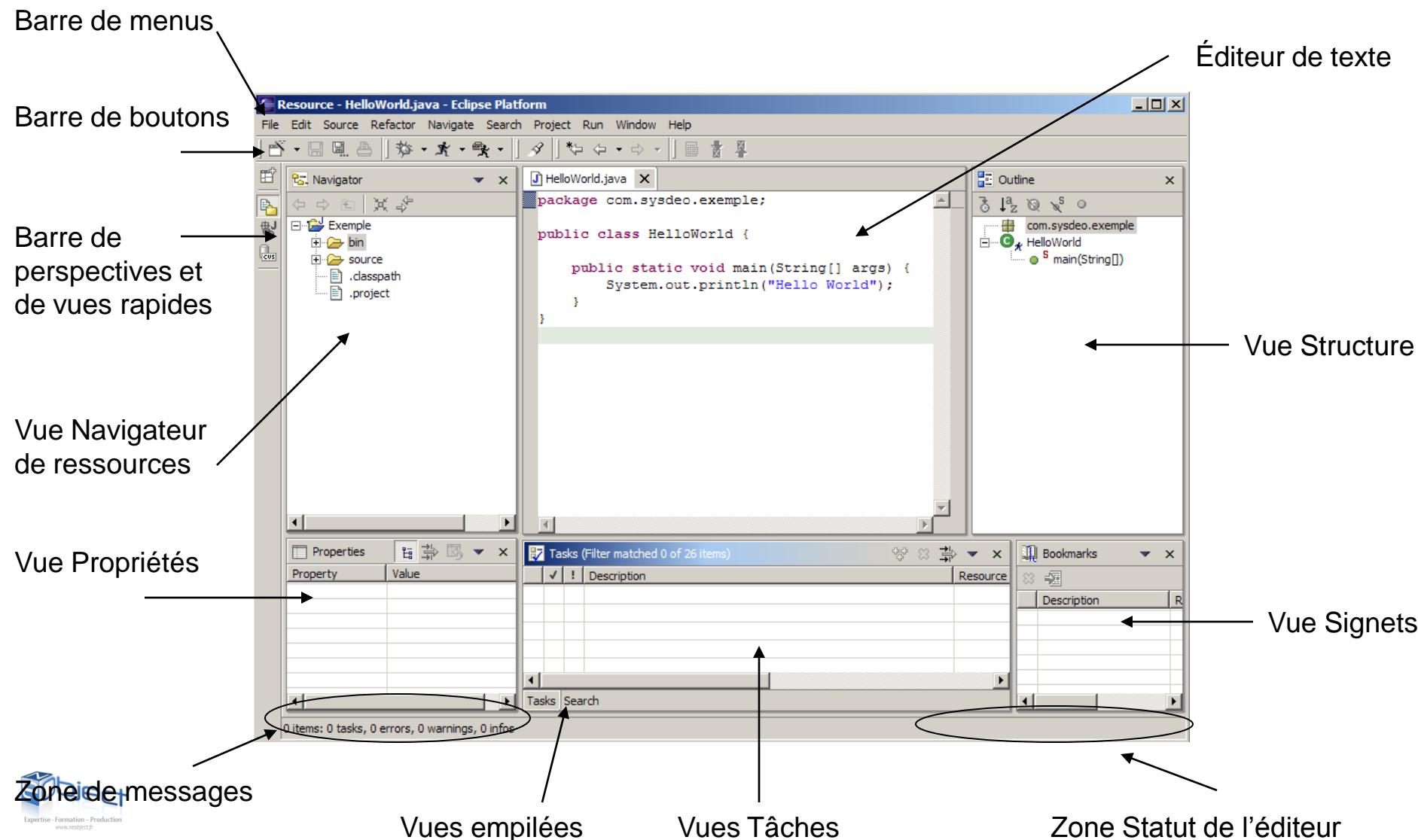
□ Installation :

- Installer un JRE
- Décompresser le fichier zip de la distribution Eclipse téléchargée sur ***http://www.eclipse.org***

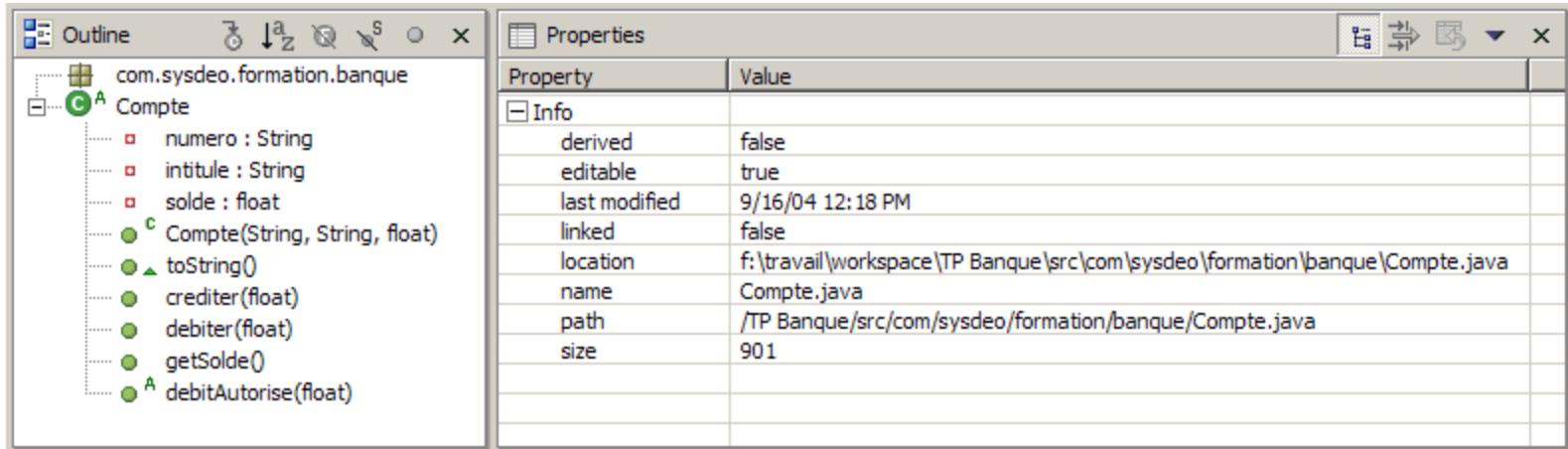
□ Lancement :

- Exécuter le programme `eclipse.exe`
- A la première exécution, un sous-répertoire 'workspace' est créé
 - Contient les projets et tous les paramètres de configuration
 - Pour faciliter la mise à jour d'Eclipse, ne pas laisser Eclipse créer ce répertoire dans son arborescence mais utiliser l'option `-data` pour indiquer ce répertoire, par exemple :
 - `eclipse.exe -data d:\travail\workspace`

- Vue**
- Éditeur**
- Perspective**
- Plugin**



- Une vue est une présentation graphique de ressources
- Offre un point de vue sur une ressource donnée
- Peut être spécifique à certains types de ressources
- Plusieurs vues différentes peuvent afficher la même ressource
 - Ex : vues **Outline** et **Properties** sur la ressource **Compte.java**

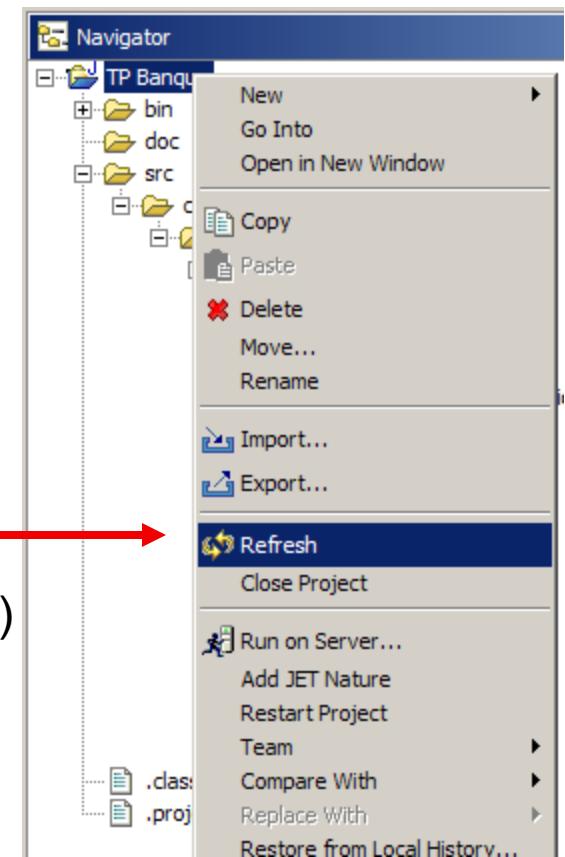


❑ Explorateur de fichiers classique

- Créer répertoires et fichiers
- Copier
- Déplacer
- Supprimer
- Renommer

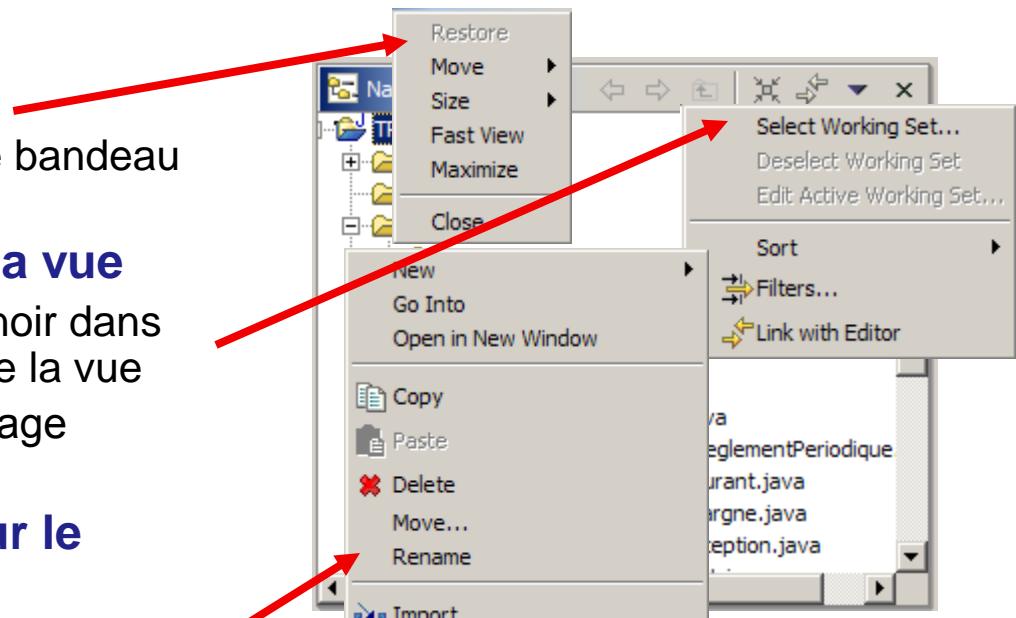
❑ Fonctionnalité importante : Refresh

- Synchroniser avec le système de fichiers
- Nécessaire après modification d'une ressource à l'extérieur d'Eclipse
(fichiers ou répertoires ajoutés, modifiés,...)



❑ Un menu spécial vue

- Dans son icône
- Accès via clic droit dans le bandeau de la vue



❑ Un menu fonctionnel sur la vue

- Accès via le petit triangle noir dans le coin droit du bandeau de la vue
- Généralement lié à l'affichage
 - Filtrage, tri, etc.

❑ Un ou plusieurs menus sur le contenu de la vue

- Accès via un clic dans la vue ou sur un élément ou une sélection dans la vue
- Les options affichées sont en général fonction de la sélection effectuée

❑ Boutons d'outils dans le bandeau de la vue

- Une sorte de vue permettant la modification d'une ressource

- Ressource modifiée marquée par une étoile



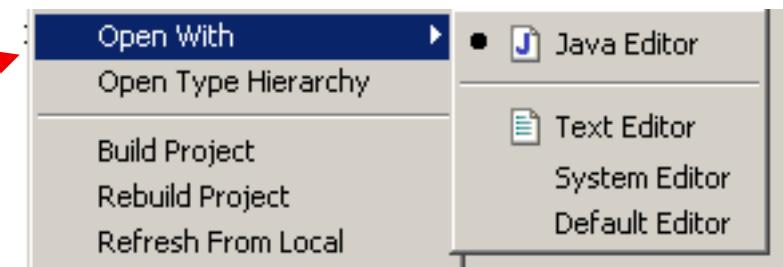
- A chaque type de ressource, un ou plusieurs éditeurs

- Les éditeurs sont de deux types :

- Interne : vue permettant d'éditer une ressource
 - Externe : programmes lancés par Eclipse

- Ouvrir un éditeur :

- Double-clic sur la ressource
 - Menu contextuel : 'Open et 'Open with'

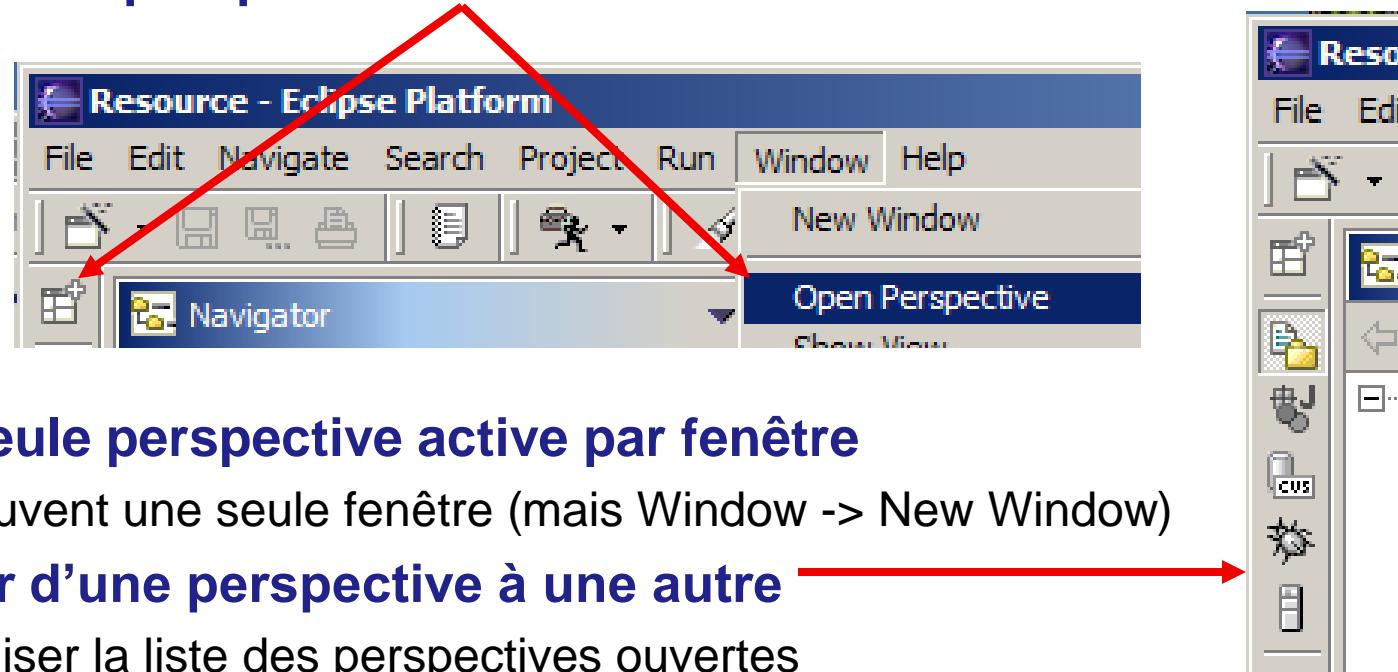


Une Perspective est un agencement prédéfini

- De plusieurs vues, menus et barres d'outils

 Correspond à une notion d'activité autour des ressources

- Des perspectives sont prédéfinies pour les tâches courantes

 Ouvrir une perspective **Une seule perspective active par fenêtre**

- Souvent une seule fenêtre (mais Window -> New Window)

 Passer d'une perspective à une autre

- Utiliser la liste des perspectives ouvertes

□ **Les vues peuvent être déplacées à l'intérieur d'une perspective**

- Principe du 'glisser - déposer'
- 'glisser' en sélectionnant la barre de titre et en restant enfoncé
- 'déposer' dépend du curseur affiché et de la vue 'survolée' :



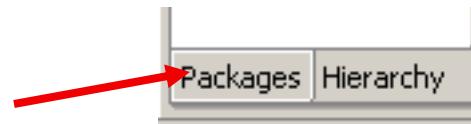
- Insérer au dessus, en dessous, à droite, à gauche de la vue survolée



- Impossible de déposer



- Empiler : affichage sous forme d'onglets
les onglets sont en dessous par défaut,
pour changer Preferences->Workbench->Appearance



- **Les vues peuvent être redimensionnées (déplacer la séparation)**
- **Pour agrandir une vue, double-cliquer sur sa barre de titre**

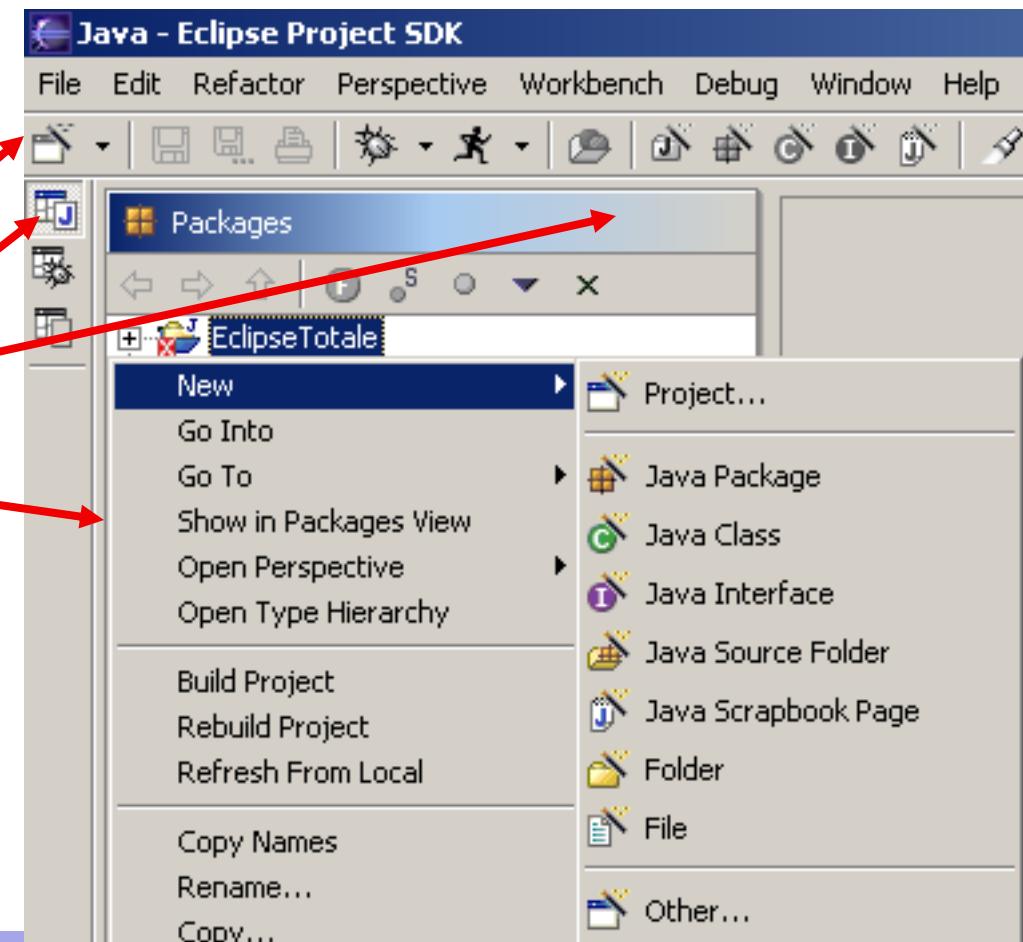
□ Menu Window (extrait)

- Open Perspective – ouvrir une perspective
- Show View – ajouter une vue
- Save Perspective As... – enregistrer une nouvelle perspective
- Reset Perspective – rétablir l'agencement par défaut
- Customize Perspective... – organiser menus et barres d'outils
- Close Perspective – fermer la perspective
- Close All Perspectives – fermer toutes les perspectives

□ La création de projets, de packages, de classes et d'interface se fait par des assistants.

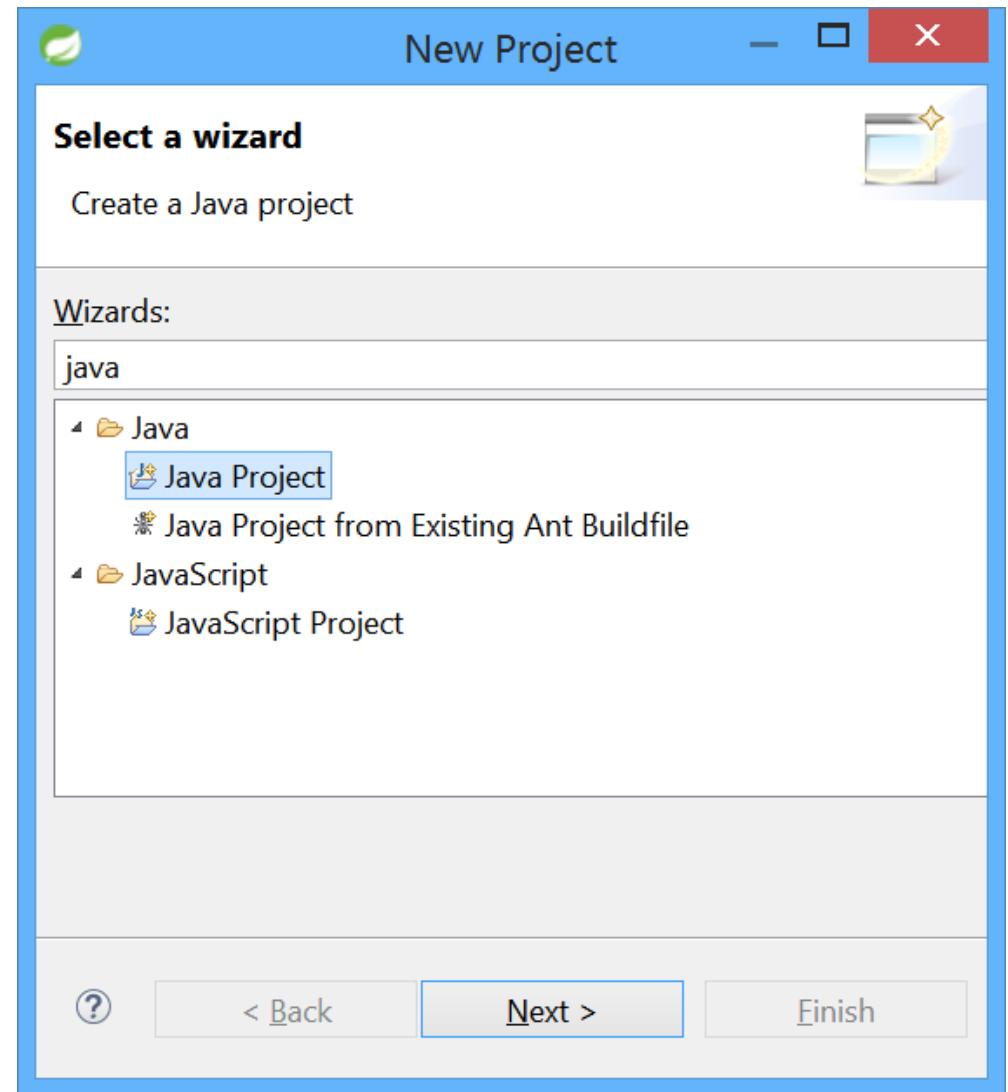
4 façons d'accéder à ces assistants :

- File -> New
- Barre d'outils
- Menu contextuel



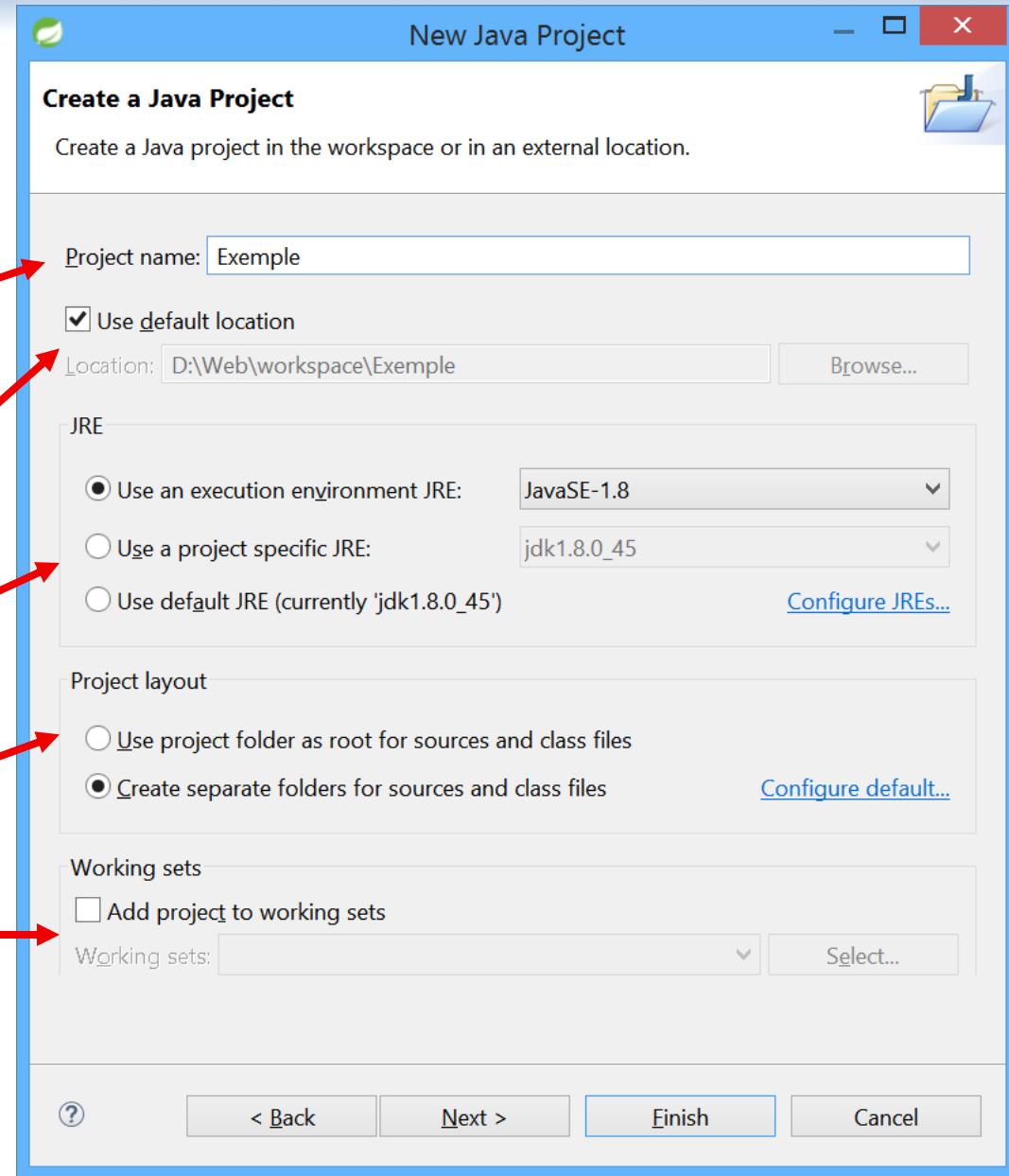
□ Eventuellement :

1ère page de l'assistant :
choix du type de projet



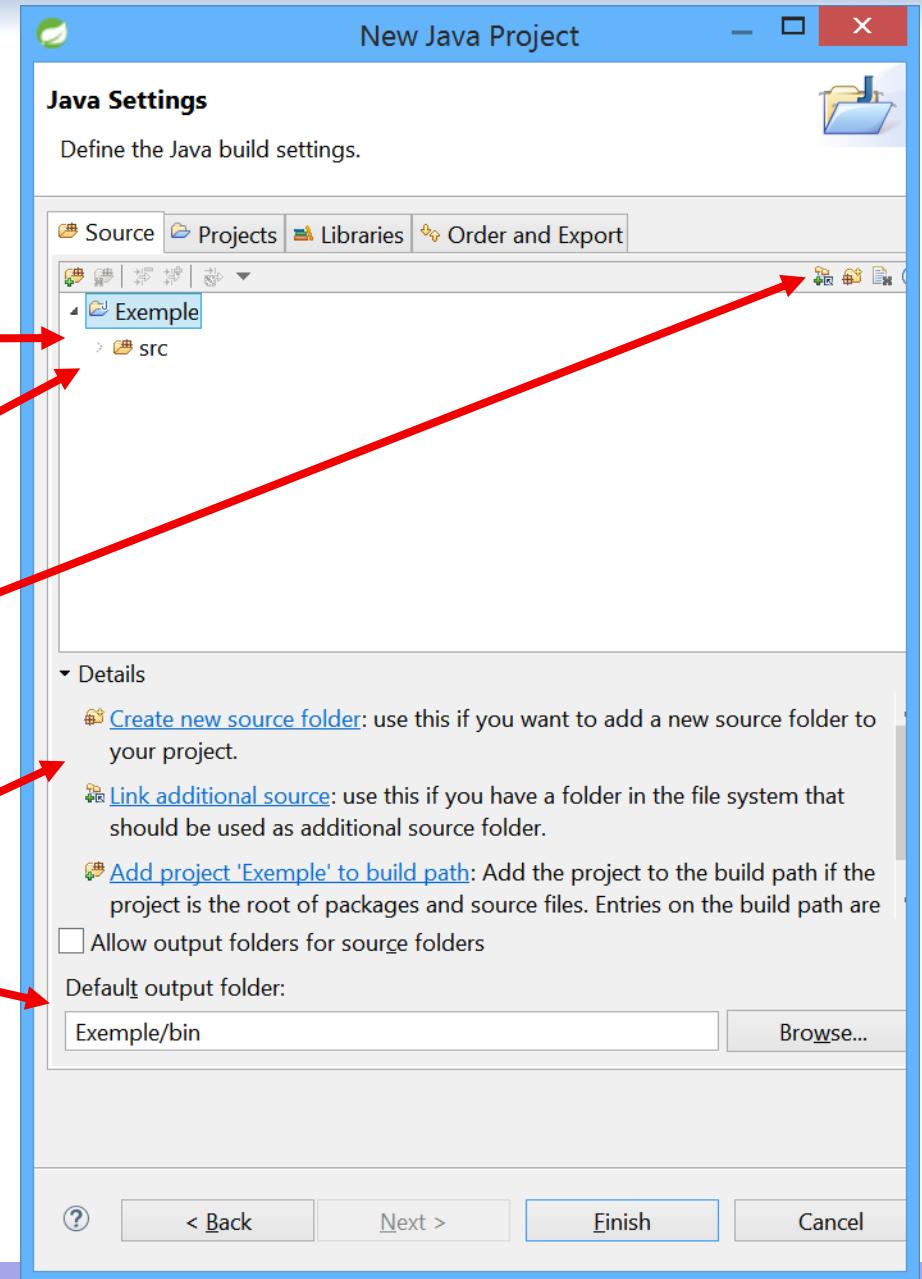
□ 2ème page de l'assistant :

- Nom du projet
- Emplacement du projet
le projet ne doit pas forcément être dans un sous-répertoire d'Eclipse
- Choix du JRE à adopter pour ce projet
- Configuration de la structure
- Choix éventuel d'un working set



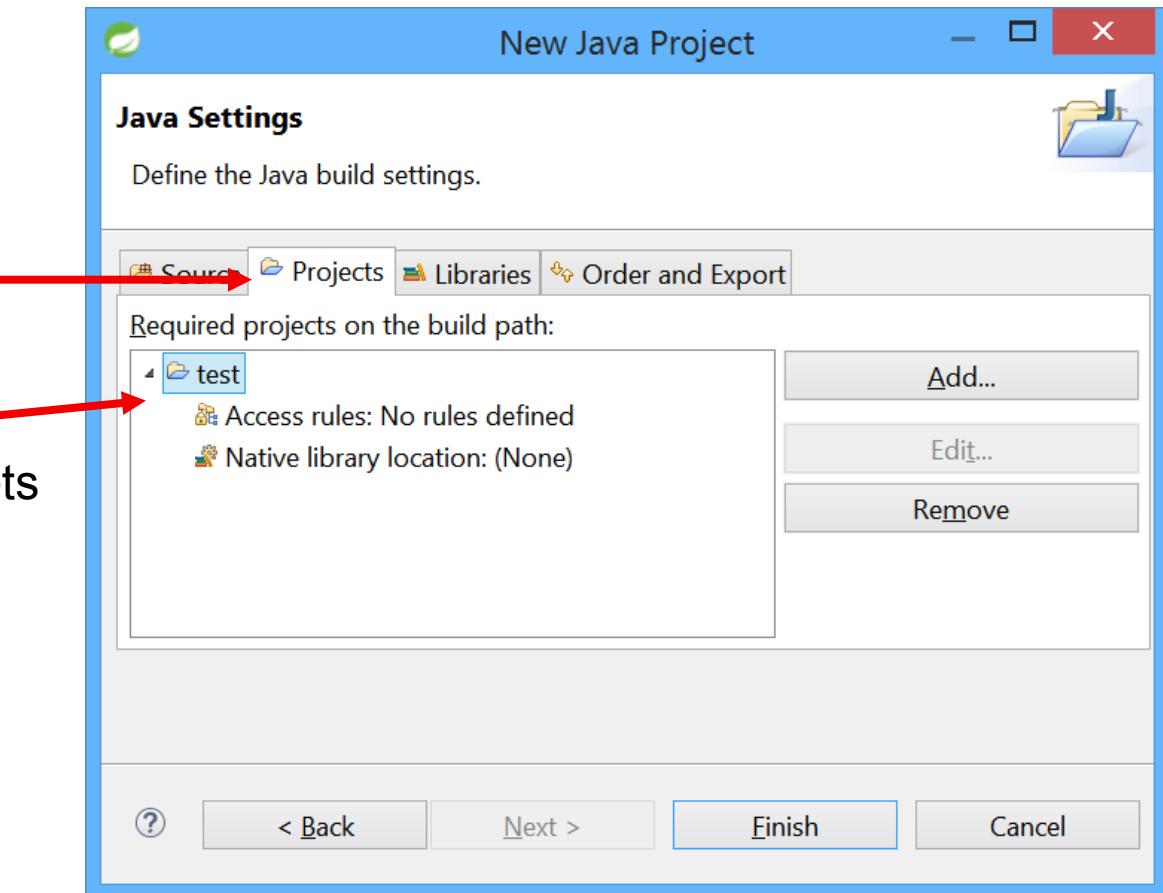
□ 3ème page de l'assistant,
onglet Source

- Dans un projet, plusieurs sous-répertoires peuvent contenir le code source
- Par défaut Eclipse indique que les fichiers sources sont à la racine du projet (*)
- Il est possible de créer de nouveaux répertoires destinés à contenir du code source
- Paramétrage avancé (**)
- Le code compilé se retrouve dans un seul répertoire (***)



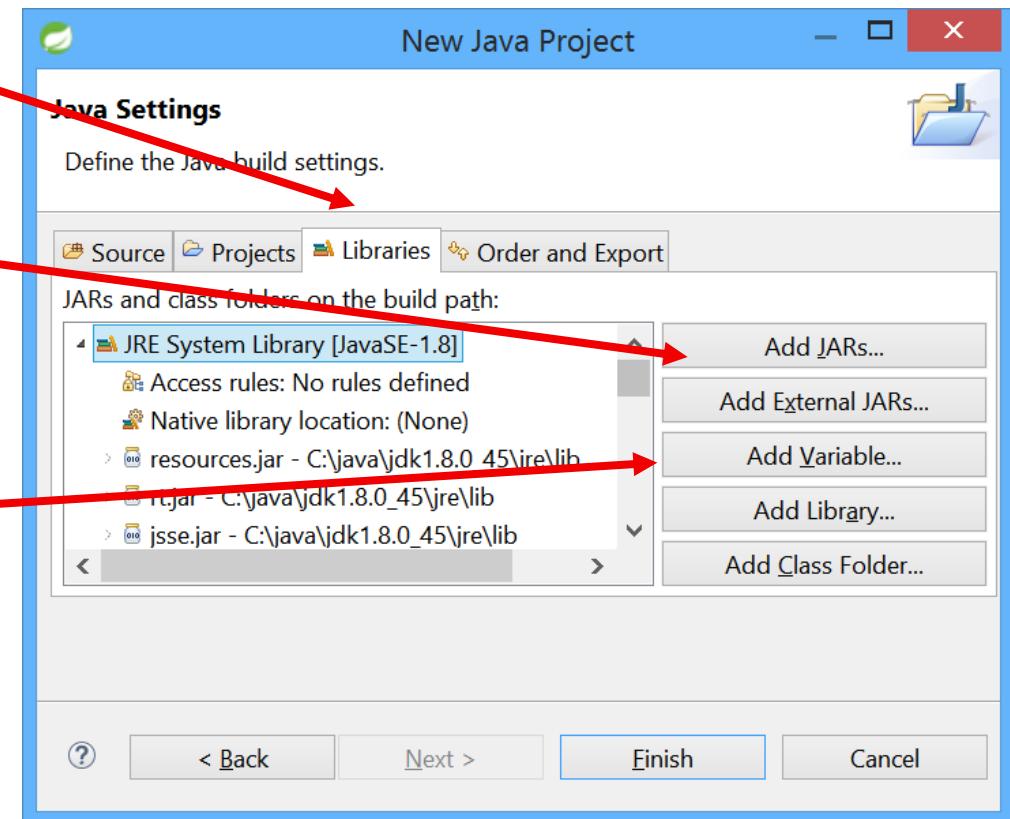
□ 3ème page de l'assistant, onglet Projets

- cet écran permet d'indiquer les projets ouvert dans le workspace dont dépend notre nouveau projet



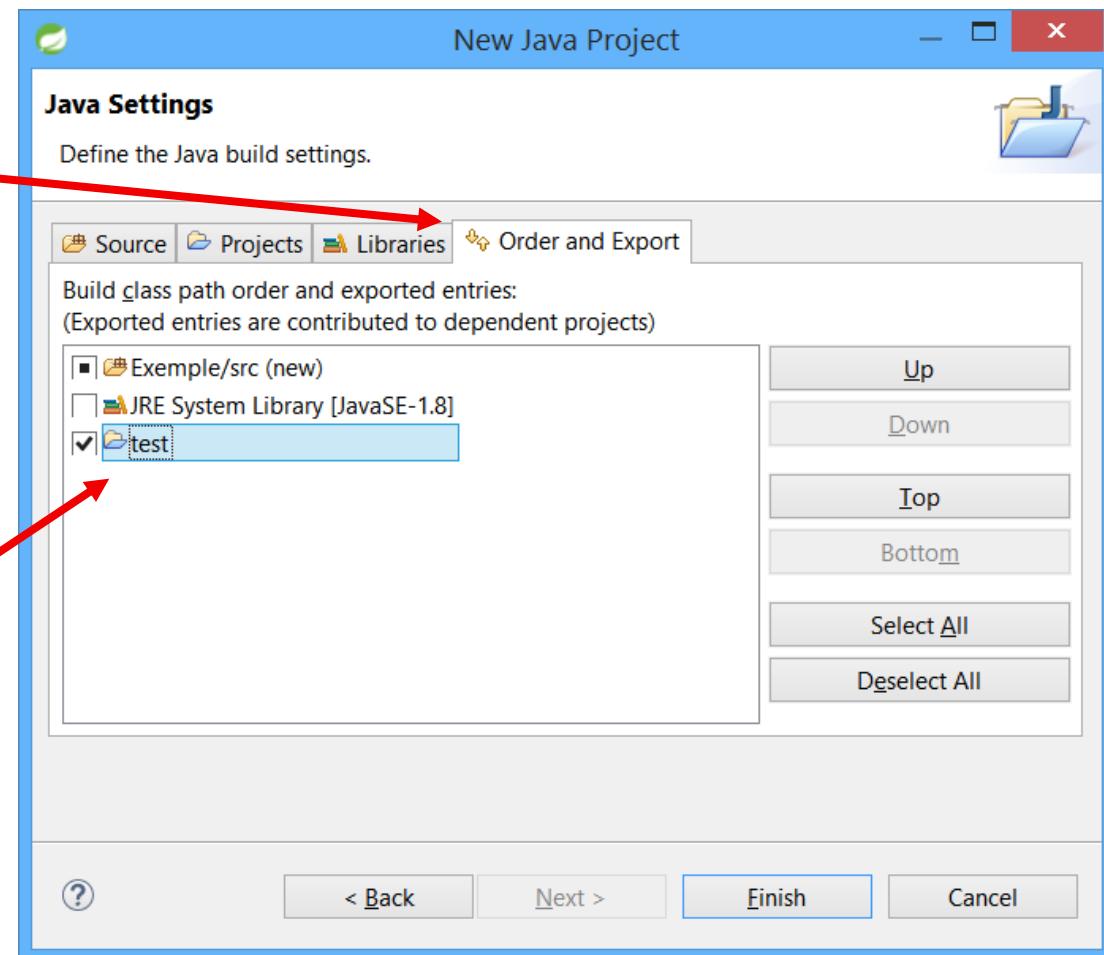
❑ 3ème page de l'assistant, onglet Bibliothèques

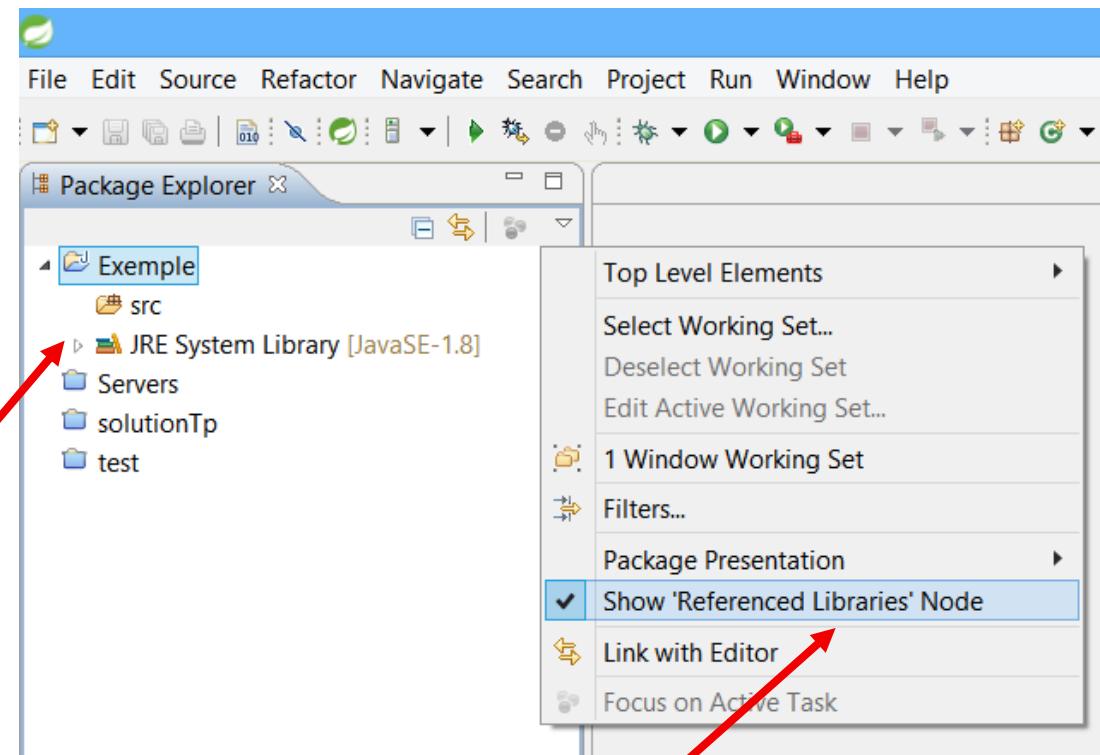
- le code de notre projet peut aussi faire appel à des classes chargées ou non dans Eclipse et stockées dans des répertoires ou des fichiers jar
- cf. préférences->Java->
Variables de chemin d'accès
aux classes (*)



□ 3ème page de l'assistant, onglet Ordre et exportation

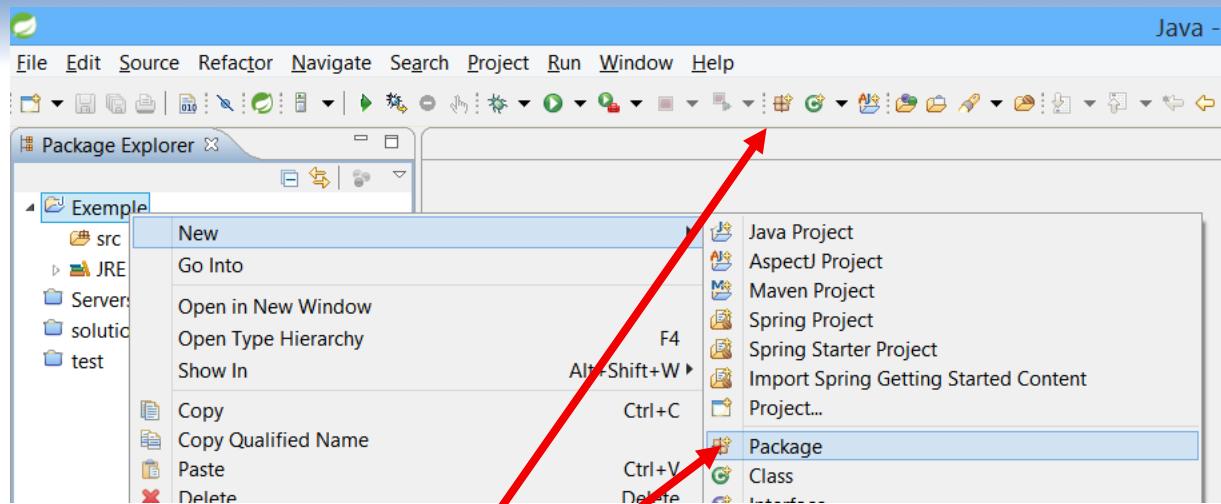
- ordonnancement des entrées du classpath
- paramétrages des entrées visibles par les projets qui dépendent de ce projet



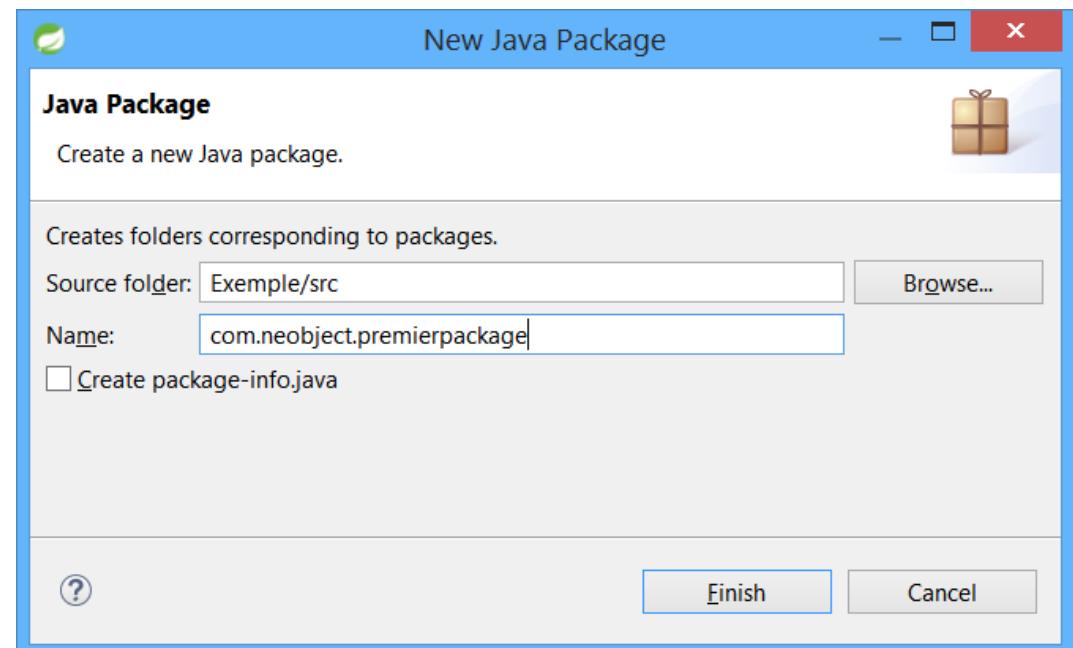


Liste des librairies référencées

Masquer les librairies

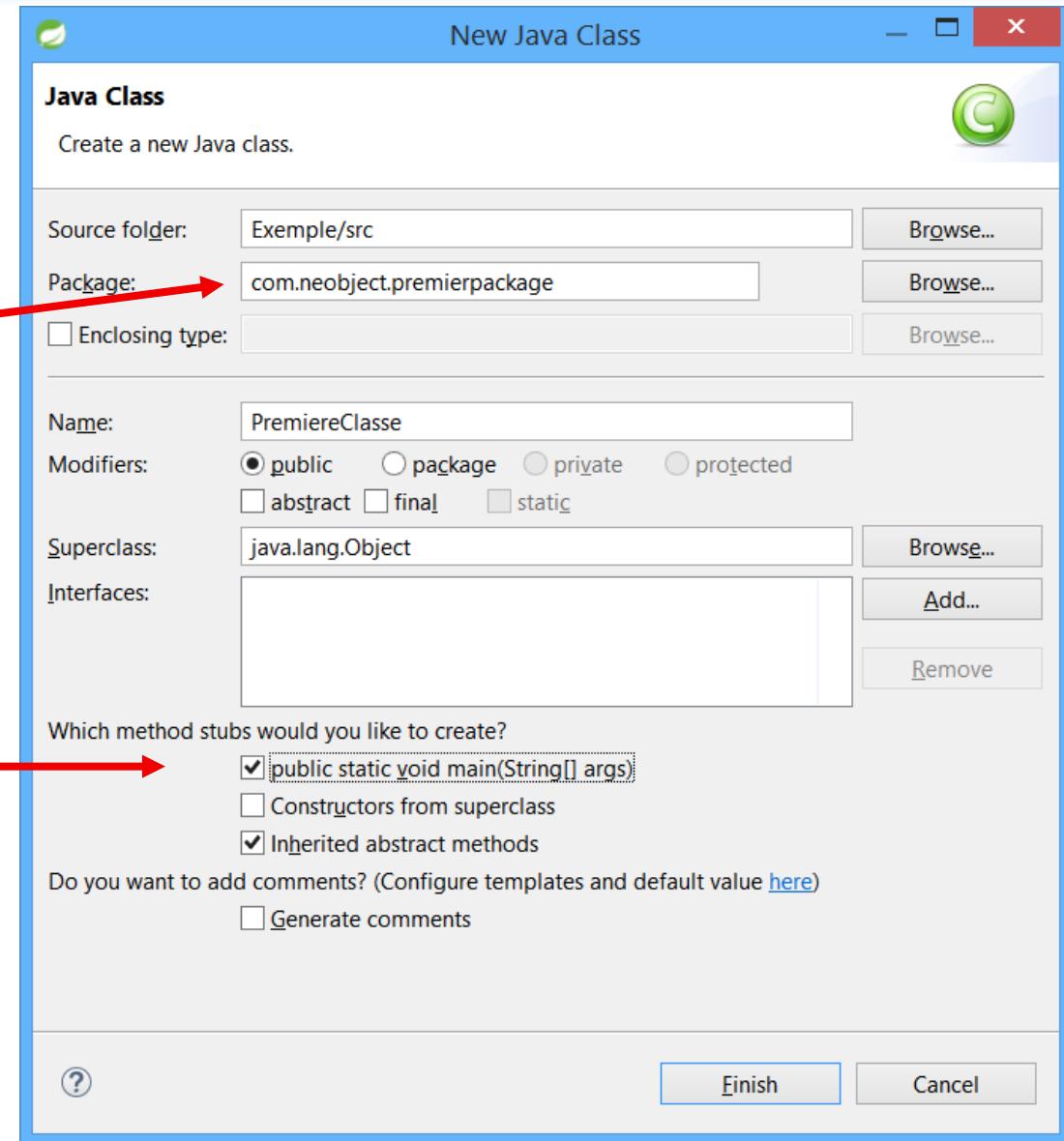


Points
d'entrée

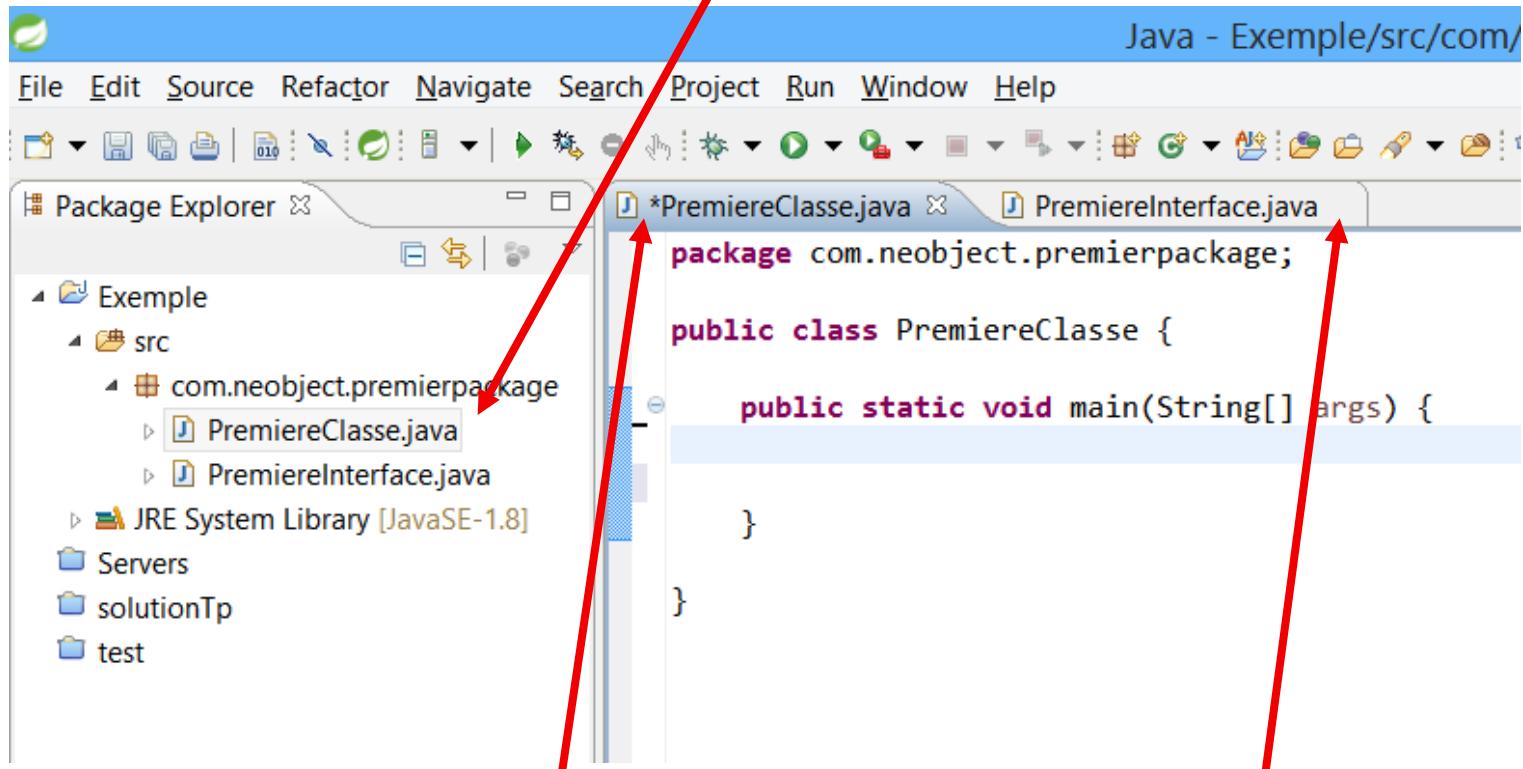


Préciser un package sinon
la classe sera dans
'le package par défaut'
(mauvaise pratique)

Création automatique
de méthodes



□ Pour ouvrir un type dans l'éditeur utiliser le double clic

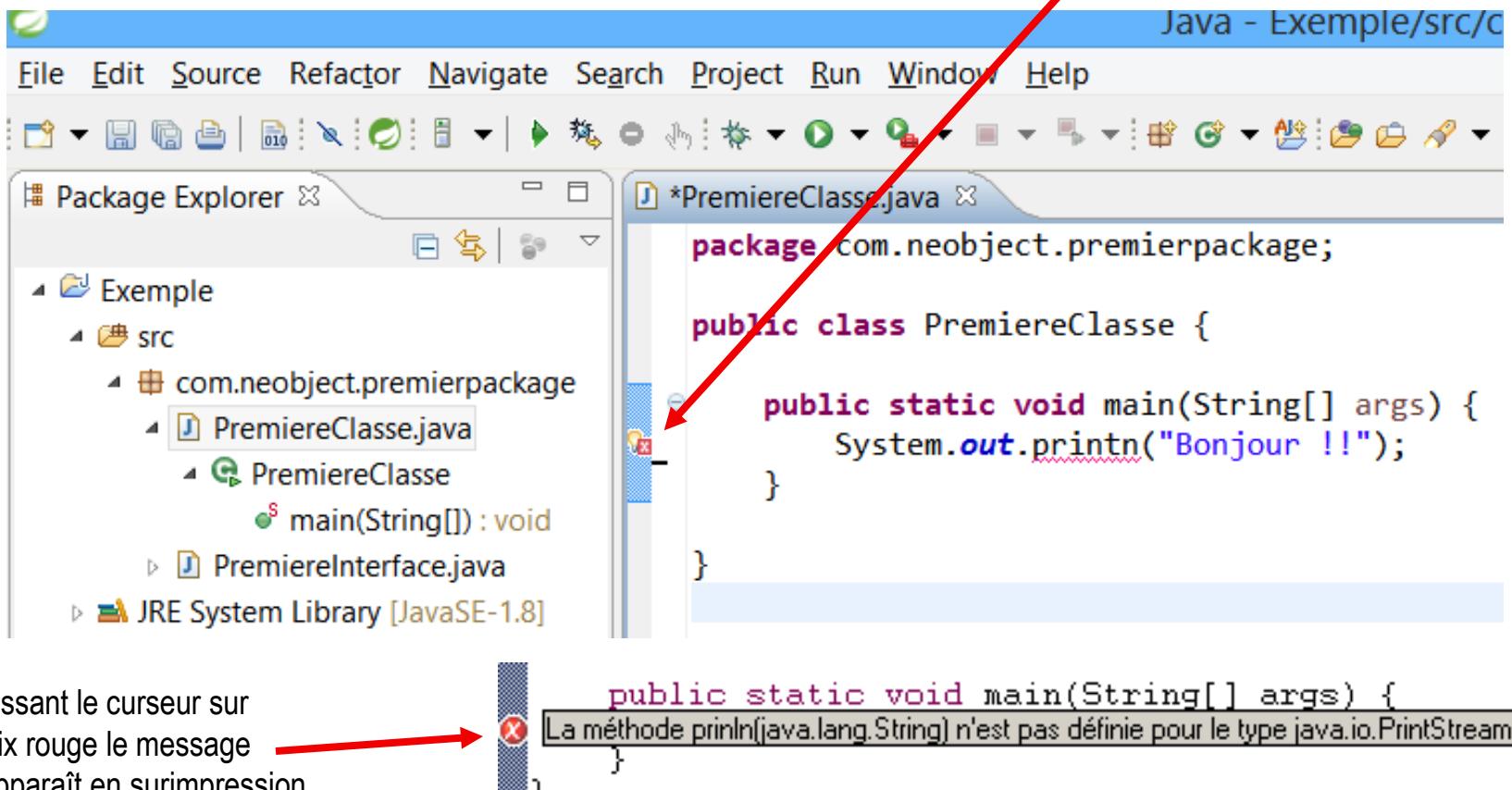


Le symbole * indique que le fichier est en cours de modification

Plusieurs fichiers peuvent être ouverts simultanément

- La compilation est faite automatiquement à la volée ou à l'enregistrement d'un fichier modifié (Ctrl-S ou menu contextuel->Sauvegarder)

Indique une erreur de compilation



□ Généralement

- Un fichier *plugin.xml*, indiquant comment le plugin se branche dans Eclipse (nouvelles vues, ajouts dans les menus, ...)
- Un fichier jar contenant le code du plugin

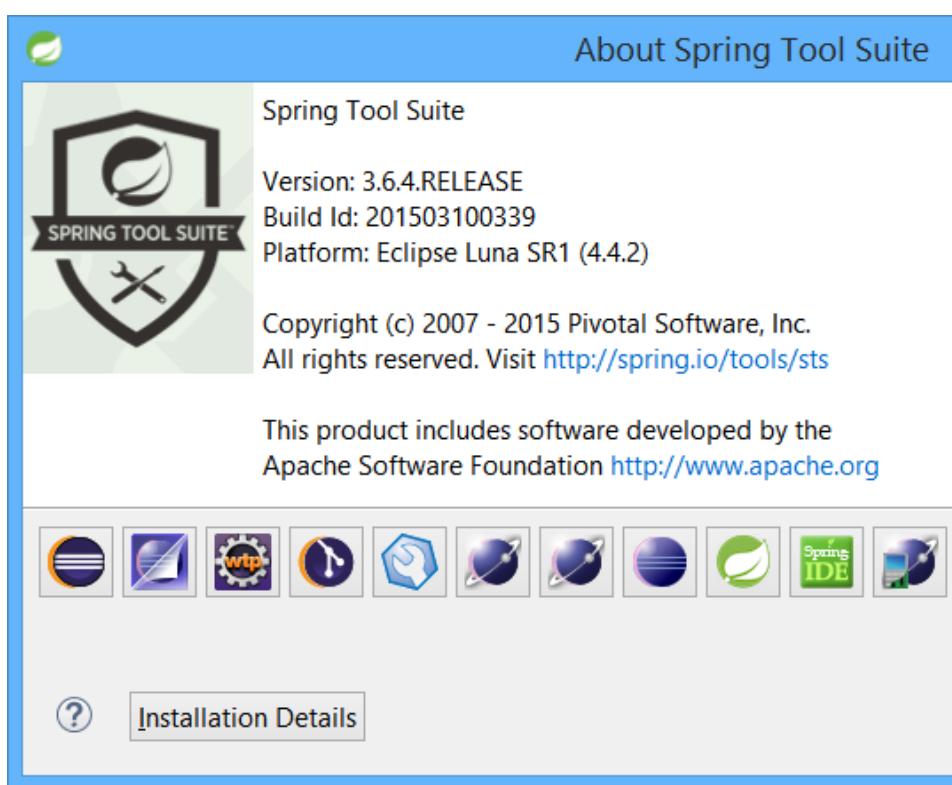
□ Installation d'un nouveau plugin

- Manuellement / pour les plugins ne disposant pas de sites Eclipse
 - copier le répertoire du plugin (*com.société.pluginname*) dans le répertoire plugins d'Eclipse
- Automatiquement pour les plugins disposant d'un *Update site*
 - Via le menu *Help* → *Software Updates* → *Find and install...*
 - Dans le dialogue *Feature Updates*, choisir entre mise à jour de plugins déjà installés ou d'installation de nouveaux plugins
 - Bouton *New remote site* pour déclarer l'URL d'un site d'update de plugin
 - Recherche et Installation du plugin
- Ne sera pris en compte qu'au prochain lancement d'Eclipse

□ Pour développer des plugins :

- Utiliser la perspective PDE (Plugin Development Environment)
- Crée un projet de type PDE Project et non Java Project

□ Via le menu Help -> About Eclipse Platform
 ➤ Bouton ‘Plug-ins’



Spring Tool Suite Installation Details

Installed Software	Installation History	Features	Plug-ins	Configuration
type filter text				
Signed	Provider	Plug-in Name	Version	Plug-in I...
<input checked="" type="checkbox"/>	Eclipse.org	Annotation Controller Plu...	1.1.300.v2013020...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse Web T...	Annotation Core Plug-in	1.1.300.v2013020...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse Web T...	Annotations Core	1.2.200.v2013111...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse.org	Annotations UI Plug-in	1.1.300.v2013020...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse.org	Ant Build Tool Core	3.3.1.v20150123-...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse.org	Ant Launching Support	1.0.400.v2014051...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse.org	Ant UI	3.5.500.v2014052...	org.eclipse...
<input checked="" type="checkbox"/>	Eclipse Orbit	Aopalliance Plug-in	1.0.0.v201105210...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Ant	1.9.2.v201404171...	org.apache...
<input checked="" type="checkbox"/>	Eclipse.org	Apache Batik CSS	1.7.0.v201011041...	org.apache...
<input checked="" type="checkbox"/>	Eclipse.org	Apache Batik GUI Utilities	1.7.0.v200903091...	org.apache...
<input checked="" type="checkbox"/>	Eclipse.org	Apache Batik Utilities	1.7.0.v201011041...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache BCEL	5.2.0.v201005080...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Codec ...	1.6.0.v201305230...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Collecti...	3.2.0.v201303021...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Compr...	1.6.0.v201310281...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Httpcli...	3.1.0.v201012070...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons IO	2.0.1.v201105210...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Lang	2.6.0.v201404270...	org.apache...
<input checked="" type="checkbox"/>	Eclipse Orbit	Apache Commons Loggin...	1.1.1.v201101211...	org.apache...

?

Legal Info

Show Signing Info

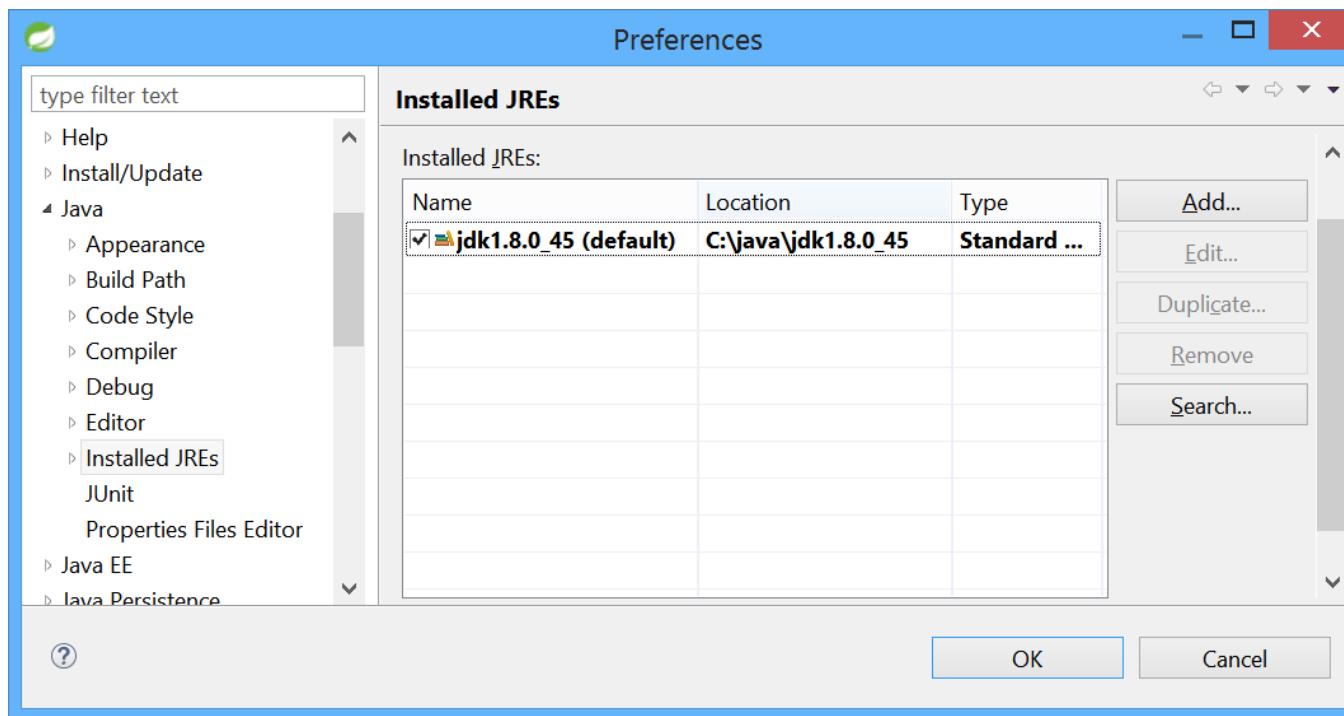
Columns...

Close

- ❑ **Java 8 propose beaucoup de nouveaux éléments de syntaxe**
 - L'environnement de développement doit supporter la syntaxe Java 8 et garder une compatibilité avec les versions précédentes.
- ❑ **Depuis Eclipse 3.1, Eclipse supporte totalement Java 5**
- ❑ **Depuis Eclipse 3.2, Eclipse supporte totalement Java 6**
- ❑ **Depuis Eclipse 3.6, Eclipse supporte totalement Java 7**
- ❑ **Depuis Eclipse 4.2, Eclipse supporte totalement Java 8**

➤ Sélectionner l'utilisation de la machine virtuelle 8.0

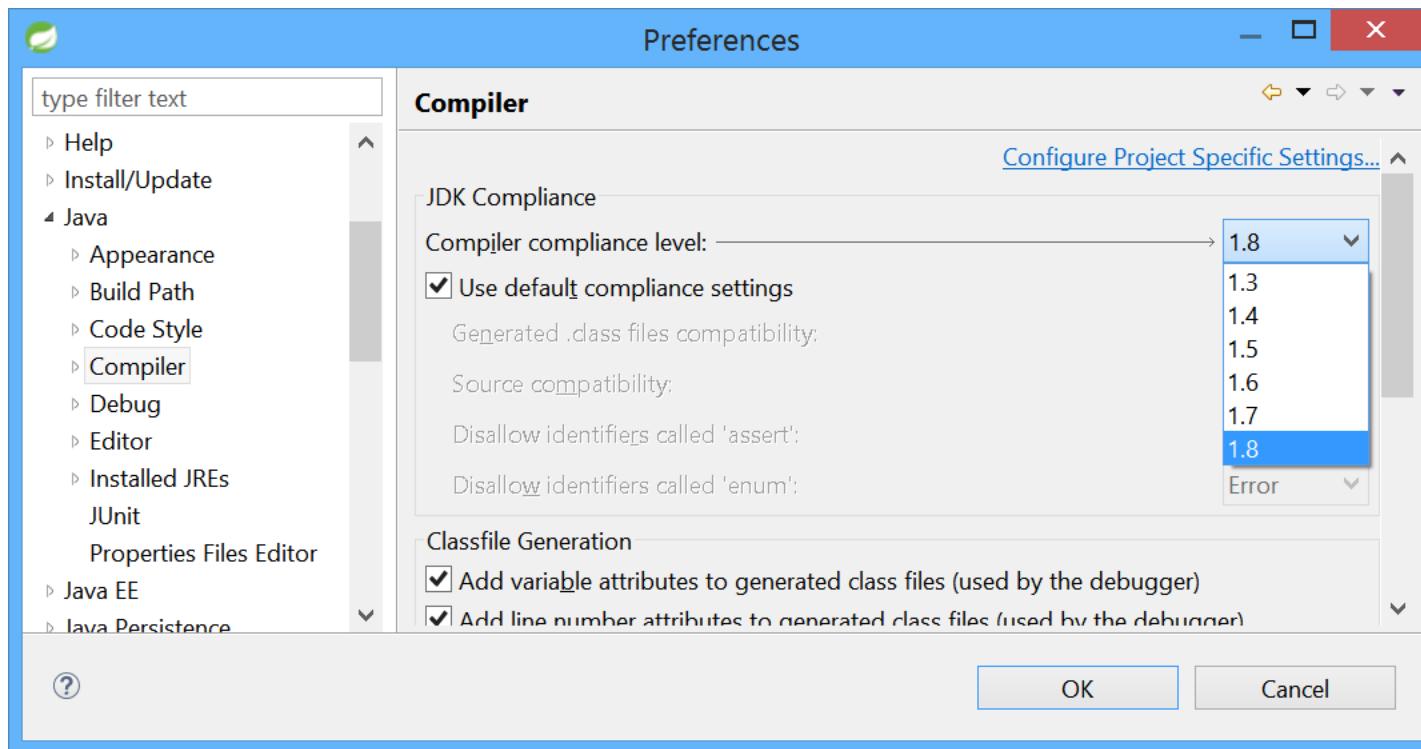
➤ Window → Preferences → Java → Installed JREs



Ci-dessus, la machine virtuelle utilisée est de niveau Java 8.
Mais il reste possible d'utiliser une syntaxe antérieure.

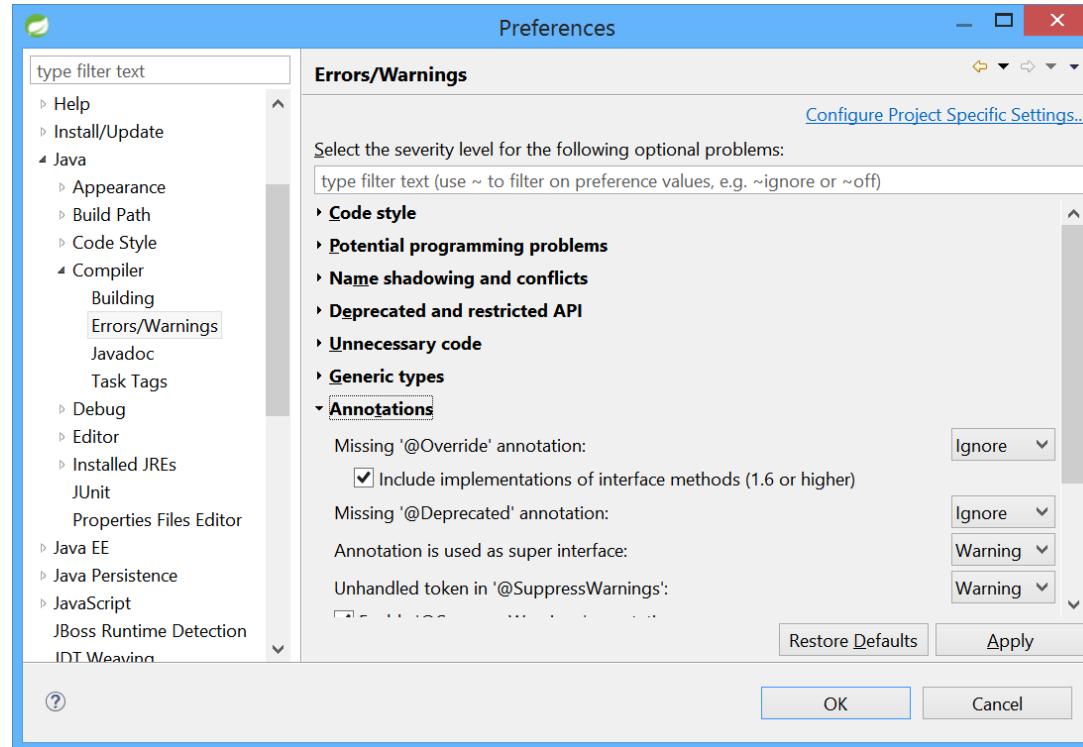
➤ Sélectionner un niveau 8.0 pour le compilateur

➤ Window → Preferences → Java → Compiler



➤ Il est possible de paramétrer un niveau de compilation Java plus précis

➤ Window → Preferences → Java → Compiler → Errors/Warnings



□ Réaliser les travaux pratiques suivants:

- TP 1 bis

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

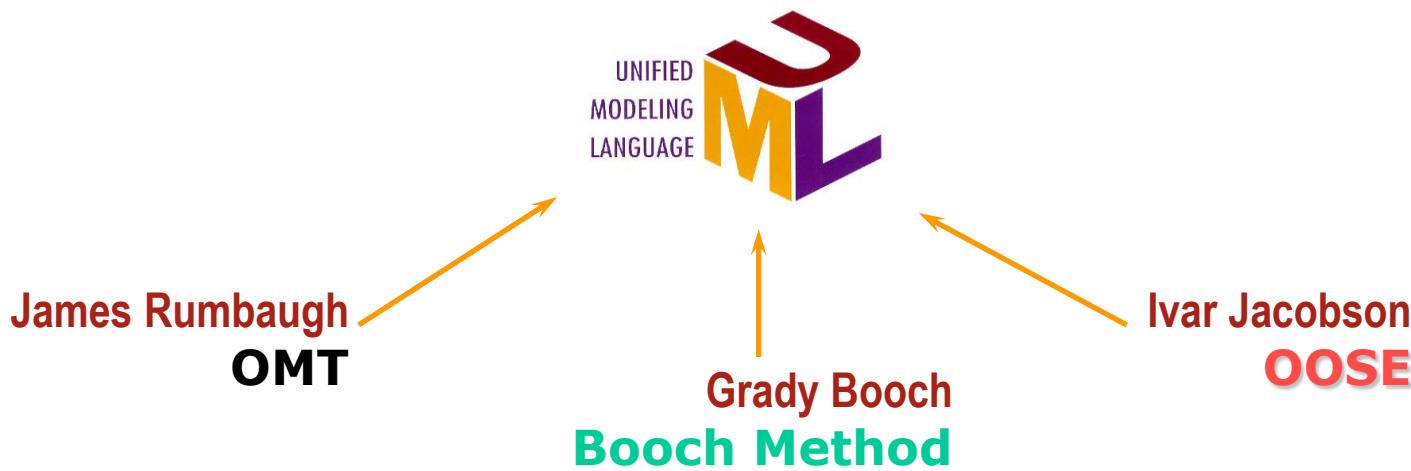
□ Années 90, plus de 30 méthodes recensées

- Booch, Classe-Relation, Fusion, HOOD, OMT, OOA/OOD, OOM, OOSE...



- Un foisonnement de concepts et de notations
- Une multitude d'acteurs en concurrence
- Une joyeuse pagaille qui ne fait plus progresser l'objet

- En 1995, trois acteurs majeurs du monde objet décident de lancer un projet ambitieux d'unification de leurs trois méthodes
- En 1997 ce projet donne naissance à UML

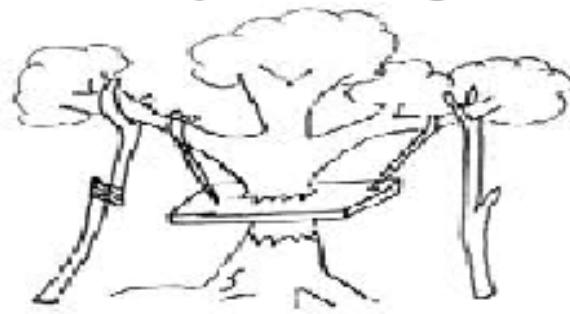




ce que demande l'utilisateur



ce que l'analyste a compris



après la mise au point



ce qui est écrit dans le cahier des charges



ce que le programmeur a réalisé



Ce qu'il fallait ...

- Moyen de clarification des idées, relations, décisions, besoins
- Support à la communication
- Les éléments modélisés sont des objets du monde réel, des éléments logiciels, etc.
- La modélisation s'attache à représenter ces éléments sous l'angle nécessaire à leur compréhension dans le contexte du système à développer
- Un modèle est composé de plusieurs diagrammes

- **Un diagramme est une représentation graphique, sous un certain point de vue d'un ensemble d'éléments faisant partie du modèle, y compris les relations entre eux (qui sont elles mêmes des éléments)**
- **Un même élément peut être présent dans plusieurs diagrammes avec des niveaux différents de représentation**
- **Les types de diagrammes peuvent être distingués en fonction du point de vue particulier qu'ils cherchent à illustrer sur les éléments qu'il présentent**
- **L'utilisation d'un outil servant de référentiel aux éléments modélisés permet d'assurer la cohérence de ces éléments à travers leur représentation graphique**

- **Un modèle n'est pas nécessairement un document**

- Modèle et document sont, de fait, 2 concepts orthogonaux

- **De nombreux développeurs n'aiment pas modéliser**

- Mauvaises expériences issues des méthodes de développement séquentielles

- **Modéliser est une activité appréciée par les « visuels »**

- Comprendre les caractéristiques cognitives des personnes avec qui on travaille

- **Il faut penser avant d'agir**

- Réaliser un diagramme avant de coder

- **La connaissance du domaine est importante**

- **Chaque type de modèle a ses forces et ses faiblesses**

- **Il faut utiliser un large éventail de types de modèles**

- Ne pas limiter aux diagrammes UML (MPD, CRC, etc...)

- **Faire attention au tout visuel**

- Le texte est encore utile pour communiquer avec les « non visuels »

- **Modéliser est un art difficile !!**

□ UML est un langage de modélisation objet

- Plusieurs diagrammes standardisés(facettes complémentaires d'un système)

□ UML est un langage de communication

- utilisation d'un même formalisme par tous les intervenants
- permet de lever les ambiguïtés du langage naturel

□ UML est un langage simple de haut niveau

- facile à apprêhender car visuel
- indépendant de tout langage de programmation

❑ UML est un langage semi-formel

- basé sur un méta-modèle décrit en UML
- les concepts manipulés et leur sémantique sont clairement établis par le méta-modèle
- les différents diagrammes sont cohérents entre-eux
- UML est largement outillé

❑ UML est un langage ouvert

- possibilité d'ajouter des notes (texte)
- utilisation de stéréotypes permettant d'étendre la notation, améliorant ainsi la sémantique des éléments modélisés

❑ Génie Logiciel:

- Abstraction
- Encapsulation
- Modularité
- Hiérarchie
- Connexion

- Description simplifiée d'un système qui met en avant certains détails ou propriétés de celui-ci et en supprime d'autres

- Mise en valeur des caractéristiques essentielles d'un objet du point de vue de l'observateur

- Regroupement de code et de donnée**
- Masquage d'informations au monde extérieur**
- Garantit l'intégrité des données**
- Facilite l'évolution d'une application car stabilise l'utilisation des objets.**

□ Définition :

- « La modularité est la propriété d'un système qui a été décomposé en un ensemble de modules fortement cohérents et faiblement couplés »

□ Exemples :

- Découpage des programmes Pascal ou C en modules, package en Java
- Package PL/SQL ORACLE

- Permet de classer et d'ordonner les abstractions
- Il s'agit d'une relation de classification (taxinomie)
- Exemple :
 - Un Singe est un Mammifère

- Relations entre les objets
 - Collaboration des objets
 - Interaction entre objets
- Exemple : la facture référence un produit.

□ Une définition

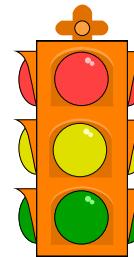
Objet = Identité + Etat + Comportement

- **Tout objet possède une identité qui lui est propre et qui le caractérise**
- **L'identité permet de distinguer tout objet de façon non ambiguë, indépendamment de l'état**
- **L'identifiant unique ou « Oid » (Objet identifier)**
 - Au niveau programme un OID est le plus souvent un pointeur (au besoin persistant)

- L'état regroupe les valeurs instantanées de tous les attributs d'un objet**
- L'état évolue au cours du temps**
- L'état d'un objet à un instant donné est le résultat de ses comportements antérieurs et de son état initial**

□ Exemple :

- Le feu tricolore est rouge
- Le feu tricolore est vert
- Le feu tricolore est orange



L'état du feu change au cours du temps, mais il s'agit bien du même objet

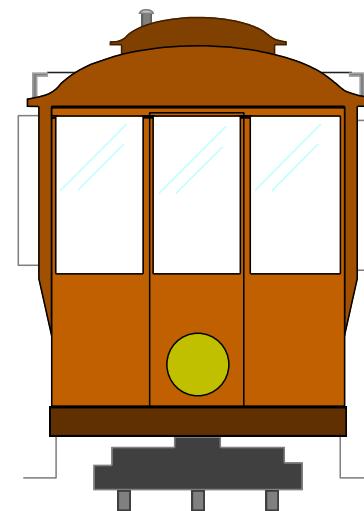
□ Le comportement

- décrit les actions et les réactions d'un objet,
- regroupe tout le savoir-faire d'un objet,
- se représente sous la forme d'opérations.

Le code de ces opérations est appelé méthode

□ Exemple : un tramway

- Ouvrir porte
- Accélérer
- Freiner



□ Quel différence existe-t-il entre

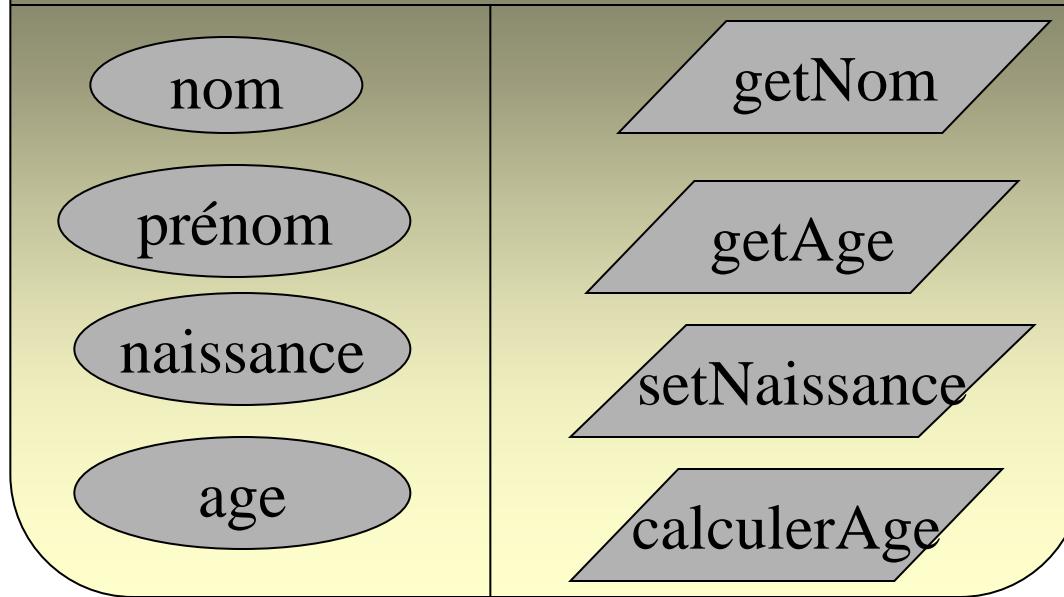
- dire de deux objets qu'ils sont égaux et
- dire de deux objets qu'ils sont identiques

□ Deux objets sont égaux si leur état donc les valeurs de leurs attributs respectifs à l'instant de la comparaison sont égaux

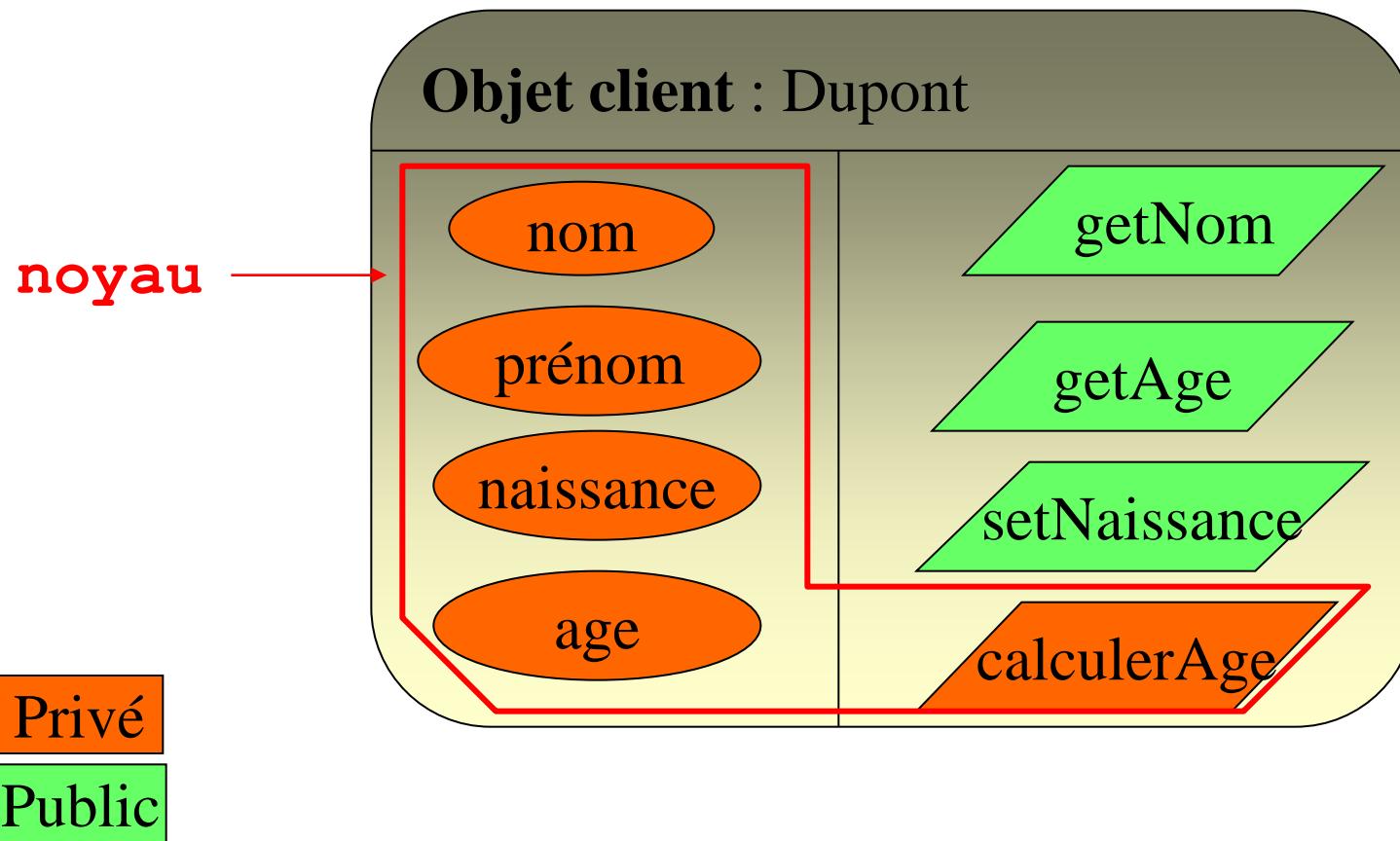
□ Deux objets sont identiques si leurs OID sont égaux

- **Un objet est un paquet logiciel qui regroupe des variables et les méthodes qui s'y rapportent**
- **Les variables d'un objet sont appelées variables d'instance (ou membres)**
- **Les méthodes d'un objet sont appelées méthodes d'instance (ou fonctions membres)**

Objet client : Dupont



- **L'implantation de l'objet est cachée par l'Encapsulation.**
- **Les variables et les méthodes sont réparties en deux catégories**
 - le noyau qui regroupe les variables et méthodes privées
 - l'interface qui regroupe les variables et méthodes publiques
- **Les objets interagissent les uns avec les autres au travers des interfaces**
 - Ce qui n'est pas dans l'interface n'est pas visible des autres objets



- L'interface permet donc de « cacher » le noyau de l'objet
- Cela permet de changer le noyau sans avoir d'influence sur l'interaction avec l'extérieur

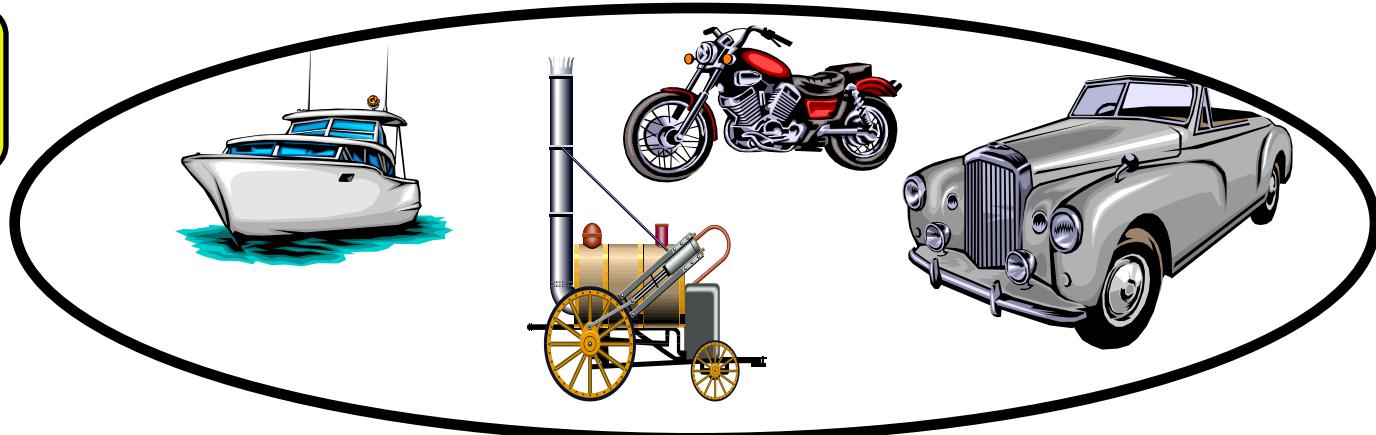
- Une classe est une description abstraite d'un ensemble d'objets
- La classification est particulièrement subjective
- L'abstraction est arbitraire et forcément liée à un “ point de vue ”

□ Décrit un ensemble d'objets ayant

- Une structure commune
- Un comportement identique

□ Tous les objets d'une classe partagent un objectif sémantique commun

Classe
Véhicule



□ Une définition

Classe = attributs + opérations + instantiation

- Propriétés décrivant les objets d 'une classe
- Unique dans une classe
- Un attribut a une valeur = partie de l 'état de l 'objet

- **Opérations applicables à un objet de la classe**
- **Chaque opération a un objet cible comme argument implicite**
- **Opération = spécification de la signature d'une méthode**
- **Deux niveaux de visibilité (au moins)**
 - publique
 - privée

□ Les opérations publiques = services

- constituent l'interface de la classe,
- sont accessibles de l'extérieur.

□ Les opérations privées

- sont déclenchées uniquement par le biais des méthodes publiques ou privées, elles ne sont accessibles directement depuis l'extérieur
- ne sont accessibles qu'au développeur de la classe.

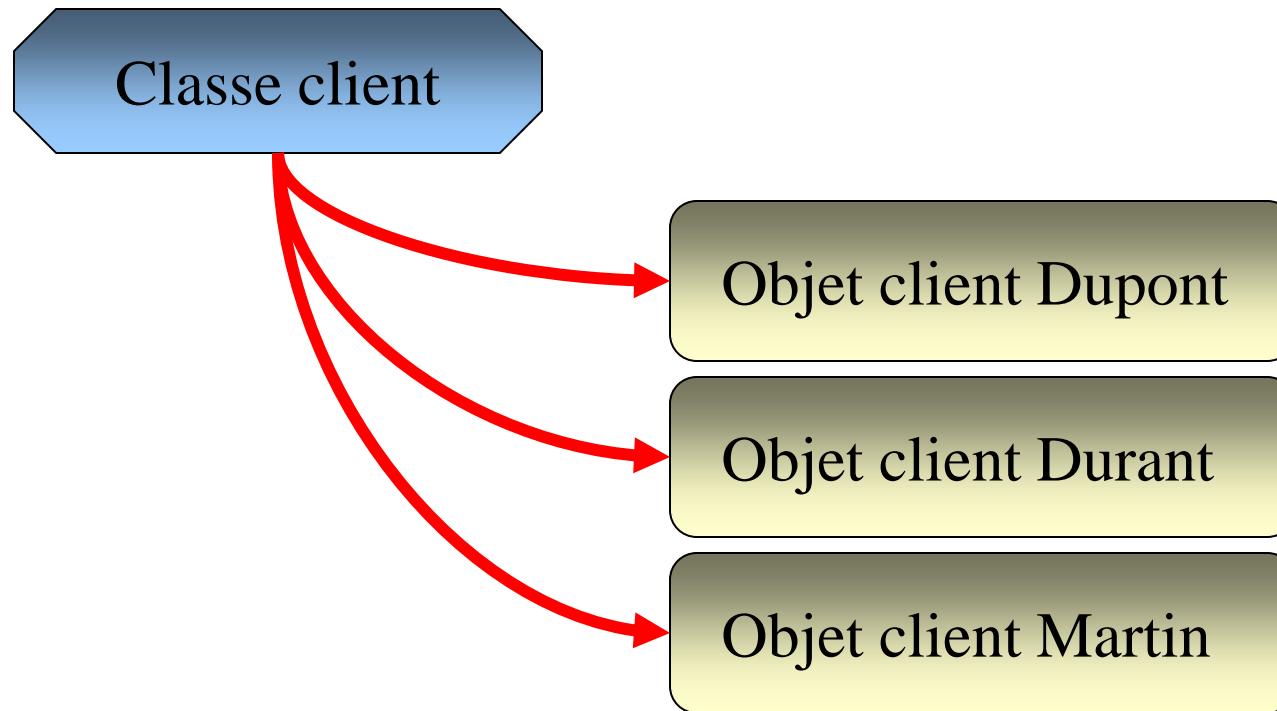
□ Séparée en deux parties

- La *spécification* d'une classe qui décrit le domaine de définition et les propriétés des instances de cette classe (type de donnée)
- La *réalisation* qui décrit comment la spécification est réalisée

- La classe est un moule servant à la fabrication d 'objets
- La classe fournit (implicitement ou non)
 - un constructeur,
 - un destructeur.

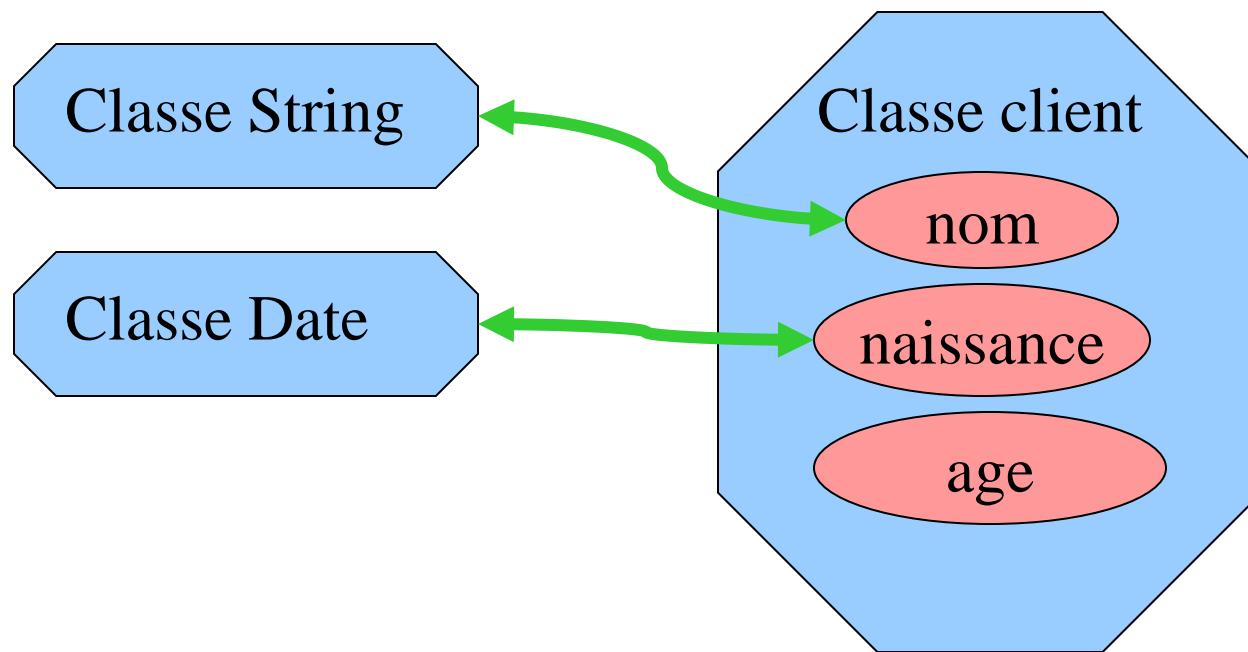
- L'instanciation est le mécanisme qui permet, à partir de la classe, la création d'un objet, une instance
- Une classe peut être instanciée plusieurs fois

- **Une instance d'une classe est un élément ou objet**
 - Exemple : Votre voiture et la mienne font parties de la classe Voiture
- **Chaque instance partage les noms des attributs mais conserve son propre état**
 - C'est à dire, chaque instance a ses propres valeurs pour chaque attribut
- **Les instances partagent l'implémentation des méthodes de la classe avec les autres instances de la classe**



- **Les objets échangent des messages pour communiquer**
- **Un message déclenche une activité dans l'objet destinataire**
- **Les messages permettent la collaboration d'un groupe d'objet**
- **Un envoi de message est l'équivalent d'un appel de procédure dans de nombreux LOO**
- **Un message est constitué :**
 - d'un destinataire,
 - d'un nom de méthode (interface du destinataire),
 - de paramètres éventuels

- La délégation est la possibilité de définir des variables d'instance comme des instances d'autres classes
- Ce mécanisme permet de la réutilisation de classes existantes
- Le principe d'encapsulation est conservé
 - un objet n'a pas accès à la partie privée de ses déléguées

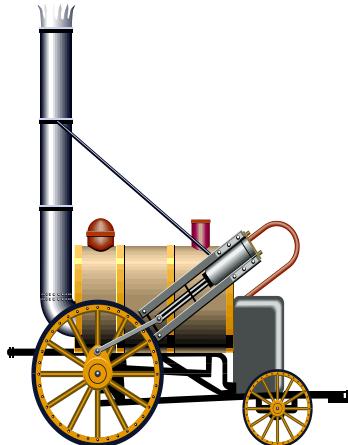


	CLASSE	INSTANCE
ATTRIBUT	<ul style="list-style-type: none"> Partagé par toutes les instances Décrit sur la classe 	<ul style="list-style-type: none"> Caractérise chaque instance Ex : N° commande, Nom, Age, Matricule,... Décrit sur la classe
METHODE	<ul style="list-style-type: none"> Ne s'applique pas à une instance en particulier Ex : Le constructeur Décrit sur la classe 	<ul style="list-style-type: none"> Opération assuré par chaque instance Ex : DonnerHeure(), Accelerer(),... Décrit sur la classe

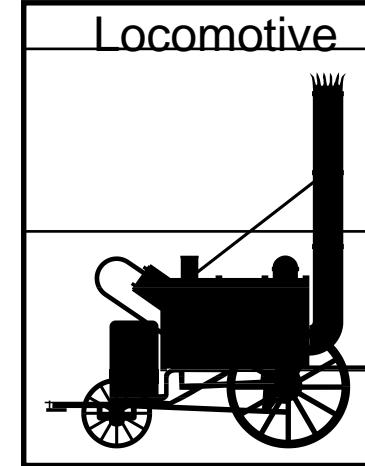
Synthèse

- Un objet est un composant du monde réel ou non
- Une classe décrit un groupe d'objets ayant un comportement (méthodes) et des propriétés (attributs) similaires
- Une instance est fabriquée par une classe

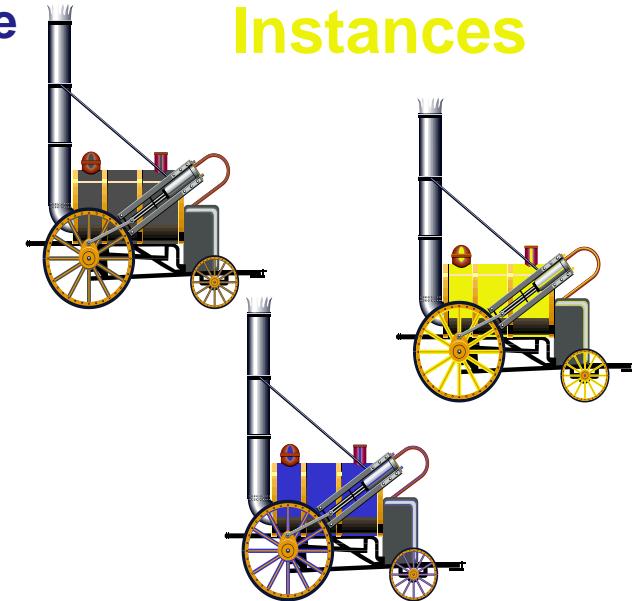
Objet



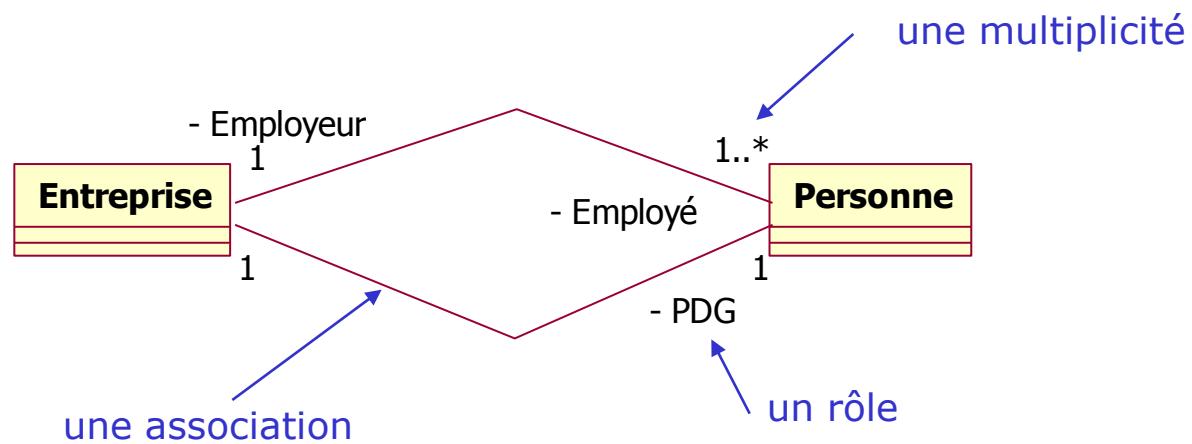
Classe



Instances



➤ Une association représente une connexion sémantique entre classes.



- L'imbrication est la possibilité de définir des variables d'instance comme des instances d'autres classes
- Ce mécanisme permet de la réutilisation de classes existantes
- Le principe d'encapsulation est conservé
 - un objet n'a pas accès au noyau de ces différentes parties

- Désigne la relation de hiérarchisation
- Ce terme est aussi utilisé pour désigner la démarche ascendante d'abstraction qui mène à la création d'éléments plus généraux dans l'arbre de classification
- Il serait préférable d'utiliser le terme de
 - hiérarchisation
 - classification
 - taxinomie.

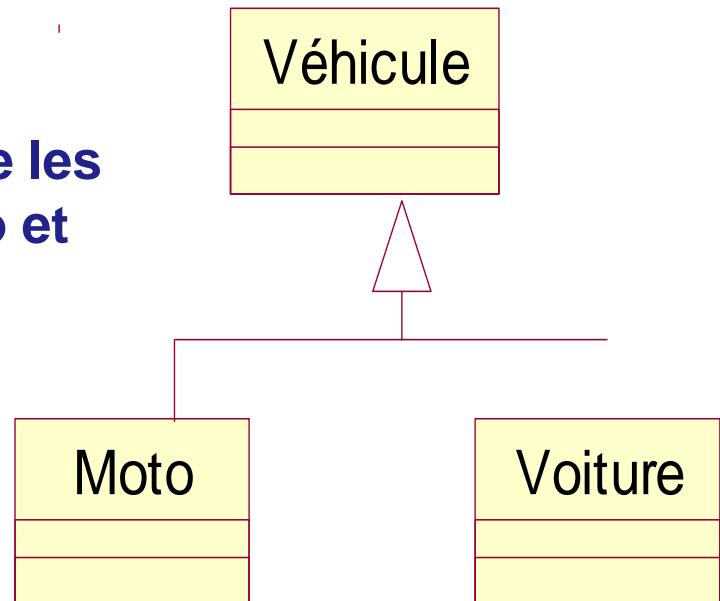
□ La généralisation

- fait correspondre à une sous classe une classe plus générale appelée super-classe
- **factorise** les éléments communs

□ La relation de généralisation est une relation « est un » (« is a »)

- Le concept de véhicule est plus général que celui de voiture ou moto

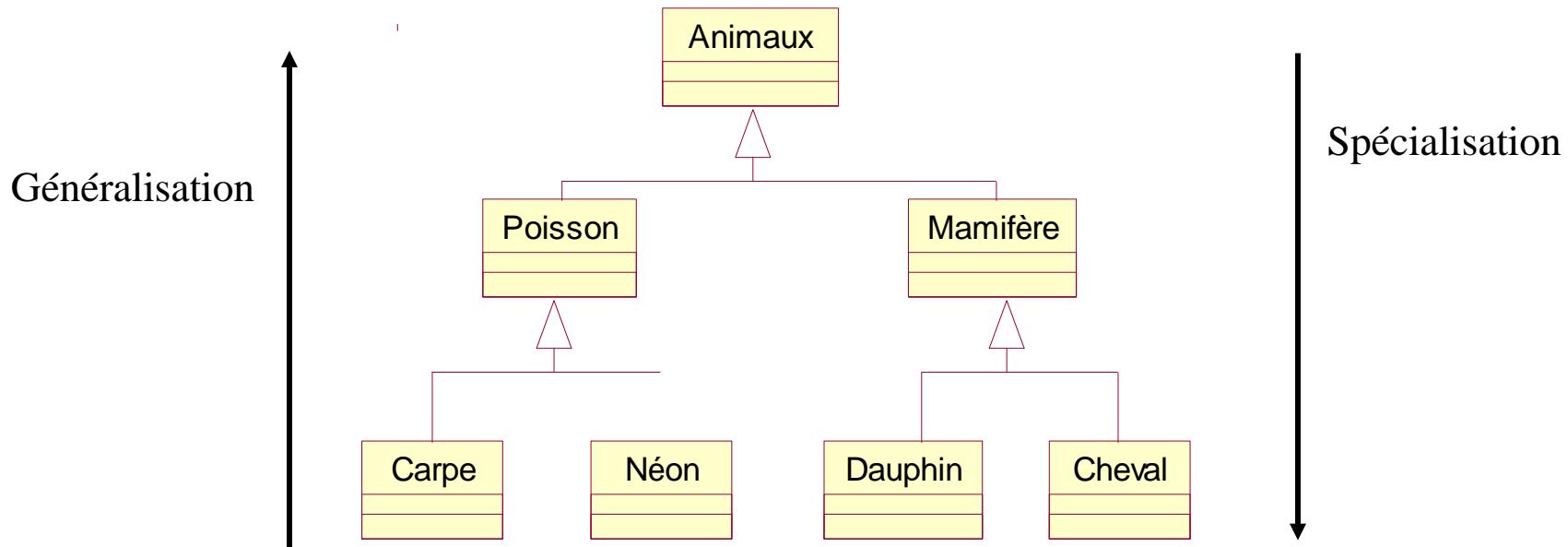
- La classe véhicule factorise les points communs entre auto et moto



- Processus inverse de la généralisation
- Fait correspondre à une classe toutes les sous-classes dont elle est la super-classe
- Enrichit et particularise

- Schématiquement on considère que :
 - L 'activité de conception favorise la généralisation
 - L 'activité de développement favorise la spécialisation
- Ne pas abuser des niveaux de spécialisation : environ 4 au plus
- Schéma : aller/retour (point de vue)

□ Exemple



- Le cheval est bien une spécialisation de la classe animal
- La carpe et le néon sont des poissons

- Est dite multiple lorsque qu'une sous classe a plusieurs ancêtres
- La sous-classe est une « fusion » de plusieurs super-classes
- Multiple doit être utilisée avec prudence et dans des cas indiscutables

- Mécanisme le plus courant de réalisation de la relation de généralisation/spécialisation
- Une classe peut être définie par héritage d'une autre classe
- Une sous-classe hérite de toutes les caractéristiques publiques de sa superclasse
- Une sous-classe peut ainsi modifier et/ou étendre ces caractéristiques
 - enrichissement et spécialisation

- L'orienté objet ne se résume pas à l'utilisation massive de l'héritage. Il n'est qu'un des moyens à disposition.
- L'important est de bâtir une arborescence de classe cohérente et sémantiquement correcte.

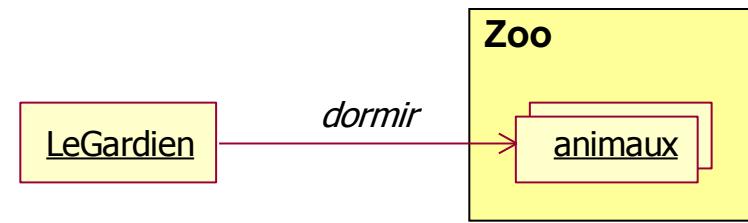
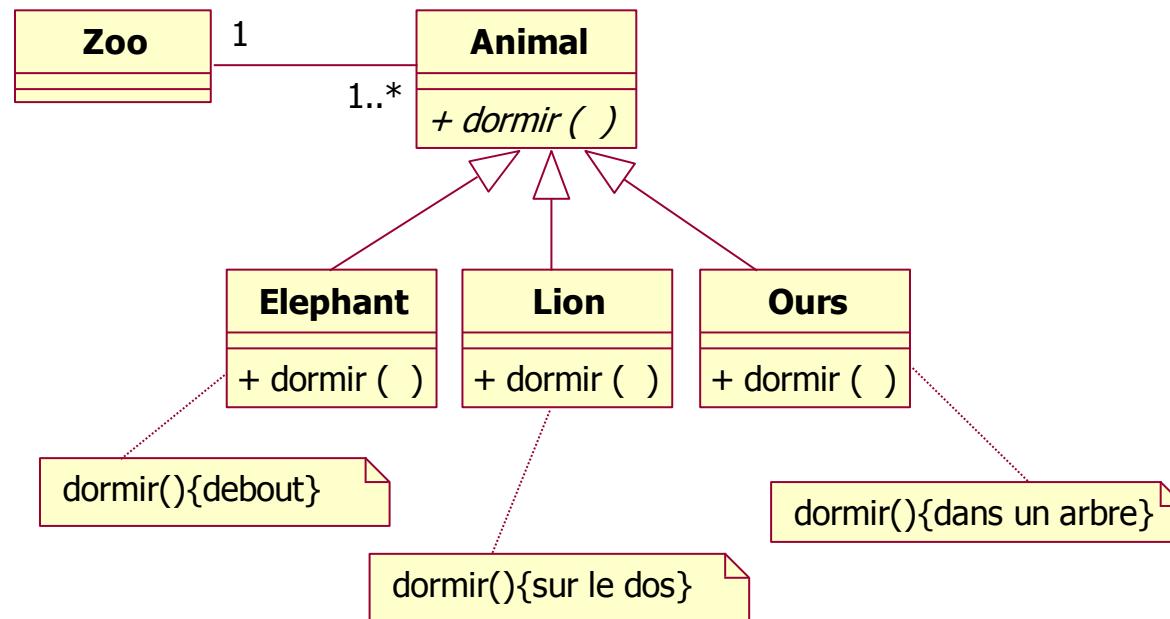
→ Enrichissement :

- Une sous-classe peut ajouter des définitions de variables et de méthodes à celles déjà existantes
- Il est possible d'enrichir et le noyau et l'interface

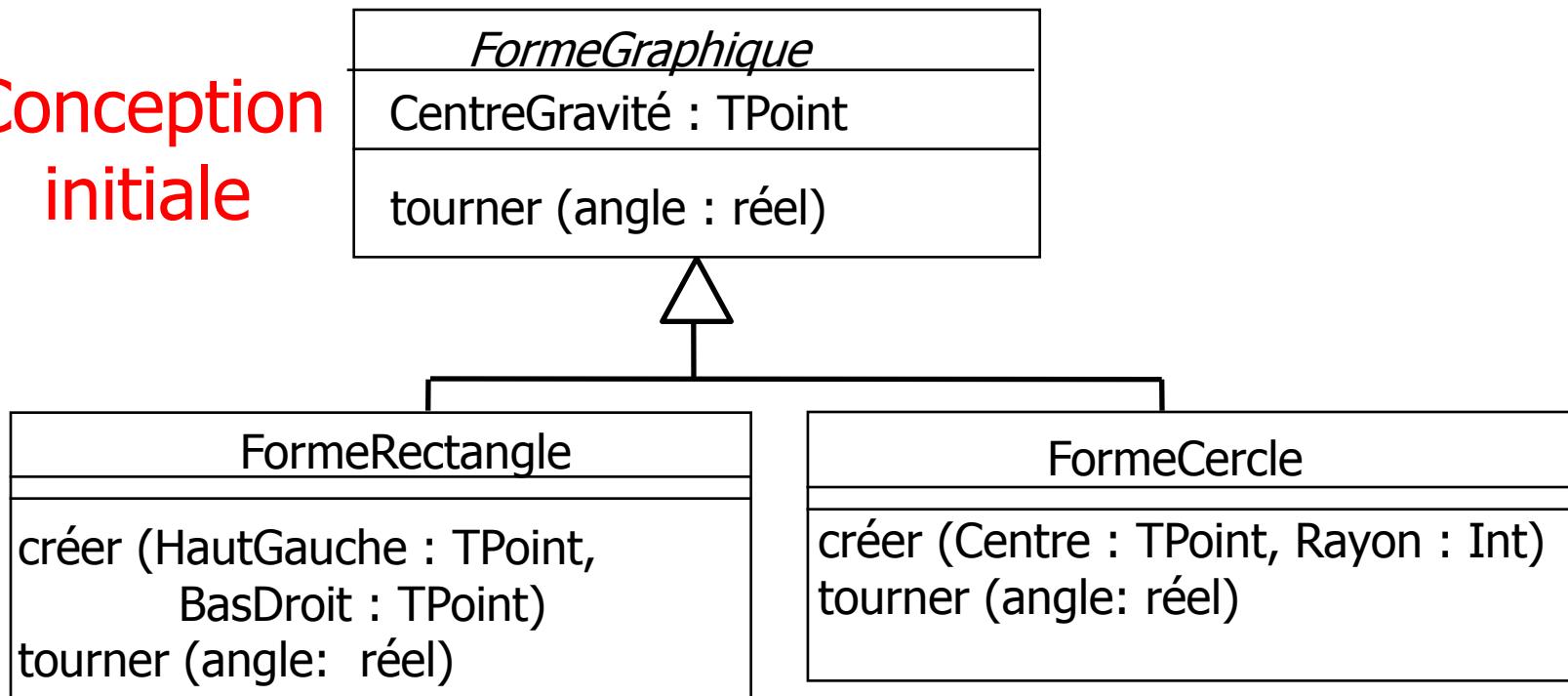
→ Spécialisation

- La spécialisation consiste en une redéfinition de méthodes de la super-classe
- La spécialisation ne modifie pas l'interface de la classe mais permet de modifier l'implantation de certaines méthodes

- Représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes
- Possibilité d'utiliser une classe parent pour adresser une classe enfant
- Augmente la générnicité du code
- Permet l'extensibilité des fonctionnalités
- Représenté par les classes virtuelles ou abstraites
 - classe définissant un comportement type
- Représenté par des redéfinitions de méthodes



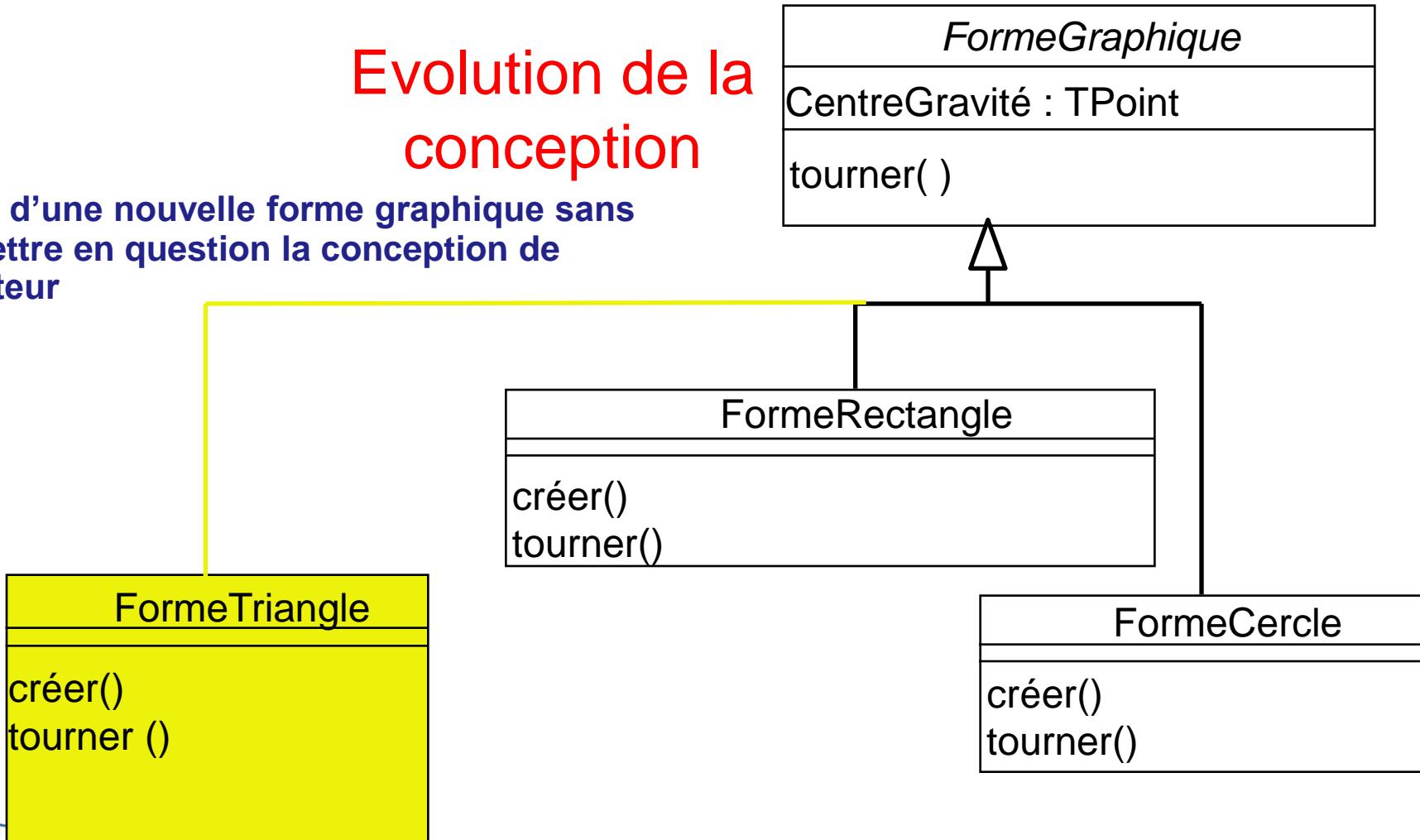
Conception initiale



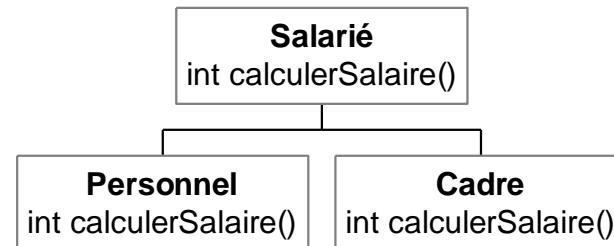
- **FormeGraphique** : Classe générale des formes disponibles dans l'éditeur
- La classe **FormeGraphique** est dite abstraite car elle ne sera jamais directement instanciée
- **FormeRectangle** et **FormeCercle** sont les formes réellement manipulées par l'utilisateur
- **Ifg = Liste de FormeGraphique; Ifg[i].tourner(x);**

Evolution de la conception

Ajout d'une nouvelle forme graphique sans remettre en question la conception de l'éditeur



□ Un arbre d'héritage



□ Code sans polymorphisme

```
for each e in E do
    case type(e)
        Salarié : calculateSalary1()
        Personnel : calculateSalary2()
        Cadre : calculateSalary3()
    end case
end for
```

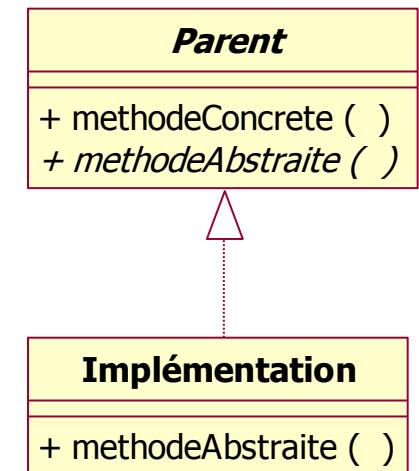
□ Code avec polymorphisme

```
for each e in E do e.calculateSalary()
```

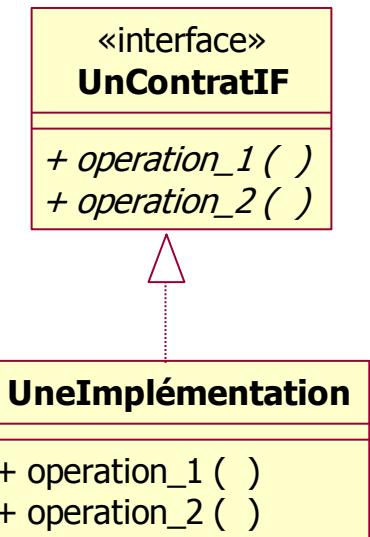
- Une classe abstraite est une classe spécialement conçue pour l'héritage
- Une classe abstraite ne peut pas être instanciée
- Elle permet la définition d'un comportement générique qui sera implémenté par les sous-classes

- **Les classes abstraites sont des classes pour lesquelles certaines opérations sont définies sans être implémentées.**
De telles méthodes sont dites abstraites.

- **L'implémentation se fait dans les classes filles.**



- Un interface est une liste de définitions de méthodes abstraite (sauf en Java 8 que nous verrons plus tard...)
- Correspondrait à une classe sans attributs dont toutes les méthodes sont abstraites.
- Permet de définir des contrats à respecter pour certaines classes.



```
public abstract class MyBaseClass {  
    public abstract String getResult();  
}
```

Une classe abstraite

```
public interface MyInterface {  
    public void myService(int pInteger);  
}
```

Une interface

```
public class MyExample implements MyInterface {  
    public void myService(int pInteger) {  
        // implémentation  
    }  
}
```

Une classe qui
implémente
une interface

```
public class MyExample extends MyBaseClass {  
    public String getResult() {  
        // implémentation  
    }  
}
```

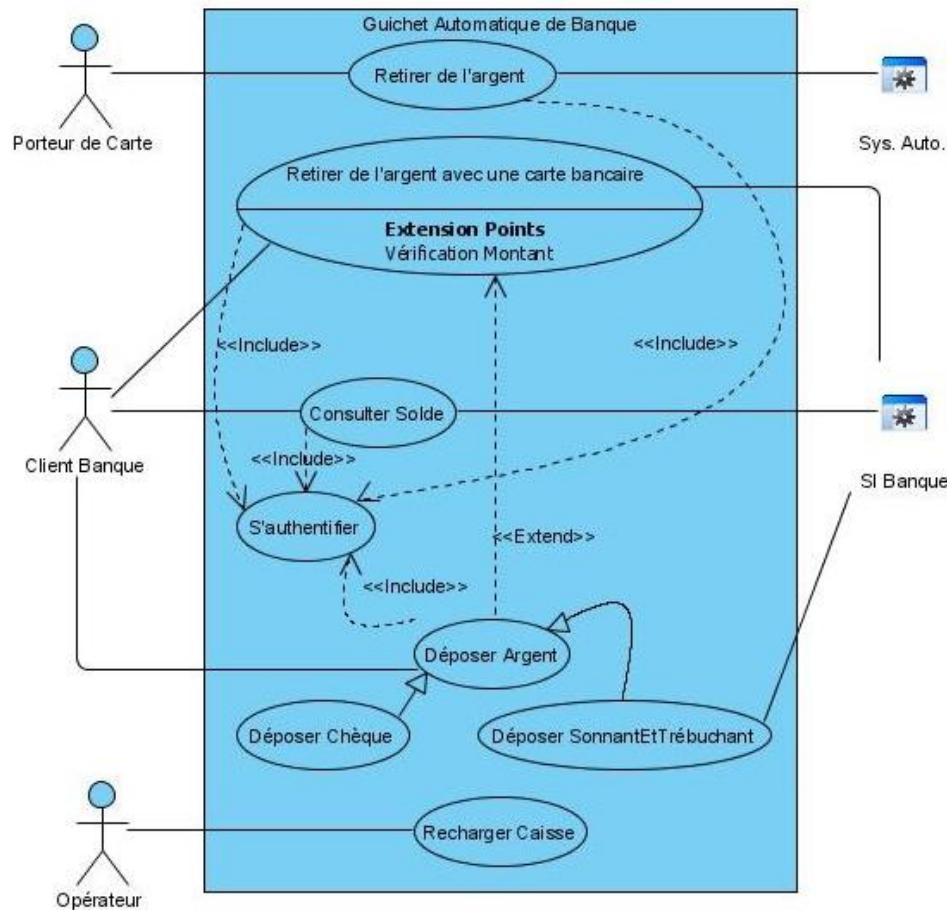
Une classe qui
hérite d'une
classe de base

```
MyBaseClass anObject = new MyExample("toto");
```

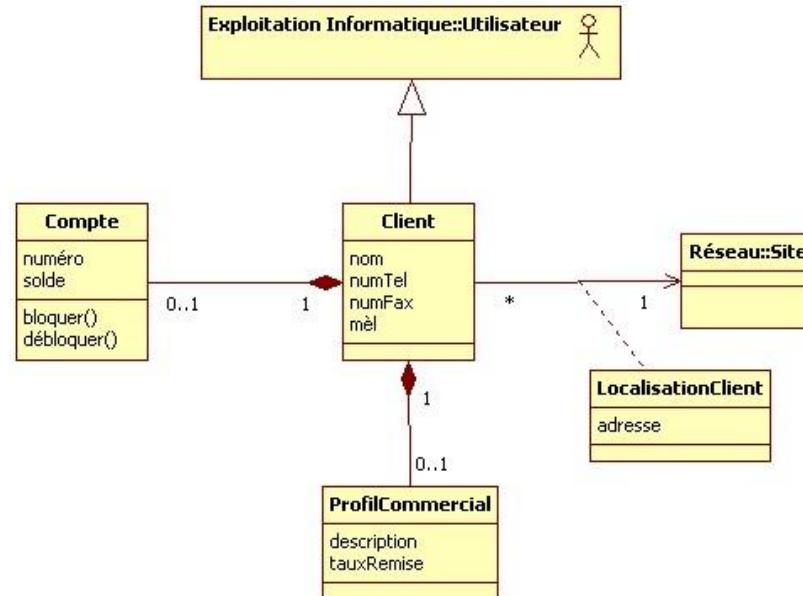
Instantiation

□ TP UML N°1

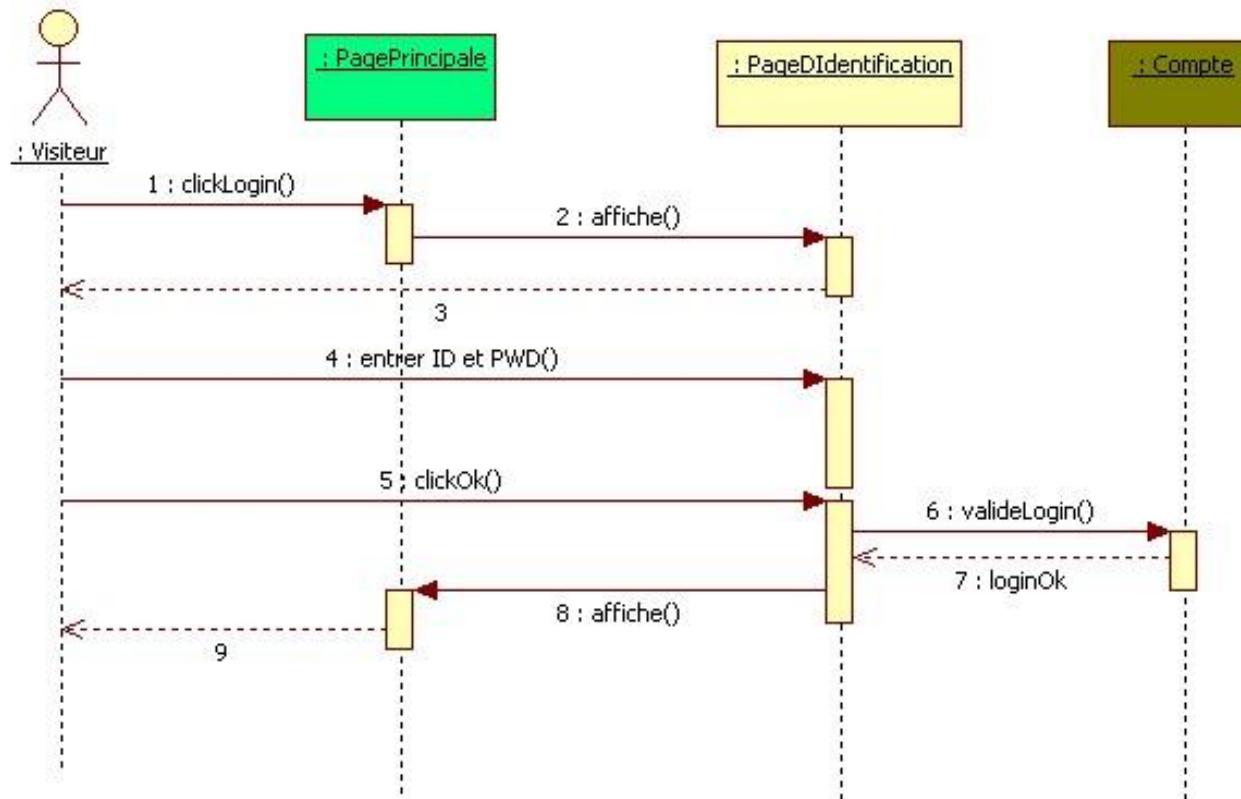
- Le DCU exprime, conjointement à une description textuelle, les exigences (besoins) d'un système



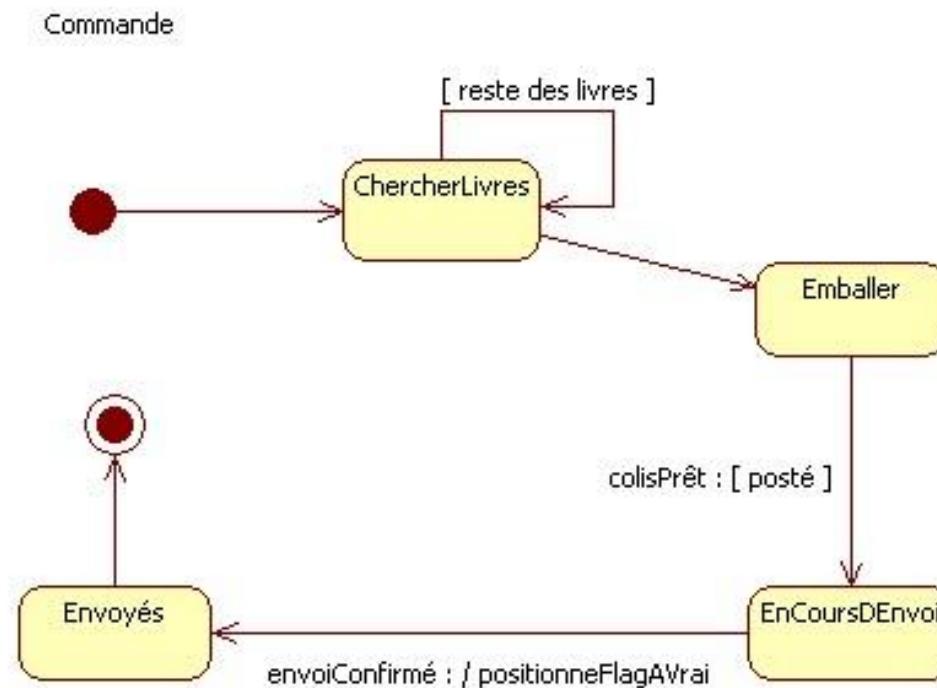
- Le DC présente les entités et les relations (statiques) entre elles
- Il permet de représenter l'ensemble des informations formalisées ayant fait fait l'objet d'une définition sur le fond et sur la forme
 - Avec les utilisateurs quant aux entités du domaine
 - Avec les développeurs quant aux entités techniques



□ Le DS met en évidence le type et l'ordre des messages échangés entre composants



- Le DE met en évidence les transitions d'états internes à un élément (classe, ensemble de composants, système tout entier) auquel il est associé



□ TP UML N°2 et 3

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- Littéraux**
- Types primitifs**
- Opérateurs**

- Représentent des valeurs « constantes » ou « littérales »
- Ont tous un type intrinsèque
 - Nombres
 - Caractères
 - Chaînes de caractères
 - Booléens
- On parle d'un langage fortement typé

□ Signés (préfixe -, optionnellement +)

□ Entiers : types **byte**, **short**, **int**, **long**

- | | | |
|---------------|------------|-----------|
| ➤ 14 (int) | 0x6a (hex) | 067 (oct) |
| ➤ 164L (long) | | |

□ Flottants : types **float** et **double**

- | | |
|-------------------|----------------------------|
| ➤ 3.14F (float) | 3.14D ou 3.14 (double) |
| ➤ 2.1E-3F (float) | 2.1E-3D ou 2.1E-3 (double) |



Les lettres peuvent être indifféremment en majuscule ou en minuscule.

□ Type char

□ Entre apostrophes

➤ 'A'

□ Précédés d'une barre inverse (backslash) si nécessaire (**« échappement » ou « escaping »**)

- '\'' (apostrophe) '\'\\\' (barre inverse) '\t' (tabulation 9)
- '\n' (ligne 10) '\f' (page 12) '\r' (retour 13)
- '\\" (guillemet) '\b' (arrière 8)

□ Avec un code Unicode en hexadécimal à 4 chiffres

➤ '\u00a9' (©) '\u0153' (œ) '\u20ac' (€)

❑ Entre guillemets

- "Bonjour«

❑ Utiliser la barre inverse si nécessaire

- "Une chaîne avec \" (un guillemet)"
- "Une chaîne avec \\ (une barre inverse)"
- "29,99 \u20ac" (29,99 €)
- "Un n\u0153ud" (un nœud)
- ...

❑ Stockés sous la forme d'objet de type `java.lang.String`

❑ Type boolean

❑ Valeurs possibles:

- true
- false

❑ ATTENTION : pas de majuscule !

□ Dans les initialisations de variables

➤ [modificateurs] type nom [= valeurInitiale];

- int i = 1;
- char a = 'P';
- boolean trouve = false;

□ Dans les expressions (voir plus loin les opérateurs)

- k = 10 * i + 1;
- a = code + 'A' ;
- i = i + 1;

□ Dans les appels

- System.out.println(trouve);
- effectuerTraitement(a, 100);

□ Taille standard sur toutes les plates-formes

- Portabilité assurée

□ Utilisés

- Pour une meilleure efficacité
- Pour une certaine compatibilité avec le C

Type	Taille
byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits
char	16 bits
boolean	1 bit

❑ Proviennent du langage C

- Pour la compatibilité
- Pour la souplesse et les performances

❑ Assument des conversions implicites

- byte → short → int → long → float → double
- char → int
- Permettent d'éviter toute perte d'information

□ Les opérations sur les nombres

- Conversions implicites
- Valables sur les char (assimilés à leur code)
- Attention aux débordements (pas d'erreur dans ce cas)
 - + Addition: $7+8=15$, 'A' + 3 = 68
 - Soustraction: $7-8=-1$
 - * Multiplication: $7*8=56$
 - / Division: $7/8=0$, $7.0/8.0=0.875$
 - % Modulo (reste de la division): $7\%8=7$, $7.0\%8.0=7.0$
 - ++ Pré/Post Incrémentation : i++, ++index
 - Pré/Post Décrémentation : k--, --index

□ Concaténation

- Les chaînes (String) se concatènent avec l'opérateur +
- Conversions implicites vers String

```
String result = "indice = " + i;
```

□ Opérations de comparaison

- Valeur booléenne true ou false
- Valable pour tous types
 - ==** Égal (*)
 - !=** Différent (non égal)

□ Opérations de relation d'ordre

- Valeur booléenne true ou false
- Seulement pour les nombres et les char
 - <** Plus petit
 - <=** Plus petit ou égal
 - >** Plus grand
 - >=** Plus grand ou égal

□ Opérations sur les booléens et expressions booléennes

!	Négation (NOT) :	!fini
&	Conjonction (AND) :	fini & (trouve)
	Disjonction (OR) :	fini (i > 10)
^	Disjonction exclusive (XOR) :	a ^ b

□ Variantes qui n'évaluent que ce qui est nécessaire

&&	Conjonction (AND) :	(x<0) && (1/x > -1.0)
	Disjonction (OR) :	(x == 0) (1/x > 0.5)

□ Permettent de positionner la valeur d'une variable

- Prend la valeur de l'expression à droite et la stocke
 - = Affectation simple : `a=10; b=(a+1)*3;`

□ Affectation combinée avec un opérateur

- Forme « var <op>= expression »
- Raccourci pour « var = var <op> expression »
- Valable pour tout opérateur
- Exemples
 - `a += 10` (raccourci pour `a = a + 10`)
 - `b *= i` (raccourci pour `b = b * i`)
 - `bool |= (1/x > 5)` (raccourci pour `bool = bool | (1/x > 5)`)

□ Opérateur ternaire permettant des choix

- (condition) ? valeur pour true : valeur pour false
- Simplifie certaines expressions (ne pas en abuser)
- Exemples

```
double x = -2.9;  
char sign = (x < 0) ? '-' : '+';  
String prefix = ((x <= 9) ? "0" : "");  
double valeur = (x < 0) ? -x : x;  
System.out.println(sign + prefix + valeur);
```

- **[] (indice), . (membre), () (appel de méthode)**
- **Opérateurs unaires, () (cast), new**
- ***, /, %**
- **+, -**
- **Décalages de bits**
- **<, <=, >, >=**
- **==, !=**
- **&&**
- **||**
- **? :**
- **Affectations (=, +=, *=, ...)**



Recommandation

Simplifier l'écriture avec des parenthèses !

□ Réaliser les travaux pratiques suivants:

- TP 2.1
- TP 2.2

❑ Bases objet de java

➤ Package, classe, instance, attribut, message, méthode

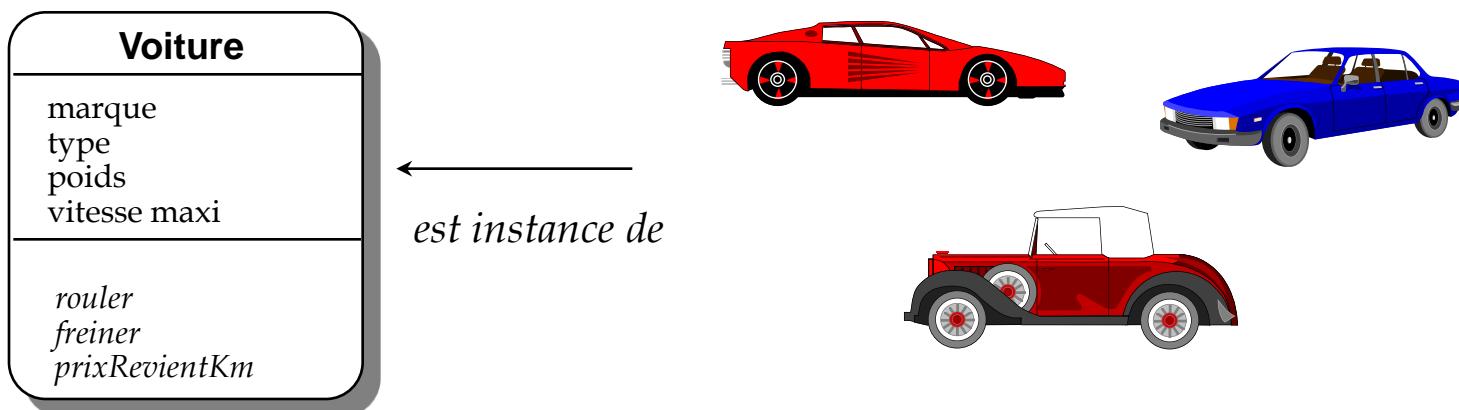
❑ Portée des variables locales

❑ Structures de contrôle

❑ Tableaux

- Définit la structure et le comportement d'objets de même type
 - structure : attributs
 - comportement : méthodes
- Permet de créer des objets d'un certain type
- Exemple : la classe des voitures

- Objet créé à partir d'une classe
- Possède ses propres valeurs d'attributs
- Partage les comportements (méthodes) des autres instances de la classe



- ❑ Librairie de composants
- ❑ Regroupement fonctionnel de classes
 - classes utilitaires
 - classes applicatives
- ❑ Niveau de granularité plus grand
- ❑ Même nom de classe possible si dans des packages différents
 - java.sql.Date
 - java.util.Date

□ Hiérarchies de packages

- organisées sous forme de répertoires
- espaces de nommage
 - packages séparés par des points
 - java.awt.event

□ Racines des hiérarchies

- basées sur le **CLASSPATH**

□ Package sans nom par défaut

❑ Principales caractéristiques d'une classe

- nom
- liste d'attributs
- liste de méthodes

❑ Déclaration avec le mot-clé class

```
[modificateurs] class ExempleDeClasse {  
    attributs et méthodes  
}
```

□ Pour indiquer à quel package une classe appartient

- `package nomDuPackage;`

- dans le fichier source
- à mettre avant les déclarations de classe

□ Classes connues par défaut

- classes du package + classes de `java.lang`

□ Souvent, besoin de classes d'autres packages

- préfixer par leur nom de package

```
java.util.Date maintenant = new java.util.Date();
```

- ou **importer la classe ou son package**

- plus besoin de préfixer le nom de la classe
- déclaration entre la déclaration de package et la déclaration de classe
- Import de package
 - `import nomDuPackage.*;`
- Import de classe
 - `import nomDuPackage.NomClasse;`

- [modificateurs] type nom [= valeurInitiale]
- Déclarés de préférence au début**
- Attributs (Variables d'instance)**
 - valeurs différentes pour chaque instance
 - `String marque;`
- Attributs statiques**
 - valeurs partagées par les instances et la classe
 - `static int nbInstances;`
- Constantes**
 - `final static int NB_MAXIMUM = 100;`

□ Les attributs peuvent être initialisés

- lors de leur déclaration
- `int a = 10;`

□ Attributs non initialisés

- sont à `null` pour les références d'objet
- contiennent une valeur par défaut pour les types primitifs
 - 0 pour les nombres
 - `false` pour les booléens
 - '`\u0000`' (caractère NUL) pour les caractères

□ Utilisation du caractère . (point)

□ Attribut (Variable d'instance)

- référencé uniquement par une instance
- `maCartePostale.texte`
- `this.image` (`this` sert à référencer l'objet courant)

□ Attribut statique

- référencé par une classe ou une instance
- `ExempleDeClasse.nbInstances`
- `obj.nbInstances`

□ type nom [= valeurInitiale]

□ **Destruction quand hors de portée**

□ **Un bloc est délimité par { . . . }**

□ **Variable locale**

- déclarée dans un bloc
- visible dans le même bloc

□ **Argument de méthode**

- visible dans la méthode

❑ Mot-clé : new

- doit être suivi par le nom de la classe et ()
- exemple : **new Voiture()**
 - crée un objet de type Voiture
 - initialise ses attributs

❑ Un objet peut être référencé par une variable

- exemple : **Voiture maVoitureBleue = new Voiture();**



La notion de constructeur sera abordée plus loin dans le cours.

- Démarrage de la construction d'une petite application qui sera enrichie au fil des TP suivants
- Réaliser les travaux pratiques suivants:
 - Suite et fin du TP 2 (2.3 à 2.6)

□ Message

- précise le comportement à exécuter d'un objet
- doit «appartenir» à l'interface de l'objet
- est envoyé à un objet par un autre objet (appel de méthode)

□ Méthode

- définit comment le comportement est exécuté
- peut accéder aux données de l'objet
- à **un** message envoyé correspond **une** méthode

❑ Composé

- d'un **sélecteur**
- d'**arguments** (optionnels)
 - les arguments sont des objets

❑ Envoyé à un receveur

❑ L'objet receveur retourne souvent un résultat

❑ Le message ignore le détail de l'exécution

- ❑ Exécutée en réponse à un message
- ❑ Script définissant la réponse
 - code
 - séquence d'expressions exécutables
- ❑ Possède le même nom que le message qui l'a déclenché

□ Une classe peut être exécutée

- si elle comporte une méthode *main*

```
public static void main(String[] args) {  
    // code à exécuter  
}
```

- ou si c'est une sous-classe d'**Applet**

```
[modificateurs] typeRetour nom(typeArg nomArg...) {  
    code...  
}
```

□ Méthodes d'instance

- appliquées aux instances
- **float calculerTaux(int n) {...}**

□ Méthodes statiques

- appliquées à la classe ou à ses instances
- **static String transformer() {...}**

□ Arguments

- passés par **valeur** pour les types primitifs
- Passés par **référence** pour les objets

□ Pas de retour attendu

- type de retour **void**
- l'expression **return**; peut être utilisée pour une sortie explicite avant la fin de la méthode

□ Retour d'un objet à la fin d'une méthode

- mot-clé **return** suivi de l'objet
- éventuellement **return null;**

□ Utilisation du caractère . (point)

□ Méthodes d'instance

- envoi à une instance (via sa variable)
- `obj.calculerTaux(5);`
- `this.envoyer();`

□ Méthodes statiques

- envoi à une classe ou une instance
- `ExempleDeClasse.transformer();`
- `obj.transformer();`

□ Réaliser les travaux pratiques suivants:

- TP 3.1
- TP 3.2

- Méthode *spéciale* sans préciser le type de retour (ni void)
- Initialisation automatique d'un objet
- Appelé lors de la création d'un objet

- **Même nom que la classe**
- **2 types de constructeurs :**
 - Par défaut : sans paramètre
 - Personnalisé : avec nombre quelconque d'arguments
- **Plusieurs constructeurs par classe**
 - par surcharge
 - signatures différentes
- **Pas de valeur de retour**

□ Constructeur sans argument

□ Défini implicitement dans chaque classe

- sans traitement spécifique autre que la création d'instance
- peut être redéfini pour ajouter des traitements spécifiques

□ Disparaît dès qu'un autre constructeur est défini dans la classe

- il faut, si nécessaire, le redéfinir explicitement

□ Assistant de création avec Eclipse

- le constructeur par défaut peut être défini explicitement (paramétrable lors de la création de la classe)

□ Deux définitions de constructeurs dans une classe

```
class CartePostale {  
    Photo image;  
    String texte;  
    Adresse adresse;  
  
    /** Définition du constructeur par défaut */  
    CartePostale () {  
        texte = "Nous passons de bonnes vacances";  
    }  
  
    /** Surcharge du constructeur par défaut */  
    CartePostale(String texte) {  
        this.texte = texte;  
    }  
}
```

□ Pour créer un objet

```
CartePostale carte = new CartePostale();
CartePostale carteRusse = new CartePostale("Bons baisers de Russie");
```

□ Par un autre constructeur (mot-clé this)

```
CartePostale(String texte) {
    this(); // appel du constructeur par défaut
    this.texte = texte;
}
```

□ Automatique en Java

- une fois que l'objet n'est plus référencé
- pas de destructeur (C++)

□ Effectuée par le *garbage collector*

- libère la mémoire
- processus de faible priorité

□ Envoi du message `finalize()` lorsque l'objet est sur le point d'être détruit



Attention ! L'appel à `finalize()` n'est pas garanti par la machine virtuelle, donc son usage est à éviter.

□ Réaliser les travaux pratiques suivants:

- TP 3.3

- **Les tableaux sont des objets, dynamiquement créés et auxquels il est possible d'affecter des variables de type `java.lang.Object`.**
- **Structure de taille fixe qui contient des composants de même type**
 - types primitifs ou objets
- **Référencés par des variables**
 - `String[] prenoms;`
 - `int[] notes = {12, 20, 3, 14};`
- **Création du tableau**
 - `prenoms = new String[3];`
- **Peuvent être multi-dimensionnels**
 - `int[][] c = new int[10][3];`

□ Accès aux éléments par leur indice (l'index est un entier positif)

- commence à l'indice 0
- opérateur d'accès : []
- longueur du tableau : length
 - permet de faire des boucles

```
String[] chaines = new String[3];
chaines[0] = "1ers mots";
System.out.println(chaines[0]);      // affiche: 1ers mots
System.out.println(chaines.length); // affiche: 3
```

□ Si accès hors des limites du tableau, erreur à l'exécution

ArrayIndexOutOfBoundsException

□ Réaliser les travaux pratiques suivants:

- TP 4.1

❑ Exécutions conditionnelles

- if-else
- switch

❑ Itérations

- for
- while
- do-while

□ La condition if-else

```
if (condition) {  
    instructions;  
}
```

```
if (condition) {  
    instructions;  
} else {  
    instructions;  
}
```

```
if (condition) {  
    instructions;  
} else if (condition2) {  
    instructions;  
} else if (condition3) {  
    instructions;  
} else {  
    instructions;  
}
```

□ Exemple

```
if (i == 0) { // retourne un boolean
    System.out.println("i est nul");
} else if (i > 0) {
    System.out.println("i est positif");
} else {
    System.out.println("i est négatif");
}
```

- La condition **switch**
- La valeur de test doit être une expression retournant un entier, un type enum (Java 5) ou une String (Java 7)
- Mot-clef **break** : sortie du niveau de traitement
- Mot-clef **default** : instruction effectuée par défaut, optionnel

```
switch (valeur) {  
    case (valeur1) :  
        instructions;  
        break;  
    case (valeur2) :  
        instructions;  
        break;  
    default :  
        instructions;  
}
```

- **La boucle for**
- **Initialisation, condition de continuation et incrémentation sont facultatifs**
- **La boucle n'est pas exécutée si la condition est fausse au départ mais l'initialisation est toujours effectuée**
- **L'incrémentation est une expression qui est exécutée après chaque itération**

```
for (initialisation; condition; incrémentation) {  
    instructions;  
}
```

- **syntaxe depuis Java 5 pour les collections**

```
for (type variable : collection) {  
    instructions;  
}
```

❑ Exemple

➤ Boucle classique

```
for (int i = 0; i < 10; i++) {  
    // calcul du carré de i  
    int carre = i * i;  
  
    // affichage du résultat  
    System.out.println("le carré de " + i + " est " + carre);  
}
```

➤ Boucle sur un tableau

```
String noms = {"Frédéric", "Valérie", "Bernard", "Arlette"};  
for (String nom : noms) {  
    // affichage de chaque nom  
    System.out.println("un nom: " + nom);  
}
```

□ Les boucle while

□ Différence entre while et do-while

- Les instructions dans le do-while sont exécutées au moins une fois

```
while (condition) {  
    instructions;  
}
```

```
do {  
    instructions;  
} while (condition);
```

□ Exemple

```
int i = 10;
while (i > 0) {
    // calcul du carré de i
    int carre = i * i;

    // affichage du résultat
    System.out.println("le carré de " + i + " est " + carre);

    // décrémentation de 1
    i--;
}
```

- **Il est possible d'agir sur le déroulement d'une boucle :**
 - le mot-clef **break** permet de sortir de la boucle.
 - Le mot-clef **continue** permet d'aller directement à l'évaluation suivante de la condition.

- **Attention : break n'est pas utilisable dans une condition**

❑ Importance des minuscules/majuscules

❑ Noms de packages

- En minuscules

❑ Noms de classes

- Commencent par une majuscule
- Chaque mot commence par une majuscule (CamelCase)

❑ Noms de variables et de méthodes

- Commencent par une minuscule
- Chaque mot commence par une majuscule

❑ Noms de constantes statiques

- Tout en majuscules, mots séparés par le caractère souligné _

- Respecter les normes de commentaires pour la génération avec javadoc
- Utiliser systématiquement les {} pour les blocs ne comportant qu'une seule ligne
- Définir une classe unique par fichier source
- Ne pas abréger les noms de variables ou de méthodes

Réaliser les travaux pratiques suivants:

- Suite et fin du TP 4

□ **Un type énuméré est un type dont la valeur fait partie d'un jeu de constantes définies**

```
public class TypeCarte {  
    public static final int PIQUE = 1;  
    public static final int COEUR = 2;  
    public static final int CARREAU = 3;  
    public static final int TREFLE = 4;  
    ...  
}
```

□ **Les inconvénients**

- Elle n'est pas « type-safe »
 - Pour une variable qui désigne un type de carte, il est possible de lui donner une valeur qui ne fait pas partie de l'énumération (n'importe quel entier)
- Elle ne fournit pas de correspondance entre la valeur de la constante et sa description

□ Opération sans contrôle de type

```
class Carte {  
    String nom;  
    int type;  
}
```

```
Carte uneCarte = new Carte(); // création d'une nouvelle carte  
  
uneCarte.type = TypeCarte.COEUR; // ok  
  
uneCarte.type = 123; // ko, 123 n'est pas un type de carte connu !
```

- Depuis Java 5 la notion de type énuméré « type-safe » a été ajoutée
- Mot clé enum

```
public enum TypeCarte {  
    PIQUE, COEUR, CARREAU, TREFLE  
}
```

- Avantages

- Elle est "type-safe"
- Il est possible d'ajouter des attributs et des méthodes à chaque valeur

```
public enum Vehicule {  
    VÉLO(2, "Bicycle"),  
    VOITURE(4, "Car");  
  
    private final int nombreDeRoues;  
    private final String libelleAnglais;  
  
    Vehicule(int nombreDeRoue, String libelleAnglais) {  
        this.nombreDeRoues = nombreDeRoue;  
        this.libelleAnglais = libelleAnglais;  
    }  
  
    public String getLibelleAnglais() {  
        return libelleAnglais;  
    }  
  
    public int getNombreDeRoues() {  
        return nombreDeRoues;  
    }  
}
```

- Permet de déclarer une méthode sans savoir le nombre d'arguments qu'elle va recevoir (depuis Java 5)
 - Concrètement : raccourci pour passer un tableau en paramètre
- Contrainte
 - Un seul varargs par méthode
 - Un paramètre varargs est toujours le dernier de la signature
- Déclaration avec ...
 - type... nomVariable

```
public void afficher(int valeur, String... args) {  
    System.out.println("valeur entière: " + valeur);  
    for (String arg : args) {  
        System.out.println("valeur string: " + arg);  
    }  
}
```

- **Les annotation permettent d'ajouter des méta-données au code source et de compiler en conséquence.**
- **Syntaxe**
 - @NomAnnotation (...)
- **Exemple d'utilisation**
 - Informer le compilateur d'ignorer un avertissement
 - @SuppressWarnings ("unused")
 - Déclarer que l'on redéfinit une méthode
 - @Override
 - Déclarer qu'une classe ne doit plus être utilisée
 - @Deprecated
- **Surtout utilisées en Java EE 6,7,8 : @Entity @Resource, @FunctionalInterface...**
- **Possible d'écrire ses propres annotations**

□ Problématique

- Un objet Avion peut transporter soit des objets de type Passager, soit des objets de type Marchandise mais pas les deux en même temps
- Passager et Marchandise n'ont rien en commun
 - Une classe parente : Object

□ Avion doit proposer des méthodes pour être chargé et déchargé

- Pour accepter autant les passagers que les marchandises, les méthodes doivent travailler avec la classe Object
 - Nécessite des conversions
 - Pas de contrôle visant à vérifier que l'on ne mélange pas les passagers et la marchandise dans un même avion
 - Risque d'erreur lors de l'exécution !

❑ Implémentation d'un cas d'exemple sans type générique

```
Passager passager1 = new Passager("Toto");
Passager passager2 = new Passager("Titi");
Marchandise marchandise = new Marchandise("paquet1");
Avion avion = new Avion();
avion.charger(0, passager1);
avion.charger(1, passager2);
avion.charger(2, marchandise); // autorisé !

// récupération de l'élément 2, nécessite une conversion explicite (cast)
// autorisé à la compilation, erreur à l'exécution !
Passager passager = (Passager) avion.decharger(2);
```

❑ Les erreurs de type ne sont détectées qu'à l'exécution du programme

- Donc trop tard...

- **Solution: typer la classe Avion !**
- **les types génériques ont été ajoutés dans ce but**
- **Possibilité de paramétriser un type (classe, interface) ou une méthode avec un ou plusieurs types dit génériques**
 - Une liste d'entiers, un dictionnaire de chaînes de caractères, un comparateur de date, ...
- **Syntaxe avec <...>**
 - type< identificateur[, identificateur2, ...]>
- **Permet au compilateur d'effectuer des contrôles de type**

❑ Implémentation d'un cas d'exemple avec type générique

```
Passager passager1 = new Passager("Toto");
Passager passager2 = new Passager("Titi");
Marchandise marchandise = new Marchandise("paquet1");
Avion<Passager> avion = new Avion<Passager>();
avion.charger(0, passager1);
avion.charger(1, passager2);

// la ligne suivante ne compile pas !
avion.charger(2, marchandise);

// récupération de l'élément 1, ne nécessite pas de conversion explicite
Passager passager = avion.decharger(1);
```

- ❑ Les erreurs de type sont détectées à la compilation
- ❑ Le code est plus lisible

❑ Cohabitation entre les types génériques et les types bruts

- Le compilateur Java compile le code mais émet des avertissements (« raw type »)

❑ Comment supprimer les avertissement ?

- Modifier le code pour utiliser les types génériques
- Suppression des avertissement avec l'annotation
 @SuppressWarnings ("unchecked")
- Désactiver le contrôle de type au niveau du compilateur
 - –Xlint:unchecked for details.
 - Très fortement déconseillé !

□ Définition d'un type générique

```
public class Avion<E> {  
    public void charger(int index, E charge) {  
        ...  
    }  
  
    public E decharger(int index) {  
        ...  
    }  
}
```

- Le type générique E peut s'utiliser comme un type normal au sein de la classe
- Beaucoup d'autre possibilités d'utilisation
 - Cela devient généralement vite complexe...
 - <https://docs.oracle.com/javase/tutorial/java/generics/>

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10. Les entrées / sorties (IO)**
- 11. L'accès aux bases de données**
- 12. Le mapping objet-relationnel**

- L'encapsulation
- La composition
- L'héritage
- Le polymorphisme
- Les interfaces Java

□ Une classe fournit un ensemble de services

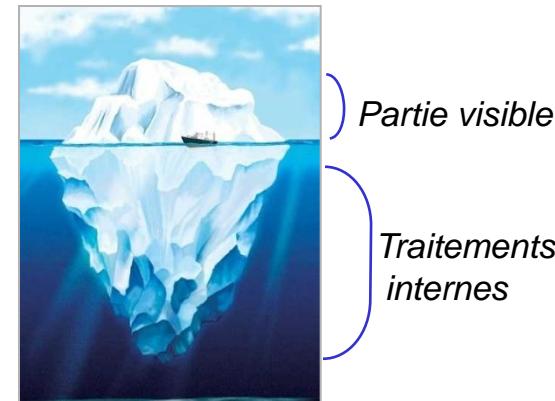
- Traitements
- Transport de données

□ Une classe doit être simple d'utilisation

- Interface d'utilisation claire et simple
- Complexité masquée à l'intérieur

□ But : masquer les traitements internes

- Dans une classe, possibilité de restreindre la visibilité des attributs et méthodes



□ Calcul de la recette d'un aéroport

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        // calcul recette des ventes billets  
        // calcul recette restaurants  
        // calcul taxes aéroport  
        // ...  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        float recette = r.calculerRecetteTotale();  
    }  
}
```

□ La méthode `calculerRecetteTotale()` est très longue. Comment la diviser en sous-méthodes ?

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //appel à toutes les autres méthodes  
    }  
  
    public long calculerVentesBilletsAvion() {...}  
    public long calculerTaxesAeroport() {...}  
    public long calculerRecetteRestaurants() {...}  
}  
  
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        long recette = r.calculerRecetteTotale();  
  
        long taxes = r.calculerTaxesAeroport();  
    }  
}
```

- **calculerRecetteTotale() fait appel aux autres méthodes de la classe RecetteAeroport**
 - Les autres méthodes sont à usage interne de la classe.
 - **Problème** : la classe Test a inutilement accès à ces autres méthodes...

□ Solution : les méthodes à usage interne sont privées

- Elles ne sont plus visibles de la classe Test

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //appel à toutes les autres méthodes  
    }  
  
    private long calculerVentesBilletsAvion()  
{ ... }  
    private long calculerTaxesAeroport() { ... }  
    private long calculerRecetteRestaurants()  
{ ... }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        long recette = r.calculerRecetteTotale();  
  
        // impossible !  
        long taxes = r.calculerTaxesAeroport();  
    }  
}
```

□ Attributs et méthodes : plusieurs niveaux de visibilité

- **public** : élément visible partout (mot-clé `public`).
- **private** : élément à usage interne de la classe, invisible de l'extérieur (mot-clé `private`).
- **package** : élément visible partout dans le même package (pas de mot-clé, cas par défaut)

```
public class Test {  
    public int var1;          // attribut public  
    private int var2;         // attribut privé  
    int var3;                // attribut de visibilité 'package'  
  
    public Test() {}          // constructeur public  
  
    private int method1() {} // méthode privée  
}
```



Un quatrième niveau de visibilité `protected` sera vu plus loin dans les concepts objets (avec l'héritage)

□ Classes : deux niveaux de visibilité seulement

- public : classe visible partout (mot-clé public)

```
public class Test {  
    // classe publique  
}
```

- package : classe visible dans son package uniquement

```
class Test {  
    // classe de visibilité package  
}
```

❑ Méthodes getter et setter

- Bonne pratique : déclarer les attributs d'une classe comme privés.
- Utiliser des méthodes get et set pour accéder aux attributs.

```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    public void setMatricule(long l) {  
        matricule = l;  
    }  
}
```



Tous les environnements de développement proposent des assistants pour générer les getters-setters

□ Intérêt des getters et setters ?

- Il est possible de jouer sur la visibilité des méthodes d'accès.
 - Ex : variable en lecture seule en dehors du package.

Visibilité publique Visibilité package

```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    void setMatricule(long l) {  
        matricule = l;  
    }  
}
```

- Un traitement spécifique peut être placé à chaque accès à une variable.
 - Message de log...

- L'encapsulation
- La composition**
- L'héritage
- Le polymorphisme
- Les interfaces Java

- Il a été vu qu'une classe peut contenir des attributs de type primitif

```
public class Avion {  
    private long matricule;  
}
```

- Une classe peut également porter des attributs typés par d'autres classes.

```
public class Moteur {  
    private int identifiant;  
    private String dateRevision;  
    //...  
}
```

```
public class Avion {  
    private long matricule;  
    private Moteur moteur;  
  
    public Moteur getMoteur() { return moteur; }  
    public void setMoteur(Moteur moteur) { this.moteur = moteur; }  
    //...  
}
```

□ Que produit le code suivant ?

```
Avion monAvion = new Avion();
Moteur monMoteur = new Moteur();
monAvion.setMoteur(monMoteur);

monMoteur.setDateRevision("21/02/2007");

System.out.println(monAvion.getMoteur().getDateRevision());
```

→ Les deux références pointent vers la même zone mémoire !

```
public static int main(String[] args)
{
    Avion monRafale = new Avion();
    Moteur monMoteur = new Moteur();
    monAvion.setMoteur(monMoteur);
    monMoteur.setRevision("21/02/2007");

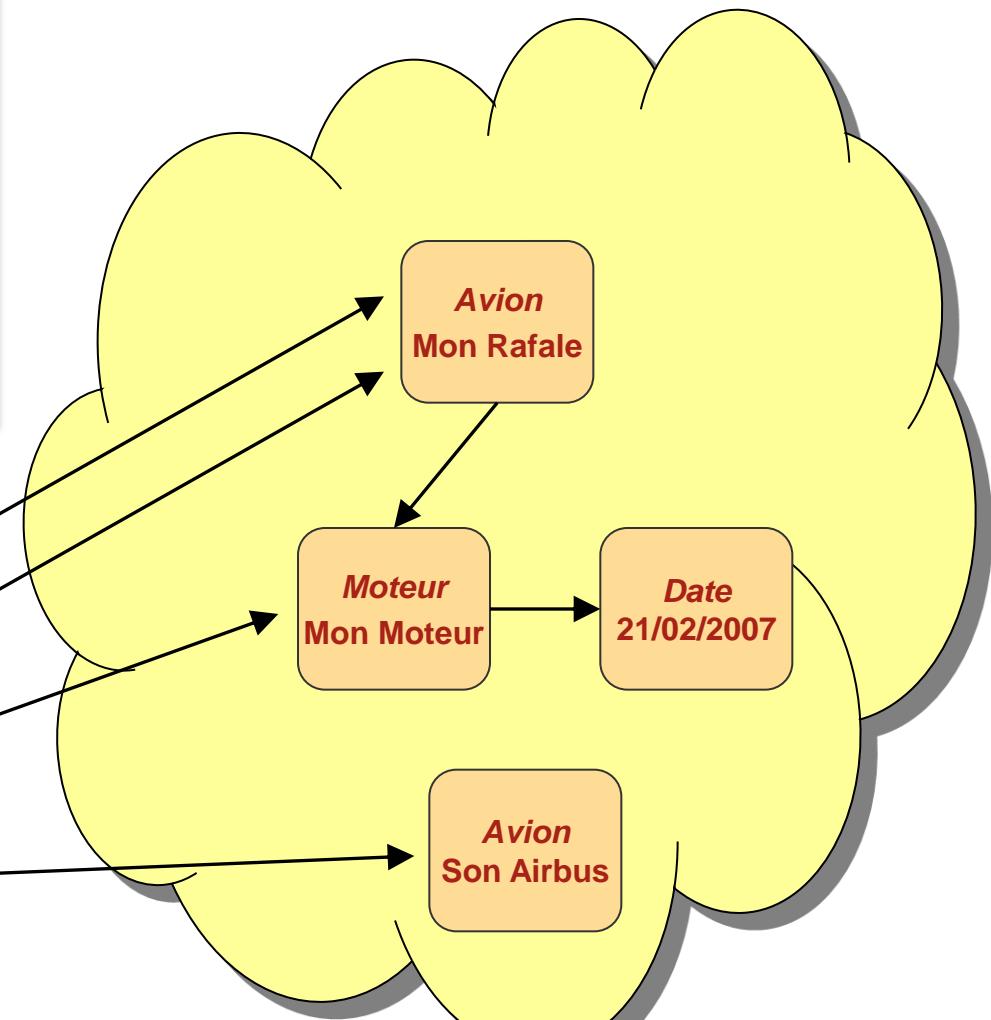
    Avion sonAirbus = new Avion();

    Avion lePlusRapide = monRafale;
}
```

The Stack

- **monRafale**
- **lePlusRapide**
- **monMoteur**
- **sonAirbus**

The Heap



□ Réaliser les travaux pratiques suivants:

- TP 5

- L'encapsulation
- La composition
- L'héritage
- Le polymorphisme
- Les interfaces Java

❑ Exemple

- Un pilote instructeur a un nom, un prénom, un matricule, une licence d'instructeur. Il sait piloter un avion (le faire décoller et atterrir); il peut également enseigner à des apprentis

❑ Modélisation basique

C Pilote	C PiloteInstructeur
<ul style="list-style-type: none">▫ matricule : long▫ nom : String▫ prenom : String● faireDecoller ()● faireAtterrir ()	<ul style="list-style-type: none">▫ matricule : long▫ nom : String▫ prenom : String▫ dateLicenceInstructeur : String● faireDecoller ()● faireAtterrir ()● enseignerDecollage ()● enseignerAtterrissage ()

❑ Duplication de code ⇒ difficile à maintenir

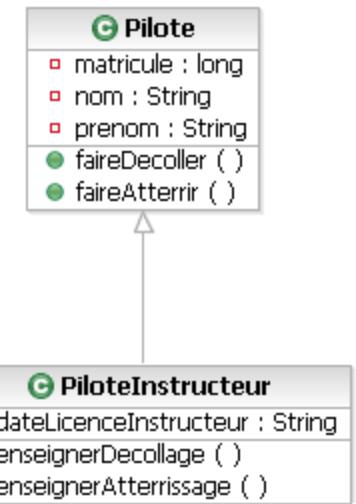
□ Autre modélisation

- Un pilote instructeur **est un type particulier de pilote** qui a une licence d'instructeur. Il sait faire tout ce qu'un pilote sait faire, et de plus, il peut enseigner à des apprentis

□ Relation d'héritage

- Pilote définit les comportements et attributs communs
- PiloteInstructeur ne définit que ce qui lui est propre

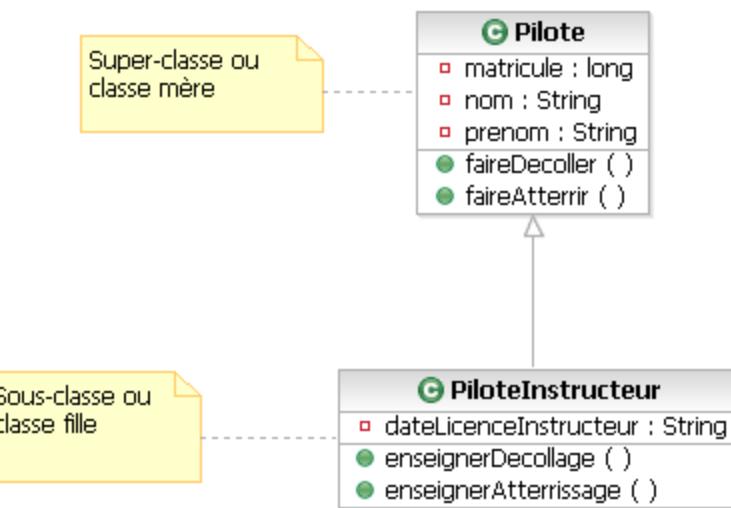
□ Pas de duplication de code



❑ Pas de code spécial dans la super-classe

❑ Mot-clé `extends` dans la déclaration de la sous-classe

➤ `public class PiloteInstructeur extends Pilote`



```
public class PiloteInstructeur extends Pilote {
    private String dateLicenceInstructeur;

    public void enseignerDecollage() {
        // code pour enseigner le décollage
    }

    public void enseignerAtterrissage() {
        // code pour enseigner le décollage
    }
}
```

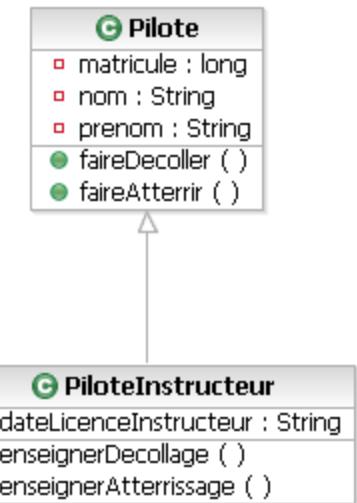
❑ L'état et le comportement d'un objet sont définis le long de la hiérarchie

➤ Structure interne (attributs)

- matricule
- nom
- prenom
- dateLicenseInstructeur

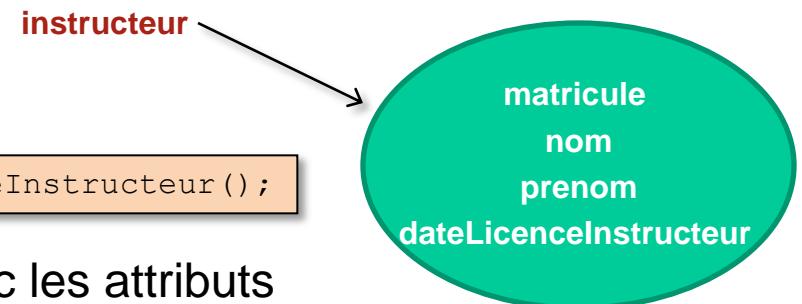
➤ Comportement (méthodes)

- enseignerDecollage ()
- enseignerAtterrissage ()
faireDecoller()
- faireAtterrir ()



□ Création

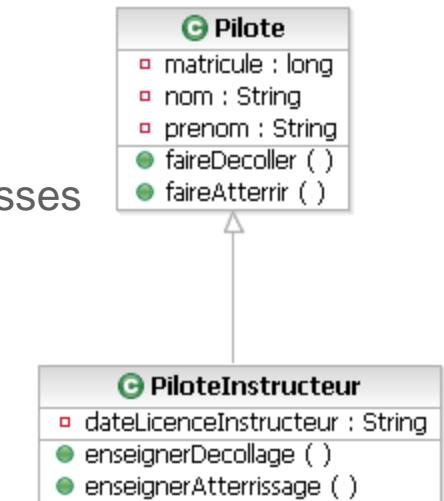
```
PiloteInstructeur instructeur = new PiloteInstructeur();
```



- 1 seul objet créé en mémoire avec les attributs de toutes les super-classes

□ Appels de méthode

- Parcours de la hiérarchie de classes
 - à partir de la classe instanciée vers les super-classes
 - Arrêt dès la première implémentation trouvée (et exécution)

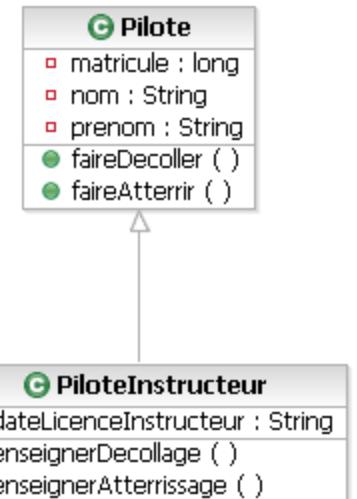


❑ Accès aux attributs et méthodes des super-classes soumis aux contraintes de visibilité

- **public** : oui
- **package** : dépend du package de la classe
- **private** : non

➤ Conséquence: le code suivant ne compile pas

```
PiloteInstructeur instructeur = new PiloteInstructeur();
instructeur.matricule = "1234567"; // attribut invisible
```



❑ **protected** autorise l'accès depuis les sous-classes

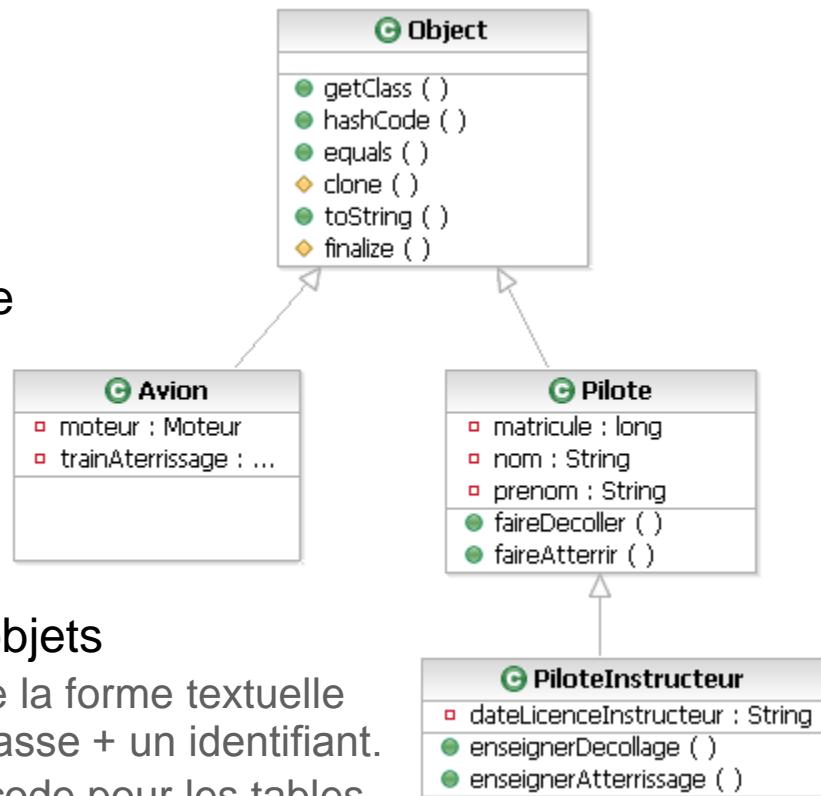
➤ Donne aussi l'accès aux classes du package

□ Arborescence

- Une classe a 1 seule super-classe

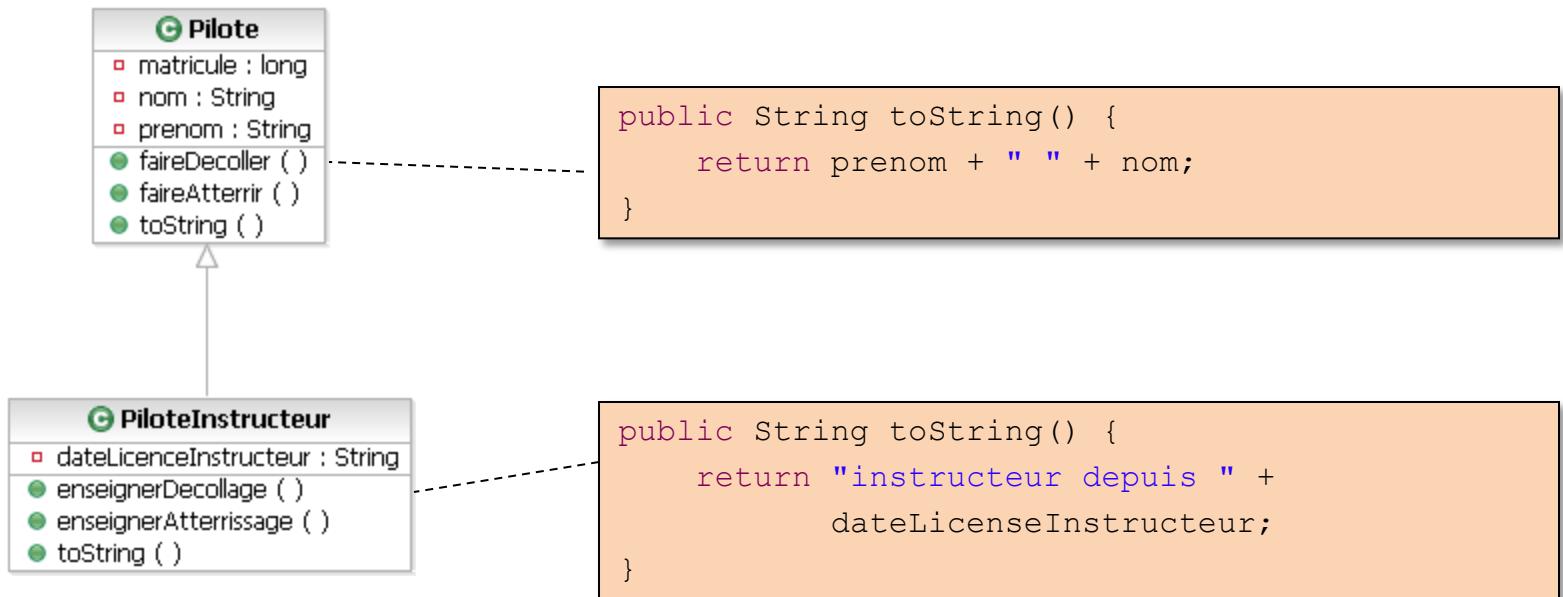
□ Racine : java.lang.Object

- Super-classe implicite
- Méthodes communes à tous les objets
 - **String toString()** : retourne la forme textuelle de l'objet, ici le nom long de la classe + un identifiant.
 - **int hashCode()** : renvoie un code pour les tables de hachage
 - **boolean equals(Object)** : indique si un objet est égal à un autre, L'« égalité » signifiant que 2 objets sont strictement de même type (classes identiques) et possèdent les mêmes valeurs d'attributs
 - ...



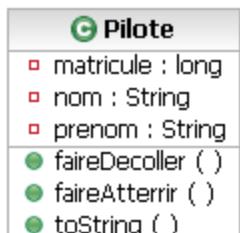
- Définir dans une sous-classe des méthodes avec la même signature que dans la super-classe
- Plusieurs sortes de redéfinitions
 - la définition de la super-classe est ignorée
 - extension de la définition de la super-classe
 - appel de la méthode de la super-classe
 - traitements spécifiques

□ 1^{er} cas : la définition de la super-classe est ignorée

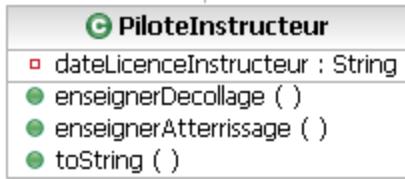


□ 2^{ème} cas : la définition de la super-classe est reprise

- Comment référencer les attributs/méthode de la super-classe ?
 - Si appel au mot-clé **this** : boucle infinie !!!
- Mot-clé **super**
 - Commence la recherche de méthode ou d'attribut dans la super-classe



```
public String toString() {  
    return prenom + " " + nom;  
}
```



```
public String toString() {  
    // appel à toString() de Pilote  
    String nom = super.toString() + "\r\n";  
    return nom + "instructeur depuis" +  
        dateLicenceInstructeur;  
}
```

❑ Contraintes

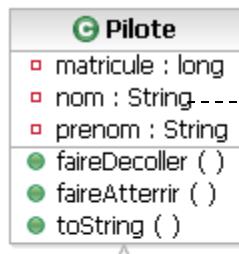
- Le type de retour doit être :
 - strictement identique
 - ou covariant (type spécialisé)
- La visibilité de la méthode de la super-classe ne peut pas être restreinte dans la sous-classe
 - Par exemple : une méthode définie comme publique dans la super-classe ne peut pas être redéfinie comme privée dans la sous-classe
- Contrainte avec le mécanisme d'exception
 - Sera abordée plus loin dans ce cours

❑ Ne pas confondre surcharge et redéfinition

- Redéfinition : exactement la même signature
- Surcharge : paramètres différents

□ Annotation @Override

- Méta-donnée précisant au compilateur que la déclaration d'une méthode redéfinit une méthode de sa super classe.
- Erreur générée par le compilateur java lorsqu'une méthode est annotée avec @Override mais ne redéfinie pas de méthode.
- Facultatif mais doublement utile.



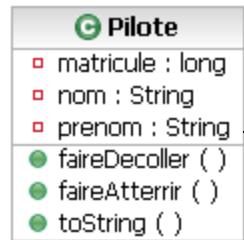
```
public String toString() {
    return prenom + " " + nom;
}
```

```
@Override
public String toString() {
    // appel à toString() de Pilote
    String nom = super.toString() + "\r\n";
    return nom + "instructeur depuis" +
           dateLicenceInstructeur;
}
```

□ Utilisation d'un constructeur hérité

- Appel d'un constructeur de la super-classe avec **super(...)**
 - Similitude avec utilisation de **this(...)**
- Puis traitements spécifiques

□ ou appel implicite à **super()**



```
public Pilote(String nom, String prenom) {
    // appel implicite à super();
    this.nom = nom;
    this.prenom = prenom;
}
```

```
public PiloteInstructeur(String nom, String prenom) {
    super(nom, prenom);
    dateLicenceInstructeur = "21/05/2015"; // aujourd'hui
}
```



super (...) doit être la première instruction du constructeur

❑ Classe avec le mot-clé final

- Interdit la création de sous-classes

```
public final class PiloteInstructeur {  
    // attributs, constructeurs, méthodes  
}
```

❑ Méthode avec le mot-clé final

- Interdit de redéfinir la méthode dans une sous-classe

```
public final void faireDecoller(Avion unAvion){  
    // code pour faire décoller l'avion  
}
```

❑ Rappel

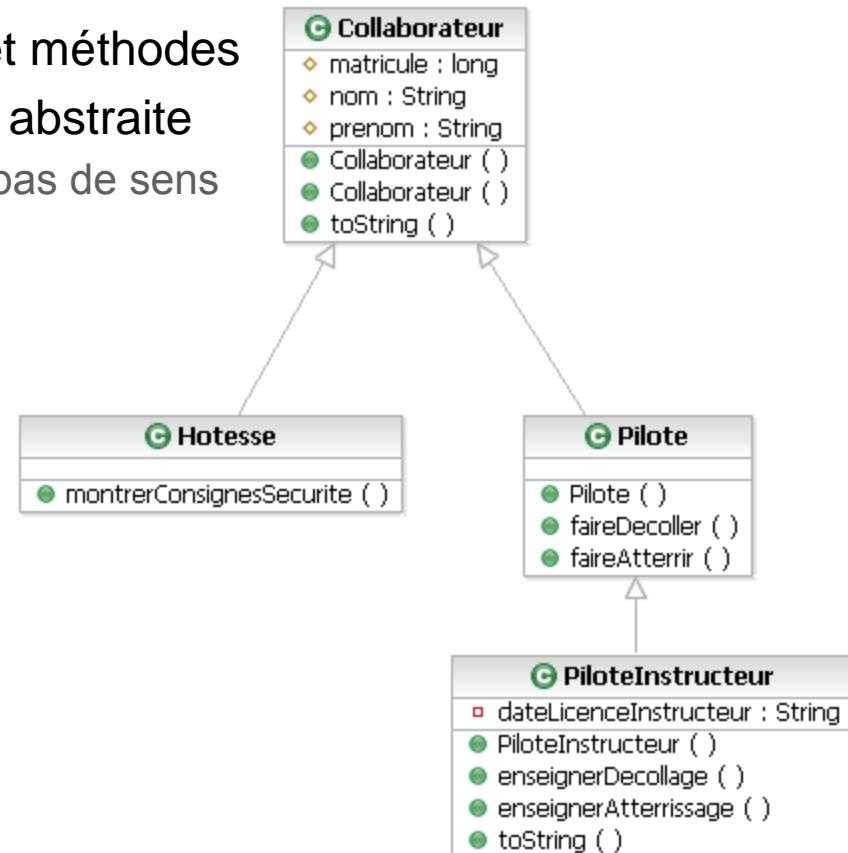
- Pour un attribut, **final** signifie qu'il ne peut pas être redéfini

□ Réaliser les travaux pratiques suivants:

- TP 6.1
- TP 6.2
- TP 6.3
- TP 6.4

□ Classe Collaborateur

- Crée pour factoriser attributs et méthodes
- Un collaborateur est une notion abstraite
 - new Collaborateur () n'a pas de sens
 - Comment l'interdire ?



❑ Classe qui ne possède pas d'instance

❑ Propose une interface pour des sous-classes

- Fournit une interface commune
 - Attributs
 - Méthodes
- Les spécificités sont dans les sous-classes
 - Ajout d'attributs
 - Redéfinition/ajout de méthodes

❑ Utilisée dans le cadre d'une généralisation

- Factorisation des attributs et des méthodes
- Super-classe d'une hiérarchie

❑ Collaborateur est abstraite

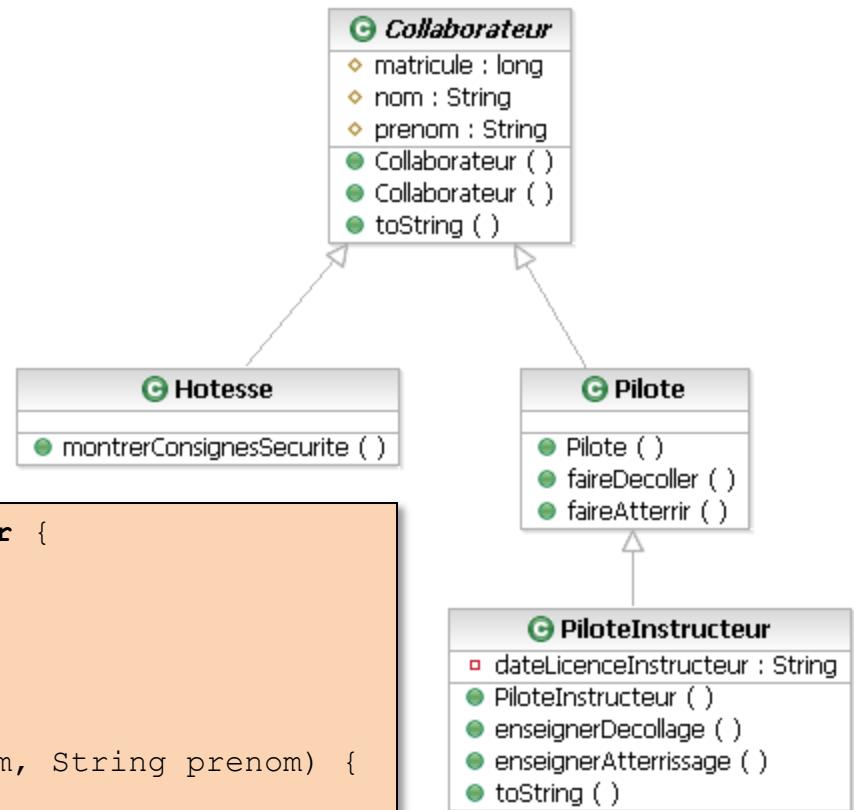
- Notée en italique

❑ Mot-clé abstract

```
public abstract class Collaborateur {
    protected long matricule;
    protected String nom;
    protected String prenom;

    public Collaborateur(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }

    public String toString() {
        return prenom + " " + nom;
    }
}
```



□ Une classe abstraite peut avoir des constructeurs

- Factorisation de l'initialisation
- Appelés par les constructeurs des sous-classes
 - Mais appel direct (`new`) interdit

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```

```
public class Pilote extends Collaborateur {  
    public Pilote(String nom, String prenom) {  
        super(nom, prenom);  
    }  
}
```

□ Exemple salaires des collaborateurs

- `calculerSalaire` renvoie le salaire
- `annoncerSalaire` écrit le salaire dans la console
- comment factoriser le code ?

Hotesse

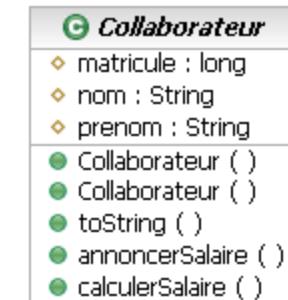
```
public long calculerSalaire() {
    return salaire;
}

public void annoncerSalaire() {
    System.out.println("Salaire : " + calculerSalaire());
}
```

Pilote

```
public long calculerSalaire() {
    return nbHeuresVol * prixHeureVol;
}

public void annoncerSalaire() {
    System.out.println("Salaire : " + calculerSalaire());
}
```



Hotesse

```
salaire : long
montrerConsignesSecurite()
calculerSalaire()
```

Pilote

```
nbHeuresVol : int
prixHeureVol : long
Pilote()
faireDecoller()
faireAtterrir()
calculerSalaire()
```

Mot-clé abstract **Non complète**

- Signature uniquement
 - Pas d'accolades { }. Se termine par ;
- Pas d'algorithme
- Spécifiée dans une classe abstraite

```
public abstract long calculerSalaire();
```

 Redéfinition obligatoire dans les sous-classes concrètes

- Définir l'algorithme

 Ne peut pas être privée, ni finale

Collaborateur

```
public abstract long calculerSalaire();

public void annoncerSalaire() {
    System.out.println("Salaire : " + calculerSalaire());
}
```

Hotesse

```
public long calculerSalaire() {
    return salaire;
}
```

Pilote

```
public long calculerSalaire() {
    return nbHeuresVol * prixHeureVol;
}
```

Collaborateur

- ◊ matricule : long
- ◊ nom : String
- ◊ prenom : String
- Collaborateur ()
- Collaborateur ()
- toString ()
- annoncerSalaire ()
- calculerSalaire ()

Hotesse

- ◻ salaire : long
- montrerConsignesSecurite ()
- calculerSalaire ()

Pilote

- ◻ nbHeuresVol : int
- ◻ prixHeureVol : long
- Pilote ()
- faireDecoller ()
- faireAtterrir ()
- calculerSalaire ()



Rappel : on recherche toujours les méthodes à partir de la classe courante.

□ La redéfinition d'une méthode abstraite

- Est obligatoire dans une sous-classe concrète
- Sinon la sous-classe doit elle-même être abstraite

□ Une classe abstraite

- Ne peut pas avoir d'instance
- Peut contenir des attributs
- Peut contenir des méthodes concrètes
- Peut contenir des constructeurs

- L'encapsulation
- La composition
- L'héritage
- Le polymorphisme
- Les interfaces Java

□ **Un objet peut être vu sous plusieurs formes**

- Exemple : un pilote instructeur peut être vu comme PiloteInstructeur, Pilote, Collaborateur ou Object

□ **Plusieurs objets différents peuvent être vus sous la même forme**

- Ex : un pilote instructeur, une hôtesse et un mécanicien peuvent être vus comme des collaborateurs

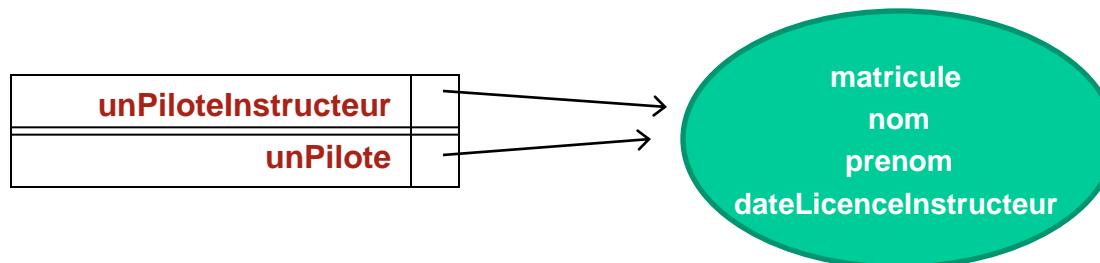
□ Définitions

- Type réel d'un objet : classe dont le constructeur est appelé pour créer l'objet
- Type déclaré : classe utilisée pour manipuler l'objet

□ Upcasting : type déclaré = superclasse du type réel

- Utilisation d'une variable du type de la super-classe à la place d'une variable du type réel de l'objet

```
PiloteInstructeur unPiloteInstructeur = new PiloteInstructeur();  
Pilote unPilote = unPiloteInstructeur;
```



❑ Comportement

➤ Compilation

- Vérification de la compatibilité type déclaré/type réel
 - si pas compatible, erreur de compilation
- Les méthodes/attributs accessibles sont ceux de la classe **déclarée**

➤ Exécution

- «liaison dynamique» : détermine le type réel de l'objet à l'exécution, comportement de la classe réelle

- Plusieurs objets différents peuvent être vus sous la même forme
- Permet d'avoir des traitements génériques
- Exemple
 - Une gestionnaire de paie veut calculer le salaire d'un collaborateur
 - Un collaborateur est un pilote, une hôtesse, un mécanicien ...
 - On peut manipuler directement le type Collaborateur

❑ Implémentation

```
public class GestionnairePaie {  
    public void editerAttestationSalaire(Collaborateur collab) {  
        long salaire = collab.calculerSalaire();  
        String texte = "attestation officielle. Montant du salaire : "  
                    + salaire;  
        System.out.println(texte);  
    }  
}
```

```
public static void main(String[] args) {  
    GestionnairePaie paie = new GestionnairePaie();  
  
    Hotesse uneHotesse = new Hotesse();  
    paie.editerAttestationSalaire(uneHotesse);  
  
    Pilote unPilote = new Pilote();  
    paie.editerAttestationSalaire(unPilote);  
}
```

□ Code plus lisible et plus maintenable

Procédural (pseudo code)

```
si collab = 'Pilote'  
    calculerSalairePilote  
  
si collab = 'Hotesse'  
    calculerSalaireHotesse  
  
si collab = 'Mecanicien'  
    calculerSalaireMecanicien
```

Objet

```
collab.calculerSalaire()  
  
(=> mise en œuvre de  
calculerSalaire dans  
toutes les classes concernées)
```

□ On associe souvent polymorphisme et redéfinition

□ Cas pratique

➤ System.out.print(Object unObjet)

```
public void print(Object obj) {  
    String texte = (obj == null) ? "null" : obj.toString();  
    write(texte);  
}
```

- Méthode générique d'affichage
- Mais le texte affiché dépend de la classe réelle de l'objet

- Besoin de convertir dans une classe fille, pour avoir accès aux méthodes spécifiques
- opérateur cast ()
 - exécution ⇒ essaie de convertir dans la classe spécifiée
 - si impossible, erreur : ClassCastException
- Exemple

```
Collaborateur unCollaborateur = new Hotesse();  
/*  
 * Je peux appliquer à unCollaborateur seulement les méthodes  
 * déclarées dans Collaborateur (et éventuellement redéfinies  
 * dans Hotesse) si je veux pouvoir appeler les méthodes  
 * propres à Hotesse  
 */  
Hotesse uneHotesse = (Hotesse) unCollaborateur;
```

□ Réaliser les travaux pratiques suivants:

- Suite et fin du TP 6

- L'encapsulation
- La composition
- L'héritage
- Le polymorphisme
- Les interfaces Java

❑ Exemple

➤ Apprentissage du pilotage : utilisation d'un simulateur puis d'un avion

- Mêmes boutons, mêmes comportements, même "contrat"
⇒ mêmes signatures de méthode
- Mais pas la même implémentation

➤ Utiliser l'héritage ?

- Inadapté : un simulateur et un avion sont très différents, pas d'attribut ou de méthode à factoriser
- Un simulateur n'est pas une spécialisation d'avion

□ Interface : contrat que respecte une classe

- Signature de méthodes, documentation
- Pas d'implémentation
- Des constantes

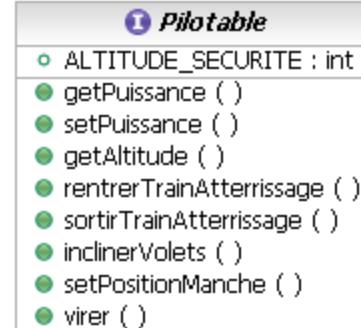
□ Mise en œuvre

- Déclaration : interface au lieu de class
- Tous les attributs sont implicitement public static final
- Toutes les méthodes sont implicitement public et abstract



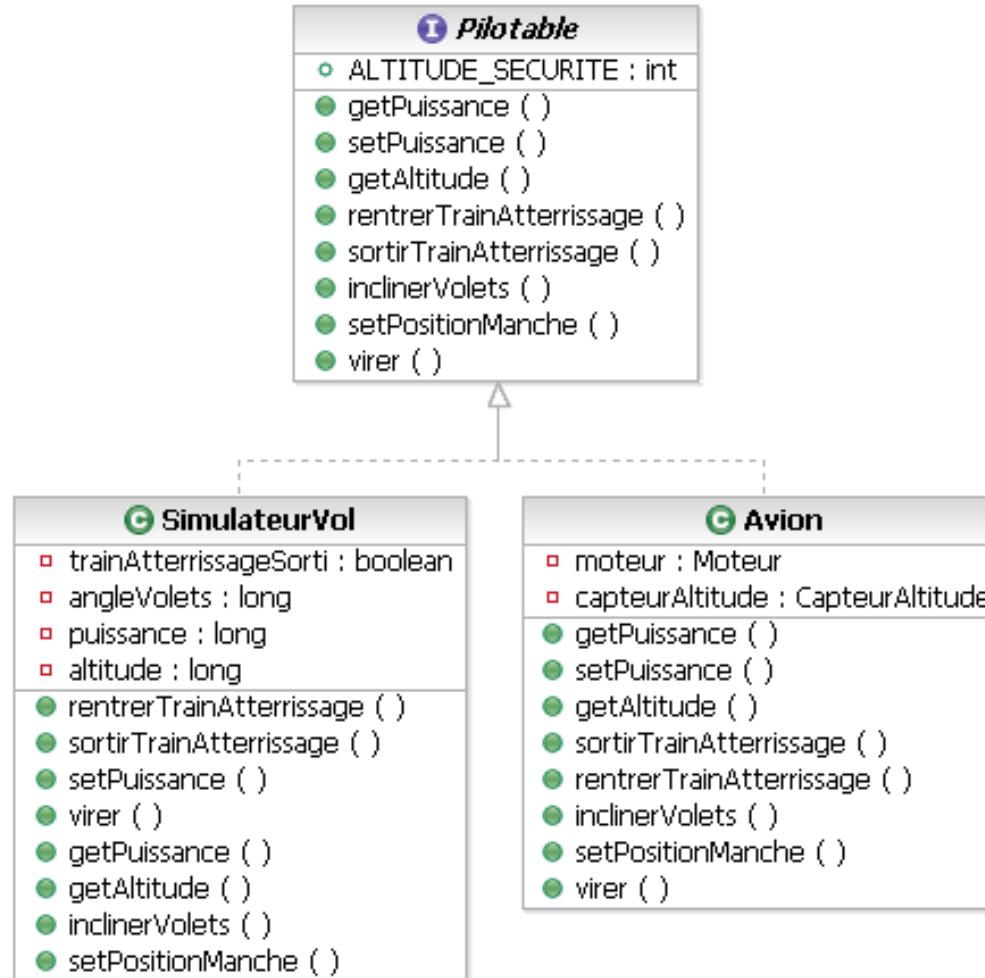
Nom de l'interface : souvent suffixe en "able" : Comparable, Pilotable...

□ Interface Piloteable



```
public interface Piloteable {  
    static final int ALTIITUDE_SECURITE = 100;  
  
    long getPuissance();  
    void setPuissance(long puissanceCible);  
    long getAltitude();  
    void rentrerTrainAtterrissage();  
    void sortirTrainAtterrissage();  
    void inclinerVolets(long angle);  
    void setPositionManche(long position);  
    void virer(long angle);  
}
```

□ Avion et SimulateurVol respectent l'interface Pilotable



□ Mot-clé implements

- La classe implémente toutes les méthodes de l'interface
 - Sinon erreur compilation
 - doivent être déclarées `public`
- Plus éventuellement d'autres méthodes

```
public class Avion implements Pilotable {  
    private Moteur moteur;  
    private CapteurAltitude capteurAltitude;  
  
    public long getPuissance() { return moteur.getPuissance(); }  
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }  
    public long getAltitude() { return capteurAltitude.getAltitude(); }  
    public void sortirTrainAtterrissage() { /* code */ }  
    public void rentrerTrainAtterrissage() { /* code */ }  
    public void inclinerVolets(long angle) { /* code */ }  
    public void setPositionManche(long position) { /* code */ }  
    public void virer(long angle) { /* code */ }  
}
```

□ Interface = groupe de services rendus par la classe

- ex : Pilotable, GestionnaireRadio, ...

□ Utilisation comme type

- Accès seulement aux méthodes et attributs de l'interface (sinon erreur à la compilation)
- Permet d'appliquer des méthodes sans connaître la nature réelle de l'objet ⇒ indépendance vis-à-vis de l'implémentation
 - Implémentation de test
 - Implémentation "réelle"
 - Autre implémentation (autre vendeur par exemple)

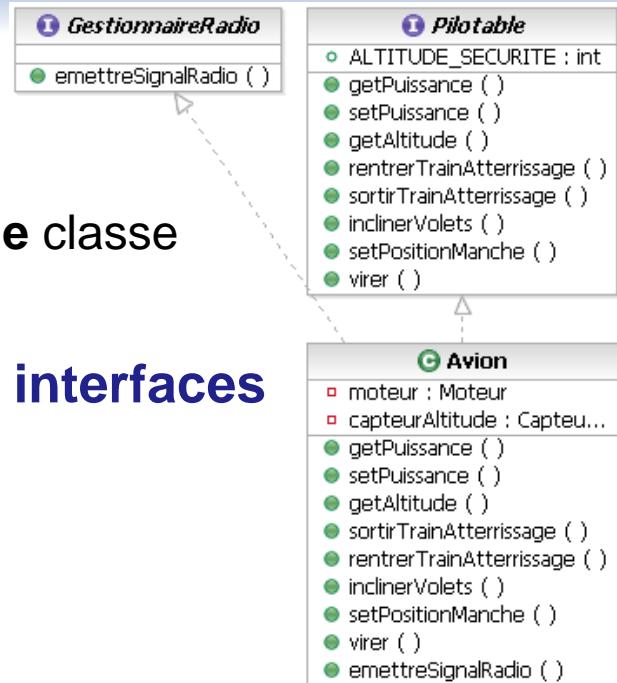
❑ Piloteable : simulateur ou avion

```
public void faireDecoller(Piloteable objetVolant) {  
    // code simpliste pour faire décoller l'avion  
    objetVolant.setPuissance(400);  
    objetVolant.inclinerVolets(10);  
    objetVolant.setPositionManche(3);  
    if (objetVolant.getAltitude() >= Piloteable.ALTIITUDE_SECURITE) {  
        objetVolant.inclinerVolets(0);  
        objetVolant.rentrerTrainAtterrissage();  
    }  
}
```

```
public static void main(String[] args) {  
    Pilote unPilote = new Pilote();  
    Piloteable pilotable = new SimulateurVol();  
    // autre possibilité : pilotable = new Avion();  
    unPilote.faireDecoller(pilotable);  
}
```

❑ N'existe pas en Java

- Une classe ne peut hériter que d'**une seule classe**



❑ Une classe peut implémenter plusieurs interfaces

- Plusieurs "casquettes"

- Objet Pilotable
- Objet avec GestionnaireRadio

```

public class Avion implements Pilotable, GestionnaireRadio {
    // attributs
    // toutes les méthodes de l'interface Pilotable
    public long getPuissance() { return moteur.getPuissance(); }
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }

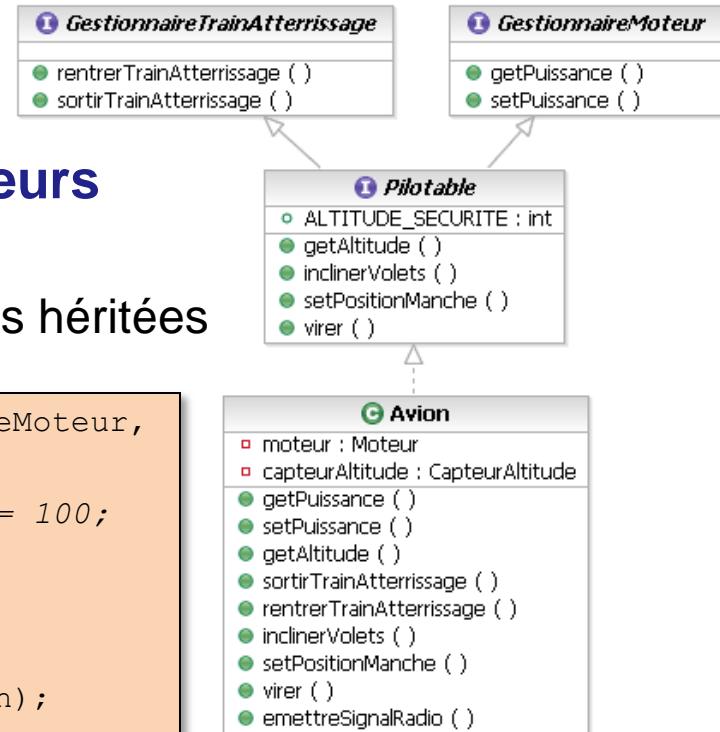
    // toutes les méthodes de l'interface GestionnaireRadio
    public void emettreSignalRadio(String message) { /* code */ }
}
  
```

□ Une interface peut hériter de plusieurs interfaces

- Ensemble des services des interfaces héritées

```
public interface Pilotable extends GestionnaireMoteur,
    GestionnaireTrainAtterrissage {
    public static final int ALTITUDE_SECURITE = 100;

    public long getAltitude();
    public void inclinerVolets(long angle);
    public void setPositionManche(long position);
    public void virer(long angle);
}
```



□ Redéfinition de toutes les méthodes dans les classes d'implémentation

Réaliser les travaux pratiques suivants:

➤ TP 7

- Pourquoi créer des constructeurs dans une classe abstraite ?

- Est-ce que je peux avoir des méthodes concrètes dans une interface ?

□ Que faut-il faire pour que ce code compile ?

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public abstract long calculerSalaire();  
  
    public void annoncerSalaire() {  
        System.out.println("Salaire : " + calculerSalaire());  
    }  
}
```

```
public class Mecanicien extends Collaborateur {  
    public void reparer(Avion unAvion) {  
        // code pour reparer un avion  
    }  
}
```

❑ Notions

- Super-classe
- Classe abstraite
- Interface

❑ En pratique, qu'utilise-t-on ?

- Différents cas d'utilisation

□ **On a déjà une classe (`Pilote`) et on veut créer un cas particulier (`PiloteInstructeur`)**

- PiloteInstructeur hérite de Pilote
 - Pilote reste inchangé
 - Ajout des attributs spécifiques à un instructeur
 - Ajout des méthodes spécifiques à un instructeur
 - Redéfinition des méthodes pour lesquelles l'instructeur répond différemment du pilote

□ On a plusieurs classes existantes et on s'aperçoit que beaucoup d'attributs et méthodes sont dupliqués

➤ Techniquement

- Création d'une super-classe
- Changement du lien d'héritage pour les classes existantes
- Déplacement des méthodes et des attributs communs vers la super-classe

➤ Y a-t-il un sens à créer des instances de la superclasse, ou ne doit-on manipuler que les sous-classes ?

- Si oui, la superclasse est concrète
- Si non, la super-classe est abstraite

□ On veut cacher la complexité d'une classe ou ne montrer qu'une partie de ses méthodes

- Ex : TP

➤ Techniquement

- Créer une interface qui ne déclare que les méthodes à exposer
- Utiliser l'interface comme type de retour et non la classe d'implémentation
 - Les clients ne doivent connaître/manipuler que l'interface

□ On veut pouvoir changer facilement l'implémentation d'un objet

- Ex : changer la persistance d'un objet – SGBDR – fichier texte
- Ex : remplacer l'objet par une implémentation de test (pilote avec simulateur / avion)

➤ Techniquement

- Créer une interface qui offre les mêmes services de la classe
- Référencer cette interface plutôt que la classe
- Paramétriser la création de l'objet

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- Approche conventionnelle**
- Gestionnaire d'exceptions**
- Objet de type Exception**
- Hiérarchie des exceptions**
- Code protégé**

❑ Historiquement: utilisation de codes d'erreur

- les fonctions retournent des codes
- tester les codes à chaque appel de fonction
- traiter les erreurs ou continuer l'exécution

❑ Maintenance difficile

❑ Pas de catégorie d'erreurs

❑ Risque de cas d'erreurs non traitées

□ Séparer les traitements “exceptionnels” du code “normal”

- Eviter les structures conditionnelles difficiles à maintenir

□ Propager les erreurs dans la pile de messages

- Mécanisme très simple et intuitif pour le développeur

□ Classifier les erreurs

- Par la création de classes représentant les exceptions
- Utilisation de l'héritage pour regrouper les exceptions

- ❑ Pour traiter une anomalie
- ❑ « Envoi » d'un objet Exception
- ❑ Recherche du gestionnaire approprié

- Si trouvé (dans la pile d'appels)
 - Traitement de l'exception
 - Retour au programme
- Sinon
 - Déclenchement d'une erreur et arrêt de l'exécution

□ Cration d'un objet Exception

- en genal sous-classe de `java.lang.Exception`
- `new ClasseException();`

□ Envoi de l'exception

- Expression a inserer dans le code de la methode
- instruction `throw`
- `throw new ClasseException();`

- La méthode “contient” une liste d’exceptions potentielles
- Il faut spécifier toutes les exceptions qui peuvent être envoyées

➤ Dans la déclaration de la méthode: clause *throws*

```
typeRetour nomMethode( . . . ) throws  
liste des types d'exceptions {  
    . . .  
}
```

➤ Dans la liste d’exceptions pouvant être levées par la méthode, chaque exception est séparée par une virgule.

- **Blocs try-catch**
- **Séparation exécution / traitement d'erreur**

```
try {  
    //exécution sécurisée  
} catch (TypeException e) {  
    //gestionnaire d'exception  
}
```

❑ Exemple d'exception

```
double diviser(int a, int b) throws DivException {
    if (b == 0) {
        throw new DivException("division par zéro");
    }
    return a / b;
}

double appelerDiviser(int x, int y) {
    try {
        return diviser(x, y);
    } catch (DivException e) {
        return 0;
    }
}
```

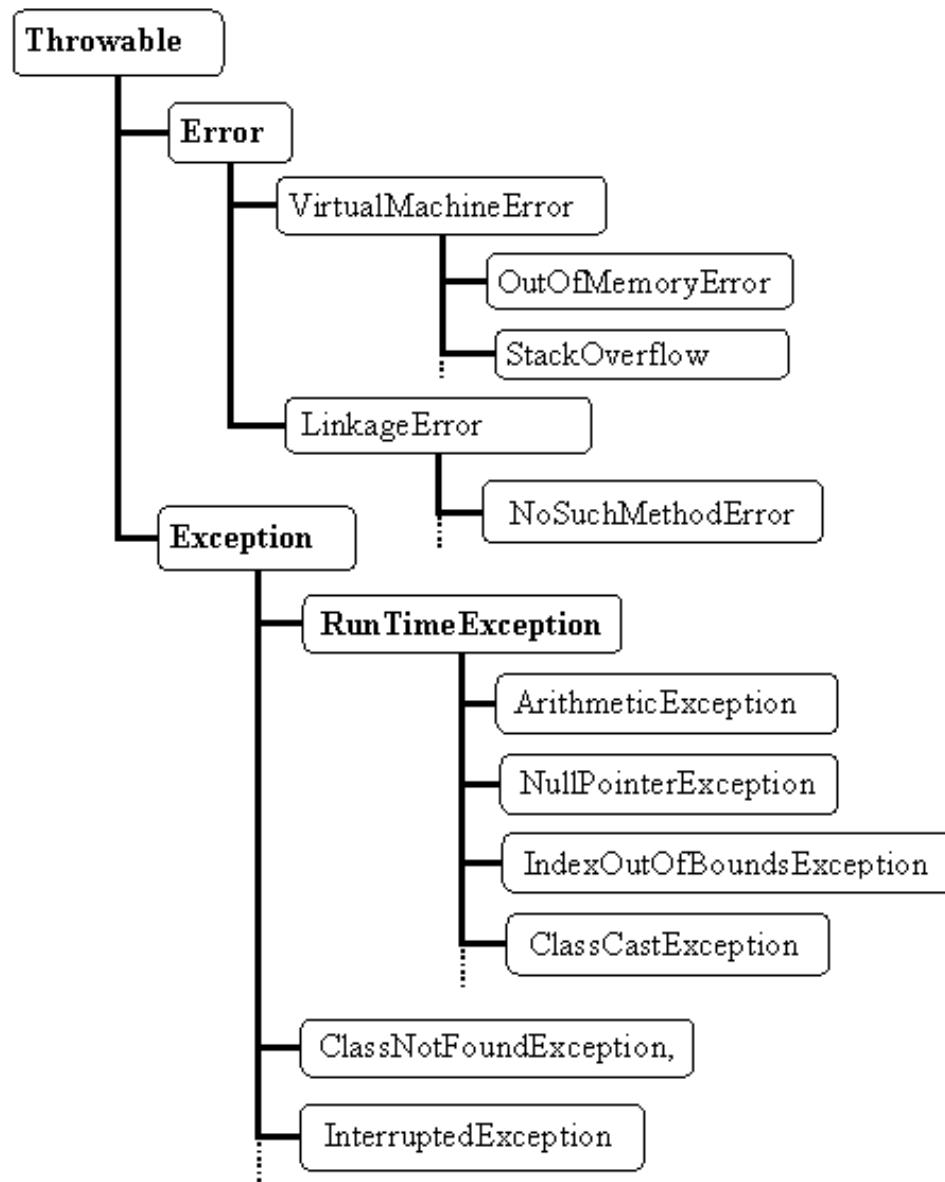
❑ Racine : classe Throwable

- Signification: « objet qui peut être renvoyé »
- Contient des méthodes générales

❑ 2 sous-classes de Throwable

= 2 catégories principales:

- **Error** (erreurs système et de compilation)
- **Exception**
 - **RuntimeException** (erreurs à l'exécution)
 - ... (erreurs applicatives)



- Une exception peut recevoir des messages
- Exemples

- e.getMessage()
- e.printStackTrace()
- e.getCause()

❑ Exceptions `RuntimeException`

- Pas de problème à la compilation (pas besoin de `try-catch`), mais erreur déclenchée à l'exécution
- Deux causes :
 - Mauvais développement → à corriger
 - Erreur déclenchée par le système (imprévisible) → pas de correction, sinon code illisible

❑ Autres exceptions

- Doivent être traitées
 - `try-catch`
 - ou `throws NomException`
 - la méthode ne traite pas l'exception mais la propage
- Sinon erreur de compilation

□ Créer une nouvelle classe

- Sous-classe directe de `Exception` ou de `RuntimeException`

```
public class DecollageException extends Exception {  
    public DecollageException() {  
    }  
  
    public DecollageException(String msg) {  
        super(msg);  
    }  
}
```

- Assure l'exécution d'un bloc d'instructions même en cas d'exception
- Assurer un état cohérent dans le programme
 - Fermeture de fichier par exemple

```
try {...}           // code protégé
catch (...) {...} // plusieurs gestionnaires
catch (...) {...} // d'exceptions
finally {...}     // bloc toujours exécuté
```

❑ Une méthode redéfinie dans une sous-classe :

- Ne doit pas ajouter d'exception supplémentaire à la liste d'exceptions de la méthode qu'elle redéfinie
 - ⇒ Sauf dans le cas d'un constructeur
- Peut déclarer une liste plus restreinte d'exceptions
- Peut déclarer une exception plus spécialisée

Réaliser les travaux pratiques suivants:

- TP 8

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- Les types primitifs permettent d'effectuer des opérations simples et rapides

```
int i;  
int j;  
i = 2 + 3;  
j = i + 2;  
j++;
```

- Pour chaque type primitif il existe une classe dite « wrapper » (emballage).

- Une classe wrapper contient une variable d'un type primitif donné et des outils facilitant la manipulation de cette variable.

Integer



int

Long



long

Character



char

...

- Classes wrapper existantes : Boolean, Byte, Short, Integer, Long, Float, Double, Character
- Méthodes proposées par les classes wrapper :
 - `toString()` convertit en chaîne de caractères la variable contenue dans le Wrapper
 - `intValue()`, `doubleValue()`, ... pour récupérer le type primitif encapsulé par le Wrapper.
 - `parseInt()`, `valueOf()`, ... pour des conversions de texte vers des nombres (type primitif et Wrapper).
 - Les wrappers de types numériques contiennent des constantes `MAX_VALUE` et `MIN_VALUE`

□ Exemples d'utilisation

➤ Méthodes d'instance

```
Integer i = new Integer(12);
String s = i.toString();
double d = i.doubleValue();
Boolean b = new Boolean(true);
boolean b2 = b.booleanValue();
```

➤ Méthodes statiques

```
String s = "12";
int i = Integer.parseInt(s);
boolean b = Boolean.valueOf("true");
```



La plupart de ces méthodes ne s'appliquent pas au type Character qui n'enveloppe pas un nombre.

□ Le mécanisme de boxing permet de convertir automatiquement :

- un type primitif en wrapper (**Autoboxing**)

```
int x = 567;  
Integer y = x;
```

- un wrapper en type primitif (**Unboxing**)

```
Integer y = new Integer(567);  
int x = y;
```

□ Exemple d'utilisation :

Avant Java 5

```
Integer y = new Integer(567);  
int x = y.intValue();  
x++;  
y = new Integer(x);  
System.out.println("y="+y);
```

Depuis Java 5

```
Integer y = new Integer(567);  
y++;  
System.out.println("y="+y);
```

□ Les String représentent les chaînes de caractères en Java

- Les Strings sont des objets
- Initialisation facilitée sans le mot-clé new
- La taille d'une String n'est pas limitée

```
String s1 = new String("hello world");  
  
// privilégier la façon de faire suivante  
String s2 = "hello world";
```

❑ Quelques méthodes utiles

➤ Comparaisons

- int compareTo(String)
- int compareToIgnoreCase(String)

```
String s1 = new String("hello");
String s2 = new String("hello");
System.out.println(s1.compareTo(s2));
```

➤ Longueur de la chaîne

- int length()

```
String s1 = new String("hello");
int longueurChaine = s1.length();
System.out.println(longueurChaine);
```

❑ Manipulation de chaînes

- trim() : supprime les espaces superflus

```
String s1 = new String("      hello world ");
System.out.println(s1);
System.out.println(s1.trim());
```

- toUpperCase(), toLowerCase() ...

```
String s1 = new String("hello world");
System.out.println(s1.toUpperCase());
```

❑ Concaténation

- concat (...) ou opérateur +

```
String s1 = new String("hello");
String s2 = s1 + (" world");
String s3 = s1.concat(" world");
```

❑ Les instances de String sont dites *immuables*

```
String s = "hello world";
s = "bye bye world";
```

Le premier objet référencé par *s* est détruit puis *s* référence un nouvel objet.



Utilisée à grande échelle (dans des boucles), la concaténation de String est une opération peu performante.

- Pour gagner en performances, on peut utiliser la classe `StringBuilder` (du package `java.lang`) à la place de la classe `StringBuffer`.

Types	Version JDK	Synchronisation	Performance
StringBuffer	à partir du 1.0	Oui	++
StringBuilder	à partir du 1.5	Non	+++++

□ Critère de sélection entre String, StringBuffer et StringBuilder:

- Si la chaîne de caractères ne va pas changer, utilisez la classe `String` parce qu'un objet de type `String` est un objet non mutable.
- Si votre chaîne de caractères va beaucoup changer et est accédée par un seul thread, utilisez `StringBuilder` parce que `StringBuilder` n'est pas synchronisé et sera donc plus rapide que `StringBuffer`.
- Si votre chaîne de caractères va beaucoup changer et est accédée par plusieurs threads, utilisez `StringBuffer` parce que `StringBuffer` est synchronisé.

```
StringBuilder message = new StringBuilder();
message.append("msg 1");
message.append( "msg 2");
message.append(" msg 3");
System.out.println(message);
```



□ Utilisation des chaînes de caractères String

Mauvaise gestion des String = « surcréation » d'objets + surconsommation mémoire + GC fréquent

- Concaténer les chaînes de caractères, utiliser la classe StringBuffer/StringBuilder;
- préférer `String s = "MaChaineDeCaracteres"` à `String s = new String("MaChaineDeCaracteres")`
- créer les chaînes de caractères avec la bonne taille (Findbugs peut vous y aider);

□ Comment détecter une mauvaise utilisation des String ?

```
package teststring;

public class Main {
    public static void main(String[] args) {
        String s = "MaChaineDeCaractere";
        for (int i = 0; i < 100000; i++) {
            s = s + "Ajout";
        }
    }
}
```

□ Utilisation des chaînes de caractères

➤ Profiler ce code et regarder la mémoire :

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]	209 170 072 B (100 %)	209 170 072 B (100 %)	40 062 (66.8 %)
java.lang.String	25 464 B (0 %)	25 464 B (0 %)	10 038 (16.7 %)
java.lang.StringBuilder	16 848 B (0 %)	16 848 B (0 %)	10 013 (16.6 %)
byte[]	680 B (0 %)	680 B (0 %)	1 (0 %)
java.util.HashMap\$Entry[]	144 B (0 %)	144 B (0 %)	5 (0 %)
sun.net.www.protocol.file.FileURLConnection	96 B (0 %)	96 B (0 %)	1 (0 %)
java.lang.Object[]	56 B (0 %)	56 B (0 %)	1 (0 %)
java.util.Hashtable\$Entry[]	56 B (0 %)	56 B (0 %)	1 (0 %)
java.lang.Package	48 B (0 %)	48 B (0 %)	1 (0 %)

→ beaucoup d'allocations de char[] ou de String.
 → ce projet est un bon candidat pour l'utilisation de StringBuffer.

➤ le temps d'exécution est de **1.73 s**

>Package	Base Time (seconds)	Average Base Time (seconds)	Cumulative Time (seconds)	Calls
# teststring	1,735517	1,735517	1,735517	1
teststring	1,735517	1,735517	1,735517	1
main(java.lang.String[]) void	1,735517	1,735517	1,735517	1

☐ changeons le String en StringBuffer

```
package teststring;

public class Main {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("MaChaineDeCaractere");
        for (int i = 0; i < 100000; i++) {
            s.append("Ajout");
        }
    }
}
```



Profiler ce code et regarder la mémoire :

→ Bcp moins de char[] en mémoire (~40000 objets à 77)

→ Temps CPU = 0.0036s

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		5 592 B (73.7 %)	77 (39.9 %)
byte[]		664 B (8.7 %)	1 (0.5 %)
java.util.HashMap\$Entry[]		144 B (1.9 %)	5 (2.6 %)
sun.net.www.protocol.file.FileURLConnection		96 B (1.3 %)	1 (0.5 %)
java.lang.String		96 B (1.3 %)	38 (38.7 %)
java.util.Hashtable\$Entry[]		56 B (0.7 %)	1 (0.5 %)

>Package	Base Time (sec...)	Average Base ...	Cumulative Tim...	Calls
teststring	0,003657	0,003657	0,003657	1
teststring	0,003657	0,003657	0,003657	1
main(java.lang.String[])	0,003657	0,003657	0,003657	1



- changeons les StringBuffer en StringBuilder (pas besoin d'être thread-safe)

```
package teststring;

public class Main {
    public static void main(String[] args) {
        StringBuilder s = new StringBuilder("MaChaineDeCaractere");
        for (int i = 0; i < 100000; i++) {
            s.append("Ajout");
        }
    }
}
```

Profiler ce code et regarder la mémoire :

- Même nombre de char[] en mémoire
 - Mais temps CPU baisse = **0.0034s**

Class Name - Allocated Objects	Bytes Allocated ▾	Bytes Allocated	Objects Allocated
char[]		38 560 B (95 %)	77 (93 %)
byte[]		664 B (1.6 %)	1 (0.5 %)
[java.util.HashMap\$Entry[]]		144 B (0.4 %)	5 (2.6 %)
[java.lang.String]		120 B (0.3 %)	38 (59.7 %)

>Package		Base Time (sec...)	Average Base ...	Cumulative Tim...	Calls
└ teststring	+	0,003423	0,003423	0,003423	1
└ teststring	◊	0,003423	0,003423	0,003423	1
└ main(java.lang.String[])	◊	0,003423	0,003423	0,003423	1

□ initialiser le StringBuilder directement avec la bonne taille

```
package teststring;

public class Main {
    public static void main(String[] args) {
        StringBuilder s = new StringBuilder(50019);
        s.append("MaChaineDeCaractere");
        for (int i = 0; i < 100000; i++) {
            s.append("Ajout");
        }
    }
}
```

111001010
0010101010
0101110101
111000\$000
1010101001
1001001010
1010101010

- Le nombre de char[] diminue (77 → 66)
- Le temps CPU baisse = **0.0032s**

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		1112 B (35.7 %)	66 (36.3 %)
byte[]		6648 B (11.3 %)	1 (0.5 %)
java.util.HashMap\$Entry[]		144 B (4.6 %)	5 (2.7 %)
sun.net.www.protocol.file.FileURLConnection		96 B (3.1 %)	1 (0.5 %)
java.lang.String		96 B (3.1 %)	38 (20.9 %)

>Package	Base Time (sec...)	Average Base ...	Cumulative Tim...	Calls
- teststring	0,003230	0,003230	0,003230	1
teststring	0,003230	0,003230	0,003230	1
main(java.lang.String[])	0,003230	0,003230	0,003230	1

□ Sortie standard : `System.out`

□ Erreur standard : `System.err`

□ Écrire

- `print(valeur)`
- `println(valeur)`
 - valeur : type primitif ou objet
- `printf(message, args)`
 - message : chaîne de caractère avec paramètres
 - args : primitives ou objets à remplacer dans le message

□ Java fournit une méthode permettant de produire des messages formatés

- Pattern identique à printf en langage C
- Définit dans `java.util.Formatter`
- Raccourcis: `String.format()`, `System.out.format()`,
`System.out.printf()`

```
double a = 3.87;
double b = 5.0;
double mult = a * b;
System.out.printf("%2$.2f * %1$.2f = %3$.2f", a, b, mult);
// affiche: 5,00 * 3,87 = 19,35
```

➤ java.util.Date

- La plupart des méthodes sont dépréciées : existent mais ne doivent pas être utilisées car elles peuvent disparaître dans une future version du JDK
- Pas de gestion des zones (Locale)

➤ java.util.Calendar

- Tient compte de la zone (fuseaux horaires, etc)
- Réfèrentent une date/heure avec accès au jour, heure, secondes...
- Calendar.getInstance()
- getTime() : retourne la Date associée à ce calendrier

➤ java.text.DateFormat et java.text.SimpleDateFormat

- conversion Date/String
- constructeur avec un motif pour la conversion
- String ⇒ Date: Date parse(String)
- Date ⇒ String: String format(Date)

```
String s1 = new String("hello");
Calendar calendrierMaintenant = Calendar.getInstance();
Date maintenant = calendrierMaintenant.getTime();
SimpleDateFormat formattage = new SimpleDateFormat("EEEE dd MMMM yyyy");
System.out.println(formattage.format(maintenant));
// affiche: samedi 31 janvier 2009
```

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- Les collections**
- Les Listes**
- Les Sets**
- Les Maps**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

- ❑ Les collections sont l'évolution logique des tableaux
- ❑ Les tableaux peuvent être utilisés pour des opérations simples
- ❑ Mais...
 - La taille d'un tableau est définie lors de sa création
 - Si on supprime un élément au milieu d'un tableau, il faut gérer le décalage de tous les éléments qui se trouvaient après l'élément supprimé
 - Les éléments sont obligatoirement indexés par des entiers

```
String[] tableau = new String[10];
tableau[0] = "Hello !";
tableau[1] = "Bonjour !";
```

❑ Les collections sont des classes Java qui facilitent la gestion d'ensembles d'éléments

- Opérations courantes :
 - Ajout d'élément
 - Suppression d'élément
 - Parcours de la collection

❑ De nombreuses collections sont fournies avec Java

- Package `java.util`
- Implémentent généralement l'interface `java.util.Collection`
- Permettent de stocker des références vers tous types d'objets
- Pas de taille maximum prédéfinie
- Elles sont optimisées en fonction de besoins précis

- Depuis Java 5, tout l'API des collections a été mis à jour pour profiter des types génériques
- Les collections peuvent donc être typées
 - Les exemples du cours utilisent les types génériques
 - Il est possible de les convertir en type brut en supprimant les générique et en ajoutant des conversions explicites
 - Par exemple, le code suivant:

Java 5

```
List<String> noms = new ArrayList<String>();  
noms.add("Fred");  
String nom = noms.get(0);
```

```
List noms = new ArrayList();  
noms.add("Fred");  
String nom = (String) noms.get(0);
```

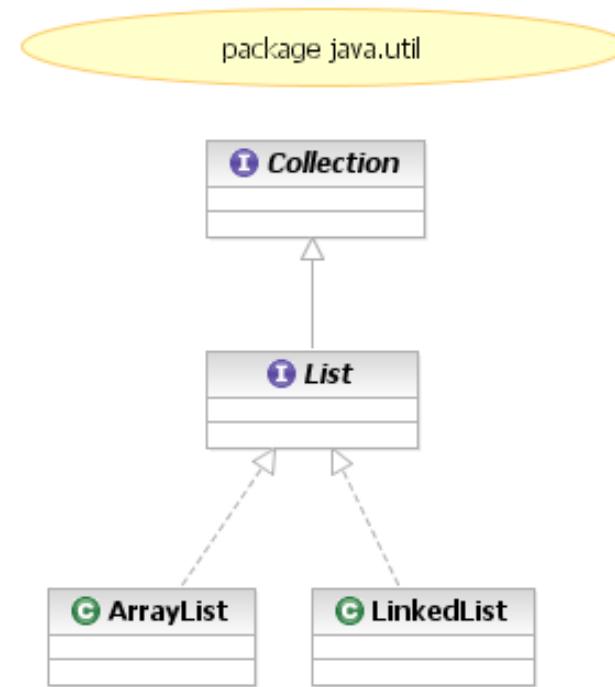
- Les collections**
- Les Listes**
- Les Sets**
- Les Maps**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

□ Les listes sont des collections

- Implémentent l'interface `java.util.List` qui hérite de l'interface `Collection`

□ Les listes sont indexées

- Une liste est donc ordonnée



❑ ArrayList

- La plus utilisée
- Souvent vue comme un "tableau dynamique"
- Excellentes performances pour le parcours d'éléments

❑ LinkedList

- Liste chaînée
- Excellentes performances lors d'insertion/suppression d'éléments
- Mauvaises performances pour le parcours d'éléments



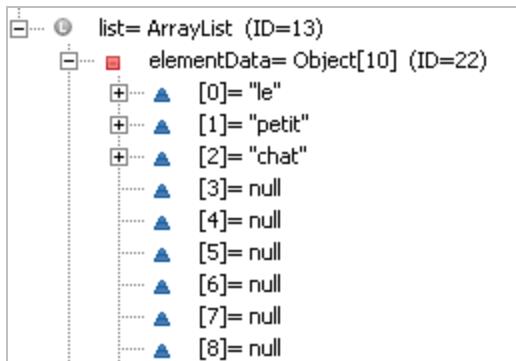
Il existe historiquement la classe Vector (généralement déconseillée car son implémentation est « synchronisée » et peu performante).

□ Les éléments sont ajoutés avec la méthode add (...)

- Par défaut, chaque élément est indexé en fonction de son ordre d'arrivée

□ Exemple : ArrayList

- Éléments stockés dans un tableau à l'intérieur de l'objet
- A chaque ajout, vérification que la taille maximum du tableau n'a pas été atteinte.
 - Si la taille maximum est atteinte, un nouveau tableau est créé



```
List<String> list = new ArrayList<String>();
list.add("le");
list.add("petit");
list.add("chat");
```



Il est possible (mais rare) d'insérer des éléments hétérogènes dans une même liste : String, Integer, CompteCourant...

□ Méthode `remove (...)`

- Possibilité de supprimer l'objet à partir d'une référence à l'objet lui-même, ou à partir de l'index de l'objet dans la liste.

```
List<User> list = new ArrayList<User>();  
User u1 = new User("jean", "dupont");  
list.add(u1);  
User u2 = new User("jean", "durand");  
list.add(u2);  
User u3 = new User("jean", "martin");  
list.add(u3);  
  
// suppression de la référence vers u3  
list.remove(u3);  
  
// suppression de l'élément en position 0  
list.remove(0); //
```

size()

- Renvoie le nombre d'éléments de la collection

 isEmpty()

- Renvoie 'true' si la collection est vide

 toArray()

- Crée un tableau contenant tous les éléments de la liste



Ces méthodes sont déclarées dans l'interface Collection.
Elles ne sont pas spécifiques aux listes.

□ Iterator

- Permet de parcourir une collection en récupérant les éléments successivement
- Méthode de parcours homogène pour tous les types de collections
- Garantit la cohérence de la collection parcourue
 - Permet de retirer d'une collection un élément pointer par l'itérateur
 - Pas de modifications externes autorisées pendant le parcours

□ Récupération d'un `Iterator<T>` sur une collection donnée

- Méthode `iterator()`
- Sur l'objet `ArrayList<String>`, retourne un `Iterator<String>`

□ `Iterator` contient 3 méthodes :

- `boolean hasNext()`
 - Renvoie `true` s'il reste des éléments à parcourir dans la liste
- `T next()`
 - Renvoie le prochain objet stocké dans la liste
- `void remove()`
 - Supprime l'élément en cours de la liste

□ Exemple

```
List<User> list = new ArrayList<User>();
User u1 = new User("jean", "dupont");
list.add(u1);
User u2 = new User("jean", "durand");
list.add(u2);
User u3 = new User("jean", "martin");
list.add(u3);

for (Iterator<User> iterator = list.iterator(); iterator.hasNext();) {
    User user = iterator.next();
    System.out.println(user);
}
```



La classe d'implémentation de l'interface Iterator n'est généralement pas connue.

□ Exemple avec boucle "for each"

```
List<User> list = new ArrayList<User>();  
User u1 = new User("jean", "dupont");  
list.add(u1);  
User u2 = new User("jean", "durand");  
list.add(u2);  
User u3 = new User("jean", "martin");  
list.add(u3);  
  
for (User user : list) {  
    System.out.println(user);  
}
```

❑ Les listes acceptent les doublons

- Ils sont positionnés à un index différent
- La liste stocke plusieurs références vers le même objet

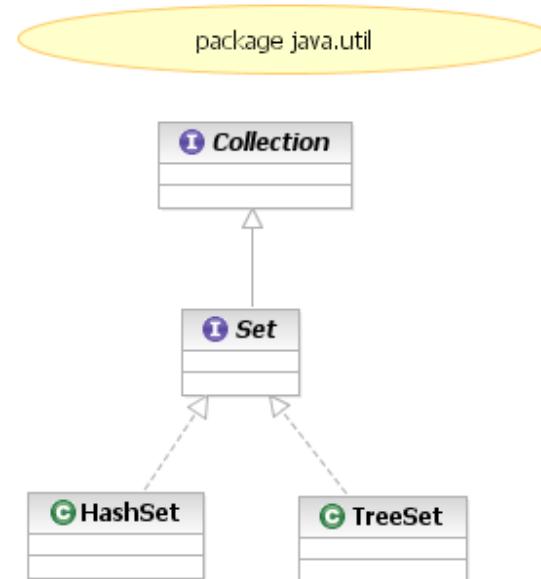
```
List<User> list = new ArrayList<User>();  
User u1 = new User("jean", "dupont");  
list.add(u1);  
list.add(u1); // deux références pointent vers le même objet
```

- Les collections**
- Les Listes**
- Les Sets**
- Les Maps**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

□ Les sets sont des collections

- Implémentent l'interface `java.util.Set` qui hérite de l'interface `Collection`

□ Les sets ne sont pas indexés



❑ HashSet

- Implémentation de base de l'interface Set
- Utilise la clé de hachage de ses éléments
- Set non ordonné

❑ TreeSet

- Implémente de 2 sous-interfaces de Set :
 - SortedSet : force les éléments du set à être triés.
 - NavigableSet : permet d'extraire des sous-ensembles.

□ Les éléments sont ajoutés avec la méthode add (...)

- Les éléments ne sont pas indexés.
- Tout type d'objet peut être inséré dans un set
- Un set peut être typé en utilisant les génériques

□ Méthode remove (...)

- Suppression de l'objet passé en paramètre

□ Un Set peut être parcouru comme une liste (Iterator ou boucle for each)

```
Set<String> set = new HashSet<String>();  
set.add("le");  
set.add("petit");  
set.add("chat");  
  
set.remove("petit");
```

□ Les Sets n'acceptent pas les doublons

- Si on ajoute un élément plusieurs fois, il ne sera présent qu'une seule fois dans le set

```
Set<String> set = new HashSet<String>();  
set.add("le");  
set.add("petit");  
set.add("chat");  
  
set.add("petit"); // sans effet
```

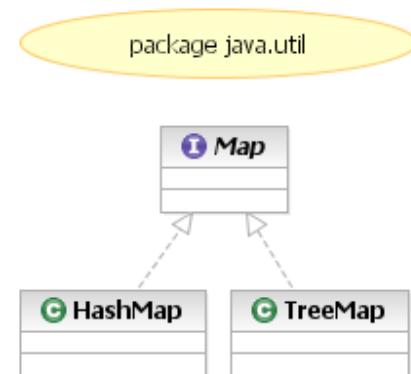


La méthode `add(...)` renvoie un booléen. Elle renvoie `false` si l'élément était déjà présent.

- Les collections**
- Les Listes**
- Les Sets**
- Les Maps**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

❑ Associations clé-valeur

- Aussi appelé dictionnaire
- Chaque élément stocké est associé à une clé
- Conceptuellement, une map est un ensemble d'éléments
- Pour des raisons techniques, l'interface `java.util.Map` n'hérite pas de `java.util.Collection`



Il existe également la classe `Hashtable` (généralement déconseillée).

❑ **HashMap**

- Implémentation de base de l'interface Map
- Utilise la clé de hachage de ses éléments
- Map non ordonnée

❑ **TreeMap**

- Implémente de sous interfaces de Map
 - SortedMap : force les éléments de la map à être triés selon les clés
 - NavigableMap : permet d'extraire des sous-ensembles.

Les éléments sont ajoutés avec la méthode put (...)

➤ `put (K key, V value)`

 Ils sont récupérés avec la méthode get (...)

➤ `V get (K key)`

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Map<String, User> map = new HashMap<String, User>();
// utilisation du nom comme clé d'association
map.put(u1.getNom(), u1);
map.put(u2.getNom(), u2);

// récupération en fonction de la clé
User ulbis = map.get("dupont");
```

❑ Méthode remove (...)

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Map<String, User> map = new HashMap<String, User>();
// utilisation du nom comme clé d'association
map.put(u1.getNom(), u1);
map.put(u2.getNom(), u2);

// suppression de l'association de la clé dupont (donc u1)
map.remove("dupont");
```



Attention à ne pas passer l'objet mais la clé en paramètre.

□ Problématique particulière

- S'agit-il de parcourir les clés ou les valeurs ?
- Méthode `Map.keySet()` : permet de récupérer un set contenant l'ensemble des clés.
- Méthode `Map.values()` : permet de récupérer une collection contenant l'ensemble des valeurs.
- Méthode `Map.entrySet()` : permet de récupérer une collection contenant l'ensemble des entrées

```
Map<String, User> map = new HashMap<String, User>();
Iterator<String> keys = map.keySet().iterator();
Iterator<User> values = map.values().iterator();

for (Map.Entry<String, User> entry : map.entrySet()) {
    String key = entry.getKey();
    User value = entry.getValue();
    System.out.println(key + " = " + value);
}
```

❑ Les maps acceptent les doublons

- Un élément peut être présent plusieurs fois s'il est référencé par des clés différentes
- Rajout d'un élément avec une clé déjà existante possible
 - La valeur précédente est écrasée

- Les collections**
- Les Listes**
- Les Sets**
- Les Map**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

❑ Considérons une classe ayant plusieurs attributs.

- Plusieurs objets de cette classe sont placés dans une collection
- Comment spécifier les critères de tri des éléments ?
 - Ex : lors d'un parcours de collection, on veut que les objets User arrivent triés par leur nom
 - S'ils ont le même nom, le deuxième critère est le prénom

User	
▫	nom : String
▫	identifiant : int
▫	prenom : String

□ Implémentation de l'interface `java.lang.Comparable`

```
public class User implements Comparable<User> {
    private String nom;
    private String prenom;
    public int compareTo(User otherUser) {
        // renvoie un nombre négatif si l'objet en cours a un nom
        // situé avant le nom de l'objet passé en paramètre (ordre
        // alphabétique)
        // renvoie un nombre positif si l'objet en cours a un nom
        // situé après le nom de l'objet passé en paramètre
        int resultat = nom.compareTo(otherUser.nom);
        if (resultat == 0) {
            resultat = prenom.compareTo(otherUser.prenom);
        }
        return resultat;
    }
}
```

□ Création d'un `java.util.Comparator`

```
public class UserComparator implements Comparator<User> {
    @Override
    public int compare(User u1, User u2) {
        int resultat = u1.getNom().compareTo(u2.getNom());
        if (resultat == 0) {
            resultat = u1.getPrenom().compareTo(u2.getPrenom());
        }
        return resultat;
    }
}
```



Avantage de cette solution : il est possible de créer plusieurs Comparators pour une même classe, et de choisir lequel appliquer selon les cas.

- Les collections**
- Les Listes**
- Les Sets**
- Les Map**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

□ java.util.Collections

➤ Méthodes utilitaires pour la manipulation des collections

```
List list = ...;

// tri de la liste (les éléments doivent implémenter l'interface Comparable)
Collections.sort(list);

// tri de la liste en fonction d'un Comparator
Collections.sort(list, new UserComparator());

// renvoie une collection non-modifiable
List l1 = Collections.unmodifiableList(list);

// liste synchronisée pour gérer les accès concurrents
List l2 = Collections.synchronizedList(list);
```

□ **java.util.Arrays**

➤ Méthodes utilitaires pour les tableaux

```
User[] usersArray = new User[3];  
  
// tri d'un tableau  
Arrays.sort(usersArray);  
  
// conversion d'un tableau en liste  
List<User> usersList = Arrays.asList(usersArray);
```

- Les collections**
- Les Listes**
- Les Sets**
- Les Map**
- Tri d'éléments**
- Classes utilitaires des collections**
- Bonnes pratiques**

□ Toujours utiliser une interface comme type déclaré d'une collection

- Dans la mesure du possible, rester générique
- Cela permet de changer l'implémentation sans impact sur le reste du code

```
HashSet<User> set = new HashSet<User>();  
Set<User> set = new HashSet<User>();  
  
ArrayList <User> list = new ArrayList<User>();  
List<User> list = new ArrayList<User>();  
  
HashMap<String, User> map = new HashMap<String, User>();  
Map<String, User> map = new HashMap<String, User>();
```

❑ Une pratique à éviter...

```
public static void mauvaisParcours() {  
    List list = ...;  
  
    for(int i = 0; i < list.size(); i++) {  
        list.get(i);  
    }  
}
```

❑ Il faut privilégier l'utilisation des Iterators et des boucles

- Permettent de rester générique
- Protègent à faible coût d'une éventuelle désynchronisation

❑ Ce code fonctionne-t-il ?

```
List<String> list;  
list.add("lundi");  
list.add("mardi");
```

□ A la dernière ligne, que contient chacun des ensembles ci-dessous ?

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Set<User> set = new HashSet<User>();
set.add(u1);
set.add(u1);
set.remove(u1);

List<User> list = new ArrayList<User>();
list.add(u1);
list.add(u1);

Map<String, User> map = new HashMap<String, User>();
map.put("dupont", u1);
map.put("durand", u2);
map.put("dupont", u1);
```

Réaliser les travaux pratiques suivants:

- TP 9

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- **Une entrée/sortie en Java consiste à échanger des données entre le programme et une autre source qui peut être:**
 - la mémoire
 - un fichier
 - ou le programme lui-même
- **Pour réaliser cela, Java emploie ce qu'on appelle un stream (qui signifie flux) entre la source et la destination des données.**
- **Un flux est une suite de valeurs (octets, caractères, objets quelconques) successivement lues ou écrites**

□ **Toute opération d'entrée/sortie en Java suit le schéma suivant:**

- Ouverture d'un flux
- Lecture ou écriture des données
- Fermeture du flux

□ Il existe 3 flux standards pour un système d'exploitation

- Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information standard
 - System.out
- Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information d'erreur
 - System.err
- Les octets circulant entre le clavier (C) et une application (A)
 - System.in

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

- **Les types utilisés pour les entrées / sorties sont définis dans le paquet `java.io`**
- **`java.io` fournit toutes les classes nécessaires à la création, lecture, écriture et traitement des flux**
- **Le package `java.io` contient plusieurs sortes de flux qui peuvent être classés suivant plusieurs critères**
 - Les flux d'entrée et les flux de sortie
 - Les flux de caractères et les flux binaires (dû au fait que java utilise l'unicode les caractères sont codés sur 2 octets et non sur un seul)
 - Les flux de communication et les flux de traitement
 - Les flux avec ou sans tampon

❑ Classe abstraite

- ❑ C'est la racine des classes qui concernent la lecture d'octets depuis un flot de données
- ❑ Type selon lequel sont vues toutes les classes de flux qui lisent des octets
- ❑ Elle possède un constructeur sans paramètre

❑ Ces flux sont des sous-classes de `InputStream` et peuvent être classés en deux catégories :

➤ Les flux de communication

- Servant essentiellement à créer une liaison entre le programme et une autre entité

➤ Les flux de traitement

- Comme leur nom l'indique, servent plutôt à traiter les données échangées

❑ Ils sont composés des classes suivantes:

- FileInputStream
 - Permet de créer un flux avec un fichier présent dans le système de fichiers
- ByteArrayInputStream
 - Permet de lire des données binaires à partir d'un tableau d'octets
- PipedInputStream
 - Permet de créer une sorte de tube d'entrée (pipe) dans lequel circuleront des octets

```
FileInputStream fis = null;
try {
    // Création d'un flux d'entrée ayant pour source un fichier nommé
    // source, cette instanciation peut lever une exception de type
    // FileNotFoundException
    fis = new FileInputStream("source");

    // La méthode read() renvoie un int représentant le nombre d'octets
    // lus, la valeur (-1) représente la fin du fichier, read peut lever
    // une exception du type IOException
    byte[] bytes = new byte[1024];
    int bytesRead;
    while ((bytesRead = fis.read(bytes)) != -1) {
        System.out.println(bytes);
    }
} catch (FileNotFoundException ef) {
    System.err.println("fichier introuvable");
} catch (IOException e) {
    System.err.println(e + "erreur lors de la lecture du fichier");
} finally {
    // Ne pas oublier de fermer le flux afin de libérer les ressources
    // qu'il utilise
    if (fis != null) {
        try { fis.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

- Ils viennent se greffer sur les flux de communication afin de réaliser un traitement sur les données lues
- Il se composent des classes suivantes qui étendent la classe `FilterInputStream`
 - `BufferedInputStream`
 - Cette classe permet la lecture de données à l'aide d'un tampon. Lorsque cette classe est instancié elle crée un tableau qui va servir du tampon
 - `DataInputStream`
 - Sert à lire des données représentant des types primitifs de Java (`int`, `boolean`, `double`, `byte`, ...) qui ont été préalablement écrits par un `DataOutputStream`
 - `SequenceInputStream`
 - Permet de concaténer deux ou plusieurs `InputStream`
 - `ObjectInputStream`
 - Permet de «déserialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un `ObjectOutputStream`

- A chaque flux d'entrée présenté précédemment, correspond un flux de sortie

- Les flux de sortie sont aussi classés en flux de communication et flux de traitement

- Interface publique de cette classe (ajouter throws IOException à toutes les méthodes)

```
abstract void write(int b)
void write(byte[] b)
void write(byte[] b, int début, int nb)
void flush()
void close()
```

❑ PipedOutputStream

- Permet la création d'un tube (pipe)

❑ FileOutputStream

- Permet l'écriture séquentielle de données dans un fichier

❑ ByteArrayOutputStream

- Permet d'écrire les données dans un tampon dont la taille s'adapte en fonction du besoin.
 - Les méthodes suivantes nous permet de récupérer les données écrites.
 - toByteArray()
 - toString()

□ **BufferedOutputStream**

- Permet d'écrire dans un tampon puis d'écrire le tampon en entier sur le fichier au lieu d'écrire octet par octet sur le fichier

□ **DataOutputStream**

- Cette classe permet l'écriture de données sous format Java et assure une portabilité inter-applications et inter-systèmes.

```
// création du flux
DataOutputStream out = new DataOutputStream(new
FileOutputStream("sortie.dat"));
out.writeInt(12);
out.writeBoolean(true);
out.writeChars("Hello !");
```

□ **ObjectOutputStream**

- Permet de sauvegarder l'état d'un objet dans un fichier ou autre.
 - L'objet doit implémenter l'interface `java.io.Serializable`
 - Seuls les membres non-transients et non statiques seront sauvegardés

```
BufferedInputStream in = null;
BufferedOutputStream out = null;
try {
    // création des flux
    in = new BufferedInputStream(new FileInputStream("source.dat"));
    out = new BufferedOutputStream(new FileOutputStream("copy.dat"));
    // copie du fichier
    byte[] buffer = new byte[1024];
    int bytesRead = -1;
    while ((bytesRead = in.read(buffer)) != -1) {
        out.write(buffer, 0, bytesRead);
    }
    // forcer l'écriture du contenu du tampon dans le fichier
    out.flush();
} catch (IOException e) {
    System.err.println(e);
} finally {
    // fermeture des flux
    if (in != null) {
        try { in.close(); } catch (IOException e) { e.printStackTrace(); }
    }
    if (out != null) {
        try { out.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

- **Ils transportent des données sous forme de caractères**
 - Java les gère avec le format Unicode qui code les caractères sur 2 octets
- **Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites Reader ou Writer**
- **Nombreuses sous-classes pour traiter les flux de caractères**

Classe abstraite qui est la classe mère de toutes les classes qui lisent des flux de caractères

Méthode	Rôle
boolean markSupported()	Indique si le flux supporte la possibilité de marquer des positions.
boolean ready()	Indique si le flux est prêt à être lu.
void close()	Ferme le flux et libère les ressources qui lui étaient associées.
int read()	Renvoie le caractère lu ou -1 si la fin du flux est atteinte.
int read(char[])	Lit plusieurs caractères et les met dans un tableau de caractères, retourne le nombre de caractères lus ou -1.
int read(char[], int, int)	Lit plusieurs caractères et les met dans un tableau de caractères à partir de l'index et pour la longueur fournie en paramètre, retourne le nombre de caractères lus ou -1.
long skip(long)	Saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères effectivement sautés.
void mark()	Permet de marquer une position dans le flux.
void reset()	Retourne dans le flux à la dernière position marquée.

- Lecture un fichier texte ligne par ligne
- Composition d'un BufferedReader avec un FileReader

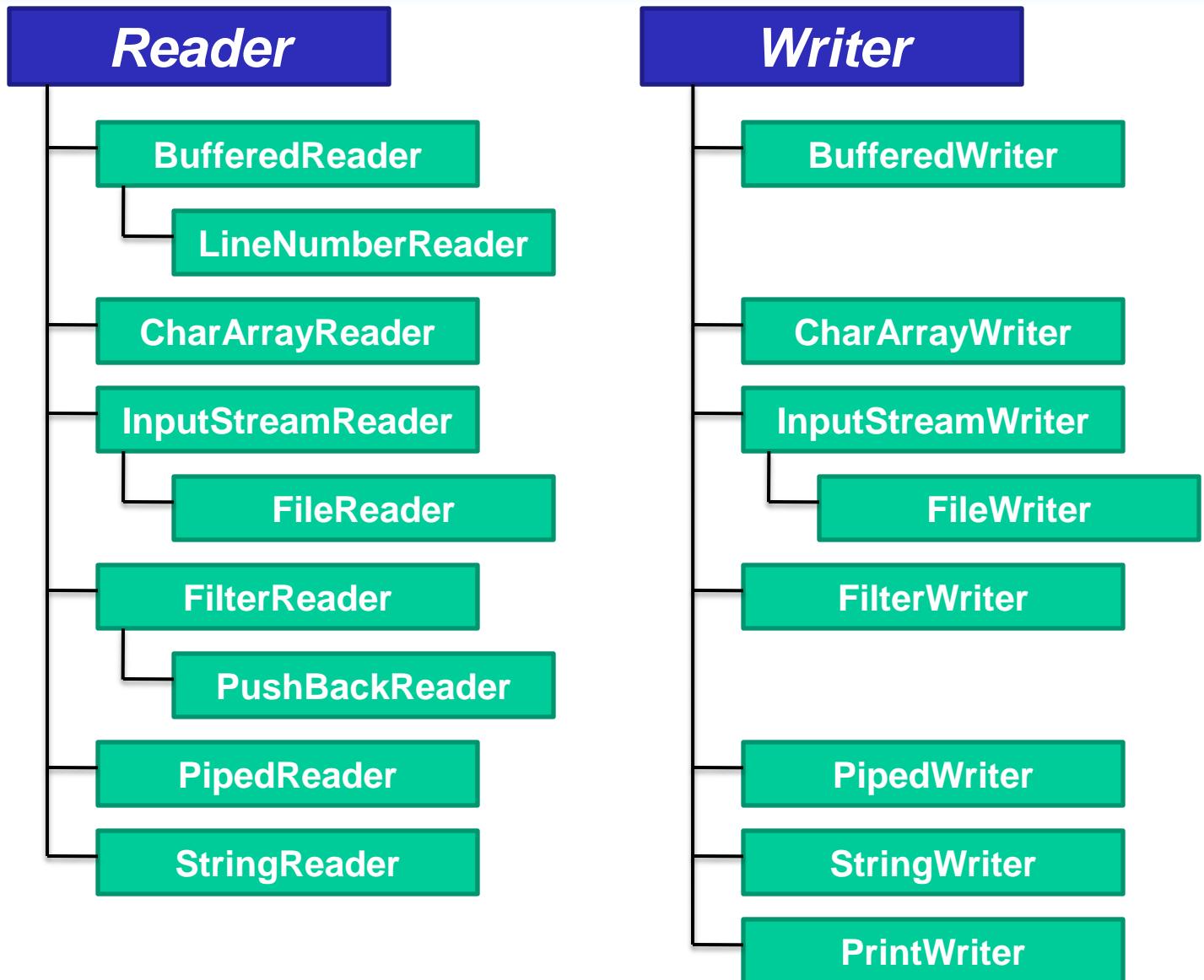
```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("source.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null) {
        System.out.println(ligne);
    }
} catch (IOException e) {
    System.err.println("erreur de lecture: " + e.getMessage());
} finally {
    if (reader != null) {
        try { reader.close(); } catch (IOException e) {
e.printStackTrace();
    }
}
}
```

□ **Classe abstraite qui est la classe mère de toutes les classes qui écrivent des flux de caractères**

Méthode	Rôle
void close()	Ferme le flux et libère les ressources qui lui étaient associées.
void write(int)	Ecrit le caractère dans le flux.
void write(char[])	Ecrit le tableau de caractères dans le flux.
void write(char[], int, int)	Ecrit le tableau de caractères dans le flux, n'utilise que les caractères entre l'index et pour la longueur fournis en paramètre.
void write(String)	Ecrit la chaîne de caractères dans le flux.
void write(String, int, int)	Ecrit une partie de la chaîne de caractères dans le flux.

- Ecriture dans un fichier texte
- Composition d'un BufferedWriter avec un FileWriter

```
BufferedWriter writer = null;  
try {  
    int nombre = 123;  
    writer = new BufferedWriter(new FileWriter("dest.txt"));  
    writer.write("Bonjour tout le monde");  
    writer.newLine();  
    writer.write("Nous sommes le " + new Date());  
    writer.write(", le nombre magique est " + nombre + ".");  
} catch (IOException e) {  
    System.err.println("erreur d'écriture: " + e.getMessage());  
} finally {  
    if (writer != null) {  
        try { writer.close(); } catch (IOException e) {  
e.printStackTrace(); }  
    }  
}
```



- **java.io.File est une représentation abstraite d'une ressource sur le système de fichier**
- **Représente soit le chemin d'accès à un dossier ou à un fichier**
- **La ressource représentée n'existe pas forcément !**
- **Ses méthodes principales pour la navigation sont :**
 - boolean exists()
 - boolean isDirectory()
 - boolean isFile()
 - String[] list()
- **Création avec le chemin sous la forme d'une String**

```
File fichier = new File("c:/source.dat");
if (fichier.exists()) {
    System.out.println(fichier.getAbsolutePath() + " existe");
} else {
    System.out.println(fichier.getAbsolutePath() + " n'existe pas");
}
```

□ Scanner permet de lire les entrées depuis la fenêtre de la console

- La lecture d'une entrée au clavier s'effectue en construisant un Scanner attaché sur le flux d'entrée standard System.in

```
Scanner clavier = new Scanner(System.in);
```

□ Les diverses méthodes de la classe Scanner permettent ensuite de lire les entrées

- Par exemple, la méthode nextLine() permet de lire une ligne de saisie complète

```
System.out.print("Veuillez entrer votre nom: ");
String nom = clavier.nextLine();
System.out.println("Bonjour " + nom + " !");
```

- **classe `java.io.Console` (introduite par Java 6)**
- **Permet de saisir un mot de passe sur la console sans qu'il ne s'affiche sur l'écran**

```
Console console = System.console();
if (console != null) {
    console.printf("Veuillez entrer votre nom: ");
    String name = console.readLine();
    console.printf("Veuillez entrer votre mot de passe:");
    char[] passwordChars = console.readPassword();
    String password = new String(passwordChars);
    console.printf("%s vous êtes connecté"
                   + " avec le mot de passe %s", name,
    password);
} else {
    System.err.println("Pas de console disponible");
}
```

❑ Les fichier de propriétés permettent de :

- Paramétriser une application
- Externaliser les ressources pour gérer l'internationalisation

❑ Une propriété est un couple { clé , valeur }

- Clé et valeur sont de type String
- Les clés identifient de manière unique les propriétés

❑ Classe `java.util.Properties`

- Hérite de la classe `Hashtable`
- Support du format XML
 - `storeToXML` : génère un fichier XML
 - `loadFromXML` : lit un fichier de propriétés au format XML

```
<properties>
    <entry key="filename">test.txt</entry>
    <entry key="line">2</entry>
</properties>
```

□ Communication par Socket

- Protocole TCP/IP
- 2 points de connexion : côté client et côté serveur

□ Classes du package java.net

➤ InetAddress

- Représente une adresse IP
- Encapsule l'accès au serveur de noms (DNS)

➤ Socket

- Utilisée par les clients TCP
- Crée un point de connexion
- Crée une connexion vers le socket serveur

➤ SocketServer

- Crée un point de communication sur un port particulier
- Se met en attente de connexions en provenance de clients

□ Exemple d'un serveur de temps

```
// Socket serveur en écoute sur le port 3000
ServerSocket server = new ServerSocket(3000);
while (true) {

    // Accepte la connexion entrante et envoie la date courante
    Socket serviceSocket = server.accept();
    DataOutputStream output = new DataOutputStream(serviceSocket.getOutputStream());
    output.writeLong(new Date().getTime());
    serviceSocket.close();
}
```

```
// Récupère la date courante auprès du serveur de temps
Socket clientSocket = new Socket("localhost", 3000);
DataInputStream input = new DataInputStream(clientSocket.getInputStream());
Date date = new Date(input.readLong());
```

□ Réaliser les travaux pratiques suivants:

- TP 10

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

- Comprendre les architectures liées aux bases de données
- Étudier différents modes d'accès
- Écrire des programmes permettant de consulter et de mettre à jour une base de données

- JDBC pour Java DataBase Connectivity
- API Java adaptée à la connexion avec les bases de données relationnelles et objet-relationnelles.
- L 'API Fournit un ensemble de classes et d 'interfaces permettant l'utilisation sur le réseau d'un ou plusieurs SGBD à partir d'un programme Java.

- Apporter un accès homogène au SGBD**
- Abstraction des SGBD cibles**
- Requêtes SQL**
- Simple à mettre en œuvre**
- Portabilité**

- Portabilité sur de nombreux système d'exploitation et sur de nombreuses SGBD (Oracle, DB2, Sybase, Microsoft, ..)
- Uniformité du langage de description des applications, des accès aux bases de données
- Liberté totale vis à vis des constructeurs
 - Le constructeur fournit juste le pilote
 - Le code est basé sur l'API JDBC

□ **L'API JDBC est disponible dans les package `java.sql` et `javax.sql`**

- Permet de formuler et gérer les requêtes aux bases de données relationnelles et objet-relationnelle
- Supporte les standard SQL-2 et SQL-3

□ **Il offre**

- Une connexion simultanée à plusieurs DB
- La gestion des transactions
- L'interrogation
- L'appel des procédures stockées

- JDBC interagit avec le SGBD par un pilote (driver)
- Il existe des drivers pour Oracle, MySQL , Sybase, DB2, Microsoft SQL Server, ...
- Tous les drivers
 - <http://www.codejava.net/java-se/jdbc/jdbc-driver-library-download>

□ Il a essentiellement 4 types de drivers JDBC :

➤ Type 1 : Pont JDBC-ODBC

- Traduit les appels de l'API JDBC en appels à l'API ODBC
- Est fourni par SUN avec le JDK 1.1
(sun.jdbc.odbc.JdbcOdbcDriver)
- A utiliser uniquement pour des tests et du prototypage

➤ Type 2 : API native

- Traduit les appels à l'API JDBC en appels natifs au client local de la base de données
- Livré par les éditeurs de SGBD et généralement payant (question de performance)
- Nécessite une installation de composants spécifiques

➤ Type 3 : Protocole réseau

- Traduit les appels à l'API JDBC en appels à un protocole indépendant du SGBD
- Nécessite un middleware (serveur d'application)
- Le plus utilisé pour les applications d'entreprise

➤ Type 4 : Protocole natif

- Traduit les appels à l'API JDBC en appels à un protocole natif lié au SGBD
- Souvent fourni par l'éditeur

- Importer le package `java.sql`
- Enregistrer le driver JDBC
- Etablir la connexion à la base de données
- Créer une zone de description de requête
- Exécuter la requête
- Traiter les données retournées
- Fermer les différents espaces

- **Méthode `forName()` de la classe `Class` :**

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- **Quand une classe Driver est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du `DriverManager`**
- **Certains compilateurs refusent cette notation et demandent plutôt**

```
Class.forName("driver_name").newInstance();
```

❑ Accès à la base via un URL de la forme :

- jdbc:<sous-protocole>:<nomBD>;param=valeur, ...

❑ L'URL spécifie

- L'utilisation de JDBC
- Le driver ou le type de SGBDR
- L'identification de la base locale ou distante
 - avec des paramètres de configuration éventuels
 - nom utilisateur, mot de passe, ...

❑ Exemple

```
String url = "jdbc:mysql://10.233.97.5:3306/neobject_db";
```

□ La méthode `getConnexion()` de `DriverManager` qui accepte de 1 à 3 arguments :

- L'URL de la base de données
- Le nom de l'utilisateur de la base
- Le mot de passe

```
Connection connection = DriverManager.getConnection(url, user, password);
```

□ Le `DriverManager` va essayer tous les drivers qui se sont enregistrés jusqu'à ce qu'il trouve un driver qui peut se connecter à la base demandée

- La classe Statement possède les méthodes nécessaires pour réaliser les requêtes sur la base de données associée à la connexion dont elle dépend
- 3 types de Statement :
 - Statement : requêtes statiques simples
 - PreparedStatement : requêtes dynamiques pré-compilées (avec paramètres d'entrée/sortie)
 - CallableStatement : procédures stockées

- L'objet Connection offre des méthodes pour créer des objets Statement

```
Statement statement1 =  
    connection.createStatement(query1);
```

```
PreparedStatement statement2 =  
    connection.prepareStatement(query2);
```

```
CallableStatement statement3 =  
    connection.prepareCall(query3);
```

❑ 3 types d'exécution de requête

❑ executeQuery ()

- Pour les requêtes (SELECT) qui retournent un ResultSet (tuples résultants)

❑ executeUpdate ()

- Pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, ...) qui retournent un entier (nombre de tuples traités)

❑ execute ()

- Pour requêtes diverses
- Renvoie true si la requête a donné lieu à la création d'un objet ResultSet

- ❑ **executeQuery () et executeUpdate () acceptent comme argument une chaîne de caractère (String) indiquant la requête SQL à exécuter**

```
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM user");
int nb = st.executeUpdate("INSERT INTO user(email) "
                          + "VALUES ('admin@mywebsite.com')");
```

- Les instances de l'interface `ResultSet` contiennent les résultats d'une requête SQL. Ils contiennent les tuples (lignes) satisfaisant les conditions de la requête.

- La structure des `ResultSet` est très semblable à celle d'une table dans une base de données relationnelle

- On pourra parcourir le ResultSet ligne par ligne en utilisant la méthode next ()
 - Retourne `false` si c'est le dernier tuple lu, sinon `true`
 - Chaque appel fait avancer le curseur sur le tuple suivant
 - Initialement, le curseur est positionné avant le premier tuple
 - Exécuter `next ()` au moins une fois pour avoir le premier

```
ResultSet rs = st.executeQuery("SELECT * FROM user");  
while (rs.next()) {  
    // traitement de chaque tuple  
}
```

- Possibilité de revenir au tuple précédent ou d'accéder à un tuple à partir de son index.

- Les colonnes sont référencées par leur numéros ou par leur noms
- L'accès aux valeurs des colonnes se fait par les méthodes de la forme `getXXX()`
 - Soit en utilisant le numéro de colonne (déconseillé !)
 - Soit en utilisant le nom de la colonne

□ Exemple

```
// accès à la valeur de la 1ere colonne du tuple courant
int val = rs.getInt(1);

// accès à la valeur de la colonne nommée ID du tuple
courant

String id = rs.getString("ID") ;
```

ID	Nom	Prénom	email
1	Toto	Titi	toto.titi@mysite.com
2	Guibert	Fabien	fguibert@gmail.com

```
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery("SELECT id, nom, email, FROM user");
while(rs.next()) {
    long id = rs.getLong("id");
    String nom = rs.getString("nom");
    String email = rs.getString("email");
}
```

□ **Le driver JDBC traduit le type JDBC retourné par le SGBD en un type Java correspondant**

- Le `Xxx` de `getXxx()` est le nom du type Java correspondant au type JDBC attendu
- Chaque driver a des correspondances entre les types SQL du SGBD et les types JDBC
- Le programmeur est responsable du choix de ces méthodes
 - `SQLException` générée si mauvais choix

Type JDBC	Type Java	Méthode
INT	int	getInt()
REAL	float	getFloat()
FLOAT	double	getDouble()
DOUBLE	double	getDouble()
CHAR	String	getString()
VARCHAR	String	getString()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()

□ Pour repérer les valeurs NULL dans une colonne d'une table de la base de données

- Utiliser la méthode `wasNull()` de `ResultSet`
 - Renvoie `true` si l'on vient de lire un `NULL`, sinon `false`
- Les méthodes `getXXX()` de `ResultSet` convertissent une valeur `NULL` SQL en une valeur acceptable par le type d'objet demandé
 - Les méthodes retournant un objet (`getString()`, `getObject()`,...) retournent `null` au sens Java
 - Les méthodes numériques (`getByte()`, `getInt()`,...) retournent 0
 - La méthode `getBoolean()` retourne `false`

- Pour terminer proprement un traitement, il faut fermer les différentes ressources ouvertes
- Chaque objet possède une méthode `close()`

```
resultSet.close();
statement.close();
connection.close();
```

- **SQLException est levée dès qu'une connexion ou un ordre SQL ne se passe pas correctement**
 - La méthode `getMessage()` donne le message d'erreur en clair
 - Renvoie aussi des informations spécifiques au gestionnaire de la base comme :
 - `SQLState`
 - Code d'erreur fabricant

- **SQLWarning : avertissements SQL**

- Pour récupérer des informations sur la base de données elle-même, on utilise la méthode `getMetaData()` de l'objet `Connection`
 - Elle renvoie un objet de type `DataBaseMetaData`

- On peut connaître entre autres :
 - Les tables de la base : `getTables()`
 - Le nom de l'utilisateur : `getUserName()`
 - ...
- La complétude des méta-données dépend du SGBD avec lequel on travaille

□ La méthode `getMetaData()` permet d'obtenir des informations sur les types de données du ResultSet

- Elle retourne un objet de type `ResultSetMetaData`
- On peut connaître entre autres :
 - Le nombre de colonnes : `getColumnCount()`
 - Le nom d'une colonne : `getColumnName(int col)`
 - Le type d'une colonne : `getColumnType(int col)`
 - Le nom de la table : `getTableName(int col)`
 - Si un NULL SQL peut être stocké dans une colonne :
`isNullable()`

□ Réaliser les travaux pratiques suivants:

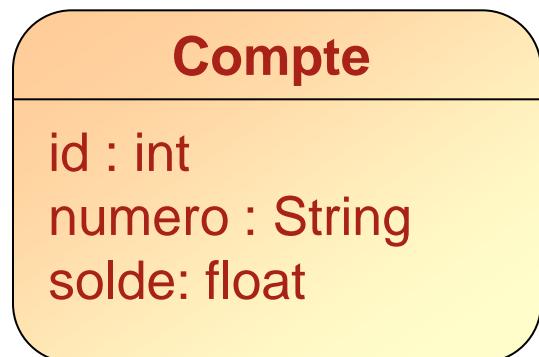
- TP 11

- 1. Présentation de Java**
- 2. Java en ligne de commande**
- 3. Présentation d'Eclipse**
- 4. Rappels UML**
- 5. Les bases du langage**
- 6. Les concepts objet avec Java**
- 7. Les exceptions**
- 8. Les classes de base**
- 9. Les collections**
- 10.Les entrées / sorties (IO)**
- 11.L'accès aux bases de données**
- 12.Le mapping objet-relationnel**

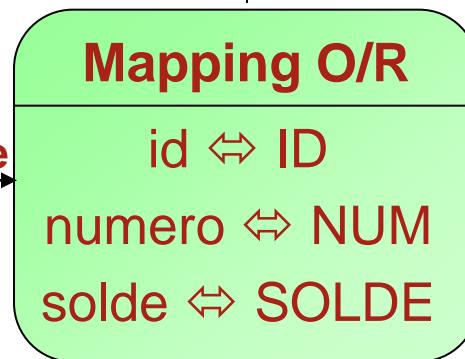
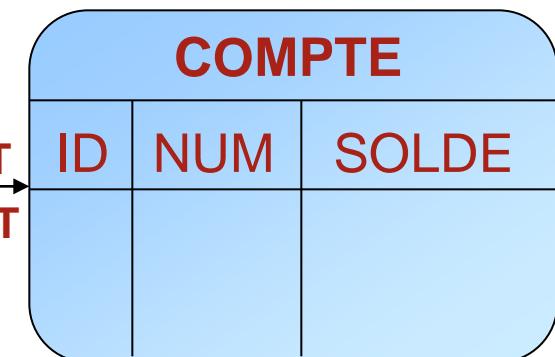
- **L'utilisation de bases de données objet n'est pas encore un choix pour les entreprises**
 - données existantes et vitales
 - mauvaise image des SGBD objet
 - il existe des solutions « hybrides » : Oracle 10g par exemple

- **Comment rendre persistants des objets dans une base de données relationnelle ?**
 - adopter une stratégie pour choisir entre plusieurs solutions de mise en correspondance objet-relationnel
 - effectuer des compromis entre pureté d'un modèle et performance

- **Objet = données (attributs) + comportement (méthodes)**
- **Enregistrement = données**
- **Les relations entre objets sont de différentes natures**
 - composition
 - Héritage
- **Un modèle objet est plus informatif qu'un modèle relationnel**
 - le passage « objet vers relationnel » nécessite de faire un choix parmi plusieurs solutions
 - le passage « relationnel vers objet » n'est pas souvent pertinent car
 - le modèle objet résultant est très pauvre et dénaturé
 - les objets identifiés n'ont pas de comportement

Monde Objet

create
get

**Monde Relationnel**

INSERT
SELECT

- **L'identité d'un objet est implicite**
 - deux objets ayant les mêmes attributs (i.e. le même état) sont tout de même différents
- **L'identité d'un enregistrement est représentée par sa clé primaire**
- **Comment maintenir l'identité d'un objet ?**

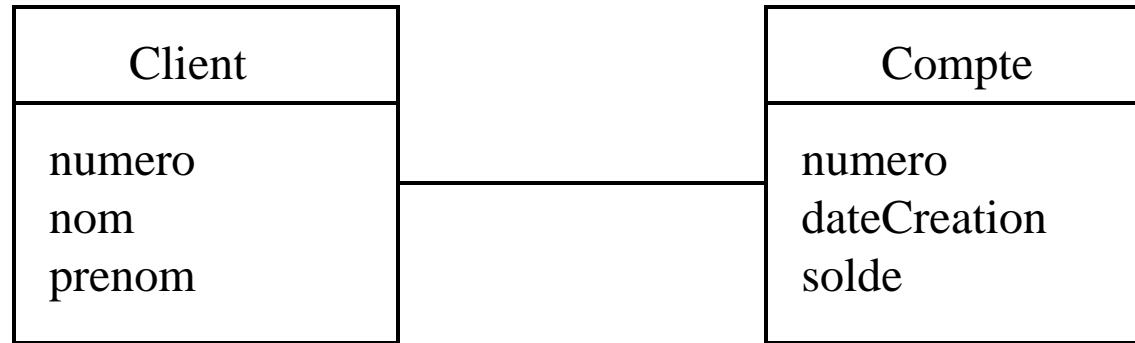
□ Dans la table

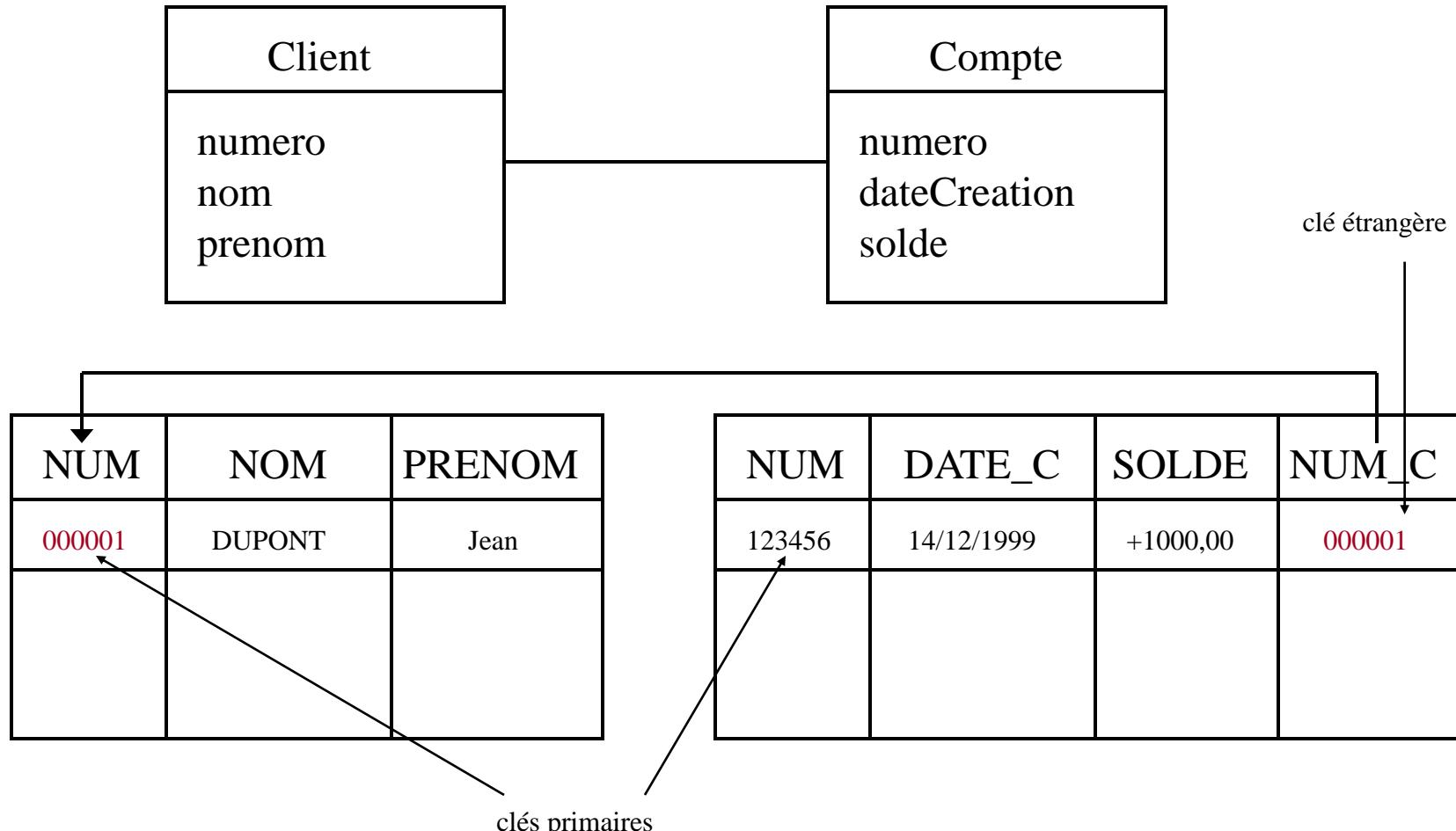
- définir une colonne comme clé primaire (ce qui est usuel)

□ Dans la classe

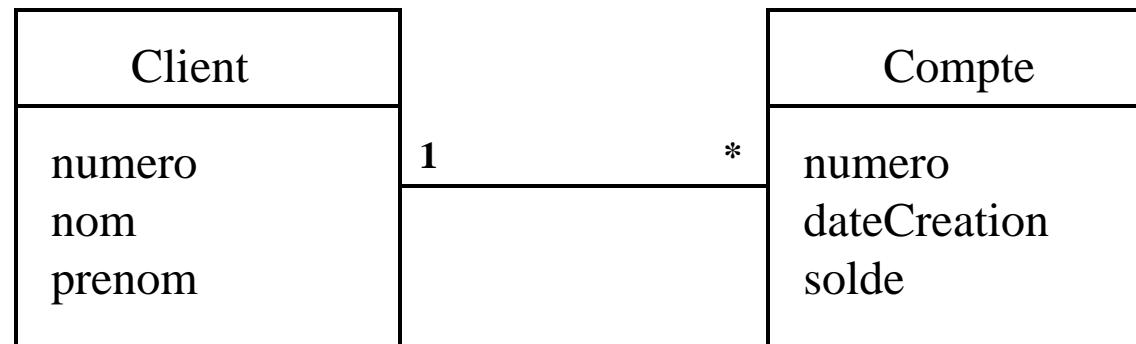
- définir un attribut qui identifie de façon unique un objet
- assurer que le processus de création d'un nouvel objet ne génère pas de doublon de cet attribut

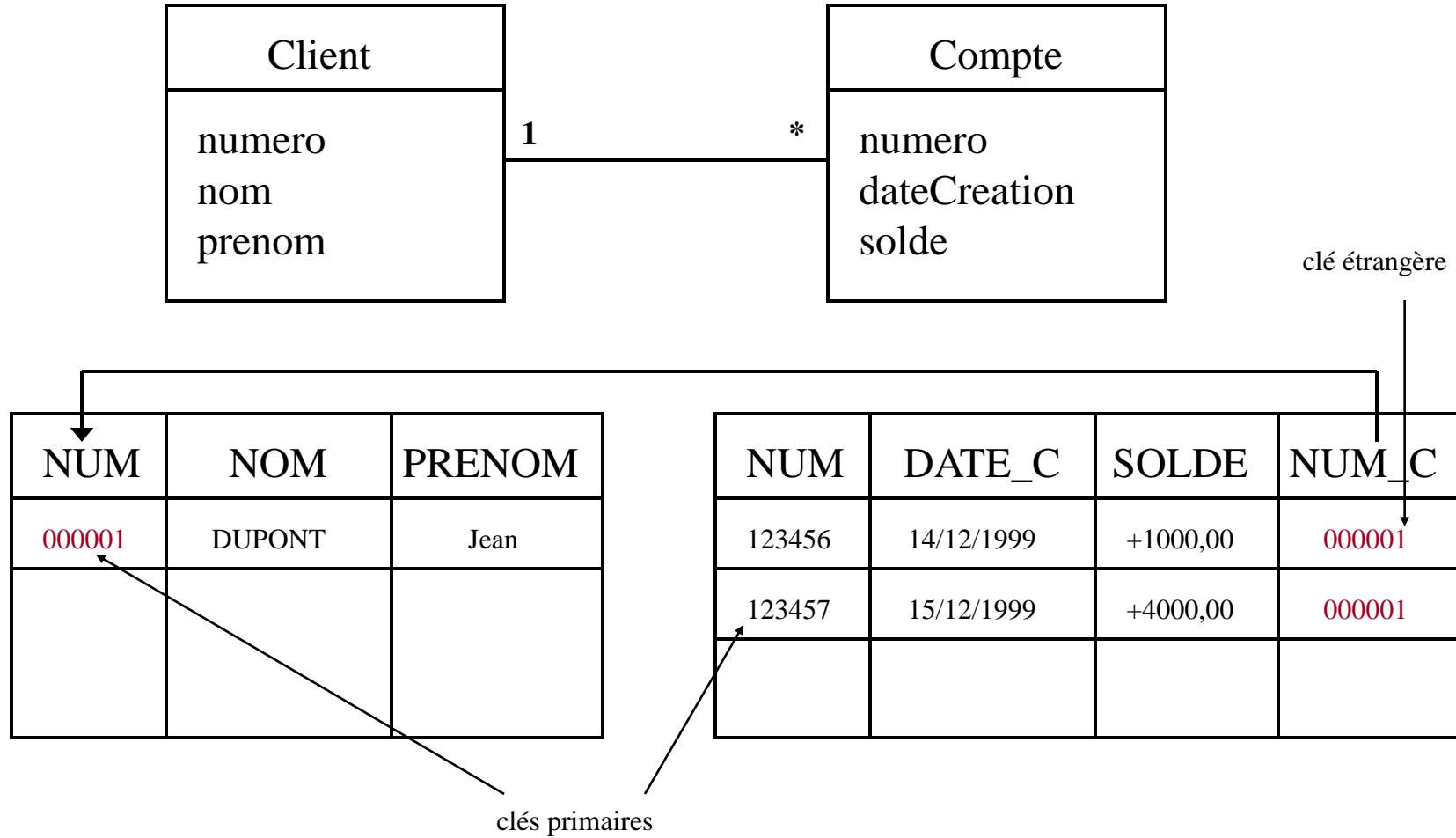
- On modélise une association 1-1 par l'ajout d'une clé étrangère à l'une des tables
- Si l'association est dirigée, le choix de la clé étrangère est sans équivoque
- Sinon, choisir la plus pertinente pour les futures requêtes



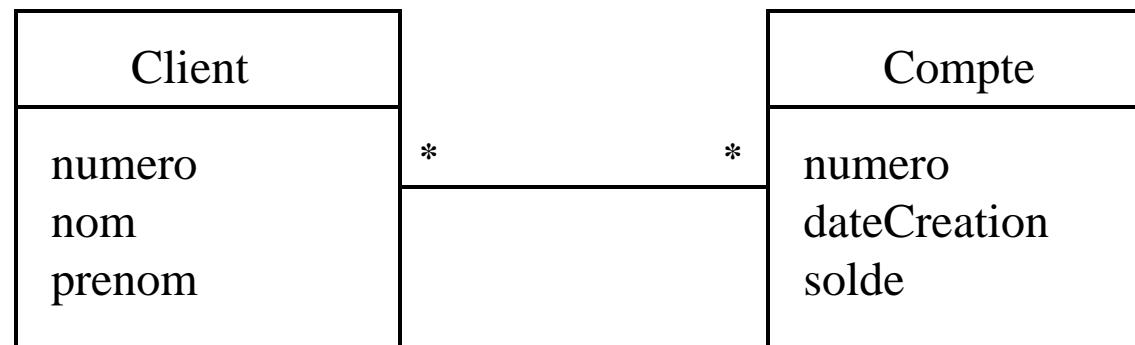


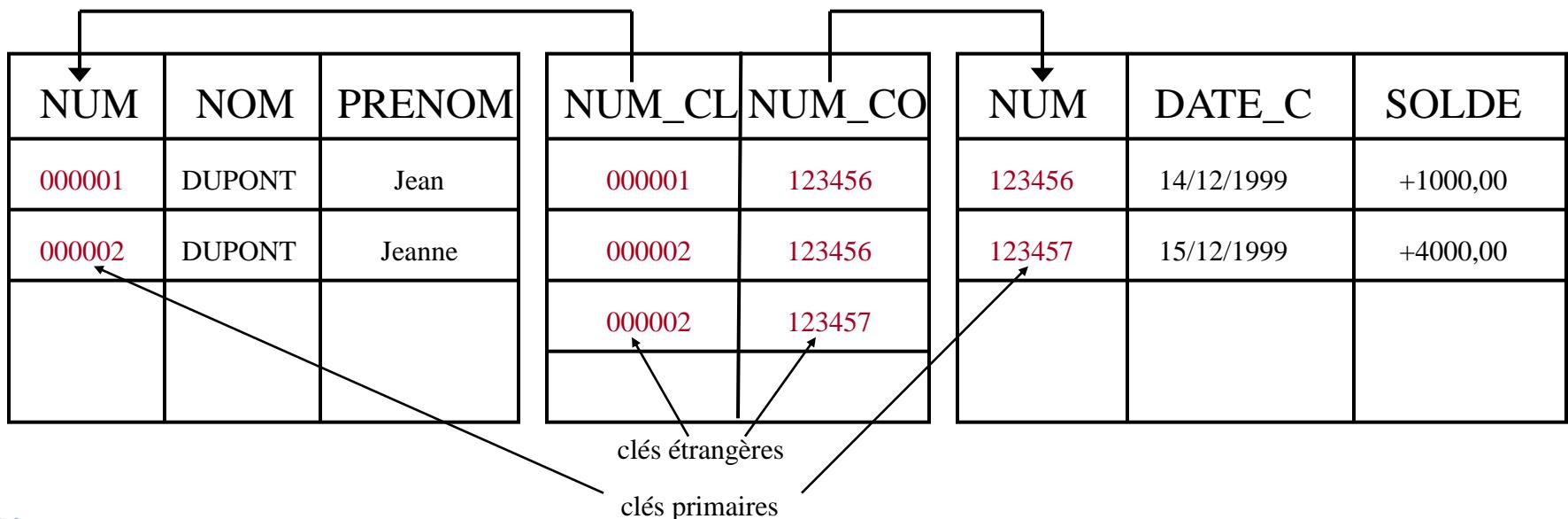
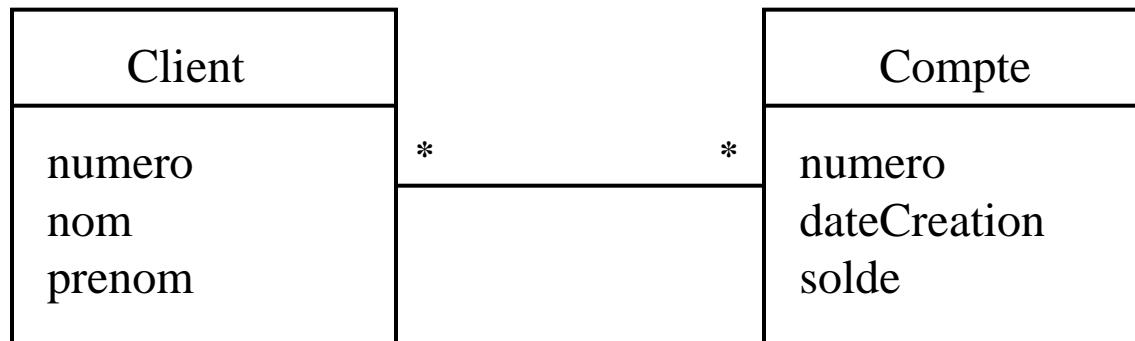
- On modélise une association 1-n en ajoutant une clé étrangère à la table représentant la multiplicité de l'association



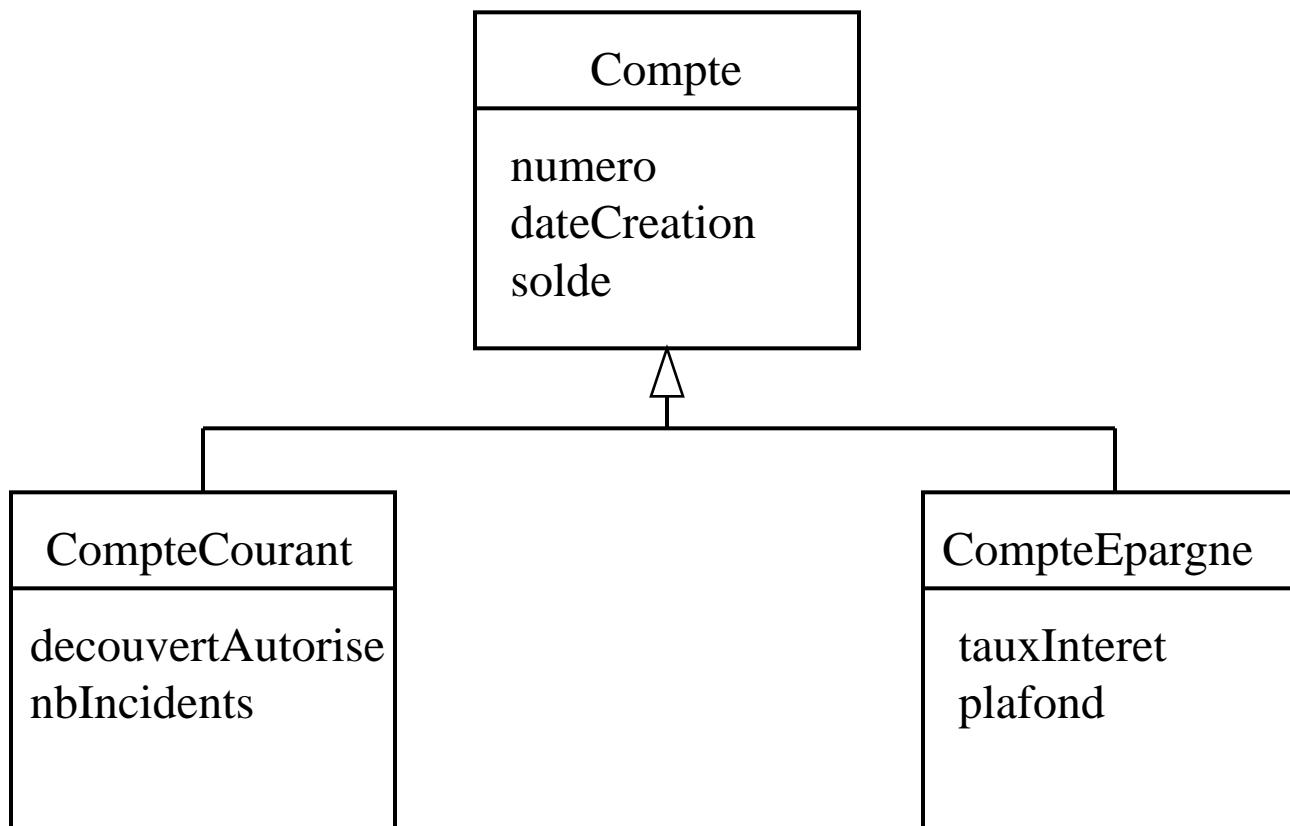


- On modélise une association n-n par une table d'association.
- Elle est construite selon les principes des associations 1-n



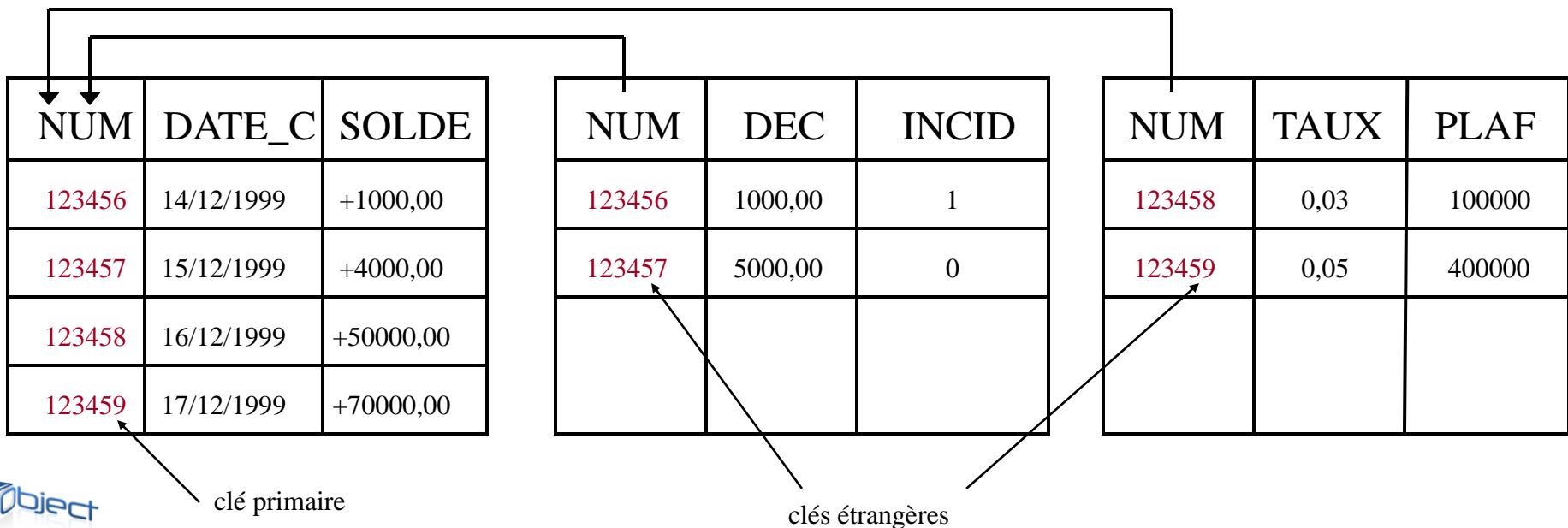


□ Il existe plusieurs solutions pour le modéliser ...



□ 1 table par classe

- C'est la solution théorique la plus pure, mais sûrement pas la plus performante...
- L'héritage est modélisé par les clés étrangères



□ 1 table par sous-classe concrète

- C'est la solution le plus satisfaisante en terme de performances, mais...
 - attention à l'unicité globale des clés primaires sur toutes les tables
 - duplication de colonnes
- La super-classe est généralement abstraite

NUM	DATE_C	SOLDE	DEC	INCID
123456	14/12/1999	+1000,00	1000,00	1
123457	15/12/1999	+4000,00	5000,00	0

NUM	DATE_C	SOLDE	TAUX	PLAF
123458	16/12/1999	+50000,00	0,03	100000
123459	17/12/1999	+70000,00	0,05	400000

clés primaires

□ 1 seule table par hiérarchie de classes

- Une table unique avec autant de colonnes que d'attributs définis dans chacune des classes composant la hiérarchie
- C'est conceptuellement une solution faible
- Beaucoup de problèmes pour les mises à jour
- Perte d'espace très importante dans la table
 - colonnes non renseignées en fonction de la sous-classe de l'objet correspondant à un enregistrement donné)

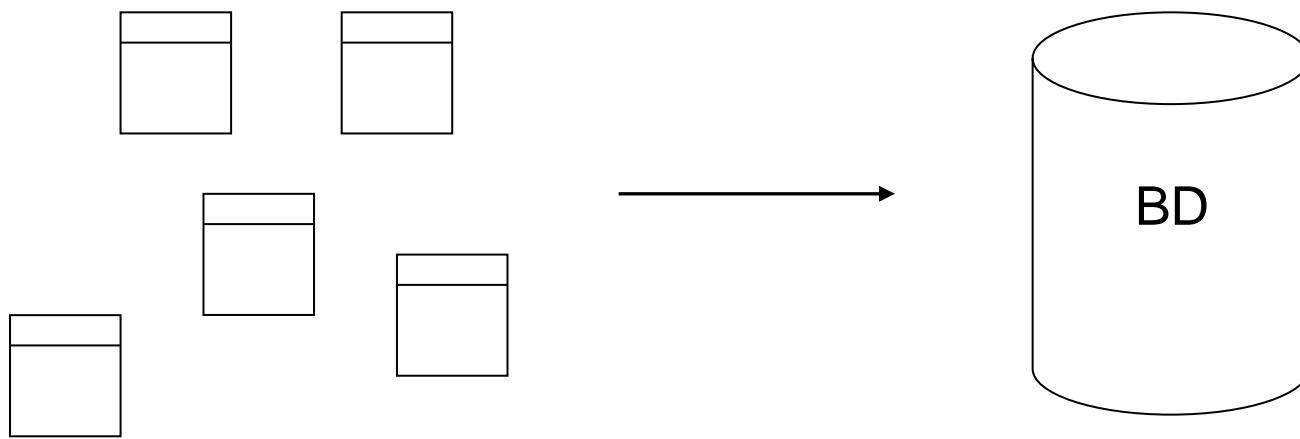
NUM	DATE_C	SOLDE	DEC	INCID	TAUX	PLAF
123456	14/12/1999	+1000,00	1000,00	1	NULL	NULL
123457	15/12/1999	+4000,00	5000,00	0	NULL	NULL
123458	16/12/1999	+50000,00	NULL	NULL	0,03	100000
123459	17/12/1999	+70000,00	NULL	NULL	0,05	400000

clé primaire

□ Les différentes architectures

- l'accès direct
- les Beans d'accès aux données
- utilisation de classes dédiées
- les Enterprise JavaBeans
- utilisation d'un framework de persistance

- Solution la plus simple**
- Peut se faire depuis une applet, via JDBC**
- Uniquement pour des applications très simples**
- Couplage fort avec la base de données**
 - non extensible
 - non réutilisable, maintenance difficile
 - ne supporte pas un modèle complexe



Classes métier

❑ Applications simples

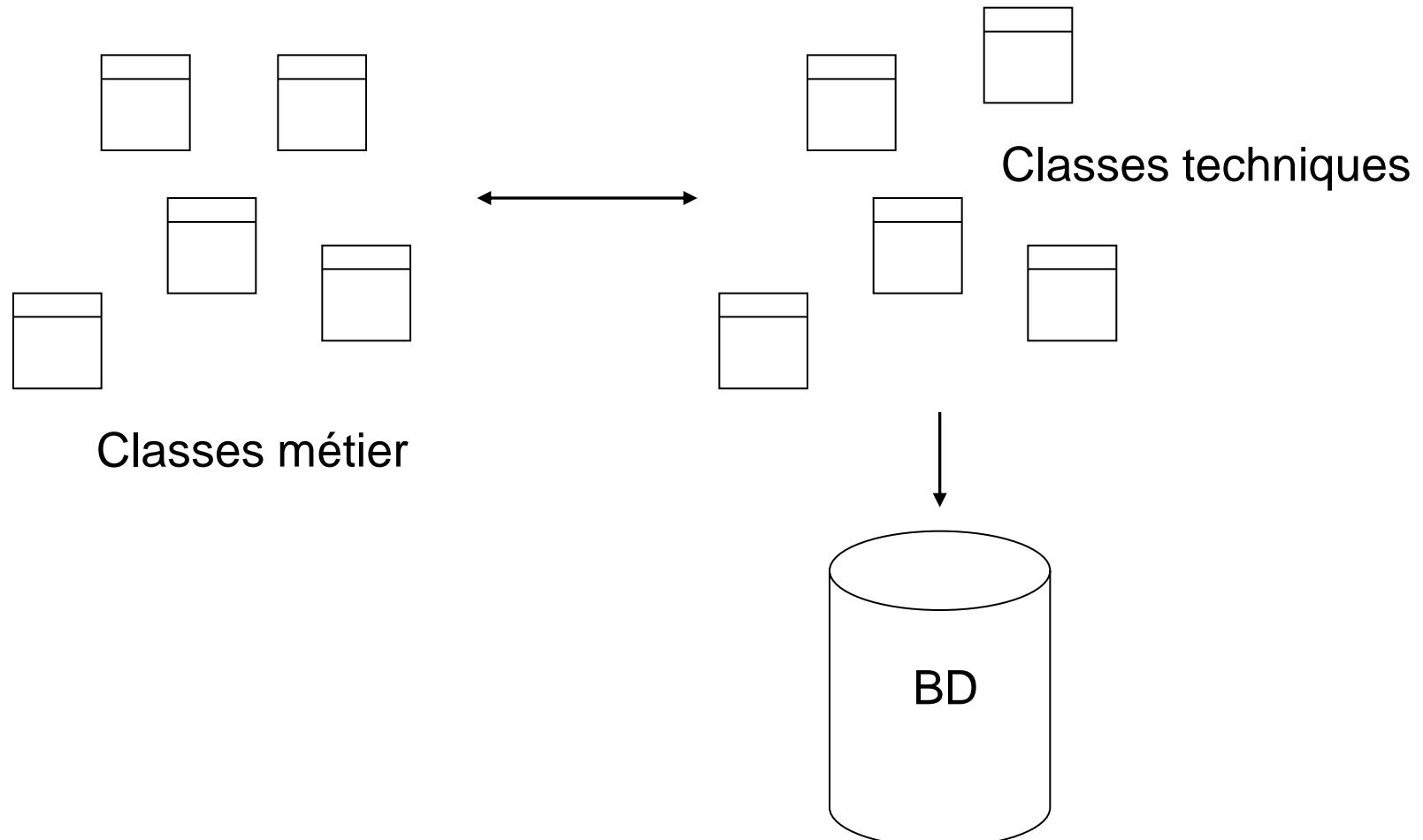
- courte durée de vie
- requêtes simples
- relations entre objets peu complexes

❑ Peu extensible

❑ Inadapté aux modèles complexes

- **Création de classes dédiées qui gèrent la persistance d'une classe d'objets données**
 - se chargent des relations entre objets, des requêtes

- **Séparation de la logique métier et de la logique d'accès aux données**



- **Encapsulent les données métier et la logique associée dans un composant**

- **Persistance à l'aide d'une base de données**
 - Container-Managed Persistence : solution de persistance simple et automatique (gérée par le container d'EJB)
 - Bean-Managed Persistence : persistance gérée par le Bean, donc mise en œuvre par le développeur

- **Solution la plus aboutie, mais la plus compliquée à mettre en œuvre**
- **Permet au développeur de ne pas connaître la structure de la base**
- **Prend en charge mappage, associations et héritages complexes**
- **Quelques exemples :**
 - Dans le standard JEE : EJB3 et JPA (Java Persistence API)
 - Framework de mapping O/R: très utilisés
 - Open-source: Hibernate
 - Commerciaux: Oracle TopLink, Kodo...
 - JDO : Java Data Objects: modèle alternatif qui n'a jamais percé

Fin Partie 1 - Java