

Formation Java

Table des matières

- **Introduction**
- **Définition de classes Java**
- **Les bases du langage**
- **Polymorphisme et héritage**
- **Classes utilitaires et Collection**
- **Gestion des exceptions**
- **Package IO**
- **Accès aux données**
- **Gestion des logs avec Log4j**
- **Tests unitaires avec JUnit**

Introduction

SOMMAIRE

- Qu'est-ce que Java ?
- Principales propriétés de Java
- Autres propriétés importantes
- Programmation procédurale: Inconvénients
- Qu'est-ce qu'un objet ?
- LES RÈGLES D'ENCAPSULATION
- LES OBJETS COMMUNIQUENT
- LES CLASSES ET LES OBJETS
- Historique de Java
- Spécifications de Java
- Plate-forme Java
- Votre environnement de développement
- Compilation en Java → bytecode
- Exécution du bytecode

Introduction

Qu'est-ce que Java ?

- Java est un langage de programmation mis au point par Sun Microsystems dans les années 90.
- La première version sort en 1995.

Introduction

Principales propriétés de Java

- **Langage orienté objet**, à classes (les objets sont décrits/regroupés dans des classes)
- de syntaxe proche du langage C
- fourni un **JDK** (*Java Development Kit*) :
 - outils de développement
 - ensemble de paquetages très riches et très variés
- **portable** grâce à l'exécution par une machine virtuelle : « *Write once, run everywhere* »

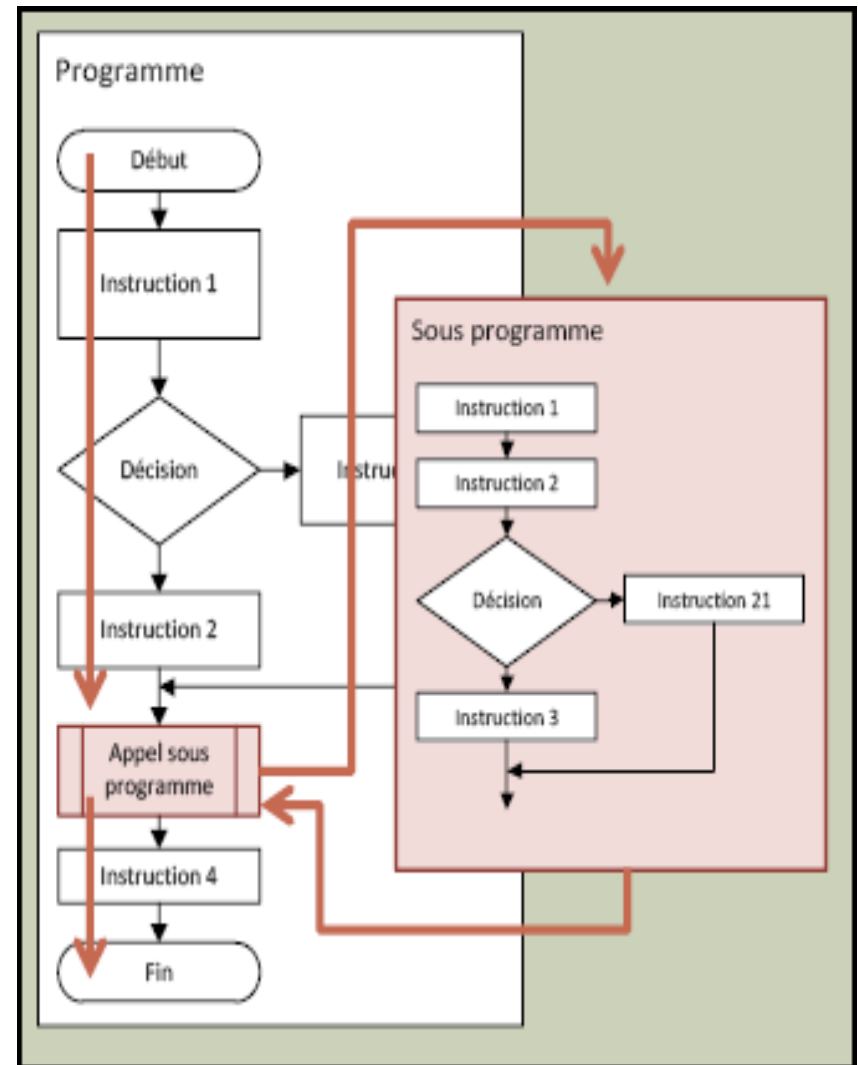
Introduction

Autres propriétés importantes

- **multi-tâches** (*thread*)
- **sûr**
 - fortement typé
 - nombreuses vérifications au chargement des classes et durant leur exécution
- **adapté à Internet**
 - chargement de classes en cours d'exécution (le plus souvent par le réseau : *applet ou RMI*)
 - facilités pour distribuer les traitements entre plusieurs machines (sockets, RMI, Corba, EJB...)

Introduction

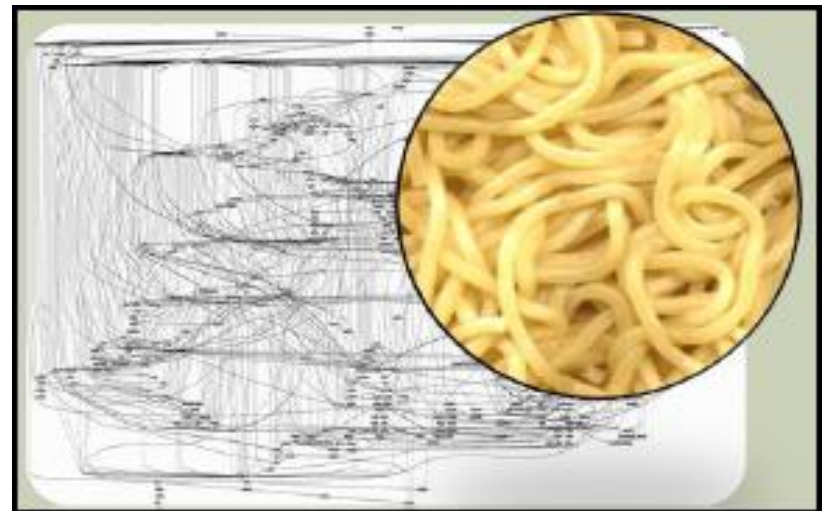
- **La programmation procédurale**
 - Une suite d'instructions s'exécutant les unes après les autres
 - Avec des procédures ou des fonctions (sous-programmes)
- Cette approche a permis de décomposer les fonctionnalités d'un programme en procédures qui s'exécutent séquentiellement



Introduction

Programmation procédurale: Inconvénients

- **le développeur doit penser de manière algorithmique**, approche proche du langage de la machine , le développeur doit faire un **effort supplémentaire pour structurer le programme**
- **Le procédural est très éloigné de notre manière de penser**
- **Le code est peu lisible** => Et **difficile à modifier, à maintenir**
- **L'ajout de fonctionnalités difficile** => Réutilisation du code incertain
 - Attention au « Copier/Coller »
- **Le travail d'équipe est délicat**



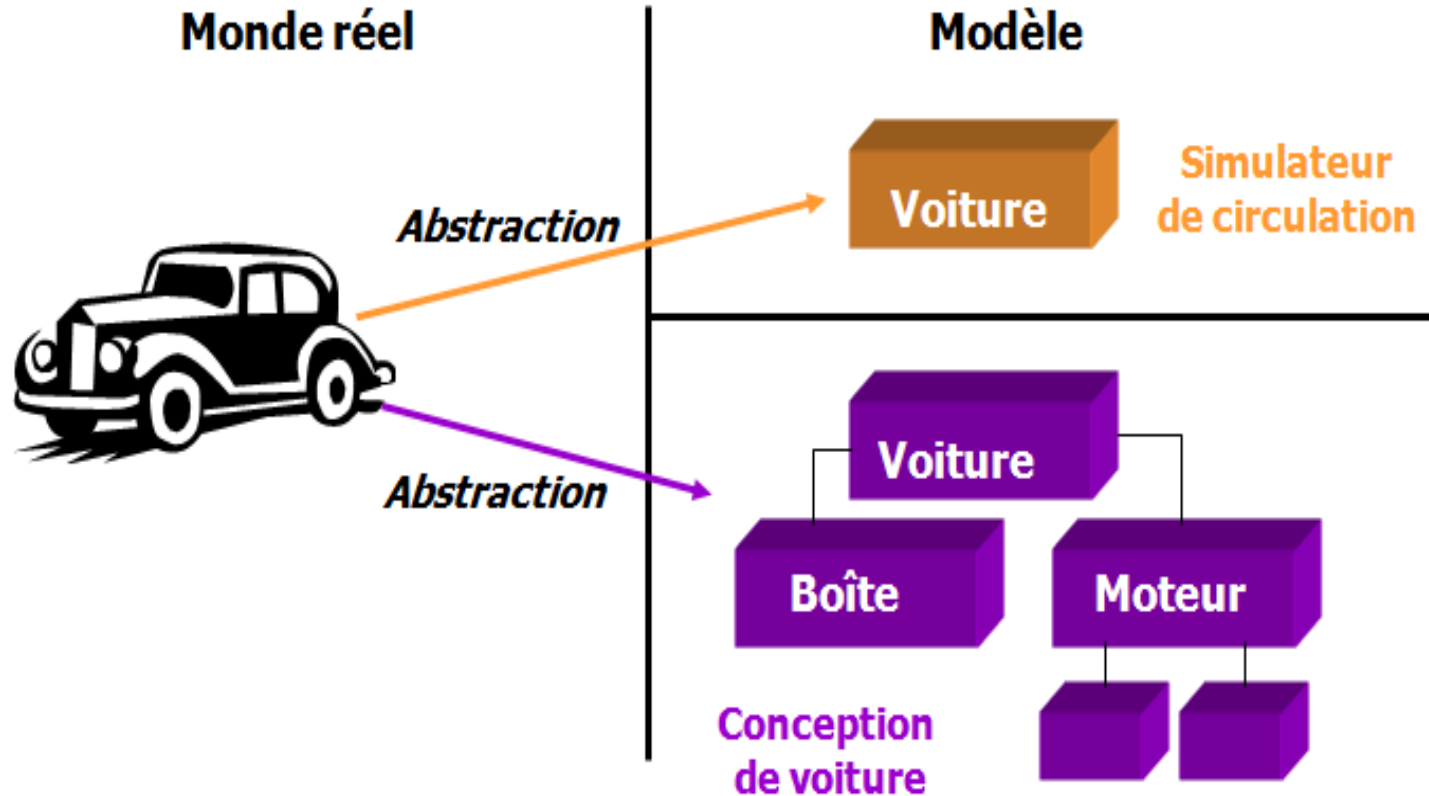
Introduction

- Finalement, il est peut-être plus simple de **s'inspirer du monde réel**
 - **Le monde réel est composé d'objets, d'êtres vivants, de matière**
 - Pourquoi ne pas programmer de manière plus réaliste ?
- **Les objets ont des propriétés**
 - Un chat à 4 pattes, un serpent aucune
- **Les objets ont une utilité, une ou plusieurs fonctions**
 - Une voiture permet de se déplacer
- Alors, plutôt que de focaliser sur les procédures, intéressons-nous d'avantage aux données
- De cette analyse est née **la programmation orientée objet**

Introduction

Qu'est-ce qu'un objet ? (1/3)

- C'est une abstraction d'une entité du monde réel



Introduction

Qu'est-ce qu'un objet ? (2/3)

- **Un élément qui modélise toute entité, concrète ou abstraite, manipulée par le logiciel**
 - Exemple : Une voiture avec 4 roues, un volant, un moteur,
- **Un élément qui réagit à certains messages qu'on lui envoie de l'extérieur**
 - C'est son comportement, ce qu'il sait faire
 - Exemple : Avec cette voiture, je peux tourner, accélérer, freiner, ...
- **Un élément qui ne réagit pas toujours de la même manière**
 - Son comportement dépend de l'état dans lequel il se trouve
 - Exemple : Essayez de démarrer sans essence !!?

Introduction

Qu'est-ce qu'un objet ? (3/3)

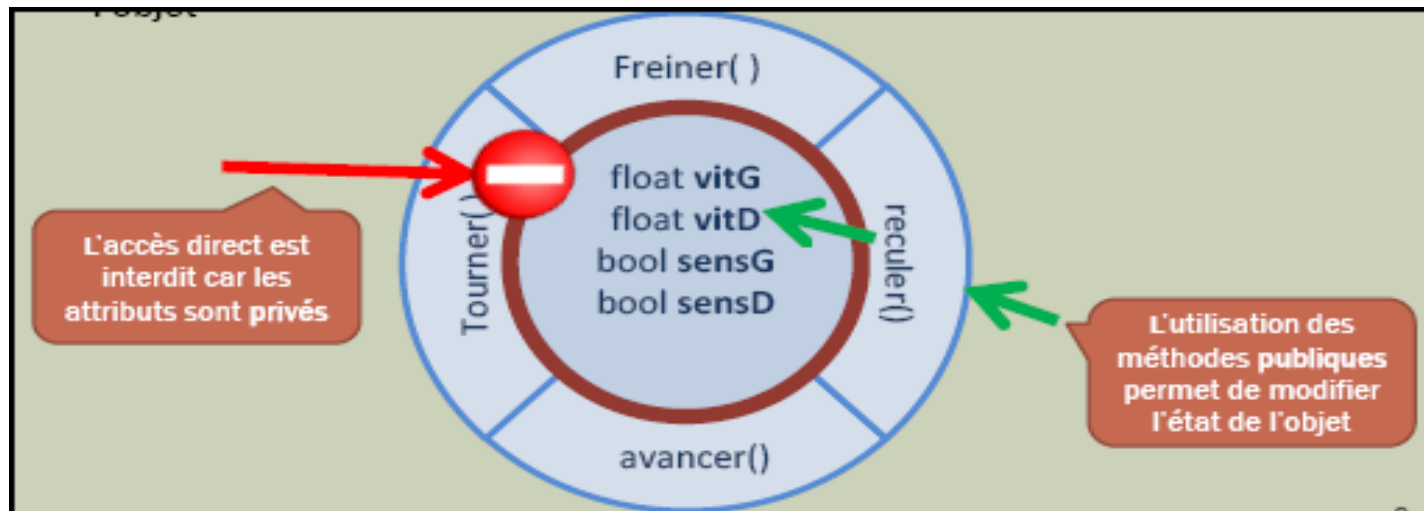
- **Une identité unique** (Qui permet de le distinguer d'un autre)
 - Ex: La **plaque d'immatriculation** pour 2 voitures de même modèle
- **Un état interne** Donné par des valeurs de variables internes appelées attributs ou membres
 - Exemple : **float vitesse; int nb_voyageurs;**
- **Un comportement** (Les méthodes ou opérations)
 - Exemple : La méthode « **freiner()** » de l'objet « **voiture** »

Introduction

LES RÈGLES D'ENCAPSULATION

Principe

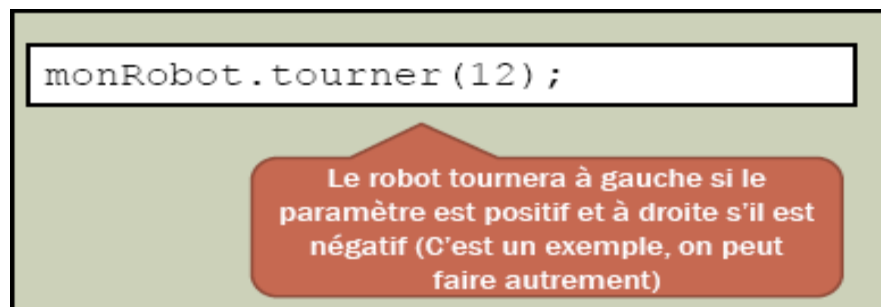
- Les attributs de l'objet sont **privés**
 - **Inaccessibles directement depuis l'extérieur**
- Les méthodes sont **publiques** et forme une interface de manipulation de l'objet



Introduction

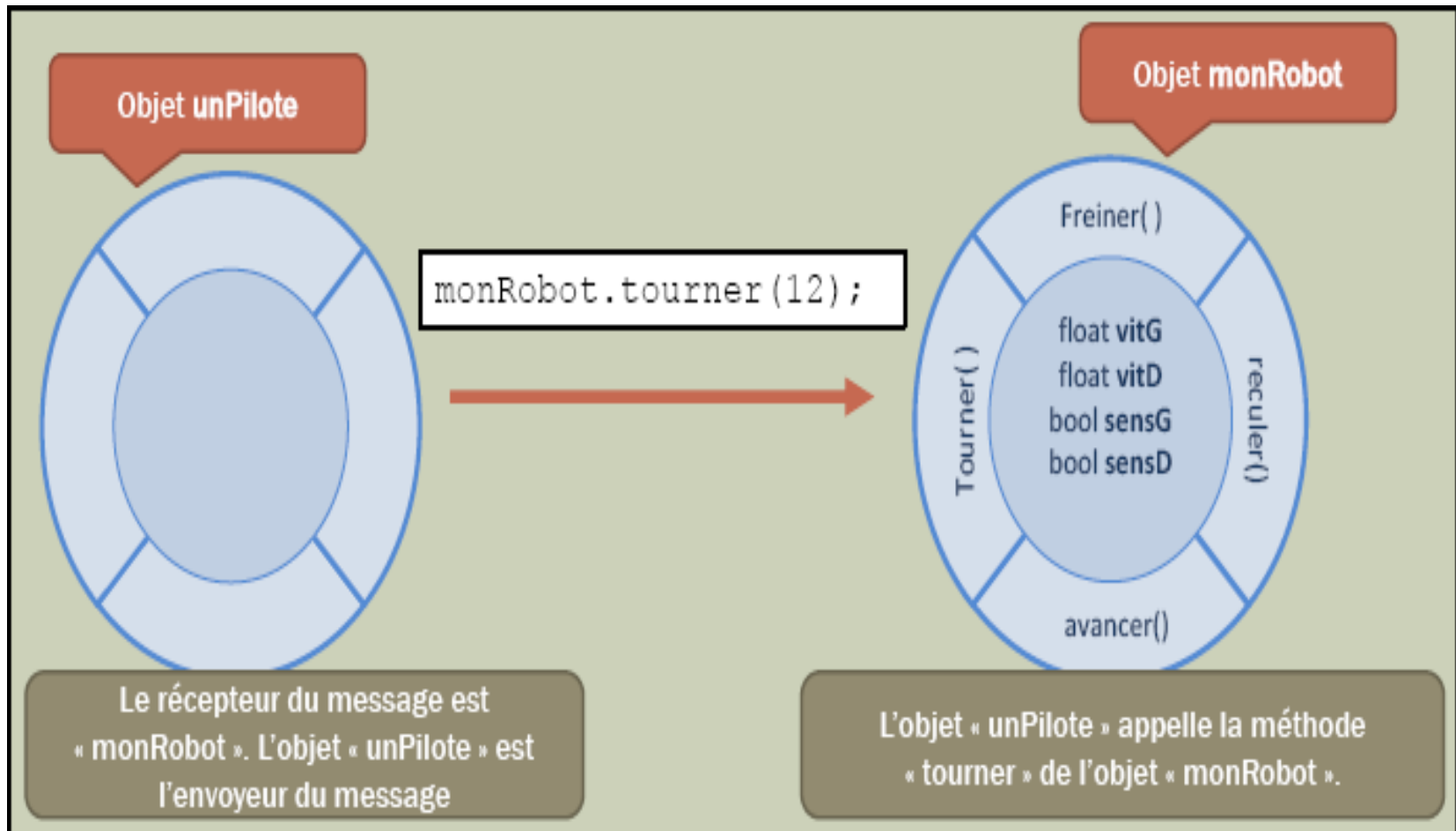
LES OBJETS COMMUNIQUENT

- **Les objets vont être capables d'interagir et de communiquer entre eux**
 - Par l'intermédiaire des méthodes publiques
 - Ils vont s'envoyer des messages par l'intermédiaire des méthodes publiques
- **Pour envoyer un message au robot :**
 - On appelle la méthode (exemple : tourner())
 - En spécifiant l'objet cible (exemple : monRobot)
 - En précisant d'éventuels paramètres



Introduction

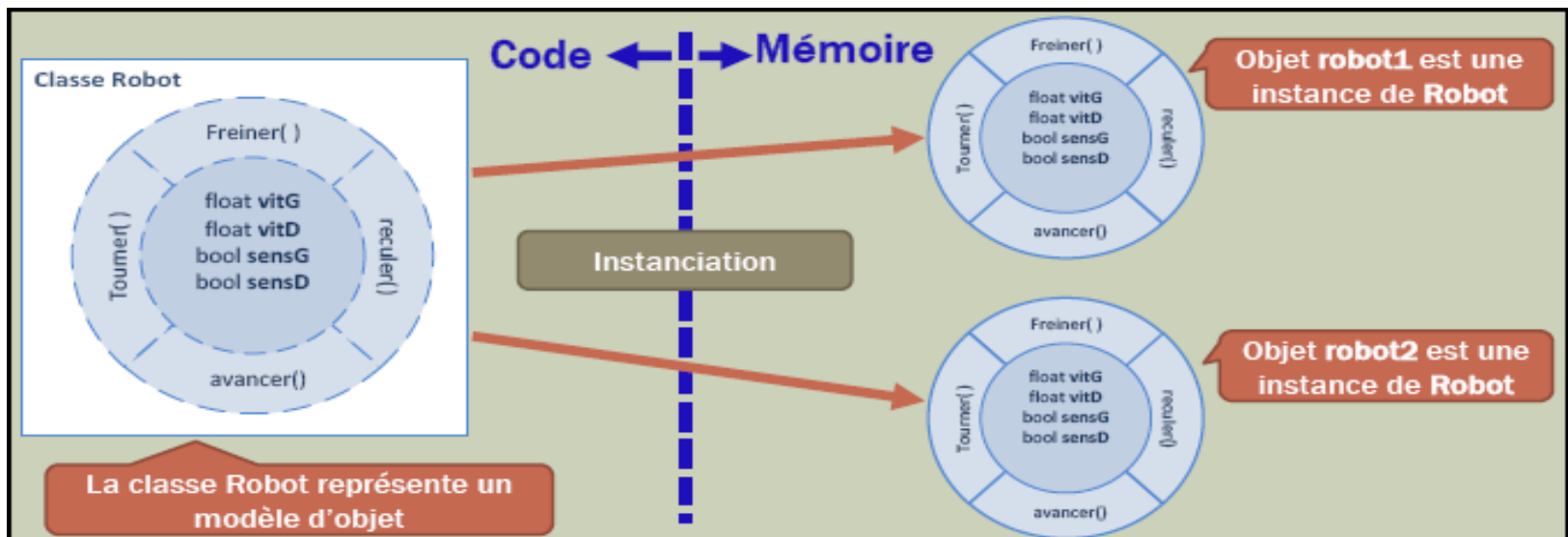
L'appel de méthode est effectué par un autre objet



Introduction

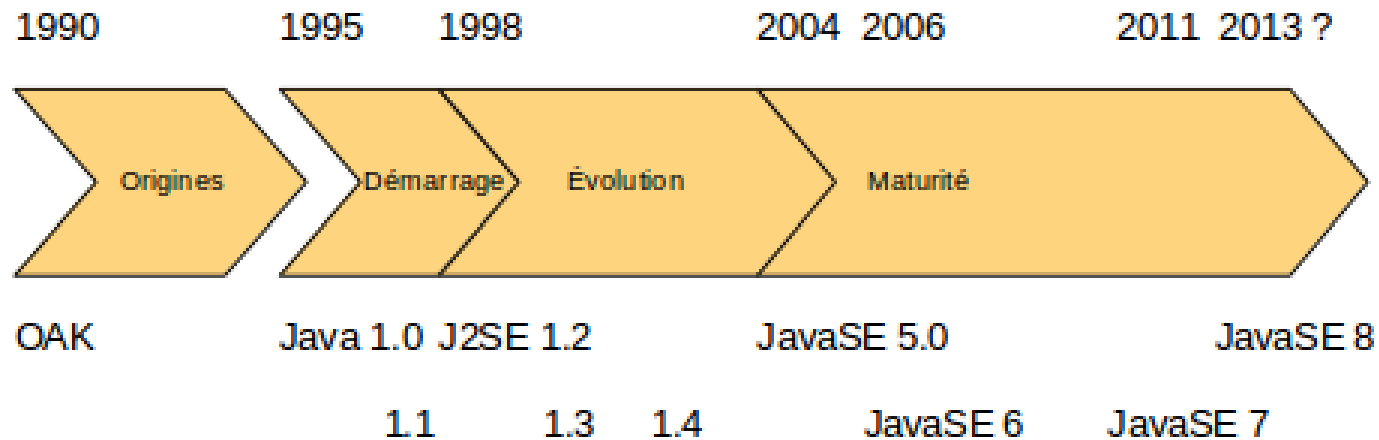
LES CLASSES ET LES OBJETS

- Avant de créer des objets, il faut définir un modèle
- Des objets pourront être créés à partir de ce modèle
- Ce modèle s'appelle **une classe**
- Les objets fabriqués à partir du modèle sont **des instances**



Introduction

Historique de Java



Introduction

Spécifications de Java

- Java, c'est en fait
 - **le langage Java** : <http://java.sun.com/docs/books/jls/>
 - **une JVM** : <http://java.sun.com/docs/books/vmspec/>
 - **les API** : selon la documentation *javadoc* fournie avec les différents paquets
- Java n'est pas normalisé ; son évolution est gérée par le JCP (Java Community Process ; <http://www.jcp.org/>) dans lequel Oracle tient une place prépondérante

Introduction

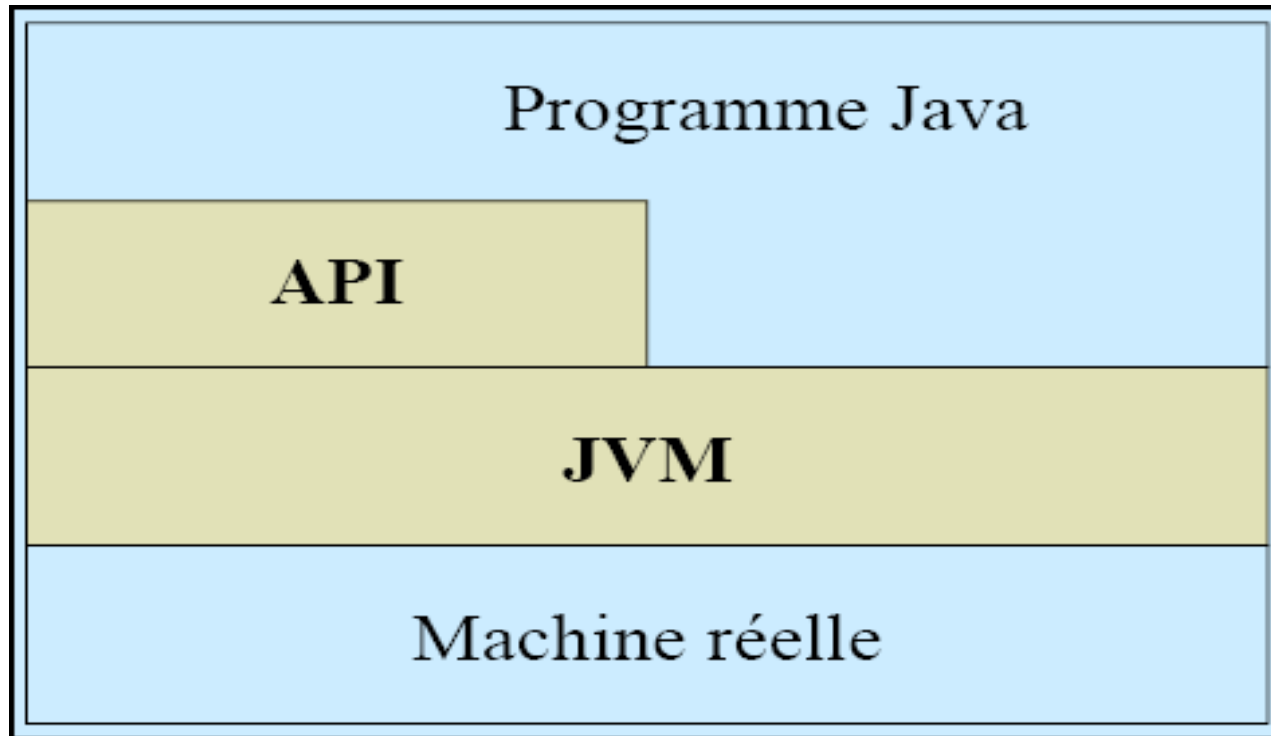
Implémentation de référence

Oracle accompagne les spécifications Java

- d'une implémentation de référence
- de nombreux tutoriels

Introduction

Plate-forme Java



- *API (Application Programming Interface)* : bibliothèques de classes standard

Introduction

3 éditions de Java

- **Java SE (JSE)** : Java Standard Edition ; JDK = *Java SE Development Kit*
- **Java EE (JEE)** : Enterprise Edition qui ajoute les API pour écrire des applications installées sur les serveurs dans des applications distribuées : servlet, JSP, JSF, EJB,...
- **Java ME (JME)** : Micro Edition, version pour écrire des programmes embarqués (carte à puce/*Java card*, *téléphone portable*,...)

Introduction

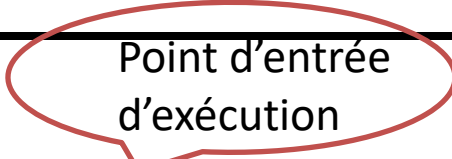
Votre environnement de développement

- Éditeur de texte (***emacs***, avec *JDE*)
- Compilateur (***javac***)
- Interpréteur de *bytecode* (***java***)
- **Aide en ligne** sur le JDK (sous navigateur Web)
- Générateur automatique de documentation (***javadoc***)
- Testeur pour applet (***appletviewer***)
- Débogueur (***jdb***)
- ...
- Après l'étude des paquetages, éventuellement un **IDE** tel que NetBeans ou Eclipse

Introduction

Premier programme Java

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



Point d'entrée d'exécution

- La classe HelloWorld est **public**, donc le fichier qui la contient doit s'appeler (en tenant compte des majuscules et minuscules) **HelloWorld.java**

Introduction

Compilation en Java → bytecode

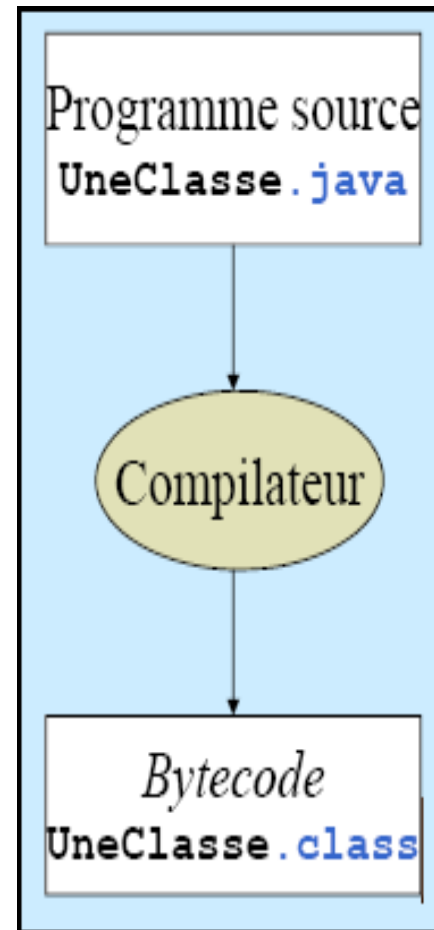
- En Java, le code source n'est pas traduit **directement** dans le langage de l'ordinateur
- Il est d'abord traduit dans un langage appelé « ***bytecode*** », langage d'une ***machine virtuelle*** (JVM ; Java Virtual Machine) définie par Oracle
- Ce langage est **indépendant** de l'ordinateur qui va exécuter le programme

Introduction

La compilation fournit du bytecode

Programme écrit en Java

Programme en *bytecode*,
indépendant de l'ordinateur



Introduction

Compilation avec javac

Oracle fournit le compilateur javac avec le JDK

javac HelloWorld.java

crée un fichier « **HelloWorld.class** » **qui** contient le *bytecode*,
situé dans le même répertoire que le fichier « **.java** »

- Le fichier à compiler peut être désigné par un chemin absolu ou relatif :

javac util/Liste.java

Introduction

Exécution du bytecode

- Le *bytecode* doit être exécuté par une JVM
- Cette JVM n'existe pas ; elle est simulée par un programme qui **interprète** le *bytecode* :
 - lit les instructions (en *bytecode*) du programme **.class**,
 - les traduit dans le langage natif du processeur de l'ordinateur
 - lance leur exécution

Introduction

Exécution avec java

- *Oracle fournit le programme java qui simule une JVM*

java HelloWorld ← pas de suffixe `.class`

exécute le *bytecode de la méthode **main*** de la classe **HelloWorld**

- **HelloWorld est un nom de classe et pas un nom de fichier.**

Introduction

Où doit se trouver le fichier .class ?

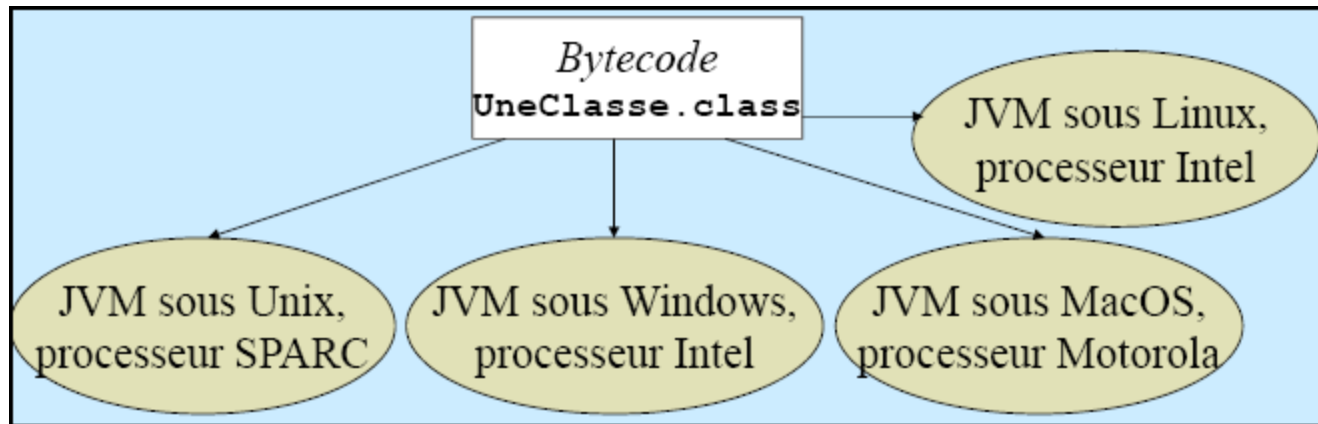
java HelloWorld

HelloWorld.class doit se trouver dans le *classpath*

- *Le classpath peut recevoir une valeur*
 - avec l'option **-classpath** de la commande java :
java -classpath rep1/rep2 HelloWorld
 - avec la variable d'environnement **CLASSPATH** (pas recommandé)

Introduction

*Le **bytecode** peut être exécuté par n'importe quelle JVM*



- Si un système possède une JVM, il peut exécuter tous les fichiers **.class** compilés sur n'importe quel autre système

Introduction

Avantages de la JVM pour Internet

- Grâce à sa portabilité, le **bytecode** *d'une classe* peut être chargé depuis une machine distante du réseau, et exécutée par une JVM locale
- La JVM fait de nombreuses vérifications sur le *bytecode avant son exécution pour s'assurer qu'il* ne va effectuer aucune action dangereuse
- La JVM apporte donc
 - de la **souplesse** pour le chargement du code à exécuter
 - mais aussi de la **sécurité** pour l'exécution de ce code

Table des matières

- **Introduction**
- **Définition de classes Java**
- **Les bases du langage**
- **Polymorphisme et héritage**
- **Classes utilitaires et Collection**
- **Gestion des exceptions**
- **Package IO**
- **Accès aux données**
- **Gestion des logs avec Log4j**
- **Tests unitaires avec JUnit**

Définition de classes Java

SOMMAIRE

- Type de programme
- Les classes en Java
- Conventions pour les identificateurs
- Les constructeurs(surcharge, this)
- Les méthodes(Accesseurs, Paramètres, Type retour)
- Surcharge de méthode
- Méthode toString
- Les variables (variables d'instances, variables locales)
- Méthodes et variables de classe

Définition de classes Java

Deux types de programmes

- *Applications indépendantes*
- *Applets* référencée par une page HTML et exécutée dans la JVM d'un navigateur Web

Définition de classes Java

Application indépendante

- Lancement de l'exécution de la classe de lancement de l'application (dite classe principale ; main en anglais) ;
par exemple :

java TestPoint

- *java lance l'interprétation du code de la méthode*
main() de la classe **TestPoint**

Définition de classes Java

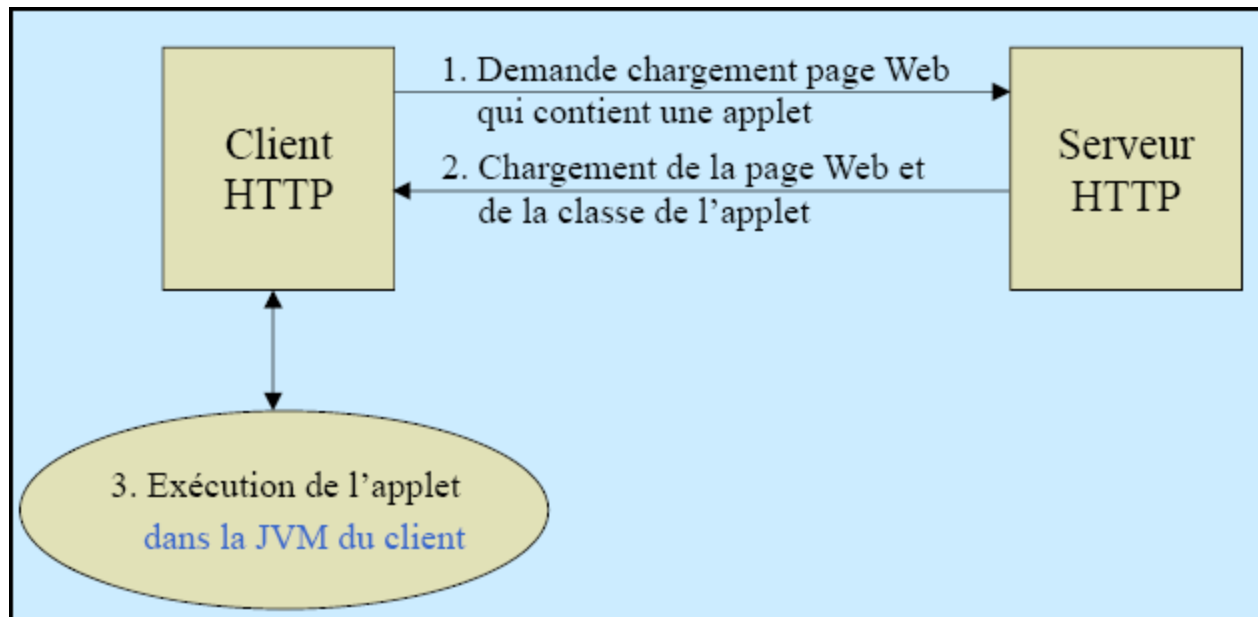
Méthode **main**

- Le « *profil* » d'une méthode est donné par son entête de définition ; celui de `main()` **doit être** :
`public static void main(String[] args)`
- **Signature d'une méthode** : nom de la méthode et ensemble des types de ses paramètres
- Signature de la méthode `main()` : **`main(String[])`**
- En Java, le type de la valeur de retour de la méthode ne fait pas partie de sa signature (au contraire de la définition habituelle d'une signature)

Définition de classes Java

Applet

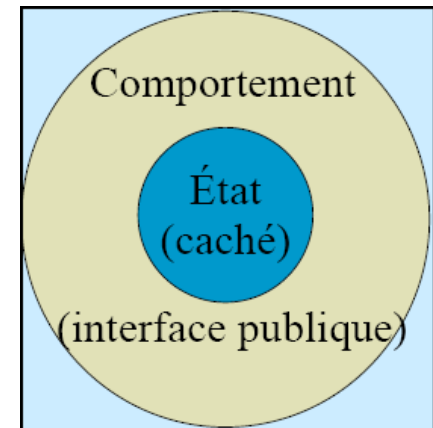
- Objet Java, référencé dans une page Web (écrite dans le langage HTML)
- En fait cet objet doit appartenir à une classe Java qui hérite de la classe **Applet**
- Le lancement d'un applet se fait quand la partie de la page Web qui référence l'applet est affichée par le client Web



Définition de classes Java

Les classes en Java

- Une classe est
 - un **type** qui décrit une **structure** (variables d'instances) et un **comportement** (méthodes)
 - un **module** pour décomposer une application en entités plus petites
 - un **générateur d'objets** (par ses constructeurs)
- Une classe permet d'encapsuler les objets :
 - les membres **public** sont **vus de l'extérieur**
 - les membres **private** sont **cachés**



Définition de classes Java

Éléments d'une classe

- Les **constructeurs** (il peut y en avoir plusieurs) servent à créer les **instances** (les objets) de la classe
- Quand une instance est créée, son état est conservé dans les **variables d'instance**
- Les **méthodes** déterminent le comportement des instances de la classe quand elles reçoivent un message
- Les variables et les méthodes s'appellent les **membres** de la classe

Définition de classes Java

Exemple : classe Livre

```
public class Livre {  
    private String titre, auteur;  
    private int nbPages;  
    // Constructeur  
    public Livre(String unTitre, String unAuteur) {  
        titre = unTitre;  
        auteur = unAuteur;  
    }  
    public String getAuteur() { // accesseur  
        return auteur;  
    }  
    public void setNbPages(int nb) { // modificateur  
        nbPages = nb;  
    }  
}
```

Variables d'instance

Constructeurs

Méthodes

Définition de classes Java

Conventions pour les identificateurs

- Les noms de classes commencent par une majuscule (ce sont les seuls avec les constantes) : **C**ercle, **O**bject
- Les mots contenus dans un identificateur commencent par une majuscule : **U**ne**C**lasse, une**M**ethode, une**A**utre**V**ariable
- Les constantes sont en majuscules avec les mots séparés par le caractère souligné « _ » : **U**NE_**C**ONSTANTE
- Si possible, des noms pour les classes et des verbes pour les méthodes

Définition de classes Java

Les constructeurs

- Une instance d'une classe est créée par un des **constructeurs** de la classe

Classes et instances

- Une fois qu'elle est créée, l'instance
 - a **son propre** état interne (les valeurs des variables d'instance)
 - **partage le code** qui détermine son comportement (les méthodes) avec les autres instances de la classe

Définition de classes Java

Constructeurs d'une classe

- Une classe a **un ou plusieurs constructeurs** qui servent à
 - créer les instances
 - initialiser l'état de ces instances
- Un constructeur
 - a le même nom que la classe
 - n'a pas de type retour

Définition de classes Java

Création d'une instance

```
public class Employe {  
    private String nom, prenom; } variables  
    private double salaire;    } d'instance  
  
    // Constructeur  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
    }  
    . . .  
    public static void main(String[] args) {  
        Employe e1;  
        e1 = new Employe("Dupond", "Pierre");  
        e1.setSalaire(1200);  
        . . .  
    }  
}
```

création d'une instance
de Employe

Définition de classes Java

Plusieurs constructeurs (surcharge)

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
  
    // 2 Constructeurs  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
    }  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
}
```

```
. . .  
e1 = new Employe("Dupond", "Pierre");  
e2 = new Employe("Durand", "Jacques", 1500);
```

Définition de classes Java

Désigner un constructeur par **this()**

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
  
    // Ce constructeur appelle l'autre constructeur  
    public Employe(String n, String p) {  
        this(n, p, 0);  
    }  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
    . . .  
    e1 = new Employe("Dupond", "Pierre");  
    e2 = new Employe("Durand", "Jacques", 1500);  
}
```

Définition de classes Java

Constructeur par défaut

- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe **Classe**, ce constructeur par défaut sera

[public] Classe() { }



Même accessibilité que
la classe (**public ou non**)

Définition de classes Java

toString()

- Il est conseillé d'inclure une méthode **toString** dans **toutes les classes que l'on écrit**
- Cette méthode renvoie une chaîne de caractères qui décrit l'instance
- Une description compacte et précise peut être très utile lors de la mise au point des programmes
- **System.out.println(*objet*)** affiche la valeur retournée par ***objet.toString()***

Définition de classes Java

- Les méthodes (Accesseurs)
- Deux types de méthodes servent à donner accès **aux variables** depuis l'extérieur de la classe :
 - ❖ les accesseurs **en lecture** pour lire les valeurs des variables ;
« accesseur en lecture » est souvent abrégé en « accesseur »
; *getter en anglais*
 - ❖ les accesseurs **en écriture**, ou **modificateurs**, ou mutateurs,
pour modifier leur valeur ; *setter en anglais*

Définition de classes Java

Autres types de méthode

- La plupart des méthodes permettent aux instances de la classe d'offrir des services plus **complexes** aux autres instances
- Enfin, des méthodes (**private**) **servent de** « sous-programmes » **utilitaires** aux autres méthodes de la classe

Définition de classes Java

Paramètres d'une méthode

- Souvent les méthodes ou les constructeurs ont besoin qu'on leur passe des données initiales sous la forme de paramètres
- On doit indiquer le type des paramètres dans la déclaration de la méthode :

setSalaire(double unSalaire)

calculerSalaire(int indice, double prime)

- Quand la méthode ou le constructeur n'a pas de paramètre, on ne met rien entre les parenthèses : **getSalaire()**

Définition de classes Java

Type retour d'une méthode

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode :

double calculSalaire(int indice, double prime)

- Le pseudo-type **void** indique qu'aucune valeur n'est renvoyée :

void setSalaire(double unSalaire)

Définition de classes Java

Exemples de méthodes

```
public class Employe {  
    . . .  
    public void setSalaire(double unSalaire) {  
        if (unSalaire >= 0.0)  
            salaire = unSalaire;  
    }  
    public double getSalaire() {  
        return salaire;  
    }  
    public boolean accomplir(Tache t) {  
        . . .  
    }  
}
```

Modificateur

Accesseur

Définition de classes Java

Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a **le même nom mais pas la même signature** qu'une autre méthode :
- Exemple:
calculerSalaire(int)
calculerSalaire(int, double)

Définition de classes Java

Surcharge d'une méthode

- En Java, il est interdit de surcharger une méthode en changeant seulement le type de retour
- Autrement dit, on ne peut différencier 2 méthodes par leur type retour
- Par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :

int calculerSalaire(int)

double calculerSalaire(int)

Définition de classes Java

Exemple

```
public class Livre {  
    ...  
    public String toString() {  
        return "Livre [titre=" + titre  
            + ",auteur=" + auteur  
            + ",nbPages=" + nbPages  
            + "]" ;  
    }  
}
```


Définition de classes Java

Les variables

- Les **variables d'instances**
 - sont déclarées en dehors de toute méthode
 - conservent l'état d'un objet, instance de la classe
 - sont accessibles et partagées par toutes les méthodes de la classe
- Les **variables locales**
 - sont déclarées à l'intérieur d'une méthode
 - conservent une valeur utilisée pendant l'exécution de la méthode
 - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées

Définition de classes Java

Déclaration des variables

- Toute variable doit être déclarée avant d'être utilisée
- Déclaration d'une variable : on indique au compilateur que le programme va utiliser une variable de ce nom et de ce type

```
double prime;  
Employe e1;  
Point centre;
```

Affectation

- L'affectation d'une valeur à une variable est effectuée par l'instruction *variable* = expression;
- L'expression est calculée et ensuite la valeur calculée est affectée à la variable

```
x = 3;  
x = x + 1;
```

Définition de classes Java

Initialisation d'une variable

- Une variable doit être initialisée (recevoir une valeur) avant d'être utilisée dans une expression
- Si elles ne sont pas initialisées par le programmeur, les **variables d'instance** (et les variables de classe étudiées plus loin) reçoivent les valeurs par défaut de leur type (0 pour les types numériques, par exemple)
- L'utilisation d'une **variable locale** non initialisée par le programmeur provoque une erreur (pas d'initialisation par défaut) à la compilation

Définition de classes Java

Initialisation d'une variable

- On peut initialiser une variable en la déclarant
- La formule d'initialisation peut être une expression complexe :

```
double prime = 200.0;  
Employe e1 = new Employe("Dupond", "Jean");  
double salaire = prime + 500.0;
```

Définition de classes Java

this

- Lorsqu'il n'y a pas d'ambiguïté, **this** est optionnel pour désigner un membre de l'instance courante
- Exemple de **this implicite**

```
public class Employe {  
    private double salaire;  
    . . .  
    public void setSalaire(double unSalaire) {  
        salaire = unSalaire;  
    }  
    public double getSalaire() {  
        return salaire;  
    }  
    . . .  
}
```

Implicitement this.salaire

Implicitement this.salaire

Définition de classes Java

this explicite

- **this** est utilisé surtout dans 2 occasions :

- pour distinguer une variable d'instance et un paramètre qui ont le même nom :

```
public void setSalaire(double salaire)
    this.salaire = salaire;
}
```

- un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

- **this** se comporte comme une variable **final** (mot-clé étudié plus loin), c'est-à-dire qu'on ne peut le modifier ; le code suivant est **interdit** : **this = valeur;**

Définition de classes Java

Exemple de this explicite

```
public class Document {  
    ...  
    public void imprimer(Imprimante imprimante) {  
        imprimante.ajouterRequete(this);  
    }  
    ...  
}
```

```
public class Imprimante {  
    ...  
    public void ajouterRequete(Document doc) {  
        // Ajoute le fichier associé au document  
        // dans la file d'attente d'impression  
        fileAttente.ajouter(doc.getFichier());  
    }  
    ...  
}
```

Définition de classes Java

Variables de classe

- Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les **variables de classe** (modificateur **static**)
- Si une variable de classe est initialisée dans sa déclaration, cette **initialisation** est exécutée une seule fois quand la classe est chargée en mémoire

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
    private static int nbEmployes = 0;  
    // Constructeur  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
        nbEmployes++;  
    }  
    . . .  
}
```


Définition de classes Java

Méthodes de classe

- Une méthode de classe (modificateur **static** en Java) exécute une action **indépendante d'une instance particulière** de la classe
- Une méthode de classe peut être considérée comme un **message envoyé à une classe**

- Exemple :

```
public static int getNbEmployes() {  
    return nbEmployes;  
}
```

- Depuis une autre classe, on la préfixe par le nom de la classe :
- Depuis sa classe, le nom de la méthode suffit

```
int n = Employe.getNbEmploye();
```

Définition de classes Java

Blocs d'initialisation **static**

- Ils permettent d'initialiser les variables **static trop complexes à initialiser dans leur déclaration**

```
class UneClasse {  
    private static int[] tab = new int[25];  
    static {  
        for (int i = 0; i < 25; i++) {  
            tab[i] = -1;  
        }  
    }  
    . . .  
}
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

Tables des matières

- **Introduction**
- **Définition de classes Java**
- **Les bases du langage**
- **Polymorphisme et héritage**
- **Classes utilitaires et Collection**
- **Gestion des exceptions**
- **Package IO**
- **Accès aux données**
- **Gestion des logs avec Log4j**
- **Tests unitaires avec JUnit**

Les bases du langage

SOMMAIRE

Mots-clés Java

Les commentaires

Type de données prédéfinis

Les boucles

Instructions liées aux boucles

Expression conditionnelle

Distinction de cas suivant une valeur

Opérateurs sur les types primitifs

Opérateur instanceof

Les modificateurs

Les variables

Les tableaux

Droits d'accès

Les bases du langage

Mots-clés Java

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

*: pas encore utilisé

** : depuis Java SE 5

Les bases du langage

Les commentaires

- Commentaires multi lignes

```
/*  
*/
```

- Commentaires sur une seule ou fraction de ligne

```
//
```

- Commentaires destinés au générateur de documentation
javadoc

```
/**  
*  
*  
*/
```

Les bases du langage

Type de données prédéfinis

Nombres entiers

- **byte** $-2^7, (2^7) - 1$ -128,127
- **short** $-2^{15}, (2^{15}) - 1$ -32768,32767
- **int** $-2^{31}, (2^{31}) - 1$ -2147483648, 2147483647
- **long** $-2^{63}, (2^{63}) - 1$
 -9223372036854775808, 9223372036854775807
- Les entiers peuvent être exprimés en octal (0323), en décimal (311) ou en hexadécimal (0x137).

Les bases du langage

Nombres réels

- **float** simple précision sur 32 bits
1.4023984 e-45 3.40282347 e38
- **double** double précision sur 64 bits
4.94065645841243544 e-324
1.79769313486231570 e308
- Représentation des réels dans le standard IEEE 754 Un suffixe *f* ou *d* après une valeur numérique permet de spécifier le type.
- Exemples : *double x = 145.56d ;*
 float y = 23.4f ;
 float f = 23.65 ; // Erreur

Les bases du langage

- **boolean**

- Valeurs *true* et *false*

- **char**

- Une variable de type char contient un seul caractère codé sur 16 bits (jeu de caractères 16 bits Unicode contenant 34168 caractères).

- Des caractères d'échappement existent :

- | | |
|----------------------|---------------------------|
| • \b Backspace | \t Tabulation horizontale |
| • \n Line Feed | \f Form Feed |
| • \r Carriage Return | \\" Guillemet |
| • \' Apostrophe | \\ BackSlash |

Les bases du langage

Valeurs par défaut

- Si elles ne sont pas initialisées, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean	false	
char	'\u0000'	
Entier (byte short int long)	0	0L
Flottant (float double)	0.0F	0.0D
Référence d'objet	null	

Les bases du langage

Les boucles

- **for/each**

```
for(Type variable: collection) {  
    body;  
}
```

```
for(String entry: entries) {  
    System.out.println(entry);  
}
```

- **for**

```
for(init; test; maj) {  
    body;  
}
```

```
for(int i=0; i<max; i++) {  
    System.out.println("Number: " + i);  
}
```

- **while**

```
while (test) {  
    body;  
}
```

```
int i = 0;  
while (i < max) {  
    System.out.println("Number: " + i);  
    i++; // "++" means "add one"  
}
```

- **do**

```
do {  
    body;  
} while (continueTest);
```

```
int i = 0;  
do {  
    System.out.println("Number: " + i);  
    i++;  
} while (i < max);
```

Les bases du langage

Instructions liées aux boucles

- **break** sort de la boucle et continue après la boucle
- **continue** passe à l'itération suivante
- **break et continue peuvent être suivis d'un nom d'étiquette** qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle)

```
int somme = 0;
for (int i = 0; i < tab.length; i++) {
    if (tab[i] == 0) break;
    if (tab[i] < 0) continue;
    somme += tab[i];
}
System.out.println(somme);
```

Qu'affiche ce code avec le tableau
1 ; -2 ; 5 ; -1 ; 0 ; 8 ; -3 ; 10 ?

```
boucleWhile: while (pasFini) {
    ...
    for (int i = 0; i < t.length; i++) {
        ...
        if (t[i] < 0)
            continue boucleWhile;
        ...
    }
    ...
}
```

Les bases du langage

Instructions de contrôle

Alternative « **if** » ou « **if... else** »

if (*expression Booléenne*)

bloc-instructions ou instruction

else

bloc-instructions ou instruction

« **else** »

facultatif

```
int x = y + 5;  
if (x % 2 == 0) {  
    type = 0;  
    x++;  
} else  
    type = 1;
```

Un bloc serait préférable, même
s'il n'y a qu'une seule instruction

Les bases du langage

if emboîtés

- Lorsque plusieurs **if** sont emboîtés les uns dans les autres, un bloc **else** se rattache au dernier bloc **if** qui n'a pas de **else** (utiliser les accolades pour ne pas se tromper)

```
int x = 3;  
int y = 8;  
if (x == y)  
    if (x > 10)  
        x = x + 1;  
    else  
        x = x + 2;
```

Quelle valeur pour x
à la fin de ce code ?

Facile de se tromper

Plus clair en mettant
des accolades !

```
if (x == y) {  
    if (x > 10) {  
        x = x + 1;  
    }  
} else {  
    x = x + 2;  
}
```

Les bases du langage

Expression conditionnelle

expressionBooléenne ? expression1 : expression2

Exemple: `int y = (x % 2 == 0) ? x + 1 : x;`

est équivalent à

```
int y;  
if (x % 2 == 0)  
    y = x + 1;  
else  
    y = x;
```

Les bases du langage

Distinction de cas suivant une valeur

```
switch(expression) {  
case val1: instructions;  
break;  
...  
case valn: instructions;  
break;  
default: instructions;  
}
```

Attention, sans **break**, les instructions du cas suivant sont exécutées !

- *expression* est de type **char**, **byte**, **short**, ou **int**, ou de type énumération ou **String** (depuis le JDK 5)
- S'il n'y a pas de clause **default**, *rien n'est exécuté si expression ne correspond à aucun case*

Les bases du langage

Exemple de switch

```
char lettre;  
int nbVoyelles = 0, nbA = 0,  
    nbT = 0, nbAutre = 0;  
...  
switch (lettre) {  
case 'a' : nbA++;  
case 'e' :           // pas d'instruction !  
case 'i' : nbVoyelles++;  
            break;  
case 't' : nbT++;  
            break;  
default : nbAutre++;  
}
```

```
public int nbJours(String mois) {  
    switch(mois) {  
        case "avril": case "juin" :  
        case "septembre" : case "novembre":  
            return 30;  
        case "janvier": case "mars": case "mai":  
        case "juillet": case "août": case "décembre":  
            return 31;  
        case "février":  
            ...  
        default:  
            ...  
    }  
}
```

Les bases du langage

Opérateurs sur les types primitifs

- Voici les plus utilisés :

= + - * / % ++ -- += -= *= /=

== != > < >= <=

&& || ! (et, ou, négation)

- x++** : la valeur actuelle de **x** est utilisée dans l'expression et juste après x est incrémenté
- ++x** : la valeur de **x** est incrémentée et ensuite la valeur de x est utilisée

Les bases du langage

Opérateur **instanceof**

- La syntaxe est :
objet instanceof nomClasse
- Exemple : if (x instanceof Livre)
- Le résultat est un booléen :
 - **true** si x est de la classe Livre
 - **false** sinon

Les bases du langage

Représentation mémoire

- Java manipule différemment les types primitifs et les objets
- Les variables contiennent
 - des **valeurs** de types primitifs
 - des **références** aux objets
- L'espace mémoire alloué à une variable locale est situé dans la **pile**
- Si la variable est d'un type primitif, sa valeur est placée dans la pile

Les bases du langage

Exemple d'utilisation des références

```
int m() {  
    A a1 = new A();  
    A a2 = a1;  
    ...  
}
```

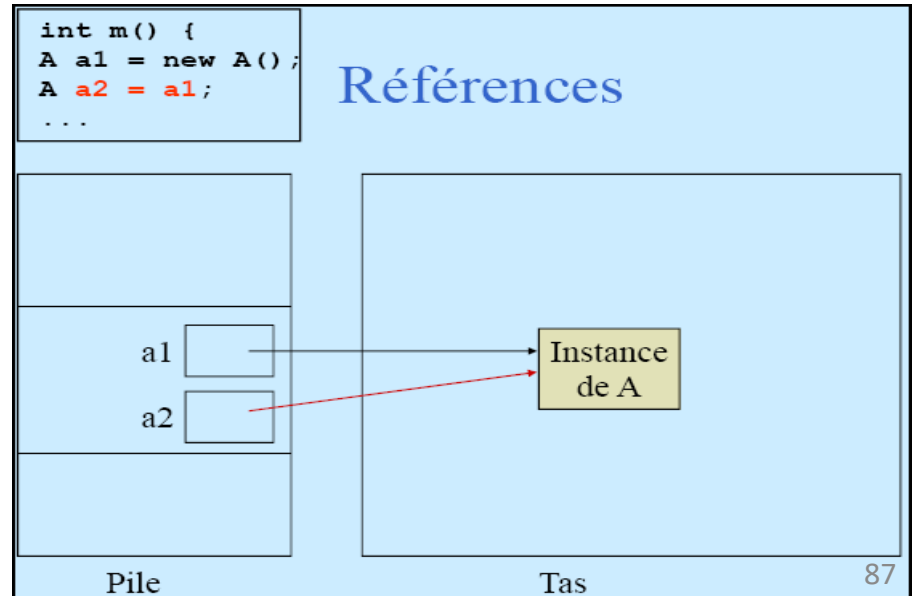
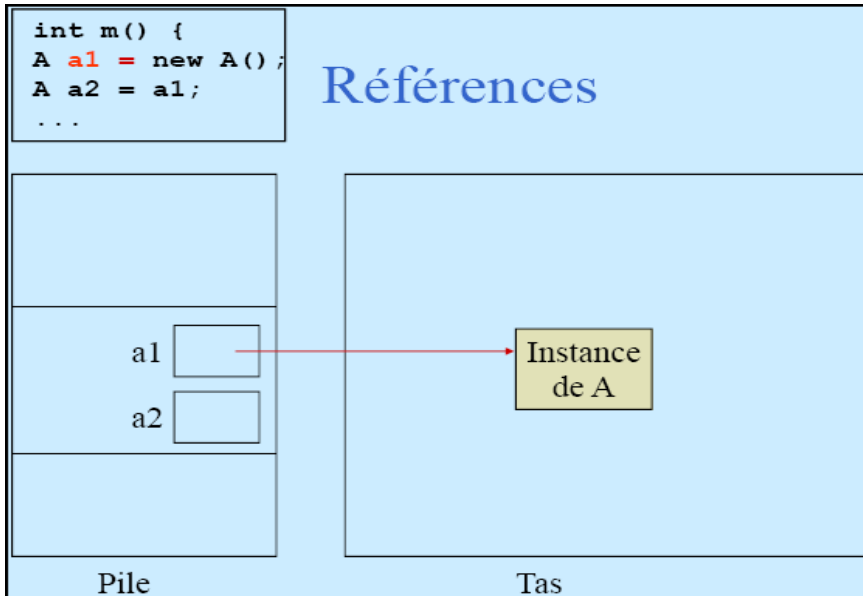
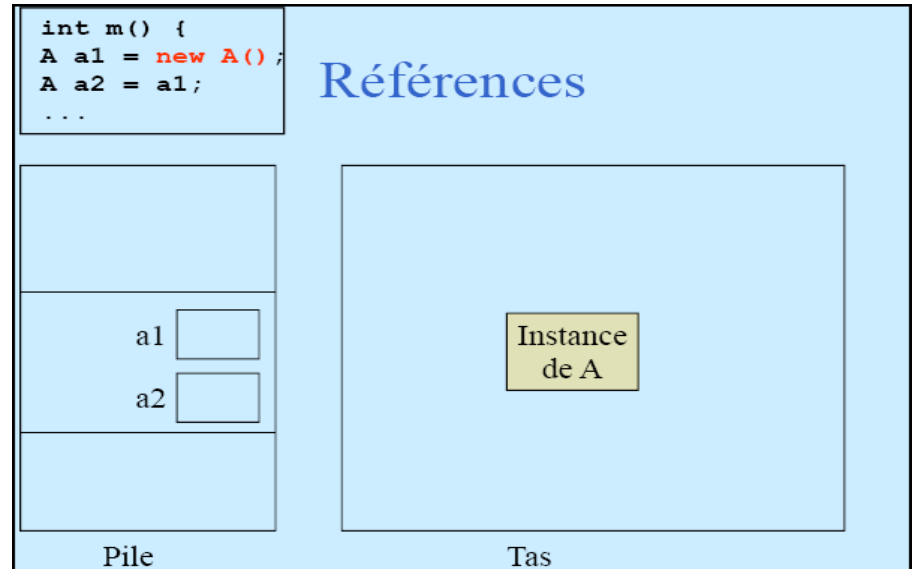
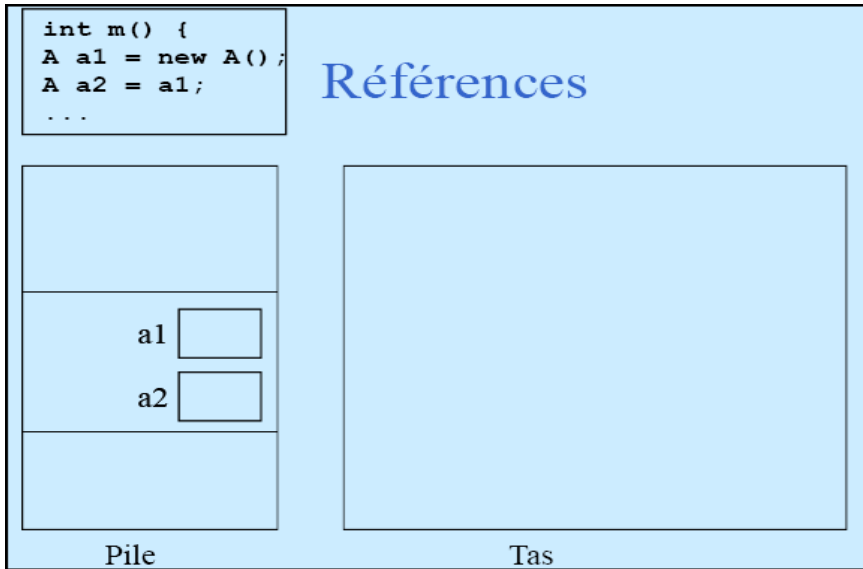
- Que se passe-t-il lorsque la méthode **m()** est appelée ?

Les bases du langage

Représentation mémoire

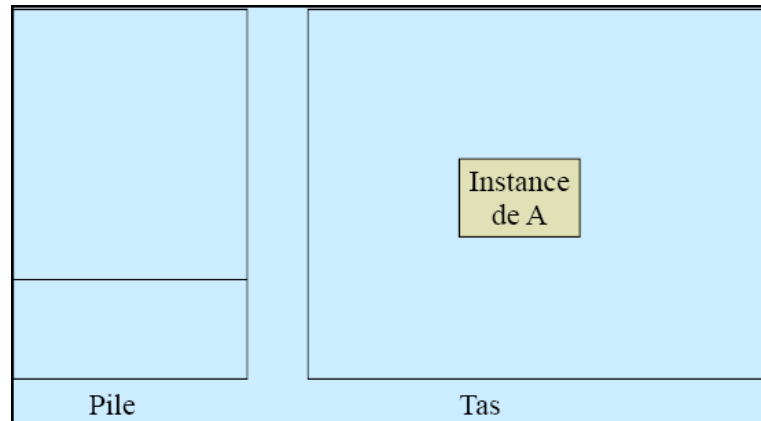
- Sinon la variable contient une référence à un objet ; la valeur de la référence est placée dans la pile mais l'objet référencé est placé dans le tas
- Lorsque l'objet n'est plus référencé, un « ramasse-miettes » (*garbage collector, GC*) libère la mémoire qui lui a été allouée

Les bases du langage

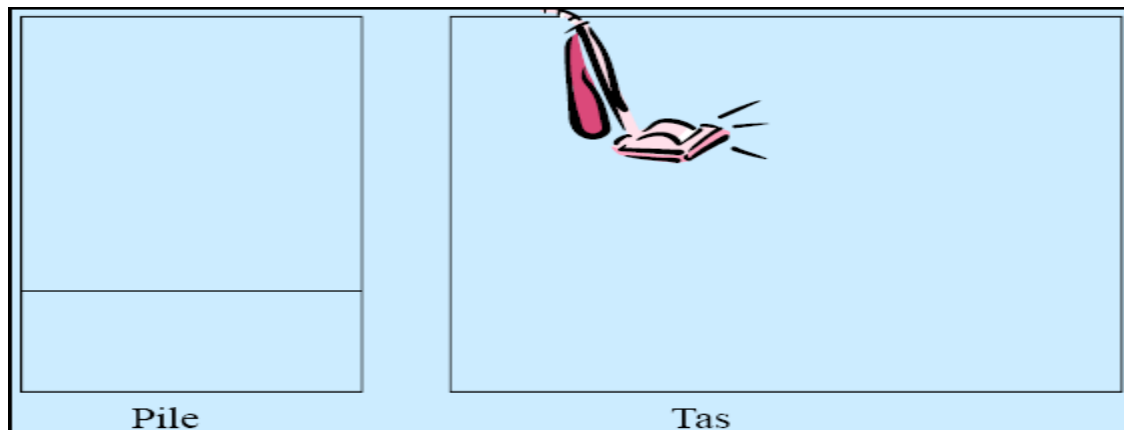


Les bases du langage

- Après l'exécution de la méthode `m()`, l'instance de **A** n'est plus **référéncée** mais reste dans le tas



- ...le ramasse-miette interviendra à un moment aléatoire...



Les bases du langage

Ramasse-miettes

- Le ramasse-miettes (*garbage collector*) est une tâche qui
 - libère la place occupée par les instances non référencées
 - compacte la mémoire occupée
- Il travaille en arrière-plan et intervient
 - quand le système a besoin de mémoire
 - ou, de temps en temps, avec une priorité faible

Les bases du langage

Modificateur **final**

- Le modificateur **final** indique que la valeur de la variable ne peut être modifiée : on pourra lui donner une valeur une seule fois dans le programme

Variable de classe **final**

- Une variable de classe **static final** est constante dans tout le programme ; exemple :
static final double PI = 3.14;

Variable locale **final**

- Une variable locale peut aussi être **final** : sa valeur ne pourra être donnée qu'une seule fois

Les bases du langage

Variable d'instance **final**

- Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 Instances
- Une variable d'instance **final** peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

Variable **final**

- Si la variable est d'un type primitif, sa valeur ne peut changer
- Si la variable référence un objet, elle ne pourra référencer un autre objet mais l'état de l'objet pourra être modifié

```
final Employe e = new Employe("Bibi");  
.  
.  
.  
e.nom = "Toto";           // Autorisé !  
e.setSalaire(12000);      // Autorisé !  
e = new Employe("Bob");   // Interdit
```

Les bases du langage

Les tableaux

- En Java les tableaux sont considérés comme des objets (dont la classe hérite de **Object**) :
 - les tableaux sont créés par l'opérateur **new**
 - ils ont une variable d'instance (**final**) : **final int length**
 - ils héritent des méthodes d'instance de **Object**
- Les tableaux peuvent être déclarés suivant les syntaxes suivantes :
 - type [] nom ;***
 - type nom [] ;***
 - ***Exemples :***
 - int table [] ;***
 - double [] d1,d2 ;***

Les bases du langage

- La taille d'un tableau est allouée dynamiquement par l'opérateur *new*

```
table = new int [10] ;  
int table2 [ ] = new int [20] ;  
int table3 [ ] = {1,2,3,4,5} ;  
Employe[] employes = {  
    new Employe("Dupond", "Sylvie"),  
    new Employe("Durand", "Patrick")  
}
```

- La taille n'est pas modifiable et peut être consultée par la propriété *length*

```
System.out.println (table3.length) ;  
int [ ] [ ] Matrice = new int [10][20] ;  
System.out.println (Matrice.length) ; // 1ère dimension  
System.out.println (Matrice[0].length) ; // 2ème dimension
```

Les bases du langage

Droits d'accès

- Toutes les méthodes et données membres définies au sein d'une classe sont utilisables par toutes les méthodes de la classe.
- Lors de la conception d'une classe, il faut décider des méthodes/variables qui seront visibles à l'extérieur de cette classe.
- Java implémente la protection des 4 P (**public**, **package**, **protected**, **private**).

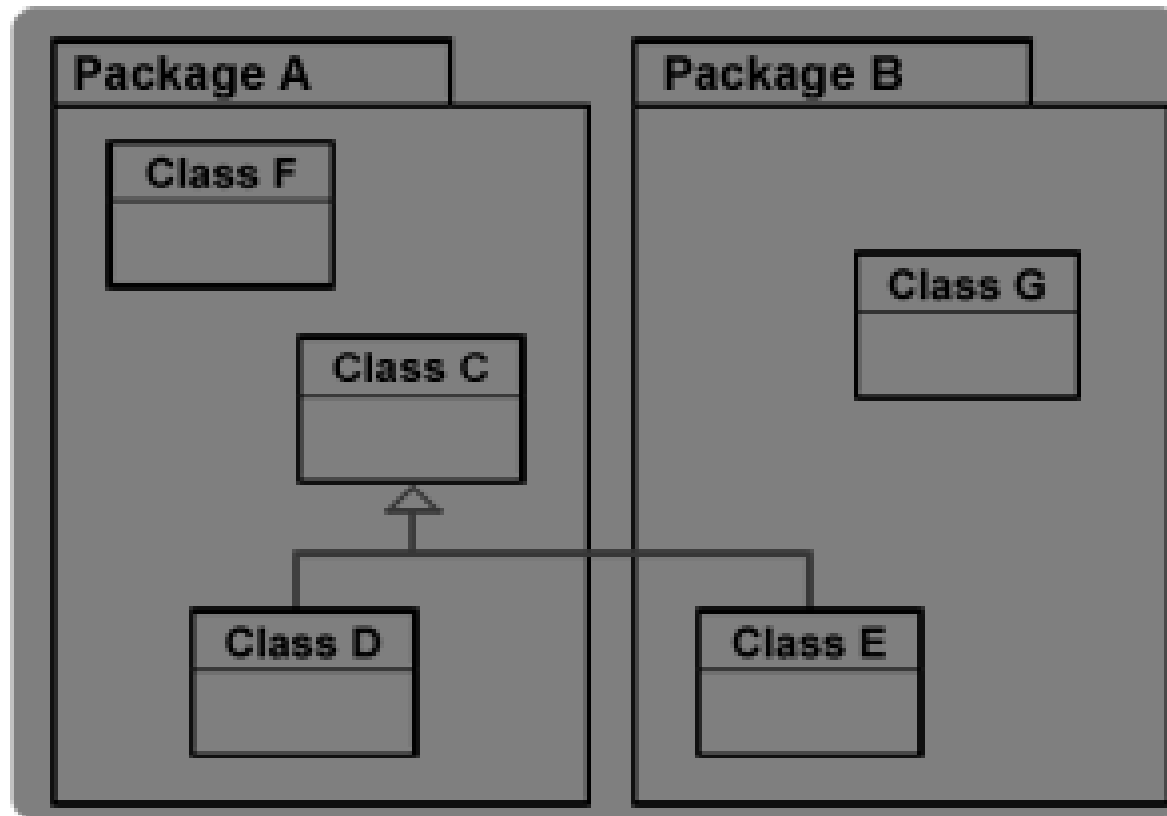
Les bases du langage

La protection des 4 P

- **private** : visible uniquement au sein de la classe.
- **public** : visible partout
- **Le droit par défaut** est une visibilité des classes/données/membres pour toutes les classes au sein d'un même package. Il n'y a hélas pas de mot clé pour préciser explicitement cet accès.
- **protected** : visible uniquement dans la classe et dans les classes dérivées de cette classe.

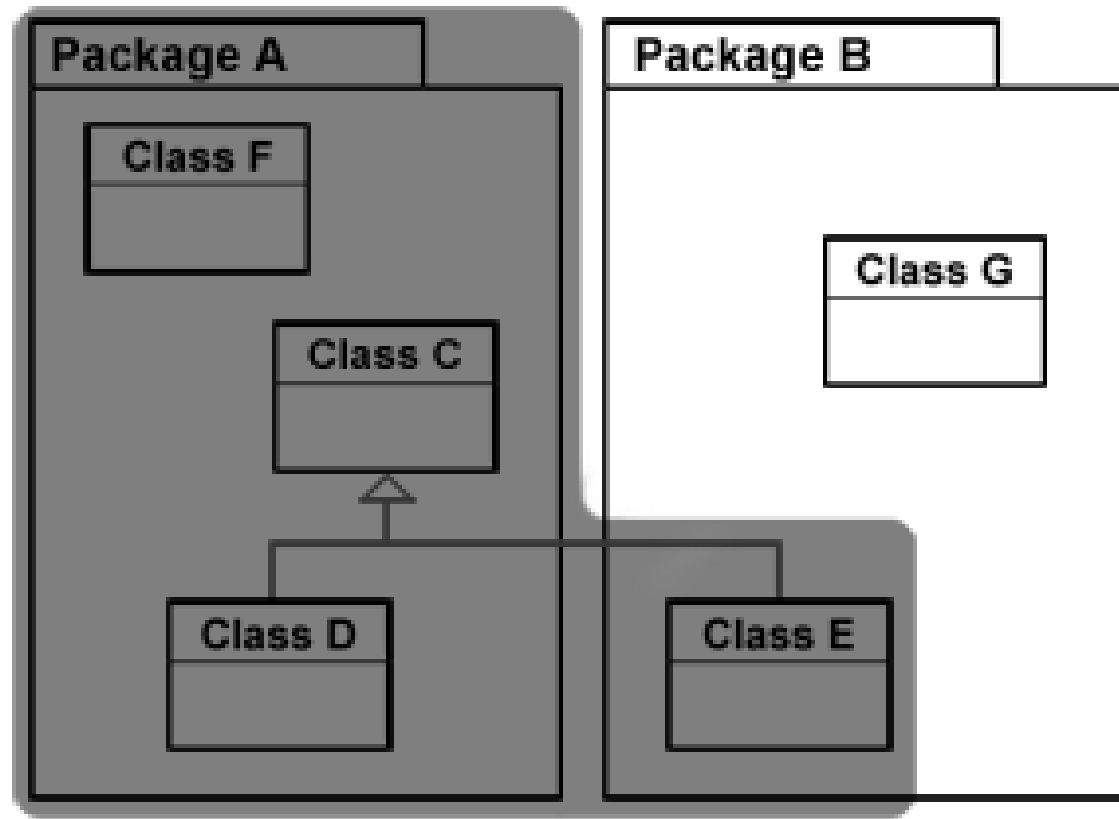
Les bases du langage

Public



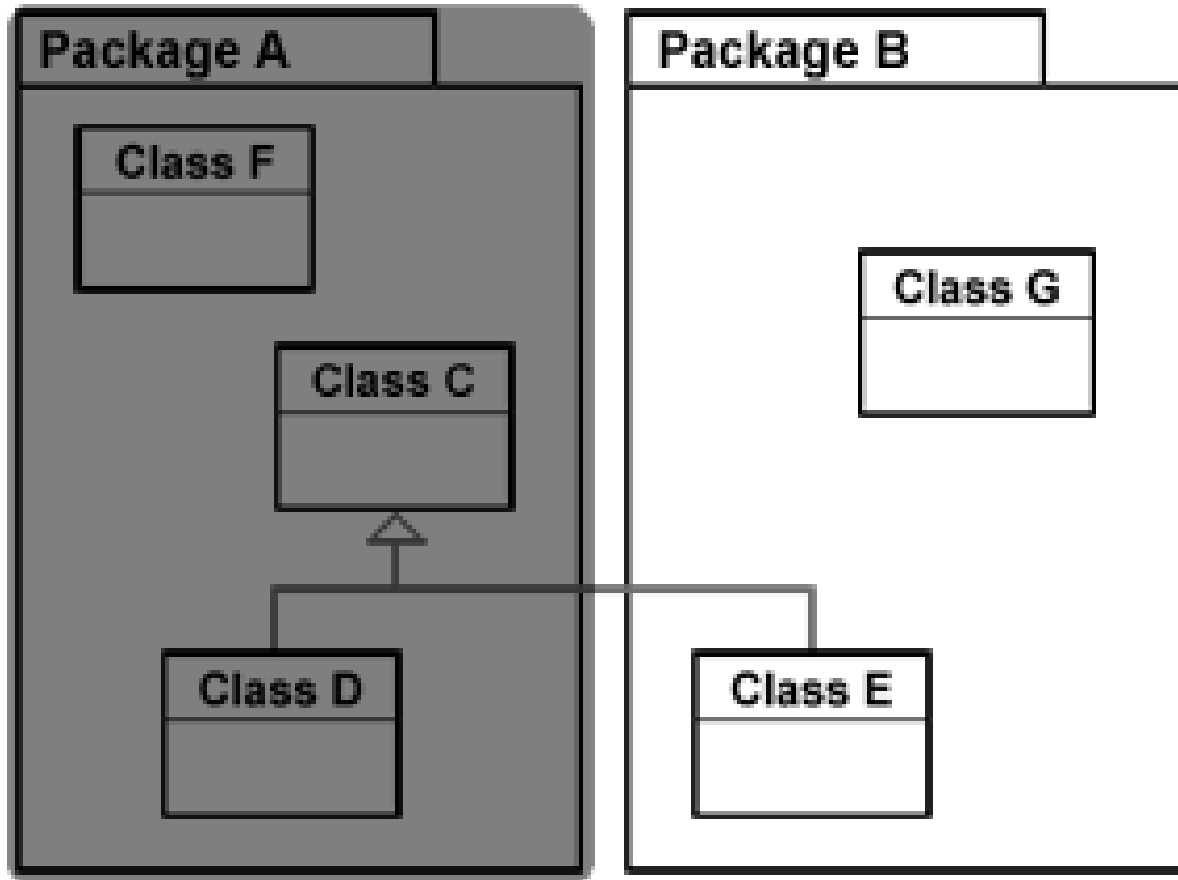
Les bases du langage

Protected



Les bases du langage

Le droit par défaut (**Package**)



Les bases du langage

Private

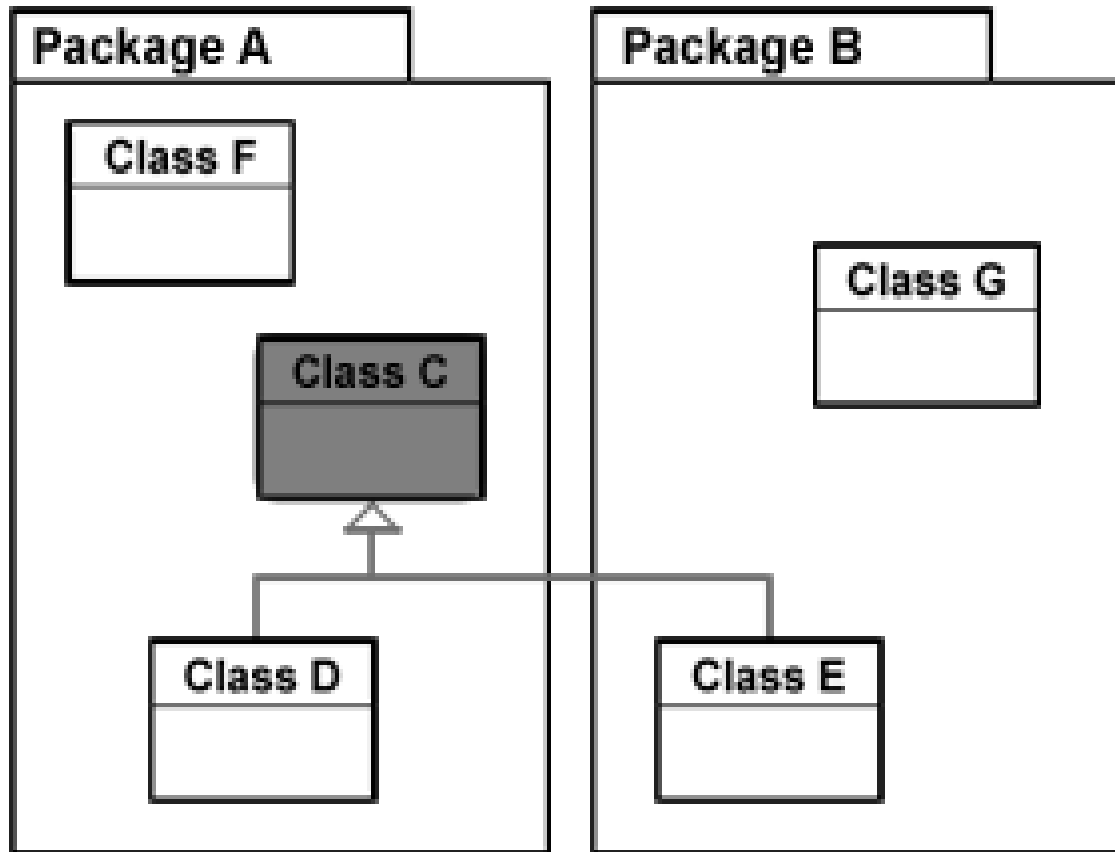


Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

Polymorphisme et héritage

SOMMAIRE

- Délégation
- Redéfinition et surcharge
- Constructeur
- Interfaces
- Classes abstraites
- Méthodes abstraites
- Wrapper
- Boxing/unboxing

Polymorphisme et héritage

Réutilisation

- **Objectif:** raccourcir les temps d'écriture et de mise au point du code d'une application
- **Moyen:** réutiliser du code déjà utilisé

Polymorphisme et héritage

Réutilisation par une classe **C2** du code d'une classe **C1**

- Soit **C1** une classe déjà écrite dont on ne possède pas le code source
- On veut utiliser la classe **C1** pour écrire le code d'une classe **C2**
- Plusieurs moyens :
 - **C2** **hérite** de **C1**
 - **C2** peut **déléguer** à une instance de **C1** une partie de la tâche qu'elle doit accomplir

Polymorphisme et héritage

Délégation « pure »

- Une méthode **m2()** de la classe **C2** délègue une partie de son travail à un **objet c1** de la classe **C1**, créé par la méthode **m2** :

création d'une instance de C1

```
public int m2() {  
    // ...  
    C1 c1 = new C1();  
    r = c1.m1();  
    // ...  
    return 2;  
}
```

utilisation de l'instance

L'objet **c1** de la classe **C1** est passé en paramètre de la méthode **m2**

```
public int m2(C1 c1) {  
    // ...  
    r = c1.m1();  
    // ...  
    return 2;  
}
```


Polymorphisme et héritage

2 façons de voir l'héritage

Généralisation:

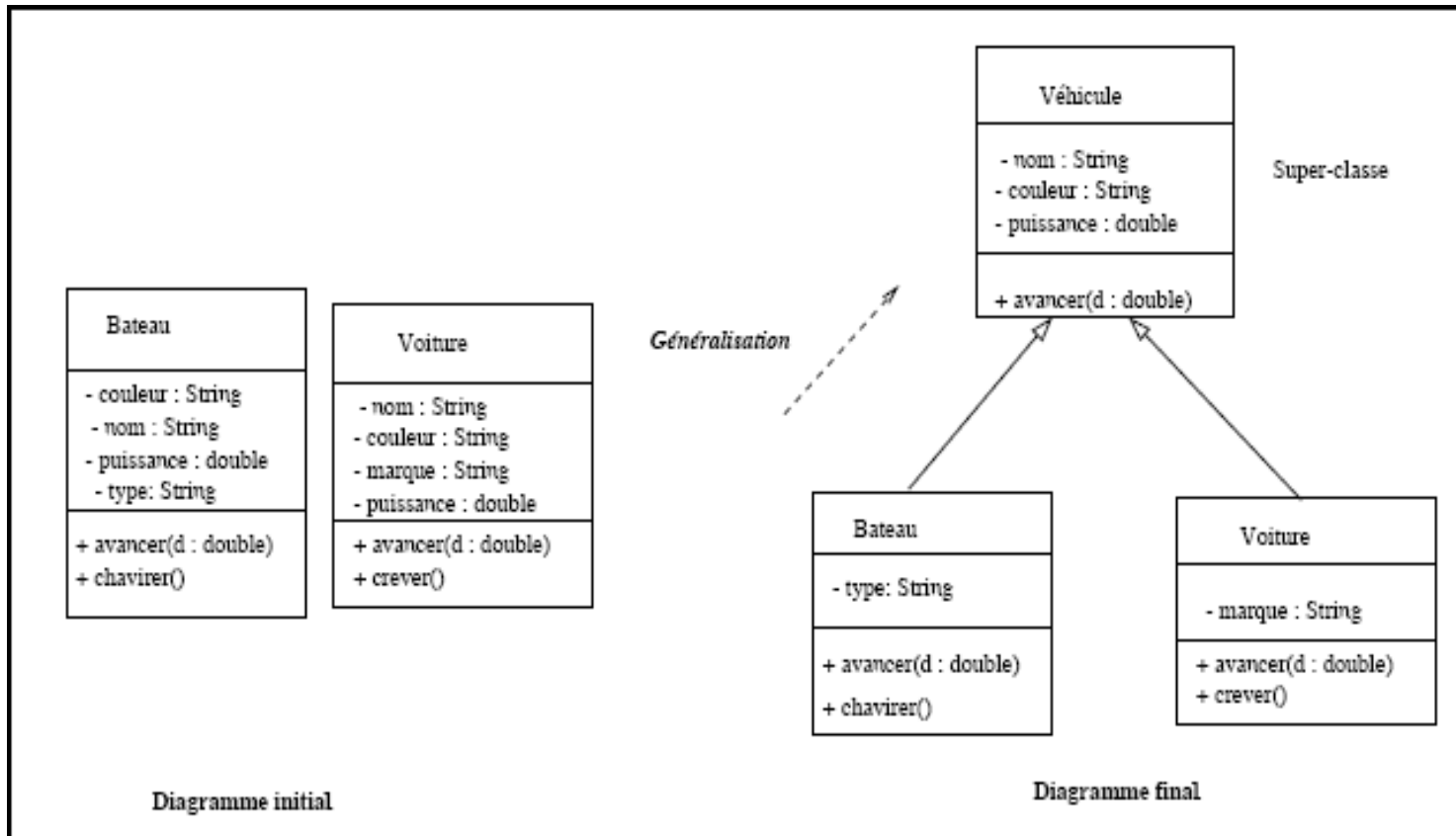
- Réunir des objets possédant des caractéristiques communes dans une nouvelle classe plus générale appelée super-classe

Spécialisation:

- Séparer des objets suivant des caractéristiques plus spécifiques dans une nouvelle classe plus spécifique appelée sous-classe

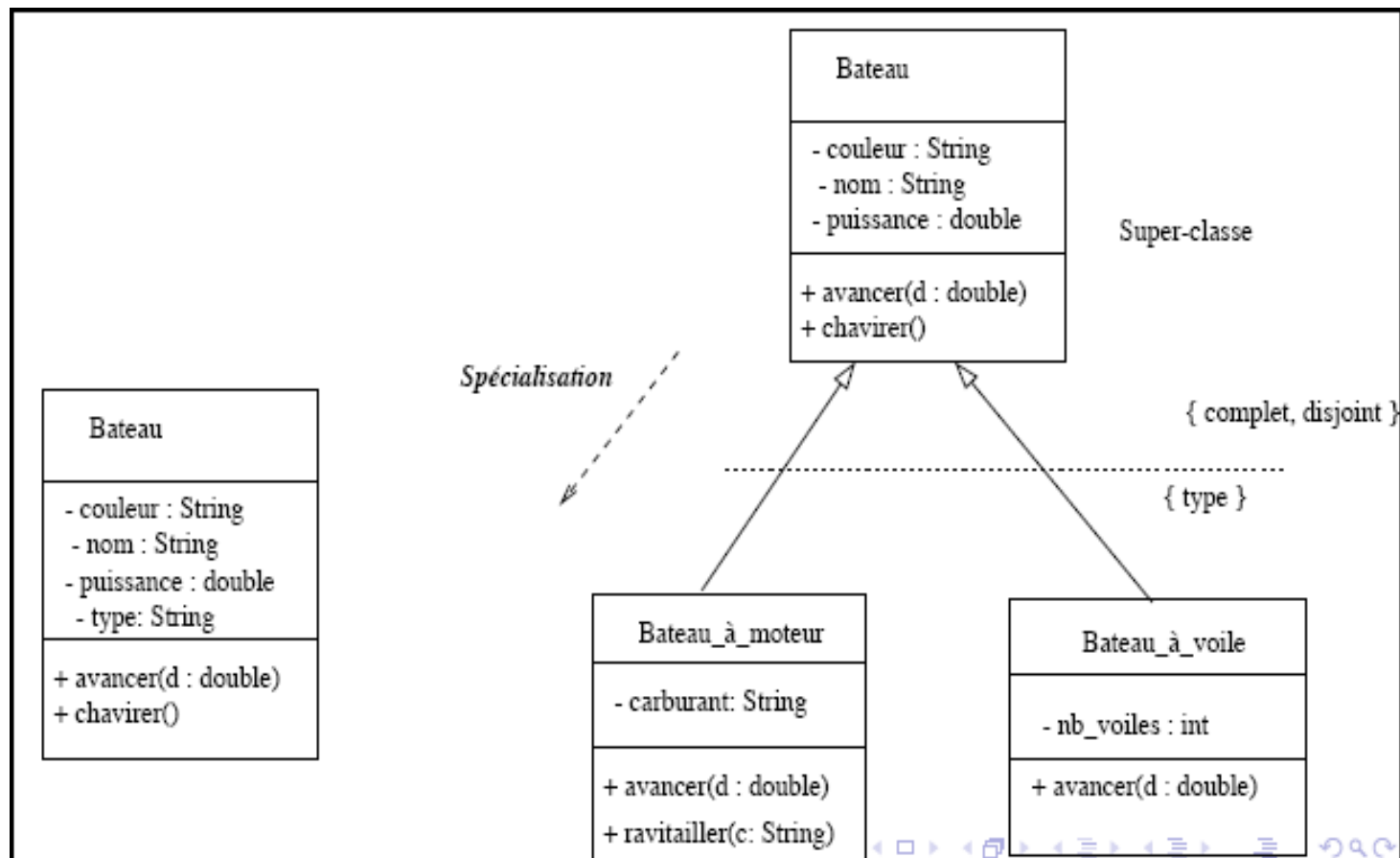
Polymorphisme et héritage

Généralisation



Polymorphisme et héritage

Spécialisation



Polymorphisme et héritage

L'héritage en Java

- En Java, chaque classe a **une et une seule classe mère** (pas d'héritage multiple) dont elle hérite les variables et les méthodes
- Le mot clef **extends** indique la classe mère :

EX: `class RectangleColore extends Rectangle`

- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**

Polymorphisme et héritage

Exemples d'héritages

```
class A
{
}

class B extends A
{
}

class C extends A
{
}
```

Héritage à un seul niveau



```
class A
{
}

class B extends A
{
}

class C extends B
{
}
```

Héritage à plusieurs niveaux



- ❖ class Voiture extends Vehicule
- ❖ class Velo extends Vehicule
- ❖ class VTT extends Velo
- ❖ class Employe extends Personne
- ❖ class ImageGIF extends Image

Polymorphisme et héritage

Principe important lié à la notion d'héritage

- Si « **B extends A** », le grand principe est que **tout B est un A**
- Par exemple, un rectangle coloré *est un* rectangle ; un poisson *est un* animal ; *une* voiture *est un* véhicule
- Le type d'une variable détermine les données que la variable peut contenir/référencer
- **B est un sous-type de A** si on peut ranger une expression de type B dans une variable de type A
- Les sous-classes d'une classe **A** sont des **sous-types** de **A**

Par exemple: si **B hérite de A**

A a = new B(...); est autorisé

Polymorphisme et héritage

Ce que peut faire une classe fille

- La classe qui hérite peut
 - **ajouter** des variables, des méthodes et des constructeurs
 - **redéfinir** des méthodes (même signature)
 - **surcharger** des méthodes (même nom mais pas même signature)
- Mais elle ne peut retirer aucune variable ou méthode

Polymorphisme et héritage

Redéfinition et surcharge

- Une méthode **surcharge** une méthode (héritée ou définie dans la même classe) quand elle a le même nom, mais pas la même signature, que l'autre méthode

```
public class BarreDeProgression {  
    private float pourcent;  
  
    // ...  
    public void setPourcent(float valeur) {  
        pourcent = valeur;  
    }  
  
    public void setPourcent(int effectue, int total) {  
        pourcent = total / effectue;  
    }  
    // ...  
}
```


Polymorphisme et héritage

Redéfinition et surcharge

- Une méthode **redéfinit** une méthode **héritée** quand elle a la même signature que l'autre méthode

```
public class Entier {  
    private int i;  
  
    public Entier(int i) {  
        this.i = i;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (o == null || (o.getClass() != this.getClass()))  
            return false;  
        return i == ((Entier) o).i;  
    }  
}
```

Redéfinition de la méthode de la classe **Object**
« **boolean equals(Object)** »

Polymorphisme et héritage

Compléments sur les constructeurs d'une classe

- La **première** instruction d'un constructeur peut être un appel
 - à un constructeur de la classe mère :
super(...)
 - ou à un autre constructeur de la classe :
this(...)
- Interdit de placer **this()** ou **super()** ailleurs qu'en première instruction d'un constructeur

Polymorphisme et héritage

Constructeur de la classe mère

```
public class Rectangle {  
    private int x, y, largeur, hauteur;  
  
    public Rectangle(int x, int y, int largeur, int hauteur) {  
        this.x = x;  
        this.y = y;  
        this.largeur = largeur;  
    }  
    // . . .  
}
```

Constructeurs de la classe fille

```
public class RectangleColore extends Rectangle {  
  
    public RectangleColore(int x, int y, int largeur, int hauteur, Color couleur) {  
        super(x, y, largeur, hauteur);  
        this.couleur = couleur;  
    }  
  
    public RectangleColore(int x, int y, int largeur, int hauteur) {  
        this(x, y, largeur, hauteur, Color.black);  
    }  
    // . . .  
}
```

Polymorphisme et héritage

Appel implicite du constructeur de la classe mère

- Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute au début un appel implicite **super()** au **constructeur sans paramètre** de la classe mère (erreur de compilation s'il n'existe pas !)

⇒ Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

- Donc la toute, toute première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) de la classe **Object** !

C'est le seul qui sait comment créer un nouvel objet en mémoire

Polymorphisme et héritage

Complément sur le constructeur par défaut d'une classe

- Ce constructeur par défaut n'appelle pas explicitement un constructeur de la classe mère
⇒ un appel du constructeur sans paramètre de la classe mère est automatiquement effectué

Question...

```
class A {  
    private int i;  
  
    A(int i) {  
        this.i = i;  
    }  
}
```

Compile ?
S'exécute ?

```
class B extends A { }
```

Polymorphisme et héritage

Interfaces

- Une interface est une « classe » purement abstraite dont **toutes** les méthodes sont abstraites et publiques

```
public interface Figure {  
    public abstract void dessineToi();  
    public abstract void deplaceToi(int x, int y);  
}
```

```
public interface Figure {  
    void dessineToi();  
    void deplaceToi(int x, int y);  
}
```

public abstract
peut être implicite



Polymorphisme et héritage

Classe final (et autres final)

- **Classe** final : ne peut avoir de classes filles (String est final)
- **Méthode** final : ne peut être redéfinie
- **Variable** (locale ou d'état) final : la valeur ne pourra être modifiée après son initialisation
- **Paramètre** final (d'une méthode ou d'un catch) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode

Polymorphisme et héritage

- Les interfaces compensent un peu l'absence d'héritage multiple.
- Le mot clé **interface** remplace le mot clé **class** en tête de déclaration.
- Une interface ne peut contenir que des variables constantes ou statiques et des entêtes de méthodes.
- Toutes les signatures de méthodes d'une interface ont une visibilité publique.
- Le mot clé pour implémenter une interface est **implements**.
- Une classe implémentant une interface s'engage à surcharger toutes les méthodes définies dans cette interface (**contrat**).
- Une interface permet d'imposer un comportement à une classe
- Une classe peut implémenter autant d'interfaces qu'elle le souhaite.

Polymorphisme et héritage

Classe qui implémente une interface

```
public class C implements I1 { ... }
```

- 2 seuls cas possibles :
 - soit la classe **C** implémente **toutes** les méthodes de **I1**
 - soit la classe **C** doit être déclarée **abstract** ; Les méthodes manquantes seront implémentées par les classes filles de **C**

Implémentation de plusieurs interfaces

- Une classe peut implémenter une ou plusieurs interfaces (et hériter d'une classe...) :

```
public class CercleColore extends Cercle implements Figure, Coloriable { }
```

Polymorphisme et héritage

Contenu des interfaces

- Une interface ne peut contenir que
 - des méthodes **abstract et public**
 - des définitions de constantes publiques
(« **public static final** »)
- Les modificateurs **public, abstract et final** sont optionnels (en ce cas, ils sont implicites)
- Une interface ne peut contenir de méthodes **static, final, synchronized ou native**

Polymorphisme et héritage

Les interfaces comme types de données

- Une interface peut servir à déclarer une variable, un paramètre, une valeur retour, un type de base de tableau, un *cast*,...

Par exemple,

Comparable v1;

indique que la variable **v1** référencera des objets
dont la classe implémentera l'interface **Comparable**

- **InstanceOf**

Si un objet **o** est une instance d'une classe qui implémente une interface **Interface**,

o instanceof Interface est vrai

Polymorphisme et héritage

Classes abstraites

- **Définition**

Une classe abstraite est une classe dans laquelle au moins une méthode n'est pas implémentée.

- **But d'une classe abstraite**

- de fournir à d'autres développeurs une partie de l'implémentation d'une classe
- de laisser aux autres développeurs la manière d'implémenter le reste de la classe
- d'imposer aux autres développeurs d'implémenter certaines méthodes s'ils veulent pouvoir utiliser ses classes

- **Exemple:**

```
abstract public class Animal {  
    ...  
}
```

Polymorphisme et héritage

Méthodes abstraites

Deux points importants :

- Une méthode abstraite n'a pas de corps !
- Une méthode abstraite est toujours contenue dans une classe abstraite.

```
abstract public class Canin extends Animal {  
    public abstract void manger() ; // pas de corps  
}
```

- Il est **interdit** de créer une instance d'une classe abstraite
- Une méthode **static ne peut être abstraite** (car on ne peut redéfinir une méthode **static**)

Compléments sur les classes

Classes enveloppes de type primitif (Wrapper)

- En Java certaines manipulations nécessitent de travailler avec des **objets** (instances de classes) et pas avec des valeurs de **types primitifs**
- Le paquetage **java.lang** fournit des **classes** pour envelopper les types primitifs : **Byte, Short, Integer, Long, Float, Double, Boolean, Character**
- Attention, les instances de ces classes ne sont pas modifiables (idem **String**)

Compléments sur les classes

Méthodes utilitaires des classes enveloppes

- Les classes enveloppes offrent des méthodes utilitaires (le plus souvent **static**) pour faire des conversions avec les types primitifs (et avec la classe **String**)
- Elles offrent aussi des constantes, en particulier, **MAX_VALUE** et **MIN_VALUE**
- Les transparents suivants indiquent comment faire des conversions d'entiers ; il existent des méthodes similaires pour toutes types primitifs (**double par exemple**)

Compléments sur les classes

Conversions des entiers

- String → int : (méthode de Integer) static int `parseInt`(String ch)
- int → String (méthode de String) : static String `valueOf`(int i)
- int → Integer : `new Integer`(int i)
- Integer → int : int `intValue`()
- String → Integer : static Integer `valueOf`(String ch)
- Integer → String : String `toString`()

Exemple de conversion

```
/**
 * Afficher le double du nombre passé en paramètre
 */
public class AfficheParam {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        System.out.println(i * 2);
    }
}
```


Compléments sur les classes

Listes et types primitifs

- Le code est alourdi lorsqu'une manipulation nécessite d'envelopper une valeur d'un type primitif
- Ainsi on verra qu'une liste ne peut contenir de type primitif et on sera obligé d'écrire :

```
liste.add(new Integer(89));  
int i = liste.get(n).intValue();
```

Compléments sur les classes

Boxing/unboxing

- Le « *autoboxing* » (*mise en boîte*) automatise le passage des types primitifs vers les classes qui les enveloppent
- Cette mise en boîte automatique a été introduite par la version 5 du JDK
- L'opération inverse s'appelle « *unboxing* »
- Le code précédent peut maintenant s'écrire :

```
liste.add(89);  
int i = liste.get(n);
```

Compléments sur les classes

Autres exemples de boxing/unboxing

- **Integer a = 89;**
a++;
int i = a;
- **Integer b = new Integer(1);**
// unboxing suivi de boxing
b = b + 2;
- **Double d = 56.9;**
d = d / 56.9;

Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

Classes utilitaires et Collection

SOMMAIRE

- StringBuilder
- Collection
- Set
- Map
- Lst
- Itérateurs

Classes utilitaires

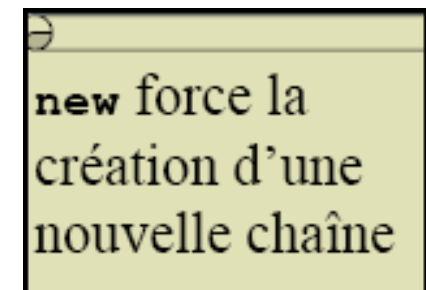
Chaînes de caractères

- 3 classes du paquetage **java.lang** :
 - **String** pour les chaînes constantes
 - **StringBuilder** ou **StringBuffer** pour les chaînes variables
- On utilise le plus souvent String, sauf si la chaîne doit être fréquemment modifiée
- Commençons par **String**

Classes utilitaires

Affectation d'une valeur littérale

- L'affectation d'une valeur littérale à un String s'effectue par :
chaîne = "Bonjour";
- La spécification de Java impose que
chaîne1 = "Bonjour";
chaîne2 = "Bonjour";
crée **un seul objet** String (référéncé par les 2 variables)
- chaîne1 = "Bonjour";
chaîne2 = **new String("Bonjour")**;
provoque la création d'une String inutile



new force la
création d'une
nouvelle chaîne

Classes utilitaires

Nouvelle affectation avec les **String**

```
String chaine = "Bonjour";  
chaine = "Hello";
```

Cet objet
String n'est
pas modifié



- La dernière instruction correspond aux étapes suivantes :
 - 1) Une nouvelle valeur (*Hello*) est créée
 - 2) La variable **chaine** référence la nouvelle chaîne *Hello (et plus l'ancienne chaîne Bonjour)*
 - 3) La place occupée par la chaîne *Bonjour* pourra être récupérée à un moment ultérieur par le ramasse-miette

Classes utilitaires

Concaténation de chaînes

```
String s = "Bonjour" + " les amis";
```

- Si un des 2 opérandes de l'opérateur **+** est une String, l'autre est traduit automatiquement en String :

```
int x = 5;  
s = "Valeur de x = " + x;
```

- les types primitifs sont traduits par le compilateur
- les instances d'une classe sont traduites en utilisant la méthode **toString()** de la classe

Classes utilitaires

Égalité de Strings

- La méthode **equals** teste si 2 instances de String contiennent la même valeur :

```
String s1, s2;  
s1 = "Bonjour " ;  
s2 = "les amis";  
if ((s1 + s2).equals("Bonjour les amis"))  
    System.out.println("Egales") ;
```

- « **==** » teste si les 2 objets ont la même adresse en mémoire ; **il ne doit pas être utilisé pour comparer 2 chaînes**, même s'il peut convenir dans des cas particuliers
- equalsIgnoreCase()** ignore la casse des lettres

Classes utilitaires

Quelques méthodes de **String**

- `substring(int début, int fin)`
- `substring(int début)`

Extraire une sous-chaîne

- `indexOf(String sousChaine)`
- `indexOf(String sousChaine, int debutRecherche)`

Rechercher
l'emplacement
d'une sous-
chaîne

- Autres : `startsWith` , `endsWith` , `trim` (enlève les espaces de début et de fin) ,
- `toUpperCase`, `toLowerCase`, `valueOf` (conversions en **String** de types primitifs, tableaux de caractères)

Classes utilitaires

Différences entre **StringBuilder** et **StringBuffer**

- StringBuilder a été introduite par le JDK 5.0
- Mêmes fonctionnalités et noms de méthodes que StringBuffer mais ne peut être utilisé que par un seul *thread*
- StringBuilder fournit de meilleures performances que StringBuffer

Classes utilitaires

String et String{Buffer|Builder}

- Utiliser plutôt la classe **String** qui possède de nombreuses méthodes
- Si la chaîne de caractères doit être souvent modifiée, passer à **StringBuilder** avec le constructeur **StringBuilder(String s)**
- Repasser de **StringBuilder** à **String** avec **toString()**

Classes utilitaires

Généralités sur les collections

Définition:

Une collection est un objet dont la principale fonctionnalité est de contenir d'autres objets, comme un tableau

- Le JDK fournit des types de collections sous la forme de classes et d'interfaces
- Ces classes et interfaces sont dans le paquetage **java.util**

Classes utilitaires

Généricité

- Avant le JDK 5.0, les objets contenus étaient déclarés de type **Object**
- A partir du JDK 5.0, on peut indiquer le type des objets contenus dans une collection grâce à la générique : **List<Employe>**
- Il est préférable d'utiliser les collections génériques

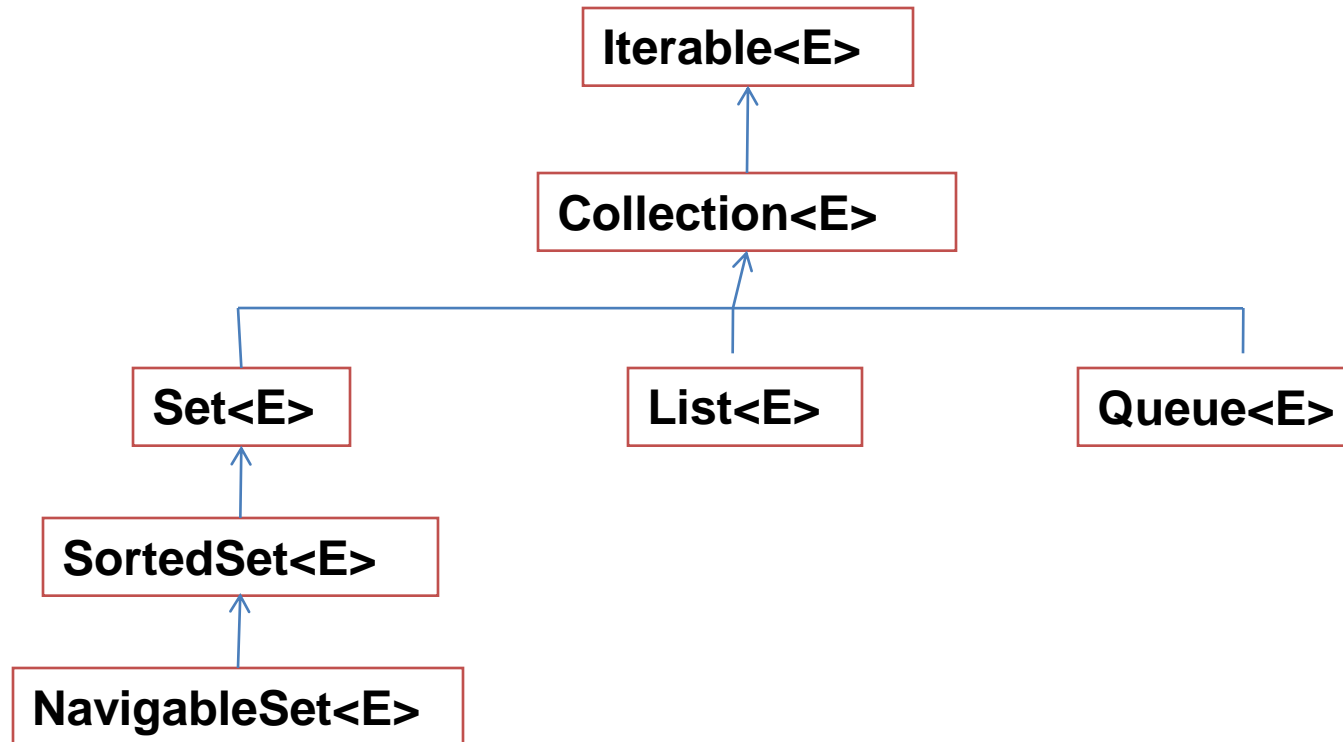
Classes utilitaires

Les interfaces

- Des interfaces dans 2 hiérarchies d'héritage principales :
 - **Collection<E>**
 - **Map<K,V>**
- **Collection** correspond aux interfaces des collections proprement dites
- **Map** correspond aux collections indexées par des clés ; un élément de type *v* d'une *map* est retrouvé rapidement si on connaît sa clé de type *K* (comme les entrées de l'index d'un livre)

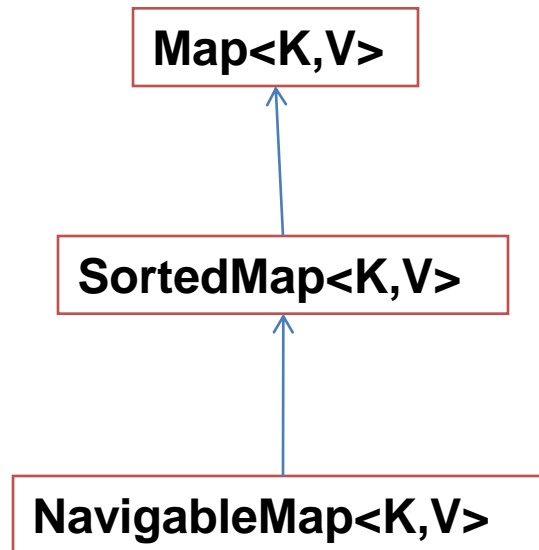
Classes utilitaires

- Hiérarchie des interfaces - **Collection**



Classes utilitaires

Hiérarchie des interfaces – Map



Classes utilitaires

Les classes abstraites

- **AbstractCollection<E>, AbstractList<E>, AbstractMap<K,V>,...** implantent les méthodes de base communes aux collections (ou *map*)
- Elles permettent de factoriser le code commun à plusieurs types de collections et à fournir une base aux classes concrètes du JDK

Les classes concrètes

- **ArrayList<E>, LinkedList<E>, HashSet<E>, TreeSet<E>, HashMap<K,V>, TreeMap<K,V>,...** héritent des classes abstraites
- Elles ajoutent les supports concrets qui vont recevoir les objets des collections (tableau, table de hachage, liste chaînée,...)
- Elles implantent ainsi les méthodes d'accès à ces objets (*get, put, add,...*)

Classes utilitaires

Classes concrètes d'implantation des interfaces

		Classes d'implantations			
		Table de hachage	Tableau	Arbre balancé	Liste chaînée
Interfaces	Set<E>	HashSet<E>		TreeSet<E>	
	List<E>		Array List<E>		LinkedList<E>
	Map<K, V>	HashMap<K, V>		TreeMap<K, V>	
	Queue<E>		Array Dequeue<E>		LinkedList<E>

- Nous étudierons essentiellement les classes **ArrayList** et **HashMap** comme classes d'implantation de **Collection** et de **Map**

Classes utilitaires

Interface List<E>

- L'interface List<E> correspond à une collection d'objets indexés par des numéros (en commençant par 0)
- Classes qui implantent cette interface :
 - ArrayList<E>, tableau à taille variable
 - LinkedList<E>, liste chaînée
- On utilise le plus souvent ArrayList, sauf si les insertions/suppressions au milieu de la liste sont fréquentes (LinkedList évite les décalages des valeurs)

Classes utilitaires

Nouvelles méthodes de List<E>

- void **add**(int indice, E elt)
- boolean **addAll**(int indice, Collection<? extends E> c)
- E **get**(int indice)
- E **set**(int indice, E elt)
- E **remove**(int indice)
- int **indexOf**(Object obj)
- int **lastIndexOf**(Object obj)
- ListIterator<E> listIterator()
- ListIterator<E> listIterator(int indice)
- List<E> subList(int depuis, int jusquà)

insertion avec décalage
vers la droite

suppression avec
décalage vers
la gauche

indice du 1er élément
égal à **obj** (au sens de
equals) (ou -1)

depuis « inclus »,
jusquà « exclu »

Classes utilitaires

Exemple d'utilisation de **ArrayList**

```
List<Employe> le = new ArrayList<Employe>();  
Employe e = new Employe("Dupond");  
le.add(e);  
// Ajoute d'autres employés  
// . . .  
// Affiche les noms des employés  
for (int i = 0; i < le.size(); i++) {  
    System.out.println(le.get(i).getNom());  
}
```

Classes utilitaires

Itérateurs (Iterator<E>)

- Un itérateur (instance d'une classe qui implante l'interface Iterator<E>) permet d'énumérer les éléments contenus dans une collection
- Toutes les collections ont une méthode **iterator()** qui renvoie un itérateur
- Méthodes de l'interface **Iterator<E>**
 - boolean hasNext()
 - E next()
 - remove() enlève le dernier élément récupéré (elle est optionnelle)

Classes utilitaires

Obtenir un itérateur

- L'interface `Collection<E>` contient la méthode

`Iterator<E> iterator()`

qui renvoie un itérateur pour parcourir les éléments de la collection

Exemple d'utilisation de Iterator

```
List<Employe> le = new ArrayList<Employe>();
Employe e = new Employe("Dupond");
le.add(e);
// Ajoute d'autres employés dans le
// . . .
Iterator<Employe> it = le.iterator();
while (it.hasNext()) {
    // le 1er next() fournit le 1er élément
    System.out.println(it.next().getNom());
}
```

Classes utilitaires

Interface **Iterable<T>**

- Nouvelle interface (depuis JDK 5.0) du paquetage java.lang qui indique qu'un objet peut être parcouru par un itérateur
- Toute classe qui implémente Iterable peut être parcourue par une boucle « for each »
- L'interface Collection en hérite

Boucle « normale » :

```
List<Employee> coll = new ArrayList<Employee>();  
for (Iterator<Employee> it = coll.iterator(); it.hasNext();) {  
    Employee e = it.next();  
  
}
```

Boucle « for each »

```
List<Employee> coll = new ArrayList<Employee>();  
for (Employee e : coll) {  
    String nom = e.getNom();  
}
```

Classes utilitaires

Interface Set<E>

- Correspond à une collection qui **ne contient pas 2 objets égaux au sens de equals** (comme les ensembles des mathématiques)
- Éviter de modifier les objets d'un **Set**
Le comportement du Set peut être altéré si un objet placé dans un Set est **modifié** d'une manière qui affecte la valeur renvoyée par equals
- Classes qui implémentent cette interface :
 - **HashSet<E>** implémente Set avec une table de hachage ; temps constant pour les opérations de base (set, add, remove, size)
 - **TreeSet<E>** implémente NavigableSet avec un arbre ordonné

Classes utilitaires

Exemple d'utilisation de **Set<E>**

```
// Create the set
Set<String> col = new HashSet<String>();

// Add elements to the set
col.add("a");
col.add("b");

// Remove elements from the set
col.remove("b");

// Iterating over the elements in the set
for (String al: col) {
    System.out.println(al);
}
```

Classes utilitaires

Interface Map<K,V>

Définition

- L'interface **Map<K,V>** correspond à un groupe de couples clé-valeur
- Une clé repère une et une seule valeur
- Dans la map il ne peut exister 2 clés égales au sens de **equals()**

Fonctionnalités

- ajouter et enlever des couples clé – valeur
- récupérer une référence à une des valeurs en donnant sa clé
- savoir si une table contient une valeur
- savoir si une table contient une clé

Classes utilitaires

Implémentations

- `HashMap<K,V>`, table de hachage ;
- `TreeMap<K,V>`, arbre ordonné suivant les valeurs des clés

Exemple d'utilisation de `HashMap`

```
Map<String,Employee> hm = new HashMap<>();
Employee e = new Employee("Dupond");
e.setMatricule("E125");
hm.put(e.getMatricule(), e);
// Crée et ajoute les autres employés dans la table de hachage
. . .
Employee e2 = hm.get("E369");
Collection<Employee> employees = hm.values();
for (Employee employee : employees) {
    System.out.println(employee.getNom());
}
```

Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

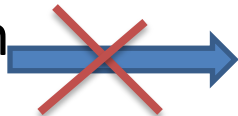
Gestion des exceptions

SOMMAIRE

- Concepts
- Classification des exceptions
- Mise en oeuvre de la gestion d'exceptions
- Levée d'exception
- Traitement d'exception
- Classes et sous-classes d'exception
- Relancer une exception

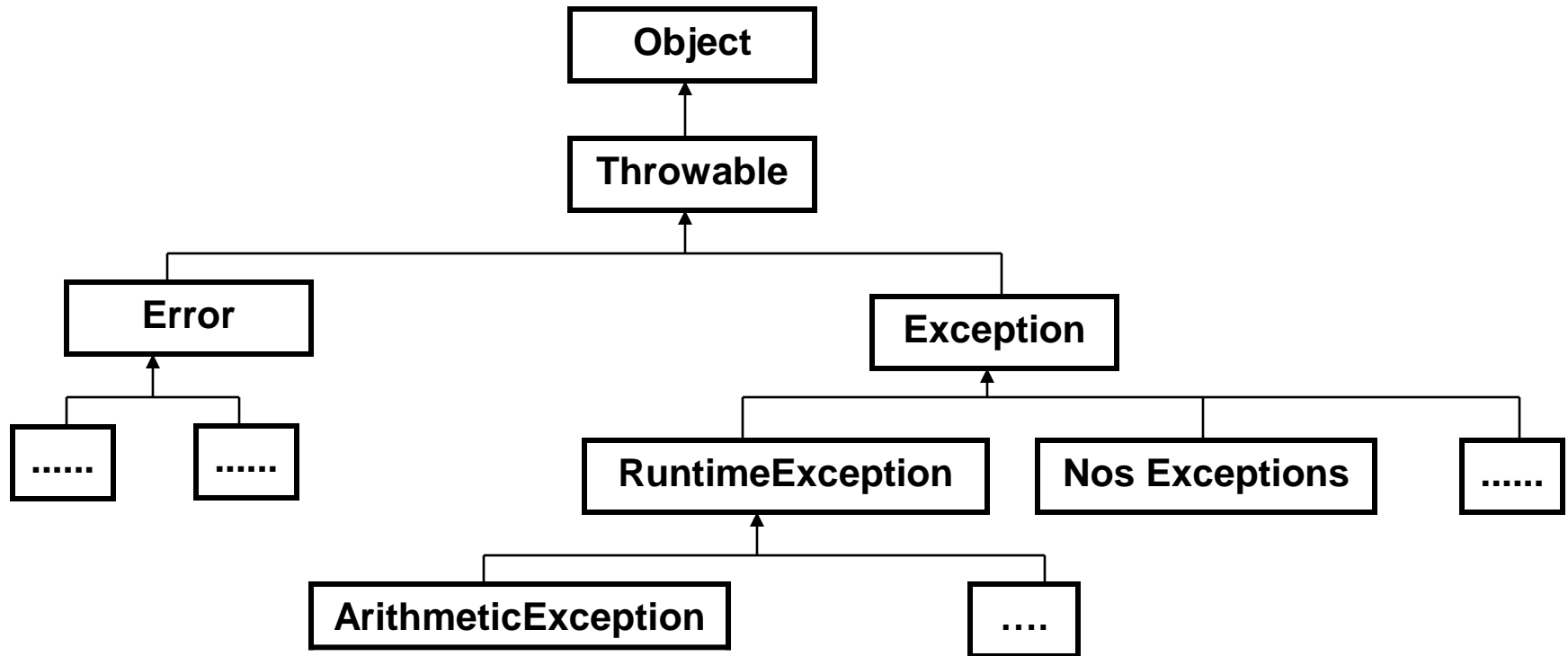
Gestion des exceptions

Concepts

- Une exception représente une erreur.
- Une **exception** est un signal qui se déclenche en cas de problème.
- Il est possible de lancer une exception pour signaler une erreur.
- Lancer une exception  Interrompre un programme
- le programmeur peut gérer les erreurs sans que le programme ne s'arrête définitivement
- La gestion des exceptions se décompose en deux phases :
 - La levée d'exceptions,
 - Le traitement d'exceptions.

Gestion des exceptions

Classification des exceptions



En Java, une exception est représentée par une classe.

Gestion des exceptions

Mise en oeuvre de la gestion d'exceptions

- Vous pouvez mettre en oeuvre la gestion d'exceptions dans un programme à l'aide des mots clés suivants :

➤ try

➤ catch

➤ throw

➤ throws

➤ finally

Gestion des exceptions

Levée d'exception

- Une exception est levée grâce à l'instruction *throw* :
- Une exception peut être :

```
if (k<0)
    throw new EstNegatifException("Message");
```

- traitée directement par la méthode dans laquelle elle est levée
- être envoyée à la méthode appelante grâce à l'instruction *throws* (à ne pas confondre avec *throw*)

```
public void maMethode(int entier) throws IOException
{
    //code de la methode
}
```

si une exception de type *IOException* est levée durant l'exécution de *maMethode*, l'exception sera envoyée à la méthode appelant *maMethode*, qui devra la traiter.

Gestion des exceptions

Levée d'exception

- Certaines exceptions sont levées implicitement par la machine virtuelle :
 - `NullPointerException` quand une référence nulle est déréférencée (accès à un membre),
 - `ArrayIndexOutOfBoundsException` quand l'indice d'un tableau dépasse sa capacité,
 - `ArithmeticException` quand une division par zéro a lieu.
- Celles-ci n'ont pas besoin d'être déclarées avec l'instruction *throws* car elles dérivent de la classe `RuntimeException`.

Gestion des exceptions

Traitement d'exception

- Le traitement des exceptions se fait à l'aide de la séquence d'instructions *try...catch...finally*.

```
try{  
    // instructions susceptible de lever des exceptions.  
}catch(ExceptionName obj){  
    // traitement pour un type particulier d'exceptions  
}  
finally{  
    // sert à définir un bloc de code à exécuter dans tous les cas  
}
```

- Il peut y avoir plusieurs instructions *catch* pour une même instruction *try*.
- Il faut au moins une instruction *catch* ou *finally* pour chaque instruction *try*.

Gestion des exceptions

Exemple :

```
public String lire(String nomDeFichier) throws IOException
{
    try
    {
        // La ligne suivante est susceptible de lever une exception
        // de type FileNotFoundException
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        // Cette ligne est susceptible de lever une exception
        // de type IOException
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (FileNotFoundException fnfe)
    {
        fnfe.printStackTrace(); // Indique l'exception sur le flux d'erreur standard
    }
    finally
    {
        System.err.println("Fin de méthode");
    }
}
```

Gestion des exceptions

Classes et sous-classes d'exception

- **L'héritage entre les classes d'exceptions** peut conduire à des erreurs de programmation.
- En effet, une instance d'une sous-classe est également considérée comme une instance de la classe de base.
- **L'ordre des blocs catch est important** : il faut placer les sous-classes avant leur classe de base.
- Dans le cas contraire le compilateur génère l'erreur exception *classe_exception* has already been caught.

Gestion des exceptions

Exemple d'ordre incorrect :

```
try{
    FileReader lecteur = new FileReader(nomDeFichier);
}
catch(IOException ioex) // capture IOException et ses sous-classes
{
    System.err.println("IOException caught :");
    ioex.printStackTrace();
}
catch(FileNotFoundException fnfex) // <-- erreur ici
// FileNotFoundException déjà capturé par catch(IOException ioex)
{
    System.err.println("FileNotFoundException caught :");
    fnfex.printStackTrace();
}
```

Gestion des exceptions

L'ordre correct est le suivant :

```
try{
    FileReader lecteur = new FileReader(nomDeFichier);
}
catch(FileNotFoundException fnfex)
{
    System.err.println("FileNotFoundException caught :");
    fnfex.printStackTrace();
}
catch(IOException ioex) // capture IOException et ses autres sous-classes
{
    System.err.println("IOException caught :");
    ioex.printStackTrace();
}
```

Gestion des exceptions

Sous-classes et clause throws

- Une autre source de problèmes avec les sous-classes d'exception est la clause throws.
- Ce problème n'est pas détecté à la compilation.

Exemple :

```
public String lire(String nomDeFichier) throws FileNotFoundException
{
    try
    {
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (IOException ioe) // capture IOException et ses sous-classes
    {
        ioe.printStackTrace();
    }
}
```

Cette méthode ne lancera jamais d'exception de type `FileNotFoundException` car cette sous-classe de `IOException` est déjà capturée.

Gestion des exceptions

Relancer une exception

- Une exception peut être partiellement traitée, puis relancée.
- On peut aussi relancer une exception d'un autre type, cette dernière ayant l'exception originale comme cause.
- Dans le cas où l'exception est partiellement traitée avant propagation, la relancer consiste simplement à utiliser l'instruction `throw` avec l'objet exception que l'on a capturé.

Gestion des exceptions

Exemple:

```
public String lire(String nomDeFichier) throws IOException
{
    try
    {
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (IOException ioException) // capture IOException et ses sous-classes
    {
        // ... traitement partiel de l'exception ...
        throw ioException; //<-- relance l'exception
    }
}
```

Gestion des exceptions

Relancer une exception -2

- Une exception d'un autre type peut être levée.
- Par exemple pour ne pas propager une exception de type `SQLException` à la couche métier, tout en continuant à arrêter l'exécution normale du programme :

```
...  
    catch (SQLException sqlException) // capture SQLException et ses sous-classes  
    {  
        throw new RuntimeException("Erreur (base de données)...", sqlException);  
    }  
...
```

Gestion des exceptions

Définir sa propre exception

- Si on veut pouvoir traiter, un événement exceptionnel, d'un type non prévu par l'[API](#), **il faut définir une nouvelle classe** étendant la classe `java.lang.Exception` ;
- La classe étendue ne contient en général pas d'autre champ qu'un (ou plusieurs, ou zéro) constructeur(s) et une redéfinition de la méthode `toString`.
- Lors du lancement de l'exception, (à l'aide du mot réservé **throw**), on crée une instance de la classe définie.

Gestion des exceptions

Exemple:

```
public class ExceptionRien extends Exception {
    int nbChaines;
    public ExceptionRien(int nombre) {
        nbChaines = nombre;
    }

    public String toString() {
        return "ExceptionRien : aucune des " + nbChaines + " chaines n'est valide";
    }
}
```


Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

Package IO

SOMMAIRE

- Gestion de Fichiers
- Notion de Flux
- Flux de Caractères
- Flux d'Octets

Package IO

Gestion de Fichiers:

- La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe **File**.
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.
- Un objet de la classe File peut représenter un fichier ou un répertoire.

Package IO

- Voici un aperçu de quelques constructeurs et méthodes de la classe File :
 - **File** (String name)
 - **File** (String path, String name)
 - **File** (File dir, String name)
 - boolean **isFile**() / boolean **isDirectory**()
 - boolean **mkdir**()
 - boolean **exists**()
 - boolean **delete**()
 - boolean **canWrite**() / boolean **canRead**()
 - File **getParentFile**()

Package IO

Exemple de gestion de fichier :

```
import java.io.*;
public class Listeur
{
    public static void main(String[] args)
    {
        litrep(new File("."));
    }
    public static void litrep(File rep)
    {
        if (rep.isDirectory())
        { //liste les fichier du répertoire
            String t[]=rep.list();
            for (int i=0;i<t.length;i++)
                System.out.println(t[i]);
        }
    }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est un fichier ou un répertoire

Package IO

Notion de Flux

- Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).
- Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.
- Un flux peut être :
 - Soit une **source** d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
 - Soit une **destination** d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.

Package IO

Notion de Flux

- Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :
 - les **fichiers**,
 - les **tableaux de données** en mémoire,
 - les **lignes de communication** (connexion réseau)
- L'intérêt de la notion de flux est qu'elle permet une gestion homogène :
 - quelle que soit la ressource associée au flux de données,
 - quel que soit le flux (entrée ou sortie).
- Certains flux peuvent être associés à des **filtres**
 - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

Package IO

- Les flux sont regroupés dans le paquetage **java.io**
- Il existe de nombreuses classes représentant les flux
 - *il n'est pas toujours aisé de se repérer.*
- Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :
 - E / S bufferisées, traduction de données, ...
- Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.

Package IO

Flux de Caractères et Flux d'Octets

- Il existe des flux de bas niveau et des flux de plus haut niveau (travaillant sur des données plus évoluées que les simples octets).
- Citons :

☐ Les flux de caractères

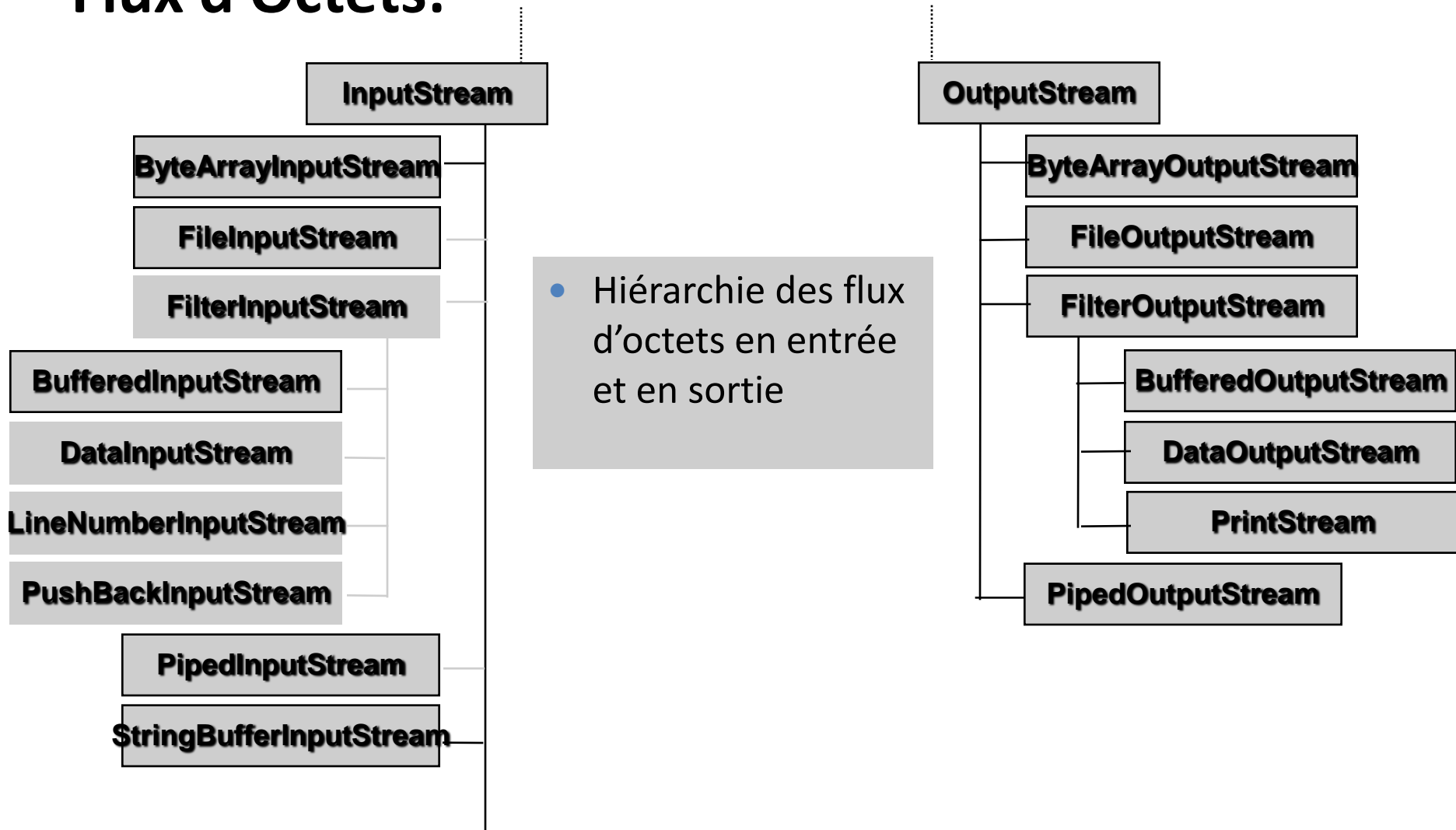
- classes abstraites Reader et Writer et leurs sous-classes concrètes respectives.

☐ Les flux d'octets

- classes abstraites InputStream et OutputStream et leurs sous-classes concrètes respectives,

Package IO

Flux d'Octets:



Package IO

Flux d'Octets / InputStream:

- Un InputStream est un flux de lecture d'octets.
- Les méthodes principales sur un InputStream sont :
 - ❑ **public abstract int read () throws IOException** qui retourne l'octet lu ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.
 - ❑ **int read (byte[] b)** qui emplit un tableau d'octets et retourne le nombre d'octets lus
 - ❑ **void close ()** qui permet de fermer un flux,
 - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.

Package IO

Flux d'Octets / OutputStream:

- Un OutputStream est un flux d'écriture d'octets.
- Les méthodes principales qui peuvent être utilisées sur un OutputStream sont :
 - **public abstract void write (int) throws IOException** qui écrit l'octet passé en paramètre,
 - **void write (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
 - **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
 - **flush ()** qui permet de purger le tampon en cas d'écritures bufferisées

Package IO

Flux d'Octets / Flux de données prédéfinis

- Il existe 3 flux prédéfinis :
 - l'entrée standard **System.in** (instance de InputStream)
 - la sortie standard **System.out** (instance de PrintStream)
 - la sortie standard d'erreurs **System.err**(instance de PrintStream)

Package IO

Flux d'Octets / Flux de données prédéfinis:

Exemple de flux prédéfini (la classe `InputStream` ne propose que des méthodes élémentaires):

```
import java.io.*;  
public class Saisie {  
  
    static public void main (String [] args) throws IOException {  
        InputStream in = System.in;  
        PrintStream out = System.out;  
        out.println ("Saisie : ");  
        for (int i=0; i<5; i++) {  
            // Lecture à partir du clavier  
            int x = in.read ();  
            // Ecriture sur écran  
            out.write (Character.toUpperCase((char)x));  
        }  
        out.println ();  
    }  
}
```

Package IO

Flux De Caractères / Reader et Writer:

Reader est un flux de lecture de caractères et Writer est un flux d'écriture de caractères.

Ces flux utilisent le codage de caractères Unicode.

Les méthodes principales qui peuvent être utilisées sur un Reader et un Writer sont :

- **public abstract int read(char c[]) throws IOException;**
qui emplit un tableau de caractères et retourne le nombre de caractères lus.
- **void close ();** qui permet de fermer un flux,
- **public abstract int write(char c[]) throws IOException;**
qui écrit les octets lus depuis un tableau de caractères.

Package IO

Flux De Caractères / Reader et Writer:

- Pour écrire des chaînes de caractères et des nombres sous forme de texte
 - on utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.
- Pour lire des chaînes de caractères sous forme texte, il faut utiliser, par exemple,
 - **BufferedReader** qui possède une méthode **readLine()** .
 - Pour la lecture de nombres sous forme de texte, il n'existe pas de solution toute faite : il faut par exemple passer par des chaînes de caractères et les convertir en nombres.

Package IO

Flux Caractères / Flux de données prédéfinis:

La classe **BufferedReader** ici, va permettre de récupérer des chaînes de caractères à partir du clavier.

```
try {  
    Reader reader = new InputStreamReader(System.in);  
    BufferedReader keyboard = new BufferedReader(reader);  
  
    System.out.print("Entrez une ligne de texte : ");  
    String line = keyboard.readLine();  
    System.out.println("Vous avez saisi : " + line);  
}  
catch(IOException e) {  
    System.out.print(e); }
```

Package IO

Empilement de Flux Filtrés

Exemple d'empilement de flux filtrés :

```
import java.io.*;  
public class CatFile {  
    public static void main(String args[]) throws IOException {  
        FileReader in;  
        String line;  
        in = new FileReader(args[0]);  
        BufferedReader dataIn = new BufferedReader(in);  
        while ((line = dataIn.readLine()) != null)  
            System.out.println(line);  
        in.close();  
    }  
}
```

Package IO

Empilement de Flux Filtrés

Exemple d'empilement de flux filtrés :

```
import java.io.*;
public class Ecrire
{
    public static void main(String[] args)
    {
        try
        {
            FileWriter fw=new FileWriter("c:\\temp\\essai.txt");
            BufferedWriter bw= new BufferedWriter(fw);
            bw.write("Ceci est mon fichier");
            bw.newLine();
            bw.write("Il est à moi...");
            bw.close();
        }
        catch (Exception e)
        { System.out.println("Erreur "+e);}
    }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileWriter puis à partir de celui-ci, on crée un BufferedWriter

Attention, lorsque l'on a écrit, il ne faut pas oublier de fermer le fichier

Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

Accès aux données

SOMMAIRE

- Objectifs de JDBC
- Architecture
- Principe de fonctionnement
- Drivers JDBC
- L'API JDBC
- Mise en oeuvre de JDBC

Accès aux données

Objectifs de JDBC

- Permettre aux programmeurs Java d'écrire un ***code indépendant de la base de données*** et du moyen de connexion utilisé
- API JDBC (*Java DataBase Connectivity*) 3.0
 - interface uniforme permettant un accès homogène aux SGBD
 - simple à mettre en oeuvre
 - indépendant du SGBD support
 - supportant les fonctionnalités de base du langage SQL

Atouts

- portabilité sur de nombreux OS et sur de nombreux SGBDR
- Uniformité des accès aux bases de données
- liberté totale vis-à-vis des constructeurs (Oracle, Informix, Sybase, ..)

Accès aux données

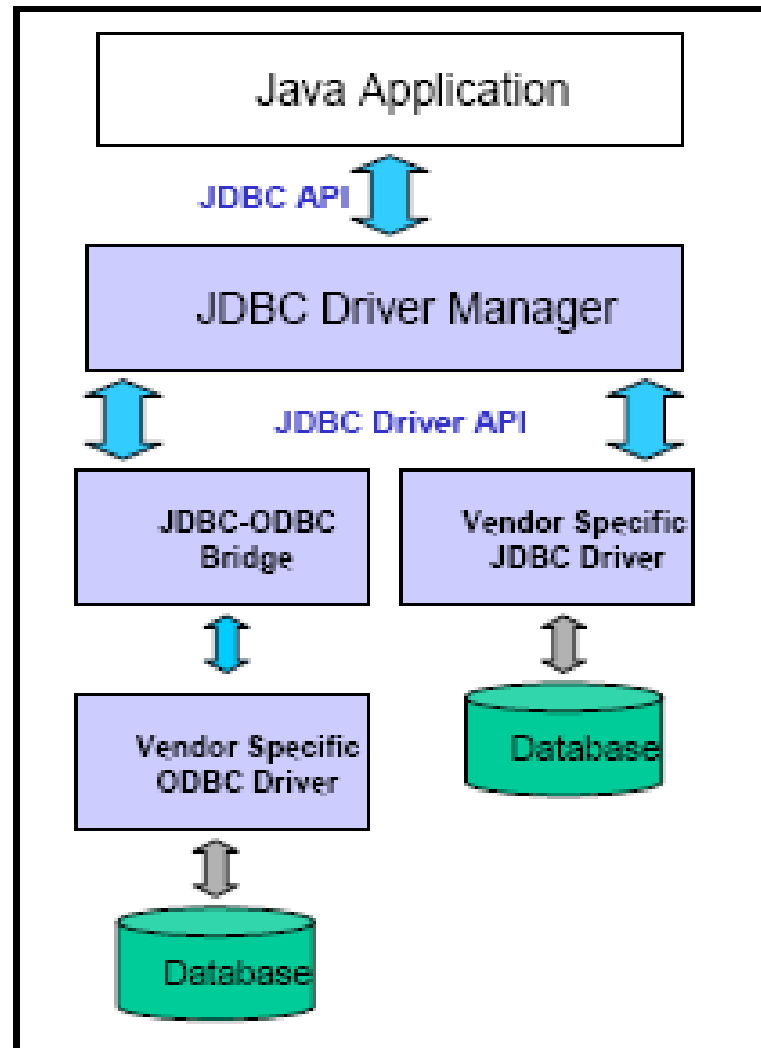
C'est quoi JDBC

- Un package contenant
 - un ensemble de classes et d'interfaces
 - pour écrire des requêtes destinées aux SGBD (SQL)

Attention: JDBC ne fournit pas les classes qui implantent les interfaces
- JDBC est composé de deux parties(un modèle à deux niveaux):
 - JDBC API
 - c'est la couche visible et utile pour développer des applications Java accédant à des SGBD
 - JDBC DriverManager
 - Communique avec le driver spécifique à une base de donnée

Accès aux données

Architecture



Accès aux données

Principe de fonctionnement

- Drivers
 - chaque SGBD utilise un pilote (driver) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBD
 - le driver est un ensemble de classes qui implantent les interfaces de JDBC
 - les drivers font le lien entre le programme Java et le SGBD
 - ces drivers dits JDBC existent pour tous les principaux SGBD: Oracle, Sybase, Informix, DB2, MySQL,...

Accès aux données

Drivers JDBC

4 types de drivers

- **Type I** : *JDBC-ODBC bridge driver*
 - pont JDBC-ODBC
- **Type II** : *Native-API, partly-Java driver*
 - driver faisant appel à des fonctions natives non Java de l'API du SGBD
- **Type III** : *Net-protocol, all-Java driver*
 - driver qui permet l'utilisation d'un middleware
- **Type IV** : *Native-protocol, all-Java driver*
 - driver écrit entièrement en Java qui utilise le protocole réseau du SGBD

Accès aux données

L'API JDBC est contenue dans les packages

- java.sql (JSE)
- javax.sql (JEE)

L'API Java.sql

fournit toutes les fonctionnalités primaires pour l'accès aux BDD

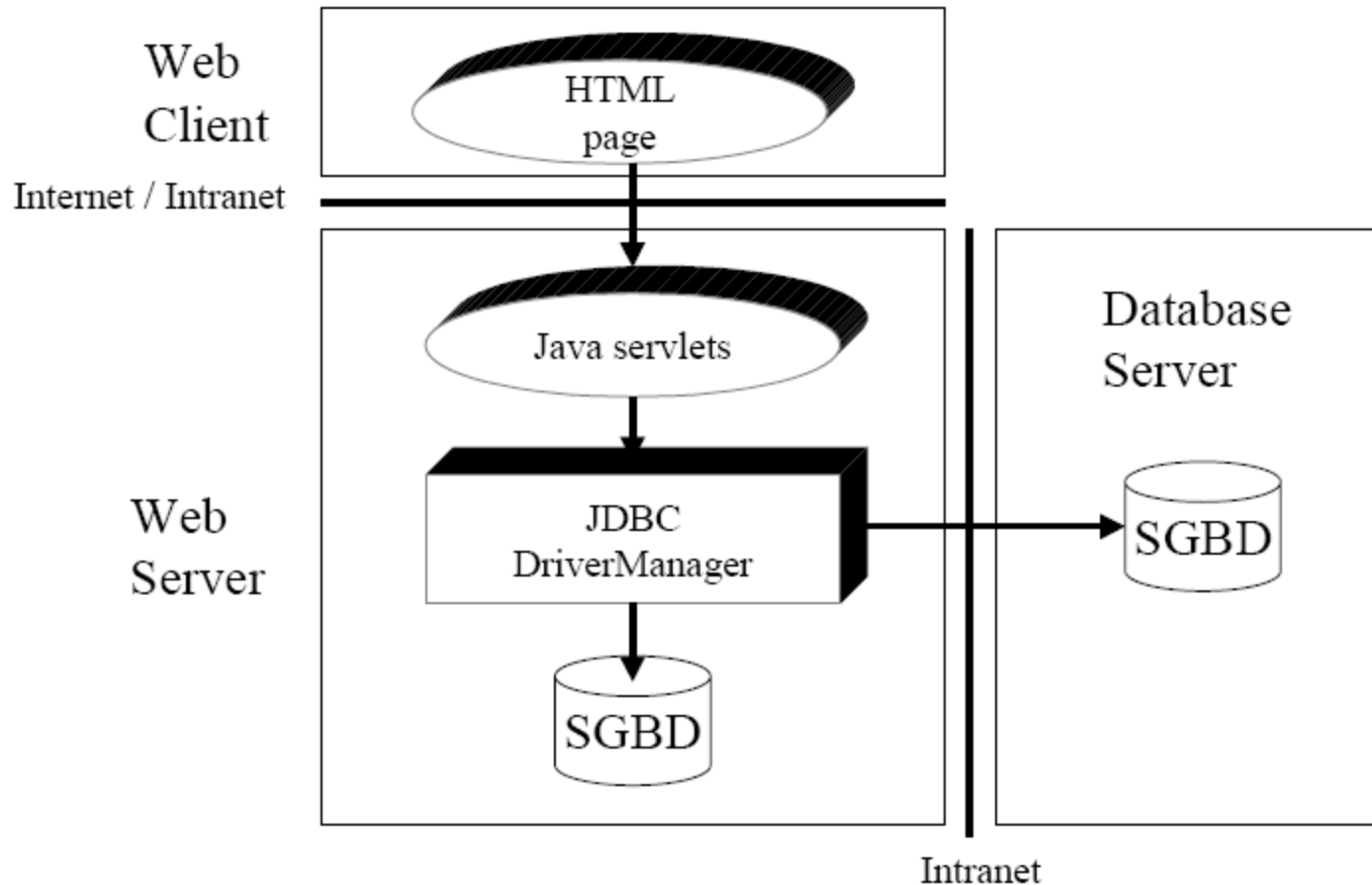
- Connexion à la base
- Envoi de résultat de l'exécution des requêtes
- Lecture des requêtes
- Traitement des méta-informations (structures de la base)

Accès aux données

- Javax.sql fournit des fonctionnalités liées à la vocation « entreprise » des applications :
 - Accès aux données étendu à toutes les sources de données, représentées par l'objet DataSource
 - L'objet DataSource est créé à l'aide d'un service d'annuaire JNDI (Java Naming Directory Interface) qui contient toutes les informations nécessaires à l'établissement d'une connexion
 - L'objet DataSource gère un pool de connexions permettant d'optimiser les temps de traitement des opérations

Accès aux données

Exemple:



Accès aux données

java.sql

8 interfaces :

- **Driver**: renvoie une instance de Connection
- **Connection**: connexion à une base
- **Statement**: instruction SQL
- **PreparedStatement**: instruction SQL paramétrée
- **CallableStatement**: procédure stockée dans la base
- **ResultSet**: n-uplets récupérés par une instruction SQL
- **ResultSetMetaData**: description des n-uplets récupérés
- **DatabaseMetaData**: informations sur la base de données

Accès aux données

Mise en oeuvre de JDBC

1. Charger un pilote
2. Créer une connexion à la base
3. Créer une requête (Statement)
4. Exécuter une requête
5. Traiter les données retournées
6. Fermer la connection

Accès aux données

Première étape "*Charger un pilote*"

- Les pilotes sont chargés dynamiquement par la méthode statique *forName()* de la classe Class
- Initialisation dans le programme

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
} catch(ClassNotFoundException ex) {...}
```

nom-de-classe(Driver)

A blue arrow points from the text 'nom-de-classe(Driver)' in the diagram below to the string 'com.mysql.jdbc.Driver' in the code block above.

Accès aux données

Seconde étape "**Créer une connexion à la base**" -1

- Demande de connexion grâce méthode statique *getConnection(String)* de la classe *DriverManager*
- Structure de la chaîne décrivant la connexion

jdbc:protocole:URL

- Exemples

jdbc:odbc:epicerie

jdbc:mysql://localhost:3306/db

jdbc:oracle:thin:@blabla:1715:test

Accès aux données

Seconde étape "*Créer une connexion à la base*" -2

- Connexion sans information de sécurité

```
Connection con = DriverManager.getConnection  
("jdbc:odbc:epicerie");
```

- Connexion avec informations de sécurité

```
Connection con = DriverManager.getConnection  
("jdbc:odbc:epicerie", user, password);
```

- Dans tous les cas faut récupérer l'exception

```
java.sql.SQLException
```

Accès aux données

Troisième étape "*Créer une requête*" -1

- L'interface **Statement** possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend
- 3 types de Statement :
 - **Statement** : requêtes statiques simples
 - **PreparedStatement** : requêtes dynamiques précompilées (avec paramètres d'entrée/sortie)
 - **CallableStatement** : procédures stockées

Accès aux données

Troisième étape "*Créer une requête*" -2

- À partir de l'instance de l'objet Connection, on récupère le Statement associé

Exemple:

- `Statement req1 = con.createStatement(str);`
- `PreparedStatement req2 = con.prepareStatement(str);`
- `CallableStatement req3 = con.prepareCall(str);`

Exemple: str= select * from table

Accès aux données

Quatrième étape "Exécuter une requête" - 1

3 types d'exécution

- **consultation** (requêtes de type SELECT)
`executeQuery()` : retourne un ResultSet (n-uplets résultants)
- **modification** (requêtes de type INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE)
`executeUpdate()` : retourne un entier (nombre de n-uplets traités)
- **nature inconnue**, plusieurs résultats ou procédures stockées
`execute()`

Accès aux données

Quatrième étape "Exécuter une requête" - 2

- `executeQuery()` et `executeUpdate()` de la classe `Statement` prennent comme argument une chaîne (`String`) indiquant la requête SQL à exécuter

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT nom, prenom FROM clients " +
    "WHERE nom='perrin' ORDER BY prenom");
int nb = st.executeUpdate("INSERT INTO dept(DEPT) "+ "VALUES(54)");
```

Attention aux espaces

Accès aux données

Quatrième étape "Exécuter une requête" - 3

Deux remarques :

- le code SQL n'est pas interprété par Java.
 - c'est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL
 - si une requête ne peut s'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l'exception *SQLException* est levée
- le driver JDBC effectue d'abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l'exécuter.

Accès aux données

Cinquième étape "Traiter les données retournées"

Interface ResultSet

- `executeQuery()` renvoie une instance de `ResultSet` qui permet d'accéder aux champs des n-uplets sélectionnés
- seules les données demandées sont transférées en mémoire par le driver JDBC
- il faut donc les lire "**manuellement**" et les stocker dans des variables pour un usage ultérieur

Accès aux données

Résultat avec ResultSet (1)

- Parcours itératif ligne par ligne
 - Méthode `next()`
 - retourne false si dernier n-uplet lu, true sinon
 - un appel fait avancer le curseur sur le n-uplet suivant
 - au départ, le curseur est positionné avant le premier n-uplet
 - exécuter `next()` au moins une fois pour avoir le premier
- ```
while(rs.next()) {
 // Traitement de chaque n-uplet
}
```
- Impossible de revenir au n-uplet précédent ou de parcourir l'ensemble dans un ordre non séquentiel

# Accès aux données

## Résultat avec ResultSet (2)

- Les colonnes sont référencées par leur **numéro** (commencent à 1) ou par leur **nom**
- L'accès aux valeurs des colonnes se fait grâce à l'utilisation de méthodes de la forme **getXXX()**
  - lecture du type de données Java XXX dans chaque colonne du n-uplet courant

**int val = rs.getInt(3) ; // accès au 3e attribut**

**String prod = rs.getString("PRODUIT") ;**

# Accès aux données

## Types de données JDBC/SQL

- Tous les SGBD n'ont pas les mêmes types SQL (même pour les types de base, il peut y avoir des différences importantes)
  - 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

# Accès aux données

## Équivalence de types entre Java et SQL

| • Type JDBC/SQL (classe Type) | Méthode Java      |
|-------------------------------|-------------------|
| CHAR, VARCHAR                 | getString()       |
| LONGVARCHAR                   | getAsciiStream()  |
| NUMERIC, DECIMAL              | getBigDecimal()   |
| BINARY, VARBINARY             | getBytes()        |
| LONGVARBINARY                 | getBinaryStream() |
| BIT                           | getBoolean()      |
| INTEGER                       | getInt()          |
| BIGINT                        | getLong()         |
| SMALLINT                      | getShort()        |
| TINYINT                       | getByte()         |
| REAL                          | getFloat()        |
| DOUBLE, FLOAT                 | getDouble()       |
| DATE                          | getDate()         |
| TIME                          | getTime()         |
| TIME STAMP                    | getTimeStamp()    |

| • Type JDBC/SQL (classe Type) | Méthode Java |
|-------------------------------|--------------|
| ARRAY                         | getArray()   |
| BLOB                          | getBlob()    |
| CLOB                          | getClob()    |
| REF                           | getRef()     |
| AUTRE                         | getObject()  |

# Accès aux données

## Fermer la connection

- Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts
  - sinon le ramasse-miettes s'en occupera mais moins efficace
- Chaque objet possède une méthode `close()` :  
`resultset.close();`  
`statement.close();`  
`connection.close();`

# Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit

# Gestion des logs avec Log4j

## SOMMAIRE

- Objectifs
- Principe
- Configuration de Log4J
- Loggers
- Appenders
- Layouts
- Les niveaux de journalisation
- Outils

# Gestion des logs avec Log4j

## Objectifs

Être une solution «propre»de logging:

- Remplace les System.err ou System.out

Avantages d'une API de logging:

- Activer/Désactiver la capture d'information pour la rediriger dans une console ou un fichier.
- Produire des rapports formatés de logs
- Catégoriser les logs, i.e. sélectionner ce que l'on veut logger.
- Définir des niveaux de log
- **Ne jamais toucher au code!**

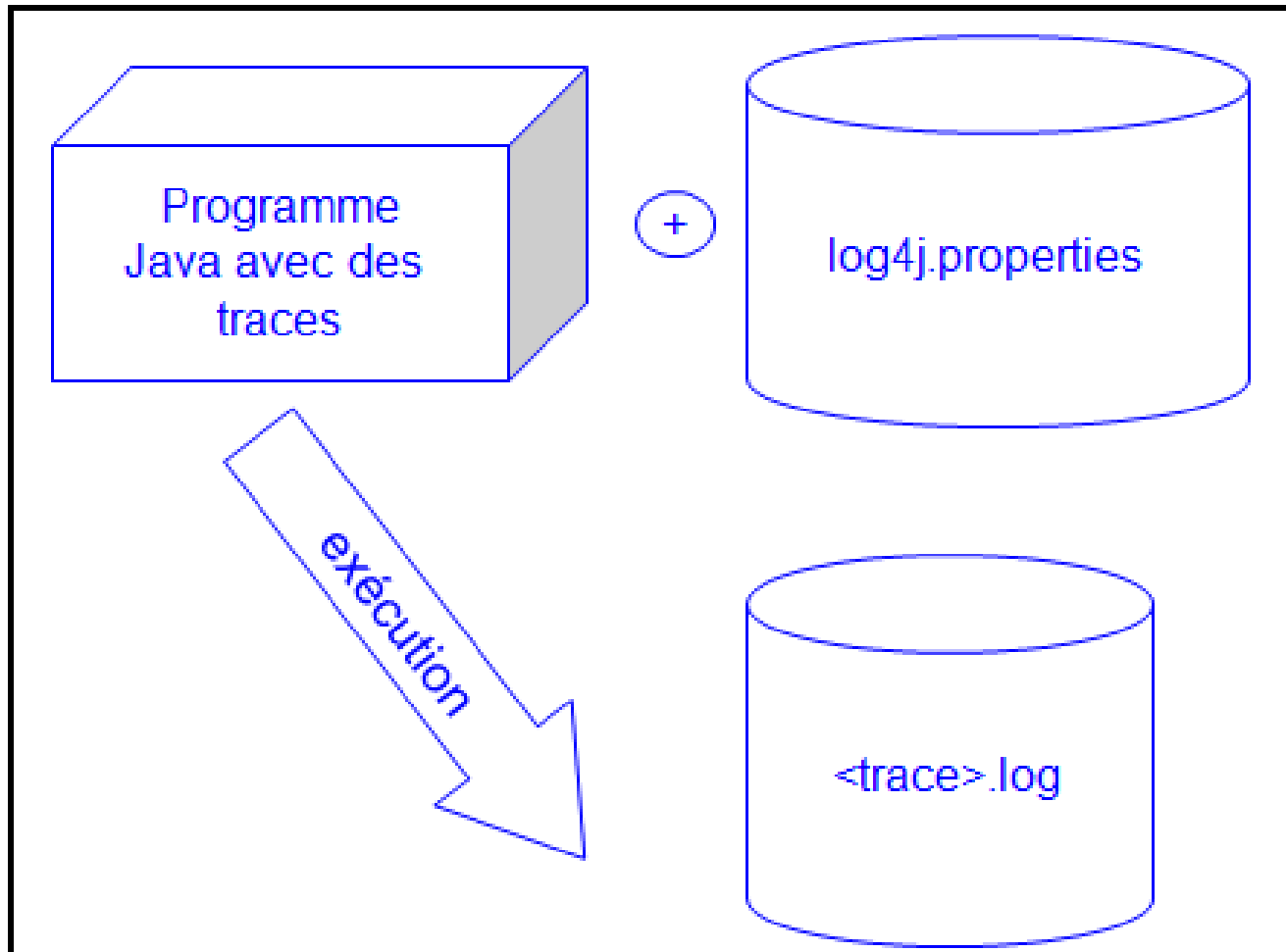
D'autres API de logs:

- JDK depuis 1.4,
- commons-logging(Factory: log4j, commons, jdk)



# Gestion des logs avec Log4j

## Principe



# Gestion des logs avec Log4j

- Projet de la fondation Apache
- Il est distribué en Open Source
- Les premières versions sont apparue en 1996.
- Implémentations
  - log4j : Java
  - log4cxx : C++
  - log4net : .Net
  - log4php : PHP
- Documentation

<http://logging.apache.org/>

# Gestion des logs avec Log4j

## Configuration de Log4J

- Créer et configurer un fichier **log4j.properties** ou **log4j.xml**
  - peut être rechargé par l'application
- **Note:** log4j.xml prime sur log4j.properties

# Gestion des logs avec Log4j

## Présentation

- Log4J est constitué de 3 composants principaux qui permettent de configurer le dispositif de journalisation :
  - les **Loggers** pour écrire les messages,
  - les **Appenders** pour sélectionner la destination des messages
  - et les **Layouts** pour la mise en forme des messages.

# Gestion des logs avec Log4j

## La classe Logger

- Le Logger est l'entité de base pour effectuer la journalisation, il est mis en oeuvre par le biais de la classe `org.apache.log4j.Logger`.
- **L'obtention d'une instance de Logger** se fait en appelant la méthode statique `Logger.getLogger` :

```
import org.apache.log4j.Logger;

public class MaClasse {
 private static final Logger logger = Logger.getLogger(MaClasse.class);
 // suite
}
```
- Il est possible de donner un nom arbitraire au Logger.
- Cependant, il est préférable d'utiliser le nom de la classe pour des raisons de facilité.

# Gestion des logs avec Log4j

## Exemple Logger

### Logger

```
1 //Logging
2 import org.apache.log4j.*;
3
4 // use default logger
5 static Logger rootLogger = Logger.getLogger();
6
7 // create your own logger with name
8 static Logger myLogger = Logger.getLogger("myLogger");
9
10 // create your own logger with class name
11 static Logger myObjectLogger = Logger.getLogger(MyObject.class);
```

# Gestion des logs avec Log4j

## Les niveaux de journalisation - 1

- La notion de niveau de journalisation ou de priorité d'un message représente l'importance du message à journaliser.
- Un message n'est journalisé que si sa priorité est supérieure ou égale à la priorité du Logger effectuant la journalisation.
- L'API Log4j définit 5 niveaux de logging
  - DEBUG < INFO < WARN < ERROR < FATAL
  - ALL
  - OFF

# Gestion des logs avec Log4j

## Les niveaux de journalisation – 2

|              |       | Will Output Messages Of Level |      |      |       |       |
|--------------|-------|-------------------------------|------|------|-------|-------|
| Logger Level |       | DEBUG                         | INFO | WARN | ERROR | FATAL |
|              | DEBUG |                               |      |      |       |       |
|              | INFO  |                               |      |      |       |       |
|              | WARN  |                               |      |      |       |       |
|              | ERROR |                               |      |      |       |       |
|              | FATAL |                               |      |      |       |       |
|              | ALL   |                               |      |      |       |       |
|              | OFF   |                               |      |      |       |       |

- La version 1.3 introduira le niveau **TRACE** qui représente le niveau le plus fin (utilisé par exemple pour journaliser l'entrée ou la sortie d'une méthode).



# Gestion des logs avec Log4j

## Les niveaux de journalisation – 3

- Pour les niveaux de base, des méthodes de raccourcis sont fournies, elle portent le nom du niveau :

```
try {
 // équivaut à logger.info("Message d'information");
 logger.log(Level.INFO, "Message d'information");
 // Code pouvant soulever une Exception
 //...
} catch (UneException e) {
 // équivaut à logger.log(Level.FATAL, "Une exception est survenue", e);
 logger.fatal("Une exception est survenue", e);
}
```

# Gestion des logs avec Log4j

## L'interface Appender

- Utilisé par log4j pour enregistrer les événements de journalisation.
- Chaque Appender a une façon spécifique d'enregistrer ces événements.
- Log4j vient avec une série d'Appendes :
  - `org.apache.log4j.jdbc.JDBCAppender` : Effectue la journalisation vers une base de données ;
  - `org.apache.log4j.net.JMSAppender` : Utilise JMS pour journaliser les événements ;
  - `org.apache.log4j.net.SMTPAppender` : Envoie un email lorsque certains événements surviennent (ne pas activer si niveau =DEBUG...) ;
  - `org.apache.log4j.net.SocketAppender` : Envoie les événements de journalisation vers un serveur de journalisation ;
  - .....

# Gestion des logs avec Log4j

## Les Layouts

- Les Layouts sont utilisés pour mettre en forme les différents événements de journalisation avant qu'ils ne soient enregistrés.
  - `org.apache.log4j.SimpleLayout` : les événements journalisés ont le format Niveau - Message[Retour à la ligne] ;
  - `org.apache.log4j.PatternLayout` : Layout le plus flexible, le format du message est spécifié par un motif (pattern) composé de texte et de séquences d'échappement indiquant les informations à afficher.
  - `org.apache.log4j.XMLLayout` : Comme son nom l'indique, formate les données de l'événement de journalisation en XML
  - `org.apache.log4j.HTMLLayout` : Les événements sont journalisés au format HTML.

# Gestion des logs avec Log4j

## Utilisation de Log4j

- d'`efinir un fichier de configuration **log4j.properties**
- le placer dans **src** sous Eclipse
- ou le copier dans le répertoire de *build*
- mettre **log4j-1.2.15.jar** dans le répertoire lib

## Exemple

### Configuration de base

```
1 log4j.rootLogger=DEBUG, stdout
2 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
3 log4j.appender.stdout.Target=System.out
4 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
5 log4j.appender.stdout.layout.ConversionPattern=%5p %c{1}:%L - %m%n
```

# Gestion des logs avec Log4j

## Exemple avec fichier

### Configuration avec fichier

```
1 # set appender to file
2 log4j.rootLogger=DEBUG, file
3 log4j.appender.file=org.apache.log4j.FileAppender
4 log4j.appender.file.File=file.log
5 log4j.appender.file.layout=org.apache.log4j.SimpleLayout
```

# Gestion des logs avec Log4j

## Outils

- Chainsaw(Java et Java Web Start)
- Plugin Eclipse
  - Log4E
    - <http://log4e.jayefem.de>

# Gestion des logs avec Log4j

## Chainsaw Web Start

The screenshot shows the Chainsaw v2 - Log Viewer application. The interface includes a menu bar (File, View, Current tab, Help), a toolbar with various icons, and a main log display area. On the left, a tree view shows the logger hierarchy: Root Logger, org, apache, log4j, chainsaw, and net. The main log display area shows a table of log entries with columns: ID, Timestamp, Level, Logger, and Message. The log entries are filtered to show only those from the org.apache.log4j.chainsaw.net.SocketReceiver logger. The log level is set to DEBUG. The log message for the selected entry is "Adding to Map org.apache.log4j.net.SocketReceiver".

| ID | Timestamp               | Level | Logger                                | Message                                                         |
|----|-------------------------|-------|---------------------------------------|-----------------------------------------------------------------|
| 27 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.Chainsaw... | Adding col 'hostname', columnNames=[Logger, Timestamp, L...     |
| 28 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.help                    |
| 29 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.help.HelpManager        |
| 30 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.plugins                 |
| 31 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.plugins.PluginCl...     |
| 32 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.receivers               |
| 33 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.receivers.Recei...      |
| 34 | 2005-04-29 08:15:21,453 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.ApplicationPrefe...     |
| 35 | 2005-04-29 08:15:23,750 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.LogPanel                |
| 36 | 2005-04-29 08:15:23,750 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.LogPanelLogger...       |
| 37 | 2005-04-29 08:15:23,750 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.LoggerNameTre...        |
| 38 | 2005-04-29 08:15:23,750 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.ChainsawCydicB...       |
| 39 | 2005-04-29 08:15:24,875 | DEBUG | org.apache.log4j.chainsaw.receiver... | updateReceiverTreeInDispatchThread, should not be need...       |
| 40 | 2005-04-29 08:15:24,875 | DEBUG | org.apache.log4j.net.SocketReceiver   | Simple Receiver closing server socket                           |
| 41 | 2005-04-29 08:15:24,875 | ERROR | org.apache.log4j.net.SocketReceiver   | error starting SocketReceiver (Simple Receiver), receiver di... |
| 42 | 2005-04-29 08:15:25,984 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.chainsaw.receivers.Recei...      |
| 43 | 2005-04-29 08:15:25,984 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.net                              |
| 44 | 2005-04-29 08:15:25,984 | DEBUG | org.apache.log4j.chainsaw.LogPane...  | Adding to Map org.apache.log4j.net.SocketReceiver               |

Level: DEBUG  
Logger: org.apache.log4j.chainsaw.LogPanelLoggerTreeModel  
Time: 2005-04-29 08:15:25,984  
Thread: Thread-17  
Message: Adding to Map org.apache.log4j.net.SocketReceiver  
NDC: null

0 hidden loggers ...

Welcome | Drag & Drop XML log files here | chainsaw-log

Welcome to Chainsaw v2!

# Table des matières

- Introduction
- Définition de classes Java
- Les bases du langage
- Polymorphisme et héritage
- Classes utilitaires et Collection
- Gestion des exceptions
- Package IO
- Accès aux données
- Gestion des logs avec Log4j
- Tests unitaires avec JUnit



# Tests unitaires avec JUnit

## SOMMAIRE

- Cycle de développement
- Tests unitaires
- JUnit 3.8 / JUnit 4
- Assertion
- Ensemble de tests

# Tests unitaires avec JUnit

## Tests unitaires

- ***Unité***
  - Portion élémentaire d'un programme
  - Classe ou fonction/méthode
- ***Test unitaire***
  - Teste une unité
  - Assertions à vérifier sur différents cas critiques
  - Tests indépendants
- ***Bug de régression***
  - Evolution des programmes au cours du temps
  - Apparition de nouveaux bugs
  - Nécessité de tester l'ensemble du programme après chaque modification

# Tests unitaires avec JUnit

## Cycle de développement

### Cycle

1. Ajout d'un test
2. Ecriture d'une première version de la fonction qui fait echouer le test
3. Exécution du test
4. Ecriture d'une version brouillon de la fonction qui passe le test
5. Exécution du test
6. Raffinement du code de la fonction

### Avantages

- Code plus robuste
- Travail collaboratif : plus grande confiance dans un code fourni avec des tests

# Tests unitaires avec JUnit

## Limites

- Evaluation partielle
- Ne garantit pas le fonctionnement de l'intégralité du programme
- Possibilité de bug dans les tests

## xUnit

- Ensemble d'environnements pour programmer des tests unitaires
- Collection de méthodes, macros, classes, etc.

- ▶ SUnit pour Smalltalk (Kent Beck)
- ▶ CUnit pour C
- ▶ CppUnit pour C++
- ▶ NUnit pour C#
- ▶ JUnit pour Java
- ▶ ...

# Tests unitaires avec JUnit

## JUnit

- Fichier JAR : `junit.jar`
- Versions actuellement utilisées : `3.8` et `4.x`

## Utilisation

- Une classe de test par classe de programme
  - Classe `Exemple`
  - Classe de test `ExempleTest`
- Une ou plusieurs méthodes de test par méthode de la classe
  - Méthode `methode`
  - Méthode de test `testMethode`
- Une ou plusieurs assertions par méthode de test
- Fichiers sources des classes de test séparées des classes du programme

# Tests unitaires avec JUnit

## Classe de test

### JUnit 3.8

- `Package junit.framework.* ;`
- `public class ExempleTest extends TestCase`
- `public void testMethode()`

### JUnit 4

- `Package org.junit.* ;`
- Pas de classe à étendre
- Utilisation de l'annotation `@Test`

`@Test`

`public void testMethode()`

# Tests unitaires avec JUnit

## Assertion

- Utilisation de la classe Assert
- Méthodes statiques
- Deux versions de chaque méthode :
  - Arguments du test uniquement
  - Message (String) et arguments du test

# Tests unitaires avec JUnit

## **AssertEquals (test d'égalité)**

- `void assertEquals(Object expected, Object actual)`
- `void assertEquals(String message, Object expected, Object actual)`
- `void assertEquals(long expected, long actual)`
- `void assertEquals(String message, long expected, long actual)`
- `void assertEquals(double expected, double actual, double delta)`
- `void assertEquals(String message, double expected, double actual, double delta)`

## **AssertTrue et AssertFalse()**

- `void assertTrue(boolean condition)`
- `void assertTrue(String message, boolean condition)`
- `void assertFalse(boolean condition)`
- `void assertFalse(String message, boolean condition)`



# Tests unitaires avec JUnit

## AssertNull et AssertNotNull (Teste si un objet est null ou pas)

- `void assertNull(Object object)`
- `void assertNull(String message, Object object)`
- `void assertNotNull(Object object)`
- `void assertNotNull(String message, Object object)`

## AssertSame et AssertNotSame

- `void assertSame(Object expected, Object actual)`
- `void assertSame(String message, Object expected, Object actual)`
- `void assertNotSame(Object unexpected, Object actual)`
- `void assertNotSame(String message, Object unexpected, Object actual)`

## Autres

- ▶ `assertArrayEquals` : tests d'égalité entre deux tableaux (byte, char, int, long, short ou Object)
- ▶ `assertThat` : tests plus complexes
- ▶ `fail` : échec (pour des tests codés sans utiliser l'API JUnit)

# Tests unitaires avec JUnit

## Exemple

```
import org.junit.* ;
public class CalculTest {
 @Test
 public void testAdd() {
 Assert.assertEquals("2+2=4", 4, Calcul.add(2, 2) ;
 Assert.assertEquals("5+3=8", 8, Calcul.add(5, 3) ;
 Assert.assertEquals("0+7=7", 7, Calcul.add(0, 7) ;
 }
}
```

# Tests unitaires avec JUnit

## Contexte des tests

- Plusieurs tests d'une même classe initialisent des objets de la même façon
- Création de deux méthodes :
  - ▲ Initialisation des objets
  - ▲ Désallocation des ressources
- Méthodes exécutées automatiquement avant et après
- Objets stockés comme attributs

# Tests unitaires avec JUnit

## Contexte des tests

### JUnit 3.8

- void setUp()
- void tearDown()

### JUnit 4

- @Before
- @After
- @BeforeClass
- @AfterClass

# Tests unitaires avec JUnit

## Exemple JUnit 3.8

```
import junit.framework.*;
public class FichierTest {
 File fichier ;
 public void setUp() {
 this.file = new File("toto");
 }
 public void tearDown() {
 this.file.close();
 }
 ...
}
```

# Tests unitaires avec JUnit

## Exemple JUnit 4

```
import org.junit.* ;
public class FichierTest {
 File fichier ;
 @Before
 public void ouvreFichier() {
 this.file = new File("toto");
 }
 @After
 public void fermeFichier() {
 this.file.close();
 }
 ...
}
```

# Tests unitaires avec JUnit

## Exemple JUnit 4

```
import org.junit.* ;
public static class FichierTest {
 File fichier ;
 @BeforeClass
 public static void ouvreFichier() {
 FichierTest.file = new File("toto") ;
 }
 @AfterClass
 public static void fermeFichier() {
 FichierTest.file.close();
 }
 ...
}
```

# Tests unitaires avec JUnit

## Ensemble de tests

- Exécution d'un ensemble de classes de test
- Pour tester tout un programme
- Pour tester certains packages



# Tests unitaires avec JUnit

## Ensemble de tests

- Exécution d'un ensemble de classes de test
- Pour tester tout un programme
- Pour tester certains packages

## Mise en œuvre

- Création d'une classe qui regroupe tous les tests (AllTests)
- Annotations

# Tests unitaires avec JUnit

## Exemple JUnit 4

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses(value={Exemple1Test.class,Exemple2Test.class})
public class AllTests { }
```

## Exécution des tests JUnit 4

```
java -cp ./bin :/usr/share/java/junit4-4.3.1.jar
 org.junit.runner.JUnitCore exemple.AllTests
```

# Tests unitaires avec JUnit

## L'interface du TestRunner

