PROGRAMMATION OBJET EN JAVA

Papa DIOP

UFR Sciences Et Technologies

Université de THIES

Programmation Orientée Objet avec Java

- **Concepts de base de la POO**
- **Classes et Objets**
- **Packages**
- **Exceptions**
- **Héritage**
- **Classes abstraites et interfaces**
- **Applets**
- **Interfaces Graphiques**
- **JDBC**

Bibliographie (1/2)

- **Le langage Java**, K.Arnold, J. Gosling, Thomson, 1996.
- **Java La synthèse – Des concepts objet aux architectures Web**, G. Clavel, et al., Dunod, 2000.
- **Introduction à la programmation objet en Java**, J. Brondeau, Dunod, 1999.
- **Le langage Java – Programmer par l'exemple**, T. Leduc, D. Leduc, Technip, 2000.
- **Algorithmique et programmation objet en Java**, V. Granet, Dunod, 2001.
- **Initiation à l'algorithmique objet**, A. Cardon, C. Dahancourt, Eyrolles, 2001.

Bibliographie (2/2)

- **Les Cahiers du Programmeur Java,** E. Pubaret, Eyrolles, 2004.
- **Thinking in Java**, B. Eckel, Prentice Hall, 2000. Disponible sous forme électronique à l'URL http://www.BruceEckel.com
- **Le poly de Java**, H. Garetta, Polycopié, Université de la Méditerranée.
- **Développons en Java**, J. M. Doudoux, Tutorial en ligne, http://perso.wanadoo.fr/jm.doudoux/java/tutorial/
- **Site Java de Sun** : http://java.sun.com

CONCEPTS DE BASE DE LA POO

Notion d'orienté objet

• La programmation structurée repose sur l'équation de N. Wirth :

Algorithmes + Structures de données = Programme

• Généralement, les données sont plus stables et plus pérennes que les traitements qu'on leur applique.

• Donc, il est préférable de structurer un système par rapport à ses données plutôt que par rapport à ses fonctions.

➔Concept de POO : tout est dirigé par les données.

Notion d'orienté objet

➔La POO est fondée sur le concept d'objet.

• Un objet est une entité regroupant des données et des procédures (*méthodes*) agissant sur ces données.

POO : Méthodes + Données = Objet

• Derrière cette association, se cache le concept d'***encapsulation des données.***

• Il n'est pas possible d'agir directement sur les données d'un objet.

• Il faut nécessairement passer l'intermédiaire de ses méthodes qui jouent ainsi le rôle d'interface obligatoire.

Encapsulation

- Mécanisme consistant à empaqueter données et traitements au sein d'une même structure en ne la rendant accessible que par le biais des opérations (**interface**) laissées visibles à cet usage.

- On empêche l'accès aux données par un autre moyen que les services proposés.
- Cette structure qui encapsule les données et les traitements s'appelle l'**objet**.
- L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

Notion d'objet

Un objet est défini à la fois par :

- ✓ *des informations* :
  - • données portées par l'objet;
  - • on les nomme **attributs**, **variables d'instance** ou **données**;
- ✓ *des comportements* :
  - • traitements applicables à l'objet;
  - • on les nomme **méthodes** ou **opérations**.

Notion d'objet

Un objet se définit par un triplet :

- un **identificateur** (*OID ou Object Identifier*) qui permet de le référencer de façon unique.
- des données (**attributs**) caractérisant l'objet. Ce sont des variables stockant des informations *d'état de l'objet*.
- et des procédures (**méthodes**) agissant sur ces données.

Notion d'objet

- Les **méthodes** :
  - caractérisent son comportement, c-à-d l'ensemble des *opérations* que l'objet est à même de réaliser.
  - permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets).

Notion de classe

- Une *classe* est la description d'une famille d'objets ayant même structure et même comportement.

- Elle regroupe :

  ✓ un ensemble d'**attributs** : données représentant l'état de l'objet

  ✓ et un ensemble de **méthodes** : opérations applicables aux objets.

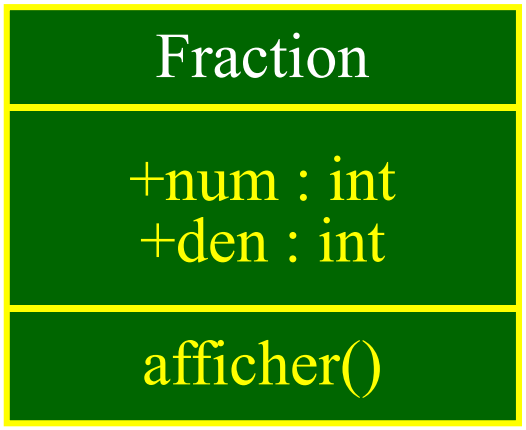
- **Objet = instanciation d'une classe.**

Notion de classe

- Décrit les propriétés de ses instances et constitue une sorte de « moule » pour la création d'objets.

- Répertorie les traitements applicables à ces objets.

- Représente un modèle à partir duquel seront construits des objets ayant les mêmes propriétés de structures et d'utilisation.



Notion de classe : exemple

- **ObjetPostal** :
  - ✓ classe décrivant les caractéristiques communes des objets passant par la Poste.
  - ✓ Tous ces objets ont un poids, un tarif d'affranchissement, une valeur déclarée et peuvent être recommandés.

➢ Champs :

poids, tarif, valeur, recommande

➢ Méthodes :

aValeurDeclaree, poidsObjet, recommander.

Notion de classe

- La notion de **classe** peut être assimilable à la notion de **type** :
  - ✓ une classe définit les propriétés de tous les objets qui lui sont associés.

- Un objet est une sorte de ***matérialisation de sa classe***.

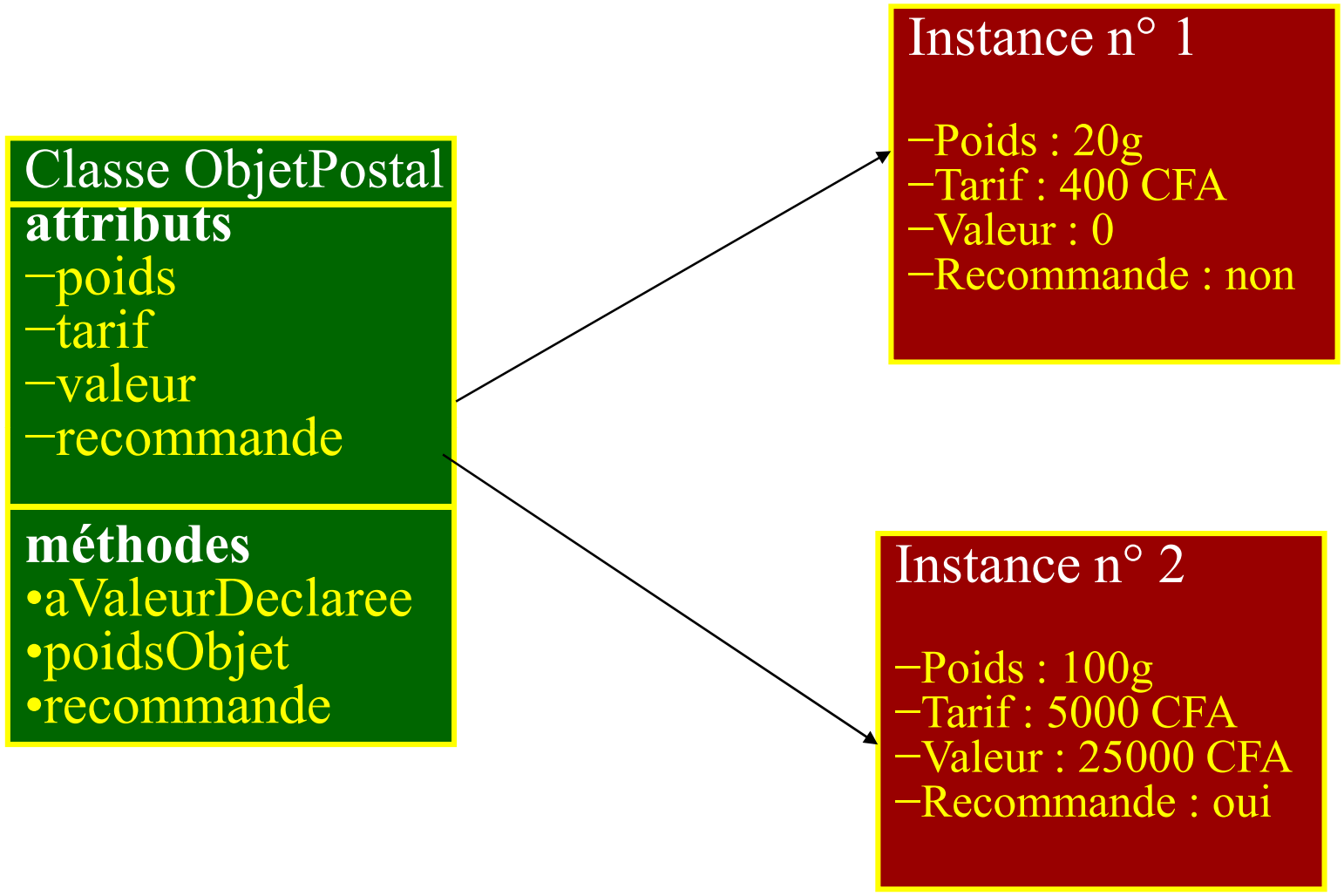
Notion de classe

- Les objets d'une même classe portent tous les attributs de leur classe, ces attributs peuvent avoir des valeurs différentes suivant les instances.

- Donc, la valeur des attributs est propre à chaque objet.

- Mais toutes les instances d'une même classe partagent les comportements définis dans la classe.

Instanciation



Notion de message

- L'activation de méthode se fait par **envoi d'un message** à l'objet concerné (objet récepteur).

- **Message** = **récepteur + signature de méthode**

- **Signature de méthode** = (nom méthode + paramètres)

Abstraction

- Un objet est caractérisé par :
  - une **partie publique** directement accessible par les autres objets
    - ✓attributs publics : consultables et modifiables sans aucune restriction;
    - ✓et surtout méthodes publiques : interface de l'objet.
  - une **partie privée** qui n'est pas directement accessible : ensemble d'attributs manipulables qu'à travers des méthodes publiques.
  - une **partie implémentation** : réalisation interne de l'objet.

Abstraction et développement en équipe

• L'abstraction est un concept indispensable pour le développement qui permet de distinguer 2 catégories d'acteurs :

- ✓ **le concepteur de la classe** : fabrique et implémente la classe; a accès à tous les détails de la classe.
- ✓ **l'utilisateur de la classe** : instancie et utilise des objets de la classe; n'a besoin de connaître que l'interface de la classe.

Abstraction et développement en équipe

➔ Les détails de l'implémentation peuvent être cachés aux programmeurs qui n'ont pas conçu la classe.

➔ Cela rend la programmation sûre, car les utilisateurs d'une classe n'ont pas les moyens de violer l'intégrité des objets.

Notion d'héritage

- Mécanisme de transmission des propriétés d'une classe à une autre (sous-classe) par dérivation.

- *Spécialisation* : L'héritage permet de propager des spécifications d'une classe "générale" vers des sous-classes "particulières" en offrant la possibilité de les enrichir successivement.

Le polymorphisme

- Faculté d'une méthode donnée à s'éxécuter différemment suivant le contexte de la classe où elle se trouve.

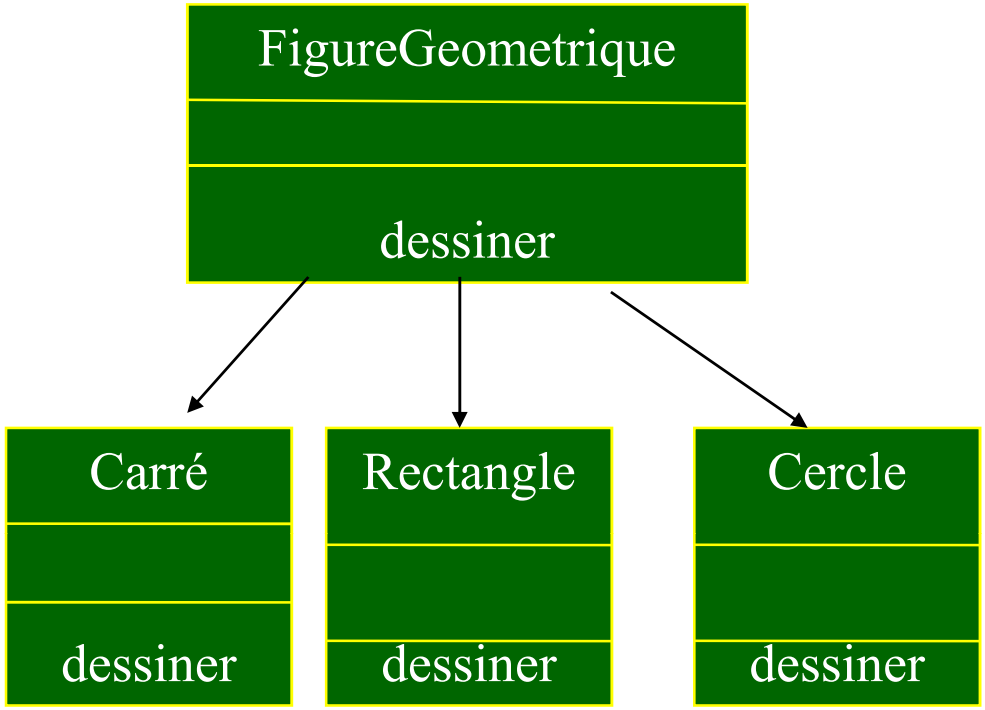
- Une méthode définie dans une superclasse peut s'éxécuter de manière différente selon la sous-classe où elle est héritée.

- C'est le cas lorsqu'une sous-classe hérite d'une méthode sans la surcharger.

Le polymorphisme

- La liaison dynamique consiste dans ce cas, lors de l'exécution, à :
  - parcourir de façon ascendante le graphe d'héritage pour trouver l'objet récepteur.
  - exécuter la première méthode dont la signature correspond à celle du message sera exécutée.

Le polymorphisme (exemple)



- La méthode « *dessiner* » est implémentée différemment dans les classes dérivées.
- Le système appellera la méthode « *dessiner* » spécifique selon la forme de l'objet concerné.

CLASSES ET OBJETS EN JAVA

Définition d'une classe

```
public class NomDeLaClasse {
        déclarations des champs (attributs, propriétés)
        déclarations des méthodes
}

public class Fraction {
// champs
public int num ;
public  int den ;
// méthodes
public void afficher () {
                if  (this.num % this.den == 0)
                        System.out.println(this.num/this.den ) ;
                else
                        System.out.println(this.num+"/"+this.den) ;
        }
}
```

Le désignateur (ou auto-référence) this

- **this** est une pseudo-variable qui permet :
  - ✓ de faire référence à l'objet courant (celui qu'on est en train de définir).
  - ✓ ou de désigner ses attributs ou ses méthodes.

- this est généralement utilisé pour lever l'ambiguité sur un identificateur lorsque le paramètre d'une méthode porte le nom de l'attribut de l'objet qu'il est chargé de modifier.

- *Exemple :*

```
void fixerNumerateur (int num) {
        this.num = num ;
}
```

Utilisation d'une classe

```
public class FractionUtil {
        public static void main (String[] args) {
                Fraction f = new Fraction();
                f.num = 8; f.den = 5;
                f.afficher();
        }
}
```

- Instanciation avec l'opérateur ***new***

- Java s'occupe de la destruction des objets non utilisés (mécanisme de *garbage collector*).

➔ pas besoin de "destructeurs".

Exemple d'application Java avec le JDK

• Le fichier source prend le nom de la classe qu'il définit avec l'extension ***.java***

==> *FractionUtil.java*.

• La compilation génère un fichier binaire Java avec l'extension ***.class***.

javac FractionUtil.java

==> *FractionUtil.class*

• Ce fichier peut être exécuté grâce à un interpréteur java.

java FractionUtil

Constructeurs spécifiques

```
public class Fraction {
        public int num, den ;

        public Fraction (int n, int d) { this.num=n; this.den=d;}
        public Fraction (int n) { this.num=n; this.den=1;}

        public Fraction inverser() {
                if  (this.num == 0) return null;
                return new Fraction(this.den, this.num);
        }
}
```

• Les divers constructeurs doivent avoir des signatures différentes (*nombre d'arguments ou types des arguments différents*).

Les variables (1)

- **variables d'instance** : elles déterminent les attributs ou l'état d'un objet donné (*attributs d'instance*)
- **variables de classe** : elles déterminent les attributs ou l'état d'une classe donnée (*attributs de classe*)
- **variables locales** : déclarées et utilisées dans les définitions de méthodes.

Syntaxe:

```
Type_de_la_variable identificateur_de_la_variable;
```

Exemples:

```
int annee;

String nom, prenom, adresse;
```

Les variables (2)

```
Class Une_Classe {
        ...
        déclarations des variables de classe et d'instance
        ...
        public void une_Methode (déclarations des
        paramètres) {
                ...
                déclarations des variables locales
        }
}
```

Références et objets

•Un **objet** doit être considéré comme accessible par **référence**. L'**identificateur** d'un objet désigne une **référence** à l'objet et non l'objet lui-même.

•Ce sont ces **références** qui sont affectées ou passées comme arguments lors des appels de méthodes.

•*Exemple:*

```
Fraction f1, f2;
f1 = new Fraction();
f2 = f1;
```

f1 et *f2* sont des variables qui se réfèrent au ***même objet***.

Cet objet existe en un ***seul exemplaire*** mais avec ***deux références***.

Les membres de classe

- ✓ **Les attributs ou méthodes sont accessibles:**
  - soit via une instance de la classe: membres d'instance
  - soit via la classe elle-même: membres de classe ou membres statiques.

- ✓ **Les membres de classe :**
  - sont introduits par le mot static
  - existent en un seul exemplaire
  - sont partagés par toutes les instances de la classe.

- ✓ **Ils ne peuvent être invoqués que sur la classe.**

Attributs de classe

- Ils sont initialisés au moment du chargement de la classe.
- Ils sont modifiables par tous les objets de la classe.
- Toute modification est répercutée au niveau des autres objets.
- Ils permettent d'éviter la duplication de certains types d'attributs (comme les constantes) dans chaque instance.

Exemple

L'attribut **out** est un attribut de la classe java.lang.System

```
System.out.println(" Salut! ");
```

Les méthodes de classe

- Elles ne peuvent pas modifier des attributs d'instance.
- Lorsqu'une méthode ne manipule que des attributs (ou méthodes) de classe, elle doit être déclarée statique.

Exemples

- exit() est une méthode statique de la classe java.lang.System

```
System.exit();
```

- sqrt() est une méthode statique de la classe java.lang.Math

```
System.out.println(Math.sqrt(100));
```

Visibilité des champs et méthodes

- Le principe d'encapsulation permet d'introduire celui de protection des attributs et méthodes.

- **Les Modificateurs de visibilité en Java :**
  - **public** : accessible par toutes les classes. Hérité par les sous classes.
  - **private** : accessible que par les seules méthodes de sa classe. Non hérité.
  - **protected** : accessible par les classes du même *package*. Hérité par les sous classes.
  - ***par défaut* :** accessible par les classes du même *package*. Hérité par les sous classes que si elles se trouvent dans le même package.

Visiblité : exemple

```
public class Etudiant {
public String nom;
private String numIns;

public Etudiant(String nom) {
        this.nom = nom;
}

// methodes accesseurs (getters) et modificateurs (setters)
public String getNumIns() { return  this. numIns; }
public void setNumIns(String numIns) {
        if (numIns==null || numIns.length!=11) return;
        this.numIns = numIns;
}

public void afficher(){
        System.out.println("etudiant :"+ nom);
}
}
```

PACKAGES

Les packages : définition et utilité

Définition

• Un package est un ensemble de classes et d'autres packages regroupés sous un nom.

• C'est l'adaptation du concept de librairie ou de bibliothèque.

Utilité

• Servent à structurer l'ensemble des classes et interfaces.

• Augmentent la lisibilité des applications en structurant l'ensemble des classes selon une arborescence.

•Faciltent la recherche de l'emplacement physique des classes

• Empêchent la confusion entre des classes de même nom

• etc.

Les packages : généralités

• Lors de la définition d'un fichier source Java, on peut préciser son appartenance à un package en utilisant le mot-clé **package**:

```
package mon_package;
```

• Toutes les classes définies dans ce fichier font ainsi partie du package mon_package.

exemple: `package formesGeo;`

• Si la classe `Carre` se trouve dans un package `formesGeo`,

- son nom complet est `formesGeo.Carre`;
- le fichier `Carre.class` doit être placé dans un répertoire nommé `formesGeo`.

Les packages : importation

• Pour pouvoir utiliser une classe d'un package dans un autre fichier, il faut l'importer :

```
import nom_du_package.nom_classe;
```

```
import formesGeo.Carre;
```

• On peut importer toutes les classes d'un package en même temps:

```
import nom_du_package.*;
```

```
import java.io.*;
```

• Le package java.lang qui gère les types de données et les éléments de base du langage est automatiquement importé.

Les packages : compilation

- Pour compiler une classe contenant l'instruction suivante, il faut indiquer le chemin au compilateur.

```
import monPackage.ClasseImportee;
```

- *soit* **: Option –*classpath* de javac**

javac –classpath *chemin_du_fichier* MaClasse.java

- *soit :* **Variable d'environnement CLASSPATH**

doit contenir le chemin d'accès au répertoire racine du package.

Les packages : exécution

Pour exécuter la commande *java* monPackage.MaClasse :

- *soit* : Se placer dans le répertoire contenant le répertoire monPackage
- *soit* **: Utiliser l'option –classpath de la commande java**
- *soit* **: Variable d'environnement CLASSPATH**

doit contenir le chemin d'accès au répertoire monPackage.

Les packages prédéfinis (1/2)

- Les API Java sont organisées en deux parties :

  - **Java Core API** : API de base implantée dans tout interpréteur Java. Ces classes appartiennent au package “java”.

  - **Java Standard Extension API** : extensions normalisées par Sun au travers d’interfaces et implantées par les éditeurs.

    Ces classes appartiennent au package “javax”.

- Documentation sur les toutes classes Java disponible sur le site officiel de Java : http://java.sun.com/

Les packages prédéfinis (2/2)

Les classes sont regroupées par thèmes dans des packages :

- **java.lang** : types de données et éléments de base du langage (classes Object, String, Boolean, Integer, Float, Math, System (fonctions système), Runtime (mémoire, processus), etc.)
- **java.util**: structures de données et utilitaires (Dates, Collections, etc.)
- **java.io**: bibliothèque des entrées / sorties, fichiers.
- **java.net**: bibliothèque réseau
- **java.awt**: librairie graphique
- **java.applet**: librairie des applets.
- **java.security**: sécurité (contrôle d'accès, etc.)
- **java.rmi**: applications distribuées.
- **java.sql**: accèes aux BD (JDBC)

EXCEPTIONS

Les exceptions (1/8)

• Une **exception** est un **signal** qui indique qu'un évènement exceptionnel comme une erreur est survenue au cours de l'exécution d'un programme :

• FileNotFoundException survient lorsqu'on tente d'ouvrir un fichier inexistant.

• IOException survient lorsqu'on fait une lecture incorrecte, lorsqu'un fichier ne peut être fermé, etc.

Les exceptions (2/8)

•Java permet de gérer les erreurs par exception, i.e. de prévoir dans son application des blocs de code où sera traitée systématiquement, telle ou telle condition d'erreur.

• Une exception est un objet de la classe **java.lang.Exception**.

• Pour les gérer, on utilise l'**instruction *try***.

Les exceptions (3/8)

```
try {
        // bloc d'instructions à protéger
}

catch ( Exception e)  {
        // traitement de l'exception
}

finally {
        // à exécuter quelque soit la façon dont on sort du try
}
```

Les exceptions (4/8)

• Un bloc d'instructions **try** permet de regrouper une ou plusieurs instructions de programme (appel de méthode, instructions de contrôle, etc.) où des exceptions peuvent se produire et être traitées.

• Un bloc **try** est suivi d'instructions ***catch*** et ***finally***.

• **catch** spécifie le traitement associé aux exceptions, i.e ce que l'on fait si une exception se produit (afficher un message d'erreur par exemple).

• **finally** est exécuté inconditionnellement, quelque soit la façon dont on sort du bloc **try** (pour fermer par exemple un fichier ouvert dans le bloc try).

Les exceptions (5/8)

```
import java.io.FileReader;
public class LectureFichier {
public static void main(String[] args) {
                    try{
                            FileReader f=new FileReader("fic.txt");
                            System.out.println("Le fichier existe !");
                            f.close();
                            System.out.println("Le fichier a ete ferme !");
                    }
                    catch(FileNotFoundException fe) {
                            System.out.println("Fichier inexistant !");
                    }
                    catch(IOException ie) {
                            System.out.println("Erreur de fermeture !");
                    }
          }
}
```

Les exceptions (6/8)

Chaque **méthode susceptible de générer une exception** qui lui est propre doit la déclarer dans son en-tête. On utilise l'instruction ***throws*** pour cela :

```
public class Pile {
final int taille_max = 100 ;
private int sommet ;
private int[ ] elements ;

public void empiler (int x) throws ErreurPilePleine {
                    if  (sommet == taille_max)
                                throw new ErreurPilePleine()
                    else      elements[++sommet] = x ;
}
}
class ErreurPilePleine extends  java.lang.Exception {
ErreurPilePleine(){ System.out.println("Pile Pleine");}
}
```

Les exceptions (7/8)

Controle de saisie avant de calculer la factorielle d'un nombre :

```
  public static void main(String arg[]){
String s = null;
do
    try {
            s = JOptionPane.showInputDialog("Entrez un nbre: ");
            if (s != null) {
                    long n = Long.parseLong(s);
                    long factN = factorielle(n);
                    JOptionPane.showMessageDialog(null, n + "! = "+
factN);
            }
    }
    catch(NumberFormatException ex){
            JOptionPane.showMessageDialog(null, s + " n'est pas un
entier");
    }
…
```

Les exceptions (8/8)

Controle de saisie avant de calculer la factorielle d'un nombre :

```
…
            finally {
            reponse = JOptionPane.showConfirmDialog(null, "Voulez-
vous quitter ?");
}
   while (reponse == JOptionPane.NO_OPTION);
}

// methode factorielle
public static long factorielle(long  n){
if (n <= 1)
            return 1;
return n * factorielle(n-1);
 }
```

Exception avec la classe Scanner

• Génère une exception de type **InputMismatchException** en cas d'erreur de saisie. Cela permet de contrôler la saisie:

```
    java.util.Scanner entree = new java.util.Scanner(System.in);
    int x = 0;
    for ( ; ;) {
System.out.print("Donnez un entier x: ");
try{
        x = entree.nextInt();  break; }
catch (java.util.InputMismatchException ime) {
        entree.nextLine();
        System.out.print("Erreur - Recommencez"); }
}
System.out.print("x = " + x);
    }
```

HERITAGE

L'héritage

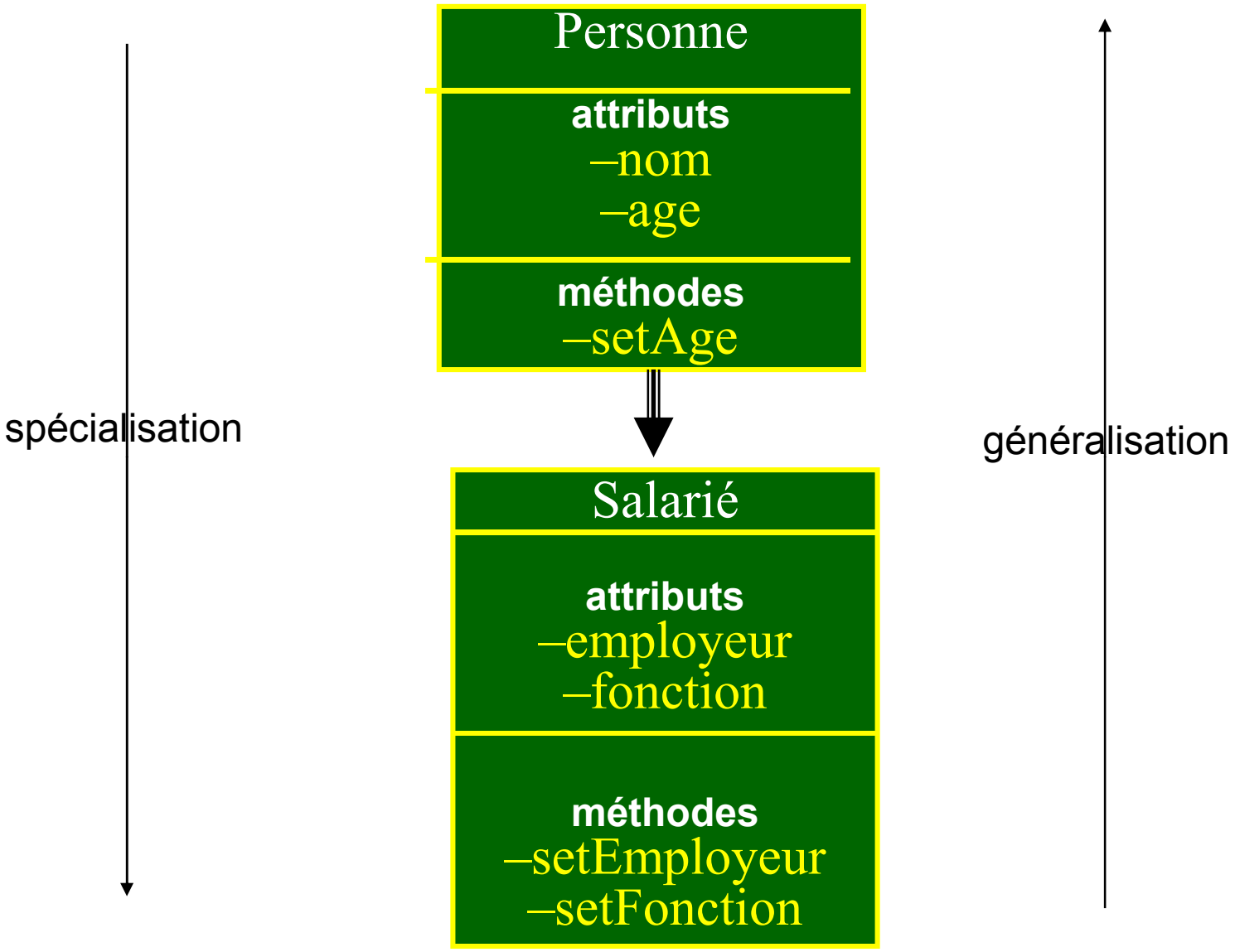
- L'héritage permet de définir une nouvelle classe à partir d'une (ou plusieurs) classe(s) existante(s).

- Les classes sont souvent organisées en hiérarchies; toutes les classes Java héritent de la classe **java.lang.Object**.

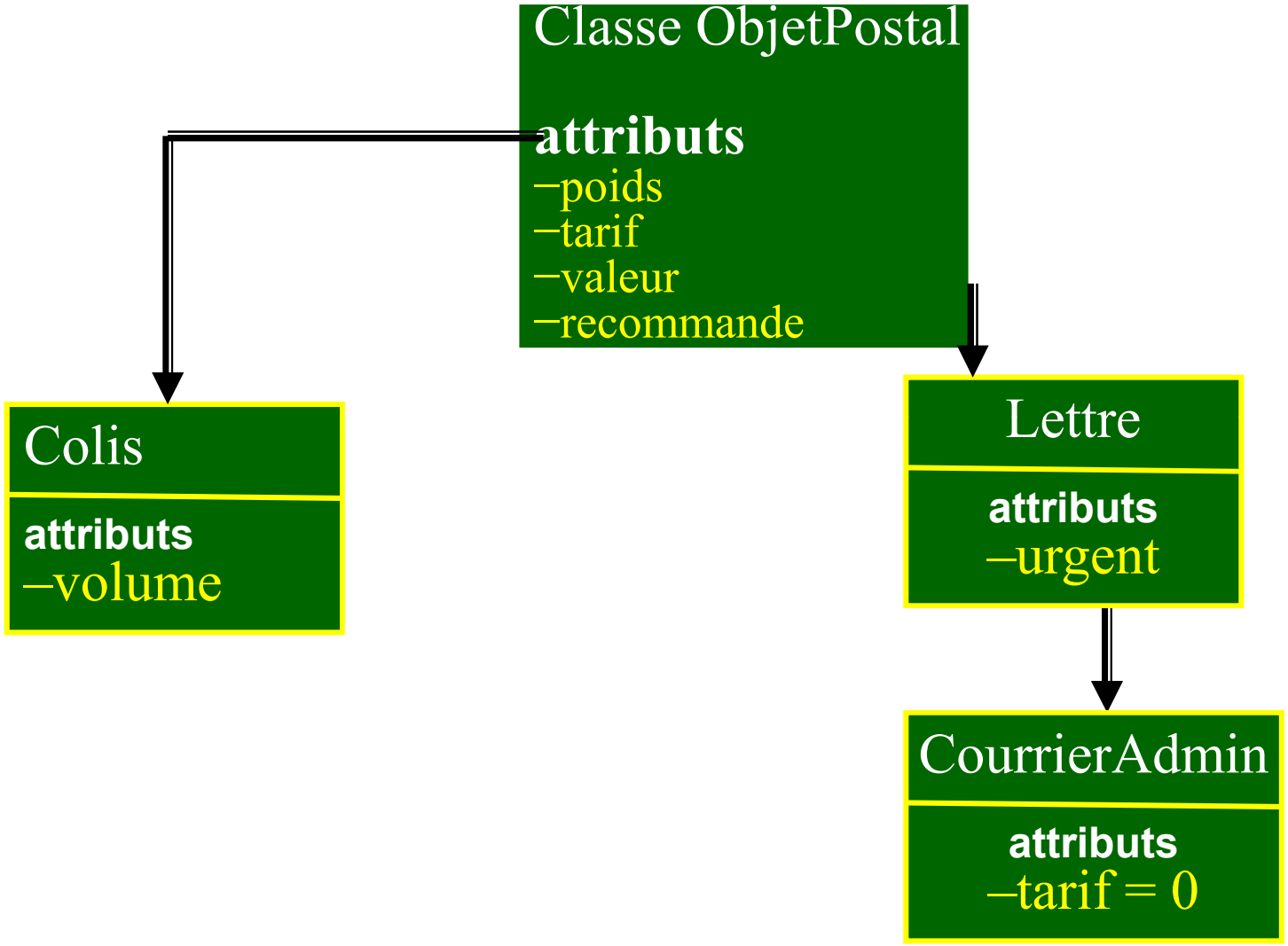
- Java n'autorise que l'héritage simple.

- L'héritage multiple est "remplacé" par la notion d'**interface**.

L'héritage : exemple 1



L'héritage : exemple 2



Héritage : Redéfinition de méthode

• Réécrire l'implémentation d'une méthode héritée sans modifier sa signature (seul l'objet récepteur diffère)

• Une méthode héritée peut être redéfinie dans la sous-classe: on conserve la même signature et le même type de retour.

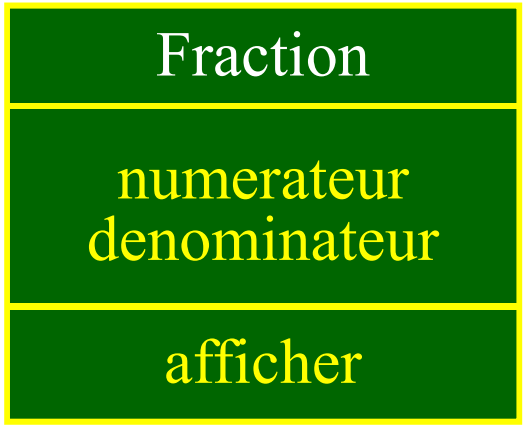
• On peut étendre le comportement d'une méthode héritée: la méthode de la super-classe reste accessible pour les objets de la sous-classe (*mot-clé* ***super***).

Héritage : Surcharge de méthodes

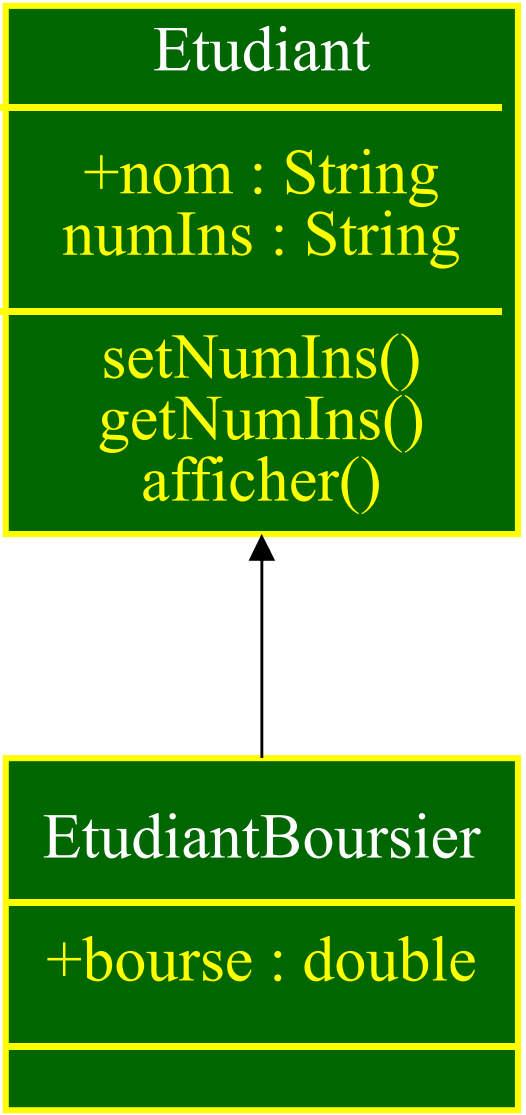
- Redéfinition de la signature et du code d'une méthode héritée
- La surcharge concerne 2 méthodes de même nom mais avec des signatures différentes.
  - ✓ *exple*: constructeurs de la classe *Fraction*

    Fraction (int numerateur, int denominateur)

    Fraction (int numerateur)



L'héritage simple : exemple



L'héritage simple : exemple

```
public class EtudiantBoursier extends Etudiant {

public double bourse;

public EtudiantBoursier(String nom, double bourse) {
        super(nom); //appel au constructeur Etudiant(String)
        this.bourse = bourse;
}
public EtudiantBoursier(String nom) {
        this(nom, 0); //appel au constructeur précédant
}

public void afficher() { // redefinition
        System.out.println("etudiant :"+ this.nom);
        System.out.println("bourse :"+ this.bourse);
}
}
```

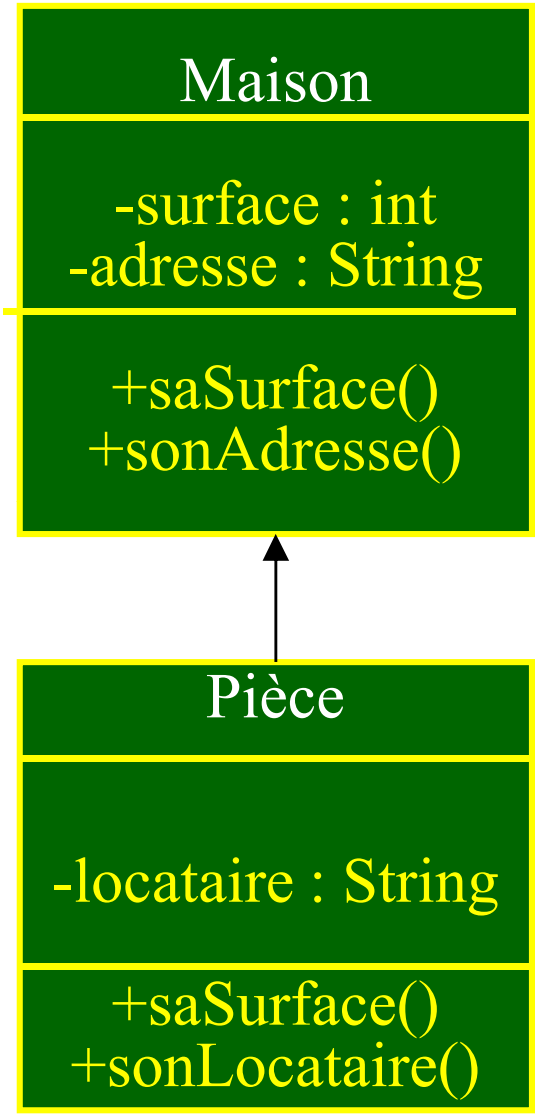
La pseudo-variable super

• En faisant hériter une classe d'une autre, on peut définir une méthode dont l'identificateur est le même que celui de sa classe mère.

➔ masquage de la méthode de la classe mère.

• La pseudo-variable super permet d'accèder à la méthode masquée de la classe mère.

La pseudo-variable super : exemple



La classe Object

• Toutes les classes héritent de la classe Object qui contient certaines propriétés et méthodes intéressantes permettant, par exemple :

- Connaître la classe d'un objet : getClass()
- Comparer des objets : equals()
- Afficher des objets : toString()

• *Exemple:*

```
public class Fraction {
public int num, den ;
public Fraction (int n, int d) { this.num=n; this.den=d;}
public String toString() {
                    return this.num + "/" + this.den;
        }
}
```

Exemple d'héritage : compte et compte d'epargne (1/3)

```
class Compte {

private String id;
private float solde;

public Compte (String id, float depot){
        this.id = id;
        this.solde = depot;
}
public String getId (){
        return this.id;
}
public float getSolde (){
        return this.solde;
}
}
```

Exemple d'héritage : compte et compte d'epargne (2/3)

```
class CompteEpargne extends Compte {
private float taux;
private int annees;
public CompteEpargne (String id, float depot, float taux){
        super (id, depot);
        this.taux = taux;
}
public void setAnnees (int annees){
        if (annees >= 0)  this.annees = annees;
}
public int getAnnees(){ return this.annees; }
public float getTaux(){  return this.taux; }
public float getSolde (){
        float solde = super.getSolde();
        for (int i=0; i<this.annees; i++) solde *= 1 + this.taux;
        return solde;
}
}
```

Exemple d'héritage : compte et compte d'epargne (3/3)

```
class CalculInterets{
   public static void main(String arg[]){
Compte compte1 = new Compte("A01", 100000f);
CompteEpargne compte2 = new CompteEpargne("E99", 100000f, 0.1f);
compte2.setAnnees(5);

Compte c;
String s = "L'argent qui dort ne rapporte rien:";
c = compte1;
s += "\n solde du compte n° " + c.getId() + ":" + c.getSolde();

c = compte2;
s += "\n solde du compte n° " + c.getId() + ":" + c.getSolde();
javax.swing.JOptionPane.showMessageDialog(null,s);
}
}
```

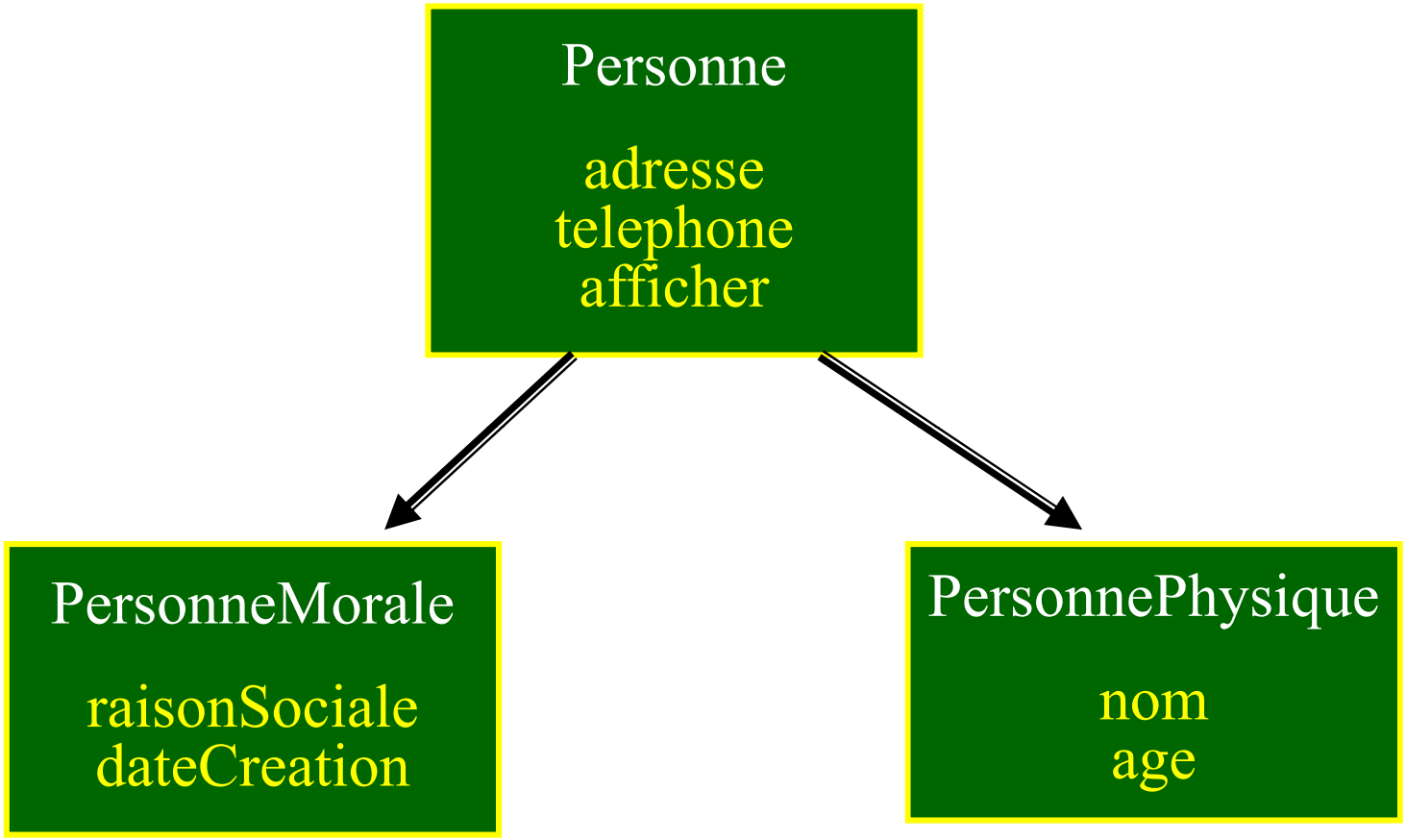
CLASSE ABSTRAITES ET INTERFACES

Classe abstraite

- Une classe abstraite est une classe utilisée pour factoriser des propriétés communes à plusieurs classes mais qui est trop générique pour être instanciée.

- Une classe abstraite est telle que l'on ne peut pas caractériser de manière précise ses instances.

Classe abstraite : exemple



Les classes abstraites

- Elles permettent de définir l'interface des méthodes.
- Elles contiennent au moins une méthode abstraite.
- Une méthode abstraite est précédée du mot-clé abstract et ne possède pas de corps.
- Elles ne peuvent pas être instanciées.
- Elles doivent être dérivées en sous-classes fournissant une implémentation à toutes les méthodes abstraites.
- Elles sont définies par l'introduction du mot-clé abstract.

```
abstract class FormeGeometrique {
    abstract double perimetre();
    abstract double surface();
}
```

Les interfaces : généralités

• Ce sont des classes abstraites dont l'instanciation serait sans intérêt. Elles ne peuvent donc pas être instanciées.

• Une interface Java contient une liste de méthodes abstraites que doit implémenter une classe pour rendre un service.

• Lorsqu'une classe implémente une interface, elle doit implementer toutes les méthodes définies dans l'interface. Une classe peut implémenter une ou plusieurs interfaces.

• Elles peuvent hériter d'autres interfaces (héritage simple).

Les interfaces : déclaration

• Une interface Java se déclare comme une classe en faisant précéder son identificateur du mot-clé ***interface***.

```
public interface UneInterface {
    // déclaration des champs et des méthodes
}
```

• Tous les champs sont des constantes dont les modificateurs sont implicitement *public static final*

• Toutes les méthodes sont publiques et non implémentées; elles utilisent implicitement des modificateurs *public abstract* et sont suivies d'un **;**.

Les interfaces : implémentation

• Une interface doit être implémentée par une ou plusieurs classes.

• Une classe peut implémenter une ou plusieurs interfaces.

• Lorsqu'une classe implémente une interface, elle doit implementer toutes les méthodes définies dans l'interface.

```
class UneClasse implements UneInterface {
   // champs et méthodes de la classe UneClasse
  // et implémentation des méthodes de l'interface UneInterface
}
```

Les interfaces : exemple

```
public interface FormeGeometrique {
public double perimetre();
public double surface();
}

public class Rectangle implements FormeGeometrique {
public double larg, long;
public Rectangle(double long, double larg) {
        this.long=long; this.larg=larg;
}
public double perimetre() {return 2*(larg+long);}
public double surface() {return larg*long;}
}
```

Les interfaces : exemple

```
public class Carre implements FormeGeometrique {
public  double cote;
public Carre(double cote) {this.cote = cote;}
public double perimetre() {return 4*cote;}
public double surface() {return cote*cote;}
}
public class Cercle implements FormeGeometrique {
public final static double PI = 3.1416;
public  double rayon;
public Cercle (double rayon) {this.rayon = rayon;}
public double perimetre() {return 2*PI*rayon;}
public double surface() {return PI*rayon*rayon;}
public void afficher() {
        System.out.println("cercle de rayon"+rayon); }
}
```

Les interfaces : exemple

```
public class Figures {
        public static void main (String[] args) {
                FormeGeometrique[] formes;
                formes = new FormeGeometrique[3];
                formes[0] = new Cercle(10);
                formes[1] = new Carre(4);
                formes[2] = new Rectangle(5,8);
                for (int i=0;i<formes.length;i++){
                        System.out.print("surface de la forme"+i+" : ");
                        System.out.println(formes[i].surface());
                }
        }
}
```

INTERFACES GRAPHIQUES

Interfaces graphiques : AWT & Swing

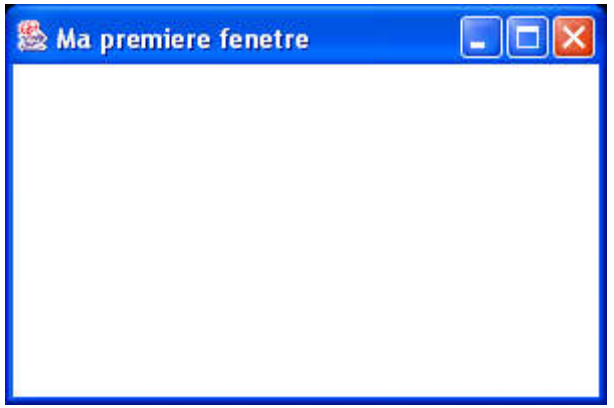
- AWT (Abstract Window Toolkit)
  - ✓ permet de créer des fenêtres (Frame),
  - ✓ d'utiliser :
    - des boutons de commandes (Button),
    - des zones de texte (TextField),
    - des zones de liste,
  - ✓ de dessiner, etc.
  - ✓ l'apparence de certains composants dépend de la plate-forme d'exécution (utilise des objets de l'env. d'exécution).

Interfaces graphiques : AWT & Swing

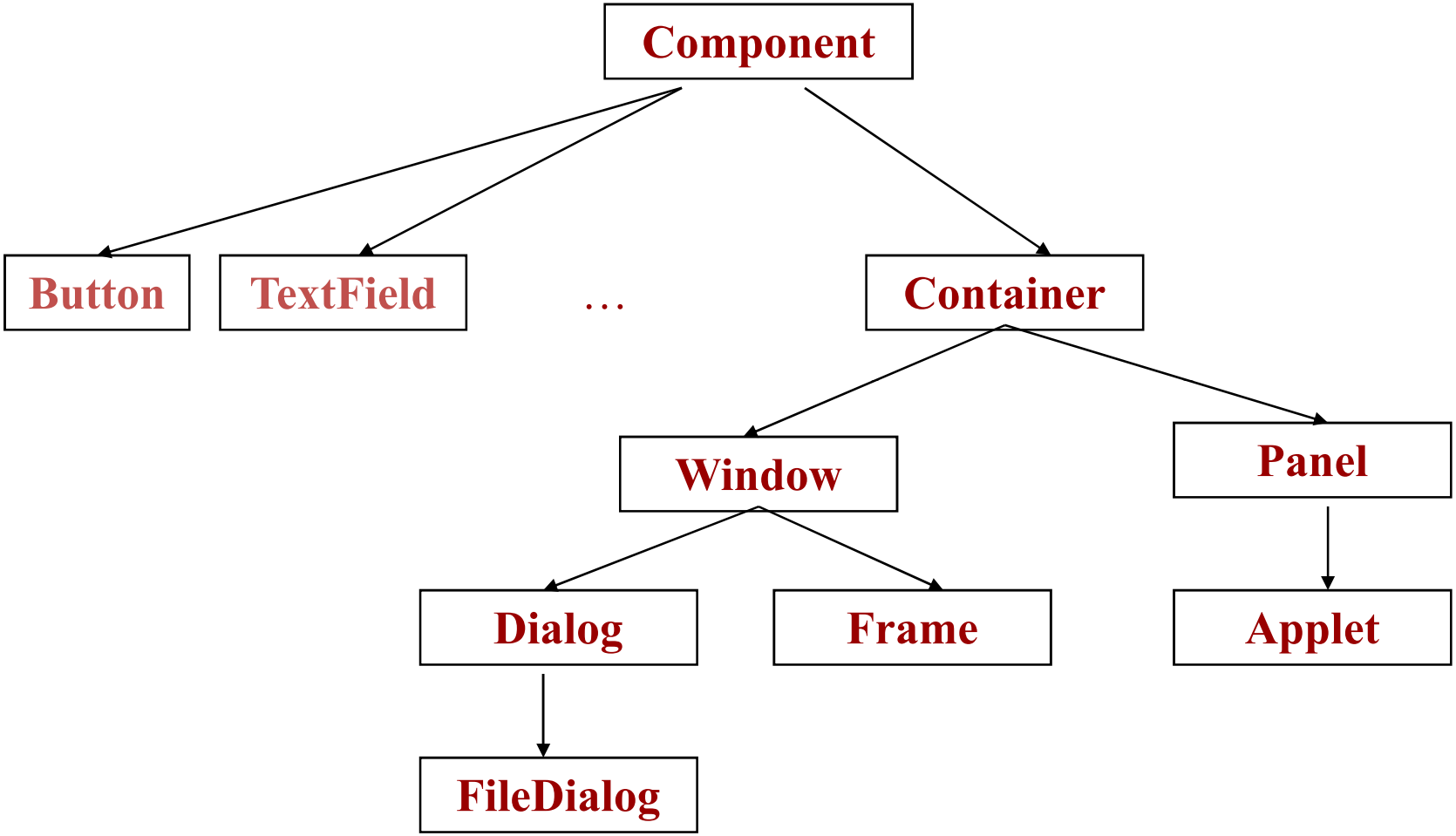
- Swing propose des composants
  - ✓ indépendants de la plate-forme (écrits en Java)
  - ✓ analogues à ceux de AWT
    - JFrame,
    - JButton,
    - JTextField, etc.
  - ✓ avancés :
    - JTabbedPane (onglets),
    - JTree (arbres), etc.

Ma première application avec IHM (GUI)

```
import java.awt.Frame;

public class SimpleGUI
{
    public static void main(String args[]){
      Frame cadre = new Frame("Ma premiere fenetre");
      cadre.setSize(300,200);
      cadre.setVisible(true);
    }
}
```



Hiérarchie des composants AWT



Interfaces graphiques : Composants

- classe **java.awt.Component**

- Comportement d'un objet **Component** :

  - ✓ se dessiner (paint)
  - ✓ définir / obtenir taille et position du composant (setSize, setMinimumSize, etc.)
  - ✓ définir / obtenir propriétés graphiques (setBackground, setForeground, setFont, etc.)
  - ✓ être source d'évènements

Interfaces graphiques : Conteneurs (1/2)

- classe **java.awt.Container**

- Espaces graphiques destinés à recevoir plusieurs composants (boutons, zones de texte, autres conteneurs …) :

  - ✓ Frame,
  - ✓ Panel,
  - ✓ Dialog,
  - ✓ Applet …

Interfaces graphiques : Conteneurs (2/2)

- La **mise en page** d'un conteneur est confiée à un gestionnaire qui se charge du placement des composants ajoutés par la **méthode add().**

- Les panneaux (Panel) doivent être placés dans un espace existant (autre conteneur, browser Web).

Interfaces graphiques : Fenêtres (1/2)

- **java.awt.Window** : sous-classe de **Container**

- Possibilité d'être visible sans nécessiter d'être incluse dans un autre composant.

- Sous-classes de **Window :**

  - ✓ **java.awt.Dialog (**boîtes de dialogue**)**

  - ✓ **java.awt.Frame (**cadres**)**

Interfaces graphiques : Fenêtres (2/2)

- **Cadre :** fenêtre avec *bord*, *titre* et, éventuellement, *menu*.

Une application contient, en général, un seul cadre utilisé pour modifier la taille et la forme de l'IHM.

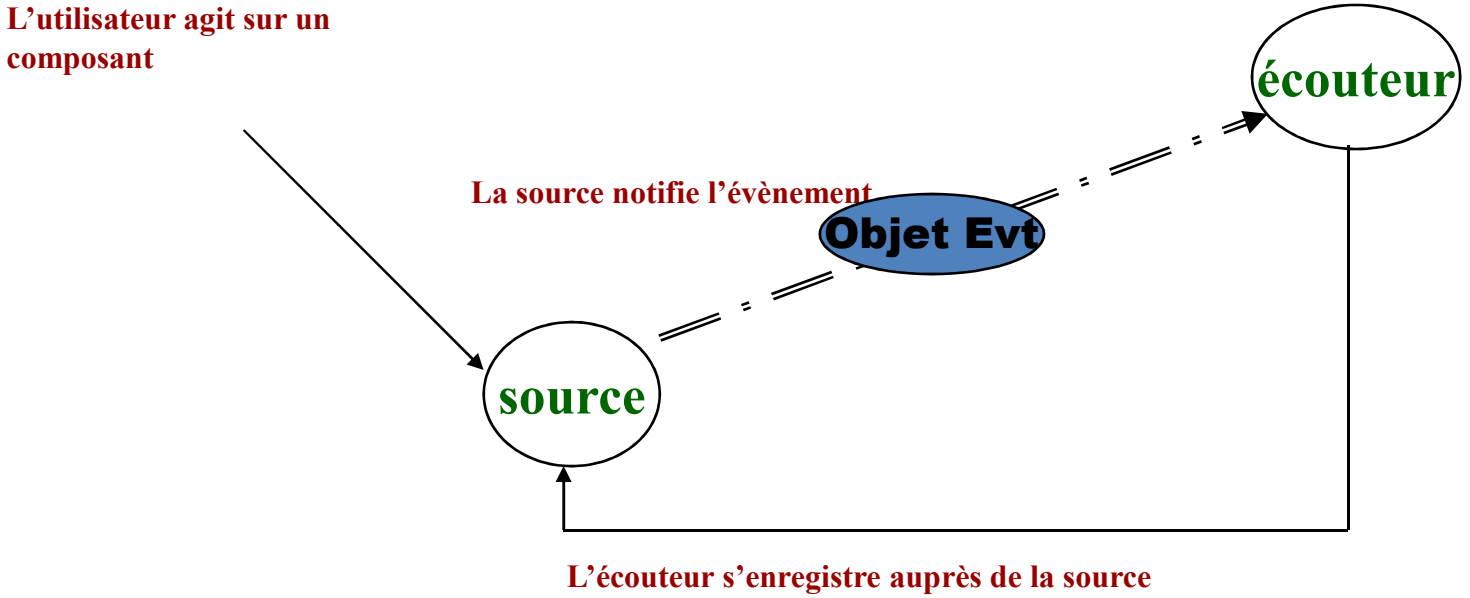
- **Boîte de dialogue :** fenêtre utilisée pour afficher un message, poser une question, etc.

Interfaces graphiques : Exemple 1

```
import java.awt.*;
public class Fenetre extends Frame {
        public Panel p; public TextField texte;
        public Button bouton1, bouton2;
        public Fenetre() {
            super("exemple de fenetre");
            p = new Panel(); this.add(p);
            p.add( new Label("nbre a multplier par 2") );
            p.add( texte = new TextField(20) );
            p.add( bouton1 = new Button("doubler") );
            p.add( bouton2 = new Button("fin") );
            this.pack();
        }
        public static void main(String[] args){
            Fenetre f = new Fenetre();        f.show();
        }
}
```

Interfaces graphiques : événements

- Modèle des évènements de Java (1.2)



Interfaces graphiques : événements

Un événement fait interagir trois objets :

- ✓ L'objet source de l'événement (bouton cliqué)
- ✓ Un objet (classe **ActionEvent**) mémorisant l'événement (date, source, …)
- ✓ Un écouteur d'événement (**listener**) capable de traiter l'événement (s'enregistre en implémentant l'interface correspondante)

Interfaces graphiques : types d'événements (1/2)

- ActionListener : actions (clic sur un bouton, choix dans une liste, etc.) sur un composant.

L'interface contient une seule méthode appelée lorsqu'une action est effectuée.

```
void actionPerformed(ActionEvent e)
```

- MouseListener : événements souris (clics)

- MouseMotionListener : mouvements de la souris

Interfaces graphiques : types d'événements (2/2)

- KeyListener : événements clavier (touche pressée/relâchée).
- FocusListener : acquisition ou perte focus par un composant.
- WindowListener : actions sur une fenêtre.

Interfaces graphiques : Exemple 1 (suite)

```
import java.awt.*; import java.awt.event.*;
public class Fenetre extends Frame implements ActionListener{
    public Panel p; public TextField texte; public Button ...
    public Fenetre() {
        …
        bouton1.addActionListener(this); // la fenetre devient ecouteur
        bouton2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e){
        if (e.getSource()==bouton1){
            double x = Double.parseDouble(texte.getText());
  x = 2*x; texte.setText(“”+x);
        }
        else if (e.getSource()==bouton2){
  this.dispose(); System.exit(0); // destruction fenetre et arret progr.
        }
    }
    public static void main(String[] args){ … }
}
```

Interfaces graphiques : Exemple 2 : Détecter la souris …

- **Les évènements souris sont notifiés par les méthodes suivantes de l'interface MouseListener :**
  - **void mousePressed (MouseEvent e)**
  - **void mouseReleased (MouseEvent e)**
  - **void mouseClicked (MouseEvent e)**
  - **void mouseEntered (MouseEvent e)**
  - **void mouseExited (MouseEvent e)**

- **Un écouteur d'évènements souris "s'enregistre" par :**
  - **void addMouseListener (MouseListener l)**

Interfaces graphiques : Exemple 2 (suite)

```
import java.awt.*;                          import java.awt.event.*;

public class CadreSensible extends Frame implements MouseListener {
  CadreSensible(){
                addMouseListener(this);
                setBounds(100, 100, 300, 200);
                setVisible(true);
  }

  public void mouseClicked(MouseEvent e){
                System.out.println("Clic en ("+ e.getX() + "," + e.getY() + ")");
  }

  public void mousePressed (MouseEvent e){}
  public void mouseReleased(MouseEvent e){}
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}

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

Interfaces graphiques : Gestonnaires de positionnement (1/4)

- **La disposition des composants dans un conteneur est assurée par un gestionnaire de positionnement qui se charge de les placer correctement.**

- **Cela permet de rester indépendant de la plate-forme et d'éviter des calculs fastidieux.**

- **Le gestionnaire par défaut est FlowLayout qui se contente de placer les composants les uns derrière les autres.**

Interfaces graphiques : Gestonnaires de positionnement (2/4)

- **Avec FlowLayout on peut utiliser un paramètre indiquant le type d'alignement à utiliser :**
  - ✓ **FlowLayout.RIGHT pour droite,**
  - ✓ **FlowLayout.LEFT pour gauche,**
  - ✓ **FlowLayout.CENTER pour centré qui est l'alignement par défaut.**

Interfaces graphiques : Gestonnaires de positionnement (3/4)

- **BorderLayout permet de placer les composants en faisant référence à une position de type géographique.**

  ✓ **Pour ajouter un composant, on utilisera la méthode *add* en précisant la position géographique : North, South, East, West ou Center.**

Interfaces graphiques : Gestonnaires de positionnement (4/4)

- **GridLayout permet de placer les composants dans une grille formée de cases de même taille.**

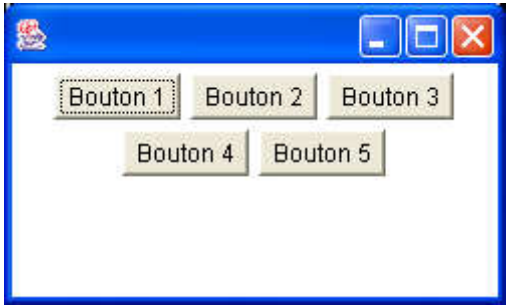
✓ **Chaque composant prend la taille d'une case.**

✓ **Le constructeur de GridLayout attend 2 paramètres : nb de lignes et nb de colonnes de la grille.**

✓ **Lors de leur ajout avec la méthode add, les composants remplissent alors la grille ligne par ligne.**

Interfaces graphiques : Gestonnaires de positionnement (exemple FlowLayout)

```
import java.awt.*;
public class TestFlowLayout1 extends Frame
{
        public TestFlowLayout1(){
  setLayout(new FlowLayout());
  for (int i=1; i<=5; i++)
          add(new Button("Bouton "+i));
  setBounds(100, 100, 250, 150);
  setVisible(true);
        }

public static void main(String args[]){
          new TestFlowLayout1();
        }
}
```

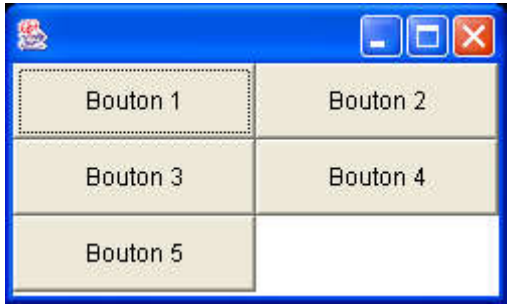


Interfaces graphiques : Gestonnaires de positionnement (exemple BorderLayout)

```
public class Fenetre extends Frame {
    public Panel p, p_sud; public TextField texte; public Button ...

    public Fenetre() {
      super("exemple de fenetre");
      p = new Panel(new BorderLayout()); this.add(p);
      p.add( new Label("nbre a multplier par 2"),BorderLayout.NORTH);
      p.add( texte = new TextField(20) ,BorderLayout.CENTER);

      p_sud=new Panel(new FlowLayout());
      p.add(p_sud, BorderLayout.SOUTH);

      p_sud.add( bouton1 = new Button("doubler") );
      p_sud.add( bouton2 = new Button("fin") );
      this.pack();
   }
   …
}
```

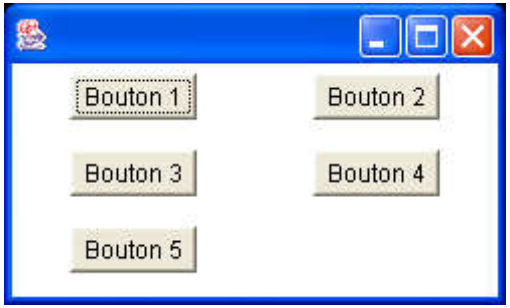
Interfaces graphiques : Gestonnaires de positionnement (exemple GridLayout)

```
import java.awt.*;
public class TestGridLayout extends Frame
{
          public TestGridLayout(){
                    setLayout(new GridLayout(3,2));
                    for (int i=1; i<=5; i++)
                              add(new Button("Bouton "+i));
                    setBounds(100, 100, 250, 150);
                    setVisible(true);
        }
          public static void main(String args[]){
             new TestGridLayout();
          }
}
```



Interfaces graphiques : Gestonnaires de positionnement (autre exemple GridLayout)

```
import java.awt.*;
public class TestGridLayout1 extends Frame
{
  public TestGridLayout1(){
          setLayout(new GridLayout(3,2));
          Panel p;
          for (int i=1; i<=5; i++){
            p = new Panel();
            p.add(new Button("Bouton "+i));
            add(p);
          }
          setBounds(100, 100, 250, 150);
          setVisible(true);
    }

   public static void main(String args[]){
              new TestGridLayout1();
    }
}
```



Interfaces graphiques : Composants Swing de saisie et de choix

- ✓ **JTextField :** champ de saisie simple (zone de texte)
- ✓ **JPasswordField** : champ de saisie pour les passwords
- ✓ **JTextArea** : champ de saisie multiligne
- ✓ **JButton** : bouton
- ✓ **JCheckBox** : boite à cocher
- ✓ **JRadioButton** : bouton radio
- ✓ **JList** : liste de valeurs
- ✓ **JComboBox** : liste deroulante
- ✓ etc.

: Composants Swing arbre et tableau

- ✓ **JTree :** arbre hiérarchique
- ✓ **JTable** : tableau capable d'afficher différents types de données

: Composants Swing conteneurs

- ✓ **JFrame :** fenêtre avec bord et barre de titre
- ✓ **JWindow** : fenêtre sans décoration
- ✓ **JDialog** : boîte de dialogue
- ✓ **JPanel :** panneau vide
- ✓ **JTabbedPane** : panneau à onglets
- ✓ **JSplitPane** : panneau partagé (horizontalemt ou verticalemt)
- ✓ **JScrollPane** : panneau à ascenseur
- ✓ Etc.

: Composants Swing boîtes de dialogue

- ✓ **JFileChooser :** choix de fichier
- ✓ **JOptionPane** : dialogues standards : message, confirmation
- ✓ Etc.

: Composants Swing menus

- ✓ **JMenuBar :** barre de menu
- ✓ **JMenu** : menu
- ✓ **JMenuItem** : élément de menu
- ✓ **JPopupMenu** : menu contextuel
- ✓ Etc.

Interfaces graphiques : Exemple Swing (1/3)

Interfaces graphiques : Exemple Swing (2/3)

```
import java.awt.*;
import javax.swing.*;
class Authentification {
public static void main (String[] args){
        JFrame fen = new JFrame("Identification");
        Container p = fen.getContentPane();
        p.setLayout(new BorderLayout());

        JPanel p_nord = new JPanel();
        p.add(p_nord,BorderLayout.NORTH);
        p_nord.add(new JLabel("Login :"));
        p_nord.add(new JTextField(10));

        JPanel p_centre = new JPanel();
        p.add(p_centre,BorderLayout.CENTER);
        p_centre.add(new JLabel("Mot de passe :"));
        p_centre.add(new JPasswordField(10));
```

Interfaces graphiques : Exemple Swing (3/3)

```
        JPanel p_sud = new JPanel();
        p.add(p_sud,BorderLayout.SOUTH);
        p_sud.add(new JButton("Valider"));
        p_sud.add(new JButton("Annuler"));

        fen.pack();

        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        fen.setVisible(true);
}
}
```

APPLETS

Les applets (1)

• Une **applet** est un programme Java interactif, présent sur le serveur Web, qui peut être téléchargé sur le client Web et s'exécute dans une page Web.

• Elle permet d'améliorer la présentation de cette dernière ou de remplir une fonction spécifique.

• Les applets sont considérées comme des **objets des pages HTML** qui les contiennent.

• Les applets n'ont pas accès, comme les applications Java, à toutes les ressources du système sur lequel elles s'exécutent.

Les applets (2)

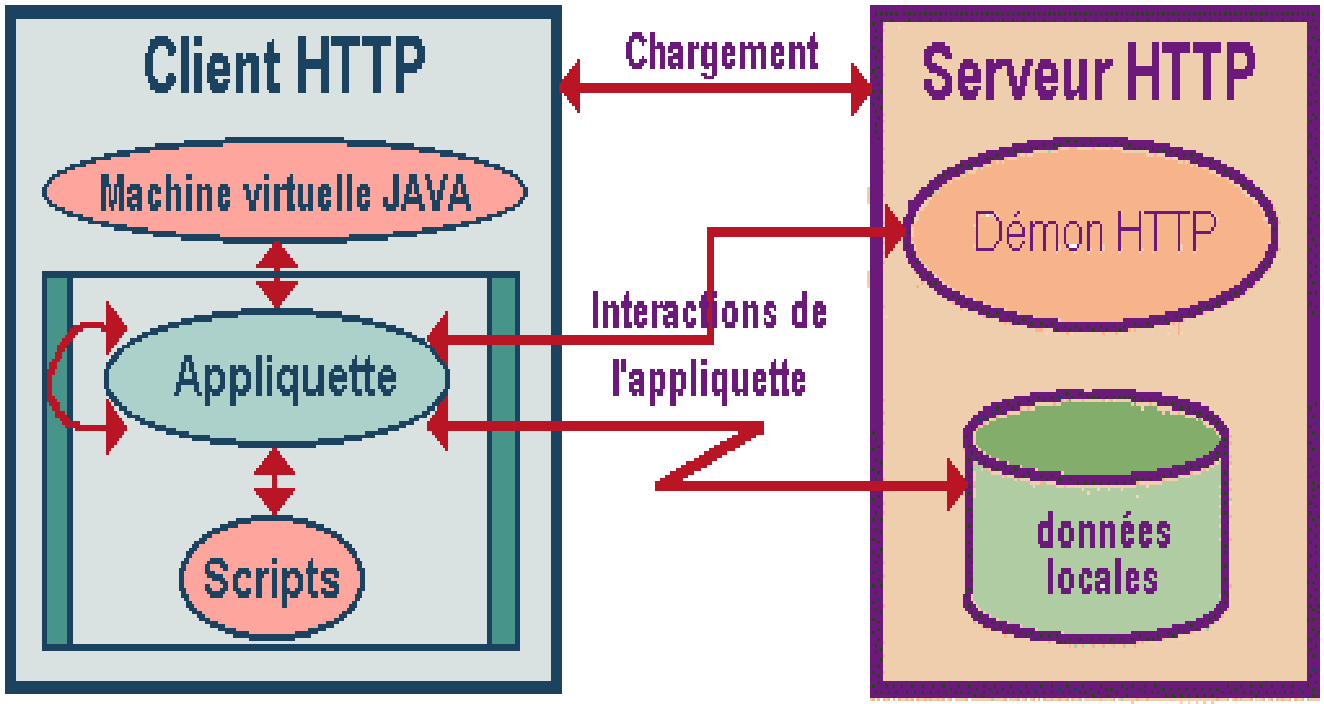
Pour des raisons de sécurité, une applet ne peut pas:

- lire ou écrire le système de fichiers de la machine où elle s'exécute,

- exécuter des programmes,

- communiquer avec d'autres machines, à part le serveur à partir duquel elle a été téléchargée.

Les applets (3)



Appliquette JAVA

Les applets (4)

Pour pouvoir créer et exécuter une applet, il faut écrire deux fichiers différents:

- un fichier source *.java* et
- un fichier *.html*.

Exemple: une applet qui affiche "Bonjour »

- *Bonjour.java*
- *Bonjour.html*

Les applets (5)

Bonjour.java:

```
import java.awt.*;
import javax.swing.*;

public  class Bonjour extends JApplet
{
      public void init {
            Container p = getContentPane();
            p.setLayout(new FlowLayout());
            Jlabel label = new Jlabel("Bonjour!");
            p.add(label);
      }
}
```

Les applets (6)

Bonjour.html

```
<HTML>
    <HEAD>
        <TITLE> Une applet simple </TITLE>
    </HEAD>
    <BODY>
        Voici le résultat :
        <APPLET CODE="Bonjour.class" WIDTH=150
        HEIGHT=25>
        </APPLET>
    </BODY>
</HTML>
```

La classe JApplet

- Se trouve dans le package ***javax.swing***

- Hérite de la classe ***java.awt.Applet***.

- La classe **Applet** étend la classe **Panel** qui étend la classe **Container**.

- Les objets **JApplet** sont des conteneurs.

La classe Applet (1)

• La **classe Applet** est une classe dérivée de la classe ***java.awt.Component*** qui décrit les objets graphiques.

• Elle surcharge des méthodes de cette classe:

- **paint(), update()** (pour dessiner et afficher dans la fenêtre de l'applet)

- et **handleEvent()** (gestion d'événements relatifs à la souris et au clavier).

La classe Applet (2)

Elle contient aussi quatre méthodes pour traiter les événements correspondant à la vie de l'applet:

- **init():** appelée au moment du chargement de l'applet
- **start():** pour démarrer l'exécution de l'applet
- **stop():** pour arrêter l'exécution de l'applet
- **destroy():** pour libérer les ressources ou interrompre certaines activités (animation par exemple)

La balise HTML Applet (1)

- Elle permet d'introduire une applet dans un document HTML.
- Lorsque le browser rencontre la balise Applet, il fait charger le bytecode (fichier .*class*) depuis le serveur Web.

```
<APPLET
    CODE   = nom du fichier java
    WIDTH  = taille initiale de l'applet en pixels
    HEIGTH = taille initiale de l'applet en pixels
    Attributs optionnels
>
```

- On peut passer des arguments à une applet en utilisant la balise HTML **PARAM** et la méthode **getParameter()** de Applet.

Applet avec paramètres (1)

```
import java.awt.*;
import java.applet.*;

public class Bonjour extends Applet
{
        String s1, s2;
        public void init() {
                s1 = getParameter("param1");
                s2 = getParameter("param2");
        }
        public void paint (Graphics g){
                g.drawString(s1 + "          " + s2, 50, 20);
        }
}
```

Applet avec paramètres (2)

```
<HTML>
    ...
    <BODY>
        <applet code=Bonjour.class width=200 heigth=100>
        <param name = param1 value = "Bonjour">
        <param name = param2 value = "Comment vas tu?">
        </applet>
    </BODY>
</HTML>
```

Méthodes avec nombre d'arguments variable

Méthodes avec nombre d'arguments variable

✓ Permet d'appeler une même méthode avec une liste d'arguments variable.

Exemple 1 :

```
public class MultiArgs {
        static void ecrireLesMots (String ... mots){
                for (String mot : mots)
                        System.out.print(mot + "  ");
                System.out.println();
        }
        public static void main(String arg[]){
                ecrireLesMots("UGB UFR SAT");
                ecrireLesMots("MIAGE INFO »);
        }
}
```

Méthode avec nombre d'arguments variable

Exemple 2 :

```
public class MultiArgs {
    static double moyenne(String nom, Number... notes) {
        int nombre = notes.length;
        if (nombre > 0) {
            double somme = 0;
            for (Number x : notes)
                somme += x.doubleValue();
            return somme / nombre;
        }
        return -1;
        }
        public static void main(String arg[]){
            double moy = moyenne("Amy",10, 12.4, 18);
            System.out.println(moy);
        }
}
```

Importation de membres statiques

Importation de membres statiques

- ✓ Classiquement, on n'importe que des noms de classes.
- ✓ **Java 5 permet d'importer des membres statiques.**

Exemple :

```
import static java.lang.System.out;
import static java.lang.Math.random;

public class TestImportStatic {
    public static void main(String arg[]){
            out.println("voici un nb aleatoire :"+random());
    }
}
```

COLLECTIONS

Limites des tableaux (1/2)

- Simples à programmer et peu gourmands en mémoire, les tableaux sont utilisés pour gérer des ensembles dont la taille est connu à l'avance.

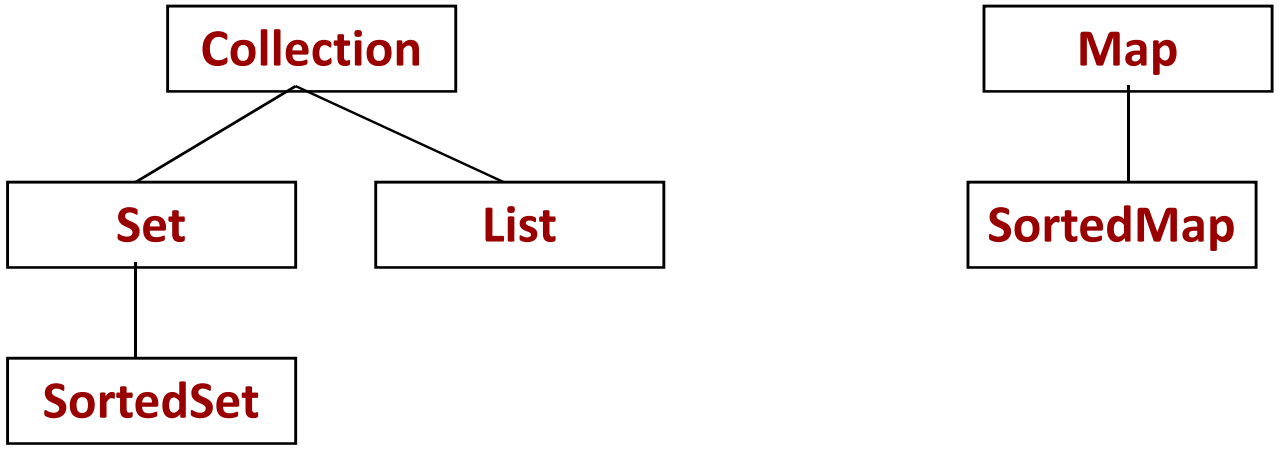
- Les tableaux sont inadéquats pour gérer une quantité importante d'informations du même type lorsque leur nombre n'est pas connu à l'avance.

Limites des tableaux (2/2)

- Les tableaux ne sont pas redimensionnables.
- L'insertion d'un élément au milieu d'un tableau n'est pas aisée.
- La recherche d'un élément dans un grand tableau est longue s'il n'est pas trié.
- Les indices pour accéder aux éléments d'un tableau doivent être entiers.

Collections Java

- La package ***java.util*** fournit un ensemble d'interfaces et de classes facilitant la manipulation de collections d'objets.
- Une **collection** est un objet qui sert à stocker d'autres objets.
- Interfaces sur les collections organisées en deux catégories : Collection et Map.



Collections Java : interfaces

- **Collection** : groupe d'objets où la duplication peut être autorisée.

L'interface Collection spécifie des méthodes comme add, remove, addAll, removeAll, contains, containsAll, size, etc.

- **Set** : objets collections non ordonnées et sans doublons. SortedSet est un Set trié.

- **List** : collections ordonnées autorisant des doublons. Chaque objet possède une position dans la séquence et l'interface offre des méthodes permettant l'accès direct à un objet donné.

- **Map** : tableaux associatifs (dictionnaires) permettant de retrouver une information grâce à un index. SortedMap est un Map trié.

Collections Java : implémentations

	Classes d'implémentation				
		Table de Hachage	Tableau (Vecteur)	Arbre	Liste chaînée
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Les classes du package java.util

- **ArrayList** : idéale pour ajouter à la suite les uns des autres des éléments dans un ensemble ordonné.

- **LinkedList** : idéale pour insérer de nombreux éléments au milieu d'un ensemble ordonné.

- **HashSet** : pour gérer un ensemble dont chaque élément doit être unique. Les performances de recherche sont améliorées.

- **TreeSet** : idéale pour gérer un ensemble trié d'objets uniques.

- **HashMap** : idéale pour accéder aux éléments d'un ensemble grâce à une clé.

- **TreeMap** : idéale pour gérer un ensemble d'éléments trié dans l'ordre de leur clé d'accès.

Collections Java : Itérateurs

- Les collections peuvent être parcourues à l'aide d'itérateurs : interface Iterator.

- *Exemple :*

```
import java.util.*;
public class test_iterateur {
    public static void main (String[] args) {
            Set ufrs = new TreeSet();
            ufrs.add("sat"); ufrs.add("seg"); ufrs.add("sjp");
            // iterer
            Iterator iter = ufrs.iterator();
            while (iter.hasNext()) {
                String ufr = (String) iter.next();
                System.out.println(ufr);
            }
        }
}
```

La boucle "for each" et les collections

La boucle « *for each* » permet le parcours d'une instance d'une classe implémentant l'interface *java.util.Iterable*.

```
// affichage des elements d'un ensemble
    Set ufrs = new TreeSet();
    ufrs.add("sat");
    ufrs.add("seg");
    ufrs.add("sjp");
    ufrs.add("lsh");

    for (Object s : ufrs)
        System.out.print(s + "  ");
// s va prendre successivement les valeurs contenues dans ufrs
```

Collections Java : Map

- Les objets Map permettent d'établir une correspondance entre deux classes d'objets (*par exemple des termes anglais et français*).

- Ensemble de paires (clé, valeur) tq l'ensemble des clés est sans doublons.

- L'interface interne Entry permet de manipuler les éléments d'une paire au moyen des méthodes :

  ✓ getKey et getValue : retournent respectivement la clé et la valeur associée à cette clé.

  ✓ setValue : permet de modifier la valeur d'une paire.

Collections Java : Map

L'interface ***Map*** offre des méthodes permettant d'itérer sur les clés, les valeurs et sur les paires:

```
Map m = new HashMap();
// sur les clés
for (Iterator i = m.keySet().iterator(); i.hasNext();)
    System.out.println(i.next());
 // sur les valeurs
for (Iterator i = m.values().iterator(); i.hasNext();)
    System.out.println(i.next());
// sur les paires clé/valeur
for (Iterator i = m.EntrySet().iterator(); i.hasNext();) {
    Map.Entry e = (Map.Entry)i.next();
    System.out.println(e.getKey() + "/" + e.getValue());
}
```

Exemple : gérer un glossaire avec HashMap

```
import java.util.HashMap;
import javax.swing.JOptionPane;

public class Glossaire {
    public static void main (String arg[]){
    String definitionInstance = "Objet cree a partir d'une classe";
    String definitionCollection = "Instance d'une classe gerant un ensemble
d'elements";
    String definitionSousClasse = "Classe heritant d'une autre classe";

    HashMap glossaire = new HashMap();
    glossaire.put("instance",definitionInstance);
    glossaire.put("collection",definitionCollection);
    glossaire.put("sous classe",definitionSousClasse);
    glossaire.put("classe derivee",definitionSousClasse);

 ...
```

Exemple : gérer un glossaire avec HashMap

```
while (true){
   String motCherche = JOptionPane.showInputDialog("Que cherchez vous ?");
   if (motCherche == null) System.exit(0);
   String definition = (String)glossaire.get(motCherche);
   if (definition != null)
                    JOptionPane.showMessageDialog(null,
                    motCherche + " : " + definition,
                    "Resultat de la recherche",
                    JOptionPane.INFORMATION_MESSAGE);
   else
                    JOptionPane.showMessageDialog(null,
                    motCherche + " : " + "non defini",
                    "Resultat de la recherche",
                    JOptionPane.WARNING_MESSAGE);
  }
  }
}
```

JDBC

JDBC

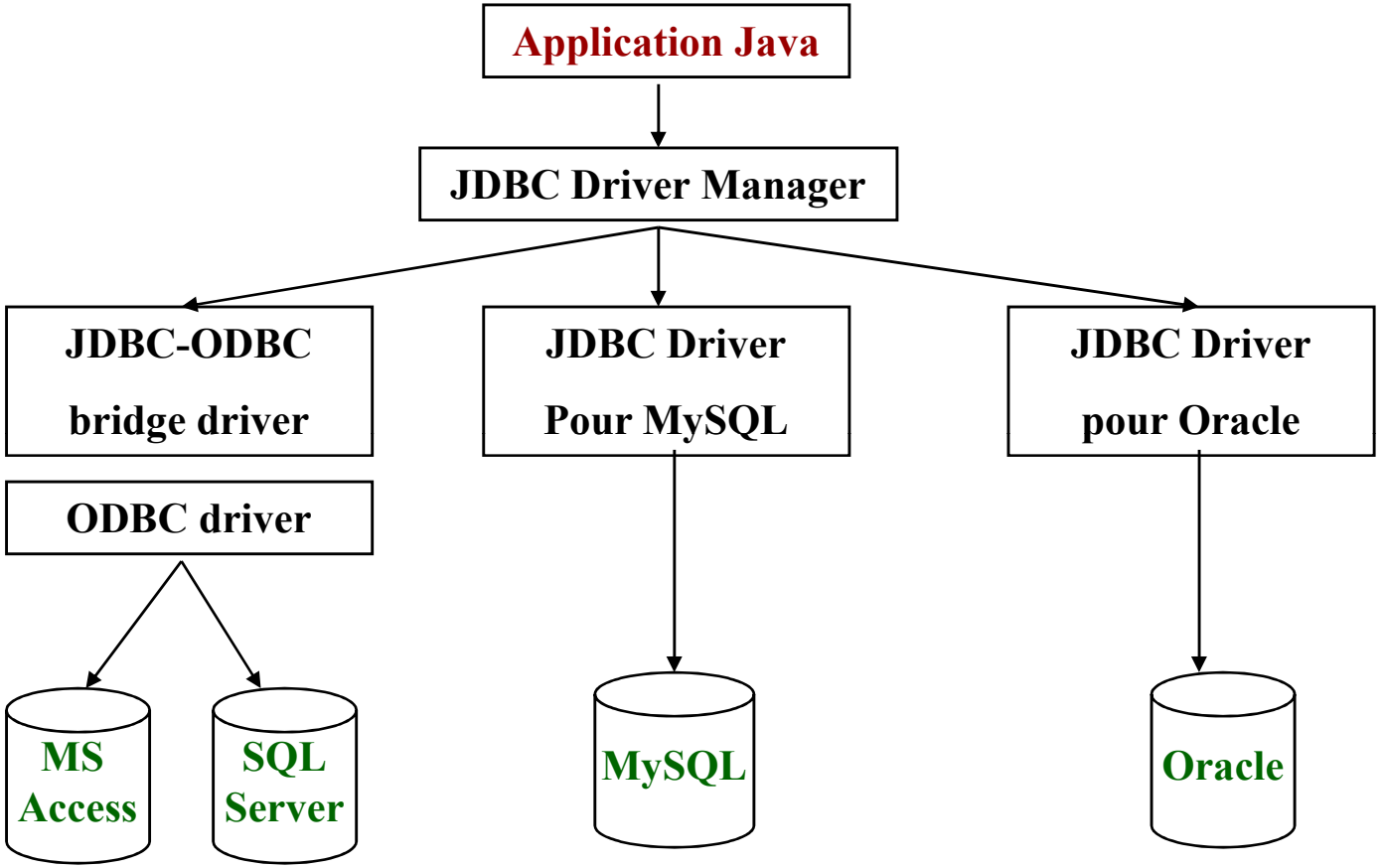
- *Java Database Connectivity*

- API Java permettant l'accès aux SGBDR (MS Access, SQL Server, MySQL, PostgreSQL, Oracle)

- Fonctionne en client/serveur (appl. Java / SGBD)

- Les classes de l'API JDBC se trouvent dans le package ***java.sql***

JDBC : architecture



JDBC : architecture

- JDBC Driver Manager : Gestionnaire de drivers permettant à chaque application de charger le(s) driver(s) dont il a besoin.

- Driver JDBC : gère les détails de communication avec un type de SGBD.

  ✓ Un driver par SGBD (Oracle, MySQL, ...)

  ✓ JDBC-ODBC : driver générique pour toutes les sources accessibles via ODBC (*Open DataBase Connectivity*. interface Microsoft permettant la communication entre des clients bases de données fonctionnant sous Windows et les SGBD du marché).

JDBC : le driver JDBC

- Chaque éditeur de SGBDR fournit un driver JDBC sous la forme d'une archive jar.

- C'est un ensemble de classes qui implémentent les interfaces du package *java.sql*.

- Il faut ajouter l'archive au CLASSPATH afin de pouvoir y accéder dans vos programmes.

JDBC : fonctionnement

JDBC fonctionne comme suit :

- Création d'une connexion à la BD
- Envoi de requêtes SQL (pour récupérer ou maj des données)
- Exploitation des résultats provenant de la base
- Fermeture de la connexion

JDBC : connexion à un SGBD

1. Charger la classe du driver JDBC

Cette classe implémente l'interface java.sql.Driver et peut être chargée en appelant la méthode ***forName*** de ***java.lang.Class***

Exemple avec le driver de MySQL

```
Class.forName("org.gjt.mm.mysql.Driver");
```

2. Appeler la méthode getConnection de java.sql.DriverManager

```
java.sql.Connection co;
co = DriverManager.getConnection("jdbc:mysql://localhost/MABD");

co = DriverManager.getConnection("jdbc:mysql://localhost/MABD,
                                 "admin", "passer");
```

JDBC : exemple de connexion

```
import java.sql.*; import javax.swing.JOptionPane;
public static Connection initConnection() {
Connection co = null;
String url = "jdbc:mysql://localhost/MABD";
try{
        Class.forName("org.gjt.mm.mysql.Driver");
        co = DriverManager.getConnection(url,"root",null);
        JOptionPane.showMessageDialog(null,"Connection OK");
        // co.close();
        return co;
}
catch (ClassNotFoundException fe) {
   System.out.println("driver introuvable : " +fe.getMessage());
}
catch (SQLException se) {
   System.out.println("connexion impossible : " +se.getMessage());
}
}
```

JDBC : requêtes SQL

- JDBC permet divers types de requêtes SQL : interrogation, maj, création de tables.

- Les objets suivants sont disponibles :

  ✓ **ResultSet** : contient des informations sur une table (noms des colonnes) ou le résultat d'une requête SQL.

```
Statement st = co.createStatement();
ResultSet rs = (ResultSet)st.executeQuery("Select …)";
```

  ✓ **ResultSetMetaData** : contient des informations sur le nom et le type des colonnes d'une table

```
ResultSetMetaData rsmd = rs.getMetaData();
int nbre_Colonnes = rsmd.getColumnCount();
```

  ✓ **DataBaseMetaData** : contient les informations sur la BD (noms des tables, index, etc.)

JDBC : l'interface java.sql.Connection (1/2)

- **createStatement** : retourne une instance de *java.sql.Statement* utilisée pour éxecuter une instruction SQL sur la base de données

- **prepareStatement** :précompile des instructions SQL paramétrées et retourne une instance de *java.sql.PreparedStatement*

- **prepareCall** : prépare l'appel à une procédure stockée et renvoie un instance de *java.sql.CallableStatement*

JDBC : l'interface java.sql.Connection (2/2)

- **setAutoCommit, commit, rollback** : gèrent les transactions

- **getMetaData** : renvoie une instance de *java.sql.DatabaseMetaData* pour obtenir des informations sur la base de données

- **close, isClosed** : gèrent la fermeture de la connection

JDBC : java.sql.Statement

- La méthode createStatement d'une connection retourne une instance de *java.sql.Statement* dont les méthodes les plus utilisées sont :

✓ **executeUpdate** : permet de mettre à jour les données d'une base en exécutant des instructions SQL de maj

✓ **executeQuery** : permet d'exécuter des requêtes sélection; renvoie une instance de *java.sql.ResultSet*

JDBC : java.sql.ResultSet (1/2)

- Permet de récupérer et d'exploiter les résultats d'une requête Sélection

- Des méthodes **next, first, last** permettent de parcourir la liste des enregistrement retournés par la sélection SQL

```
java.sql.ResultSet rs = st.executeQuery("Select …");
while (rs.next()) {
        // interrogation des infos de l'enregistrement courant
}
```

JDBC : java.sql.ResultSet (2/2)

- Des méthodes get (getString, getInt, getDate, getObject, etc) renvoient la valeur d'un des champs de l'enregistrement.

```
System.out.println(rs.getString(1), rs.getString("prenom"),
rs.getDouble(3));
```

JDBC : exemple de requête SQL

```
public test_jdbc {
public static void main (String[] args) {
        Connection maCo = initConnection();
        if (maCo == null) return;
        String req = "Select nom, prenom, age from Personne";
        try{
            Statement st = maCo.createStatement();
            ResultSet rs = st.executeQuery(req);
            while (rs.next()) {
                System.out.print("nom :"+rs.getString("nom"));
                System.out.print("prenom:"+rs.getString(2));
                System.out.println("age :"+rs.getDouble(3));
            }
            rs.close(); st.close(); maCo.close();
        }
        catch (SQLException se) {
                System.out.println("connexion impossible");
        }
}
}
```