

L'Essentiel de Java et de la programmation orientée objet

Objectifs

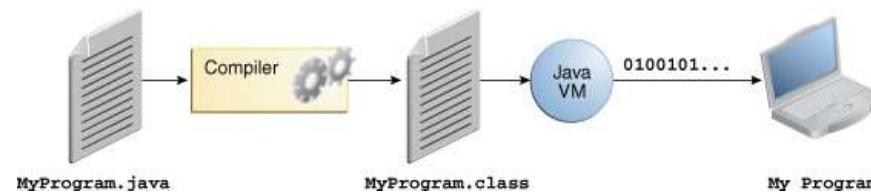
- Mettre en œuvre les principes de la Programmation Orientée Objet
- Maîtriser la syntaxe du langage Java
- Maîtriser les principales librairies standards Java
- Maîtriser un environnement de développement intégré pour programmer en Java

Plan

- Introduction
 - Les variables
 - Les conditions
 - Les collections
- L'approche objet
- Classe & Héritage
- Classes abstraites et interfaces
- Exceptions
- Entrée/Sorties
- Interface graphique

Qu'est ce que java ?

- Java est un langage de programmation orientée objet apparu dans les années 90. Il permet de développer des applications lourdes, des servlets, des applets...
- Le développeur écrit un code source **java (.java)**
- Il lance sa compilation en **bytecode (.class)**
- L'utilisateur exécute le bytecode via sa machine virtuelle java (**JVM**)
 - la JVM est fournie par un runtime java (JRE)
 - elle traduit le bytecode (indépendant de la plateforme) en code natif (spécifique à la plateforme) → **portabilité** du programme



Les commentaires

- `/*` commentaire sur une ou plusieurs lignes `*/`
 - **Identiques à ceux existant dans le langage C**
- `//` commentaire de fin de ligne
 - **Identiques à ceux existant en C++**
- `/**` commentaire d'explication `*/`
 - **Les commentaires d'explication se placent généralement juste avant une déclaration (d'attribut ou de méthode)**
 - **Ils sont récupérés par l'utilitaire javadoc et inclus dans la documentation ainsi générée.**

Instructions, blocs et blancs

- Les instructions Java se terminent par un ;
- Les blocs sont délimités par :

{ pour le début de bloc

} pour la fin du bloc

Un bloc permet de définir un regroupement d'instructions. La définition d'une classe ou d'une méthode se fait dans un bloc.

- Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.

Point d'entrée d'un programme Java

- Pour pouvoir faire un programme exécutable il faut toujours une classe qui contienne une méthode particulière, la méthode « main »
 - c'est le point d'entrée dans le programme : le microprocesseur sait qu'il va commencer à exécuter les instructions à partir de cet endroit

```
public static void main(String arg[ ])  
{  
    .../  
}
```

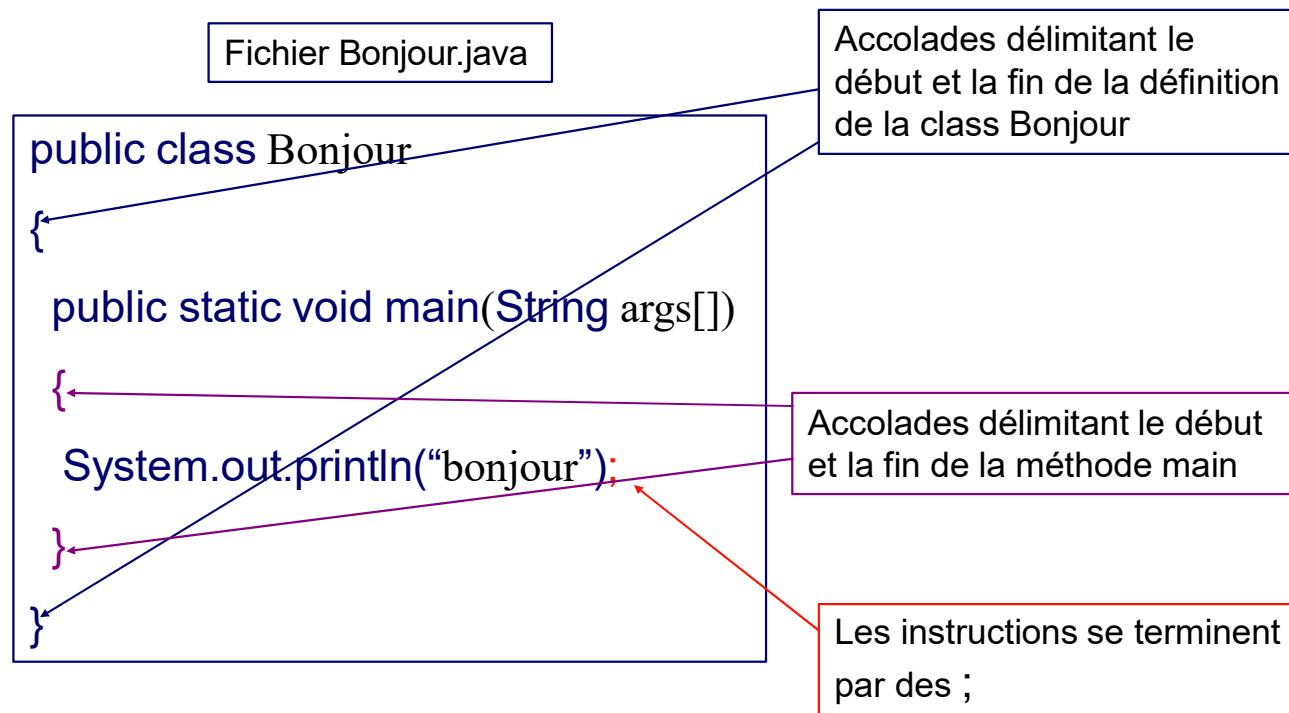
Exemple (1)

Fichier Bonjour.java

```
public class Bonjour
{ //Accolade débutant la classe Bonjour
  public static void main(String args[])
  { //Accolade débutant la méthode main
    /* Pour l'instant juste une instruction */
    System.out.println("bonjour");
  } //Accolade fermant la méthode main
} //Accolade fermant la classe Bonjour
```

La classe est l'unité de base de nos programmes. Le mot clé en Java pour définir une classe est **class**

Exemple (2)



Exemple (3)

Fichier Bonjour.java

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

Une méthode peut recevoir des paramètres. Ici la méthode main reçoit le paramètre args qui est un tableau de chaîne de caractères.

Identificateurs (1)

- On a besoin de nommer les classes, les variables, les constantes, etc. ; on parle d'identificateur.
- Les identificateurs commencent par une lettre, _ ou \$

Attention : Java distingue les majuscules des minuscules

- Conventions sur les identificateurs :
 - Si plusieurs mots sont accolés, mettre une majuscule à chacun des mots sauf le premier.
 - exemple : uneVariableEntiere
 - La première lettre est majuscule pour les classes et les interfaces
 - exemples : MaClasse, UneJolieFenetre

Identificateurs (2)

- Conventions sur les identificateurs :
 - La première lettre est minuscule pour les méthodes, les attributs et les variables
 - exemples : setLongueur, i, uneFenetre
 - Les constantes sont entièrement en majuscules
 - exemple : LONGUEUR_MAX

Les mots réservés de Java

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

Présentation de l'environnement de développement

- Ce dont on a besoin pour pouvoir programmer en Java:
 - Java Development Kit (JDK) : outils de développement nécessaires à cette étape.
 - javac (compilateur), jar (archiveur), javadoc (générateur de doc), jdb (debugger)
 - Potentiellement un Environnement de Développement Intégré (IDE)

Avantages d'un IDE

- Coloration syntaxique
- Affichage structuré des sources sous forme de projet
- Compilation en temps réel après chaque sauvegarde
- Auto complétion du code
- Gestion des dépendances (librairies externes)
- Raccourci de codes:
 - Ctrl+shift+o : import automatique des librairies
 - Ctrl+space : auto completion courante

Les variables

Variables et types

- Un programme est amené à mémoriser des informations qui vont évoluer au fil de son exécution. Elles sont stockées dans des **variables**.
- Une variable est une donnée manipulée par un nom qui représente son identifiant
- Ces informations peuvent être de natures différentes (valeur numérique, chaîne de caractères, liste de contacts...). Leur nature se décline grâce à un **type**.
- Java fournit de nombreux types de bases.

Variables et types

- Les types entiers:
 - Ils servent à représenter les nombres entiers relatifs.
- Les différents types d'entiers:

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

Variables et types

- Les types flottants:
 - Ils servent à représenter une approximation des nombres réels.
- Les différents types de flottants:

Type	Taille (octets)	Précision (chiffres significatifs)	Valeur absolue minimale	Valeur absolue maximale
float	4	7	1.7e-308 (Float.MIN_VALUE)	1.7e+308 (Float.MAX_VALUE)
double	8	15	-3.4e-38 (Double.MIN_VALUE)	3.4e+38 (Double.MAX_VALUE)

Variables et types

- Le type caractère:
 - Comme la plupart des langages, Java permet de manipuler des caractères. Cependant, Java fournit une représentation mémoire sur 2 octets en utilisant l'unicode.
- Une variable de type caractère se déclare en utilisant le mot-clé *char* :
 - `char c1, c2; // c1 et c2 sont deux variables de type caractère`
- Le type booléen:
 - Ce type sert à représenter une valeur logique du type vrai/faux.
 - Les deux constantes du type booléen se notent *true* et *false*
- Une variable de type booléen se déclare en utilisant le mot-clé *boolean*
 - `boolean var ;`

Variables et types

- Initialisation d'une variable:
 - Une variable peut recevoir une valeur initiale lors de sa déclaration:
 - `int n = 3 ;`
 - Une variable n'ayant pas encore reçue de valeur ne peut pas être utilisée, sous peine de générer une erreur de compilation.
- Exemple:

```
package test;

public class Hello{
    public static void main( String [] argv ) {
        int x = 7;
        System.out.println(x); }
}
```

Les types de bases (1)

- En Java, tout est objet sauf les types de base.
- Il y a huit types de base :
 - un type booléen pour représenter les variables ne pouvant prendre que 2 valeurs (vrai et faux, 0 ou 1, etc.) : **boolean** avec les valeurs associées **true** et **false**
 - un type pour représenter les caractères : **char**
 - quatre types pour représenter les entiers de divers taille : **byte**, **short**, **int** et **long**
 - deux types pour représenter les réelles : **float** et **double**
- La taille nécessaire au stockage de ces types est indépendante de la machine.
 - Avantage : portabilité
 - Inconvénient : "conversions" coûteuses

Les types de bases (2) : les entiers

- Les entiers (avec signe)
 - **byte** : codé sur 8 bits, peuvent représenter des entiers allant de -2^7 à $2^7 - 1$ (-128 à +127)
 - **short** : codé sur 16 bits, peuvent représenter des entiers allant de -2^{15} à $2^{15} - 1$
 - **int** : codé sur 32 bits, peuvent représenter des entiers allant de -2^{31} à $2^{31} - 1$
 - **long** : codé sur 64 bits, peuvent représenter des entiers allant de -2^{63} à $2^{63} - 1$

Les types de bases (3) : les entiers

- Notation
 - 2 entier normal en base décimal
 - 2L entier au format long en base décimal
 - **0**10 entier en valeur octale (base 8)
 - **0x**F entier en valeur hexadécimale (base 16)
- Opérations sur les entiers
 - opérateurs arithmétiques +, -, *
 - / :division entière si les 2 arguments sont des entiers
 - % : reste de la division entière
 - exemples :
 - 15 / 4 donne 3
 - 15 % 2 donne 1

Les types de bases (4) : les entiers

- Opérations sur les entiers (suite)
 - les opérateurs d'incrémentation ++ et de décrémentation --
 - ajoute ou retranche 1 à une variable
`int n = 12;`
`n ++; //Maintenant n vaut 13`
 - `n++`; « équivalent à » `n = n+1`;
`n--`; « équivalent à » `n = n-1`;
 - `8++`; est une instruction illégale
 - peut s'utiliser de manière suffixée : `++n`. La différence avec la version préfixée se voit quand on les utilisent dans les expressions. En version suffixée la (dé/inc)rémentation s'effectue en premier

```
int m=7; int n=7;  
int a=2 * ++m; //a vaut 16, m vaut 8  
int b=2 * n++; //b vaut 14, n vaut 8
```

Les types de bases (5) : les réels

- Les réels
 - float : codé sur 32 bits, peuvent représenter des nombres allant de -10^{35} à $+10^{35}$
 - double : codé sur 64 bits, peuvent représenter des nombres allant de -10^{400} à $+10^{400}$
- Notation
 - 4.55 ou 4.55**D** réel double précision
 - 4.55**f** réel simple précision

Les types de bases (6) : les réels

- Les opérateurs
 - opérateurs classiques +, -, *, /
 - attention pour la division :
 - 15 / 4 donne 3 ***division entière***
 - 15 % 2 donne 1
 - 11.0 / 4 donne 2.75
(si l'un des termes de la division est un réel, la division retournera un réel).
 - puissance : utilisation de la méthode pow de la classe Math.
 - double y = Math.pow(x, a) équivalent à x^a , x et a étant de type double

Les types de bases (7) : les booléens

- Les booléens
 - boolean
contient soit vrai (**true**) soit faux (**false**)
- Les opérateurs logiques de comparaisons
 - Egalité : opérateur **==**
 - Différence : opérateur **!=**
 - supérieur et inférieur strictement à :
opérateurs **>** et **<**
 - supérieur et inférieur ou égal :
opérateurs **>=** et **<=**

Les types de bases (8) : les booléens

- Notation

`boolean x;`

`x= true;`

`x= false;`

`x= (5==5);` // l'expression (5==5) est évaluée et la valeur est affectée à x qui vaut alors vrai

`x= (5!=4);` // x vaut vrai, ici on obtient vrai si 5 est différent de 4

`x= (5>5);` // x vaut faux, 5 n'est pas supérieur strictement à 5

`x= (5<=5);` // x vaut vrai, 5 est bien inférieur ou égal à 5

Les types de bases (9) : les booléens

- Les autres opérateurs logiques
 - et logique : **&&**
 - ou logique : **||**
 - non logique : **!**
 - Exemples : si a et b sont 2 variables booléennes

```
boolean a,b, c;  
a= true;  
b= false;  
c= (a && b); // c vaut false  
c= (a || b); // c vaut true  
c= !(a && b); // c vaut true  
c=!a; // c vaut false
```

Les types de bases (10) : les caractères

- Les caractères
 - **char** : contient une seule lettre
 - le type char désigne des caractères en représentation Unicode
 - Codage sur 2 octets contrairement à ASCII/ANSI codé sur 1 octet. Le codage ASCII/ANSI est un sous-ensemble d'Unicode
 - Notation hexadécimale des caractères Unicode de ' \u0000 ' à ' \uFFFF '.
 - Plus d'information sur Unicode à : www.unicode.org

Les types de bases (11) : les caractères

- Notation

```
char a,b,c; // a,b et c sont des variables du type char  
a='a'; // a contient la lettre 'a'  
b= '\u0022' //b contient le caractère guillemet : "  
c=97; // x contient le caractère de rang 97 : 'a'
```


Les types de bases (12)

exemple et remarque

```
int x = 0, y = 0;  
float z = 3.1415F;  
double w = 3.1415;  
long t = 99L;  
boolean test = true;  
char c = 'a';
```

- Remarque importante :
 - Java exige que toutes les variables soient définies et initialisées. Le compilateur sait déterminer si une variable est susceptible d'être utilisée avant initialisation et produit une erreur de compilation.

Afficher du texte, saisir du texte

```
// instance de Scanner  
Scanner scanner = new Scanner(System.in);  
  
// on affiche du texte  
System.out.println("Votre nom ?");  
// on attend une entrée de la part de l'utilisateur  
// et on la stock dans une variable  
String nom = scanner.nextLine();  
// on affiche son nom  
System.out.println(nom);
```

Les conditions

Opérateurs

- Opérations sur des valeurs numériques (**int**, **float**, **double**...)
 - +, *, -, /, % (modulo)
 - **Attention** : / sur des int revient à faire une division euclidienne
 - Cast (conversion) : $q = (\text{double}) \ x / y;$
 - Affectation : =, +=, -=, *=, /=, ++ (pré, post), -- (pré, post)
 - Comparaison : <, >, <=, >=, ==, !=
- Opérations sur des booléens (**boolean**)
 - ||, &&, !
 - Attention : un boolean ne peut valoir que true ou false en Java.
- Opérateur conditionnel ... ? ... : ...
 - `boolean estPair = (n % 2 == 0 ? true : false);`
- Sur les Strings : + concatène deux chaînes.

Test if [else if] [else]

- Les blocs "else if" et "else" sont optionnels

```
if (t <= 0) {  
    System.out.println("Glace");  
} else if (t < 100) {  
    System.out.println("Eau");  
} else { // sous-entendu t >= 100  
    System.out.println("Vapeur");  
}
```

- **Attention :** on ne peut pas écrire $0 < t < 100$ qui est évalué ainsi :
 - $(0 < t) < 100$
 - $true < 100$

Exercice

- Demander à l'utilisateur deux entiers
- Afficher leur produit et informer s'il est négatif ou positif.

Test switch case

- Tests sur un ensemble de valeurs discrètes.
 - **Attention** : uniquement sur un type de base (String n'en est pas un)
 - On reste dans le switch jusqu'à rencontrer un **break**.

```
int x = 6;
switch (x) {
    case 1:
        System.out.println("un");
        break;

    case 2:
        System.out.println("deux");
        break;

    default:
        System.out.println("autre");
}
```

```
// Caractère : simple quote
char c = 'x';
switch (c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y':
        // String : double quote
        System.out.println("Voyelle");
        break;

    default: // chiffres, consonnes
        System.out.println("Autre");
}
```

TP : switch

- Ecrire un programme qui demande un choix à l'utilisateur sous forme d'une lettre. Ensuite, il l'informe d'un message.

1) (O)uvrir

2) (Q)uitter

3) (S)auver

Boucle while

- Permet d'itérer une section de code
 - **Attention aux boucles infinies**
 - On reste dans le while tant que le test est vrai.

```
int i = 0;
while (i < 10) { // Écrit i = 0 ... i = 9
    System.out.println("i = " + i);
    i++;
}
```

- On peut interrompre le while avec le mot clé **break**.
- Le mot clé **continue** permet d'ignorer la fin de l'itération courante.

```
int i = 0;
while (i < 10) { // Boucle infinie :(
    if (i == 2) {
        continue;
    }
    i++; // Plus évalué quand x atteint 2
}
```

Boucle do while

- Même principe que le while sauf que l'on rentre au moins la première fois dans le bloc.

```
int i;
Scanner sc = new Scanner(System.in);

// Demander la valeur de i jusqu'à ce qu'elle soit
// comprise entre 0 et 10
do {
    System.out.println("i (entre 0 et 10) = ? ");
    i = sc.nextInt();
} while (i > 10 || i < 0);

System.out.println("i = " + i);
```

```
i (entre 0 et 10) = ?
-2
i (entre 0 et 10) = ?
5
i = 5
```

Boucle for

- for(initialisation; test; post)
- On reste dans le for tant que le test est vrai.

```
for (int i = 0; i < 10; ++i) {  
    System.out.println("i = " + i);  
}
```

```
// Écrit 0, 1, 3, 4, 5, 6, 7  
for (int i = 0; i < 10; ++i) {  
    if (i == 2) {  
        continue;  
    }  
  
    System.out.println(i);  
    if (i > 6) {  
        break;  
    }  
}
```

Exercice

- Demander un entier positif n.
- Proposer à l'utilisateur de dessiner :
 - - une ligne d'étoile
 - - un carré d'étoiles
 - - ou un triangle d'étoiles.
- Dessiner ce que l'utilisateur a décidé.

- Exemple n = 5 :

```
* * * * *
```

```
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

```
*  
* *  
* * *  
* * * *  
* * * * *
```

Exercice

- Demander deux entiers (hauteur largeur) positifs
- Dessiner un rectangle.

Foreach

- Le « for each » permet principalement de parcourir des tableaux ou des collections pour en manipuler tous les éléments.

```
int [] intArray = { 10, 20, 30, 40, 50 };  
  
for( int value : intArray ) {  
    System.out.println( value );  
}
```

```
List<Integer> collection = new ArrayList<>();  
collection.add( 10 );  
collection.add( 20 );  
collection.add( 30 );  
collection.add( 40 );  
collection.add( 50 );  
  
for( int value : collection ) {  
    System.out.println( value );  
}
```

While / For / Foreach

- Supposons un tableau `tabPersonnes[]`

// SOLUTION 1 : boucle while

```
int y = 0;
while(y < tabPersonnes.length){
    String personne = tabPersonnes[y];
    y++;
}
```

// SOLUTION 2 : boucle for

```
for (int i = 0; i < tabPersonnes.length; i++){
    String personne = tabPersonnes[i];
}
```

// SOLUTION 3 : boucle foreach

```
for (String personne : tabPersonnes) {
}
```

Les collections

Definition

- Une **collection** regroupe plusieurs données de même nature
 - Exemples : promotion d'étudiants, sac de billes, ...
- Une **structure collective** implante une collection
 - plusieurs implantations possibles
 - ordonnées ou non, avec ou sans doublons, ...
 - accès, recherche, tris (algorithmes) plus ou moins efficaces
- Objectifs
 - adapter la structure collective aux besoins de la collection
 - ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, ...)

Structures collectives classiques

- Tableau

`type[]` et `Array`

- accès par index
- recherche efficace si le tableau est trié (dichotomie)
- insertions et suppressions peu efficaces
- défaut majeur : nombre d'éléments borné

- Liste

`interface List`

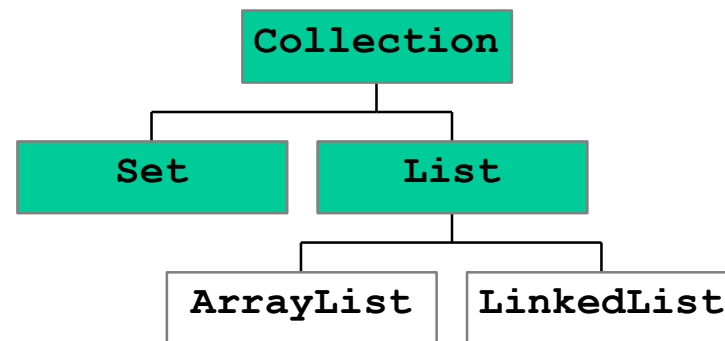
- accès séquentiel : premier, suivant
- insertions et suppressions efficaces
- recherche lente, non efficace

`class ArrayList`

- Tableau dynamique = tableau + liste

Paquetage `java.util`

- Interface `Collection`
- Interfaces `Set` et `List`
- Méthodes
 - `boolean add(Object o)`
 - `boolean remove(Object o)`
 - ...
- Plusieurs implantations
 - tableau : `ArrayList`
 - liste chaînée : `LinkedList`
- Algorithmes génériques : tri, maximum, copie ...
 - ♦ méthodes statiques de `Collection`



Méthodes communes

- `boolean add(Object)` : ajouter un élément
- `boolean addAll(Collection)` : ajouter plusieurs éléments
- `void clear()` : tout supprimer
- `boolean contains(Object)` : test d'appartenance
- `boolean containsAll(Collection)` : appartenance collective
- `boolean isEmpty()` : test de l'absence d'éléments
- `Iterator iterator()` : pour le parcours (cf Iterator)
- `boolean remove(Object)` : retrait d'un élément
- `boolean removeAll(Collection)` : retrait de plusieurs éléments
- `boolean retainAll(Collection)` : intersection
- `int size()` : nombre d'éléments
- `Object[] toArray()` : transformation en tableau
- `Object[] toArray(Object[] a)` : tableau de même type que a

Exemple : ajout d'éléments

```
import java.util.*;

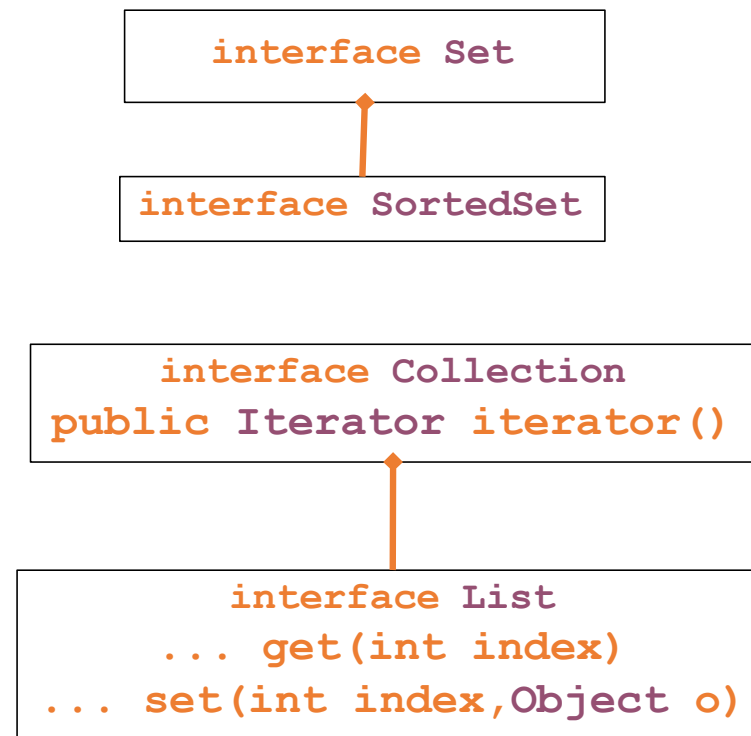
public class MaCollection {
    static final int N = 25000;
    List listEntier = new ArrayList();

    public static void main(String args[]) {
        MaCollection c = new MaCollection();
        int i;

        for (i = 0; i < N; i++) {
            c.listEntier.add(new Integer(i));
        }
    }
}
```

Caractéristiques des collections

- Ordonnées ou non
 - Ordre sur les éléments ? voir tri
- Doublons autorisés ou non
 - *liste* (**List**) : avec doubles
 - *ensemble* (**Set**) : sans doubles
- Besoins d'accès
 - indexé
 - séquentiel, via **Iterator**



Fonctionnalités des Listes

- Implantent l'interface **List**
 - **ArrayList**
 - Liste implantée dans un tableau
 - accès immédiat à chaque élément
 - ajout et suppression lourdes
 - **LinkedList**
 - accès aux éléments lourd
 - ajout et suppression très efficaces
 - permettent d'implanter les structures FIFO (file) et LIFO (pile)
 - méthodes supplémentaires : **addFirst()**, **addLast()**, **getFirst()**, **getLast()**, **removeFirst()**, **removeLast()**

Recherche d'un élément

- Méthode
 - `public boolean contains (Object o)`
 - interface `Collection`, redéfinie selon les sous-classes
- Utilise l'égalité entre objets
 - égalité définie par `boolean equals (Object o)`
 - par défaut (classe `Object`) : égalité de références
 - à redéfinir dans chaque classe d'éléments
- Cas spéciaux
 - doublons : recherche du premier ou de toutes les occurrences ?
 - structures ordonnées : plus efficace, si les éléments sont comparables (voir tri)

Exercice

- Écrivez un programme Java pour créer un ArrayList nommé «languages», ajoutez-y des String (Ex: PHP, Java, C++, Python) et affichez la collection.

Exercice

- Créer une liste nommée «lotoNumbers», ajoutez-y des entiers.
Parcourez-la en utilisant :
 - 1) Une boucle while
 - 2) Une boucle for
 - 3) Un foreach

TP

- Créer une interface qui demande à l'utilisateur si il souhaite (grâce à une lettre) :
 - A) ajouter un étudiant
 - B) voir la liste des étudiants
 - C) sortir du programme
- Coder chaque fonctionnalités

L'approche Objet

Intérêt de la POO

- **Le code est plus sûr**
- **Les programmes sont plus clairs**
- **La maintenance des applications est facilitée**
- **Le code est facilement réutilisable**
- **Il est facile de créer de nouveaux algorithmes légèrement différents par clonage d'un algorithme existant**
- **Il est facile de faire évoluer des programmes**

Les caractéristiques de l'Objet

La 1ère étape consiste à déterminer

- les entités que l'on souhaite manipuler
- la description générique qui relie toutes ses entités.

Exemple

- Prenons ces informations :
 - 14 Juillet 1789 (Prise de la Bastille)
 - 11 Novembre 1918 (Armistice)
 - 07 Mars 2005
- Ce sont des dates.
- Chaque date se caractérise par :
 - un *jour*
 - un *mois*
 - une *année*.

Les Dates

Date

- Jour
- Mois
- Année

Prise de la Bastille

- 14
- Juillet
- 1789

Une Date

- 7
- Mars
- 2005

Armistice

- 11
- Novembre
- 1918

Classes

- Le *Jour*, le *Mois* et l'*Année* sont les **attributs** d'une Date.
- Cet ensemble d'attributs est appelé une **Classe**.

Objets

- Le 14 Juillet 1789 et le 11 Novembre 1918 sont chacune des **instances** de la classe Date.
- Chacune de ces dates est appelée un **Objet**.

Autre Exemple : Les planètes

- Saturne : planète gazeuse. Son diamètre est de 120.536 km. Elle est à une distance moyenne de 1.426.725.400 km du Soleil.
- Mars : planète rocheuse. Son diamètre est de 6794 km. Elle est à une distance moyenne de 227.936.640 km du Soleil.
- Jupiter : planète gazeuse. Son diamètre est de 142.984 km. Elle est à une distance moyenne de 779 millions de km du Soleil.
- Terre : planète rocheuse. Son diamètre est de 12.756,28 km. Elle est à une distance moyenne de 150.000.000 km du Soleil.

Planète (suite)

Ces planètes ont en commun :

- Le Type : Rocheuse, Gazeuse
- La Distance au Soleil : 227936640, 779 millions, 1426725400, 1500000000
- Le Diamètre : 12756.28, 120536, 142984, 6794

La classe Planete

Planete	Terre	Saturne
<ul style="list-style-type: none">• <i>Type</i>• <i>DistanceAuSoleil</i>• <i>Diametre</i>	<ul style="list-style-type: none">• Rocheuse• 150000000• 12756,28	<ul style="list-style-type: none">• Gazeuse• 1426725400• 120536
	Jupiter	Mars
	<ul style="list-style-type: none">• Gazeuse• 779 millions• 142984	<ul style="list-style-type: none">• Rocheuse• 227936640• 6794

Qu'est-ce qu'un objet ?

- Toute instruction et toutes données sont stockées dans des objets.
- Un objet regroupe un ensemble de caractéristiques et de méthodes (on parle aussi de fonctionnalités) .
- Exemple
 - Un couteau est tranchant => caractéristique
 - Un couteau peut couper=> méthode

Les objets sont des entités

- Les objets ont une vie propre dans l'application.
 - La modification de l'un n'aura pas d'effet sur les autres objets
 - Dans notre application, deux objets qui auraient les mêmes valeurs pour leurs attributs seraient indistinguables l'un de l'autre.
-
- 2 pièces de 1 euro.
 - 2 timbres identiques de la même série.
 - 2 points de mêmes couleurs aux mêmes coordonnées.
 - Des jumeaux, parfois

Les entités ont une identité

- MAIS ils sont différents: leurs identité est différente
- Dans un programme, ces objets existent à des adresses différentes de la mémoire.
- Souvent dans des bases de données on leur associe un champ unique (clé).

Qu'est-ce qu'un objet ?

- Un objet est avant tout créé à partir d'un modèle(une modélisation)
 - **Modélise une chose tangible**
 - ex: ville, hôpital, scanner, véhicule, étudiant
 - **Modélise une chose conceptuelle**
 - ex: réunion, planning de réservation scanner, date, service, élément constitutif d'une application logicielle (collections..)

Modèles

- Pour modéliser et comprendre quelque chose de complexe
 - Une voiture
 - Un immeuble
 - Un logiciel
- Les modèles peuvent être spécifique à chaque destinataire n'exprimant que ce qu'il lui est nécessaire. (exemple une modélisation d'un avion pour un constructeur aura plus de détails, voir différent que pour un acheteur)

Exemple de l'immeuble

- Quel immeuble? Pour quel usage?
- De quoi aura-t-il l'air?
- Combien coûtera t-il? Combien de temps pour le construire?
- Par où passent les câbles réseau?
- Où se situera t-il?
- Combien d'étages?

Qu'est-ce qu'un modèle

- Un objet M est un modèle d'un objet O si M permet de répondre à des questions au sujet de O.
- Nous construisons M parce qu'il est plus pratique que O.
 - O n'est pas encore construit
 - Des choix restreints à faire (impossible de construire O maintenant)
 - M est moins cher à construire (simulation)

Un modèle est abstrait

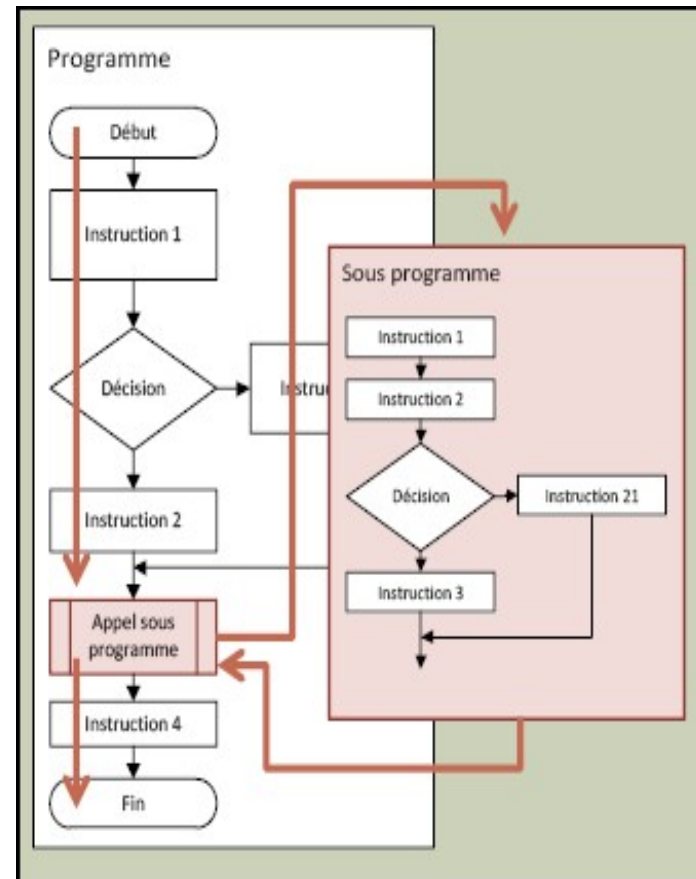
- Le modèle ne décrit pas tous les aspects de l'objet modélisé.(on ne peux pas décrire totalement un être humain... ou se sera la fin du monde...)
- M est plus simple que O.
- Parfois M est une version « idéalisée » de O.

Approche objet

- Programmation dirigée par les données et non plus par les traitements: on se concentre en premier sur la description des données et non plus sur la façon dont on les manipule
- Permet de mieux séparer les données des fonctions qui les manipulent

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

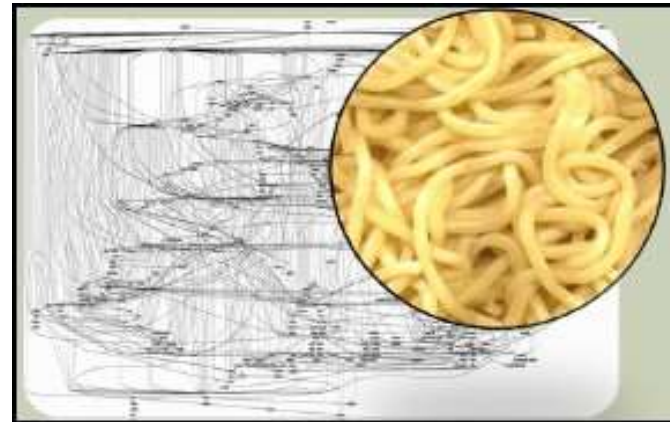


Exemple d'algorithme

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**



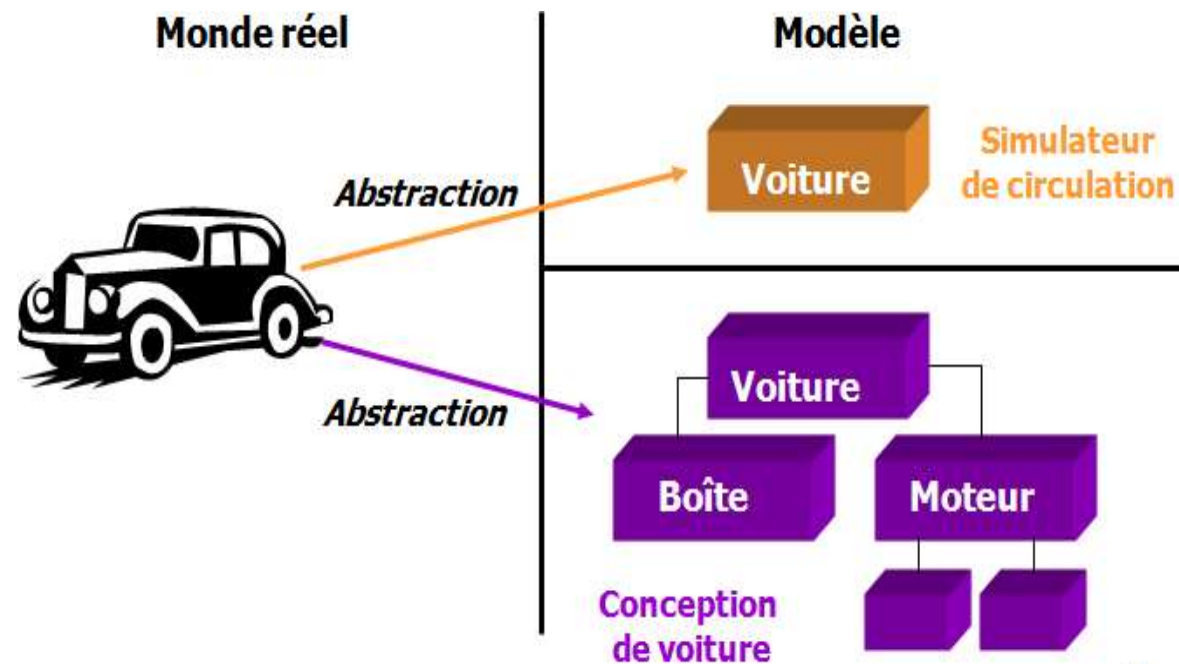
Introduction

- 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 !!?

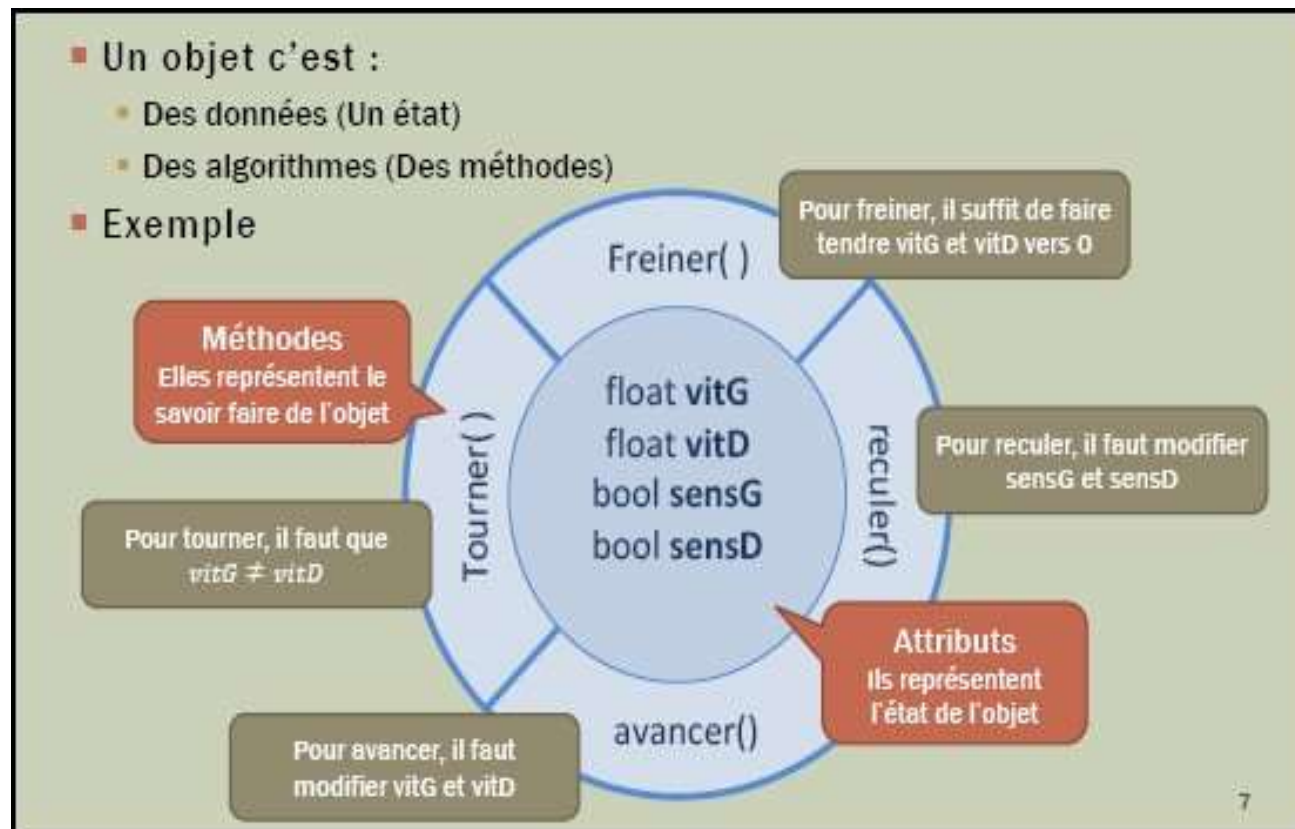
Introduction

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

A QUOI RESSEMBLE UN OBJET ?

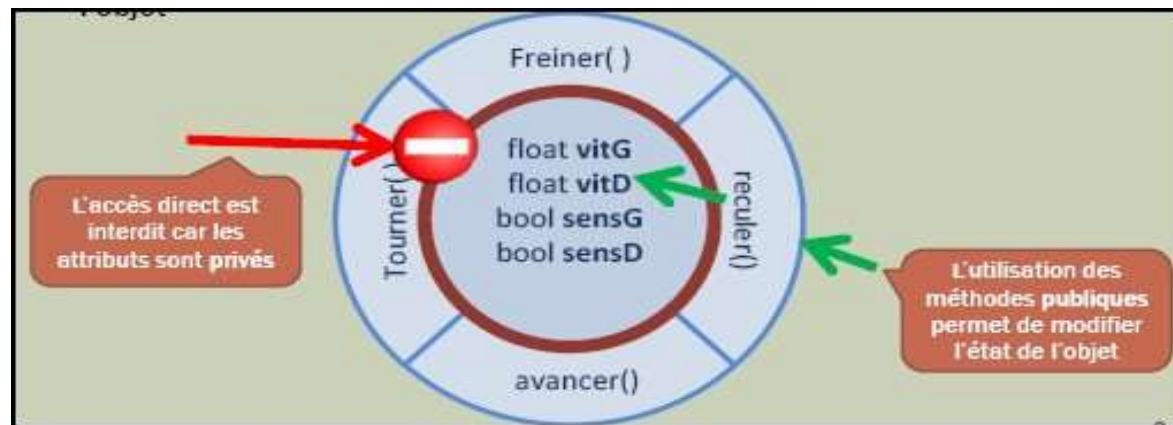


Introduction

LES RÈGLES D' ENCAPSULATION

Principe

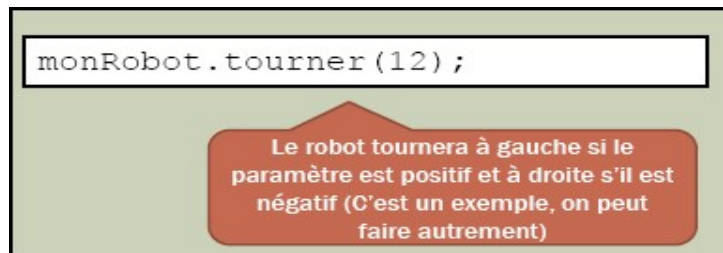
- Les attributs de l' objet sont **privés**
 - Inaccessibles directement depuis l' extérieur
- Les méthodes sont **publiques** et forme une interface de manipulation de l' objet



Introduction

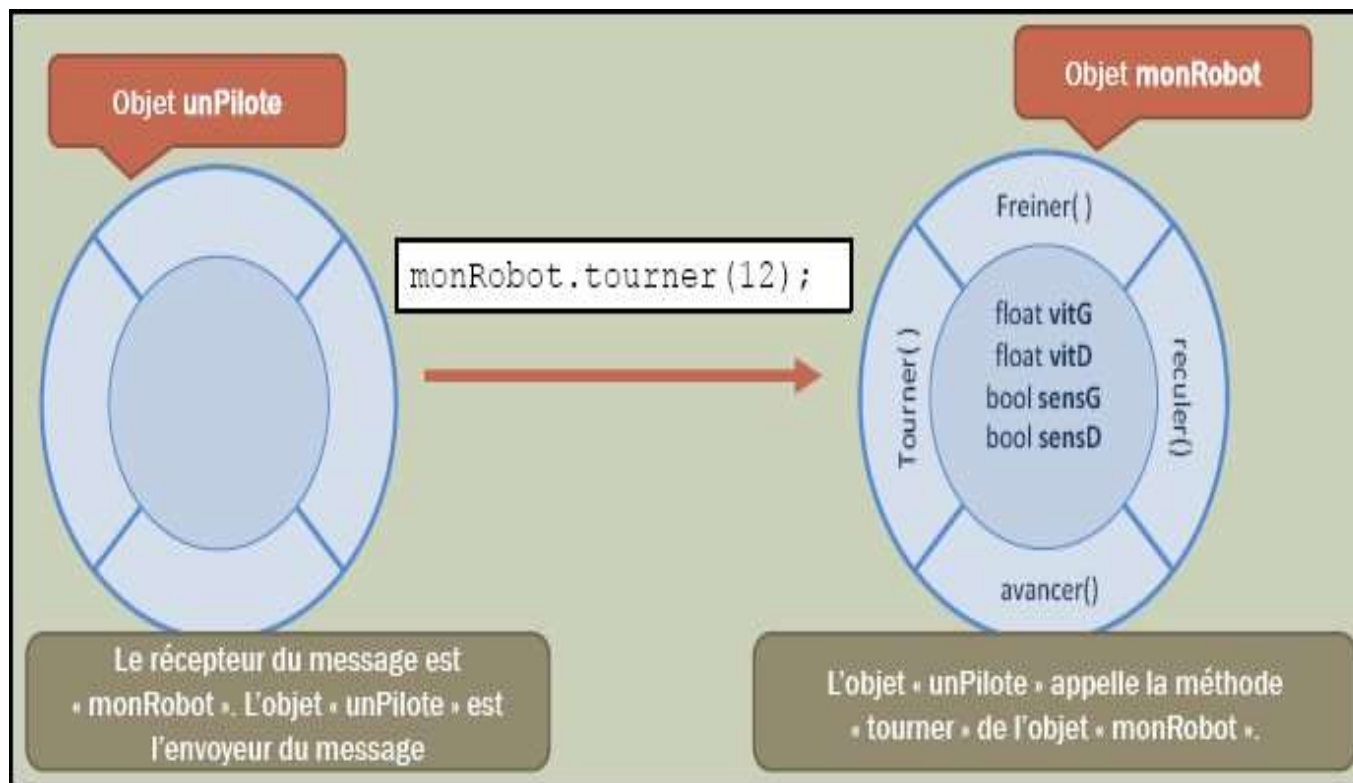
LES OBJETS COMMUNIQUENT

- **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

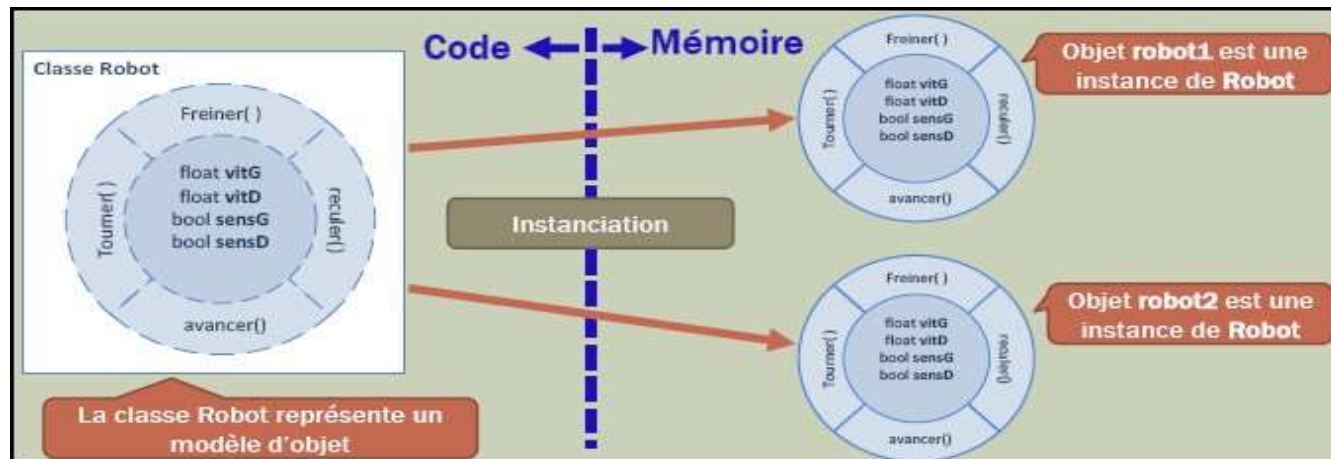
L' appel de méthode est effectué par un autre objet



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

Exemple:

```
public class Robot {  
  
    private float vitG;  
    private float vitD;  
    private int sensG;  
    private int sensD;  
  
    public void tourner(int dir) {  
        // Corps de la méthode  
    }  
  
    public float getVitesseG() {  
        return vitG;  
    }  
}
```

Le nom de la classe

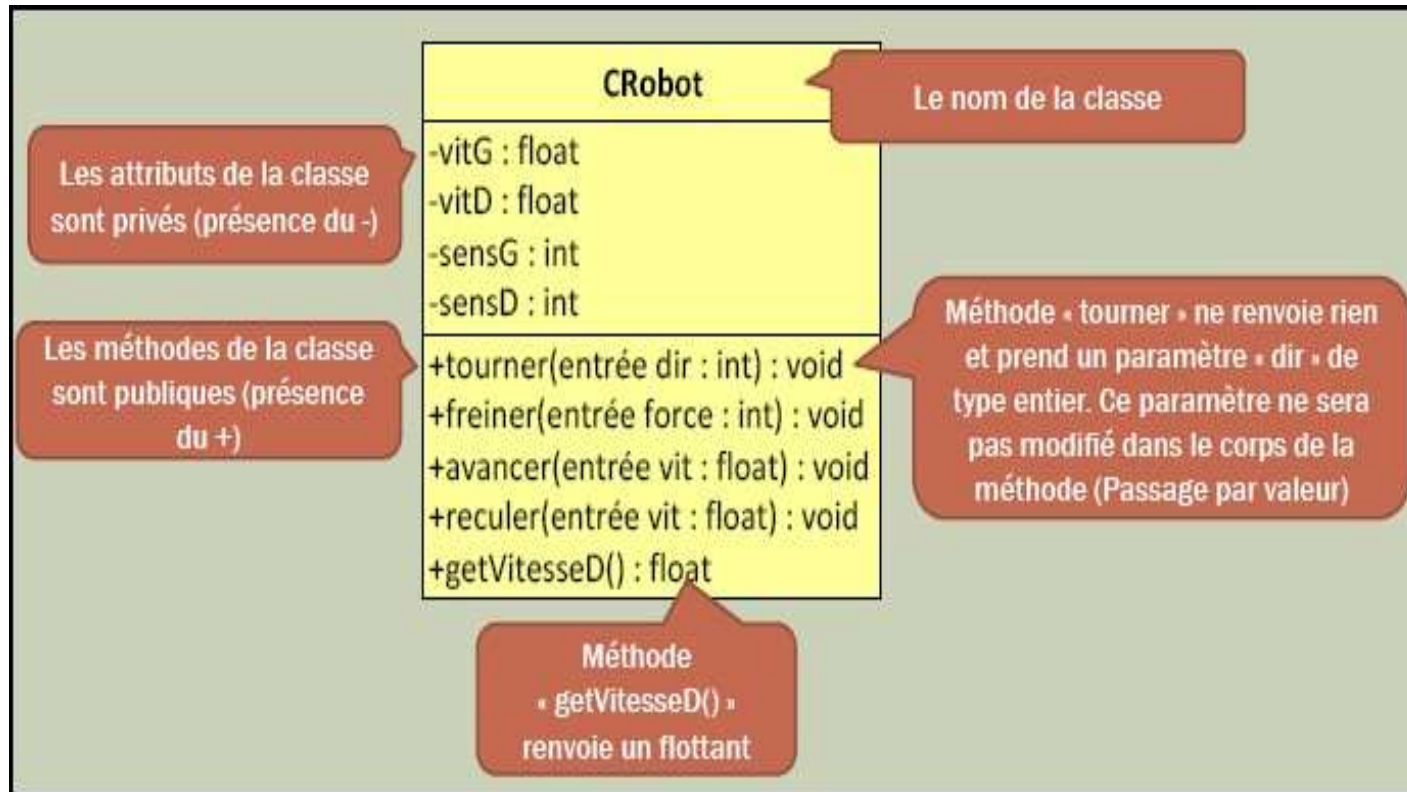
La déclaration des attributs de la classe

Prototype et corps de la méthode

Prototype et corps de la méthode **accesseur de données**

Introduction

REPRÉSENTATION UML D'UNE CLASSE:



Les objets (1/5)

- La programmation orientée objet consiste en la définition et l'assemblage de briques logicielles appelées objets
- Un objet est une structure de données qui répond à un ensemble de messages. Cette structure définit son état et l'ensemble des messages décrit son comportement

Les objets (2/5)

- Terminologie des objets
 - Les données internes sont appelées **attributs**
 - Les messages est l'appel de **méthode** dans l'objet
- Principe d'encapsulation
 - L'objet est une « boîte noire »
 - Droits d'accès aux attributs et aux méthodes

Les objets (3/5)

- Encapsulation (suite)
 - Visibilité des attributs et méthodes
 - Partie privée: connue uniquement de l'objet
 - Partie publique
 - Les données membres (attributs) doivent toujours être privées
 - Les méthodes doivent être publiques seulement si elles sont utilisées par d'autres objets.

Les objets (4/5)

- Une **classe** est un modèle utilisé pour créer les objets
- Chaque objet est une **instance** d'une classe
 - Avec son propre état (même si deux instances sont strictement identiques)
 - Toutes les instances d'une classe ont le même comportement
 - Les instances peuvent partager des attributs

Les objets (5/5)

- Exemple d'objet: un compte bancaire
 - Les données (attributs):
 - Type de compte
 - Solde
 - Les traitements (méthodes):
 - Débiter
 - Créditer

Visibilité

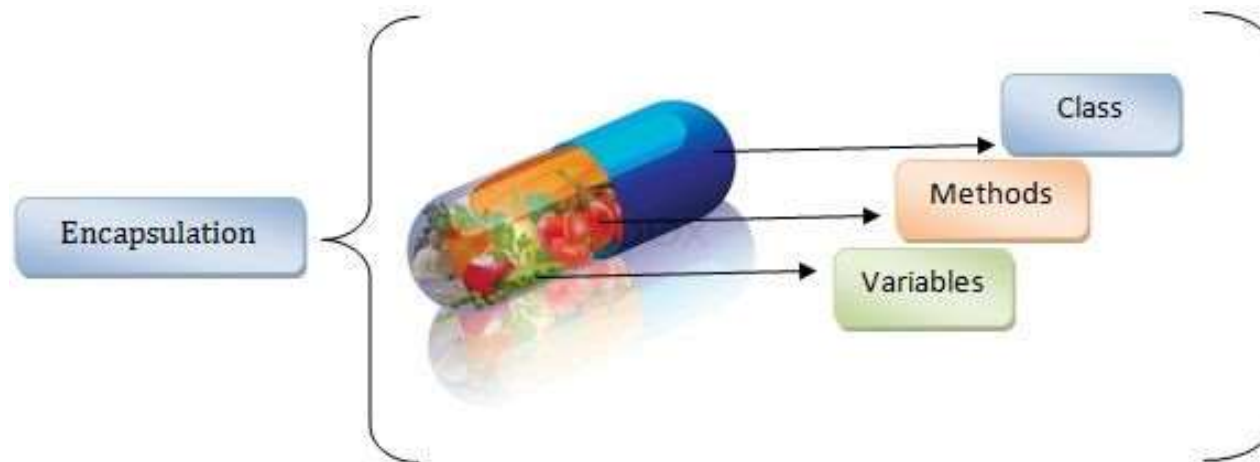
- Objectif : contrôler l'accès aux attributs / membres / méthodes
 - Garantit qu'une donnée ne peut pas être altérée n'importe comment
 - Accès en lecture seule, accès en lecture écriture, accès en écriture sous certaines conditions
 - Pour cela on leur associe une **visibilité**.
- En java, il y a 3 types de visibilité :
 - **public** : visible depuis n'importe où
 - **protected** : visible depuis la classe ou depuis ses classes filles et les classes du package !
 - **private** : visible uniquement dans la classe
 - Si on ne met pas de visibilité pour une classe, celle-ci n'est visible que dans le package. La seule visibilité possible d'une classe est **public**.
- Une visibilité s'applique à un constructeur, à un membre, à une méthode

Visibilité : règles de design

- En général les membres sont privés ou protégés. On y accède via les accesseurs qui eux sont publics.
 - Utilisation d'un accesseur **public** (getter, setter)
 - Hormis dans les accesseurs, on n'accède jamais directement à l'attribut
 - Avantage : la manière dont sont stockées les données n'impacte que les accesseurs. On peut donc les modifier aisément.

```
public class Personne{  
    private String nom;  
  
    public void setNom(String nom) {  
        // this désigne l'instance courante  
        this.nom = nom;  
    }  
    public String getNom() {  
        return this.nom;  
    }  
}
```

Encapsulation



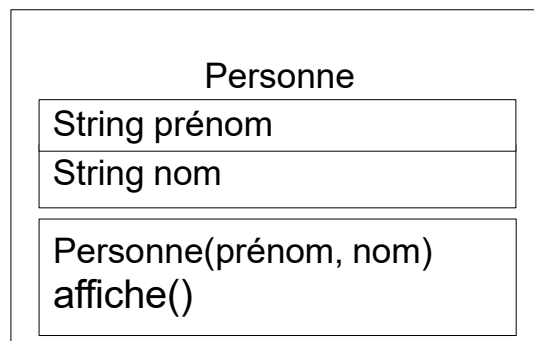
Typologie des méthodes d'une classe

- Exemple :

```
public class Personne {  
    private String prenom;  
    private String nom;  
  
    public Personne(String prenom, String nom) {  
        this.prenom = prenom;  
        this.nom = nom;  
    }  
  
    public void setPrenom(String prenom) {  
        this.prenom = prenom;  
    }  
  
    public String getPrenom() {  
        return this.prenom;  
    }  
    //...  
}
```

Constructeurs = initialisation (1/2)

- Quel que soit le langage, une classe comporte en général un ou plusieurs **constructeurs**.
 - Un constructeur alloue l'instance en mémoire et l'initialise une instance
 - Un constructeur prend ou non des paramètres
 - Pas de paramètres : "constructeur par défaut"
 - Paramètres ayant pour type la classe : "constructeur de copie"
 - ... ou d'autres paramètres.
- Le constructeur n'est pas une méthode mais il fait partie du design de l'objet. Il n'a pas de type de retour et porte le nom de la classe.



Constructeurs : en java (2/2)

- **new** invoque un constructeur.
- **super** permet d'invoquer un constructeur de la classe mère.
- **this** désigne l'instance courante (dans un constructeur, l'instance que l'on crée).

```
public class Personne {  
    private String prenom;  
    private String nom;  
  
    public Personne(String prenom, String nom) {  
        super();  
        this.prenom = prenom;  
        this.nom = nom;  
        //...  
    }  
}
```

```
Personne prof = new Personne("Julien", "Dupuis");
```

Exercice

- Définir une classe **Livre** avec les attributs suivants : *id*, *titre*, *auteur*, *prix*.
- Définir les accesseurs aux différents attributs de la classe.
- Définir un **constructeur** permettant d'initialiser les attributs d'un objet livre par des valeurs saisies par l'utilisateur. Sachant que *id* doit être auto-incrémenté.
- Définir la méthode **toString()** permettant d'afficher les informations du livre.
- Écrire un programme testant la classe Livre.

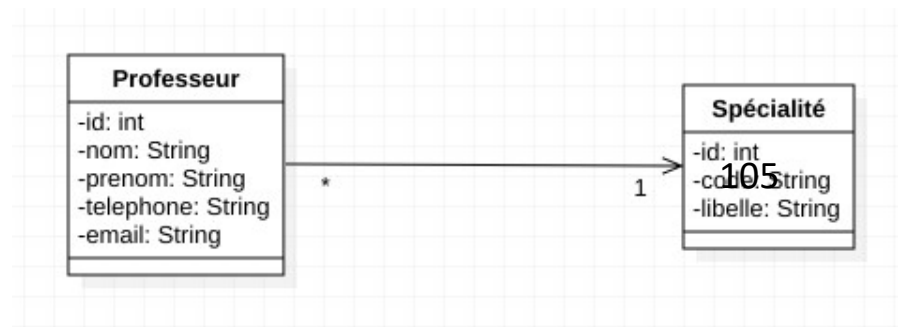
Exercice

- Définir une classe Rectangle ayant les attributs suivants : longueur et largeur.
- Ajouter un constructeur d'initialisation.
- Définir les accesseurs aux attributs de la classe.
- Ajouter les méthodes suivantes :
 - `perimetre()` : retourne le périmètre du rectangle.
 - `aire()` : retourne l'aire du rectangle.
 - `isCarre()` : vérifie si le rectangle est un carré (retourne un boolean)
 - `toString()` : expose les caractéristiques d'un rectangle comme suit :
Longueur : [...] - Largeur : [...] - Périmètre : [...] - Aire : [...] - C'est un carré / Ce n'est pas un carré
- Exemple d'exécution :

```
-Longueur : 12.6 -Largeur : 5.3 -Perimetre : 35.8 -Aire : 66.78 -Ce n'est pas un carré  
-Longueur : 3.0 -Largeur : 3.0 -Perimetre : 12.0 -Aire : 9.0 -C'est un carré  
-Longueur : 10.5 -Largeur : 5.0 -Perimetre : 31.0 -Aire : 52.5 -Ce n'est pas un carré
```


Exercice

- Développer la classe **Spécialité** et la classe **Professeur**
- Chaque classe doit comporter :
 - Un constructeur d'initialisation
 - Les accesseurs
 - La méthode toString
- **NB** : l'identifiant est auto incrémenté.
- Chaque professeur a plusieurs spécialités.
- Ajouter une methode ajouterSpecialite() qui prend en paramètre un objet de type Specialite et qui l'ajoutera à la liste des spécialités du professeur.
- Dans la classe de Main, créer **5 Spécialités** et **4 Professeurs**



Exercice

Ecrire une classe **Note** avec les attributs :

- **note**, de type double
- **commentaire**, de type String

Ecrire une classe **Eleve** avec les attributs :

- **nom**, de type String,
- **listeNotes**, qui est une ArrayList<Note>

Ajouter un constructeur permettant uniquement d'initialiser le nom de l'élève.

Ajouter les méthodes publiques :

- **getter** pour le nom de l'élève
- **getter** pour la liste des notes de l'élève
- **ajouterNote()** : qui prend en paramètre une Note et l'ajoute dans listeNotes.
 - si la note reçue en paramètre est négative, la note introduite est 0 ; si la note reçue en paramètre est supérieure à 20, la note introduite est 20.
- **calculerMoyenne()** : qui retourne la moyenne de listeNotes (double)

Classe et héritage

L'héritage (1/2)

- L'héritage est un principe de la programmation orientée objet devant favoriser la réutilisabilité et l'adaptabilité des objets
- Principe proche de l'arbre généalogique
 - Une classe « fille » est une classe qui hérite des caractéristiques de sa « mère »
- Une classe fille raffine/caractérise ce que représente la classe mère

L'héritage (2/2)

- Lors d'un héritage, une classe hérite
 - Tous les attributs et peut en ajouter
 - Toutes les méthodes et peut en ajouter ou en redéfinir
- Graphiquement, on représente l'héritage avec un triangle
- L'héritage est une relation de type « est un(e) »
 - Exemple: une voiture est un véhicule

Exercice

Ecrivez une classe `Batiment` avec l'attribut suivant:

- `adresse`

La classe `Batiment` doit disposer des constructeurs suivants:

- `Batiment()`,
- `Batiment(adresse)`.

La classe `Batiment` doit contenir des accesseurs et mutateurs pour les différents attributs.

La classe `Batiment` doit contenir une méthode `toString()` donnant une représentation du Bâtiment.

Ecrivez une classe `Maison` héritant de `Batiment` avec l'attribut suivant:

- `nbPieces`: Le nombre de pièces de la maison.

La classe `Maison` doit disposer des constructeurs suivants:

- `Maison()`,
- `Maison(adresse, nbPieces)`.

La classe `Maison` doit contenir des accesseurs et mutateurs (ou des propriétés) pour les différents attributs.

La classe `Maison` doit contenir une méthode `ToString()` donnant une représentation de la Maison.

Ecrivez aussi un programme afin de tester ces deux classes.

Polymorphisme (1/2)

- Polymorphisme: peut prendre plusieurs formes
- Polymorphisme d'héritage
 - Possibilité de redéfinir une méthode dans une classe fille (qui hérite d'une classe mère): spécialisation.
 - On peut alors appeler la méthode d'un objet sans se soucier de son type en se basant sur la classe mère (ou classe de base)

Polymorphisme (2/2)

- Exemple
 - Un jeu d'échec comporte des objets:
 - « roi », « reine », « fou », « cavalier », « tour » et « pion »
 - Chacun hérite de l'objet « pièce »
 - La méthode *déplacement()* peut avec le polymorphisme d'héritage effectuer le mouvement de la pièce sans savoir quelle pièce se cache réellement derrière le type de base

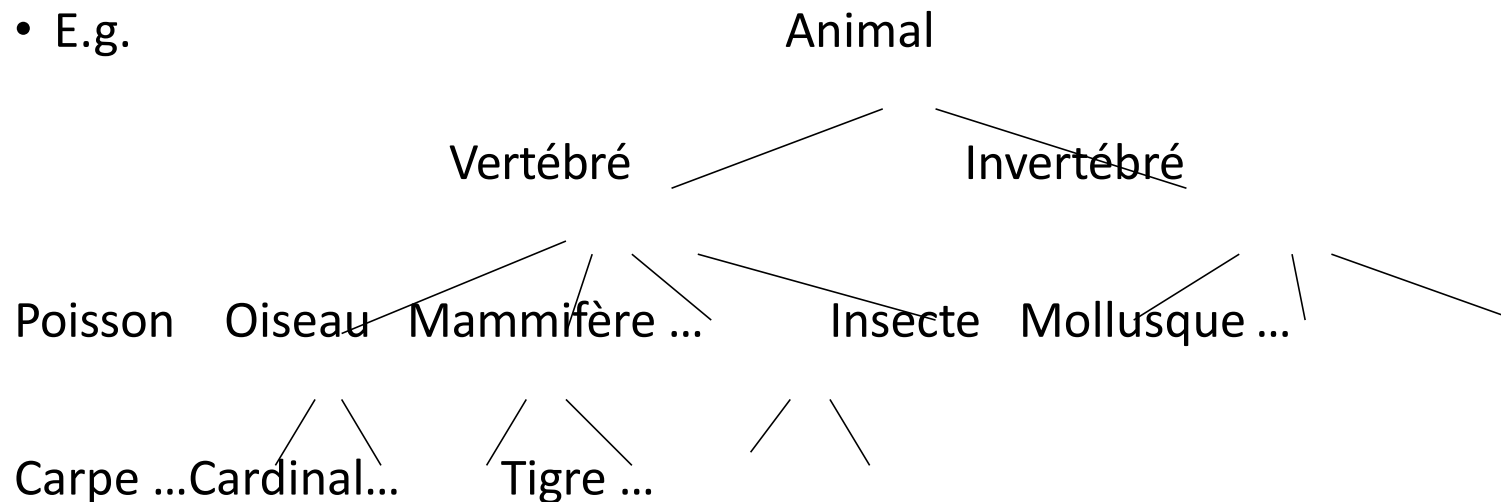
Méthode *static*

- Méthode statique: définie pour une classe
- Utilité:
 - Traitements qui ne dépendent pas de valeurs stockées dans l'objet autre que valeurs numériques (paramètres)
- Exemple:
 - Opérations mathématiques
 - `Math.sqrt(double)`, `Math.sin(double)`

Hiérarchie des classes

- Développer des classes et des sous-classes selon la hiérarchie naturelle des concepts

- E.g.



En Java

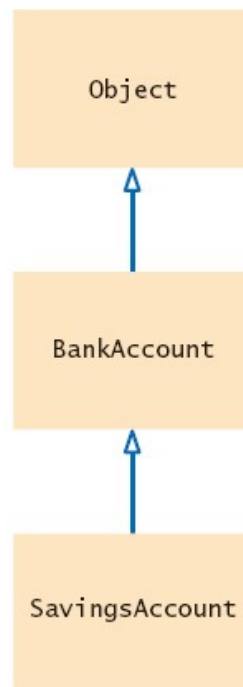


Figure 1
An Inheritance Diagram

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount (double rate)
        { interestRate = rate; }
    public void addInterest()
        { double interest = getBalance() *           //pas besoin de this –
                                                    //paramètre implicite
          interestRate / 100;
          deposit(interest); }
    private double interestRate;
}

Hérités:
private double balance // attention: pas d'accès direct //
    dans une sous-classe
public double getBalance()
public void deposit(double amount)
public void withdraw(double amount)
```

Structure créée

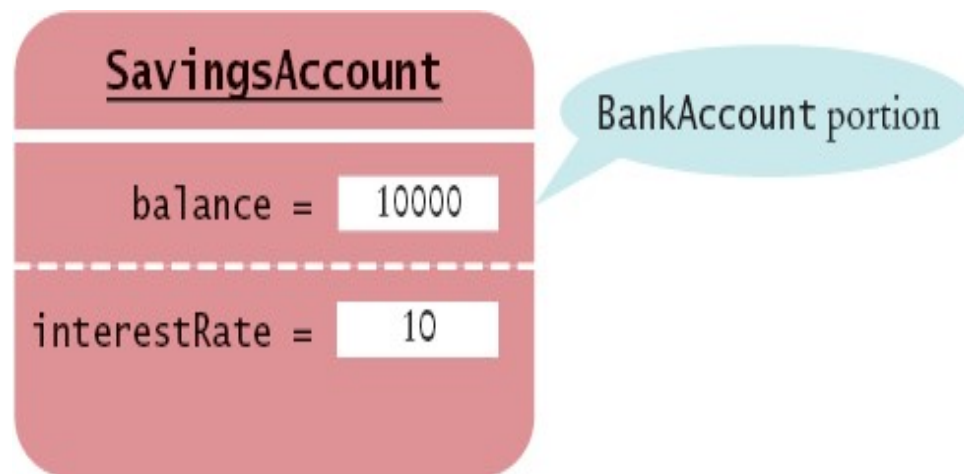
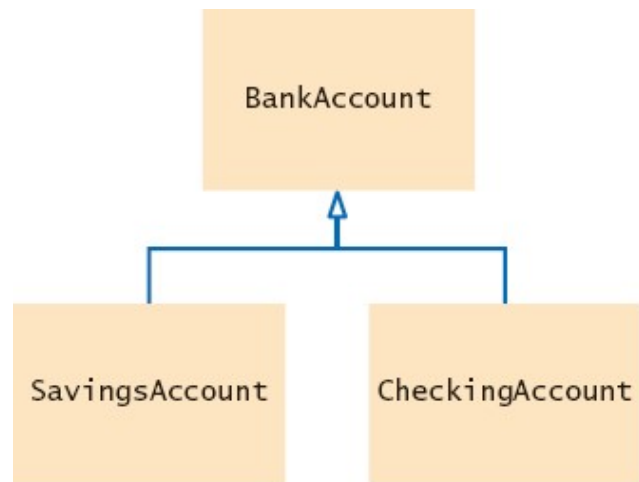


Figure 2 Layout of a Subclass Object

Créer une hiérarchie

- `class SavingsAccount extends BankAccount {...}`
- `class CheckingAccount extends BankAccount {...}`



En Java:
Héritage simple

Figure 5 Inheritance Hierarchy for Bank Account Classes

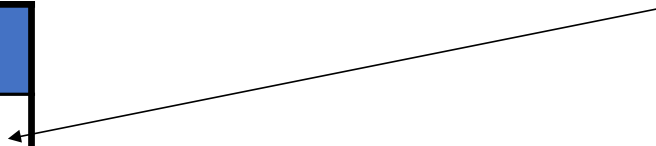
Ce qu'on a dans une sous-classe

- Attributs

- Attributs:
 - Attributs supplémentaires
 - Ne peut pas enlever les attributs hérités de la super-classe
 - Tous les attributs de la super-classe sont hérités
 - On peut définir un attribut du même nom qu'un attribut hérité (déconseillé), mais les deux co-existent
 - E.g.

```
class C1 {int a;}  
class C2 extends C1 {String a;  
                        void setA(String b) {a=b;} }
```

int	a
String	a



Ce qu'on a dans une sous-classe

- Méthodes

- Méthodes supplémentaires définies dans la sous-classe (pas de conflit de signature)
- Méthodes héritées
- Méthode réécrite (overriding) si même signature
- E.g. Overriding

```
class C1 {public void m1(int a) {System.out.println(a);}}
```

```
class C2 extends C1 {
```

```
    public void m1(String s) {System.out.println(a);}} // pas de overring
```

```
class C3 extends C1 {
```

```
    public void m1(int b) {System.out.println(a+1);}} // overriding
```

Exemple: CheckingAccount

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . } // new method
    private int transactionCount; // new instance field
}
```

- **Hérités:**

- Attribut: balance (hérité de BankAccount)
- Méthodes: getBalance(),

~~deposit(double amount), withdraw(double amount)~~

- **Ajoutés**

- Attribut: transactionCount
- Méthode: deductFees()

- **Overrid**

- Méthodes: deposit(double amount), withdraw(double amount)

```
public class BankAccount
{
    public BankAccount() {...}
    public BankAccount(double
        initialBalance) {...}
    public void deposit(double amount) {...}
    public void withdraw(double
        amount) {...}
    public double getBalance(){...}

    private double balance;
}
```


BankAccount

```
public class BankAccount
{
    public BankAccount()
    {
        balance = 0;
    }
    public BankAccount
        (double initialBalance)
    {
        balance = initialBalance;
    }
    public void deposit(double amount)
    {
        double newBalance =
            balance + amount;
        balance = newBalance;
    }
}
```

```
    public void withdraw(double amount)
    {
        double newBalance =
            balance - amount;
        balance = newBalance;
    }
    public double getBalance()
    {
        return balance;
    }

    private double balance;
}
```

Existence et Accessibilité

- class BankAccount
 - {
 - private double `balance`;
 - ...
 - }
- class CheckingAccount extends BankAccount
 - {
 - public void deposit(double amount)
 - {
 - transactionCount++;
 - // now add amount to balance
 - `balance = balance + amount;` // erreur
 - }
 - }
- `balance` existe, mais n'est pas accessible dans une sous-classe parce qu'elle est *private*
- Deux façons pour corriger la situation:
 1. Pour rendre `balance` accessible dans une sous-classe: protéger avec *protected*: ouvrir l'accès à une sous-classe
 2. Accès via la méthode `deposit` de la super-classe: `super.deposit(amount);`

Solution 1 - protected

- class BankAccount
 - {
 - protected double balance;
 - ...
 - }
- class CheckingAccount extends BankAccount
 - {
 - public void deposit(double amount)
 - {
 - transactionCount++;
 - // now add amount to balance
 - balance = balance + amount ; // OK
 - }
 - }

Solution 2 – utiliser la méthode de *super*

- class BankAccount
{
 private double **balance**;
 ...
}
- class CheckingAccount extends BankAccount
{
 public void deposit(double amount)
 {
 transactionCount++;
 // now add amount to balance
 super.deposit(amount);
 }
}

```
public class BankAccount
{
    // OK
    public BankAccount()
    {
        balance = 0;
    }
    public BankAccount
        (double initialBalance)
    {
        balance = initialBalance;
    }
    public void deposit(double amount)
    {
        double newBalance =
            balance + amount;
        balance = newBalance;
    }
    ...
}
```

Accessibilité

- Si un attribut/méthode est protégé avec
`publicprotected` `private`
- Alors il est accessible dans une sous-classe
`oui` `oui` `non`

super

- *super* évoque la classe supérieure
- Utilisation 1 (dans la définition d'une méthode de sous-classe)
 - Si une méthode (e.g. `deposit`) de la super-classe est redéfinie (`override`), dans la déclaration de la sous-classe, `deposit(100)` évoque la version de la sous-classe
 - `super.deposit(100)` évoque la version de la super-classe
 - (comparaison avec *this*)
- Utilisation 2 (dans un constructeur)
 - Utiliser `super(paramètres)` pour appeler le constructeur de la super-classe
 - Utile pour construire une version d'objet qui est ensuite enrichie ou complétée
 - Règles
 - Si un constructeur de sous-classe évoque `super(...)`, ce doit être la première instruction
 - Si un constructeur de la sous-classe n'évoque pas `super` explicitement, `super()` est implicitement évoquée comme la première instruction
 - Il faut que cette version de constructeur soit définie dans la super-classe

Exemple

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        // Construct de superclass
        super(initialBalance);
        // Initialize transaction count
        transactionCount = 0;
    }
    ...
}
```

```
public class BankAccount
{
    public BankAccount()
    {
        balance = 0;
    }
    public BankAccount
        (double initialBalance)
    {
        balance = initialBalance;
    }
    public void deposit(double amount)
    {
        double newBalance =
            balance + amount;
        balance = newBalance;
    }
    ...
}
```

Exemple

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        // Construct de superclass
        // super() implicit
        super.deposit(initialBalance);
        // Initialize transaction count
        transactionCount = 0;
    }
    ...
}
```

```
public class BankAccount
{
    public BankAccount()
    {
        balance = 0;
    }
    public BankAccount
        (double initialBalance)
    {
        balance = initialBalance;
    }
    public void deposit(double amount)
    {
        double newBalance =
            balance + amount;
        balance = newBalance;
    }
    ...
}
```


Références

- Une référence de la super-classe peut référer à un objet (instance) de
 - Super-classe
 - Sous-classe
- Une référence d'une sous-classe peut référer à un objet de
 - Sous-classe
- E.g.
 - BankAccount a1;
 - CheckingAccount a2;
 - a1 = new CheckingAccount(0); //OK
 - a2 = new BankAccount(0); //Erreur
- Comparaison
 - Appeler un étudiant en maîtrise « Eudiant »: OK
 - Appeler un étudiant « EtudiantMaitrise »: Non

Casting de référence

- *Casting* permet de changer une référence de super-classe en une référence de sous-classe (à condition que l'objet référé est réellement de la sous-classe)
- (<super>) ref
- Devient une réf. de sous-classe
- Permet de **se positionner** au niveau de la sous-classe
 - Utilité: référer aux attributs de la sous-classe

Exemple (casting pour attribut)

```
class C1
{
    public int a;
    public C1 (int x) { a=x; }
}
```

Class C2 extends C1

```
{
    public int a; // ajout un autre attribut du même nom
    public int b;
    public C2 (int x, int y) { super(0); a=x, b=y;}
}
```

Utilisations

C1 ref1;

C2 ref2;

ref1 = new C1(1); ref2 = new C2(2,3); //OK

ref1 = new C2(2,3); //OK

ref2 = new C1(1); //Erreur: réf de sous-classe → objet de super-classe

ref1 = new C2(2,3);

ref2 = ref1; // Problème de compilation: lier une réf de sous-classe à un objet super-classe

ref2 = (C2) ref1; // OK: dire au compilateur que ref1 réfère à une instance de classe C2.

System.out.println(ref1.a); // Valeur 0: attribut a de C1

System.out.println(((C2) ref1).a); // Valeur 2: attribut de C2. De même pour ref2.a

ref2 = new C2(2,3);

System.out.println(((C1) ref2).a); // valeur 0

C1: a

C2: a; a, b

Problème si super(0) n'y est pas

Référence de super-classe

- Peut référer à tous objets de la classe (super-classe) par la même référence (variable) - peut simplifier certains traitements
- E.g. parcourir tous les objets d'une classe et de ses sous-classes avec la même référence
- Supposons: `BankAccount oneAccount;`
- `oneAccount.<attribute>`:
 - selon la classe `BankAccount`
- Question: `oneAccount.<méthode>`?
 - `oneAccount.deposit(100)` = quelle version?

Règle

- La version de la méthode évoquée à partir d'une référence correspond à la version de l'instance (objet) référée
- `oneAccount.deposit(100)` = version de l'instance
- E.g.

```
BankAccount a1;
```

```
a1 = new CheckingAccount(0);
```

```
a1.deposit(100); //version de CheckingAccount
```

Condition

- Utiliser une référence de la super-classe pour évoquer une méthode:
 - Il faut que la méthode soit définie dans la super-classe
 - Sinon, une erreur de compilation: méthode non disponible
- E.g.

Object anObject = new BankAccount(); anObject.deposit(1000); // **Wrong!**
deposit(...) n'est pas une méthode disponible dans Object

Règle sur référence

- Si une méthode peut être appliquée sur toutes les instances d'une super-classe et ses sous-classe
 - Définir une version pour super-classe
 - Modifier (si nécessaire) dans les sous-classes
- Comparaison:
 - Pour tout Etudiant, on peut connaître sa moyenne, même si la façon de calculer la moyenne est différente pour des étudiants gradués et sous-gradués
 - Définir une méthode `getMoyenne()` pour tout Etudiant
 - Raffiner (redéfinir) dans les sous-classes

Polymorphisme

- La capacité qu'une même évocation de méthode change de comportement selon l'instance avec laquelle la méthode est appelée.
- E.g.

```
BankAccount a1;
```

```
for ( ...)      // une suite de BankAccount de natures différents
```

```
{
```

```
    a1.deposit(1000); // la version de l'instance
```

```
}
```


Autre exemple

```
class C1
{
    public int a;
    public C1 (int x) { a=x; }
    public void print() { System.out.println("Version 1 " + a); }
}
Class C2 extends C1
{
    public int b;
    public C2 (int x, int y) { super(x); b=y;}
    public void print() { System.out.println("Version 2 " + b); }
}
```

Utilisation

```
C1 ref1;
C2 ref2;
ref1 = new C1(1);
ref2 = new C2(2,2);

ref1.print();           // Version 1 1
ref2.print();           // Version 2 2
ref1 = ref2;
ref1.print();           // Version 2 2: version de l'objet
((C1)ref2).print();     // Version 2 2: Différence avec casting pour attribut –
                        // casting ne change pas de version de méthode
```

C1: a
print()
|
C2: a; b
print()

Connaître la classe d'une instance référée (*instanceof*)

- On peut avoir besoin de connaître la classe de l'instance pour
 - déterminer si une méthode est applicable
 - choisir un traitement ou un autre
 - ...
- Test: *object instanceof TypeName*
 - Tester si *object* est de la classe *TypeName*

```
e.g. if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    ...
}
```

Récapitulation

- Attribut
 - Détermine l'attribut accédé selon la classe de la référence
 - Possibilité de *cast* pour modifier la classe de référence temporairement
- Méthode
 - Détermine la version de la méthode selon la classe réelle de l'instance
 - Pas de *casting* possible

Hiérarchie de classe en Java

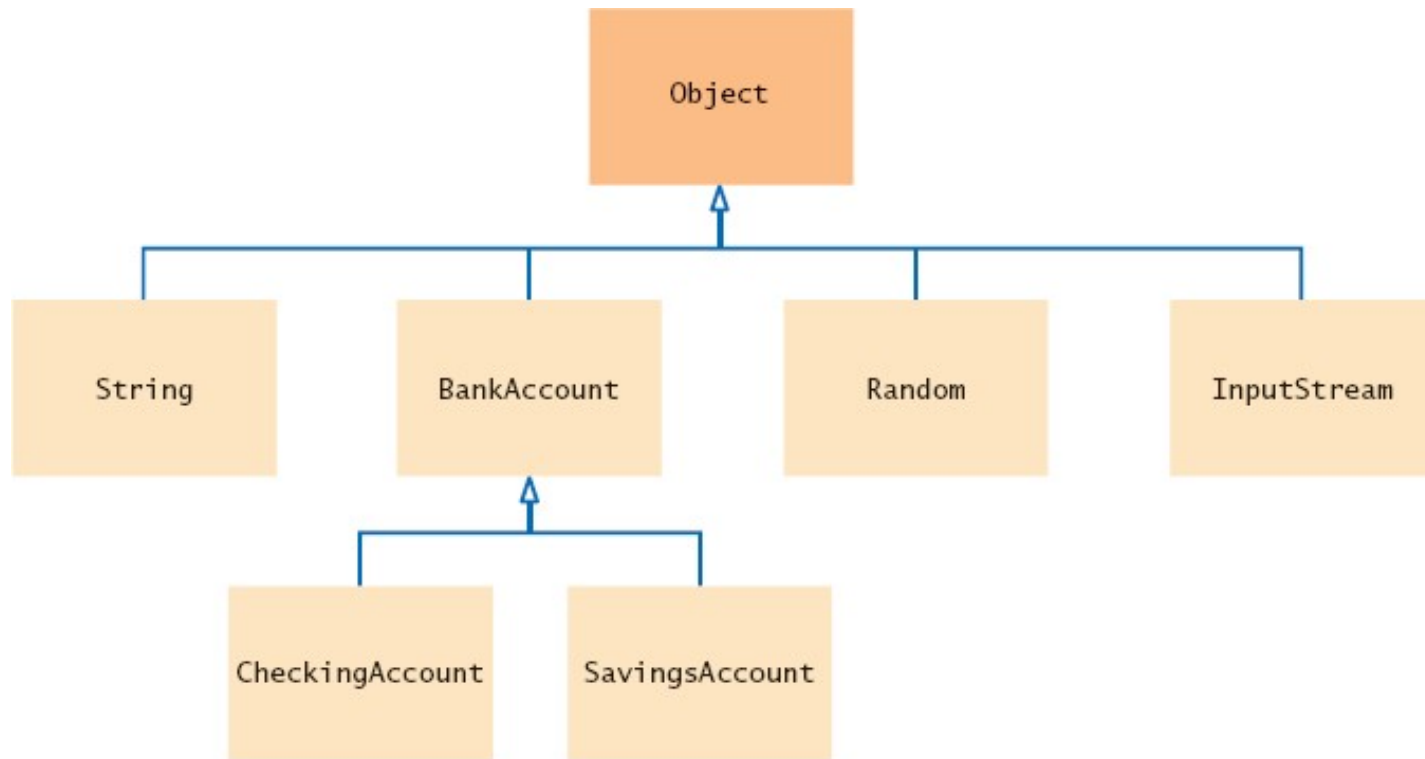


Figure 8 The Object Class Is the Superclass of Every Java Class

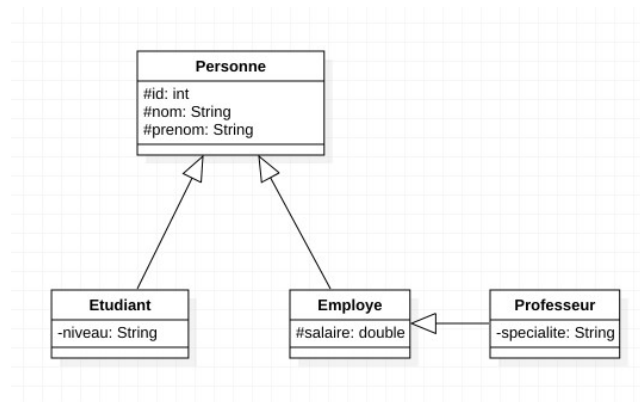
Les classes “final”

- Une classe “final” ne peut pas être étendue avec l’héritage

```
final class NotExtendable {  
    // ...  
}
```

- Une méthode “final” ne peut pas être redéfinie avec l’héritage
- C’est une bonne idée de définir toutes les classes comme “final”, sauf celles pour laquelle on veut laisser la possibilité d’extension par l’héritage
 - Question: est-ce qu’il est une bonne idée de définir une classe abstraite comme final?

Exercice



Visibilités dans un diagramme de classe :

: protected

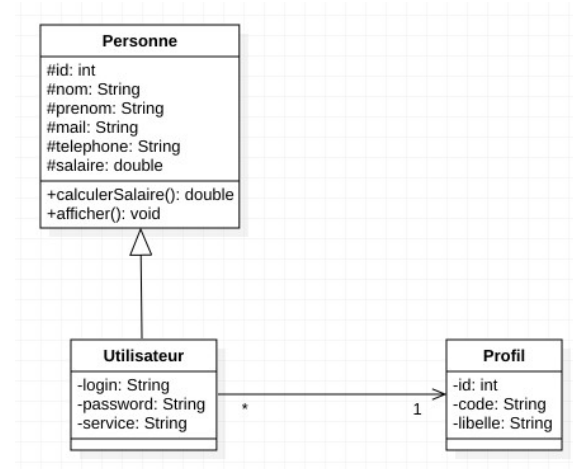
- : private

+ : public

- La classe **Etudiant** hérite de la classe **Personne**.
- La classe **Professeur** hérite de la classe **Employe** et la classe **Employe** hérite de la classe **Personne**.
- Un **Etudiant** est une **Personne**.
- Un **Professeur** est un **Employe** et un **Employe** est une **Personne**.
- Puis créer :
 - deux étudiants.
 - deux employés.
 - deux professeurs.
 - afficher les informations de chaque personne.

Exercice

- Développer les classes du diagramme.
NB : l'identifiant est auto incrément.
- Redéfinir la méthode **calculerSalaire ()** et la méthode **afficher()** dans la classe Utilisateur.
Sachant que :
 - Le manager aura une augmentation de 10% par rapport à son salaire normal,
 - Le directeur général aura une augmentation de 40% par rapport à son salaire normal.
- Créer les profils :
 - Chef de projet (CP),
 - Manager (MN),
 - Directeur de projet (DP),
 - Directeur des ressources humaines (DRH),
 - Directeur général (DG),
- Créer des utilisateurs avec les différents profils métiers.
- Afficher la liste des utilisateurs.



Classes abstraites et interfaces

Classes abstraites et classes concrètes

- Une **classe abstraite** est une classe qui ne définit pas toutes ses méthodes
 - Elle ne peut pas être instanciée
- Une **classe concrète** définit toutes ses méthodes
 - Une classe concrète peut hériter d'une classe abstraite
 - Elle peut être instanciée
- Avec les classes abstraites, on peut faire des programmes "génériques"
 - On définit les méthodes manquantes avec l'héritage, pour obtenir une classe concrète qu'on peut ensuite instancier et exécuter

Un exemple d'une classe abstraite

```
abstract class Benchmark {  
    abstract void benchmark();  
  
    public long repeat(int count) {  
        long start=System.currentTimeMillis();  
        for (int i=0; i<count; i++)  
            benchmark();  
        return (System.currentTimeMillis()-start);  
    }  
}
```

Une classe abstraite

- Voici comment on peut faire la même chose avec les valeurs procédurales:

```
fun {Repeat Count Benchmark}  
  Start={OS.time}  
in  
  for I in 1..Count do {Benchmark} end  
  {OS.time}-Start  
end
```

- La fonction Repeat joue le rôle de la méthode repeat dans la classe Benchmark
- L'argument Benchmark est une procédure qui joue le rôle de la méthode benchmark
- Conclusion: avec les classes abstraites on peut faire comme si on passait une procédure en argument
 - On utilise l'héritage pour simuler le passage d'une procédure (= valeur procédurale)

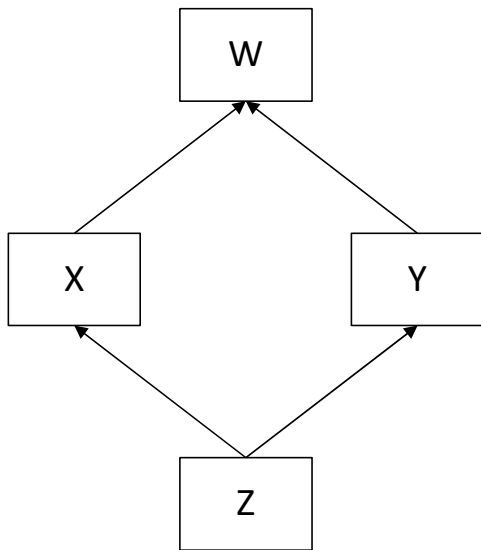
Les interfaces

- Une **interface** en Java est comme une classe abstraite sans aucune définition de méthode
 - L'interface décrit les méthodes et les types de leurs arguments, sans rien dire sur leur implémentation
- Java permet l'héritage multiple sur les interfaces
 - On regagne une partie de l'expressivité de l'héritage multiple

Un exemple d'une interface

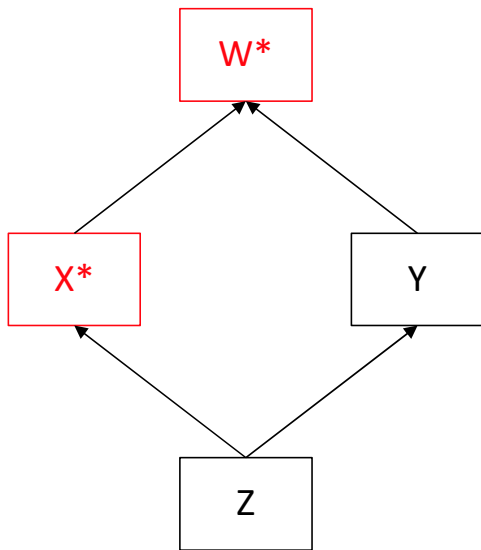
```
interface Lookup {  
    Object find(String name);  
}  
  
class SimpleLookup implements Lookup {  
    private String[] Names;  
    private Object[] Values;  
    public Object find(String name) {  
        for (int i=0; i<Names.length; i++) {  
            if (Names[i].equals(name))  
                return Values[i];  
        }  
        return null;  
    }  
}
```

Le problème avec l'héritage multiple des classes



- Voici un cas où l'héritage multiple classique a un problème: l'héritage en losange
- Quand W a de l'état (des attributs), qui va initialiser W? X ou Y ou les deux?
 - Il n'y a pas de solution simple
 - C'est une des raisons pourquoi l'héritage multiple est interdit en Java
- Nous allons voir comment les interfaces peuvent résoudre ce problème

Une solution avec les interfaces

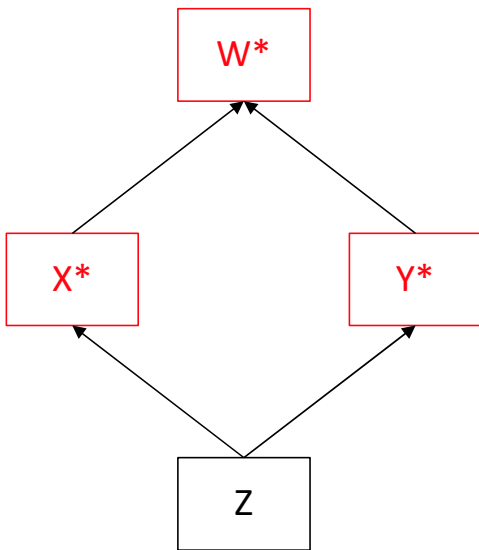


- Les interfaces sont marquées en rouge et avec un astérisque (*)
- Il n'y a plus d'héritage en losange: la classe Z hérite uniquement de la classe Y
- Pour les interfaces, l'héritage est uniquement **une contrainte sur les entêtes des méthodes** fournies par les classes

La syntaxe Java pour l'exemple du losange

```
interface W { }  
interface X extends W { }  
class Y implements W { }  
class Z extends Y implements X { }
```


Une autre solution pour la même hiérarchie



- Cette fois, Z est la seule vraie classe dans l'hiérarchie
- Voici la syntaxe:

```
interface W { }  
interface X extends W { }  
interface Y extends W { }  
class Z implements X, Y { }
```

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.

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 {
```

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**

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

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 {  
    ...  
}
```

Méthodes abstraites

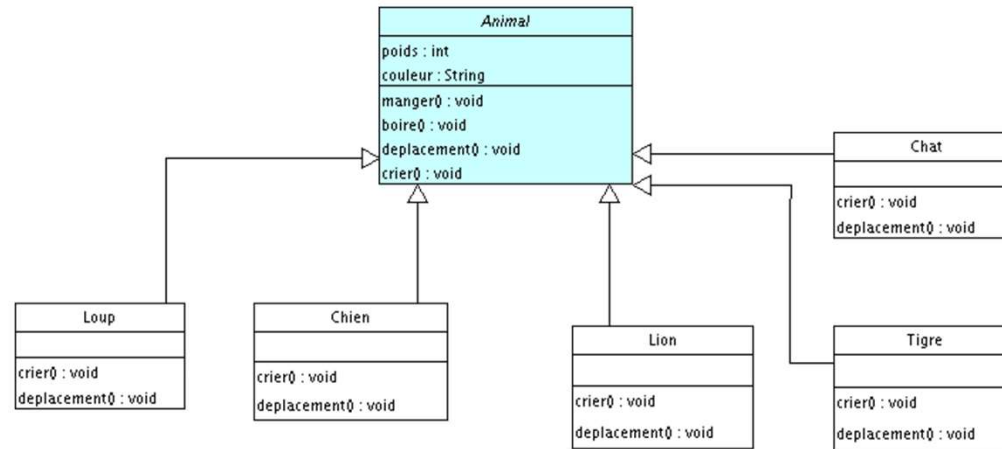
Deux points importants :

- 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**)

Exemple classe abstraite



(la classe *Animal* est abstraite car elle est indiquée en *italique*)

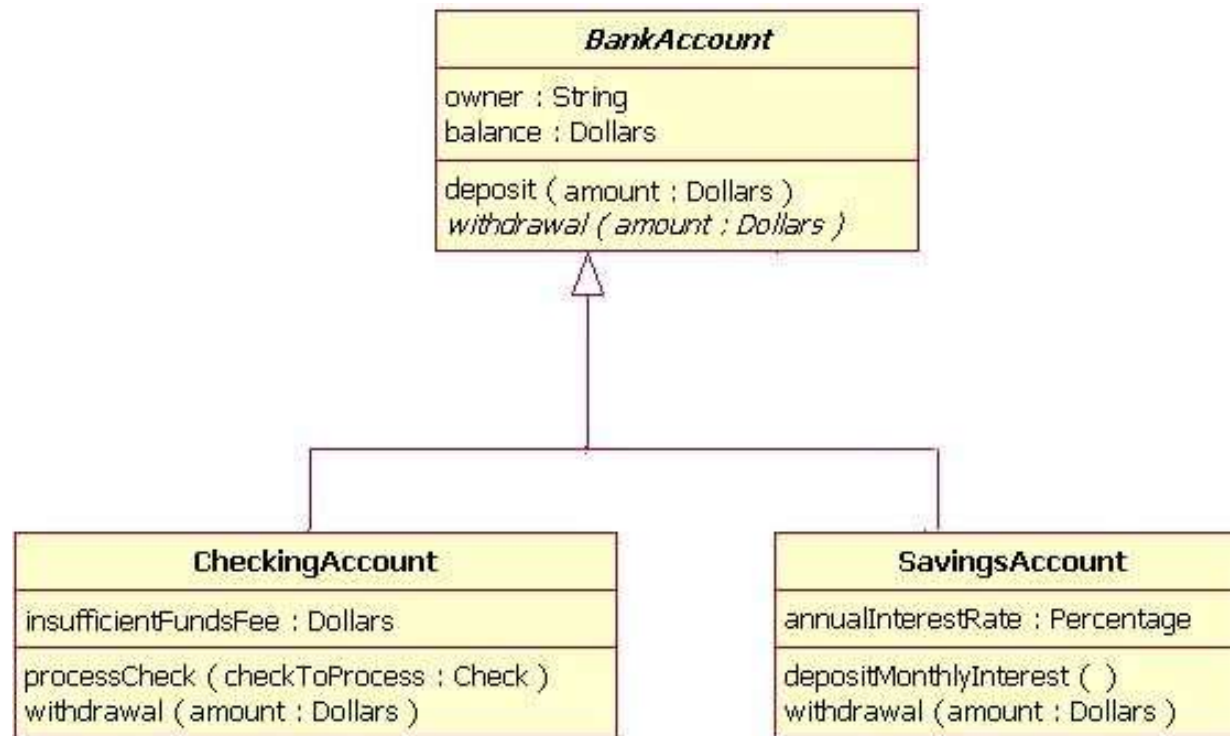
Pour synthétiser

- Une interface ne contient que des classes abstraites
- Quand une interface est implémentée dans une classe, la classe doit implémenter toutes les méthodes de l'interface.
- Une classe abstraite doit contenir au moins une méthode abstraite.
- Quand une classe hérite d'une classe abstraite, elle n'est pas obligée de tout implémenter.

Pourquoi une interface ? Exemple...

- Disons que j'ai une application qui gère des fichiers logs : Dans une version 1 de mon application, les fichiers logs ont un format texte et je crée donc une classe `LogText` qui gère ce fichier et je l'utilise partout dans le code de mon application
- Dans une version 2, les dirigeants ont décidé que le nouveau format du fichier log sera en XML et là j'ai un problème: je dois revoir tout le code de mon application là où j'ai travaillé avec `LogText` et les remplacer par la nouvelle classe `LogXML`
- Si depuis le début (version 1), j'avais créé une interface `ILogFile` qui décrit les méthodes que doit retourner un fichier log et fait implémenter cette interface par `LogText` et de même pour `LogXml`, j'aurais eu beaucoup moins de problèmes.
- A noter que dans le reste du code, il faut que j'utilise le plus souvent possible l'interface et non l'implémentation.

Abstract



Interfaces

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

```
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

Visibilité

- Objectif : contrôler l'accès aux attributs / membres / méthodes
 - Garantit qu'une donnée ne peut pas être altérée n'importe comment
 - Accès en lecture seule, accès en lecture écriture, accès en écriture sous certaines conditions
 - Pour cela on leur associe une **visibilité**.
- En java, il y a 3 types de visibilité :
 - **public** : visible depuis n'importe où
 - **protected** : visible depuis la classe ou depuis ses classes filles et les classes du package !
 - **private** : visible uniquement dans la classe
 - Si on ne met pas de visibilité pour une classe, celle-ci n'est visible que dans le package. La seule visibilité possible d'une classe est **public**.
- Une visibilité s'applique à un constructeur, à un membre, à une méthode

Visibilité : règles de design

- En général les membres sont privés ou protégés. On y accède via les accesseurs qui eux sont publics.
 - Utilisation d'un accesseur **public** (getter, setter)
 - Hormis dans les accesseurs, on n'accède jamais directement à l'attribut
 - Avantage : la manière dont sont stockées les données n'impacte que les accesseurs. On peut donc les modifier aisément.

```
public class Personne{  
    private String nom;  
  
    public void setNom(String nom) {  
        // this désigne l'instance courante  
        this.nom = nom;  
    }  
    public String getNom() {  
        return this.nom;  
    }  
}
```

Typologie des méthodes d'une classe

- Parmi les différentes méthodes que comporte une classe, on peut distinguer:
 - Les constructeurs;
 - Les méthodes d'accès (accessor en anglais) qui fournissent des informations relatives à l'état d'un objet, sans les modifier;
 - Les méthodes d'altération (mutator en anglais) qui modifient l'état d'un objet.
- On rencontre souvent l'utilisation de noms de la forme getXXX pour les méthodes d'accès et setXXX pour les méthodes d'altération.

Les méthodes de classes:

- Certaines méthodes sont classiques...
 - Accès en lecture : `String getPrenom()`, `int getAge()`, ...
 - Accès en écriture : `void setPrenom(String sPrenom)`, `void setAge(int iAge)`, ...
 - On verra par la suite l'intérêt des getters et des setters.
 - En java l'affichage : `String toString()`
- ...d'autres dépendent de ce qu'on veut implémenter :
 - `void faireDevoirs()` ...

Exemple (Société)

- Une société est définie par deux informations (nom et ville).
- Une société a toujours un nom qui est représenté par une chaîne de caractères.
- La société peut ne pas être associée à une localité
- Une société est capable de se décrire en affichant les informations qui la caractérisent via une méthode.
- Le but ici est de définir ce qui caractérise cet objet et ce que cet objet peut faire.

Exemple (Société)

```
public class Societe{  
    // nom et localite : variables d'instance générées  
    // pour chaque objet de la classe  
    String nom;  
    String loc;  
    // methode qui affiche les caractéristiques  
    // de l'objet  
    void decrisToi(){  
        System.out.println("La societe s'appelle " + nom);  
        System.out.println("Elle est localisee a " + loc);  
    }  
}
```

Une classe = un fichier java

Exemple (Société)

```
public class TestSociete{  
    public static void main(String args[]){  
        Societe laSociete;  
        laSociete =new Societe();  
        laSociete.nom= new String("google");  
        laSociete.decrisToi();  
        laSociete.loc="PARIS";  
        laSociete.decrisToi();  
    }  
}
```

Une classe de test avec la création d'une instance de la classe Société.

Exercice (Société)

- Reproduire la classe Société avec une fonction affiche. Cette fonction renvoie les informations (nom et la localité) d'une société.
- Instancier deux sociétés et afficher les informations des deux sociétés.

Exercice (Société)

- Instancier la classe société avec un constructeur avec paramètres.
- En respectant les règles de design, faire les modifications nécessaires pour ajouter des getters/setters dans la classe société.

Polymorphisme (1/2)

- Polymorphisme: peut prendre plusieurs formes
- Polymorphisme d'héritage
 - Possibilité de redéfinir une méthode dans une classe fille (qui hérite d'une classe mère): spécialisation.
 - On peut alors appeler la méthode d'un objet sans se soucier de son type en se basant sur la classe mère (ou classe de base)

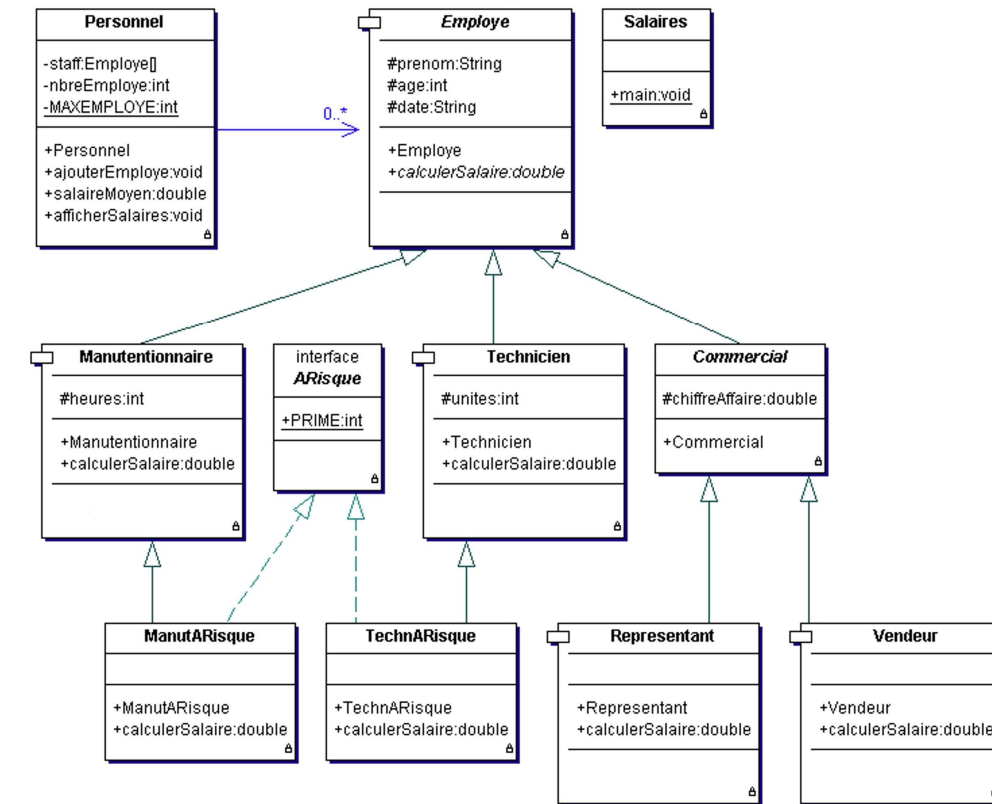
Polymorphisme (2/2)

- Exemple
 - Un jeu d'échec comporte des objets:
 - « roi », « reine », « fou », « cavalier », « tour » et « pion »
 - Chacun hérite de l'objet « pièce »
 - La méthode *déplacement()* peut avec le polymorphisme d'héritage effectuer le mouvement de la pièce sans savoir quelle pièce se cache réellement derrière le type de base

Exercice (Société)

- Reprendre l'exemple de la Société, et remplacer l'attribut loc (pour localité) par une variable de type Adresse.
- Nous écrirons donc la classe Adresse avec les informations suivantes:
- nom de rue, numéro de rue et nom de la ville.
- Il faudrait aussi mettre à jour les accesseurs de Société et la fonction affiche.
- Simuler dans la main la création de 2 sociétés. Ces dernières seront affectée par une même adresse.
- Afficher les informations sur les sociétés puis modifier l'adresse. Réafficher les informations et observez le résultat.

Exercice (ne pas faire la classe Salaires)



Exercice

Supposons des comptes en banque...

- Classes :
 - Compte (id, nom, solde) :
 - getId
 - getNom
 - getSolde
 - consulterSolde()
 - déposerArgent()
 - retirerArgent()
 - faireUnVirement()
 - voirHistorique() -> pour voir l'historique de toutes les transactions (virements, depots et retraits)

Exercice (compte bancaire)

- Un compte bancaire possède à tout moment une donnée : son solde. Ce solde peut être positif (compte créditeur) ou négatif (compte débiteur).
- Chaque compte est caractérisé par un code incrémenté automatiquement.
- le code et le solde d'un compte sont accessibles en lecture seulement.
- A sa création, un compte bancaire a un solde nul et un code incrémenté.
- Il est aussi possible de créer un compte en précisant son solde initial.
- Utiliser son compte consiste à pouvoir y faire des dépôts et des retraits. Pour ces deux opérations, il faut connaître le montant de l'opération.
- L'utilisateur peut aussi consulter le solde de son compte par la méthode `ToString()`.
- Un compte Epargne est un compte bancaire qui possède en plus un champ « `tauxInteret=6` » et une méthode `calculInteret()` qui permet de mettre à jour le solde en tenant compte des intérêts.
- Un ComptePayant est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 10 centimes.

- Définir la classe `Compte`.
- Définir la classe `CompteEpargne`.
- Définir la classe `ComptePayant`.
- Créer un programme permettant de tester ces classes avec les actions suivantes:
 - Créer une instance de la classe `Compte`, une autre de la classe `CompteEpargne` et une instance de la classe `ComptePayant`
 - Faire appel à la méthode `deposer()` de chaque instance pour déposer une somme quelconque dans ces comptes.
 - Faire appel à la méthode `retirer()` de chaque instance pour retirer une somme quelconque de ces comptes.
 - Faire appel à la méthode `calculInteret()` du compte `Epargne`.
 - Afficher le solde des 3 comptes.

Exercice suite

Donner la possibilité de faire des virements entre comptes

Exceptions / écriture

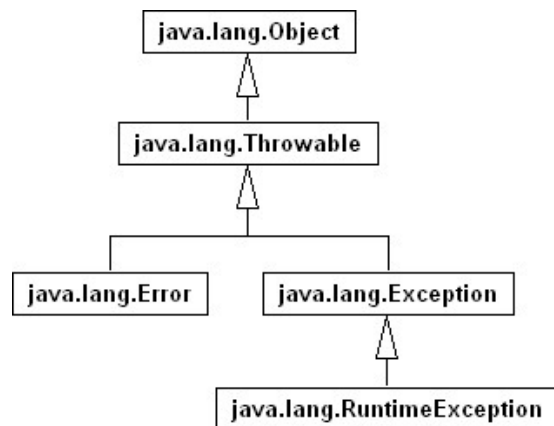
Exceptions

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java.

Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) mais aussi de les lever ou les propager (throw et throws).

Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.

Ces mécanismes permettent de renforcer la sécurité du code Java.



```
try {
    operation_risquée1;
    opération_risquée2;
} catch (ExceptionInteressante e) {
    traitements
} catch (ExceptionParticulière e) {
    traitements
} catch (Exception e) {
    traitements
} finally {
    traitement_pour_terminer_proprement;
}
```

Try / Catch exemple

```
public static void main(String[] args){  
    try {  
        System.out.println(" =>" + (1/0));  
    } catch (ClassCastException e) {  
        e.printStackTrace();  
    }  
    finally{  
        System.out.println("action faite systématiquement");  
    }  
}
```

```

public class Main {

    public static void main(String[] args) {

        try {
            Ville paris = new Ville( nom: "Paris", nbHabitants: -4334);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}

```

```

public class Ville {

    private String nom;
    private int nbHabitants;

    public Ville(String nom, int nbHabitants) throws Exception {
        this.nom = nom;
        if (nbHabitants < 0)
            throw new Exception("Nombre habitant négatif");
        else
            this.nbHabitants = nbHabitants;
    }

}

```

```

/Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java "-javaagent:
java.lang.Exception: Nombre habitant négatif
    at com.company.Ville.<init>(Ville.java:11)
    at com.company.Main.main(Main.java:8)

```

Process finished with exit code 0


```
public class Main {
```

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

```
        try {
```

```
            Ville paris = new Ville( nom: "Paris", nbHabitants: -4334);
```

```
        } catch (NombreHabitantException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

```
public class Ville {
```

```
    private String nom;
```

```
    private int nbHabitants;
```

```
    public Ville(String nom, int nbHabitants) throws NombreHabitantException {
```

```
        this.nom = nom;
```

```
        if (nbHabitants < 0)
```

```
            throw new NombreHabitantException();
```

```
        else
```

```
            this.nbHabitants = nbHabitants;
```

```
    }
```

```
}
```

```
public class NombreHabitantException extends Exception {
```

```
    public NombreHabitantException(){
```

```
        System.out.println(
```

```
            "Impossible d'instancier une classe Ville " +
```

```
            "avec un nombre d'habitants négatif !");
```

```
    }
```

```
}
```

```
/Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java "-javaagent  
Impossible d'instancier une classe Ville avec un nombre d'habitants négatif !  
com.company.NombreHabitantException  
    at com.company.Ville.<init>(Ville.java:11)  
    at com.company.Main.main(Main.java:8)
```

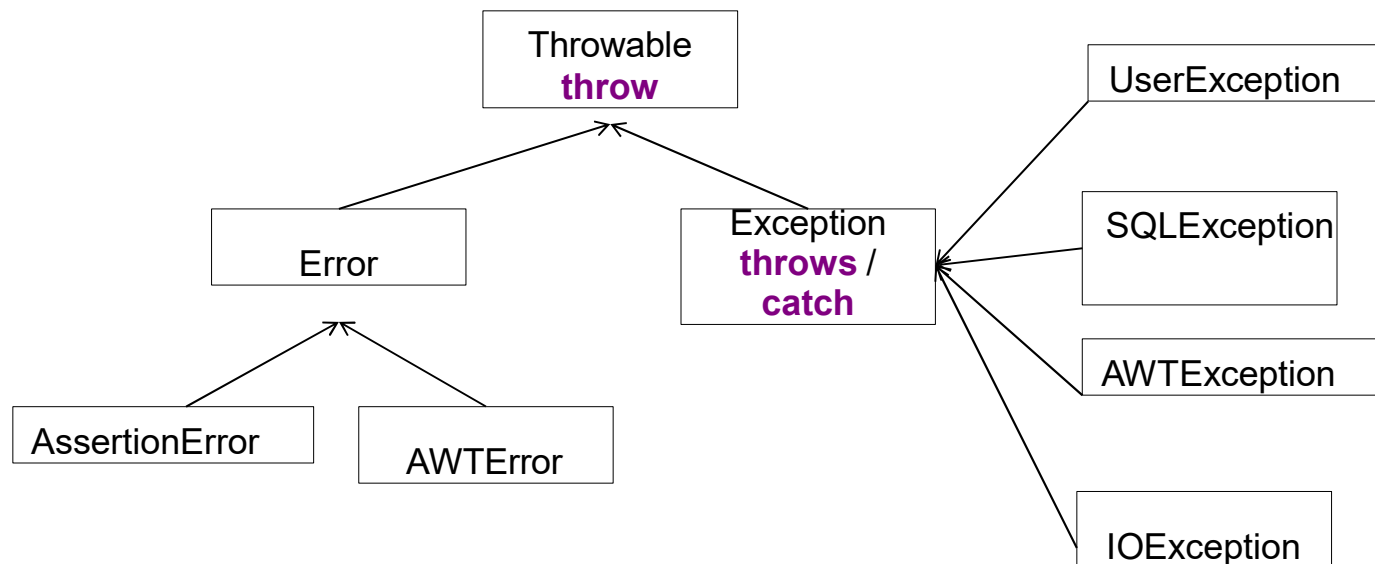
```
Process finished with exit code 0
```

Principe

- En java, déclarer une exception impacte le prototype de la méthode
 - `double` diviser(`double` a, `double` b) `throws` `DivisionParZeroException`
 - Une méthode peut lever différentes exceptions et en plusieurs endroits.
 - `void` f(`int` x) `throws` `MonException1`, `MonException2`
- Une exception interrompt l'exécution de la méthode
 - Dans ce cas-là, on pourra rattraper (`try ... catch`) au niveau de la méthode qui a appelé « diviser » un objet de type `Exception`.
 - Une méthode peut recevoir une exception et se contenter de la transmettre à la fonction qui l'a appelée
- On peut définir ses propres exceptions
 - Java fournit un certain nombre d'exceptions prédéfinies
 - `Exception` = exception générique

Erreurs et exceptions

- Pour les erreurs très graves (sous-entendues qu'on ne va pas pouvoir catcher proprement), on utilise la classe **Error**
 - hérite de Throwable (donc on peut faire un **throw**)
 - a priori pas déclarée derrière **throws** et pas rattrapées



Syntaxe

- Une méthode lance une exception avec le mot clé **throw**.
- En java, elle doit être indiquée derrière le **throws** du prototype.
- En java, si une méthode f appelle une méthode g et si g peut lever une (des) exception(s), alors f doit rattraper cette (ces) exception(s)
 - bloc **try** { } **catch** (...) { }

```
public class TestException {  
    public static void main(String args[]){  
        try{  
            System.out.println('Ma division ' + 3.2/0);  
            // lever une exception => dans un try  
        } catch (Exception e) {  
            System.out.println("Division par 0 !");  
            e.printStackTrace();  
        }  
    }  
}
```

Syntaxe

- On peut imbriquer des blocs **try** ... **catch**
- Si une méthode lève plusieurs types d'exceptions, on peut traiter chaque exception différemment
 - de la plus pertinente à la moins pertinente
 - **catch**(Exception e) {...} rattrape toutes les exceptions pas encore rattrapées
 - Le bloc **finally** est toujours exécuté (exception levée ou non)
 - Fermeture d'un flux, d'un socket...

```
try {  
    f(); // f throws MonException1, MonException2, MonException3  
} catch (MonException2 e) { // si f lève MonException2  
    return;                // --> finally  
} catch (Exception e) {    // si f lève MonException1, MonException3  
    e.printStackTrace();  
} finally {  
    // traité quoi qu'il arrive  
}
```

TP : Les exceptions

- Créer une classe `FeuSignalisation` possédant une méthode destiné à pouvoir changer la couleur du feu avec une couleur en argument.
- Lancer une exception si l'argument est incorrect.

Définir ses propres exceptions

- Il suffit de créer une classe qui doit hériter de **Exception**
 - On ne peut appliquer le mot clé **throw** que si l'objet hérite de Throwable (ce qui est le cas de l'objet Exception).
 - ... et les constructeurs qui peuvent être instanciés derrière un **throw**.
 - Traditionnellement postfixée Exception.

```
public class DivisionParZeroException extends Exception {  
    private double numerateur;  
    private double denominateur;  
  
    public DivisionParZeroException(double n, double d) {  
        this.numerateur = n;  
        this.denominateur = d;  
    }  
  
    public getNumerateur() { return this.numerateur; }  
    public getDenominateur() { return this.denominateur; }  
}
```

Entrées et sorties

Les objets `FileInputStream` et `FileOutputStream`

C'est par le biais des objets `FileInputStream` et `FileOutputStream` que nous allons pouvoir :

- lire dans un fichier
- écrire dans un fichier.

Entrées et sorties

```
1 //Package à importer afin d'utiliser l'objet File
2 import java.io.File;
3
4 public class Main {
5     public static void main(String[] args) {
6         //Création de l'objet File
7         File f = new File("test.txt");
8         System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());
9         System.out.println("Nom du fichier : " + f.getName());
10        System.out.println("Est-ce qu'il existe ? " + f.exists());
11        System.out.println("Est-ce un répertoire ? " + f.isDirectory());
12        System.out.println("Est-ce un fichier ? " + f.isFile());
13
14        System.out.println("Affichage des lecteurs à la racine du PC : ");
15        for(File file : f.listRoots())
16        {
17            System.out.println(file.getAbsolutePath());
18            try {
19                int i = 1;
20                //On parcourt la liste des fichiers et répertoires
21                for(File nom : file.listFiles()){
22                    //S'il s'agit d'un dossier, on ajoute un "/"
23                    System.out.print("\t\t" + ((nom.isDirectory() ? nom.getName()+"/" :
24nom.getName()));
25                    if((i%4) == 0){
26                        System.out.print("\n");
27                    }
28                    i++;
29                }
30                System.out.println("\n");
31            } catch (NullPointerException e) {
32                //L'instruction peut générer une NullPointerException
33                //s'il n'y a pas de sous-fichier !
34            }
35        }
36    }
37 }
```

Problems @ Javadoc Declaration Console X

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 août 2012 13:40:53)

Chemin absolu du fichier : C:\workspace\File\test.txt
Nom du fichier : test.txt
Est-ce qu'il existe ? true
Est-ce un répertoire ? false
Est-ce un fichier ? true
Affichage des lecteurs à la racine du PC :
C:\

\$Recycle.Bin/	.rnd	Apps/	autoexec.bat
BACKUP DD/	config.sys	dell/	dell.
Documents and Settings/	Drivers/	eula.1028.txt	
eula.1033.txt	eula.1036.txt	eula.1040.txt	
eula.1042.txt	eula.1049.txt	eula.2052.txt	
glassfish3/	globdata.ini	hiberfil.sys	
install.ini	install.res.1028.dll	install.res.1	
install.res.1036.dll	install.res.1040.dll	insta	
install.res.1049.dll	install.res.2052.dll	insta	
IO.SYS	LandparkIP/	log2.txt	logPe
mcdp.log	MSDOS.SYS	MSOCache/	
Partage/	PerfLogs/	Program Files/	

Exemple pour écrire sur un fichier

```
ArrayList<String> lignes = new ArrayList<>();  
lignes.add("première ligne");  
lignes.add("deuxième ligne");  
Path path = Paths.get(first: "test.txt");  
Files.write(path, lignes, StandardCharsets.UTF_8);
```

IO Simples

Un fichier peut-être lu/écrit en utilisant différentes classes de Java.

Cela permet d'accéder aux données du plus bas au plus haut niveau, suivant que le développeur nécessite de comprendre les données.

Au niveau le plus bas, la lecture peut se faire au niveau de l'octet.

Dans ce cas, le fichier est vu comme un flux d'octets. Une lecture s'arrête quand l'opération **read()** renvoie -1.

De manière très similaire, un fichier peut-être vu comme un flux de caractères.

La différence réside dans la gestion de l'internationalisation: le caractère, stocké en Unicode, est projeté dans le jeu de caractères local.

```
FileReader  
inputStream = new  
FileReader("xanadu.t  
xt");  FileWriter  
outputStream = new  
FileWriter("caracte  
routput.txt");  
// code identique au précédent  
pour lecture/écriture
```

```
FileInpu  
tStream  
in =  
null;  
FileOutp  
utStream  
out =  
null;  
try { in = new  
FileInputStream("xanadu.txt");  
out = new  
FileOutputStr  
eam("outagain  
.txt");  int  
c;  
while ((c  
=  
in.read(  
)) != -  
1) {  
out.writ  
e(c);  
} catch ...
```

Sérialisation

Les objets peuvent être écrits directement dans des fichiers de donnée: on appelle cela le processus de sérialisation. Un objet est sérialisable s'il implémente l'interface **Serializable**. Aucune méthode n'est à implémenter: il s'agit d'une sorte de flag signalant que l'objet *peut* être sérialisé. Dans l'exemple ci-dessous, une instance de **Maison** peut être sérialisée :

```
/**
 *Une classe agrégé qui est attribut de
 *PersonneIO.
 */
package io;

import java.io.Serializable;

/**
 *Important: doit lui aussi être Serializable
 *sous peine de:
 *java.io.NotSerializableException: io.Maison
 *lors de la serialisation de PersonneIO.
 */
public class Maison implements Serializable {
    public String adresse;
    public Maison(String adresse) {
        this.adresse = adresse;
    }
}
```

Sérialisation: dépendances

L'intérêt de la sérialisation est d'automatiser l'écriture des dépendances de classes. L'écriture d'une classe agrégant plusieurs objets provoque l'écriture des objets agrégés. Dans l'exemple suivant, l'instance de **Maison** est écrit automatiquement dans le fichier lorsque l'instance de **PersonneIO** est sérialisée.

```
/** Classe qui va être sérialisée. */  
package io;  
import java.io.Serializable;  
  
public class PersonneIO implements  
Serializable {  
  
    private int num_compte_bancaire = 8878;  
    public String nom = "none";  
    public Maison m;  
    public PersonneIO() {  
        m = new Maison("Mehun");  
    }  
    protected void  
        setNom(String  
            n) { nom = n;  
        }  
    public void setAdresse(String a) {  
        this.m.adresse = a;  
    }  
}
```

197

Sérialisation: écriture

Lors du processus d'écriture, les objets sont sérialisés à l'aide de la classe **ObjectOutputStream**.

Un identifiant unique de sérialisation est calculé à la compilation :

chaque instance sérialisée possède cet identifiant.

Cela empêche le chargement d'un objet sérialisé ne correspondant plus à une nouvelle version de la classe correspondante.

```
package io;
import
java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException; import
java.io.ObjectOutputStream;
import java.io.Serializable;
public class PersonneIOMainWrite implements
Serializable {
    public static void
        main(String[]
            args) { PersonneIO
                p = new
                PersonneIO();
                p.setNom("JFL");
                PersonneIO p2 =
                new
                PersonneIO();
                p2.setAdresse("?
                ");
                FileOutputStream
                fos;
                try {    fos = new
                    FileOutputStream("file.out
                    "); ObjectOutputStream
                    oos = new
                    ObjectOutputStream(fos);
                    oos.writeObject(p);
                    oos.write
                    Object(p2
                    );
                    oos.close
                    (); }
                catch (FileNotFoundException e) {
                    e.printStackTrace(); }
                catch (IOException e) { e.printStackTrace();
                }}}}
```

Sérialisation: lecture

A la relecture des objets sérialisés, il est impératif de connaître l'ordre de sérialisation (on peut toutefois s'en sortir en faisant de l'introspection).

L'exception particulière qu peut être levée est **ClassNotFoundException** dans le cas ou la JVM ne trouve pas la définition de la classe à charger.

```
package io;
import java.io.FileInputStream;
import
java.io.FileNotFoundException;
import java.io.IOException; import
java.io.ObjectInputStream;
import java.io.Serializable;
public class PersonneIOMainRead implements
Serializable {
    public static void main(String[] args) {
        try { FileInputStream is =
            new
                FileInputStream("file.out"
                ); ObjectInputStream in =
                new ObjectInputStream(is);
                PersonneIO p =
                (PersonneIO)in.readObject(
                ); PersonneIO p2 =
                (PersonneIO)in.readObject(
                );
                System.out.println(p.nom +
                " habite " + p.m.adresse);
                System.out.println(p2.nom +
                " habite " + p2.m.adresse);
                in.close();
            } catch
                (FileNotFoundException e) {
                    e.printStackTrace();
                } catch
                (IOException e) {
                    e.printStackTrace();
                } catch
                (ClassNotFoundException e) {
                    e.printStackTrace();
                }
        }
    }
}
```

Graphical User Interface (GUI)

- Pour créer une application lourde java graphique, il est nécessaire de créer une **GUI** (Graphical User Interface).
- On utilise principalement deux packages
 - **awt** : abstract window toolkit
 - **swing** : basé sur awt. Remplace progressivement awt.
 - plus performant
 - architecture **MVC** (Modèle Vue Contrôleur)
- Pour créer la GUI on peut :
 - coder l'interface à la main
 - utiliser un **WYSIWYG** (what you see is what you get)
 - eclipse + plugin (**window builder**, jigloo...)
 - netbeans (IDE concurrent d'eclipse)

Présentation de Swing

- Qu'est-ce que Swing ?
 - C'est une librairie écrite en JAVA
 - Développée par Sun pour remplacer AWT
 - Permet de dessiner des interfaces graphiques complexes
 - Présente de manière native depuis Java 1.2
 - Le but étant d'être complètement détaché de l'OS sur lequel l'application sera lancée. Là où AWT était plus lié au système.

Pourquoi Swing ?

- Existe depuis de nombreuses années et bénéficie de nombreuses ressources sur Internet
- Propose de multiple Look & Feel disponibles partout sur la toile
- Multiplateforme simple et efficace

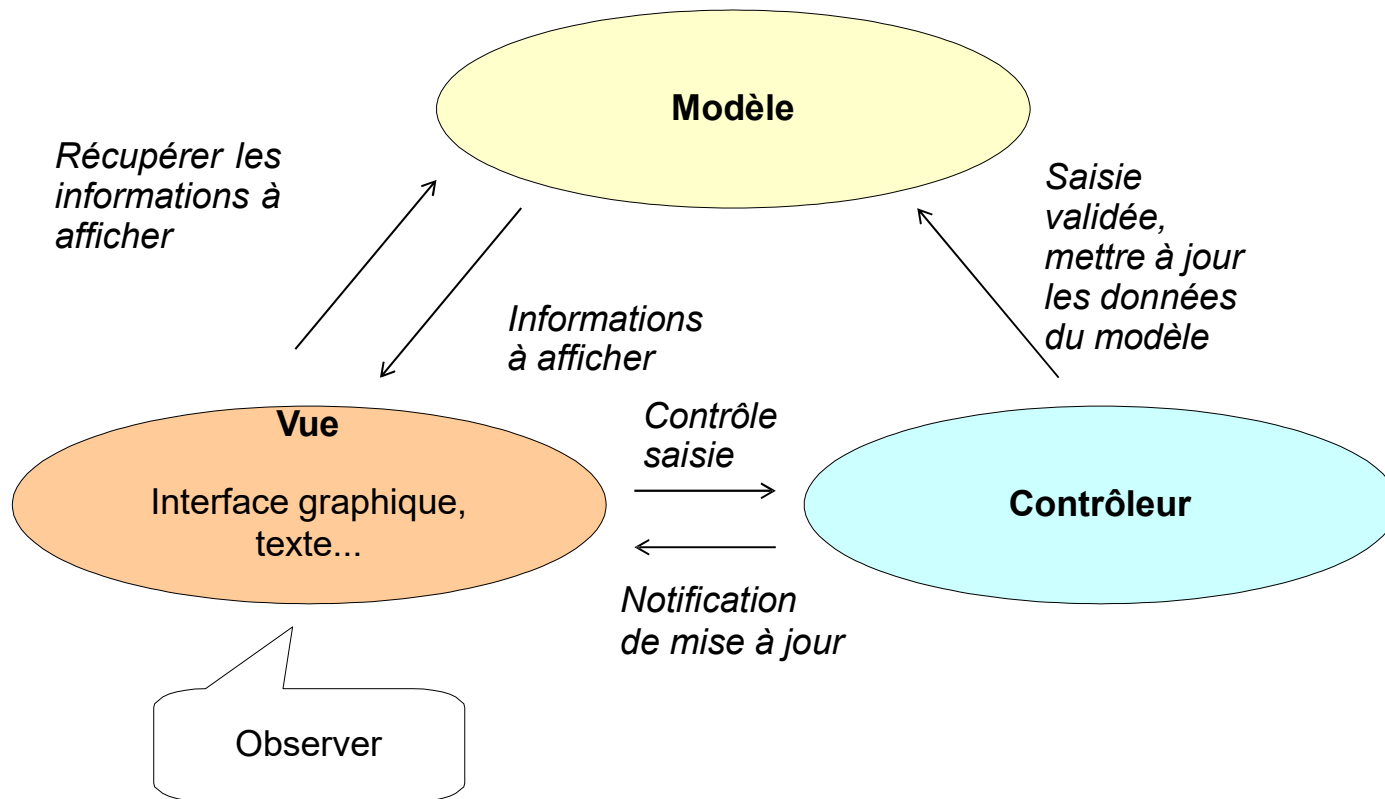
Conception d'une interface

- Principales étapes de conception d'une interface graphique
 - Choisir le type d'interface en fonction du besoin (applet, fenêtre...)
 - Pour chaque composant de l'interface
 - Le créer
 - Si besoin, le personnaliser
 - Le placer sur l'interface
 - Y associer un comportement

Modèle Vue Contrôleur (MVC)

- Le **modèle** contient l'intelligence de l'application
 - Exemple : les mouvements et la position des pièces du jeu d'échec
- La **vue** permet à l'utilisateur de manipuler le programme
 - Interface graphique (client lourd, page web...)
 - Interface en mode texte
- Le **contrôleur** pilote le programme.
 - Réagit aux événements de la vue (clic sur un bouton...)
 - Informe la vue quand elle doit être mise à jour
 - Afficher un message d'erreur
 - Mettre à jour les informations affichées
 - Contrôle les données reçues de la vue avant de les transmettre au modèle
 - Exemple : coordonnées des cases dans le jeu d'échec
- Le programme principal (main) lie les trois.

Modèle Vue Contrôleur (MVC)



Composants de base : JComponent

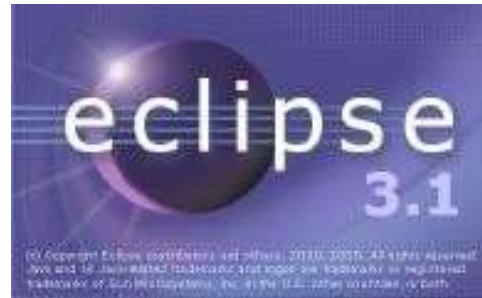
- Composant de base pour la quasi-totalité des objets utilisables en Swing à l'exception de :
 - JWindow, JFrame, JDialog, JApplet
- On retrouvera grâce à cela un nombre d'options toujours disponibles comme :
 - La transparence
 - L'état : actif, inactif
 - La bulle d'information
 - La bordure
 - Taille minimale, maximale
 - Alignement
 - ...

Définir une fenêtre

- Les principales fenêtres existantes :
 - JWindow
 - JFrame
 - JDialog
 - JOptionPane
 - JFileChooser / JColorChooser

Fenêtre : Jwindow

- Fenêtre simple sans bordures, décorations et boutons
- Utilisé pour les splashscreens ou les notifications
- Exemple:



Fenêtre :JFrame

- Propose la barre de titre et les boutons
- Possibilité d'enlever cette décoration

```
setUndecorated(true|false);
```



- Fermer l'interface
- Cliquer sur la croix ne suffit pas à arrêter le programme

```
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

Exemple : swing

- Les classes swing sont préfixées J... (JFrame...)

- http://fr.wikipedia.org/wiki/Swing_%28Java%29

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorld {
    public static void main(String[] args) {
        // On crée une fenêtre dont le titre est "Hello World!"
        JFrame frame = new JFrame("Hello World!");

        // La fenêtre doit se fermer quand on clique sur la croix rouge
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        // On ajoute le texte "Hello, World!" dans la fenêtre
        frame.getContentPane().add(new JLabel("Hello, World!"));

        // On demande d'attribuer une taille minimale à la fenêtre
        // (juste assez pour voir tous les composants)
        frame.pack();

        frame.setLocationRelativeTo(null); // Centrer la fenêtre
        frame.setVisible(true);           // Rendre la fenêtre visible
    }
}
```

La classe JFrame

- Une JFrame de base n'est pas utilisable/paramétrée.
 - On peut préciser si la fenêtre est redimensionnable, toujours au premier plan, avec ou sans contours / boutons, sa taille, sa position, son comportement quand on clique sur fermer, si elle est visible...
 - On fait un **héritage** sur JFrame pour chacune de nos fenêtres

```
import javax.swing.JFrame;

public class Fenetre extends JFrame{

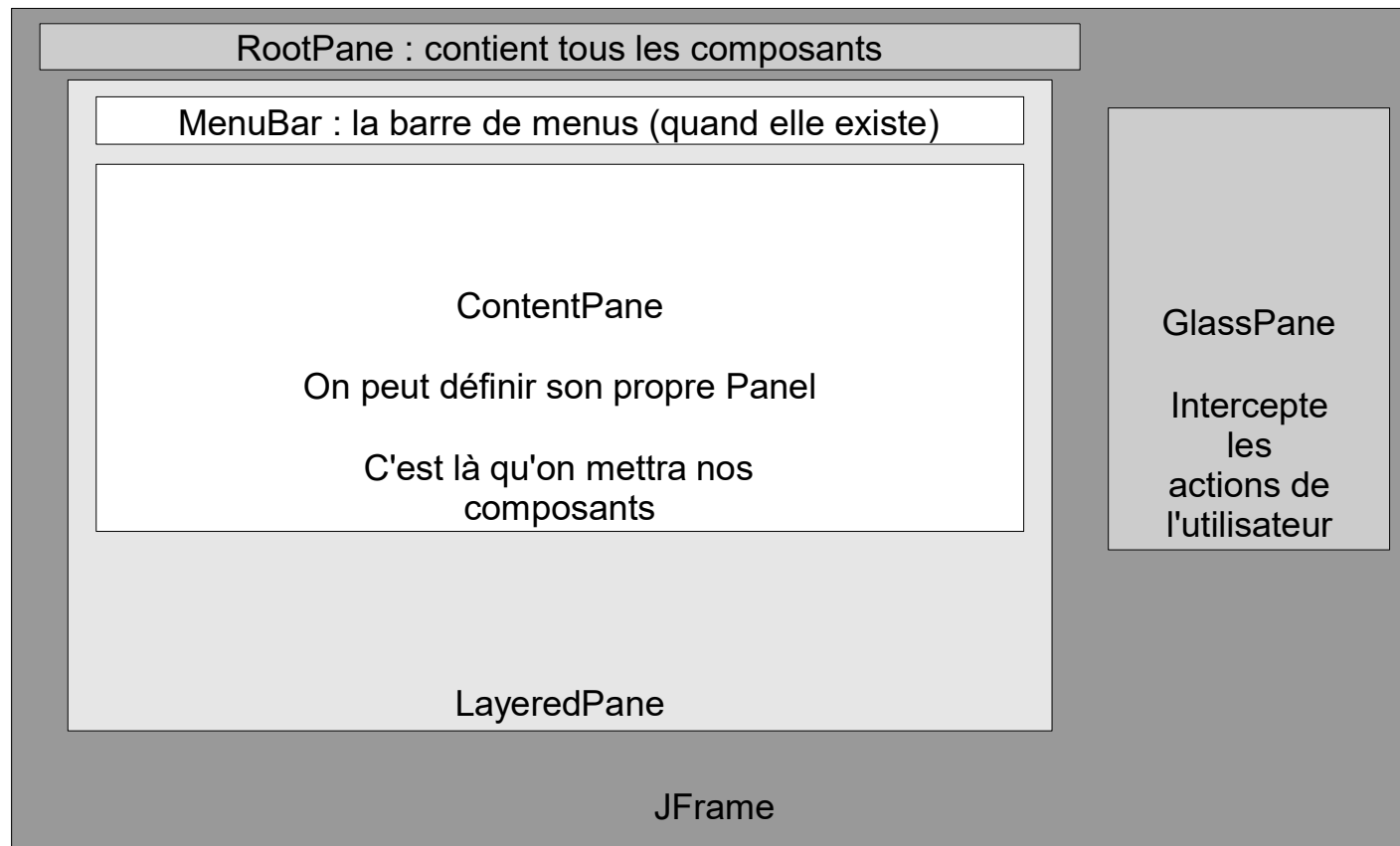
    public Fenetre(){
        this.setTitle("Titre de la fenêtre");
        this.setSize(400, 500);

        // Centrer à l'écran
        this.setLocationRelativeTo(null);

        // Fermer lorsqu'on clique sur "Fermer"
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

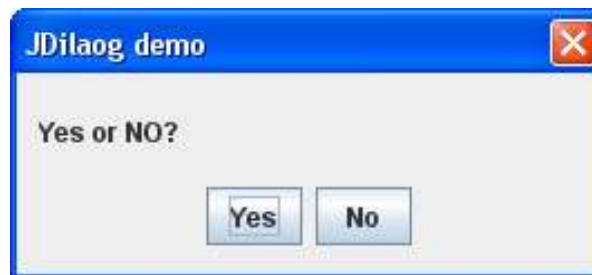
        // Par défaut un élément n'est pas visible !
        this.setVisible(true);
    }
}
```

La classe JFrame : structure



Fenêtre : JDialog

- Fenêtre fille d'une application
- Apparaît au dessus d'une autre fenêtre
- Possibilité de la rendre modale ou non
- Modale signifie ne pas pouvoir interagir avec la fenêtre mère tant que la boite de dialogue est ouverte



Fenêtre : JOptionPane

- Ce sont des boîtes de dialogues prédéfinies
- 3 types de boîtes :
 - Message d'information
`JOptionPane.showMessageDialog(...)`
 - Message contenant une question
`JOptionPane.showConfirmDialog(...)`
 - Message de saisie de données
`JOptionPane.showInputDialog(...)`



Fenêtre : JFileChooser

- Un JFileChooser permet de sélectionner un fichier en parcourant l'arborescence du système de fichier.
- Ex :
 - ```
JFileChooser fc = new JFileChooser();
int returnVal = fc.showOpenDialog(aFrame);
if (returnVal == JFileChooser.APPROVE_OPTION)
{
 File file = fc.getSelectedFile();
}
```



## Fenêtre : JColorChooser

- Un JColorChooser permet de choisir une couleur



- Une méthode :  
`public static Color showDialog(Component c , String title , Color initialColor);`



# Contenaire

- Les conteneurs contiennent et gèrent des composants graphiques.
- Ils dérivent de `java.awt.Container`.
- A l'exécution, ils apparaissent généralement sous forme de panneaux, de fenêtres ou de boîtes de dialogues.
- La totalité de votre travail de conception d'interfaces utilisateurs se fait dans des conteneurs.
- Les conteneurs sont aussi des composants. En tant que tels, ils vous laissent interagir avec eux, c'est-à-dire définir leurs propriétés, appeler leurs méthodes et répondre à leurs événements.

# Contenaire

- Composants qui ont pour but principal de contenir d'autres composants
- Les principaux conteneurs :
  - JPanel
  - JScrollPane
  - JTabbedPane
  - ...

# JPanel

- Le container le plus simple
- N'a quasiment aucun impact sur l'aspect graphique d'une interface
- Sert surtout à positionner de nouveaux composants
- Principales méthodes :
  - `setLayout(...)`
  - `add(...)`
- Un JPanel peut également jouer le rôle de surface graphique dans laquelle le développeur peut dessiner ou afficher des images. Il suffit pour cela de redéfinir la méthode `paintComponent(Graphics)`.

# La classe JPanel

- Contient le fond + les contrôles de la fenêtre
  - Agence les composants, redessine en cas de besoin...
  - On peut obtenir les dimensions du `JPanel`
  - On peut personnaliser n'importe quel composant (bouton, panel...) en réimplémentant `paintComponent(Graphics)` : son fond, un dessin
  - L'objet `Graphic` permet de dessiner des formes, des lignes, du texte, une image...
    - Pour aller plus loin caster `Graphics` en `Graphics2D`

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
 public void paintComponent(Graphics g) {
 g.setColor(Color.ORANGE);
 g.fillOval(20, 20, 75, 75);
 }
}
```

```
// Dans le constructeur Fenêtre() on ajoute :
this.setContentPane(new Panneau());
```

# JScrollPane

- Permet d'avoir des barres de défilement lorsque le composant qu'il contient est trop grand par rapport à la taille qui lui est allouée.



# JTabbedPane

- Reprend le système des onglets
- Permet d'avoir plusieurs panneaux sur la même surface

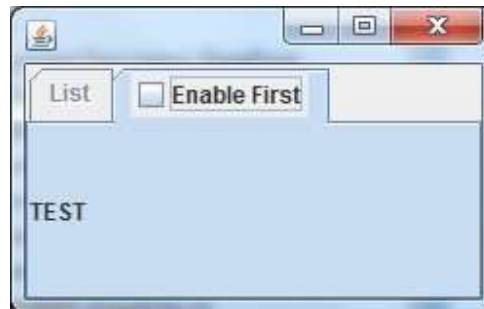
```
...
JTabbedPane tab = new JTabbedPane();
tab.addTab("onglet 1", new JLabel("Panneau 1"));
JTabbedPane tab = new JTabbedPane();
tab.addTab("onglet 2", new JLabel("Panneau 2"));
...
```



# JTabbedPane

- Il est possible d'ajouter un composant à la place du titre

```
JTabbedPane tab = new JTabbedPane();
tab.addTab("List" , new JLabel("Panneau 1"));
tab.addTab(null, new JLabel("TEST"));
tab.setTabComponentAt(1, new JCheckBox("Enable First"));
```



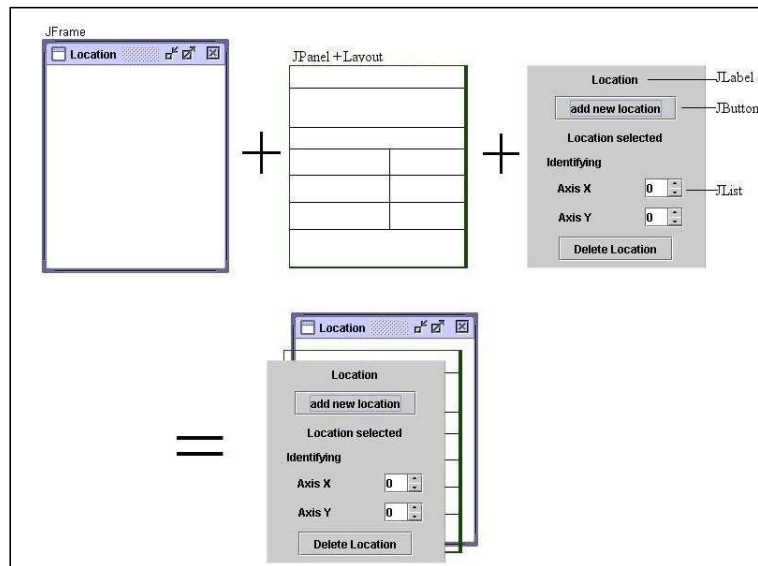
## Les composants

- Les composants sont les blocs de construction utilisés par les outils de conception visuelle de Netbeans ou autres pour construire un programme. (bouton, textbox....)
- Chaque composant représente un élément de programme, tel un objet de l'interface utilisateur, une base de données ou un utilitaire système. Vous construisez un programme en choisissant et **reliant ces éléments**.



# Schéma complet de développement d'une fenêtre Swing

- 1- création d'une fenêtre.(Jframe..)
- 2- Intégration d'un JPanel dans la fenêtre (sert à intégrer des composants)
- 3- utilisation d'un layout à l'intérieur du JPanel pour ordonnancer les composants.
- 4- intégration des composants dans le JPanel en utilisant son layout.



# JLabel

- Un JLabel permet d'afficher du texte ou une image.
- Un JLabel peut contenir plusieurs lignes et il comprend les balises HTML.

- ```
public JLabel(String s);  
public JLabel(Icon i);
```



JButton

- Un JButton nous permet d'avoir un bouton sur notre interface.

```
JButton bouton = new JButton("Select Me");  
panel.add(bouton);
```



La classe JButton

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame{
    private JPanel pan = new JPanel();
    private JButton bouton = new JButton("Mon Bouton");

    public Fenetre(){
        this.setTitle("Mon titre");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        pan.add(bouton); // sera centré par le JPanel
        // Si on n'utilise pas de JPanel, le panel ne remet pas
        // en forme le bouton qui occupera toute la JFrame !

        this.setContentPane(pan);
        this.setVisible(true);
    }
}
```

TextField

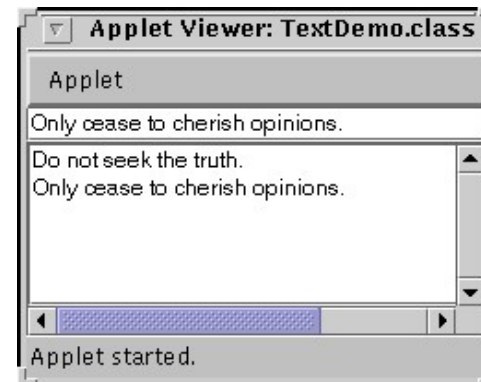
- Un `TextField` est un composant qui permet d'écrire du texte. Il est composé d'une seule ligne contrairement au `TextArea`
- Le `JPasswordField` permet de cacher ce qui est écrit

- Quelques méthodes:

```
public JTextField(String s);
```

```
public String getText();
```

```
public String setText();
```



JList

- Une JList propose plusieurs éléments rangés en colonne. Une JList peut proposer une sélection simple ou multiple. Les JList sont souvent contenues dans un scrolled pane.

- Quelques méthodes:

```
public JList(Vector v);  
public JList( ListModel l);  
Object[] getSelectedValues();
```



Les Layouts

- Un Layout permet de découper un Panel en zones

- BorderLayout : *CENTER, NORTH, SOUTH, EAST, WEST*

```
// Dans le constructeur Fenetre
this.setLayout(new BorderLayout());

//On ajoute le bouton au contentPane de la JFrame
this.getContentPane().add(new JButton("CENTER"), BorderLayout.CENTER);
this.getContentPane().add(new JButton("NORTH"), BorderLayout.NORTH);
this.getContentPane().add(new JButton("SOUTH"), BorderLayout.SOUTH);
this.getContentPane().add(new JButton("WEST"), BorderLayout.WEST);
this.getContentPane().add(new JButton("EAST"), BorderLayout.EAST);
```

- GridLayout : découpe le panneau en une grille dont le nombre de lignes et colonnes peut être précisé
 - FlowLayout : éléments placés de gauche à droite, passe à la ligne si besoin
 - Il en existe plein d'autres : CardLayout, GridBagLayout, ...
 - http://www.siteduzero.com/tutoriel-3-10480-votre-premier-bouton.html#ss_part_2

BorderLayout

- Le BorderLayout sépare un container en cinq zones: NORTH, SOUTH, EAST, WEST et CENTER
- Lorsque l'on agrandit le container, le centre s'agrandit. Les autres zone prennent uniquement l'espace qui leur est nécessaire.



- ```
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
contentPane.add(new JButton("Button 1 (NORTH)"),
BorderLayout.NORTH);
```



# FlowLayout

- Un FlowLayout permet de ranger les composants dans une ligne. Si l'espace est trop petit, une autre ligne est créée.



```
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

# GridLayout

- Un GridLayout permet de positionner les composants sur une grille.



- Ex:

```
Container contentPane = getContentPane();
contentPane.setLayout(new GridLayout(0,2));
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

# Gestion des évènements

- Les composants Swing créent des événements, soit directement, soit par une action de l'utilisateur sur le composant. Ces événements peuvent déclencher une action exécutée par d'autre(s) composant(s).
- Un composant qui crée des événements est appelé **source**. Le composant source délègue le traitement de l'événement au composant auditeur.
- Un composant qui traite un événement est appelé **auditeur** (listener)

# Gestion des évènements

- Les événements que nous venons de décrire sont occasionnés par l'utilisateur. on dira que ces événements sont générés par les composants graphiques eux-mêmes. Ces objets sont appelés des sources d'événements.
- Ex: JButton

|             |                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Action      | Lorsque le bouton a été actionné par l'utilisateur (appuyé et relâché)                                                                 |
| MouseMotion | Lorsque la souris est déplacée ou draguée sur la surface du bouton                                                                     |
| Mouse       | Lorsque le bouton de la souris est appuyé, relâché, ou cliqué, ou encore<br>lorsque la souris sort ou entre dans la surface du bouton. |
| Focus       | Lorsque le bouton obtient ou perd le focus (suite par exemple à la<br>pression de la touche TAB)                                       |
| Component   | Lorsque le bouton a été déplacé ou caché                                                                                               |
| Key         | Lorsqu'une touche du clavier est pressée alors que le bouton possède le<br>focus.                                                      |

# PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

- **1re étape:** choisir une catégorie d'événements
- Voici un tableau de tous les événements pour chaque composant, suivi de la liste des classes associées pour chaque événement.

| Component                            | Listener |       |        |                            |      |                   |        |                                                             |
|--------------------------------------|----------|-------|--------|----------------------------|------|-------------------|--------|-------------------------------------------------------------|
|                                      | action   | caret | change | document,<br>undoable edit | item | list<br>selection | window | other                                                       |
| <a href="#">button</a>               | x        |       | x      |                            | x    |                   |        |                                                             |
| <a href="#">check box</a>            | x        |       | x      |                            | x    |                   |        |                                                             |
| <a href="#">color chooser</a>        |          |       | x      |                            |      |                   |        |                                                             |
| <a href="#">combo box</a>            | x        |       |        |                            | x    |                   |        |                                                             |
| <a href="#">dialog</a>               |          |       |        |                            |      |                   |        |                                                             |
| <a href="#">editor pane</a>          |          | x     |        | x                          |      |                   | x      | <a href="#">hyperlink</a>                                   |
| <a href="#">file chooser</a>         | x        |       |        |                            |      |                   |        |                                                             |
| <a href="#">formatted text field</a> | x        | x     |        | x                          |      |                   |        |                                                             |
| <a href="#">frame</a>                |          |       |        |                            |      | x                 |        |                                                             |
| <a href="#">internal frame</a>       |          |       |        |                            |      |                   |        | <a href="#">internal frame</a>                              |
| <a href="#">list</a>                 |          |       |        |                            |      | x                 |        | <a href="#">list data</a>                                   |
| <a href="#">menu</a>                 |          |       |        |                            |      |                   |        | <a href="#">menu</a>                                        |
| <a href="#">menu item</a>            | x        |       | x      |                            | x    |                   |        | <a href="#">menu key</a><br><a href="#">menu drag mouse</a> |

# PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

| Component                                         | Listener |       |        |                            |      |                   |        |                                                                    |
|---------------------------------------------------|----------|-------|--------|----------------------------|------|-------------------|--------|--------------------------------------------------------------------|
|                                                   | action   | caret | change | document,<br>undoable edit | item | list<br>selection | window | other                                                              |
| <a href="#">option pane</a>                       |          |       |        |                            |      |                   |        |                                                                    |
| <a href="#">password field</a>                    | x        | x     |        | x                          |      |                   |        |                                                                    |
| <a href="#">popup menu</a>                        |          |       |        |                            |      |                   |        | popup menu                                                         |
| <a href="#">progress bar</a>                      |          |       | x      |                            |      |                   |        |                                                                    |
| <a href="#">radio button</a>                      | x        |       | x      |                            | x    |                   |        |                                                                    |
| <a href="#">slider</a>                            |          |       | x      |                            |      |                   |        |                                                                    |
| <a href="#">spinner</a>                           |          |       | x      |                            |      |                   |        |                                                                    |
| <a href="#">tabbed pane</a>                       |          |       | x      |                            |      |                   |        |                                                                    |
| <a href="#">table</a>                             |          |       |        |                            |      | x                 |        | table model<br>table column<br>model cell editor                   |
| <a href="#">text area</a>                         |          | x     |        | x                          |      |                   |        |                                                                    |
| <a href="#">text field</a>                        | x        | x     |        | x                          |      |                   |        |                                                                    |
| <a href="#">text pane</a>                         |          | x     |        | x                          |      |                   |        | hyperlink                                                          |
| <a href="#">toggle button</a>                     | x        |       | x      |                            | x    |                   |        |                                                                    |
| <a href="#">tree</a>                              |          |       |        |                            |      |                   |        | tree expansion<br>tree will expand<br>tree model<br>tree selection |
| viewport<br>(used by <a href="#">scrollpane</a> ) |          |       | x      |                            |      |                   |        |                                                                    |

# PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

- **2ème étape:** inscrire un auditeur pour une certaine catégorie d'événements
  - Le Composant.add**XXXListener** (*auditeur*);
  - Le Composant désigne le composant graphique auprès duquel on désire s'abonner
  - XXX désigne la catégorie d'événements concernée (Action, MouseMotion, ..)
  - auditeur désigne l'objet qui se met à l'écoute des événements

# PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

- **3<sup>ème</sup> étape:** écrire les gestionnaires d'événements dans la classe de l'auditeur

```
public void addXXXListener(XXXListener)
public void addActionListener(ActionListener)
public void
addMouseMotionListener(MouseMotionListe
ner)
```



# Les listeners

- **EventListener** : interface qui indique quels événements on rattrape pour un composant :

- Exemple **interface** `MouseListener`, ...

```
public class Bouton extends JButton implements MouseListener {
 // Constructeurs, membres, méthodes...
 // Listeners
 void mouseEntered(MouseEvent e) {}
 void mouseClicked(MouseEvent e) {}
 // ... et les autres Listeners
}
```

- Il ne reste plus qu'à implémenter les méthodes qui en découlent
  - Exemple : changer le style du composant
  - Si on n'a pas de code à mettre, on laisse les { } vides

# Les listeners

- Parfois le contrôle n'a assez d'information pour faire le traitement.

- On peut le déléguer à sa fenêtre qui peut stocker l'état du logiciel.
- Exemple : **interface** ActionListener

```
public class Fenetre extends JFrame implements ActionListener{
 private JButton bouton1 = new JButton("mon bouton 1");
 private JButton bouton2 = new JButton("mon bouton 2");
 private JLabel label = new JLabel("mon label");

 // Listeners
 public void actionPerformed(ActionEvent arg0) {
 if(arg0.getSource() == bouton1){
 label.setText("Action sur bouton 1");
 } else if(arg0.getSource() == bouton2){
 label.setText("Action sur bouton 2");
 }
 }
}
```

Pas lisible  
Pas découpé  
Pas pratique  
Pas modulaire

# Les listeners


## ■ Pour plus de lisibilité :

- on crée `Listener` personnalisé par contrôle (au lieu de `Fenetre`)
- on attache le `Listener` personnalisé avec `addListener`
- `addComponent` avec le `Listener` personnalisé correspondant

```
public class Fenetre extends JFrame{
private JButton bouton1 = new JButton("mon bouton 1");

// Constructeur
public Fenetre(ActionEvent arg0) {
 //...
 bouton1.addActionListener(new Bouton1Listener());
 // idem avec les autres composants
}

public class Bouton1Listener implements ActionListener{
 public void actionPerformed(ActionEvent arg0) {}
} // idem avec les autres composants
}
```



## TP: interface graphique

- Coder une interface pour gérer une liste des élèves
  - La liste nous permettra de voir tous les noms
  - Si on sélectionne un élément, ses informations seront affichées.111



Merci de votre attention 😊