

Formation java

A series of horizontal lines in teal, light blue, and white, stacked and offset to create a modern, layered effect across the middle of the slide.

Plan

Partie 1: notion de base

- Introduction a la POO
- Modélisation UML :
association, composition,
héritage
- Notion d'objet et structure
d'un programme

Partie 3: Programmation fonctionnelle

- Variables
- Type de base
- Fonction et visibilité
- Structures algorithmiques
- Tableaux

Partie 2 : 1^{er} pas en java

- Introduction
- Hello world
- Eclipse
- L'archiveur jar

Partie 4 : Programmation orienté objet

- Classes
- Constructeur
- Transmission de paramètre
- Héritage
- Polymorphisme

Plan

Partie 5 : notions avancés

- Lien dynamique
- Interfaces
- Les grandes règles de design
- Containers
- Classes génériques

Partie 7 : objet api

- Les flux appliqués aux fichiers
- SAX, DOM
- Exemple
- Javadoc
- Balises javadoc

Partie 6 : Exceptions

- Principe
- Syntaxe
- Créer ses propres exceptions
- Erreurs

Partie 8: GUI

- Swing

Objectifs

- Appréhender les concepts liés à la programmation orientée objet
 - classe, héritage, polymorphisme
- Comprendre les avantages apportés par la POO.
- Comment écrire un programme Java propre.
 - implémentation, documentation, tests
- Illustrer ces concepts de manière concrète au travers des TPs

Partie 1 : POO - notions de base

- Introduction a la POO
- Modélisation UML : association, composition, héritage
- Notion d'objet et structure d'un programme

La Programmation Orientée Objet (POO)

- La programmation orientée objet (POO) est un type de programmation qui a pour avantage de posséder une meilleure organisation des programmes, elle contribue à la fiabilité des logiciels et elle facilite la réutilisation de codes existants.
- C'est une programmation dirigée par les données et non par les traitements. Elle permet de mieux séparer les données des fonctions qui les manipulent, du coup on se concentre sur la description des données et non sur la façon dont on les manipule.
- La POO s'articule autour du concept de composant qu'on nomme :
" *Objet* ".

La Programmation Orientée Objet (POO)

- Donc la POO consiste en la définition et l'assemblage de briques logicielles appelées *objets* afin de résoudre une problématique donnée.
- Un objet est une entité composée de données caractérisant cet objet et de traitements s'effectuant sur ces données.
- Terminologie des objets :
 - Les données internes sont appelées « attributs » ou « membres »
 - Les traitements sur ces données sont appelés « méthodes »

La Programmation Orientée Objet (POO)

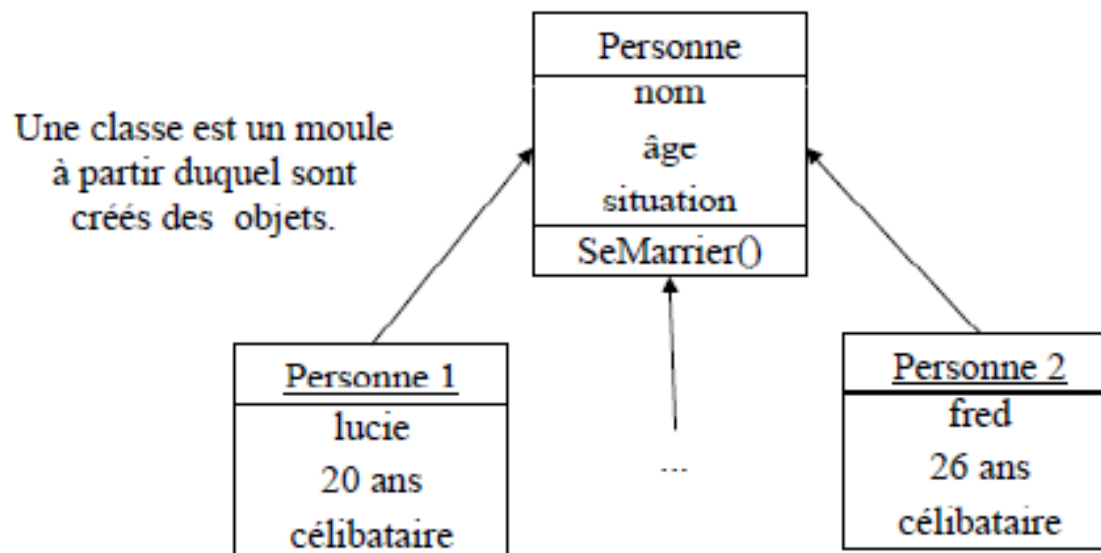
- La POO introduit de nouveaux concepts, en particulier ceux d'encapsulation, de classe et d'héritage.
- L'encapsulation: désigne le regroupement de données dans un objet, ainsi que les méthodes agissant exclusivement sur les données de l'objet.
 - Il n'est pas possible d'agir directement sur les données d'un objet; il est nécessaire de passer par ces méthodes, qui jouent ainsi le rôle d'interface obligatoire.
 - Avantage: L'objet est considéré comme une boîte noire dont les données sont sécurisées. Une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification. Enfin, l'encapsulation des données facilite grandement la réutilisation d'un objet.

La Programmation Orientée Objet (POO)

- Le concept de classe : un système complexe a un grand nombre d'objets. Pour réduire cette complexité, on regroupe les objets en classes.
- Une classe peut être considérée comme:
 - La description d'un ensemble d'objets ayant une structure de données commune (les attributs) et disposant des mêmes méthodes;
 - Un modèle (ou un moule) utilisé pour créer les objets;
 - La généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objet ayant une structure de données commune et disposant des même méthodes. Les objets apparaissent alors comme des variables d'un tel type classe.
- En POO, on dit qu'un objet est une *instance* de sa classe.

Programmation Orientée Objet (POO)

- Une classe doit être vue comme une sorte de moule.
- Chaque "variable" de ce type est appelée **instance** de classe et maintient son propre **état**.
- Chaque "champ" (ici prénom, nom, ...) est appelé **attribut** ou **membre** de classe.¹¹¹



Programmation Orientée Objet (POO)

- Le concept d'héritage: Il permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données et de nouvelles méthodes.
- L'intérêt est de pouvoir définir de nouveaux attributs et/ou méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.
- Avec l'héritage vient le concept de *Polymorphisme*: c'est la possibilité de redéfinir certaines des méthodes héritées de sa classe de base. Le polymorphisme permet ainsi de traiter de la même manière des objets de types différents, pour peu qu'ils soient issus de classes dérivées d'une même classe de base.
 - Le polymorphisme permet aux méthodes d'avoir plus de spécialisation.

Programmation Orientée Objet (POO)

- **Avantage de la POO:**
 - La modularité: le code source d'un objet peut être écrit et actualisé indépendamment du code source d'autres objets.
 - La confidentialité des informations: puisque un objet interagit avec d'autres objets via des méthodes, les informations concernant son implémentation interne reste cachées au monde externe.
 - La ré-utilisabilité: si un objet existe déjà (potentiellement créer par un autre programme) il est possible de l'utiliser dans votre programme.
 - La facilité de maintenance: si un objet devient problématique, il est possible de le remplacer par un nouveau.

La modélisation et la programmation objet

- Le développement en POO se fait en plusieurs temps :
 - 1) la **conception** du diagramme de classes (UML...), indépendante du langage
 - quelles classes ? quels liens entre les classes ?
 - 2) l'**implémentation**, spécifique au langage
 - coder, tester et documenter les classes
- Avantages : architecturer et factoriser le code, gain de temps, maintenance...

La modélisation avec UML

- Unified Modeling Language (UML) est un langage graphique qui permet de représenter et de communiquer les divers aspects d'un système d'information.
 - Aux graphiques sont bien sûr associés des textes qui expliquent leur contenu.
- UML comporte ainsi plusieurs types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information.

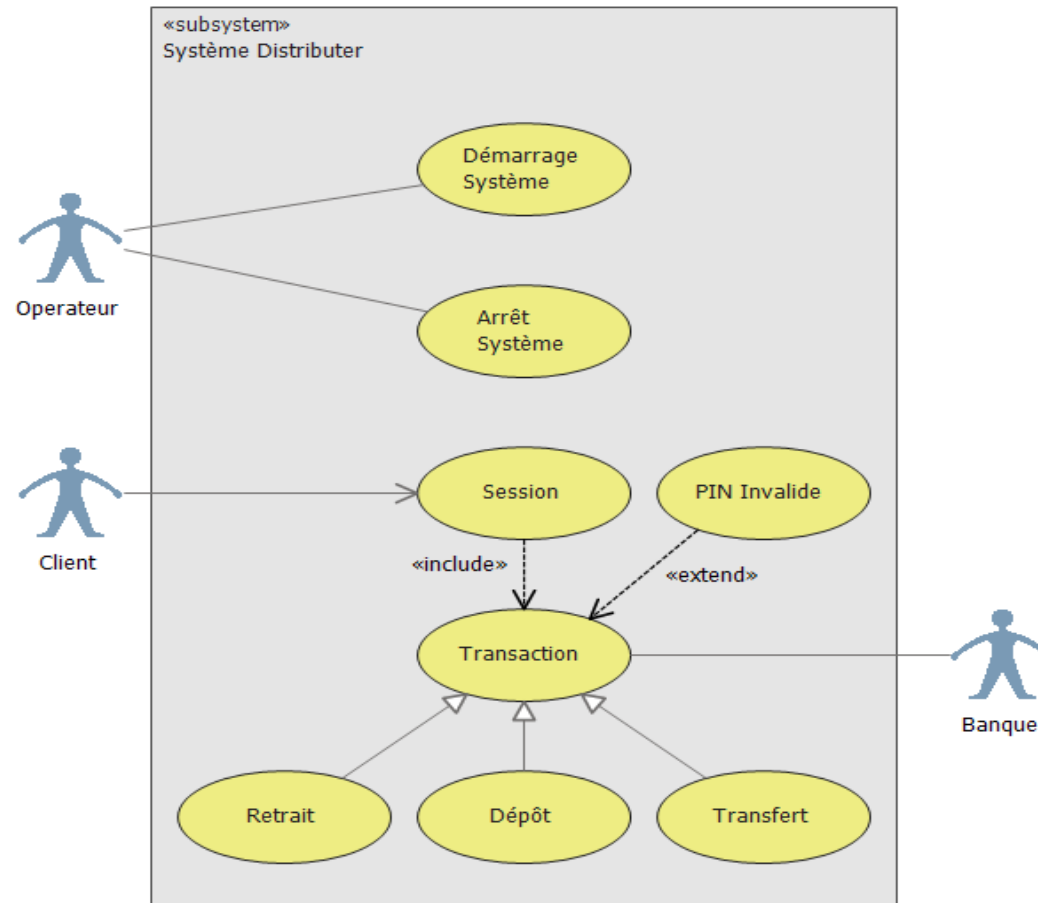
La modélisation avec UML

- UML peut être utilisé pour:
 - Représenter les limites d'un système et ses principales fonctions à l'aide de scénarios et d'acteurs,
 - Illustrer le déroulement des scénarios à l'aide de diagrammes d'interactions,
 - Représenter la structure statique du système avec des classes
 - Modéliser le comportement des objets à l'aide de diagrammes d'états
 - Révéler l'architecture d'implantation physique avec des diagrammes de déploiements
 - ...

Exemple de diagrammes UML

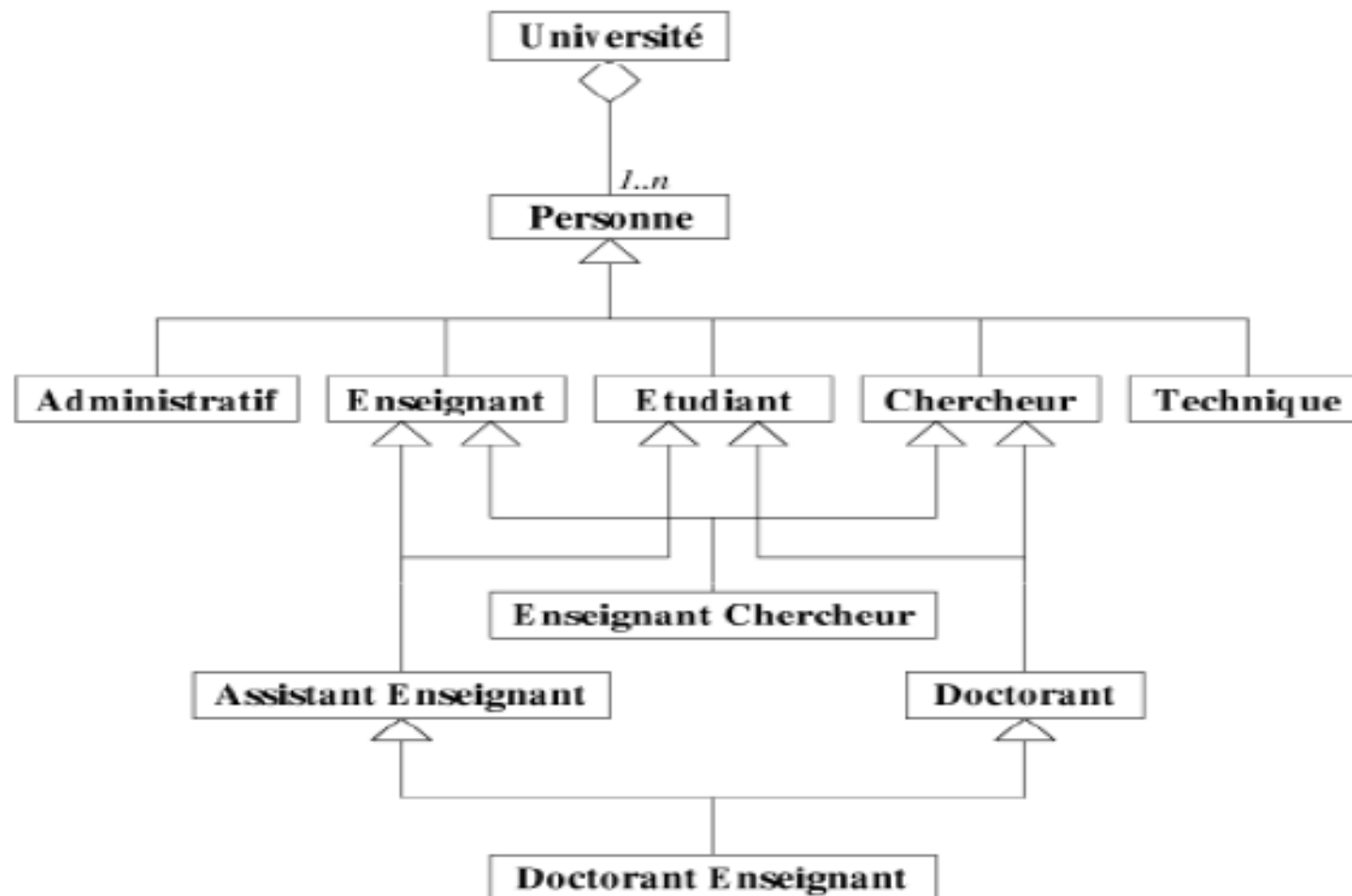
- Diagramme de cas d'utilisation:

uc Système Distributeur



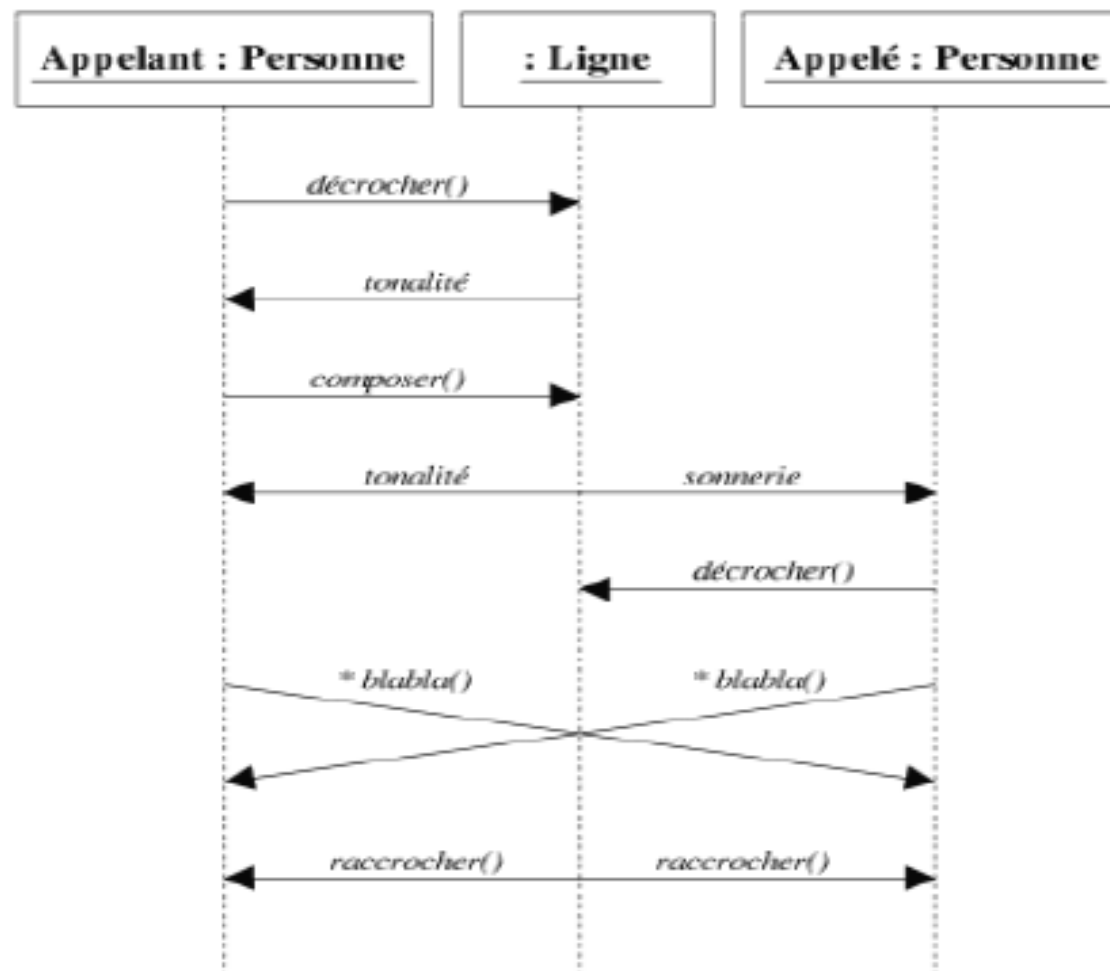
Exemple de diagrammes UML

- Diagramme de classes:



Exemple de diagrammes UML

- Diagramme de séquence:



Exemple de diagrammes UML

Relation entre les composants :

- **Association** entre deux classes : trait plein + cardinalité
 - une entreprise a plusieurs employés
 - un client commande un produit
 - **Cardinalité** :
 - * : plusieurs
 - N-M : entre N et M ($N < M$)
 - N : exactement N



Exemple de diagrammes UML

- **Agrégation** : trait plein + losange blanc du côté du conteneur
 - Un livre peut avoir une couverture
 - Lorsqu'on instancie le livre , on peut instancier la couverture au préalable.
 - Lorsqu'on détruit le livre, on détruit la couverture en cascade.



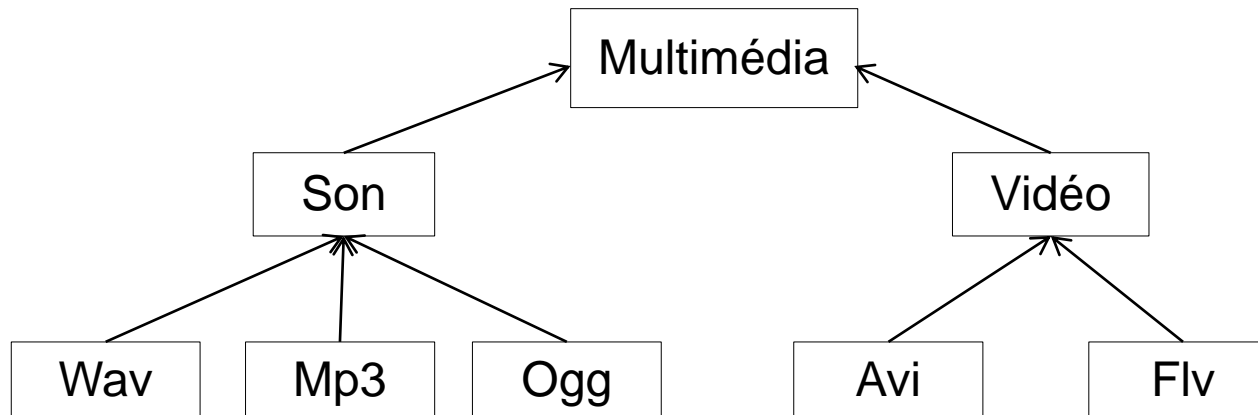
Exemple de diagrammes UML

- **Composition** : trait plein + losange du côté du conteneur
 - Une voiture contient un moteur
 - Lorsqu'on instancie la voiture, on instancie le moteur au préalable.
 - Lorsqu'on détruit la voiture, on détruit le moteur en cascade.



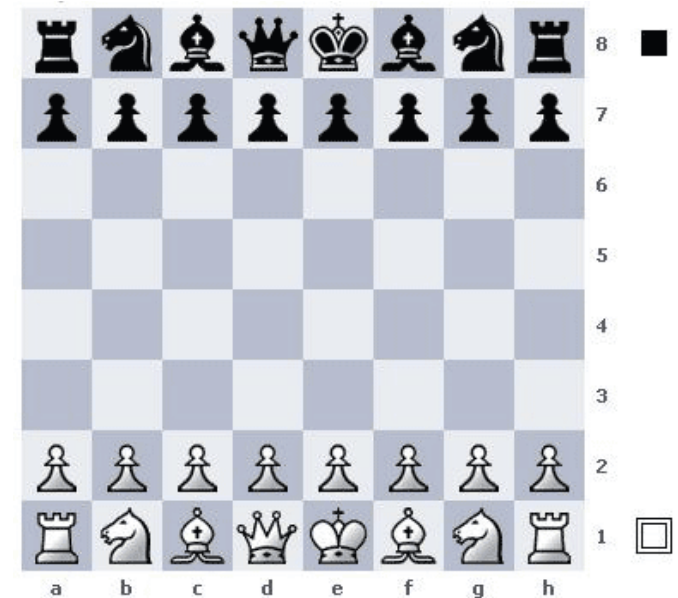
Exemple de diagrammes UML

- **Héritage** : trait + triangle du côté de la classe la plus générique
 - Une voiture est un type de véhicule
 - Un mp3 est un type de fichier son qui est un type de fichier multimédia.
 - Lorsqu'on instancie la voiture, on instancie le moteur au préalable.
 - Lorsqu'on détruit la voiture, on détruit le moteur en cascade.

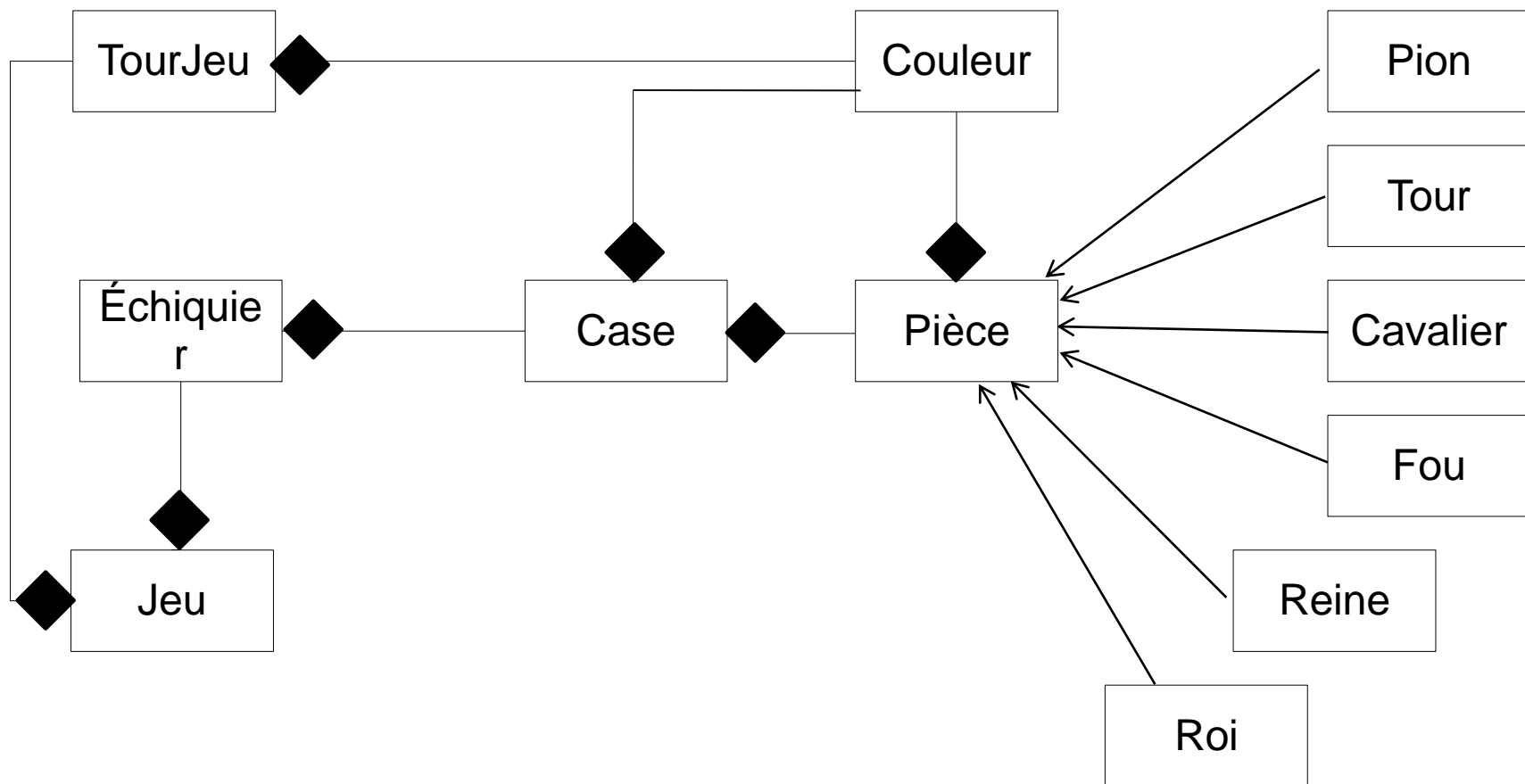


TP : UML

- Écrire un diagramme UML permettant de réaliser un jeu d'échecs.
- Les questions à se poser pour le réaliser :
 - Quels sont les classes nécessaires pour décrire un jeu d'échecs ?
 - Comment sont-elles liées ?
 - Comment piloter l'application (tour des blancs / des noirs, tester si le jeu est fini ou bloqué...) ?



TP : UML - corrigé

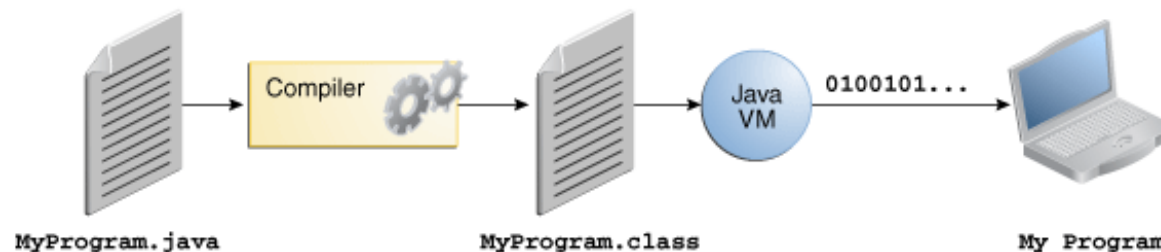


Partie 2 : premiers pas en Java

- Introduction
- Eclipse
- Hello world
- L'archiveur java : jar

Qu'est ce que java ?

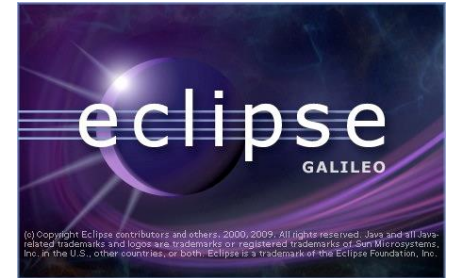
- Java est un langage de programmation orientée objet apparu en 1995. Il permet de développer des applications lourdes, des servlets, des applets...
- Le développeur écrit un **code source java (.java)**
- Il lance sa compilation en **bytecode (.class)**
- L'utilisateur exécute le bytecode via sa machine 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



Qu'est ce que java ?

- 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 Integret (IDE) tel que:
Eclipse, netbeans, ...

TP : Installation d'eclipse



- Principalement deux IDE : **eclipse** et **netbeans**.
- Sous windows (version actuelle : juno) :
 - http://www.siteduzero.com/tutoriel-3-10258-parlons-d-eclipse.html#ss_part_2
- Sous linux :
 - Via le gestionnaire de paquets. Par exemple sous debian, en root :

```
aptitude update
aptitude safe-upgrade
aptitude install eclipse
aptitude install openjdk-6-jdk # outil javadoc
```
- Traduction en français :
 - <http://babel.eclipse.org/babel/>

TP : Hello world

- Créer un nouveau projet
- Créer une nouvelle classe Hello.java dans nouveau package test/src

```
package test;

public class Hello{

    // Point d'entrée du programme
    public static void main( String [] argv ) {

        /* Pour le moment ces lignes font appel
        a plusieurs notions que nous n'avons pas
        encore vues... */

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

    }
}
```

- Compiler le programme, exécuter

Faire un fichier .jar

- À ce stade le programme peut être lancé depuis Eclipse mais seuls des « .class » ont été produits. On va les rassembler sous la forme d'une **archive java** (fichier « .jar ») pour redistribuer facilement le programme.
- Compiler un fichier « .jar » dans Eclipse (javac)
 - Clic droit sur le projet, corriger le profil de compilation à son idée
 - Clic droit sur le projet, export, java > runnable jar, next, choisir le profil associé au projet et le nom du « .jar ».

- Pour l'exécuter :

```
java -jar /le/chemin/vers/le/fichier.jar
```

- On peut passer des **arguments** au programme

```
java -jar fichier.jar argument1 argument2
```

TP : .jar

- Générer le .jar associé au programme Hello
 - L'exécuter dans un terminal

```
java -jar hello.jar
```

- Il doit afficher :

```
Hello world
```

Partie 3 : Les constructions de base du langage

- Variable
- Type de base
- Fonction et visibilité
- Structures algorithmiques
- Tableaux

Conventions java

- Les classes commençant par une majuscule, tout le reste (attributs, méthode, instances...) par une minuscule.
- On écrit un fichier .java par classe.
- Les méthodes s'écrivent suivant ce style : maPremiereMethode.
- Souvent, les noms des membres et des méthodes commencent par des minuscules.

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 leurs types

- 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.

Type	Taille (octets)	Précision (chiffres significatifs)	Valeur absolue minimale	Valeur absolue maximale
float	4	7	1.401e-045 (Float.MIN_VALUE)	3.40282e+038 (Float.MAX_VALUE)
double	8	15	2.22507e-308 (Double.MIN_VALUE)	1.79769e+308 (Double.MAX_VALUE)

Variables et types

- Le type caractère:
 - Comme la plus part des langages, Java permet de manipuler des caractères. Cependant, Java fournie 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 contantes 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 reçu 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); }
}
```

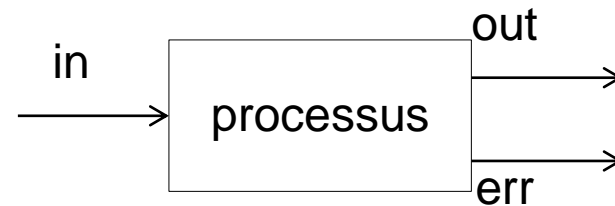
Variables et types

- Le mot-clé *final* permet de déclarer que la valeur d'une variable ne doit pas être modifiée:
 - `final int n = 20 ; // si on fait n = n + 3 on aura une erreur.`
- L'opérateur de cast:
 - Un programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'une opération nommée *cast*:
 - Exemple 1: x et y sont de type int, l'expression:
`(double) x/y ; // aura comme valeur résultante de type double`
 - Exemple 2:
`int n = 3;
byte b;
b= (byte) n;`

Flux standards

- Un processus peut manipuler trois flux :

- Un flux entrant
- Un flux de sortie "normal"
- Un flux d'erreur



- Par défaut ces flux ne sont autres que les flux standards

- l'entrée standard (System.in) : lecture au clavier dans la console
- la sortie standard (System.out) : écrit dans la console
- la sortie d'erreur standard (System.err) : écrit dans la console

- ... mais il peuvent être redirigés vers un fichier (>, >>, ...) ou vers un processus (pipe).

- En plus des flux standards on peut lire / écrire des fichiers etc...

Opérateurs

- Opérations sur des valeurs numériques (**int**, **float**, **double**...)
 - `+`, `*`, `-`, `/`, `%` (modulo)
 - **Attention** : `/` sur des `int` revient à faire une division euclidienne
 - Cast (conversion) : `q = (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.

Lire une valeur

- Le but est ici de permettre une saisie clavier. On utilise une classe java (la classe Scanner). On abordera ultérieurement les classes. Java fournit de nombreuses classes notamment pour faire une GUI.
- Afficher la console : *window > show view > console*

- Nous utiliseront une classe de l'api java :

```
import java.util.Scanner
```

- Pour l'utilisation il va falloir utiliser l'entrée standard : **System.in.**

C'est un paramètre à indiquer lors de l'instanciation de l'objet Scanner

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

Lire une valeur

- Une fois instancié, l'objet nous fournis une liste de méthode, destiné à renvoyer un type de saisie.

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();  
double d = sc.nextDouble();  
long l = sc.nextLong();  
String b = sc.nextLine();  
  
//etc
```

Lire une valeur

- Le but est ici de permettre une saisie clavier. On utilise une classe java (la classe Scanner). On abordera ultérieurement les classes. Java fournit de nombreuses classes notamment pour faire une GUI.
- Afficher la console : *window > show view > console*

```
package test;

import java.util.Scanner;

public class Hello {
    public static void main(String [] argv) {
        // Un scanner lit l'entrée standard
        Scanner sc = new Scanner(System.in);
        String prenom = sc.nextLine();
        System.out.println("Bonjour " + prenom);
    }
}
```

int : nextInt()
...

```
toto
Bonjour toto
```

TP : opérateurs

- Lire deux entiers x et y .
- Calculer et afficher leur somme, leur différence, leur produit, le quotient de x/y , le reste de x/y et les afficher.

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 :
 - $(0 < t) < 100$
 - $true < 100$

TP: conditions

- Ecrire un algorithme qui demande deux nombres en parametre et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul).

Attention toutefois : on ne doit pas calculer le produit des deux nombres.

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 algorithme qui demande un choix à l'utilisateur. Ensuite, il l'informe d'un message :
 - 1) Ouvrir
 - 2) Quitter
 - 3) Sauver

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;  
    }  
}
```

TP : boucles

- Demander un entier positif n .
- Dessiner une ligne d'étoile, un carré d'étoiles et un triangle d'étoiles.
 - Exemple $n = 5$:

```
* * * * *
```

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

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

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

Énumération

- Pour un jeu de constantes on peut utiliser des énumérations
 - Chaque élément est initialisé par défaut respectivement à 0, 1, 2...
 - On peut spécifier la valeur de chaque élément.
 - Par convention, on écrit généralement chaque constante en majuscules.
 - Ces constantes peuvent être vues comme des entiers.

```
public enum Couleur{ NOIR, ROUGE, VERT, BLEU }  
  
// Utilisation :  
// Couleur c = Couleur.ROUGE;
```

- La notion d'énumération est étendue depuis Java Tiger (1.6)
 - Voir cours sur les classes

Tableaux

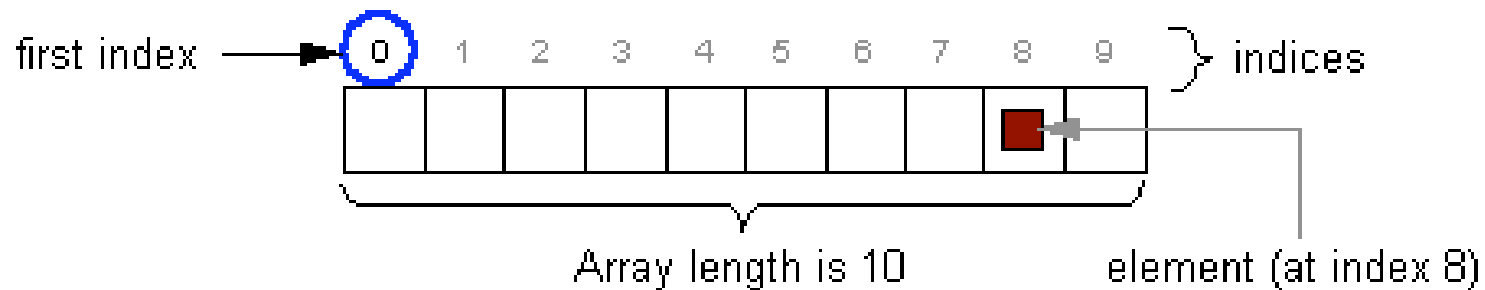
- Pour manipuler des ensembles d'un type donné nous n'avons pas forcément besoin de coder une classe personnalisée.
- Un tableau est une structure qui comporte plusieurs éléments de même type
- Déclaration de tableaux :
 - `String[] tab` : un tableau de String ;
 - `int[][] mat` : matrice 2D
 - `String tab []` ;
- Initialisation de tableaux :
 - `String tableauChaine[] = {"chaine1", "chaine2"};`
- **Création de tableaux:**
 - C'est ce qui est utilisé pour récupérer les arguments passés au programme.
 - `int[][] mat = new int[3][4];`

Tableaux

■ Utilisation/parcours de tableaux:

- On désigne une case particulière d'un tableau en précisant son emplacement à l'intérieur des crochets
- Exemple: `tab[1] = "hello" ;`
- Afin de parcourir un tableau on utilise une structure de boucle, tel que la boucle `for` .

■ Représentation d'un tableau:



Tableaux

- Pour manipuler des ensembles d'un type donné on n'a pas forcément besoin de coder une classe personnalisée.
- Utilisation de tableaux :
 - `String[]` : un tableau de `String` ; `int[][]` : matrice 2D
 - C'est ce qui est utilisé pour récupérer les arguments passés au programme.
- Java fournit également des containers souvent plus pratiques :
 - non typés : `ArrayList`, `HashTable`, `HashMap`
 - typés : `ArrayList<T>`, `HashTable<T>`, `HashMap<K,T>`
 - On les verra ultérieurement car il nous manque des concepts liés aux objets...

TP : tableau

- Écrire un programme qui affiche chacun de ses arguments (un par ligne) précédé de son numéro d'argument. Exemple :

```
java -jar fichier.jar toto titi tata
```

- ... doit afficher :

```
1:toto  
2:titi  
3:tata
```

Partie 4 : Programmation orienté objet

- Classes
- Constructeur
- Transmission de paramètre
- Héritage
- Polymorphisme

Classe = capsule pour un concept

- Java fournit des classes pour les **concepts** "évolués" (String, Scanner...) au travers de packages
 - `import java.util.Scanner;`
- Une classe ne doit encapsuler que ce qui suffit à représenter le concept qu'elle modélise.

```
package test;

public class Personne{
    private String prenom;
    private String nom;
    //...
}
```

```
package test;

public class Voiture{
    private Moteur moteur;
}
```

- Sa structure découle directement du diagramme UML.



Structure des fonctions

- On appelle *fonction* un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel de la fonction dans le **corps** du programme principal.
- Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale.
- D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).

Méthode = action

- On rattache à une classe un ensemble de fonctionnalités, appelées méthodes.
- Une **méthode**:
 - est spécifique à l'objet dans laquelle on la définit et peut être héritée,
 - peut accéder aux données stockées dans l'instance à laquelle on l'applique,
 - peut retourner une valeur.
- La définition d'une méthode ressemble à celle d'une procédure ou d'une fonction dans les autres langages. Elle se compose d'un en-tête et d'un bloc d'instruction.

```
Porté type_de_donnee NomFonction(type argument ...n) {  
    liste d'instructions  
    [valeur de retour]  
}
```

Méthode = action

- Quand la **méthode** retourne une valeur (elle a donc un type de retour), on utilise le mot clé **return**.
- Si la méthode ne fournit aucun résultats, le mot-clé *void* figure à la place du type de la valeur de retour. Dans ce cas on n'utilise pas le mot-clé **return**.
- Dans une méthode, **this** désigne l'instance courante.

```
public class Personne {  
    private String nom;  
    private String prenom;  
  
    public void affiche() {  
        System.out.println("Mon prénom est : "+this.prenom );  
    }  
  
    // ...  
}
```

Méthode = action

- Une méthode à ellipses (introduit depuis la JDK 1.5) correspond à une fonction avec un nombre de paramètres variable:

```
public class X {  
    public void maFonction(int... e) {  
        . . .  
    }  
    //...  
}  
  
int i1, i2, i3;  
maVar = new X();  
maVar.maFonction();  
maVar.maFonction(i1);  
maVar.maFonction(i2, i3);  
//...
```


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 information (nom et la localité) d'une société.
- Instancier deux société et afficher les informations des deux sociétés.

Passage par référence et par copie

- Le passage de paramètre à une méthode peut être fait
 - **par copie** : la méthode manipule une copie de la variable originelle
 - **par référence** : la méthode reçoit « l'adresse mémoire » de la variable originelle.
 - Une référence est toujours initialisée et « pointe » sur un objet
 - ... ou vaut **null** (rien n'est pointé).
- Java fait toujours un passage par référence, sauf pour :
 - les **types de bases** (écrits en minuscules par convention)
 - **int**, **double**, **char** ...
 - les classes **non mutables** (non modifiables)
 - `String`, `Integer`, ...
- Dans ce cas, Java fait une recopie implicite

Passage par recopie

- Exemple :

```
public static void f(int x) {  
    ++x; // 8  
}  
  
public static void g() {  
    int x = 7;  
    f(x);  
    System.out.println(x); // affiche 7  
}
```

- Ici la fonction f manipule une **recopie** de la variable x définie dans g.
 - f modifie cette recopie.
 - Du coup la variable x manipulée par g n'est **pas** impactée.
- Si x était de type String (non mutable), x n'aurait également été modifiée qu'au sein de f, mais pas de g !

Passage par référence

- En java les passages de paramètres avec des classes mutables se font **par référence**.
 - Une référence peut être vue comme une adresse mémoire (à ceci prêt qu'elle est toujours initialisée).
 - Une référence peut être initialisée à **null** (ne pointe sur aucun objet).
 - Dans ce cas, il ne faut pas lui appliquer de méthode !
 - Si un objet n'est plus référencé il est détruit (**garbage collector**)

```
public static void f(Personne e) {  
    e.setNom("Alizée");  
}  
  
public static void g() {  
    Personne e = new Personne();  
    f(e);  
    System.out.println(e.getNom()); // Affiche Alizée  
}
```


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;  
    }  
}
```

Exercice (Société)

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

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.

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

- 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.

Personne
- String m_sPrenom - String m_sNom
+ Personne(m_sPrenom,m_sNom) + afficher()

Constructeurs : en java

- **new** invoque un constructeur.
- **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");
```

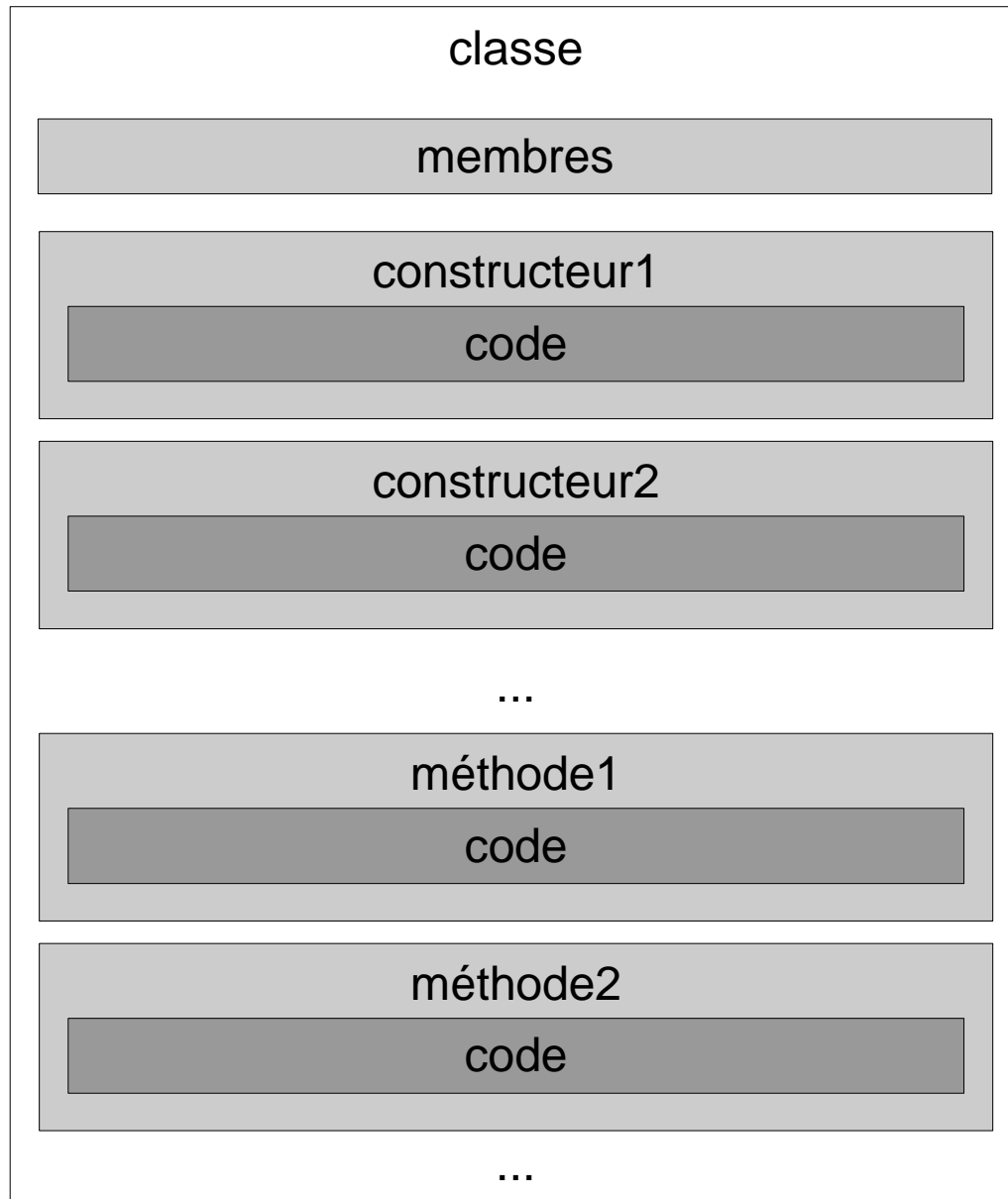
TP: Société

- Instancier la classe société avec un constructeur avec paramètres.

Résumé (1 / 2)

- Une **classe** est une généralisation d'un **type**. Elle encapsule différents **membres** ou **attributs**.
 - Exemple : un Chien (encapsule nomChien, raceChien...)
- Un **objet** ou une **instance** est une généralisation d'une **variable**.
 - Exemple : le chien Médor
- Un **constructeur** sert à initialiser un objet lorsqu'on l'instancie
 - Appel : **new**, **this**
 - Respectivement : un constructeur d'une classe, de cette classe, de la classe mère
- Une **méthode** est une "**fonction**" associée à une classe.
 - Exemples : rongerOs(Os IOsARonger), aboyer(), getNom() ...
 - **this** : accès sur les attributs / méthodes de l'objet.
 - Si **this** est inutile, la méthode est **static**.
 - Les algorithmes sont toujours implémentés dans des méthodes.

Résumé (2/2) : structure



```
public class Toto{  
    int x;  
    //...  
  
    public Toto(int x0) {  
        this.x = x0;  
    }  
  
    public Toto() {  
        this(0); // Toto(int)  
    }  
    //...  
  
    public int getX() {  
        return this.x ;  
    }  
  
    public void setX(int x0) {  
        this.x = x0;  
    }  
  
    //...  
}
```

TP: 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 le main la création de 2 sociétés. Ces dernières seront affecté par une même adresse.
- Afficher les informations sur les sociétés puis modifier l'adresse. Réafficher les informations et observez le résultat.

Exercice (École)

- Nous voulons gérer une inscription d'élèves dans une école.
- Notre école gère un ensemble de cours. Un cours est caractérisé par:
 - un nom , un nombre d'heures et une année
 - Exemple: nom : Math, nombre d'heure : 10 et l'année : 2
- Lors de l'inscription d'un élève nous devons renseigner tous ces caractéristique:
 - un nom, prénom, âge et l'année du cours qu'il suit
- Une fois l'élève inscrit, nous devons l'affecter à des cours. Sachant qu'il existe plusieurs cours par année, il ne sera possible d'affecter l'élève que dans les cours correspondant à son année lors de son inscription.

Exercice (École)

- Aide pour l'exercice:
 - Nous aurons 3 classes: Élève, Cours et École
 - Dans la classe École nous stockeront les élèves et les cours dans deux listes (listeEleve et listeCours)
- Le but de l'exercice étant d'afficher tous les cours avec la liste des élèves inscrit pour chaque cours.
- Construire au préalable un diagramme de classe.

Destructeur

- La plupart des langages permettent d'implémenter un **destructeur** qui est invoqué quand l'on veut détruire l'objet.
 - Il sert essentiellement à dés allouer tout ce que le constructeur a alloué.
 - Peut décrémenter un compteur d'instance.
- Les destructeurs n'existent pas en java.
 - Le **garbage collector** (ramasse miettes) gère la mémoire.
 - Si du code doit être déclenché au moment où l'objet est libéré, on l'implémente dans la méthode `finalize`.

```
protected void finalize() {  
    System.out.println("détruit !");  
    super.finalize();  
}
```

- Si des exceptions peuvent être levées, il faut les rattraper et mettre `super.finalize()` dans un bloc **finally**.

Les paquets

- La notion de paquetage correspond à un regroupement logique d'un ensemble de classes sous un identificateur commun.
- **Avantage:**
 - Facilite le développement et la cohabitation de logiciels en permettant de répartir les classes correspondantes dans différents paquets.
 - Limite le risque de créer deux classes de même nom dans le même paquetage.
- Un paquetage est caractérisé par un nom qui est soit un simple identificateur, soit une suite d'identificateurs séparés par des points.
 - **Exemples:**
 - MesClasses
 - MonPackage.Mathematiques
 - MonPackage.Jeux

Les paquetages

- L'attribution d'un nom de paquetage se fait au niveau du fichier source; toutes les classes d'un même fichier source appartiendront donc toujours à un même paquetage.
- Pour le spécifier on place au début du fichier l'instruction suivante :
 - `package XXXX ; //` où XXXX correspond au nom du paquetage
- Utilisation des paquetages:
 - Quand un programme fait référence à une classe le compilateur la recherche dans le paquetage par défaut.
 - Pour utiliser une classe appartenant à un autre paquetage, on doit fournir l'information correspondante au compilateur soit en :
 - Citant le nom du paquetage avec le nom de la classe
 - Utilisant une instruction `import` en y citant
 - soit une classe particulière
 - soit le nom d'un paquetage suivi d'une étoile.

Les paquetages

- Exemple:
 1. En citant le nom de la classe:

```
MesClasses.Personne p = new MesClasses.Personne("Rémy", "Truc");  
p.getNom();
```

2. En important une classe:

```
import MesClasses.Personne, MesClasses.Eleve;  
Personne p = new Personne("Lucie", "Muche");
```

```
import MesClasses.*;  
Personne p = new Personne("Charlotte", "Muche");
```

Les paquets

- Remarques:

- Il existe un paquetage particulier nommé *java.lang* qui est importé automatiquement par le compilateur. C'est lui qui nous permet d'utiliser des classes standard telles que *Math*, *Float* ou *Integer*, sans avoir à introduire l'instruction *import* .
- Pour pouvoir utiliser une classe il faut qu'on ait les droits d'accès à cette classe. C'est-à-dire qu'elle doit être déclarée *public*.
- Il est possible d'utiliser des méthodes statiques des classes importées de la sorte:

```
import static java.lang.Math.PI;  
Double d = PI * 2 ;
```

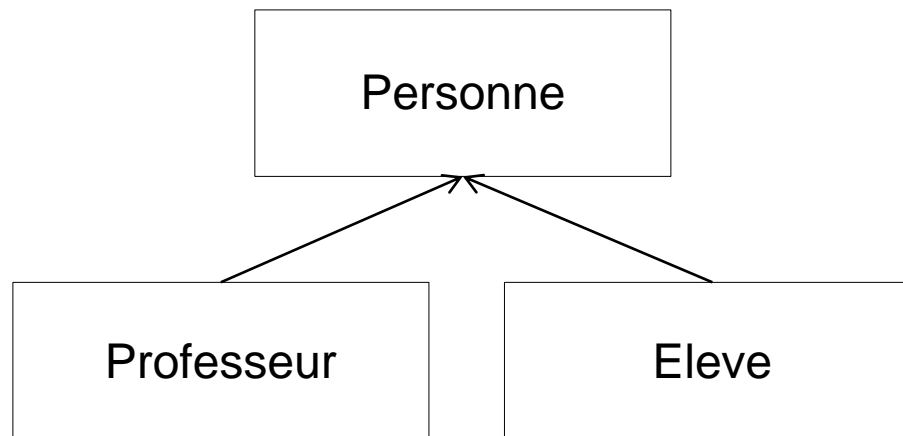
Héritage : réutilisation

- Certaines classes sont des **spécialisations** d'autres classes.
- Plusieurs classes "proches" d'un point de vue conceptuel (par exemple Élève et Professeur) ont souvent des points communs
 - Un Élève et un Professeur ont un prénom et un nom (spécialisation d'une Personne).
 - Ces points communs se retrouvent niveau des attributs, des méthodes, des constructeurs etc...
 - En java toutes les classes héritent de la classe Object.
- Grâce à un **héritage** ces points communs ne seront codés qu'une fois
 - Gestion du nom et du prénom dans la classe Personne
 - Gestion des devoirs dans la classe Eleve
 - Une méthode / un attribut doit être codé(e) dans la classe la plus générique possible et commune à toutes ses classes filles

L'héritage

- En Java il n'y a qu'un type d'héritage : **l'héritage public**
 - **extends** (pour une **class**)
 - **implements** (pour une **interface** vues par la suite)

```
public class Eleve extends Personne { /* code de la classe */ }
```



Héritage : visibilité surcharge

- Héritage public : le seul qui existe en Java
 - `public` → `public`
 - `protected` → `protected`
 - `private` → pas propagé vers la classe fille
- Les constructeurs, méthodes et membres `public` ou `protected` sont propagés aux classes filles
 - Pas besoin de les recoder !
 - Code `factorisé` = pas de redondances : plus simple, plus compact, plus facile à maintenir, mieux organisé !
 - ... mais on peut le faire : dans ce cas on parle de `redéfinition`.
 - Quand il a le choix, java prend la méthode la plus spécifique à l'objet (`lien dynamique`)

Héritage : Exemple



```
public class Personne {
    private String m_sPrenom;
    private String m_sNom;

    public Personne(String sPrenom, String sNom) {
        this.m_sPrenom = sPrenom;
        this.m_sNom = sNom;
    }
    public void setPrenom(String sPrenom) {
        this.m_sPrenom = sPrenom;
    }
    public String getPrenom() {
        return this.m_sPrenom;
    }
    //...
}
```

Héritage : Exemple

- **New** : invoque le constructeur
- **Super**: permet d'invoquer un constructeur de la classe mère
- **This**: désigne l'instance courante (dans une con

```
public class Eleve extends Personne {  
    private String m_sClasse; // spécifique à un Eleve  
  
    public Eleve(String sPrenom, String sNom, String sClasse){  
        super(sPrenom, sNom); // invoque Personne(sPrenom, sNom);  
        this.m_sClasse = sClasse;  
    }  
}
```

```
Eleve amandine = new Eleve("Amandine", "Dupuis", "1ère S 3");
```

Polymorphisme

- Il est possible de ré implémenter (**redéfinition**) une méthode existante dans une classe Mère dans une classe Fille.
 - Exemple : tester le déplacement d'une pièce aux échecs.
- En C++ il est possible de redéfinir (et surcharger) les opérateurs en plus des méthodes, mais pas en Java.
- Si l'on instancie une classe Fille mais que celle-ci est stockée dans une variable de type Mere... quelle méthode appeler ?
 - **Lien dynamique** : le type sous jacent est Fille... donc la méthode de la classe fille. En java le lien dynamique est fait systématiquement.
 - Si le lien dynamique n'est pas établi, la méthode de la classe Mère (comportement par défaut en C++, il faut utiliser le mot clé virtual pour l'établir).

Types énumérés (Tiger 1.5)

- En réalité, une énumération peut, comme une classe, comporter plusieurs membres, des méthodes, des constructeurs (type énuméré)

```
public enum Numbers {  
    ONE    ("un",    1),  
    TWO    ("deux",  2);  
  
    private String description;  
    private int value;  
  
    Numbers(String description, int value) {  
        this.description = description;  
        this.value = value;  
    }  
  
    public String getDescription() {  
        return this.description;  
    }  
  
    public int getValue() {  
        return this.value;  
    }  
}
```

Le mot clé static

- Le mot clé **static** peut s'utiliser dans trois contextes
 - Pour une **variable locale**
 - Toujours déclarée et initialisée dans la méthode qui l'héberge.
 - Visible uniquement par cette méthode.
 - Sa valeur est mémorisée entre deux appels.

```
public void f() {  
    static int nbPassageDansF = 10 ; // init° , decl°  
}
```

- Pour un **membre de classe** commun à toutes les instances de la classe (constantes, compteur d'instance...)
 - Accessibles par les méthodes statiques ou non

```
class Personne {  
    public static int nbPersonnes = 0; // init° , decl°  
    public Personne() {  
        ++Personne.nbPersonnes; // utilisation  
    }  
}
```

Le mot clé static

■ Pour une méthode

- Si **this** est inutile → la méthode *devrait* être statique.
- Méthode statique → **this** interdit.
 - Elle peut accéder à des membres ou méthodes statiques.
 - Elle ne peut pas accéder aux autres membres et méthodes, qui dépendent de **this**.
- Pour signifier que la méthode est indépendante de **this**, elle ne s'applique pas à une instance mais à la **classe** en général.

```
class Math {  
    public static double abs(double d) {  
        return d < 0 ? -d : d;  
    }  
}
```

Ne dépend
pas de **this**

```
class Personne {  
    //...  
    public static int getNbPersonnes() {  
        return Personne.nbPersonnes;  
    }  
}
```

Ne dépend
pas de **this**

Fonctions statiques

- Une fonction permet de **mutualiser** une section de code.
- Ce comportement peut dépendre d'un ou plusieurs **paramètres** (typés).
 - Une fonction ne dépend pas d'une instance (notion vue plus tard), uniquement de ses paramètres. On parle de méthode **statique**.
- La fonction peut :
 - Retourner un résultat typé : la fonction se quitte lors d'un **return**
 - Ne rien retourner : la fonction est quittée lors d'un **return** ; ou lorsque l'accolade « } » de la fonction est rencontrée.
 - Dès qu'un **return** est rencontré, la fonction est immédiatement quittée.

```
public static double valeurAbsolue(double x) {  
    if (x < 0) return -x ;  
    return x;  
}  
public static void direBonjour() {  
    System.out.println("Bonjour");  
}
```

Sous-entendus Java

- Une méthode / un membre **statique** s'applique toujours à une **classe**
 - Si la classe est sous-entendue, la classe utilisée est la classe dans laquelle on est en train de coder.

```
class Main {  
    public static void f() {}  
  
    public static void main(String [] args) {  
        f();  
    }  
}
```

Implicitement
Main.f();

- Une méthode / un membre **non statique** s'applique toujours à une **instance**
 - Si l'instance est sous-entendue, l'instance implicitement utilisée est **this**

```
class Personne {  
    public String prenom;  
    public String getPrenom() { return prenom; }  
}
```

Implicitement
this.prenom

Le mot clé final

- Le mot clé **final** verrouille en lecture seule une référence
 - L'objet pointé par la référence reste modifiable
 - Contrairement au C++ qui utilise le mot clé **const**, on ne peut pas verrouiller en lecture seul l'objet référencé.

```
void maMethode(final Personne p) {  
    p.setPrenom("Sophie");    // ok car la référence ne change pas  
    p = new Personne("Marc");  // pas ok, la référence change  
}
```

- On utilise généralement le mot clé **final** lorsqu'on (ici PI) manipule une constante.

```
class Mathematique {  
    public final static double PI = 3.14;  
}
```

Partie 5 : objets - notions avancées

- Interfaces
- Lien dynamique
- Les grandes règles de design
- Containers
- Classes génériques

Classes : garantir un comportement

- Les méthodes sont dites **abstraites** (= **virtuelles pures**) quand elles sont non implémentées. On appelle **classe abstraite** une classe comportant au moins une méthode abstraite.
- En java, lorsque toutes les méthodes sont abstraites, on n'utilise pas une classe (**class**) mais une interface (**interface**).
 - Pas instanciables.
 - Servent à contraindre le design de(s),classe(s) fille.
- Héritage
 - **implements** : si on hérite d'une interface (autant qu'on veut)
 - **extends** : si on hérite d'une classe mère (au plus 1)
 - On peut hériter d'au plus une classe et de zéro ou plusieurs interfaces.
 - **class** Eleve **extends** Personne { ... }
 - **class** Fille **implements** I1, I2 **extends** Personne { ... }

Interfaces:

- Une interface correspond à une classe abstraite n'implémentant aucune méthode et aucun attribut (hormis des constantes).
- Exemple :

```
public interface I{  
    void f(int t);    // en-tête d'une méthode f (public abstract  
                    // est facultatif )  
    void g()  
}
```

Containers

- Java fournit des containers autres que les tableaux
- Principaux containers
 - LinkedList : référence sur le maillon suivant (**null** sinon)
 - ArrayList : liste dans un tableau
 - HashSet : ensemble d'éléments uniques
 - HashTable : fait partie des Maps (associe une clé à une valeur)
 - <http://www.siteduzero.com/tutoriel-3-10409-les-collections-d-objets.html>
- Avantages / inconvénients :
 - Taille dynamique
 - Possibilité d'itérer dessus : avec un index ; un objet Iterator ; un objet Enumeration (selon le container).
 - Pas de contrainte sur les éléments dans le container... comment faire ?

Classes génériques

- Une **classe générique** est une classe **paramétrée** par une ou plusieurs autres classes
 - Syntaxe générique : `MaClasse<T1, T2, T3, ...>`
 - Une liste d'entiers : `List<Integer>`
 - Un ensemble de mot : `Set<String>`
 - Un annuaire : `Map<Personne, NumeroTelephone>`
- Évite de coder `ListeEntier`, `ListeString`
 - Les méthodes d'une classe génériques doivent **s'affranchir du(des) type(s) passés** en paramètre...
 - En général une classe générique sert de container.
 - **Attention** : `Liste<Fille>` n'hérite pas de `Liste<Mere>`

Classes génériques

- Si un algorithme ne dépend pas directement du type du contenu, on peut le coder pour importe quel contenu :
 - Chercher un élément dans une liste générique
 - Trouver un plus court chemin dans un graphe générique
- Les classes génériques permettent de **factoriser** le code
 - Gain de temps
 - Maintenance plus aisée
 - Design plus contraint donc plus rigoureux / sûr
 - On n'est jamais sûr de trouver l'objet que l'on attend dans une Collection.

Types génériques

- Une méthode peut manipuler un ou plusieurs paramètres / variables génériques.
- On peut contraindre la nature de ces types génériques.

```
// list1 doit contenir des instances de Voiture (ou des
// instances de classes qui héritent de Voiture tel que
// Monospace, Cabriolet...)
List<Voiture> list1;

// list2 doit contenir des instances de Voiture ou
// d'une classe fille de Voiture (Cabriolet, Monospace...)
List<? extends Voiture> list2;

// list3 doit contenir des instances de classe mère
// de Voiture (Véhicule...)
List<? super Voiture> list3;

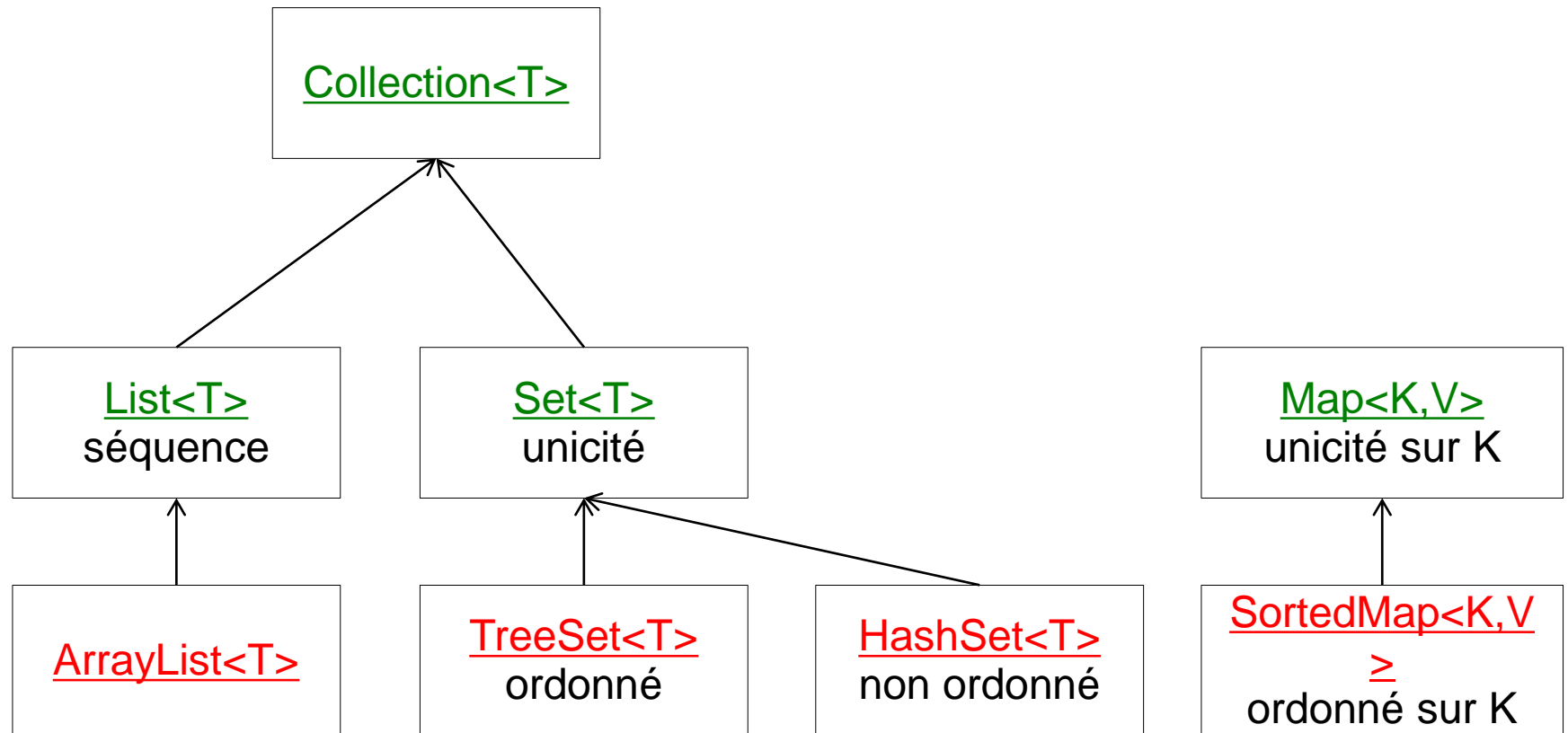
// list4 peut contenir n'importe quoi
List<?> list4;
```

- **Attention** : utiliser '?' verrouille le paramètre en lecture seule.

Containers génériques

- Java fournit depuis le JDK1.5 des containers génériques :
 - Interface (type manipulé dans un prototype) :
 - List<T>, Set<T>, Map<K,V>
 - Permet de s'affranchir de l'implémentation.
 - Implémentation (type instancié) : HashSet<T>, ArrayList<T>, ...
- Avantages :
 - Algorithmes génériques (contains, ...)
 - Les itérations sont plus simples à mettre en place (nous les verrons bientôt) :
 - Avant Tiger : Iterator + Enumeration
 - Depuis Tiger : à l'aide de l'opérateur :

Quelques containers classiques



<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html>

L'autoboxing

- Une classe générique ne peut prendre en paramètre qu'une classe pas un type primitif. Avant Tiger ceci ne compilait pas :

```
List<Integer> list = new ArrayList<Integer>();  
int myInt = 100; // int est un type primitif !  
list.add(myInt); // erreur (java.lang.ClassCastException)
```

- Il fallait écrire :

```
List<Integer> list = new ArrayList<Integer>();  
int myInt = 100;  
list.add(new Integer(myInt));
```

- Depuis Tiger et en vue de rendre le programme moins lourd à écrire et plus lisible, cette conversion n'est plus nécessaire.
 - Conversions implicites : **int** → Integer, **byte** → Byte, **double** → Double, **short** → Short, **int** → Integer, **long** → Long, **float** → Float, **char** → Character

Itérations : accès RW

- Pour itérer sur un container on peut utiliser la classe Iterator
 - Modifier le contenu d'un container peut invalider ses Iterators

```
List<String> listePrenoms = new ArrayList<String>();  
listePrenoms.add("Amandine");  
listePrenoms.add("Alizée");  
  
for(Iterator<String> it = listePrenoms.iterator(); it.hasNext();  
) {  
    String prenom = (String)it.next();  
    System.out.println(prenom);  
}
```

- Pour les maps on peut reproduire cette approche via la méthode `entrySet()`.
- Pour les HashMap on peut itérer grâce à une Enumération (voir méthode `éléments()`)

Itérations : accès R0

- Depuis Tiger on peut utiliser l'opérateur ":"
 - Itération en **lecture seule** (faute d'Iterator)

```
List<String> listPrenoms = new ArrayList<String>();  
listPrenoms.add("Amandine");  
listPrenoms.add("Alizée");  
  
for(String prenom : listPrenoms) {  
    System.out.println(prenom);  
}
```

- **Attention** : pour supprimer un élément on est obligé de passer par un Iterator

TP : Les listes

- Rechercher l'élément maximal d'une liste :
 - Écrire un algorithme qui permet de déterminer la valeur maximale d'une liste d'entiers positifs.
- Permuter deux places d'une liste :
 - Écrire un algorithme sur une liste qui échange les positions des nœuds données par deux index t et v .

Pour aller plus loin avec Tiger

- En plus des **classes génériques**, des types énumérés et de **l'autoboxing**, Tiger apporte de nombreuses nouveautés
 - Les **nouvelles API**
 - API de la concurrence : sémaphore, verrous, variables atomiques...
 - API de management et de supervision de la JVM
 - Les **annotations** (vues partiellement en fin de cours)
 - Contraintes avancées sur les paramètres des classes génériques (opérateur &)
- <http://lroux.developpez.com/article/java/tiger/?page=sommaire>

Partie 6 : exceptions

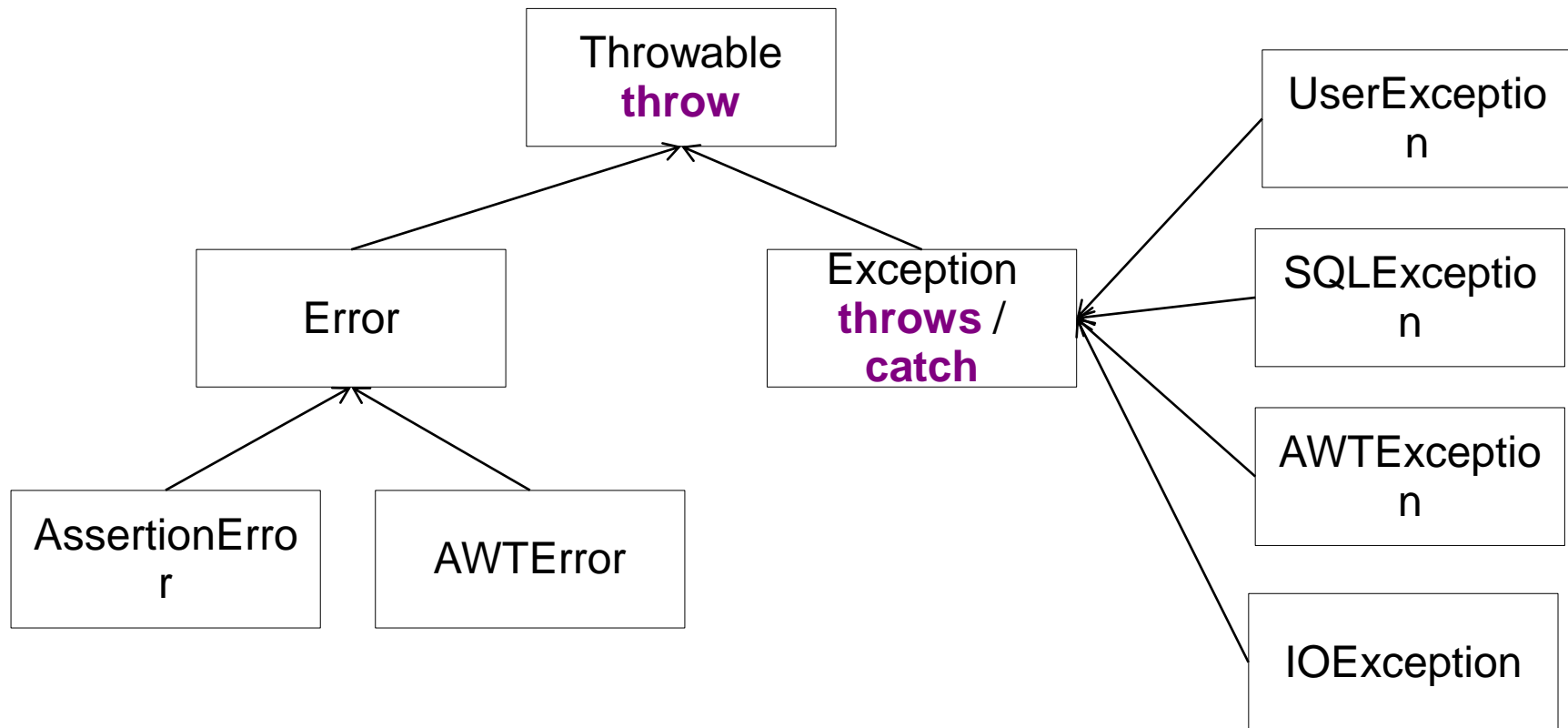
- Principe
- Syntaxe
- Créer ses propres exceptions
- Erreurs

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;}  
}
```

Les grandes règles de design

■ Règle 1 : sémantique cohérente

- Les prototypes transcrivent le langage naturel
 - Le fou peut-il en partant de cette case arriver sur cette case ?
 - `bool estMouvementValide(Case cDep, Case cArr)`
- Instance inutile → membre ou méthode statique.
- Les exceptions rattrapent les comportements invalides
 - `Case getCas(int i, int j) throws CaseInvalidException`

■ Règle 2 : intégrité

- Implique une visibilité aussi restrictive que possible.

■ Règle 3 : factorisation

- Éviter du code dupliqué (maintenance et le debuggage)
- Rend le code plus facilement évolutif

Les grandes règles de design

- Règle 4 : code **hiérarchique** et **structuré**
 - Contrôle des paramètres (on ne sait pas s'ils sont cohérents quand ils arrivent dans une méthode!)
 - On raisonne par **niveau d'abstraction** :
 - Chaque classe résout les problématiques qui lui sont propres.
 - Portée d'une variable : son **horizon** { ... }
 - Hiérarchie entre les blocs algorithmiques d'une méthode
 - Toujours supposer que le code devra être **généralisé** : son design doit être évolutif.
- Règle 5 : **performance**
 - Garantir une bonne complexité ($O(1) < O(\log(n)) < O(n) < \dots$)
 - Éviter de recalculer plusieurs fois un même résultat

Les grandes règles de design

- Règle 6 : apporter des **garanties** de code
 - Imposer des contraintes aux développeurs
 - Méthode abstraites : **abstract**
 - Force un développeur à ré implémenter certaines méthodes
- Règle 7 : **cloisonnement**
 - Changer une « considération » ne doit pas casser le code
 - Accesseurs, ...
 - Exemple : l'interface utilisateur (texte ? Graphique?) ne doit pas être mêlée au code qui implémente les règle du jeu d'échecs
- Les **design patterns** sont des manières de décrire des règles de design dans un contexte particuliers et servent à valider une ou plusieurs de ces règles.

Partie 7 : manipuler des fichiers XML

- Les flux appliqués aux fichiers
- SAX, DOM
- Exemple
- Javadoc
- Balises javadoc

La gestion des fichiers

- La Classe File:
 - Cette classe offre des fonctionnalités de gestion de fichiers comparables à celles auxquelles on accède par le biais de commandes système de l'environnement.
 - Parmi ses fonctionnalités:
 - Création/suppression, renommage de fichier ou de répertoire
 - Tester l'existence d'un fichier ou d'un répertoire
 - Lister les noms de fichiers d'un répertoire
 - ...
- Création d'un objet de type File:

```
File monFichier = new File('nomDuFichier.txt');  
// Attention monFichier est un objet créé en mémoire à ne pas confondre  
avec la création du fichier correspondant.  
  
// Pour créer notre fichier on peut faire comme suit:  
boolean ok = monFichier.createNewFile();  
  
//ok = true si la création a eu lieu.  
// si le fichier existe déjà, la création n'aura pas lieu
```


La gestion des fichiers

- On peut créer un fichier/répertoire en précisant son chemin comme argument au constructeur de la classe `File`

```
File monRep1 = new File('c:\\java\\formation\\exemple');  
// Nom de répertoire absolu sous Windows  
File monRep2 = new File('/home/lina/java/test');  
// Nom de répertoire absolu sous Unix
```

- Si vous voulez optimiser la portabilité de votre code, il vaut mieux utiliser : `File.separator`

```
String s = File.separator;  
File monRep3 = new File('java'+ s +'essais')
```

La gestion des fichiers

Type	Méthode	Descriptif
Boolean	<code>createNewFile()</code>	Créer un nouveau fichier
Boolean	<code>Delete()</code>	Supprime le fichier
Boolean	<code>Mkdir()</code>	Créer un répertoire
Boolean	<code>Exists()</code>	Vérifie si le fichier existe
Boolean	<code>isFile()</code>	Vérifie si l'objet est un fichier
Boolean	<code>isDirectory()</code>	Vérifie si l'objet est un répertoire
Long	<code>Lenth()</code>	Fournie la taille du fichier
Boolean	<code>canRead()</code>	Fournie <i>true</i> si l'objet correspond à un fichier en lecture
...

Les fichiers

- Au sens linux, un **fichier** peut être
 - un **fichier régulier** : un fichier qui **stocke** de la donnée
 - un répertoire, un device, un socket...
- Un fichier régulier peut-être
 - un fichier **texte**
 - texte brut : .txt
 - texte tabulé/séparé : .csv, .tsv
 - **.xml**
 - Cas particulier : .xhtml
 - un fichier **binaire** (suite de 0 et de 1 qui affiche des caractères cabalistiques dans un éditeur texte)
 - vidéo, son, image...

Les fichiers - Introduction

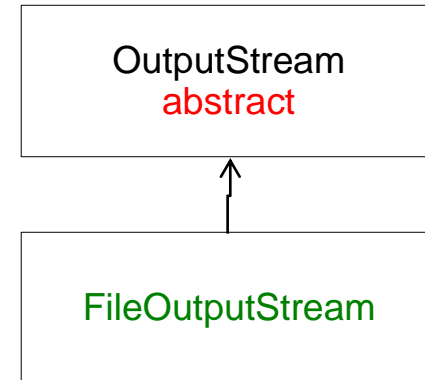
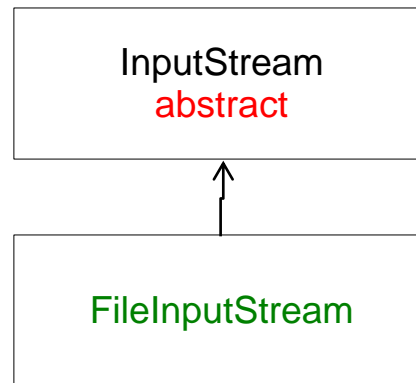
- Les flux permettent à un programme Java de lire ou écrire dans :
 - des fichiers,
 - des sockets,
 - les flux standards associés à un processus (`System.in, System.out, System.err)`
- Utilisation d'un flux :
 - **Ouverture du flux,**
 - Si le flux a été ouvert avec succès :
 - **Lecture ou écriture dans le flux,**
 - **Fermeture du flux.**

Les fichiers - Introduction

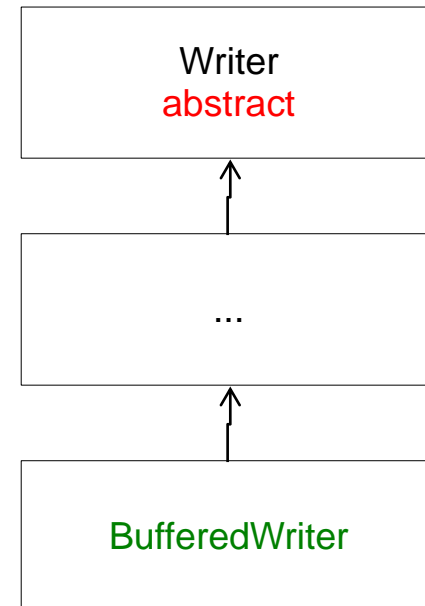
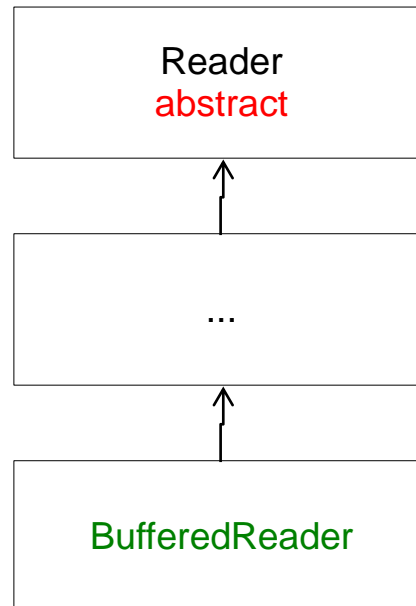
- Deux types de flux :
 - `in / reader` : flux lu par le programme.
 - `out / writer` : flux dans lequel le programme écrit.

Classes java associées aux fichiers

Le contenu est vu
comme une suite
d'octets.



Le fichier est vu
comme une suite
d'entités typées



FileInputStream

- Cette classe permet de lire un flux octet par octet.
- Si le fichier ne peut pas être **ouvert** (s'il n'existe pas, ou si les droits sont insuffisants), une exception de type `FileNotFoundException` sera levée.
 - Ainsi on ne passera pas dans le code qui **manipule** et **ferme** le fichier si le flux n'a pas pu être ouvert !

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
        FileInputStream fis = null;
        try {
            // Instanciation du FileInputStream
            fis = new FileInputStream(new File("test.txt"));

            // Tableau de byte de taille 8 pour la lecture du flux
            byte[] buffer = new byte[8];
            int n = 0;
            while((n = fis.read(buffer)) >= 0 ) {
                for(byte b : buffer) System.out.println(b);
            }
            fis.close();
        } catch(FileNotFoundException e) {
            e.printStackTrace();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ouverture

Manipulation

Fermeture

FileOutputStream

- Cette classe permet l'écriture de données dans un fichier octet par octet.
 - Pour écrire à la suite du fichier (append) on passe en 2eme paramètre la valeur **true**.
 - `exists()` teste l'existence d'un fichier.
- Si le fichier n'existe pas, il sera créé automatiquement.

```
import java.io.*;

public class Main {
    public static void main(String[] args)
    {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(new File("test.txt"), true);
            fos.write(new byte[]{'A'}); // Objet anonyme
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ouverture

Manipulation

Fermeture

BufferedReader

- Cette classe permet la lecture de données dans un fichier ligne par ligne ou octet par octet.
 - En cas de problèmes une exception est levée.

```
import java.io.*;
public class Main {
public static void main(String[] args) {
    BufferedReader br = null;
    String ligne = null;
    try {
        // Instanciation du BufferedReader
        br = new BufferedReader(new FileReader("test.txt"));

        // Affectation de la lecture de la ligne suivante dans ligne
        String ligne = br.readLine();
        System.out.println(ligne);

        br.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Ouverture

Manipulation

Fermeture

BufferedWriter

- Cette classe permet l'écriture de données dans un fichier ligne par ligne ou octet par octet.
 - Pour écrire sans effacer le contenu, mettre en 2eme paramètre, la valeur true.
 - `.exists()` de la classe `File` permet de tester l'existence d'un fichier.
- Si le fichier n'existe pas, il sera créé automatiquement.

```
package test;
import java.io.*;

public class Main {
    public static void main(String[] args) {
        BufferedWriter bw = null;
        try {
            bw = new BufferedWriter(new FileWriter("test.txt") true));
            bw.write("Bonjour");
            bw.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ouverture

Manipulation

Fermeture

TP : flux

■ Exercice 1 :

- Créer un fichier « input.txt », mettre du texte à l'intérieur.
- Lire le texte avec un `BufferedReader` et l'écrire dans la console.

■ Exercice 2 :

- Corriger le programme pour copier le fichier « input.txt » dans « output.txt ».

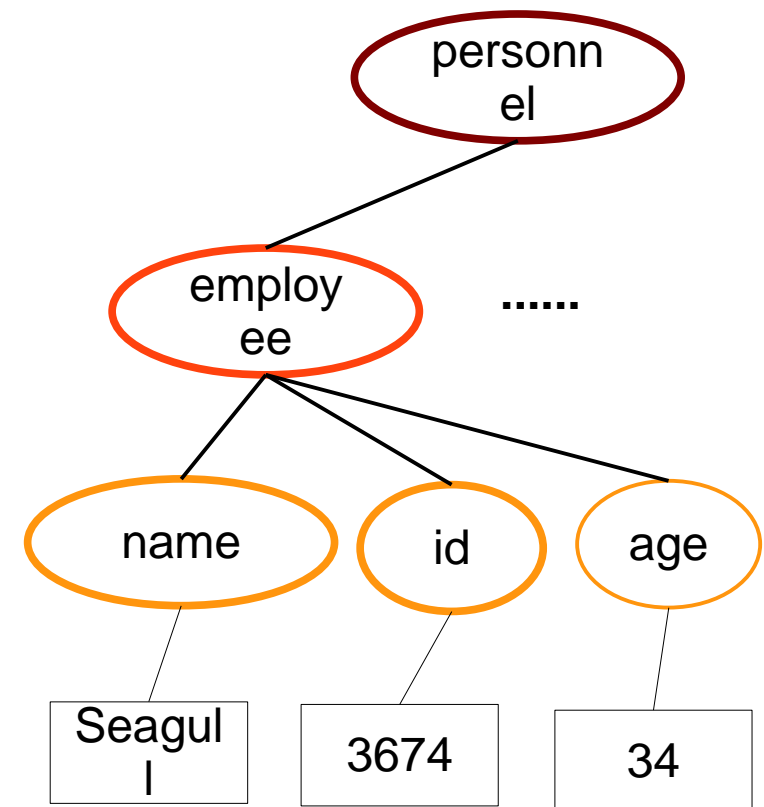
■ Exercice 3 :

- Lire une saisie utilisateur à l'aide de la classe `Scanner` et l'écrire dans le fichier.

Introduction

- Pour lire les fichiers **XML** en Java, il existe 2 méthodes essentielles appelées :
 - **SAX** (Simple API for XML)
 - **DOM** (Document Object Model)

```
<?xml version="1.0" encoding="UTF-8"?>
<Personnel>
  <Employee type="permanent">
    <Name>Seagull</Name>
    <Id>3674</Id>
    <Age>34</Age>
  </Employee>
  <Employee type="contract">
    <Name>Robin</Name>
    <Id>3675</Id>
    <Age>25</Age>
  </Employee>
  <Employee type="permanent">
    <Name>Crow</Name>
    <Id>3676</Id>
    <Age>28</Age>
  </Employee>
</Personnel>
```



```

import java.io.FileWriter;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;
import java.util.Date;
import javax.xml.namespace.QName;

public class CursorWriter {

    public static void main(String[] args) throws Exception {
        String fileName = "yourXML.html";
        XMLOutputFactory xof = XMLOutputFactory.newInstance();
        XMLStreamWriter xtw = null;
        xtw = xof.createXMLStreamWriter(new FileWriter(fileName));
        xtw.writeComment("all elements here are explicitly in the HTML namespace");
        xtw.writeStartDocument("utf-8", "1.0");
        xtw.setPrefix("html", "http://www.w3.org/TR/REC-html40");
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "html");
        xtw.writeNamespace("html", "http://www.w3.org/TR/REC-html40");
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "head");
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "title");
        xtw.writeCharacters("character");
        xtw.writeEndElement();
        xtw.writeEndElement();
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "body");
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "p");
        xtw.writeCharacters("another character");
        xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "a");
        xtw.writeAttribute("href", "http://www.java2s.com");
        xtw.writeCharacters("here");
        xtw.writeEndElement();
        xtw.writeEndElement();
        xtw.writeEndElement();
        xtw.writeEndElement();
        xtw.writeEndDocument();
        xtw.flush();
        xtw.close();
        System.out.println("Done");
    }
}

```

TP : écrire dans un fichier XML

■ Faire la méthode

```
boolean sauverPartie(
    String sFilename
){ ... }
```

■ Pour aborder le problème, il faut se demander :

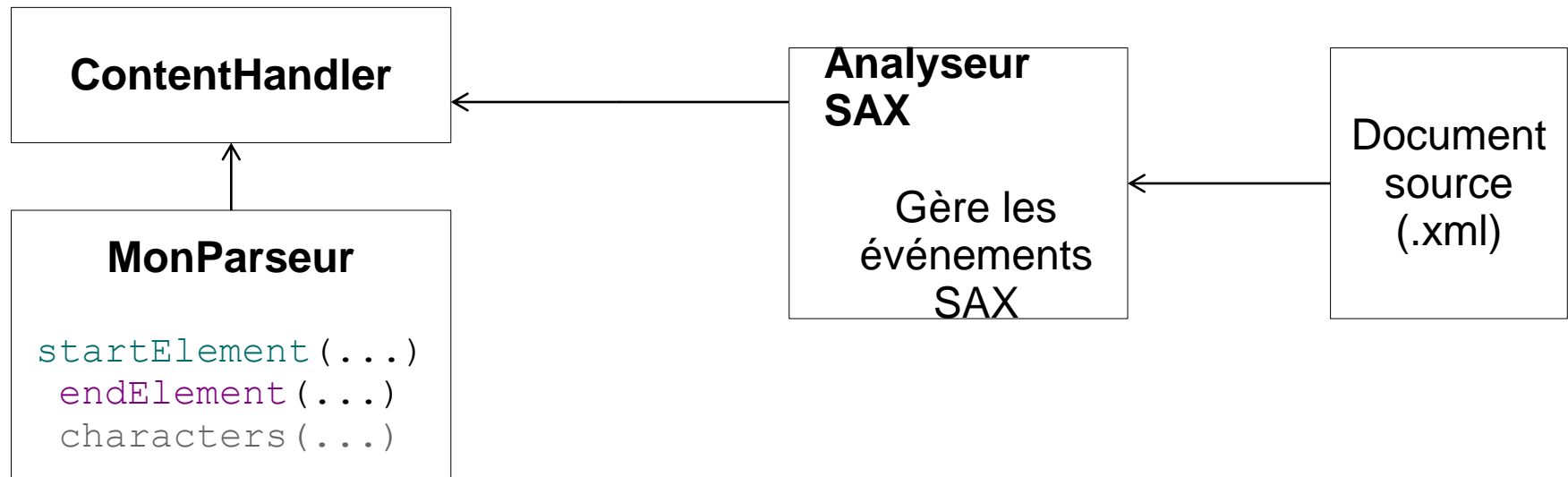
- Comment organiser le fichier XML ?
- S'inspirer du tutoriel du slide précédent
- <http://www.java2s.com/Code/Java/JDK-6/UsingXMLStreamWritertocreateXMLfile.htm>

```
<?xml version="1.0" encoding="UTF-8"?>
<Jeu>
  <TourJeu>
    <NoTour>24</NoTour>
    <Couleur>Noir</Couleur>
  </TourJeu>
</Echiquier>
<Echiquier>
  <Case>
    <Ligne>5</Ligne>
    <Colonne>5</Colonne>
    <Piece>
      <Type>C</Type>
      <Couleur>Blanc</Couleur>
    </Piece>
  </Case>
  ...
</Echiquier>
</Jeu>
```

Méthode SAX

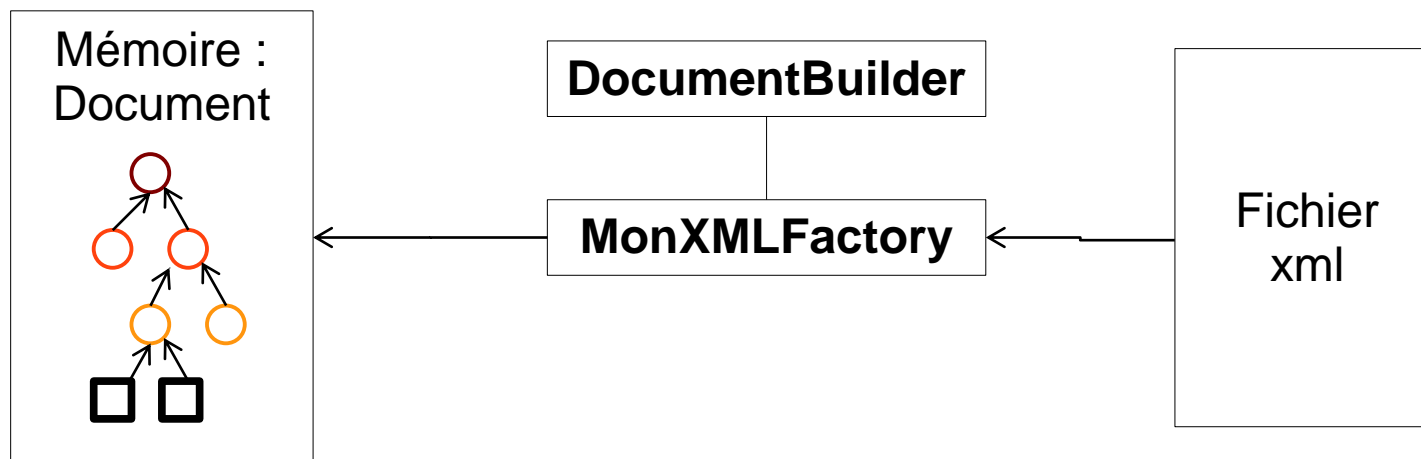
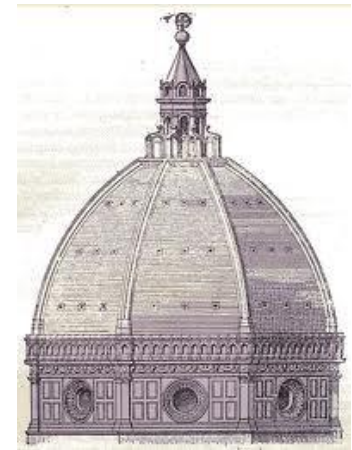


- On crée une classe qui implémente l'interface `ContentHandler`, en redéfinissant les méthodes de l'interface.
 - `<balise>` : passage par la méthode `startElement(...)`
 - `</balise>` : passage par la méthode `endElement(...)`
 - Pour interpréter le contenu d'une balise : `characters(...)`
- Le développeur implémente le comportement du parseur pour chaque type de balise qu'on peut rencontrer dans le fichier.
- Léger en mémoire, rapide : adaptés aux gros fichiers.
- L'architecture du code ne reflète pas la structure du document XML : pas modulaire.



Méthode DOM

- On utilise un objet `DocumentBuilder` afin de parcourir l'arbre via les méthodes :
 - `getElementsByTagName(...)`,
 - `getChildNodes()`,
 - `getNodeValue()`,
 - `getNodeName(...)`.
- L'analyse du fichier revient à parcourir un arbre
- Le parcours de l'arbre engendre des méthodes qui peuvent être calquées sur la structure du fichier XML.
- L'arbre doit être construit entièrement en mémoire pour en permettre le parcours !



TP : écrire dans un fichier XML

- Faire la méthode :
 - **boolean sauvegardeEleve** (String sFilename) {... }
 - Pour aborder le problème, il faut se demander:
 - Comment organiser le fichier XML ?

```
<eleves>
  <eleve>
    <nom>jean</nom>
    <prenom>paul</prenom>
    <annee>5</annee>
    <naiss>10/10/2010</naiss>
  </eleve> ...
</eleves>
```

Javadoc

- **javadoc** est le générateur de documentation fourni par le JDK.
 - Dans eclipse, sous windows, il faudra configurer le chemin vers javadoc.exe
- Javadoc tient compte des notions d'héritages pour éviter de (re)commenter des méthodes surchargées.
- Dans eclipse, les informations javadoc sont également affichées.
- javadoc se base sur les commentaires `/** ... */` et des balises particulières pour générer une documentation au format HTML
 - Voir doc/index.html dans le répertoire du projet
- La « javadoc » officielle est générée avec javadoc.

Javadoc

- 1) Placer les balises pour javadoc
 - Raccourci dans eclipse : cliquer sur la fonction à commenter, alt shift j

```
/**  
 * Indique si la lettre est une voyelle  
 * @param lettre La lettre à évaluer  
 * @return true si 'lettre' est une voyelle, false sinon  
 */  
boolean estVoyelle(char lettre){  
    return lettre == 'a' || lettre == 'e' || lettre == 'i'  
        || lettre == 'o' || lettre == 'u' || lettre == 'y';  
}
```

- 2) Invoquer javadoc
 - Dans eclipse : Project > Generate Javadoc

Balises javadoc

- Les commentaires javadoc s'appliquent à une classe, une méthode...
 - Le bloc de commentaire javadoc précède ce qu'il commente.
- Quelque soit la section de code commentée
 - **@author** : la ou les personnes qui ont participé au code
 - **@deprecated** : le code marqué est obsolète
 - **@see** : effectue un renvoi vers une classe, une méthode...
 - **@version** : indique la version
- Balises spécifiques
 - **@class** : commente une classe
 - **@exception** : commente une classe exception
 - Méthodes :
 - **@param p** : décrit le paramètre p
 - **@throws** : la méthode peut lever une exception
 - **@return** : décrit le retour d'une méthode

TP javadoc

- Commenter le code existant.
- Générer la documentation.
- Afficher la documentation dans un navigateur Internet.

Partie 8 : GUI

- Swing

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

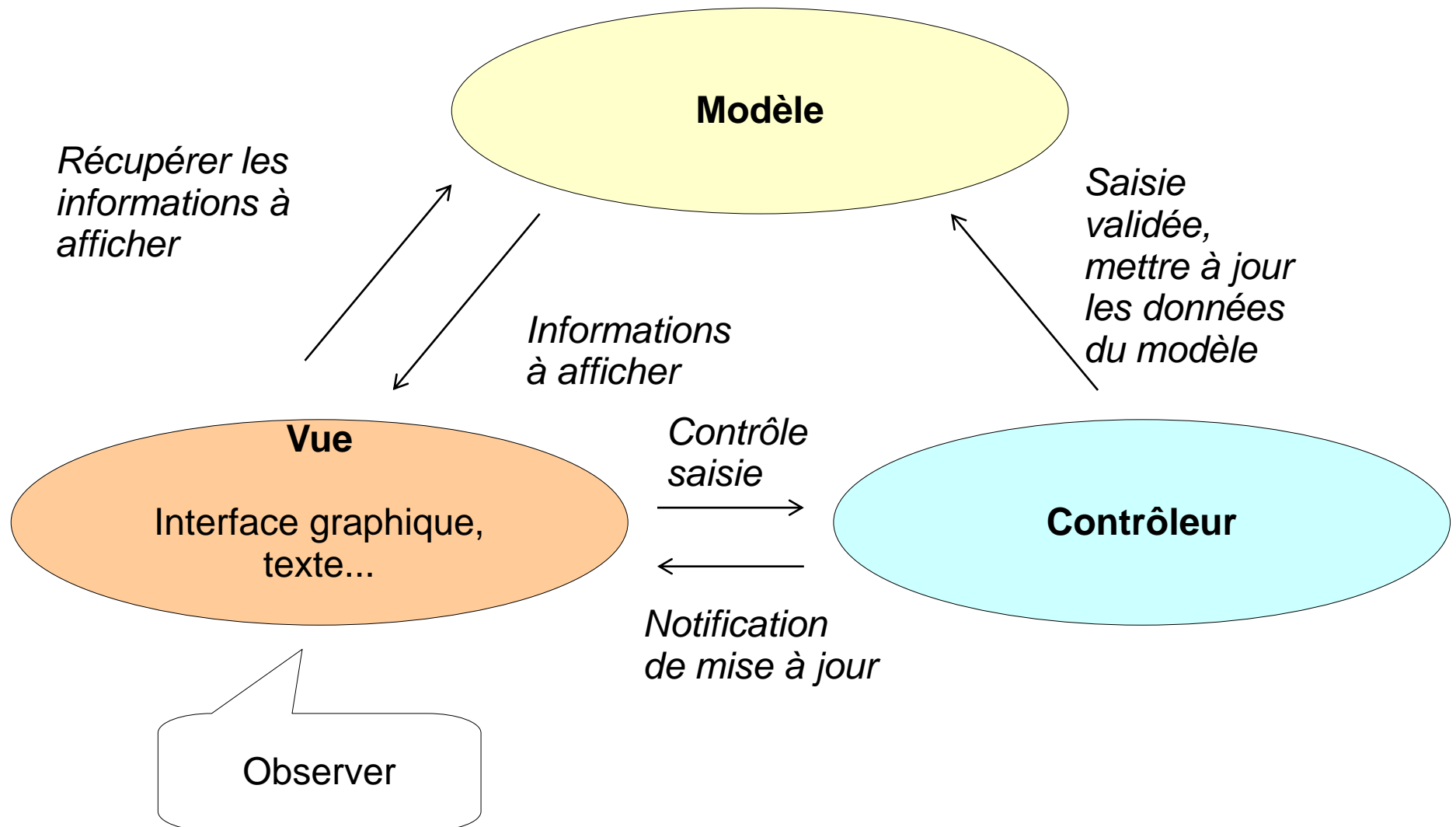
Conception d'une interface

- 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:

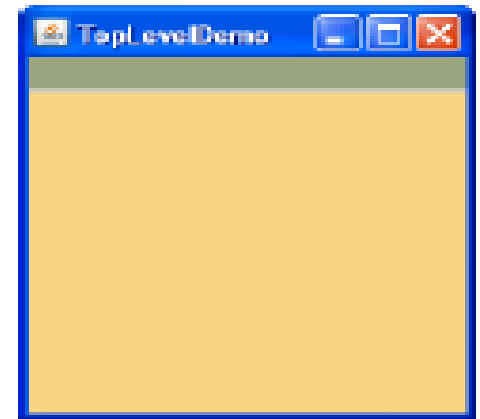


Fenêtre :JFrame

- 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{

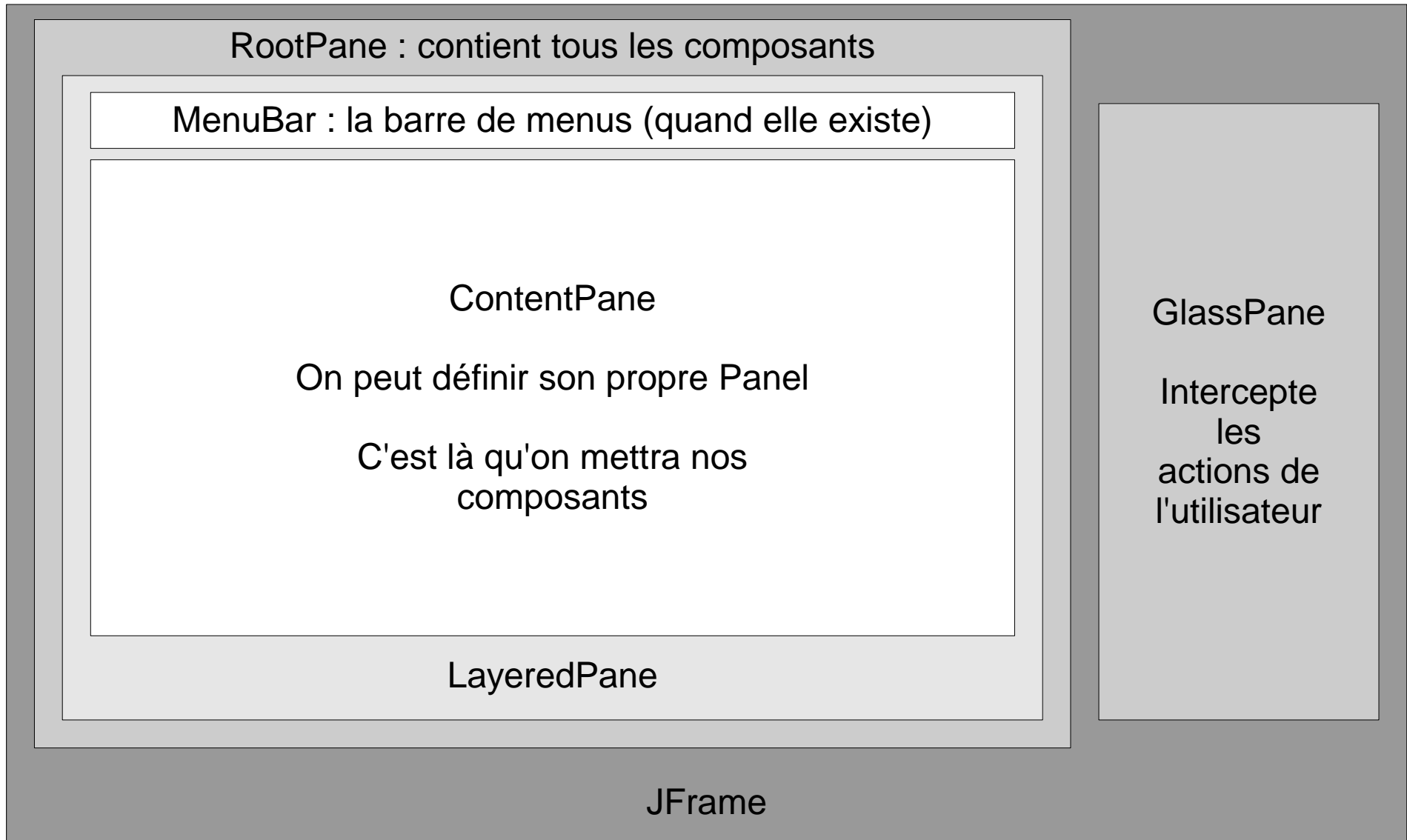
    public Fenetre(){
        this.setTitle("Titre de la fenêtre");
        this.setSize(400, 500);

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

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

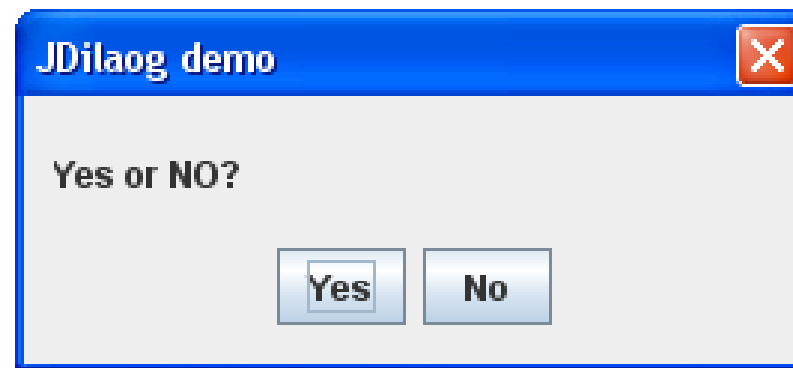
        // Par défaut un élément n'est pas visible !
        this.setVisible(true);
    }
}
```

La classe JFrame : structure



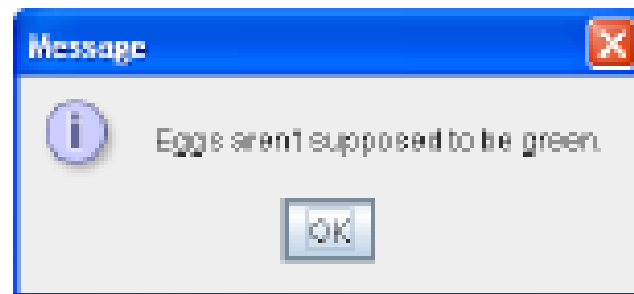
Fenêtre : JDialog

- 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 boîte de dialogue est ouverte



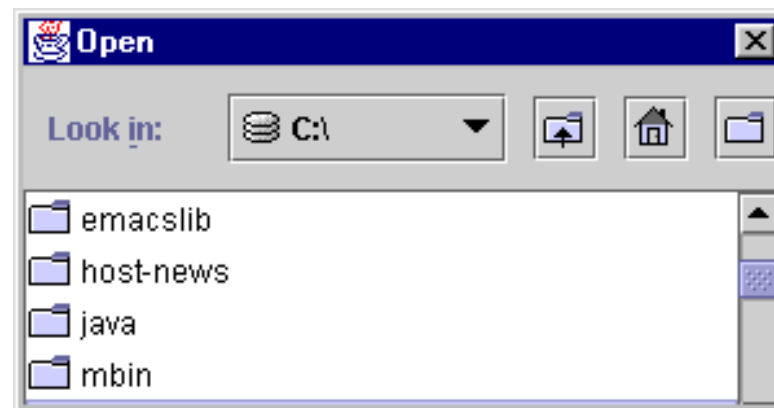
Fenêtre : JOptionPane

- Ce sont des boîtes de dialogues prédéfinies
- 3 types de boites :
 - 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.



# Conteneur

- 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

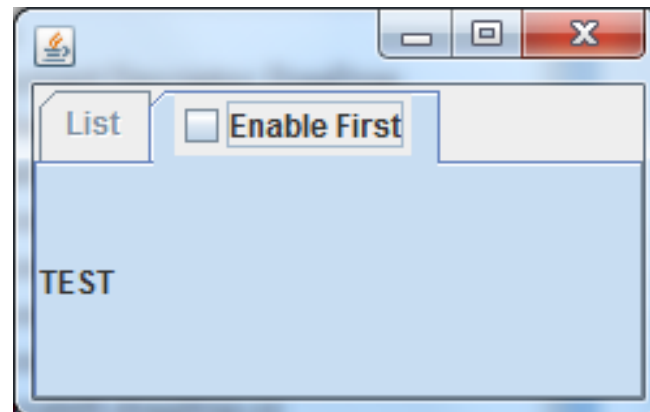
```
...
JTabbedPane tab = new JTabbedPane();
tab.addTab("onglet 1", new JLabel("Panneau 1"));
JTabbedPane tab = new JTabbedPane();
tab.addTab("onglet 2", new JLabel("Panneau 2"));
...
```



# 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"));
```

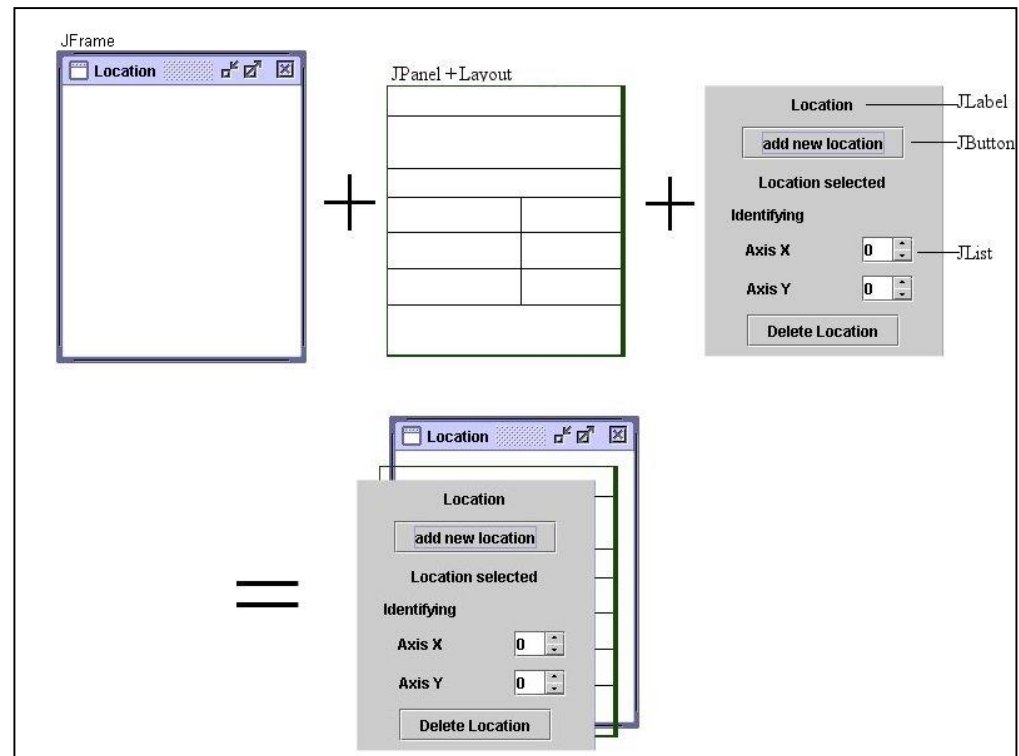


# Les composants

- 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.

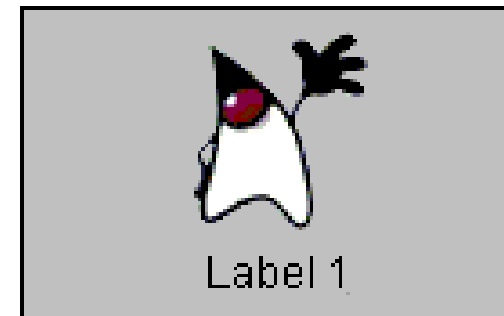




# JLabel

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

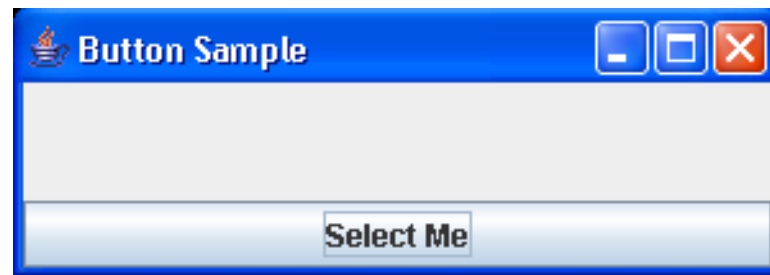
```
public JLabel(String s);
public JLabel(Icon i);
```



# 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);
 }
}
```

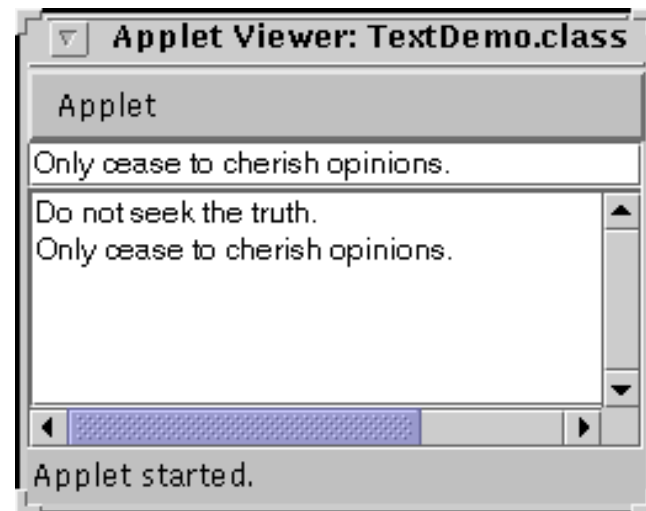
# JTextField

- Un JTextField est un composant qui permet d'écrire du texte. Il est composé d'une seule ligne contrairement au JTextArea
- 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](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

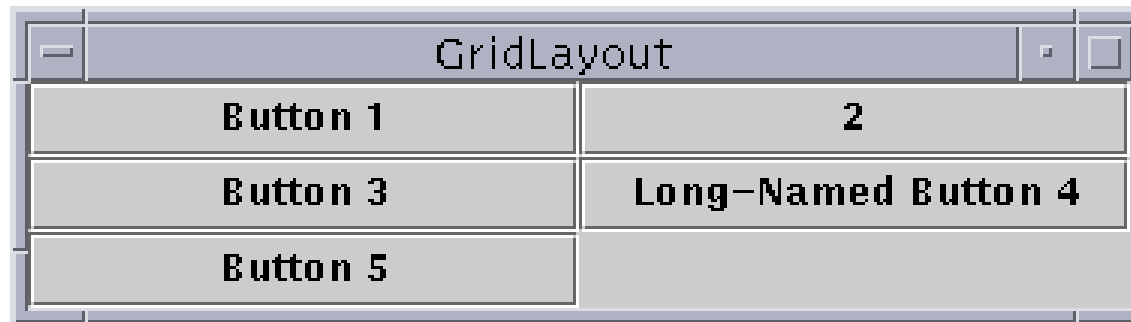
- Un FlowLayout permet de ranger les composants dans une ligne. Si l'espace est trop petit, une autre ligne est créée.



```
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 événements

- 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 événements

- 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 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 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 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, undoable edit	item	list selection	window	other
button	x		x		x			
check box	x		x		x			
color chooser			x					
combo box	x				x			
dialog								
editor pane		x		x			x	hyperlink
file chooser	x							
formatted text field	x	x		x				
frame						x		
internal frame								internal frame
list						x		list data
menu								menu
menu item	x		x		x			menu key menu drag mouse

PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

Component	Listener							
	action	caret	change	document, undoable edit	item	list selection	window	other
option pane								
password field	x	x		x				
popup menu								popup menu
progress bar			x					
radio button	x		x		x			
slider			x					
spinner			x					
tabbed pane			x					
table						x		table model table column model cell editor
text area		x		x				
text field	x	x		x				
text pane		x		x				hyperlink
toggle button	x		x		x			
tree								tree expansion tree will expand tree model tree selection
viewport (used by scrollpane)			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^{ème} 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 délèguer 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 JButton1("mon bouton 1");

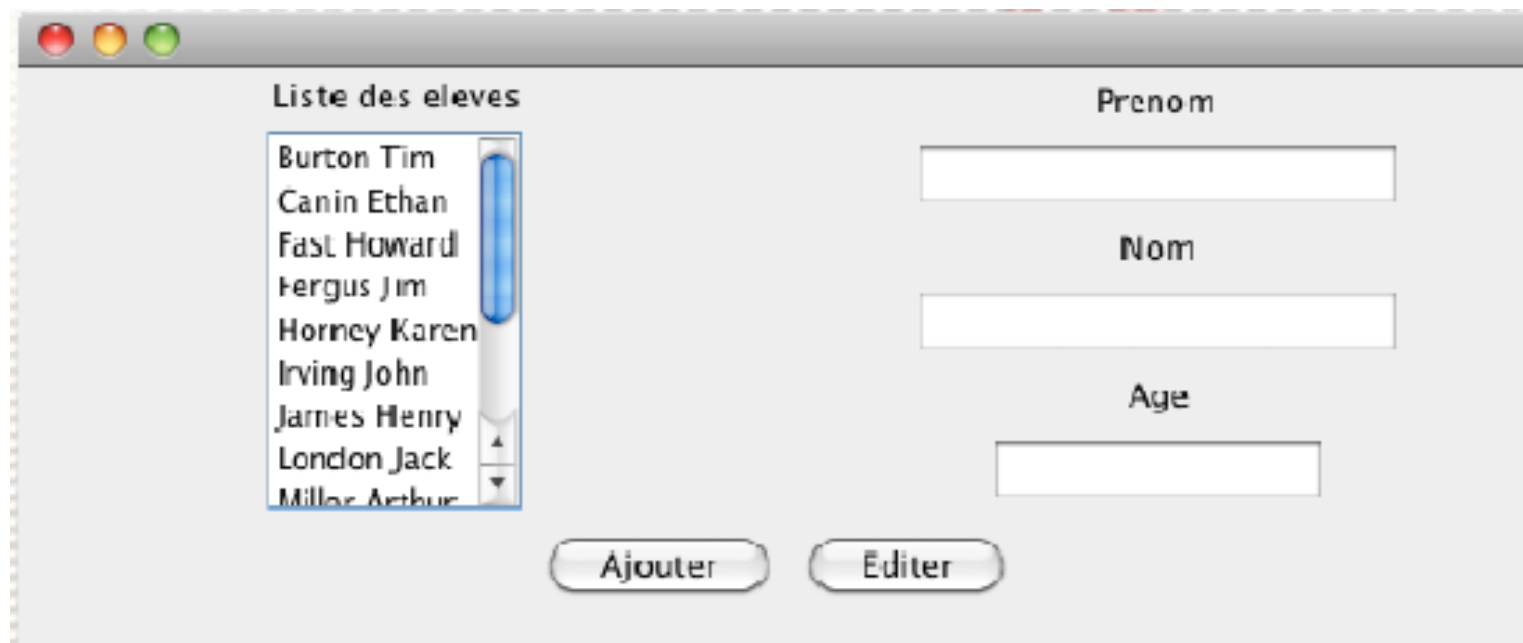
    // Constructeur
    public Fenetre(ActionEvent arg0) {
        //...
        bouton1.addActionListener(new Bouton1Listener());
        // idem avec les autres composants
    }

    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



Window Builder

■ Installer Window Builder

- Vérifiez votre version d'Eclipse (Help, About Eclipse...)
- Ajouter le repository Window Builder (Help, Install new software
 - <http://code.google.com/intl/fr/webtoolkit/tools/download-wbpro.html>
- Cocher la case permettant d'installer Swing Builder

Partie 9 : programmation par annotation

Introduction

- Une **annotation** est une **meta-information** appliquée à une déclaration (de classe, de méthode...)
 - ignorée par le compilateur et la JVM,
 - prise en compte par le **tisseur (weaver)**.
 - notation : précédée d'une **@**
 - une annotation peut être paramétrée
- Quelques annotations : **@deprecated**, **@overrides**, ...
 - On peut définir ses propres annotations
- Intérêt : facilite le développement
 - Réduit le volume du code
 - Rationnalise le code

Les annotations standards

- **@SuppressWarnings** : efface les warnings du compilateur
- **@Deprecated** : indique que la méthode ne devrait plus être utilisée car elle est obsolète. Si elle est utilisée (y compris via un héritage ou une surcharge) : warning à la compilation.
- **@Overrides** : indique que l'on est en train de surcharger une méthode de la classe mère. Sinon erreur de compilation.

■ Exemple :

```
public class Parent{
    @Deprecated
    public void m(int x) {

System.out.println("Parent.m()");
    }
}
```

```
public class Enfant extends Parent{
    @Overrides
    public void m() {

System.out.println("Enfant.m()");
    }
}
```

- Tel quel : erreur de compilation (signatures différentes → **@Overrides** non satisfait).
 - Si on corrige la signature : warning (**@Deprecated** hérité)

Plus loin avec les annotations

- On peut définir ses propres annotations
 - On utilise pour cela des méta-annotations :
 - annotation : annote du code
 - méta-annotation : annote une annotation
 - `@interface`, `@Inherited`, `@Documented`, `@Target`, `@Retention`
- Certains frameworks définissent leurs annotations :
 - Junit (Tests) : `@Before`, `@After`, `@Test`
 - Hibernate (conversion objet ↔ enregistrements SQL) : `@Entity`, `@Table`, `@Column`, `@Id`
 - ...

Fin de la formation

Merci pour votre participation 😊