

الكلية متعددة التخصصات - ورازات  
+043101+ +0X+2\*H2+- U.O\*O\*O\*+  
FACULTÉ POLYDISCIPLINAIRE DE OUARZAZATE



# INTERFACES HOMME-MACHINE

Responsable : Pr. Mohamed Benaddy

# Plan de Cours

1. **Rappels de POO (JAVA)**
2. **Introduction à l'interface homme machine**
3. **L'ergonomie IHM**
4. **Interface Homme Machine en Java (Swing)**
5. **Gestion du positionnement**
6. **Gestion des événements**

Chapitre I

# Rappels de POO (JAVA)

# Rappels de POO (JAVA)

## La programmation orientée objet

L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

Ce chapitre contient plusieurs sections :

- Le concept de classe et d'objet
- Les attributs et les méthodes
- L'héritage.
- polymorphisme

# Rappels de POO (JAVA)

## Concept de classe

- Une classe est le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.
- Java est un langage orienté objet : tout appartient à une classe sauf les variables de types primitives.
- Pour accéder à une classe il faut en déclarer une instance de classe ou objet.
- Une classe comporte sa déclaration, des variables et les définitions de ses méthodes.
- Une classe se compose de deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gèrent leur accessibilité par les composants hors de la classe.

# Rappels de POO (JAVA)

## Syntaxe de déclaration d'une classe

- La syntaxe de déclaration d'une classe est la suivante :  
modificateurs class nom\_de\_classe [extends classe\_mere] [implements interfaces] { ... }

Les modificateurs de classe (ClassModifiers) sont :

abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

# Rappels de POO (JAVA)

## Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En Java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme `nom_de_classe nom_de_variable`

## Exemple

```
MaClasse m;  
String chaine;
```

L'opérateur `new` se charge de créer une instance de la classe et de l'associer à la variable

## Exemple

```
m = new MaClasse();  
MaClasse m = new MaClasse();
```

# Rappels de POO (JAVA)

## Les objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit `c1 = c2` (`c1` et `c2` sont des objets), on copie la référence de l'objet `c2` dans `c1`. `c1` et `c2` font référence au même objet : ils pointent sur le même objet. L'opérateur `==` compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

## Exemple :

```
Rectangle r1 = new Rectangle(100,50);  
Rectangle r2 = new Rectangle(100,50);  
if (r1 == r1) { ... } // vrai  
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'une méthode à cet effet : la méthode `equals()` héritée de `Object`.



# Rappels de POO (JAVA)

## Les variables de classes

Elles ne sont définies qu'une seule fois quel que soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé **static**

### Exemple :

```
public class MaClasse() {  
    static int compteur = 0;  
}
```

- L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.
- Ce type de variable est utile pour, par exemple, compter le nombre d'instanciations de la classe.

# Rappels de POO (JAVA)

## La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égal à l'instance de l'objet dans lequel il est utilisé.

### Exemple :

```
private int nombre;  
public maclasse(int nombre) {  
    nombre = nombre; // variable de classe = variable en paramètre du constructeur  
}
```

Il est préférable de prefixer la variable d'instance par le mot clé this.

### Exemple :

```
this.nombre = nombre;
```

# Rappels de POO (JAVA)

## Modificateurs d'accès

- Ils s'appliquent aux classes, aux méthodes et aux attributs.
- Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.
- Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

Modificateur	Rôle
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Depuis la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut : package friendly	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles

# Rappels de POO (JAVA)

## Le mot clé static

Le mot clé **static** s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois en mémoire. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé **static**.

## Exemple :

```
public class Cercle {  
    static float pi = 3.1416f;  
    float rayon;  
    public Cercle(float rayon) { this.rayon = rayon; }  
    public float surface() { return rayon * rayon * pi;}  
}
```

# Rappels de POO (JAVA)

## Le mot clé final

- Le mot clé final s'applique aux variables de classe ou d'instance ou locales, aux méthodes, aux paramètres d'une méthode et aux classes. Il permet de rendre l'entité sur laquelle il s'applique non modifiable une fois qu'elle est déclarée pour une méthode ou une classe et initialisée pour une variable.
- Une variable qualifiée de final signifie que la valeur de la variable ne peut plus être modifiée une fois que celle-ci est initialisée.
- Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

# Rappels de POO (JAVA)

## Le mot clé abstract

- Le mot clé abstract s'applique aux méthodes et aux classes.
- Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous-classes.
- Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.
- Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous-classe. L'abstraction permet une validation du codage : une sous-classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.
- Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

# Rappels de POO (JAVA)

Les mots clés **synchronized**, **volatile**, et **native**

## **Le mot clé synchronized**

Il permet de gérer l'accès concurrent aux variables et méthodes lors de traitements de threads (exécution « simultanée » de plusieurs petites parties de code du programme)

## **Le mot clé volatile**

Le mot clé volatile s'applique aux variables. Il précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, sa valeur est lue et réécrite immédiatement si elle a changé.

## **Le mot clé native**

Une méthode native est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse d'exécution plus rapide.

# Rappels de POO (JAVA)

## Les propriétés ou attributs

Les données d'une classe sont contenues dans des variables nommées propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

- Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.
- Les variables de classes sont définies avec le mot clé static
- Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

Exemple :

```
public class MaClasse {  
    public int valeur1 ;  
    int valeur2 ;  
    protected int valeur3 ;  
    private int valeur4 ;  
    static int compteur ;  
    final double pi=3.14 ;  
}
```



# Rappels de POO (JAVA)

## Les méthodes

- Les méthodes sont des fonctions qui implémentent les traitements de la classe.
- Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise void.
- Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis.

## Remarque :

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante :

Exemple :

```
public class MonApp1 {  
    public static void main(String[] args) {  
        System.out.println("Bonjour");  
    }  
}
```

# Rappels de POO (JAVA)

## Les méthodes

- La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments.
- Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.
- Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes.

## Exemple :

```
class affiche{  
    public void afficheValeur(int i) {  
        System.out.println(" nombre entier = " + i);  
    }  
    public void afficheValeur(float f) {  
        System.out.println(" nombre flottant = " + f);  
    }  
}
```

# Rappels de POO (JAVA)

## Les constructeurs

- La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ.
- Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void. On peut surcharger un constructeur.
- La définition d'un constructeur est facultative. Si aucun constructeur n'est explicitement défini dans la classe, le compilateur va créer un constructeur par défaut sans argument.

### Exemple :

```
Class MaClassee{  
    private int nombre;  
  
    public MaClasse(int valeur) {nombre = valeur;}  
    public MaClasse() {nombre = 5;}  
}
```

# Rappels de POO (JAVA)

## Le destructeur

- Un destructeur permet d'exécuter du code lors de la libération, par le garbage collector, de l'espace mémoire occupé par l'objet. En Java, les destructeurs appelés finaliseurs (finalizers), sont automatiquement invoqués par le garbage collector.
- Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

# Rappels de POO (JAVA)

## Les accesseurs

L'**encapsulation** permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées **private** à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe.

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée. Par convention, les accesseurs en lecture commencent par **get** et les accesseurs en écriture commencent par **set**.

Exemple :

```
private int valeur = 13;
public int getValeur(){
    return(valeur);}
public void setValeur(int val) {
    valeur = val;}
```

**Remarque:** Pour un attribut de type booléen, il est possible de faire commencer l'accesseur en lecture par **is** au lieu de **get**.

# Rappels de POO (JAVA)

## L'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- une classe mère ou super-classe
- une classe fille ou sous-classe qui hérite de sa classe mère

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre. Les sous-classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

# Rappels de POO (JAVA)

## L'héritage

- L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super-classes et de sous-classes. Une classe qui hérite d'une autre est une sous-classe et celle dont elle hérite est une super-classe. Une classe peut avoir plusieurs sous-classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.
- Object est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.

## Exemple :

```
class Fille extends Mere { ... }
```

# Rappels de POO (JAVA)

## Redéfinition d'une méthode et polymorphisme

- La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parente (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).
- Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

Le **polymorphisme** est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé.

**Exemple** : la classe Employe hérite de la classe Personne

```
Personne p = new Personne ("Dupond", "Jean");
```

```
Employe e = new Employe("Durand", "Julien", 10000);
```

```
p = e ; // ok : Employe est une sous-classe de Personne
```

```
Object obj;
```

```
obj = e ; // ok : Employe hérite de Personne qui elle même hérite de Object
```



# Rappels de POO (JAVA)

## Les interfaces

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super-classes. Ce mécanisme n'existe pas en Java. Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot clé `interface` et sont intégrées aux autres classes avec le mot clé `implements`. Une interface est implicitement déclarée avec le modificateur `abstract`.

# Rappels de POO (JAVA)

## Les interfaces

### Exemple :

```
interface AfficheType {  
    void afficherType();  
}  
  
class Personne implements AfficheType {  
    public void afficherType() {  
        System.out.println(" Je suis une personne ");  
    }  
}  
  
class Voiture implements AfficheType {  
    public void afficherType() {  
        System.out.println(" Je suis une voiture ");  
    }  
}
```

# Rappels de POO (JAVA)

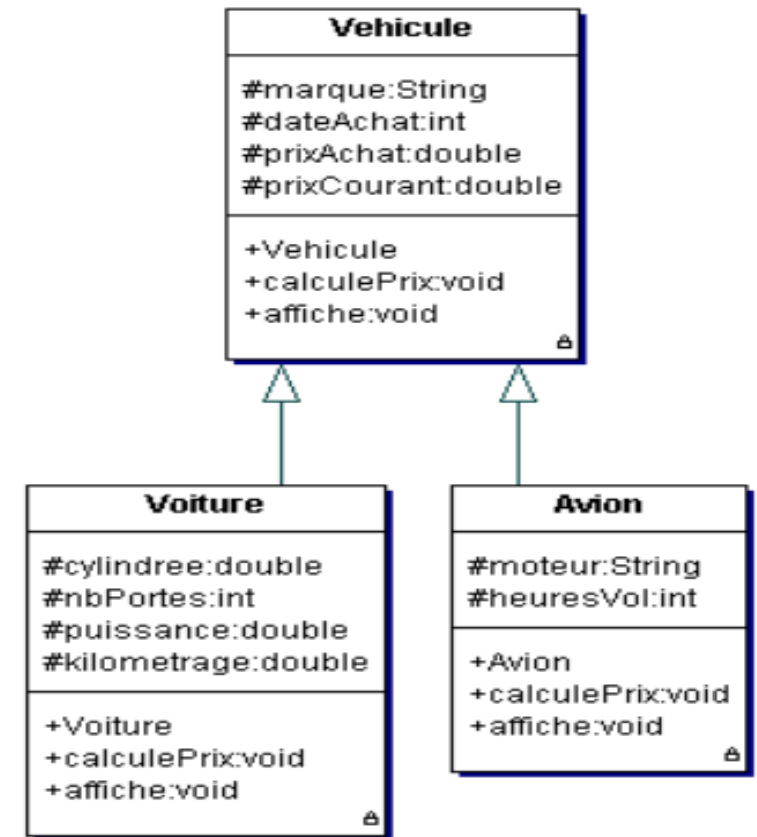
## Les interfaces

- Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles.
- Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont implicitement publiques. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package.
- Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur `static` et `final` même si elles sont définies avec d'autres modificateurs.
- Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.

# Rappels de POO (JAVA)

## Exercice 1

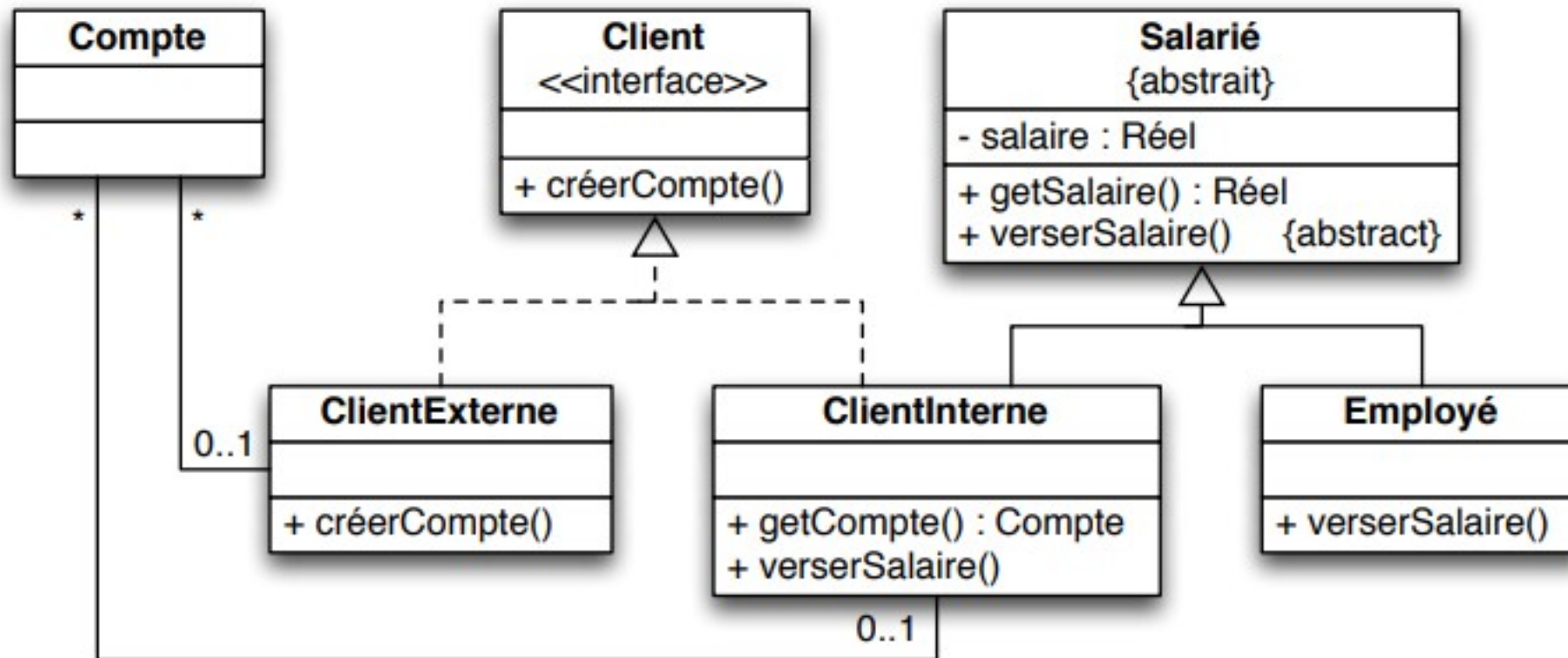
Ecrire les classes représentées dans le diagramme suivant :



# Rappels de POO (JAVA)

## Exercice 2

Ecrire les classes représentées dans le diagramme suivant :



## Chapitre II

# L'interaction homme machine

# Introduction aux Interfaces Homme-Machine

## Définitions :

- **Interface:** signifie la surface de contact, ou la frontière, entre deux corps ou deux régions d'espace.
- **IHM signifie :** -Interface Homme – Machine  
-Interactions Homme – Machine

Mais aussi ...

- Communication Homme – Machine
- Dialogue Homme – Machine
- Interaction Personne – Machine

# Introduction aux Interfaces Homme-Machine

## Définitions :

- **L'interface homme-machine** est un ensemble de dispositifs matériels et logiciels permettant de contrôler et de communiquer avec une machine.
- **L'interaction homme-machine** est une action qui provoque une réaction perceptible. Elle regroupe des notions telles que les clics souris, la frappe de touches, les pressions sur un écran tactile, etc.
- **Le dialogue Homme-Machine** est l'ensemble des échanges entre un utilisateur et une machine. Ces échanges regroupent les manipulations avec un écran tactile, l'utilisation d'un microphone, d'un scanner, d'une table à digitaliser, etc.



# Introduction aux Interfaces Homme-Machine

## Domaine d'utilisation des IHM :

- Informatique:
  - Programmation
  - Génie logiciel
  - Synthèse et reconnaissance de la parole, langue naturelle
  - Intelligence artificielle, traitement d'images
  - ...
- Psychologie cognitive
- Sciences de l'éducation, didactique
- Ergonomie cognitive, ergonomie des logiciels
- Communication, graphisme, audiovisuel, design

# Introduction aux Interfaces Homme-Machine

## Domaine d'utilisation des IHM :

- Où l'homme exerce une action sur une machine pour trouver un résultat on trouve l'IHM
  - Médecine: chirurgie
  - Téléformation
  - Transport
  - Industrie
  - Transactions financières
  - Commande de chaînes de production
  - Maintenance de processus complexes
  - Bureautique
  - ...

# Introduction aux Interfaces Homme-Machine

## Qui conçoit les IHM? :

- Les 3 participants dans la conception des interfaces homme-machine sont:
  - l'utilisateur: participant avec choix
  - la machine: participant avec programme
  - le concepteur: participant qui anticipe les choix possibles de l'utilisateur et les code dans un programme

# Introduction aux Interfaces Homme-Machine

## Caractéristiques des utilisateurs des IHM:

- Caractéristiques physiques
  - Âge
  - Handicap
- Caractéristiques psychologiques
  - Visuel/auditif, logique/intuitif, analytique/synthétique...
- Connaissances et expériences
  - Dans le domaine de la tâche (novice, expert, professionnel)
  - En informatique, sur le système (usage occasionnel, quotidien)
- Caractéristiques socio-culturelles
  - Sens d'écriture
  - Format des dates
  - Signification des icônes, des couleurs (en occident : rouge = stop, en chine : rouge = joie, mariage)

# Introduction aux Interfaces Homme-Machine

## Historique et évolution des IHM:

- 1945-1956: Premiers ordinateurs ENIAC (1943) Mark I (1944)  
Interactions HM inexistantes ou quasi-inexistantes
- 1956-1971: 2ème et 3ème générations d'ordinateurs
  - 1962 : Ivan Sutherland (fondateur de l'Informatique Graphique et début de l'interaction homme-machine)
  - 1963 : Ted Nelson Inventeur des termes hypertexte (1963, 1965 ?) et hypermedia (1968)
  - 1967: Le premier casque de réalité virtuelle, réalisé par Sutherland et Sproull affichant des images de synthèse.
  - 1968 : première souris (Doug Engelbart)
- 1980s: Ordinateurs à grand public: système de fenêtrage et utilisation de la souris et du clavier
  - 1981 : Xerox
  - 1983 : Apple Lisa,
  - 1984 : Apple Macintosh,
  - 1990 : Windows 3.0

# Introduction aux Interfaces Homme-Machine

## Historique et évolution des IHM:

- 1990: Systèmes plus conviviaux, faciles à comprendre et à utiliser
  - Interfaces graphiques
    - manipulation directe
    - action directe pour les objets représentés à l'écran
  - WYSIWYG
    - What You See Is What You Get
    - ACAI : Affichage Conforme A l'Impression
- 2000: Dispositifs évolués
  - vision 3D
  - synthèse vocale
  - combinaison de types de données
  - Interactions gestuelles
  - VR
- IA

# Introduction aux Interfaces Homme-Machine

## Paradigmes d'interfaces:

On peut observer que les IHM sont de plus en plus déconnectées de l'implémentation réelle des mécanismes contrôlés. Dans son article de 1995, *The Myth of Metaphor*<sup>7</sup>, Alan Cooper distingue trois grands paradigmes d'interface :

- **Le paradigme technologique** : l'interface reflète la manière dont le mécanisme contrôlé est construit. Cela conduit à des outils très puissants mais destinés à des spécialistes qui savent comment fonctionne la machine à piloter.
- **Le paradigme de la métaphore** qui permet de mimer le comportement de l'interface sur celui d'un objet de la vie courante et donc déjà maîtrisé par l'utilisateur. Exemple : la notion de document.
- **Le paradigme idiomatique** qui utilise des éléments d'interface au comportement stéréotypé, cohérent et donc simple à apprendre mais pas nécessairement calqué sur des objets de la vie réelle.

# Introduction aux Interfaces Homme-Machine

## Modes d'interaction

L'interaction est dite multimodale si elle met en jeu plusieurs modalités sensorielles et motrices. Un système interactif peut contenir un ou plusieurs de ces modes d'interaction :

- Mode parlé : commandes vocales, guides vocaux...
- Mode écrit : entrées par le clavier et la tablette graphique, affichage du texte sur l'écran...
- Mode gestuel : désignation 2D ou 3D (souris, gants de données, écran tactile), retour d'effort...
- Mode visuel : graphiques, images, animations...



# Introduction aux Interfaces Homme-Machine

## Les périphériques IHM

D'un point de vue organique, on peut distinguer trois types d'IHM :

- Les interfaces d'acquisition : bouton, molette, souris, clavier accord, joystick, clavier d'ordinateur, clavier MIDI, télécommande, capteur de mouvement, microphone avec la reconnaissance vocale, etc.
- Les interfaces de restitution : écran, témoin à LED, voyant d'état du système, haut parleur, etc.
- Les interfaces combinées : écran tactile, multi-touch et les commandes à retour d'effort.

## Chapitre III

# L'ergonomie IHM

# L'ergonomie IHM

## Définitions :

- L'ergonomie est l'étude scientifique de la relation entre l'homme et ses moyens, méthodes et milieux de travail. Son objectif est d'élaborer, avec le concours des diverses disciplines scientifiques qui la composent, un corps de connaissances qui dans une perspective d'application, doit aboutir à une meilleure adaptation à l'homme des moyens technologiques de production, et des milieux de travail et de vie.
- L'ergonomie IHM consiste à faire correspondre la conception et le fonctionnement des interfaces digitales aux différents besoins de l'utilisateur cible. Le but est de faciliter et d'améliorer l'utilisation des solutions ou moyens digitaux mis à sa disposition.

# L'ergonomie IHM

## Définitions :

- L'ergonomie IHM, synonyme d'utilisabilité, est une science qui vise à rendre un service maniable et facile à utiliser. C'est une discipline qui se concentre sur les Interfaces Homme-Machine en s'appuyant sur des données, analyses mais aussi sur des études de comportement. Son objectif est d'aboutir à optimiser l'adaptation et le bien-être de l'utilisateur aux moyens technologiques conçus.
- L'UX design (User eXperience Design) est le design de l'expérience utilisateur. Il se définit comme l'ensemble des techniques permettant de penser et créer une interface, répondant aux attentes de l'utilisateur lors de la conception d'une plateforme (site web ou application mobile).

# L'ergonomie IHM

## Différences entre l'ergonomie IHM et l'UX design

- **L'ergonomie web : très précise et très focus**
  - L'ergonomie web consiste à créer des systèmes pratiques, facilement utilisables et maniables avec un maximum de confort et de sécurité pour l'utilisateur cible.
  - Il s'agit d'une approche scientifique et technique, qui fait appel à des méthodes d'observation et des études de comportements des utilisateurs pour cerner le modèle mental de ces derniers.
  - L'ergonomie IHM c'est au final donner à l'utilisateur le moyen d'atteindre son objectif en assurant la fiabilité de la solution.
- **L'UX design : bien plus global**
  - L'UX design (User eXperience Design) quant à lui se focalise principalement sur l'impact émotionnel, les ressentis et le plaisir éprouvé lors ou à la suite de l'utilisation de la solution digitale. Il est centré sur l'expérience utilisateur et vise la fidélisation des utilisateurs cibles, grâce à la convivialité de la solution proposée.

# L'ergonomie IHM

## L'ergonomie cognitive

- L'ergonomie cognitive est une partie intégrante de l'ergonomie en général mais plus particulièrement de l'ergonomie IHM. Cette discipline obéit aux grandes fonctions cognitives ou modèles mentaux de l'être humain, à savoir le raisonnement, la mémoire et la perception, l'attention, etc.
- L'ergonomie cognitive s'appuie sur les sciences cognitives ainsi que sur la psychologie cognitive, dans le but d'arriver à trouver une adéquation entre les déterminants cognitifs des utilisateurs et les fonctionnalités de l'interface finale proposée.

# L'ergonomie IHM

## Objectif

- **Objectifs centrés sur les personnes :**
  - Sécurité
  - Confort, Facilité d'usage, satisfaction, plaisir
  - Intérêt de l'activité, du travail
  - Santé cognitive : favorise le développement de compétences
- **Objectifs centrés sur la performance**
  - Efficacité, Productivité
  - Fiabilité
  - Qualité

# L'ergonomie IHM

## Approches

- **Approche technocentrée:**
  - centrée sur la machine et ses possibilités
  - l'utilisateur doit s'adapter à la machine
  - L'ère des Command Lines Interfaces (CLI) – 1969-1983
- **Approche anthropocentrée**
  - centrée sur l'homme et ses besoins
  - la machine doit s'adapter à l'utilisateur
  - L'ère des Graphical User Interfaces (GUI) – 1984 à aujourd'hui



## Chapitre IV

# Interface Homme Machine en Java

# IHM en Java

## Interfaces graphiques (GUI)

Dans la majorité des applications, la partie la plus importante de l'interaction homme-machine fait appel aux périphériques de base qui sont : l'**écran**, le **clavier** et la **souris**.

- Progressivement, les interfaces en **mode caractères** (lignes de commandes entrées au clavier) ont été substituées par des interfaces utilisant le mode graphique (**Graphical User Interface** ou **GUI**).
- Les **interfaces graphiques** utilisent différents **éléments visuels** appelés **Composants graphiques** ou **Widgets** ou **Controls** et qui, en combinaison avec l'utilisation de la souris rendent les choses plus simples et plus intuitives pour l'utilisateur.

✂ Parmi ces **éléments visuels** on peut citer :

- Les **boutons** (cases à cocher, bouton radio, etc)
- Les **menus** (menus déroulants, menus contextuels, etc)
- Les **listes à choix** (listes simples, listes déroulantes)
- Les **champs de texte** (champs simples, champ multilignes, éditeurs, etc)
- Les **barres de défilement** (ascenseurs)
- . . .

# IHM en Java

## Interfaces graphiques (GUI)

- Les interfaces utilisateur qui intègrent ces éléments graphiques (composants visuels / *Widgets* / *Controls*) sont parfois désignées par l'acronyme **WIMP** qui est l'abréviation de :
  - **Window** : Notion de "fenêtre" (zone d'interaction indépendante) (c'est une notion importante qui a même donné son nom à un système d'exploitation !)
  - **Icon** : Éléments graphiques visuels (images, boutons, champs de texte, bulles d'aide, etc.)
  - **Menu** : Choix d'actions parmi une liste proposée (barres de menu, menus déroulants, menus contextuels, rubans, etc.)
  - **Pointer** : Curseur/Pointeur manipulé par la souris, et qui permet d'interagir avec les composants visuels (pointage, sélection, tracé, drag&drop)

# IHM en Java

## Plate-forme Java

- ✂ La **plate-forme Java** est composée d'un ensemble de libraires et d'outils qui constituent les briques de base (*Core Libraries*) sur lesquelles on construit les application. Ces librairies sont disponibles dans toutes les implémentations des machines virtuelles Java (indépendamment du système d'exploitation et du type de machine cible).
- ✂ Certaines de ces librairies sont destinées à la gestion des interfaces utilisateur graphiques (GUI), aux graphiques 2D, à l'impression, ainsi qu'à différentes tâches associées (*Cut&Paste*, accessibilité, etc.).
- ✂ Ces librairies contiennent les éléments de base qui servent à écrire des applications Java qui interagissent avec l'utilisateur (applications de type *client* ou *desktop* par opposition aux applications de type *serveur*).
- ✂ La gestion des interfaces utilisateur graphiques repose principalement sur deux librairies de classes : **AWT** et **Swing**.

# Le package AWT

## **Les éléments d'interfaces graphiques de l'AWT**

Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquels elles vont fonctionner. Cette librairie utilise le système graphique de la plateforme d'exécution (Windows, MacOS, X-Window) pour afficher les objets graphiques. Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.

Les deux classes principales d'AWT sont Component et Container. Chaque type d'objet de l'interface graphique est une classe dérivée de Component. La classe Container, qui hérite de Component est capable de contenir d'autres objets graphiques (tout objet dérivant de Component).

# Le package AWT

## Les composants graphiques de l'AWT

Pour utiliser un composant, il faut créer un nouvel objet représentant le composant et l'ajouter à un conteneur existant grâce à la méthode add().

```
public class FrameButton extends Frame {  
    public FrameButton() {  
        super();  
        init();  
    }  
    public void init() {  
        Button bouton = new Button("Bouton");  
        this.add(bouton);  
    }  
  
    public static void main(String[] args) {  
        new FrameButton().setVisible(true);  
    }  
}
```

# Le package AWT

## Les composants graphiques de l'AWT

- Les étiquettes

Il faut utiliser un objet de la classe `java.awt.Label`

Exemple ( code Java 1.1 ) :

```
1  Label la = new Label( );  
2  la.setText("une etiquette");  
3  // ou Label la = new Label("une etiquette");
```

Il est possible de créer un objet de la classe `java.awt.Label` en précisant l'alignement du texte

Exemple ( code Java 1.1 ) :

```
1  Label la = new Label("etiquette", Label.RIGHT);
```

Le texte à afficher et l'alignement peuvent être modifiés dynamiquement lors de l'exécution :

Exemple ( code Java 1.1 ) :

```
1  la.setText("nouveau texte");  
2  la.setAlignment(Label.LEFT);
```

# Le package AWT

## Les composants graphiques de l'AWT

- Les boutons

Il faut utiliser un objet de la classe `java.awt.Button`

Exemple ( code Java 1.1 ) :

```
1  Button bouton = new Button();  
2  bouton.setLabel("bouton");  
3  // ou Button bouton = new Button("bouton");
```

Le libellé du bouton peut être modifié dynamiquement grâce à la méthode `setLabel()` :

Exemple ( code Java 1.1 ) :

```
1  bouton.setLabel("nouveau libellé");
```



# Le package AWT

## Les composants graphiques de l'AWT

- Les panneaux

Les panneaux sont des conteneurs qui permettent de rassembler des composants et de les positionner grâce à un gestionnaire de présentation. Il faut utiliser un objet de la classe `java.awt.Panel`.

Par défaut le gestionnaire de présentation d'un panel est de type `FlowLayout`.

Exemple ( code Java 1.1 ) :

```
1 | Panel p = new Panel();
```

L'ajout d'un composant au panel se fait grâce à la méthode `add()`.

Exemple ( code Java 1.1 ) :

```
1 | p.add(new Button("bouton"));
```

# Le package AWT

## Les composants graphiques de l'AWT

- Les listes déroulantes (combobox)

Il faut utiliser un objet de la classe `java.awt.Choice`

Plusieurs méthodes permettent la gestion des sélections

Cette classe ne possède qu'un seul constructeur sans paramètres.

Exemple ( code Java 1.1 ) :

```
1 | Choice maCombo = new Choice();
```

Les méthodes `add()` et `addItem()` permettent d'ajouter des éléments à la combobox.

Exemple ( code Java 1.1 ) :

```
1 | maCombo.addItem("element 1");  
2 | // ou maCombo.add("element 2");
```

# Le package AWT

## Les composants graphiques de l'AWT

- La classe TextComponent

La classe TextComponent est la classe mère des classes qui permettent l'édition de texte : TextArea et TextField. Elle définit un certain nombre de méthodes dont ces classes héritent.

Méthodes	Rôle
String getSelectedText( );	Renvoie le texte sélectionné
int getSelectionStart( );	Renvoie la position de début de sélection
int getSelectionEnd( );	Renvoie la position de fin de sélection
String getText( );	Renvoie le texte contenu dans l'objet
boolean isEditable( );	Retourne un booléen indiquant si le texte est modifiable
void select(int start, int end );	Sélection des caractères situés entre start et end
void selectAll( );	Sélection de tout le texte
void setEditable(boolean b);	Autoriser ou interdire la modification du texte
void setText(String s );	Définir un nouveau texte

# Le package AWT

## Les composants graphiques de l'AWT

- Les champs de texte

Il faut déclarer un objet de la classe `java.awt.TextField`

- Les zones de texte multilignes

Il faut déclarer un objet de la classe `java.awt.TextArea`

- Les listes

Il faut déclarer un objet de la classe `java.awt.List`.

- Les cases à cocher

Il faut déclarer un objet de la classe `java.awt.Checkbox`

# Le package AWT

## Les conteneurs de l'AWT

Les conteneurs sont des objets graphiques qui peuvent contenir d'autres objets graphiques, incluant éventuellement des conteneurs. Ils héritent de la classe Container. Un composant graphique doit toujours être incorporé dans un conteneur :

Conteneur	Rôle
Panel	conteneur sans fenêtre propre. Utile pour ordonner les contrôles
Window	fenêtre principale sans cadre ni menu. Les objets descendants de cette classe peuvent servir à implémenter des menus
Dialog (descendant de Window)	réaliser des boîtes de dialogue simples
Frame (descendant de Window)	classe de fenêtre complètement fonctionnelle
Applet (descendant de Panel)	pas de menu. Pas de boîte de dialogue sans être incorporée dans une classe Frame.

L'insertion de composant dans un conteneur se fait grâce à la méthode `add(Component)` de la classe Container.

# Le package AWT

## Composants Swing

Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API dont le but est similaire à celui de l'API AWT mais dont les modes de fonctionnement et d'utilisation sont complètement différents. Swing a été intégré au JDK depuis sa version 1.2. Cette bibliothèque existe séparément. pour le JDK 1.1.

Les composants Swing forment une nouvelle hiérarchie parallèle à celle de l'AWT. L'ancêtre de cette hiérarchie est le composant JComponent. Presque tous ses composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow. Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute.

# Le package Swing

## Composants Swing

- Alors que Swing est une collection de composants qui ont la capacité de développer des objets d'interface graphique (GUI) indépendamment de la plate-forme, les composants AWT dépendent de la plate-forme et fonctionnent différemment sur différentes plates-formes.
- En Java, on distingue des **composants légers** (*Lightweight*) et des **composants lourds** (*Heavyweight*).
  - Les composants lourds sont basés sur du code natif de la plate-forme d'exécution (*Peer Component*).
  - Les composants légers sont entièrement écrits en Java et sont donc indépendants de la plate-forme d'exécution.
- Sauf quelques composants de haut-niveau, pratiquement tous les composants Swing sont des composants légers (contrairement à AWT).

# Le package Swing

## Composants Swing

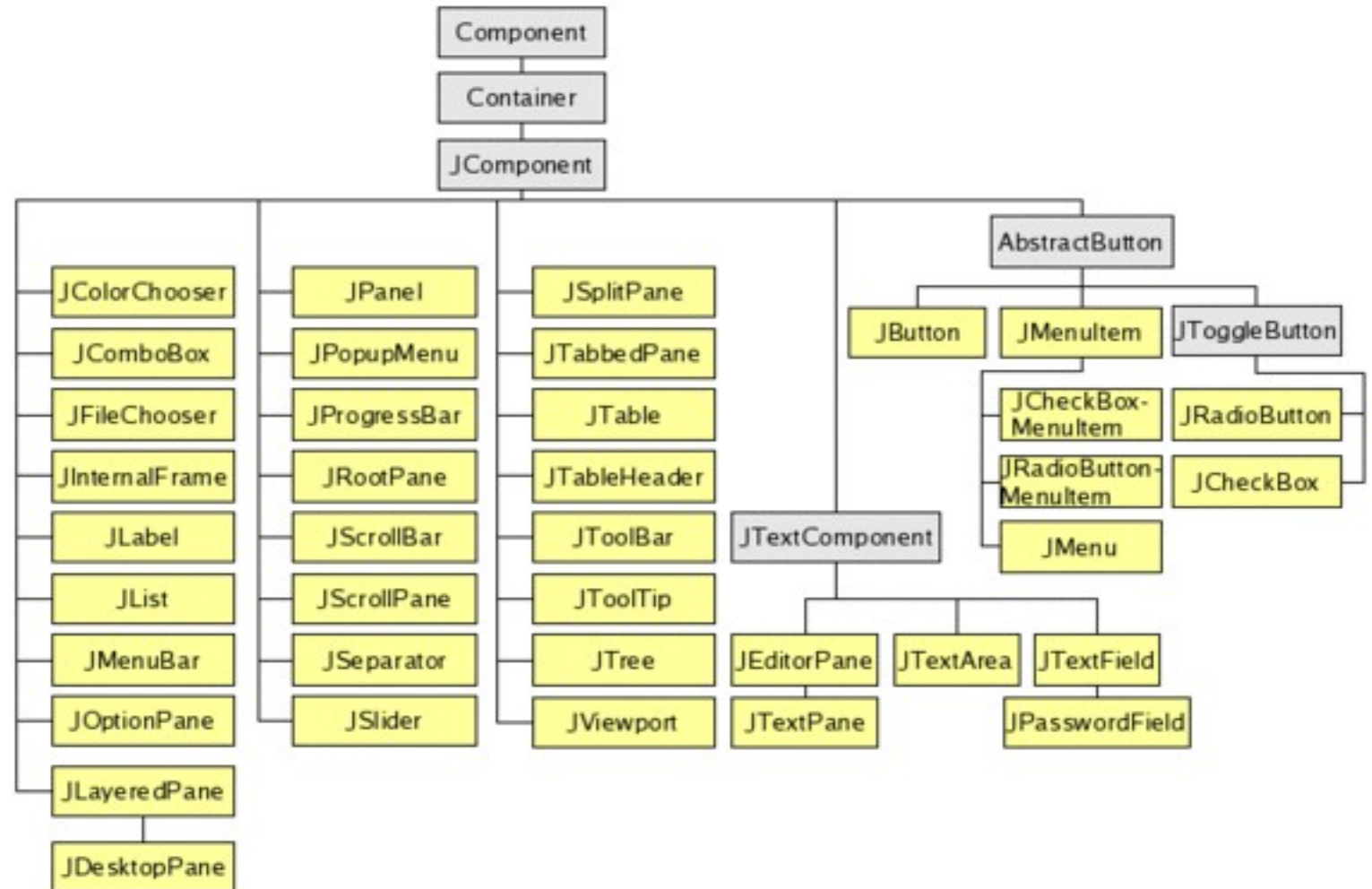
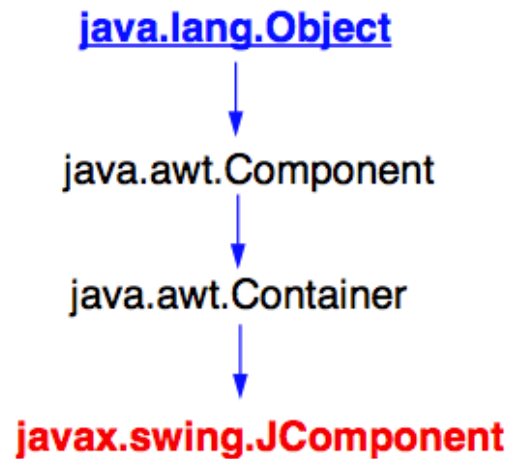
### Différence entre Swing et AWT

AWT	Swing
Les composants AWT dépendent de la plate-forme.	Les composants Java swing ne dépendent pas de la plateforme. Ils sont purement scriptés en Java.
Lourd.	Légers, les composants sont conçus pour être au TOP des composants AWT et effectuer leurs tâches. Ces composants sont généralement légers car ils n'ont pas besoin d'objets OS natifs pour implémenter leurs fonctionnalités.
Les composants AWT ne prennent pas en charge 'look & feel' (aspects et comportements).	Les composants Java swing offrent une prise en charge de 'look & feel' (aspects et comportements). L'API Java swing contient des classes telles que <code>Jbutton</code> , <code>JtextField</code> , <code>JradioButton</code> , <code>Jcheckbox</code> , <code>JtextArea</code> , <code>JMenu</code> , <code>JcolorChooser</code> , etc. qui ajoutent des fonctionnalités supplémentaires aux objets.
Les composants AWT sont moins nombreux que les composants Swing.	Les composants Java Swing sont bien plus nombreux.
Les composants AWT en Java ne suivent pas l'architecture MVC (Model View Controller)	Les composants Swing en Java suivent le modèle MVC (Model View Controller).
AWT signifie Abstract Windows Toolkit	Les composants Swing en Java sont également appelés JFC (Java Foundation Classes).



# Le package Swing

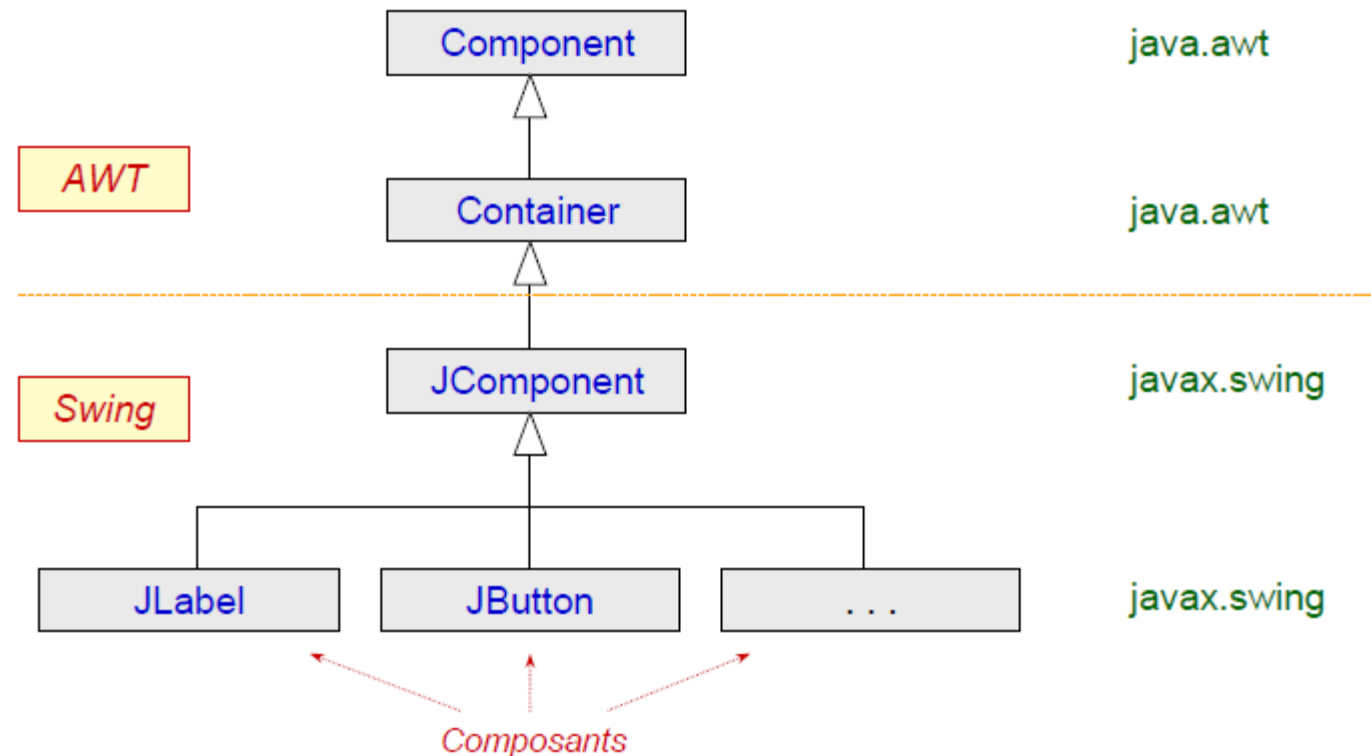
## Composants Swing



# Le package Swing

## Classes de base des composants

Pratiquement tous les composants Swing héritent de la classe **JComponent** qui elle-même hérite des classes AWT **Container** et **Component**.



# Le package Swing

## Classes de base des composants

- ✂ Par héritage, les composants Swing sont donc également des composants AWT (car la classe **Component** est une super-classe de **JComponent**).
- ✂ Les classes de la plupart des composants Swing commencent par la lettre '**J**' et se distinguent ainsi des composants AWT correspondants :
  - Composant '*Bouton*' AWT : **Button**
  - Composant '*Bouton*' Swing : **JButton**
- Il est possible de **créer de nouveaux composants** visuels ou d'étendre (spécialiser) ceux qui existent. Ce travail nécessite naturellement une bonne connaissance de l'architecture de la librairie Swing y compris de certaines classes de bas-niveau afin que les nouveaux composants soient créés dans le même esprit et s'intègrent avec les autres (interopérabilité).
- ✂ Les pages qui suivent donnent un bref aperçu (tour d'horizon) des principaux composants qui sont disponibles dans la librairie Swing.

# Le package Swing

## Composants Swing

<b>JComponent</b>	La racine de la hiérarchie de composants de Swing. Ajoute des fonctionnalités communes à Swing comme par exemple les bulles d'aide ( <i>Tooltips</i> ).
<b>JLabel</b>	Un composant simple (en affichage seul) qui affiche du texte, une image ou les deux.
<b>TextField</b>	Un composant pour afficher, saisir et éditer une ligne de texte simple. Basé sur <b>TextComponent</b> .
<b>PasswordField</b>	Un champ de saisie de texte pour des données confidentielles, comme des mots de passe. N'affiche pas le texte durant la saisie. Basé sur <b>TextField</b> .
<b>FormattedTextField</b>	Un champ de texte formaté. Permet la saisie d'informations qui doivent respecter un certain format (dates, nombres, n° d'article, etc.) Différents formateurs peuvent être associés. Basé sur <b>TextField</b> .

# Le package Swing

## Composants Swing

<b>JButton</b>	Un bouton-poussoir qui peut afficher du texte, des images ou les deux. Sert principalement à déclencher des actions. La classe <b>AbstractButton</b> est la classe de base de tous les boutons.
<b>JToggleButton</b>	Bouton à deux états (bistable). Composant père des composants <b>JCheckBox</b> et de <b>JRadioButton</b> .
<b>JRadioButton</b>	Un bouton à cocher (bouton radio) servant à afficher des choix mutuellement exclusifs. La classe <b>ButtonGroup</b> permet de grouper des boutons radio.
<b>JRadioButtonMenuItem</b>	Un bouton radio à utiliser dans les menus.
<b>JCheckBox</b>	Une case à cocher servant à afficher des choix non mutuellement exclusifs.
<b>JCheckBoxMenuItem</b>	Une case à cocher à utiliser dans les menus.

# Le package Swing

## Composants Swing

<b>JMenuBar</b>	Un composant qui affiche un ensemble de menus déroulants.
<b>JMenu</b>	Un menu déroulant dans un <b>JMenuBar</b> ou comme sous-menu.
<b>JPopupMenu</b>	Une fenêtre qui s'ouvre pour afficher une menu. Employé par <b>JMenu</b> et pour les menus contextuels indépendants.
<b>JMenuItem</b>	Un élément sélectionnable dans un menu (option).
<b>JList</b>	Un composant qui affiche une liste de choix sélectionnables. Les choix sont généralement représentés par des chaînes de caractères ou des images mais d'autres objets sont possibles.
<b>JComboBox</b>	Une combinaison de champs de saisie de texte et d'une liste déroulante de choix. L'utilisateur peut entrer une sélection ou choisir un élément dans la liste déroulante.
<b>JSpinner</b>	Un champ de saisie d'informations ordonnées permettant de passer en revue les valeurs possibles à l'aide de deux boutons de défilement (ou à l'aide du clavier). Affichage sur une seule ligne (contrairement à <b>JComboBox</b> ).

# Le package Swing

## Composants Swing

<b>JToolTip</b>	Une fenêtre légère qui affiche une documentation simple ou une astuce quant le pointeur de souris reste au-dessus d'un composant (bulle d'aide).
<b>JTextComponent</b>	Le composant racine d'un système d'affichage et d'édition de texte puissant et extrêmement personnalisable. Fait partie du paquetage <code>javax.swing.text</code> .
<b>JTextArea</b>	Un composant servant à afficher et à éditer plusieurs lignes de texte simple. Basé sur <code>JTextComponent</code> .
<b>JEditorPane</b>	Un éditeur de texte puissant, configurable via un objet <code>EditorKit</code> , avec des instances prédéfinies pour afficher et éditer du texte au format HTML et RTF.
<b>JTextPane</b>	Une sous-classe de <code>JEditorPane</code> servant à afficher et éditer un texte formaté qui n'est pas au format RTF ou HTML. Permet d'ajouter une fonctionnalité de traitement de texte simple à une application.
<b>JScrollBar</b>	Un ascenseur de défilement horizontal ou vertical.
<b>JProgressBar</b>	Un composant qui affiche la progression d'une opération.



# Le package Swing

## Composants Swing

<b>JToolBar</b>	Un composant qui affiche un ensemble d'outils ou d'actions sélectionnables par l'utilisateur.
<b>JSeparator</b>	Un composant simple qui affiche une ligne horizontale ou verticale. Utilisé pour diviser visuellement les interfaces complexes en sections.
<b>JSlider</b>	Un composant qui simule un curseur que l'utilisateur peut manipuler pour saisir et gérer une valeur numérique.
<b>JOptionPane</b>	Un composant servant à afficher des boîtes de dialogue simples. Comporte des méthodes statiques utiles pour afficher des types de dialogues communs (confirmation, avertissement, etc).
<b>JColorChooser</b>	Un composant complexe qui permet à l'utilisateur de sélectionner une couleur depuis un ou plusieurs espaces de couleurs. Employé en conjonction avec le paquetage <code>javax.swing.colorchooser</code> .
<b>JFileChooser</b>	Un composant complexe qui permet à l'utilisateur de sélectionner un fichier ou un répertoire avec possibilité de filtrage. Employé en conjonction avec le paquetage <code>javax.swing.filechooser</code> .



# Le package Swing

## Composants Swing

<b>JTable</b>	Un composant complexe et puissant servant à afficher des tableaux et à éditer leur contenu. Typiquement utilisé pour afficher des chaînes de caractères, mais peut être personnalisé pour accepter, dans ses cellules, n'importe quel type de données. Utilisé en conjonction avec le paquetage <code>javax.swing.table</code> .
<b>JTree</b>	Un composant complexe et puissant servant à l'affichage de données structurées en arborescence. Les valeurs des données sont typiquement des chaînes de caractères, mais le composant peut être personnalisé pour afficher n'importe quel type de données. Utilisé en conjonction avec le paquetage <code>javax.swing.tree</code> .

Remarque : Les composants `JMenuBar` et `JPopupMenu` peuvent être considérés comme des conteneurs (que nous verrons plus loin).  
En raison de leur usage spécialisé, ils ont malgré tout été placés dans la liste des composants visuels élémentaires (tableaux précédentes).

# Le package Swing

## Composants Swing

La classe de base d'une application est la classe JFrame. Son rôle est équivalent à la classe Frame de l'AWT et elle s'utilise de la même façon.

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

public class test extends JFrame {

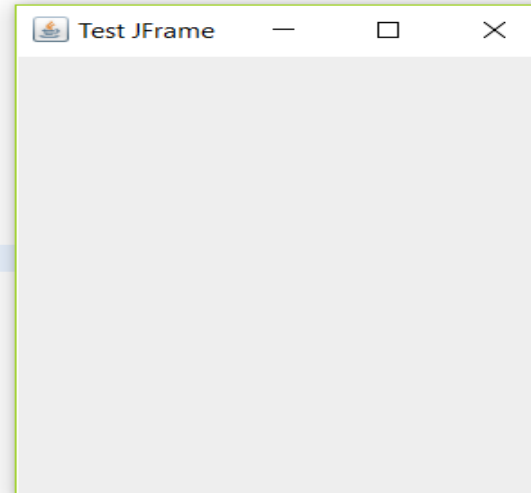
    public test() {
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Pour fermer la fenêtre
        setTitle("Test JFrame");             // Le titre de la fenêtre
        pack();                               // calcul récursif des positions et des tailles
        setSize(250, 300);                   // La taille de la fenêtre
        setVisible(true);                    // fait apparaître la fenêtre
    }

    public static void main(String[] args) {

        test t=new test();

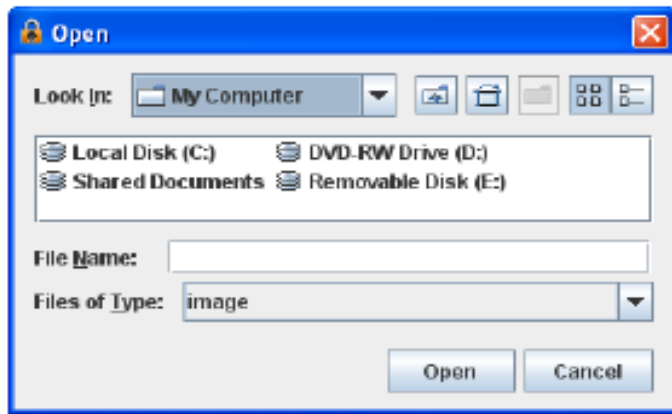
    }

}
```



# Le package Swing

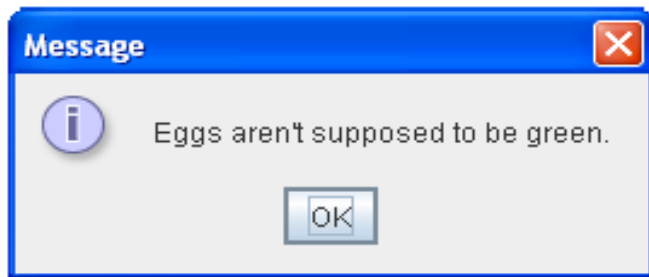
## Boîtes de dialogue prédéfinies



**JFileChooser**



**JColorChooser**



**JOptionPane** (multiples variantes)

## Particularité

- peuvent être créés :
  - comme composants internes
  - ou comme boîtes de dialogue

# Le package Swing

## Boîtes de dialogue prédéfinies

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

public class test extends JDialog {

    public test() {
        setTitle("Test JDialog");
        pack();
        setSize(250, 300);
        setVisible(true);
    }

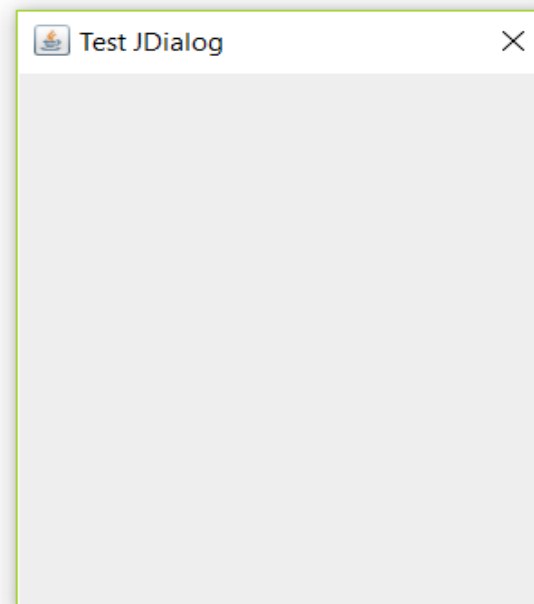
    public static void main(String[] args) {

        test t=new test();

    }

}
```

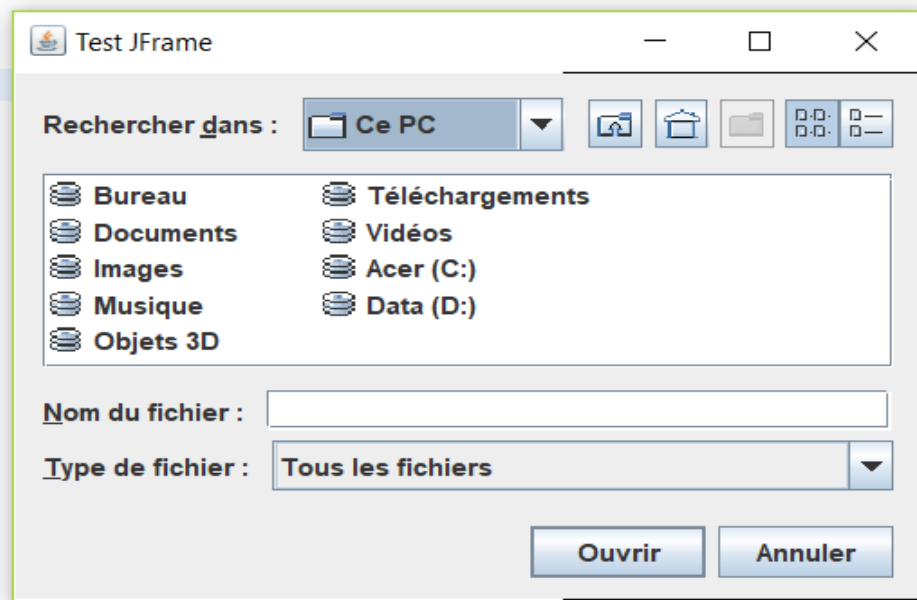
- La méthode pack() agit en étroite collaboration avec le layout manager et permet à chaque contrôle de garder, dans un premier temps sa taille optimale. Une fois que tous les contrôles ont leur taille optimale, pack() utilise ces informations pour positionner les contrôles. pack() calcule ensuite la taille de la fenêtre. L'appel à pack() doit se faire à l'intérieur du constructeur de fenêtre après insertion de tous les contrôles.



# Le package Swing

## Boîtes de dialogue prédéfinies

```
import java.awt.*;  
import javax.swing.*;  
import javax.swing.filechooser.*;  
  
public class test extends JFrame {  
  
    public test() {  
  
        JFileChooser jfc = new JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());  
  
        //JFrame  
        add(jfc);  
        setTitle("Test JFrame");  
        pack();  
        setSize(250, 300);  
        setVisible(true);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
    }  
    public static void main(String[] args) {  
  
        test t=new test();  
    }  
}
```



# Le package Swing

- **Composants Swing**

Il existe des composants Swing équivalents pour chacun des composants AWT avec des constructeurs semblables. De nombreux constructeurs acceptent comme argument un objet de type Icon, qui représente une petite image généralement stockée au format Gif.

- Les étiquettes : la classe JLabel
- Les panneaux : la classe JPanel
- Les boutons : la classe JButton
- Les cases à cocher : la classe JCheckBox
- Les boutons radio : la classe JRadioButton
- Les onglets : La classe JTabbedPane

# Le package Swing

- Composants Swing

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

public class test extends JFrame {

    public test() {

        //Panel
        JPanel p=new JPanel();
        this.add(p);
        //text
        JTextField tf= new JTextField(15);
        JTextArea ta= new JTextArea(10,10);
        p.add(ta);
        p.add(tf);

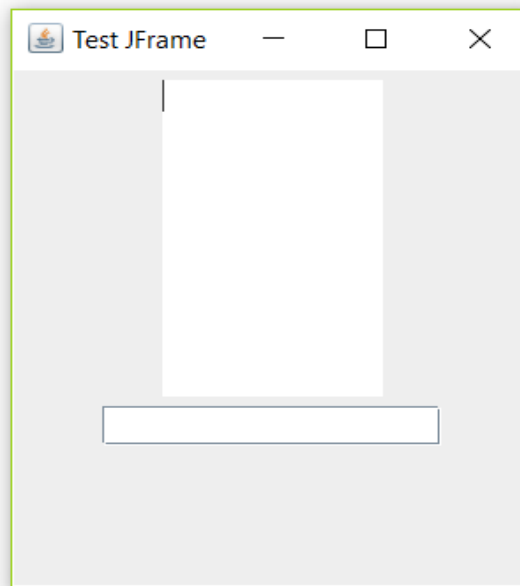
        //JFrame
        setTitle("Test JFrame");
        pack();
        setSize(250, 300);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

    }

    public static void main(String[] args) {

        test t=new test();

    }
}
```



# Le package Swing

- **Composants Swing**

- **Les menus**

Les menus de Swing proposent certaines caractéristiques intéressantes en plus de celles proposées par un menu standard :

- les éléments de menu peuvent contenir une icône
    - les éléments de menu peuvent être de type bouton radio ou case à cocher
    - les éléments de menu peuvent avoir des raccourcis clavier (accelerators)
    - Les menus sont mis en oeuvre dans Swing avec un ensemble de classe :

JMenuBar : encapsule une barre de menus

JMenu : encapsule un menu

JMenuItem : encapsule un élément d'un menu

JCheckBoxMenuItem : encapsule un élément d'un menu sous la forme d'une case à cocher

JRadioButtonMenuItem : encapsule un élément d'un menu sous la forme d'un bouton radio

JSeparator : encapsule un élément d'un menu sous la forme d'un séparateur

JPopupMenu : encapsule un menu contextuel

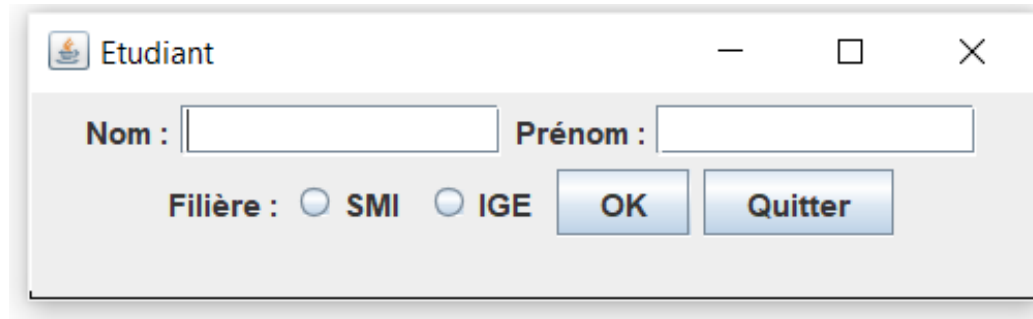
Toutes ces classes héritent de façon directe ou indirecte de la classe JComponent.



# Le package Swing

## Exercice 1

Écrire la classe correspondante à la figure suivante :



The image shows a Java Swing window titled "Etudiant". It contains two text input fields labeled "Nom :" and "Prénom :". Below these, there is a label "Filière :" followed by two radio buttons, one labeled "SMI" and one labeled "IGE". At the bottom right of the window are two buttons labeled "OK" and "Quitter". The window has standard OS window controls (minimize, maximize, close) in the title bar.

# Le package Swing

## Exercice 1

```
1 package exercice1_cours;
2
3 import javax.swing.*;
4
5 public class Etudiant extends JFrame{
6     private JLabel l1,l2,l3;
7     private JTextField t1,t2;
8     private JRadioButton rb1,rb2;
9     private ButtonGroup bg;
10    private JButton b1,b2;
11    private JPanel p1;
12
13    public Etudiant() {
14        l1=new JLabel("Nom :");
15        l2=new JLabel("Prénom :");
16        l3=new JLabel("Filière :");
17        rb1=new JRadioButton("SMI");
18        rb2=new JRadioButton("IGE");
19        bg=new ButtonGroup();
20        b1=new JButton("OK");
21        b2=new JButton("Quitter");
22        t2=new JTextField(12);t1=new JTextField(12);
23        p1=new JPanel();
24
25        bg.add(rb1);bg.add(rb2);
26
27        p1.add(l1);p1.add(t1);p1.add(l2);p1.add(t2);
28        p1.add(l3);p1.add(rb1);p1.add(rb2);
29        p1.add(b1);p1.add(b2);
30
31        this.add(p1);
32
33        setDefaultCloseOperation(EXIT_ON_CLOSE);
34        pack();
35        setTitle("Etudiant");
36        setSize(750, 300);
37        setVisible(true);
38    }
39    public static void main(String[] args) {
40        new Etudiant();
41    }
42 }
```

# Le package Swing

## Exercice 2

Ajouter à la figure de l'exercice 1 une barre de menus constituée :

- d'un menu Fichier comportant les options : Ouvrir, Sauvegarder et Fermer,
- d'un menu Edition comportant les options : Copier et Coller.

On ne cherchera pas ici à traiter les actions correspondantes.

Chapitre VI

# Gestion du positionnement

# Gestion du positionnement

## Le dimensionnement des composants

En principe, il est automatique grâce au `LayoutManager`.

Pour donner à un composant une taille donnée, il faut redéfinir la méthode `getPreferredSize()` de la classe `Component`.

La méthode `getPreferredSize()` indique la taille souhaitée mais pas celle imposée. En fonction du `Layout Manager`, le composant pourra ou non imposer sa taille.

# Gestion du positionnement

## Le dimensionnement des composants

- Une autre façon de faire est de se passer des Layout et de placer les composants à la main en indiquant leurs coordonnées et leurs dimensions.
- Pour supprimer le Layout par défaut d'une classe, il faut appeler la méthode `setLayout()` avec comme paramètre `null`.
- Trois méthodes de la classe `Component` permettent de positionner des composants :
  - `setBounds(int x, int y, int largeur, int hauteur)`
  - `setLocation(int x, int y)`
  - `setSize(int largeur, int hauteur)`
- Ces méthodes permettent de placer un composant à la position (x,y) par rapport au conteneur dans lequel il est inclus et d'indiquer sa largeur et sa hauteur.
- Toutefois, les Layout Manager constituent un des facteurs importants de la portabilité des interfaces graphiques notamment en gérant la disposition et le placement des composants après redimensionnement du conteneur.

# Gestion du positionnement

## Le positionnement des composants

- Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique : la mise en forme est dynamique. On peut influencer cette mise en page en utilisant un gestionnaire de mise en page (Layout Manager) qui définit la position de chaque composant inséré. Dans ce cas, la position spécifiée est relative aux autres composants.
- Chaque layout manager implémente l'interface `java.awt.LayoutManager`.
- Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe `FlowLayout` qui est utilisée pour la classe `Panel` et la classe `BorderLayout` pour `Frame` et `Dialog`.
- Pour affecter une nouvelle mise en page, il faut utiliser la méthode `setLayout()` de la classe `Container`.

# Gestion du positionnement

## Le positionnement des composants

- Les layout manager ont 3 avantages :
  1. l'aménagement des composants graphiques est délégué aux layout managers (il est inutile d'utiliser les coordonnées absolues)
  2. en cas de redimensionnement de la fenêtre, les contrôles sont automatiquement agrandis ou réduits
  3. ils permettent une indépendance vis à vis des plate-formes.



# Gestion du positionnement

## La mise en page par flot (FlowLayout)

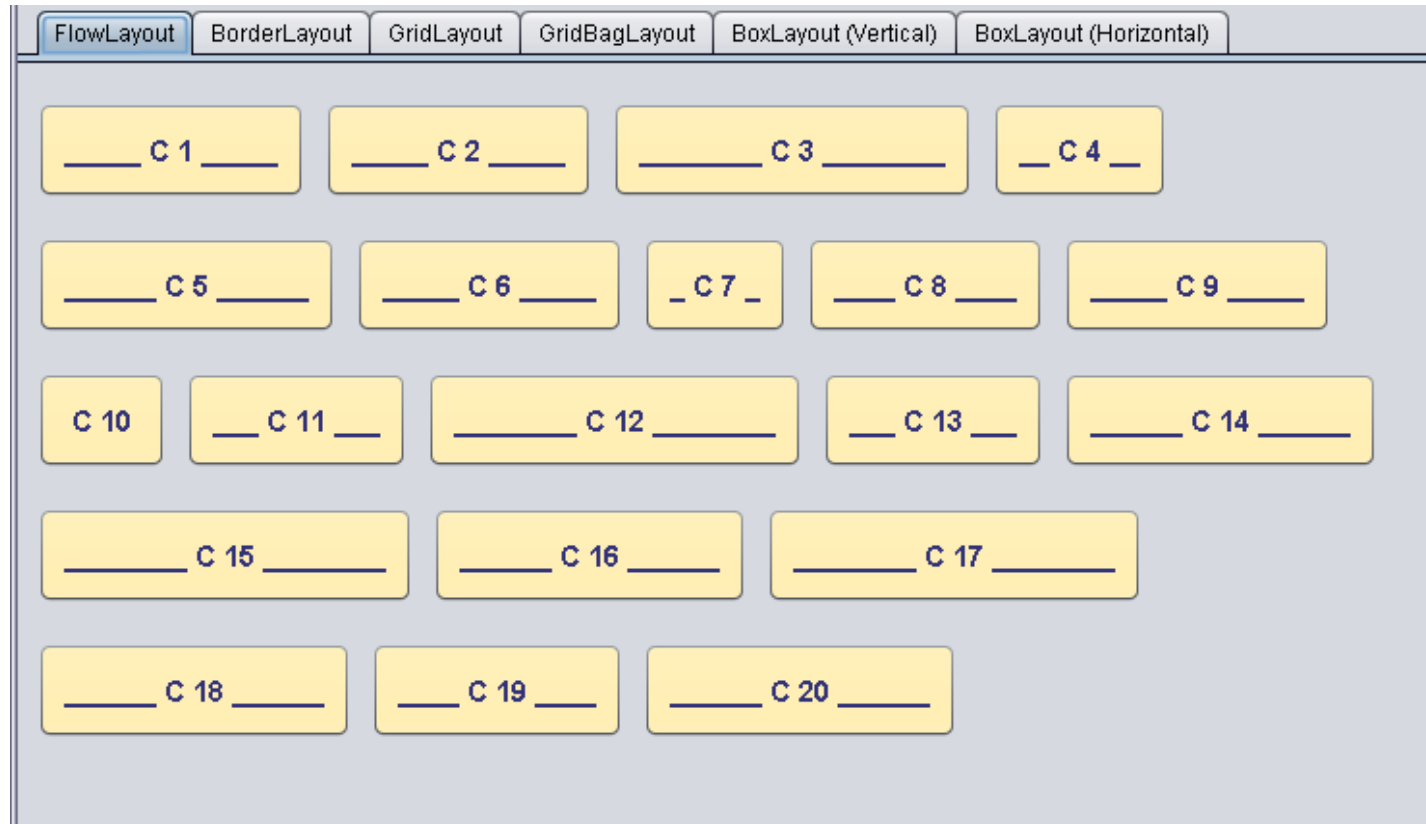
- La classe FlowLayout (mise en page flot) place les composants ligne par ligne de gauche à droite. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Chaque ligne est centrée par défaut. C'est la mise en page par défaut des applets.

Il existe plusieurs constructeurs :

Constructeur	Rôle
FlowLayout( );	
FlowLayout( int align);	Permet de préciser l'alignement des composants dans le conteneur (CENTER, LEFT, RIGHT ... ). Par défaut, align vaut CENTER
FlowLayout( int align, int hgap, int vgap);	Permet de préciser l'alignement et l'espacement horizontal et vertical dont la valeur par défaut est 5.

# Gestion du positionnement

La mise en page par flot (FlowLayout)



# Gestion du positionnement

## La mise en page bordure (BorderLayout)

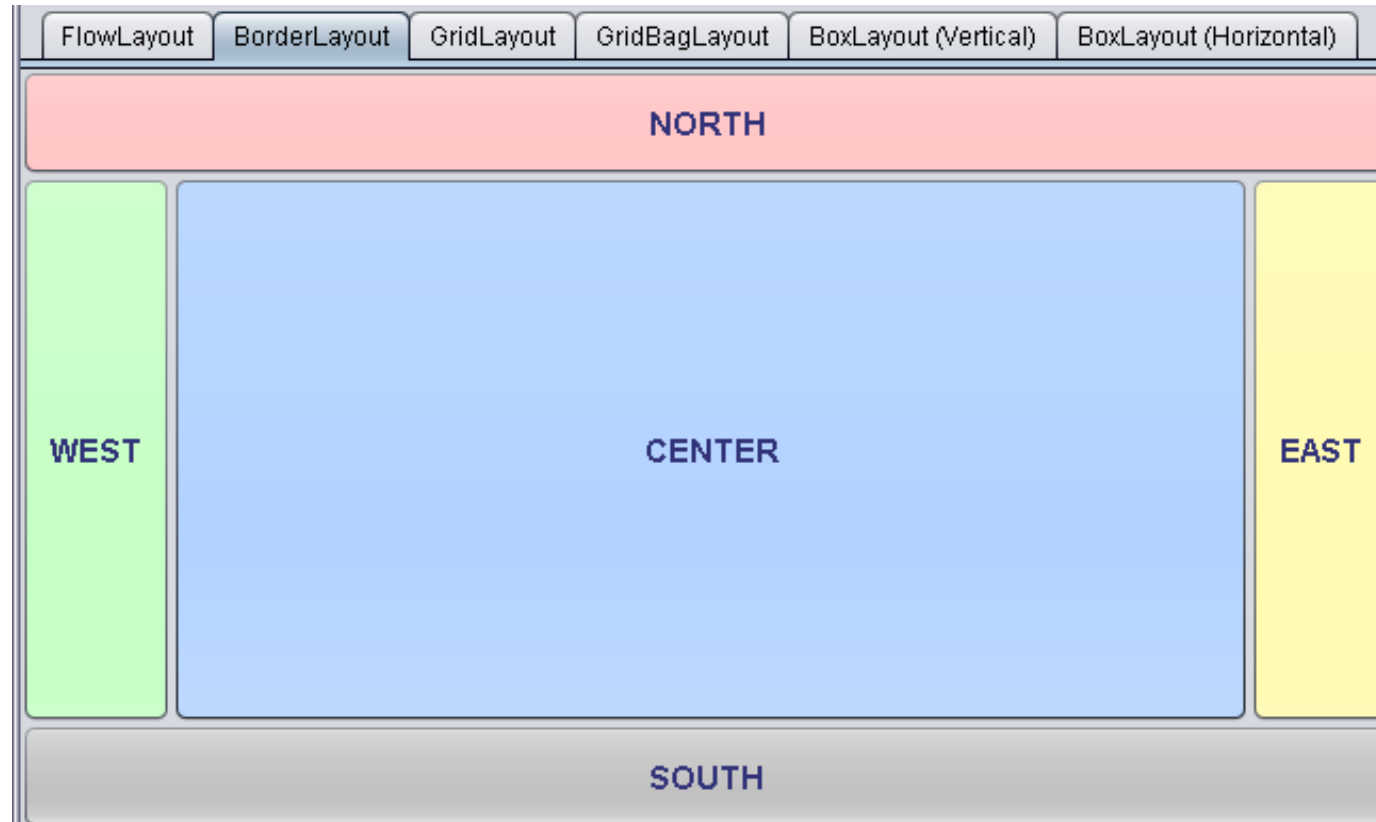
- Avec ce Layout Manager, la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center. On peut librement utiliser une ou plusieurs zones.
- BorderLayout consacre tout l'espace du conteneur aux composants. Le composant du milieu dispose de la place inutilisée par les autres composants.

Il existe plusieurs constructeurs :

Constructeur	Rôle
BorderLayout( )	
BorderLayout (int hgap,int vgap)	Permet de préciser l'espacement horizontal et vertical des composants.

# Gestion du positionnement

La mise en page bordure (BorderLayout)



# Gestion du positionnement

## La mise en page bordure (BorderLayout)

La **position des composants** est déterminée par une contrainte mentionnée lors de l'ajout du composant (paramètre de la méthode **add()**). Elle est définie en utilisant une constante de la classe **BorderLayout** (NORTH, SOUTH, EAST, WEST, CENTER).

Le gestionnaire **BorderLayout** redimensionne les composants de manière à obtenir le comportement suivant :

- Les composants "**nord**" et "**sud**" occupent **toute la largeur** du conteneur
- Les composants "**est**" et "**ouest**" occupent toute **la hauteur qui reste**
- Le composant "**centre**" occupe **toute la place restante**

Les cinq emplacement prévus ne doivent pas forcément être tous occupés (mais il ne peut pas y avoir plus d'un composant par emplacement).

### Propriétés du gestionnaire :

- L'espace (horizontal et vertical) entre les composants peut être défini en gérant les propriétés **hgap** et **vgap** (nombre de pixels entre les composants)  
Par défaut : **hgap**=0 et **vgap**=0.

# Gestion du positionnement

## La mise en page GridLayout

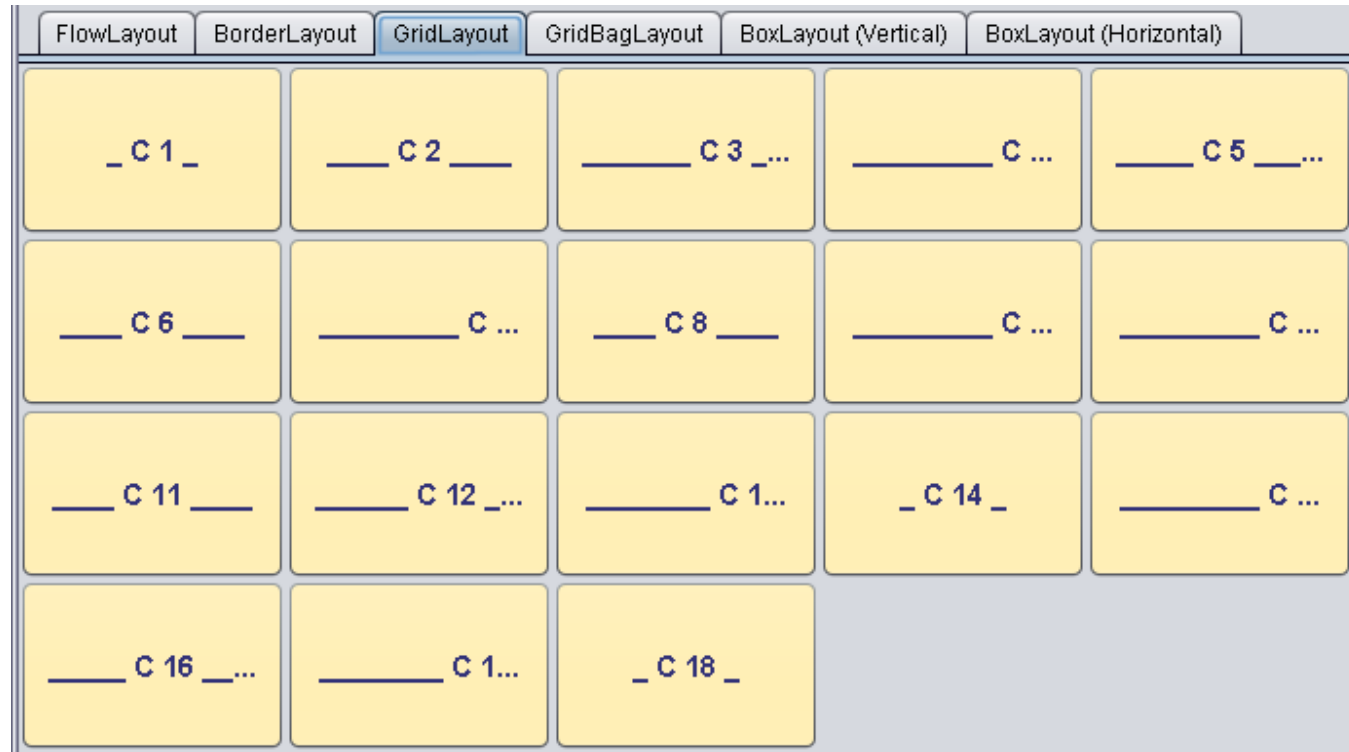
- Ce Layout Manager établit un réseau de cellules identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille.
- Les cellules du quadrillage se remplissent de gauche à droite ou de haut en bas.

Il existe plusieurs constructeurs :

Constructeur	Rôle
<code>GridLayout( int, int );</code>	Les deux premiers entiers spécifient le nombre de lignes ou de colonnes de la grille.
<code>GridLayout( int, int, int, int );</code>	permet de préciser en plus l'espacement horizontal et vertical des composants.

# Gestion du positionnement

## La mise en page GridLayout



# Gestion du positionnement

## La mise en page GridBagLayout

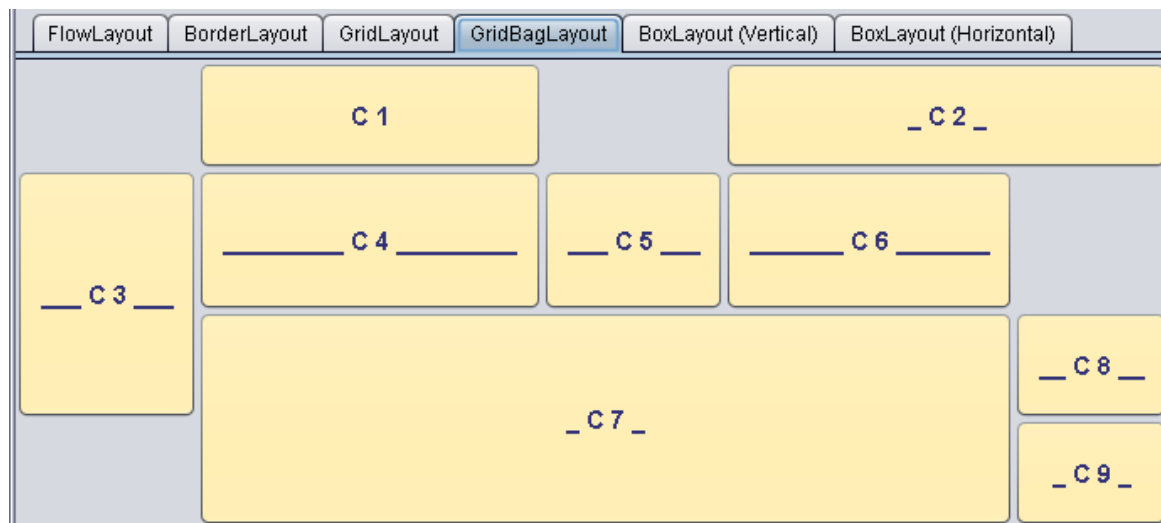
- Ce gestionnaire (grille étendue) est le plus riche en fonctionnalités : le conteneur est divisé en cellules égales mais un composant peut occuper plusieurs cellules de la grille et il est possible de faire une distribution dans des cellules distinctes. Un objet de la classe GridBagConstraints permet de donner les indications de positionnement et de dimension à l'objet GridBagLayout.
- Les lignes et les colonnes prennent naissance au moment où les contrôles sont ajoutés. Chaque contrôle est associé à un objet de la classe GridBagConstraints qui indique l'emplacement voulu pour le contrôle.



# Gestion du positionnement

## La mise en page GridBagLayout

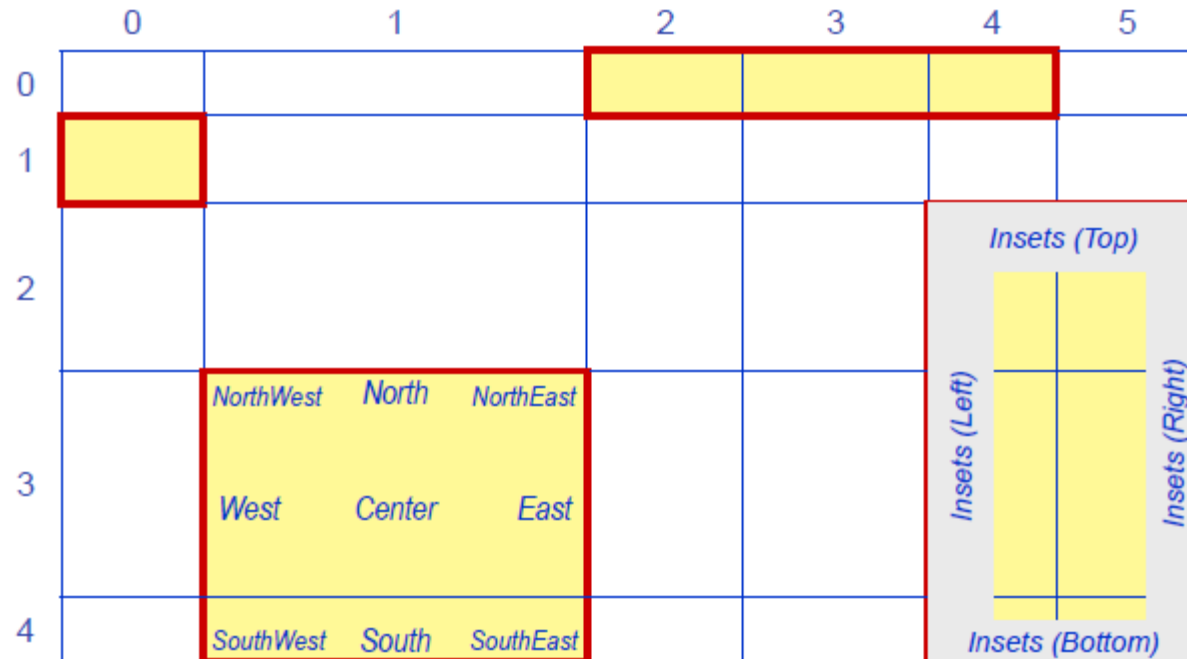
- Arrange les composants dans une grille dont les cellules sont de tailles variables. Un composant peut s'étendre (*span*) sur plusieurs cellules.
- A chaque composant est associé un ensemble de **contraintes**, spécifiées par un objet de type **GridBagConstraints**.
- Permet un contrôle précis de la dimension et du positionnement des composants, notamment quand le conteneur change de taille.



# Gestion du positionnement

## La mise en page GridBagLayout

- Les composants occupent des **zones** rectangulaires qui peuvent être composées d'une ou plusieurs cellules.
- On peut définir le point d'ancrage du composant à l'intérieur de la zone



# Gestion du positionnement

## La mise en page GridBagLayout

### GridBagConstraints

- Les **contraintes** associées à chaque composant enfant définissent :
- Les **coordonnées** (numéro de la colonne, numéro de la ligne) de la zone dans laquelle sera placé le composant (*gridx, gridy*) La zone du composant pouvant occuper plusieurs cellules, on indiquera les coordonnées de la première cellule occupée (en haut à gauche)
- Le **nombre de cellules occupées** (nombre de lignes, nombre de colonnes) par la zone (*gridwidth, gridheight*)
- La **position du composant** (ancrage) à l'intérieur de sa zone : *nord, sud, est, ouest, centre, nord-est, ...* (*anchor*)
- Un **indicateur de remplissage** qui précise si le composant doit s'agrandir pour occuper tout l'espace de la zone si la zone est plus grande que sa taille préférée. On peut distinguer le remplissage horizontal et vertical (*fill*)
- Un **indicateur d'agrandissement** qui précise dans quelle proportion les lignes et les colonnes doivent modifier leurs tailles lorsque le conteneur change de dimension (*weightx, weighty*). La valeur maximale de chaque ligne, respectivement de chaque colonne, est prise en compte.
- Une **marge externe**, en pixels, autour du composant (*insets*)
- Une **marge interne**, en pixels, à ajouter, de chaque côté, à la taille minimale du composant (*ipadx, ipady*)

[ Très rarement utilisé ]

# Gestion du positionnement

## La mise en page GridBagLayout

### GridBagConstraints

Contrainte	Type	Description	Valeurs possibles
<b>gridx</b> <b>gridy</b>	<b>int</b>	Position (n° colonne, ligne) de la première cellule (en haut, à gauche) de la zone associée au composant	Valeur entière $\geq 0$ Par défaut : placé à droite (resp. en dessous) du précédent (RELATIVE)
<b>gridheight</b> <b>gridwidth</b>	<b>int</b>	Nombre de cellules que comprend la zone (nombre de lignes et nombre de colonnes)	Valeur entière $> 0$ Par défaut : 1
<b>anchor</b>	<b>int</b>	Position d'ancrage du composant dans sa zone	CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST Par défaut : CENTER
<b>fill</b>	<b>int</b>	Indicateur de remplissage dans le sens horizontal, vertical, ou les deux (étalement du composant dans la zone)	NONE, HORIZONTAL, VERTICAL, BOTH Par défaut : NONE
<b>weightx</b> <b>weighty</b>	<b>double</b>	Poids de la répartition des espaces additionnels dans les lignes et les colonnes lors du redimensionnement du conteneur. La valeur maximale de chaque ligne resp. de chaque colonne est prise en compte dans le calcul	Valeurs réelles $\geq 0.0$ Par défaut : 0.0 Les valeurs n'ont de significations que relativement les unes par rapport aux autres.
<b>insets</b>	<b>Insets</b>	Marge externe, en pixels, autour du composant (à l'intérieur de sa zone)	Objet de type Insets (valeurs entières pour les marges)
<b>ipadx</b> <b>ipady</b>	<b>int</b>	Marge interne (en x et en y) qui augmente la taille minimale du composant (ajoutée de chaque côté)	Valeurs entières $\geq 0$ Par défaut : 0

# Gestion du positionnement

## La mise en page GridBagLayout GridBagConstraints

→ Quelques remarques additionnelles :

- Les valeurs associées à **weightx** et **weighty** influenceront le redimensionnement des colonnes, respectivement des lignes de la grille. C'est la **valeur maximale** de chaque colonne respectivement de chaque ligne qui est prise en compte dans le calcul.
- La valeur **weightx** d'une colonne est comparée à la somme des valeurs **weightx** de toutes les colonnes de la grille pour déterminer la proportion d'espace additionnel qui lui sera attribuée.
- La valeur **fill** indique si le composant doit s'adapter à la grandeur de sa zone. Le composant peut s'étaler horizontalement, verticalement, dans les deux directions ou ne pas s'étaler du tout (même si la zone s'agrandit).
- Si **weightx** (**weighty**) est égal à zéro pour tous les composants, l'espace additionnel sera ajouté autour de la grille, à gauche et à droite (resp. en haut et en bas). La grille sera ainsi centrée à l'intérieur du conteneur.
- Certaines constantes particulières de la classe **GridBagConstraints** (**RELATIVE**, **REMAINDER**) peuvent être utilisées avec les propriétés **gridx**, **gridy**, **gridwidth**, **gridheight**.

# Gestion du positionnement

## La mise en page GridBagLayout

### GridBagConstraints

```
JPanel leftPanel = new JPanel(); // Création du conteneur
JLabel lbA = new JLabel("Age");
JButton btA = new JButton("Ok");

GridBagLayout gbLayout = new GridBagLayout(); // Layout Manager

leftPanel.setLayout(gbLayout);

GridBagConstraints gbc = new GridBagConstraints(); // Contraintes
gbc.gridx = 0;
gbc.gridy = 0;
gbc.anchor = GridBagConstraints.NORTH;

leftPanel.add(lbA, gbc); // Ajout du composant lbA avec ses contraintes
gbc.gridx = 1;
gbc.anchor = GridBagConstraints.CENTER;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = 1.0;
leftPanel.add(btA, gbc); // Ajout du composant btA avec ses contraintes
```

# Gestion du positionnement

## Autres gestionnaires de disposition

Il existe d'autres gestionnaires spécialisés comme **Cardlayout**, **Boxlayout**, **GroupLayout**, **SpringLayout**...

Il est également possible de travailler **sans gestionnaire** de disposition (sans *Layout Manager*) en enregistrant la référence **null** à la place d'un objet de type **LayoutManager**.

Dans ce cas, la position absolue des composants doit être définie explicitement (x, y) par exemple en utilisant une des méthodes **setLocation()** ou **setBounds()** de **java.awt.Component**.

Sans gestionnaire de disposition, la taille des composants doit être définie à l'aide de la méthode **setSize()**.

Sauf cas particuliers, **ce mode de fonctionnement n'est pas recommandé** (il n'est d'ailleurs pas autorisé et lève une exception dans certains conteneurs).

# Gestion du positionnement

## Exercice 1

Écrire la classe correspondante à la figure suivante :

The image shows a Java Swing window titled "Article". The window has a standard title bar with minimize, maximize, and close buttons. Inside the window, there is a horizontal row of five input fields with labels: "Article :", "Reference :", "Prix HT :", "TVA :", and "Prix TTC :". Below these fields is a large text area labeled "Informations sur l'article :". At the bottom of the window, there are two buttons: "Créer" and "Quitter".



# Gestion du positionnement

## Exercice 1

```
import java.awt.GridLayout;

public class article extends JFrame implements ActionListener{

    private JLabel l1,l2,l3,l4,l5;
    private JTextField t1,t2,t3,t4,t5;
    private JTextArea ta;
    private JButton b1,b2;
    private JPanel p1,p2,p3;

    public article() {
        l1=new JLabel("Article :");
        l2=new JLabel("Reference :");
        l3=new JLabel("Prix HT :");
        l4=new JLabel("TVA :");
        l5=new JLabel("Prix TTC :");
        ta=new JTextArea("Informations sur l'article :",30,30);
        b1=new JButton("Créer");
        b2=new JButton("Quitter");
        t2=new JTextField(7);t1=new JTextField(7);
        t3=new JTextField(7);t4=new JTextField(7);t5=new JTextField(7);
        p1=new JPanel();p2=new JPanel();p3=new JPanel();
```

```
        p1.setLayout(new GridLayout(1, 10));
        p1.add(l1);p1.add(t1);p1.add(l2);p1.add(t2);p1.add(l3);p1.add(t3);p1.add(l4);p1.add(t4);p1.add(l5);p1.add(t5);
        p2.add(ta);
        p3.add(b1);p3.add(b2);

        this.add(p1,"North");
        this.add(p2, "Center");
        this.add(p3, "South");

        b1.addActionListener(this);
        b2.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setTitle("Article");
        setSize(750, 300);
        setVisible(true);
    }
    public static void main(String[] args) { new article(); }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==b1) {
            String S=new String();
            S = "Informations sur l'article :" + "\n Article :" + t1.getText() + "\n Reference :" + t2.getText()+ "\n" + t3.getText()+ "\n" + t4.getText()+ "\n" + t5.getText();
            ta.setText(S);
            t1.setText("");t2.setText("");t3.setText("");t4.setText("");t5.setText("");
        }
        if(e.getSource()==b2) { System.exit(0);}
    }
}
```

Chapitre VII

# Gestion des événements

# Gestion des événements

## Architecture MVC

- L'**architecture MVC** (*Model-View-Controller*) est un **modèle de conception** (*Design Pattern*) très classique qui a été introduit avec le langage *Smalltalk-80*.
- Le principe de base de l'architecture MVC est relativement simple, on divise le système interactif en **trois parties distinctes** :
  - le **modèle** (*Model*) qui offre l'accès et permet la gestion des données (état du système)
  - la **vue** (*View*) qui a pour tâche de présenter les informations (visualisation) et qui participe à la détection de certaines actions de l'utilisateur
  - le **contrôleur** (*Controller*) qui est chargé de réagir aux actions de l'utilisateur (clavier, souris) et à d'autres événements internes et externes

Ce modèle de conception simplifie le développement et la maintenance des applications en répartissant et en découplant les activités dans différents sous-systèmes (plus ou moins) indépendants.

# Gestion des événements

## Architecture MVC

- **Le Modèle** (*Model*) est responsable de la gestion de l'**état du système** (son contenu actuel, la valeur de ses données).
- Il offre également les **méthodes** et **fonctions** permettant de gérer, transformer et manipuler ces données (logique "métier" de l'application).
- Le modèle peut informer les vues des changements intervenus dans ses données. La communication de ces changements intervient en général sous la forme d'événements qui seront gérés par des contrôleurs (les vues s'enregistrent auprès du modèle pour être notifiées lors des changements dans les données)
- Les informations gérées par le modèle sont indépendantes de la manière dont elles seront affichées. En fait, le modèle doit pouvoir exister indépendamment de la représentation visuelle des données.

Dans certaines situations (simples) le modèle peut contenir lui-même les données, mais la plupart du temps, il agit comme un intermédiaire (*proxy*) vers les données qui sont stockées dans une base de données ou un serveur d'informations (en *Java*, le modèle est souvent défini par une **interface**).

# Gestion des événements

## Architecture MVC

- **La vue** (**View**) se charge de la **représentation visuelle** des informations.
- La vue utilise les données provenant du modèle pour afficher les informations. La vue doit être informée des modifications intervenues dans certaines données du modèle (celles qui influencent l'affichage).
- Plusieurs vues différentes peuvent utiliser le même modèle (plusieurs représentations possibles d'un même jeu de données).
- La vue intercepte certaines actions de l'utilisateur et les transmet au contrôleur pour qu'il les traite (souris, événements clavier, ...).
- Le contrôleur peut également modifier la vue en réaction à certaines actions de l'utilisateur (par exemple afficher une nouvelle fenêtre).

La représentation visuelle des informations affichées peut dépendre du *Look-and-Feel* adopté (ou imposé) et peut varier d'un système d'exploitation à l'autre. L'utilisateur peut parfois modifier lui même la présentation des informations en choisissant par exemple un thème.

# Gestion des événements

## Architecture MVC

- **Le contrôleur** (*Controller*) est chargé de réagir aux différents événements qui peuvent survenir.
- Les **événements** sont constitués soit par des **actions de l'utilisateur** (presser sur une touche, cliquer sur un bouton, fermer une fenêtre, ...) ou par des **directives venant du programme** lui-même (un changement intervenu dans un autre composant, l'écoulement d'un certain temps, etc...).
- Le contrôleur **définit le comportement** de l'application (comment elle réagit aux sollicitations).
- Dans les applications simples, le contrôleur gère la synchronisation entre la vue et le modèle (rôle de chef d'orchestre).
- Le contrôleur est informé des événements qui doivent être traités et sait d'où ils proviennent.
- Le contrôleur peut agir sur la vue en modifiant les éléments affichés.

Le contrôleur peut également, si nécessaire, modifier le modèle en réaction à certains événements.

# Gestion des événements

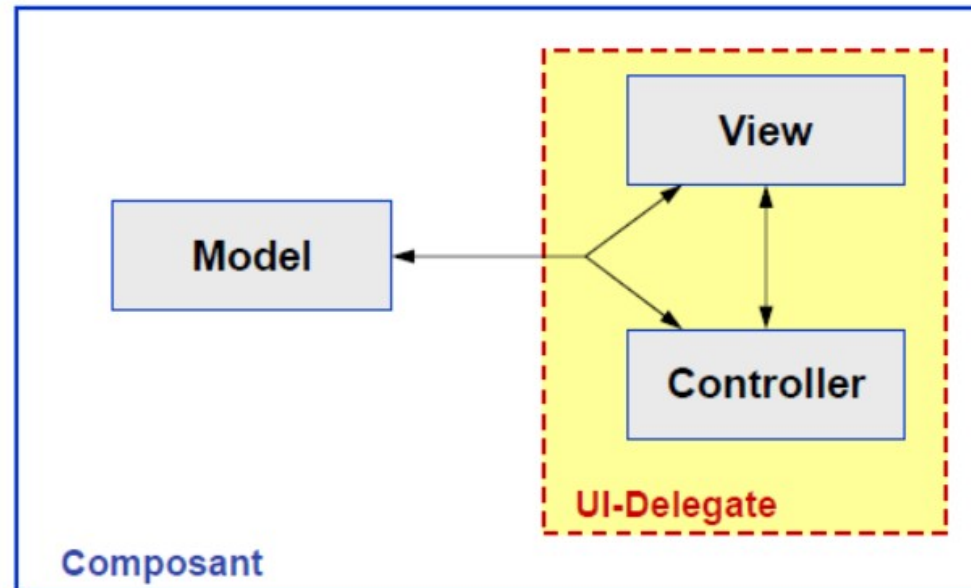
## Importance de l'architecture MVC

- Pour le développement d'applications Java, l'**architecture MVC** est intéressante à plusieurs titres.
- D'une part, c'est un **modèle de conception** (ou, plus exactement, un ensemble de modèles de conception) qu'il est recommandé d'utiliser pour le développement des applications interactives car il offre des principes de découpage du code (modularisation) qui visent à **minimiser les couplages** entre les différents éléments qui constituent le programme.
- Ce découplage entre les modules minimise les dépendances et favorise ainsi la maintenance des applications (on peut modifier un élément du système sans que tous les autres en soient affectés).
- D'autre part, la **librairie Swing** est elle-même basée sur une variante de cette architecture.
- Il est donc important de connaître l'architecture MVC pour comprendre et exploiter au mieux les différents éléments de la librairie Swing (paramétrer et personnaliser les composants, gérer les événements, ...).

# Gestion des événements

## MVC et Swing

- La librairie Swing utilise une **variante** légèrement simplifiée de l'architecture MVC appelée **Model-Delegate** (Modèle-Délégué).
- Cette variante **combine** dans un seul élément **la vue et le contrôleur**.
- Cet élément combiné est appelé **UI-Delegate**.





# Gestion des événements

## MVC et Swing

- En résumé, chaque composant Swing contient un modèle et un UI-Delegate.
- Le **modèle** est responsable de maintenir les informations associées au composant (son état, ses données).
- Le **UI-Delegate** est responsable de la représentation visuelle du composant ainsi que de la réaction du composant aux différents événements qui peuvent l'affecter (actions de l'utilisateur ou instructions provenant du programme).

Cette séparation permet :

- D'avoir **différentes représentations d'un même modèle**. Les composants **JSlider**, **JProgressBar** et **JScrollBar** partagent, par exemple, le même modèle : **BoundedRangeModel**
- De pouvoir **modifier la représentation** (*Look-and-Feel*) d'un composant sans affecter ses données (*Pluggable Look-and-Feel*). La représentation peut être modifiée dynamiquement (durant l'exécution de l'application)

# Gestion des événements

## Les événements

Les **événements** sont constitués par :

- Des **actions de la part de l'utilisateur** (un clic de souris, la pression sur une touche du clavier, le déplacement d'une fenêtre, etc.)
- Des **changements provoqués par le programme** lui-même ou par un sous-système externe (modification des propriétés d'un composant, informations provenant du réseau, échéance d'une temporisation, etc.)

- Les **événements** (groupés par genre) sont représentés par différentes classes qui ont toutes comme classe parente la classe **EventObject** (du package **java.util**).
- Les événements AWT sont représentés par des objets des classes qui se trouvent dans le package **java.awt.event**.
- La librairie Swing utilise également ces classes et en définit d'autres dans le package **javax.swing.event**.

# Gestion des événements

## Les événements

- Certains composants **génèrent** des événements; ce sont des **sources d'événements**.

Un objet qui **désire être notifié** (informé) lorsqu'un événement survient est un **récepteur d'événement** (***Event Listener***).

On pourrait également parler d'**intercepteur d'événement**, d'**auditeur d'événement**, de **traite événement**, etc.

# Gestion des événements

## Gestion des événements

**Préparation** (lors de l'initialisation du programme)

- Création d'un récepteur d'événement (contrôleur) ...*Controller*
  - └─ Instanciation d'une classe implémentant l'interface **evTypeListener**
- Enregistrement du récepteur d'événement auprès du composant qui est la source de l'événement (un bouton par exemple)
  - └─ Invocation d'une méthode **addevTypeListener()**

**Lorsque l'événement survient** (clic sur un bouton par exemple)

- Un objet "événement" est automatiquement créé par le composant source d'événement
  - └─ L'objet créé est une instance de la classe **evTypeEvent**
- La méthode du contrôleur (*evController*) associée à l'événement est automatiquement appelée par le composant source d'événement
  - └─ L'objet "événement" est toujours passé en paramètre à cette méthode
  - └─ Cette méthode se charge du traitement de l'événement

# Gestion des événements

## Gestion des événements

**Préparation** (lors de l'initialisation du programme)

≡ Création d'un récepteur d'événement (contrôleur) ...*Controller*

```
public class ClickController implements ActionListener {  
    ...  
    public void actionPerformed(ActionEvent event) {  
        ... // Traitement de l'événement  
    }  
}  
btController = new ClickController(...);
```

≡ Enregistrement du récepteur d'événement auprès du composant qui est la source de l'événement (un bouton par exemple)

```
myButton.addActionListener(btController);
```

**Lorsque l'événement survient** (clic sur le bouton)

≡ Le bouton créera une instance **event** de la classe **ActionEvent**

≡ Le bouton invoquera la méthode **actionPerformed(event)**

# Gestion des événements

## Objets représentant les événements

✂ Les **objets de type événement** (*evTypeEvent*) disposent tous d'une **méthode générale**:

- **getSource()**: Renvoie l'objet qui a généré ou déclenché l'événement
- Les différentes sous-classes d'événements définissent également des **méthodes spécialisées** qui dépendent de l'événement considéré. Quand on reçoit un **MouseEvent**, on peut donc connaître l'emplacement où l'utilisateur a cliqué, avec quel bouton, combien de fois, quels modificateurs clavier étaient simultanément activés et quand (à quel moment) il a cliqué.
- Par exemple **MouseEvent** possède les méthodes **getX()**, **getY()**, **getButton()**, **getClickCount()**, ... et hérite des méthodes **getModifiers()** et **getWhen()** de sa classe parente **InputEvent**.

# Gestion des événements

## Gestion des événements

- La gestion des événements est identique pour tous les composants, et des conventions de nommage régissent les éléments qui interviennent.
- Chaque type d'**événement** *evType* se nomme *evType* **Event** et possède un **listener** (un récepteur) qui est défini par une interface appelée *evType* **Listener** (une sous-classe de `java.util.EventListener`).
- Les interfaces *listener* définissent des **méthodes que la source d'événement invoquera** lorsqu'un type donné d'événement se produira (principe de *Callback*). Ces méthodes listener prennent toujours un **objet événement** comme unique **paramètre**.
- Les sources d'événements doivent mettre à disposition des méthodes permettant aux récepteurs (*listeners*) de **s'inscrire auprès de la source**
- Par convention, ces **méthodes d'inscription** (enregistrement du contrôleur) se nomment **add*evType*Listener()**. Les méthodes de retrait (résiliation, désinscription) se nomment **remove*evType*Listener()**.

# Gestion des événements

## Gestion des événements

- Par exemple, pour la classe **JButton** (source d'événement), le type d'événement est **Action**, le composant génère donc un événement appelé **ActionEvent** lorsque l'utilisateur clique sur le bouton.
- La classe fournit donc deux méthodes **addActionListener()** et **removeActionListener()** qui permettent à un objet (passé en paramètre) de s'inscrire (resp. de résilier l'inscription) comme récepteur (listener) pour l'événement de type **ActionEvent**.
- Si un objet s'intéresse à l'événement **ActionEvent** du bouton, il doit implémenter l'interface **ActionListener**. Cette interface contient une unique méthode **actionPerformed()** qui sera invoquée lorsque le bouton sera pressé.



# Gestion des événements

## Schéma de fonctionnement

- 1) Création d'un composant **JButton** qui génère un événement de type **ActionEvent** (**JButton** est une source d'événement)
  - 2) Création d'un récepteur d'événement qui implémente les méthodes définies dans l'interface **ActionListener** (dans ce cas, il y a une unique méthode appelée **actionPerformed()**). Le traitement de l'événement est réalisé dans le corps de la méthode **actionPerformed()**.
  - 3) Enregistrement du récepteur d'événement (listener) auprès de la source d'événement, le bouton **b1**. Ce mécanisme s'appelle **délégation** : le bouton **b1** délègue la gestion de l'événement **ActionEvent** à l'objet **r**.
- Finalement : Lorsqu'un utilisateur pressera le bouton **b1**, la méthode **actionPerformed()** sera invoquée avec, comme unique argument, l'objet événement **e** (de type **ActionEvent**). Si nécessaire, il sera possible dans le corps de **actionPerformed()** de déterminer l'objet qui a généré l'événement (**b1** dans l'exemple présenté) en invoquant la méthode **getSource()** de l'objet événement **evt**.

# Gestion des événements

## Schéma de fonctionnement

Les événements utilisateurs sont gérés par plusieurs interfaces **EventListener**.

Les interfaces **EventListener** permettent de définir les traitements en réponse à des événements utilisateurs générés par un composant.

Une classe doit contenir une interface auditrice pour chaque type d'événements à traiter, par exemple :

- **ActionListener** : clic de souris ou enfoncement de la touche Enter
- **ItemListener** : utilisation d'une liste ou d'une case à cocher
- **MouseMotionListener** : événement de souris
- **WindowListener** : événement de fenêtre

# Gestion des événements

## L'interception des actions de l'utilisateur

L'ajout d'une interface `EventListener` impose plusieurs ajouts dans le code :

1. importer le groupe de classes `java.awt.event`
2. la classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute. Pour déclarer plusieurs interfaces, il suffit de les séparer par des virgules
3. Appel à la méthode `addXXX()` pour enregistrer l'objet qui gèrera les événements `XXX` du composant. Il faut configurer le composant pour qu'il possède un «écouteur» pour l'événement utilisateur concerné.
4. implémenter les méthodes déclarées dans les interfaces

Pour identifier le composant qui a généré l'événement, on peut utiliser la méthode `getActionCommand()`, ou `getSource()` de l'objet `ActionEvent`.

- La méthode `getActionCommand()` renvoie une chaîne de caractères.
- La méthode `getSource()` renvoie l'objet qui a généré l'événement.

# Gestion des événements

## Les différentes implémentations des Listeners

La mise en œuvre des Listeners peut se faire selon différentes formes :

- une classe indépendante
- la classe implémentant elle même l'interface
- une classe interne
- une classe interne anonyme.

# Gestion des événements

## Les différentes implémentations des Listeners

### Une classe indépendante

```
package exemple;

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

public class exemple extends JFrame{
    private JButton b1,b2;
    public exemple() {
        b1 = new JButton("Bouton 1"); b2 = new JButton("Bouton 2");

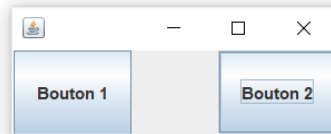
        b1.addActionListener(new AL());
        b2.addActionListener(new AL());

        this.add(b1,"West"); this.add(b2,"East");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,100);
        setVisible(true);
    }
    public static void main(String[] args) {new exemple();}
}
```

```
package exemple;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class AL implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        System.out.println(" Le bouton cliqué est : " + e.getActionCommand());
    }
}
```



Problems Javadoc Declaration Console Coverage  
exemple [Java Application] C:\Program Files\Java\jre-9.0.4\bin\javaw.exe (6 avr. 2022 à 14:48:10)  
Le bouton cliqué est : Bouton 1  
Le bouton cliqué est : Bouton 2  
Le bouton cliqué est : Bouton 2

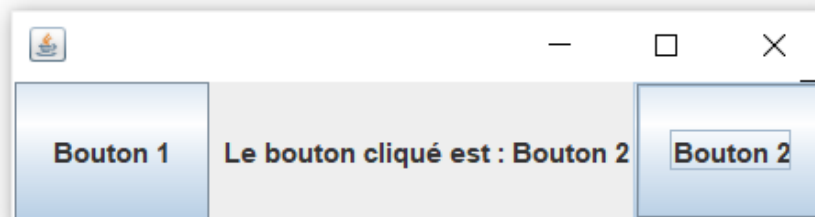
# Gestion des événements

## Les différentes implémentations des Listeners

### Une classe implémentant elle même le listener

```
public class exemple extends JFrame implements ActionListener{
    private JLabel l1;
    private JButton b1,b2;
    public exemple() {
        l1 = new JLabel(" Le bouton cliqué est : ");
        b1 = new JButton("Bouton 1"); b2 = new JButton("Bouton 2");
        b1.addActionListener(this);b2.addActionListener(this);
        this.add(b1,"West"); this.add(b2,"East");this.add(l1,"Center");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,100);
        setVisible(true);
    }

    public static void main(String[] args) {new exemple();}
    public void actionPerformed(ActionEvent e) {
        l1.setText(" Le bouton cliqué est : " + e.getActionCommand());
    }
}
```



# Gestion des événements

## Les différentes implémentations des Listeners

### Une classe interne

```
public class exemple extends JFrame{
    private JLabel l1;
    private JButton b1,b2;
    public exemple() {
        l1 = new JLabel(" Le bouton cliqué est : ");
        b1 = new JButton("Bouton 1"); b2 = new JButton("Bouton 2");

        AcLi class_AL = new AcLi();
        b1.addActionListener(class_AL);b2.addActionListener(class_AL);
        this.add(b1,"West"); this.add(b2,"East");this.add(l1,"Center");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,100);
        setVisible(true);
    }

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

    public class AcLi implements ActionListener{
        public void actionPerformed(ActionEvent e) {
            l1.setText(" Le bouton cliqué est : " + e.getActionCommand());
        }
    }
}
```



# Gestion des événements

## Les différentes implémentations des Listeners

### Une classe interne anonyme

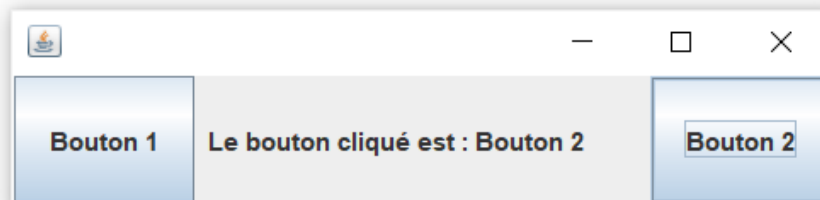
```
public class exemple extends JFrame{
    private JLabel l1;
    private JButton b1,b2;
    public exemple() {
        l1 = new JLabel(" Le bouton cliqué est : ");
        b1 = new JButton("Bouton 1"); b2 = new JButton("Bouton 2");

        b1.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                l1.setText(" Le bouton cliqué est : " + e.getActionCommand());
            }
        });

        b2.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                l1.setText(" Le bouton cliqué est : " + e.getActionCommand());
            }
        });

        this.add(b1,"West"); this.add(b2,"East");this.add(l1,"Center");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,100);
        setVisible(true);
    }

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





# Gestion des événements

## L'interception des actions de l'utilisateur

### **L'interface ActionListener**

Cette interface permet de réagir aux cliques de souris ou enfoncements de la touche Enter.

Pour qu'un composant génère des événements, il faut utiliser la méthode `addActionListener()`. Ces événements sont reçus par la méthode `actionPerformed()` qui attend un objet de type `actionEvent` en argument

# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface ActionListener

```
public class exemple extends JFrame implements ActionListener{
    private JLabel lab;
    private JButton b1,b2,b3;
    public exemple() {
        lab = new JLabel("le bouton cliqué est :");
        b1 = new JButton("boutton 1");
        b1.addActionListener(this);

        b2 = new JButton("boutton 2");
        b2.addActionListener(this);

        b3 = new JButton("boutton 3");
        b3.addActionListener(this);

        setLayout(new FlowLayout());
        add(b1);add(b2);add(b3);add(lab);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,100);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent evt) {
        String composant = evt.getActionCommand();
        lab.setText("le bouton cliqué est :" + composant);
    }

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



# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface ActionListener

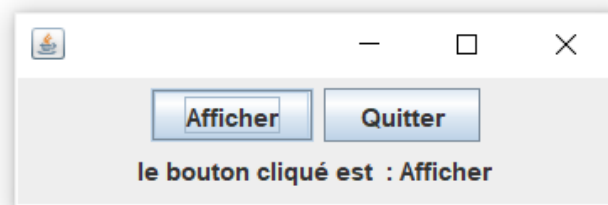
```
public class exemple extends JFrame implements ActionListener{
    private JLabel lab;
    private JButton b1,b2;
    public exemple() {
        lab = new JLabel("le bouton cliqué est :");
        b1 = new JButton("Afficher");
        b1.addActionListener(this);

        b2 = new JButton("Quitter");
        b2.addActionListener(this);

        setLayout(new FlowLayout());
        add(b1);add(b2);add(lab);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,100);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource()==b1) lab.setText("le bouton cliqué est : " + evt.getActionCommand());
        if (evt.getSource()==b2) System.exit(0);;
    }

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



# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface `ItemListener`

Cette interface permet de réagir à la sélection de cases à cocher et de listes d'options.

Pour qu'un composant génère des événements, il faut utiliser la méthode `addItemListener()`.

Ces événements sont reçus par la méthode `itemStateChanged()` qui attend un objet de type `ItemEvent` en argument

Pour déterminer si une case à cocher est sélectionnée ou inactive, utiliser la méthode `getStateChange()` avec les constantes `ItemEvent.SELECTED` ou `ItemEvent.DESELECTED`.

# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface ItemListener

```
public class exemple extends JFrame implements ItemListener{

    JCheckBox cb;

    public exemple() {
        cb = new JCheckBox(" choix ",true);
        cb.addItemListener(this);
        add(cb);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,300);
        pack();
        setVisible(true);
    }

    public void itemStateChanged(ItemEvent event) {
        if (event.getStateChange() == ItemEvent.SELECTED) System.out.println("choix selectionne");
        else System.out.println("choix selectionne");
    }

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

# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface ItemListener

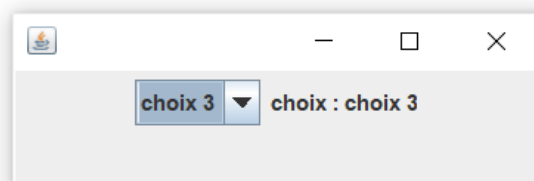
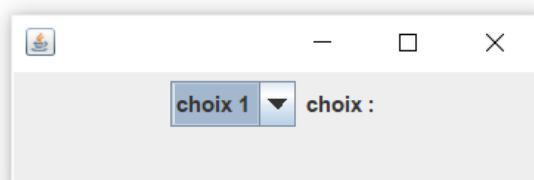
```
public class exemple extends JFrame implements ItemListener{

    JComboBox cb;
    JLabel lab;

    public exemple() {
        cb = new JComboBox();
        lab = new JLabel("choix : ");
        cb.addItem("choix 1");
        cb.addItem("choix 2");
        cb.addItem("choix 3");
        cb.addItemListener(this);
        add(cb);
        add(lab);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,100);
        setVisible(true);
    }

    public void itemStateChanged(ItemEvent event) {
        Object obj = event.getItem();
        String selection = (String)obj;
        lab.setText("choix : " + selection);
        System.out.println("choix : " + selection);
    }

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



# Gestion des événements

## L'interception des actions de l'utilisateur

### **L'interface TextListener**

Cette interface permet de réagir aux modifications de la zone de saisie ou du texte.

La méthode `addTextListener()` permet à un composant de texte de générer des événements utilisateur. La méthode `TextValueChanged()` reçoit les événements.

### **L'interface MouseMotionListener**

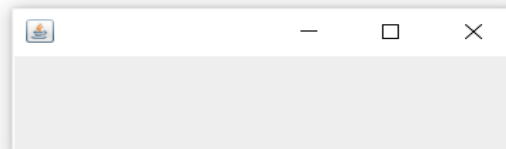
La méthode `addMouseMotionListener()` permet de gérer les événements liés à des mouvements de souris. Les méthodes `mouseDragged()` et `mouseMoved()` reçoivent les événements.

# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface MouseMotionListener

```
8 public class exemple extends JFrame implements MouseMotionListener{
9     private int x;
10    private int y;
11    public exemple() {
12
13        this.addMouseMotionListener(this);
14        setDefaultCloseOperation(EXIT_ON_CLOSE);
15        setSize(300,100);
16        setVisible(true);
17    }
18    public void mouseDragged(MouseEvent arg0) {}
19
20    public void mouseMoved(MouseEvent e) {
21        x = e.getX();
22        y = e.getY();
23        System.out.println("x = " + x + " ; y = " + y);
24    }
25    public static void main(String[] args) {new exemple();}
26
27
28 }
29
```



Problems @ Javadoc Declaration Console Coverage

exemple [Java Application] C:\Program Files\Java\jre-9.0.4\bin\javaw.exe (6 avr. 2022 à 14:07:47)

```
x = 137 ; y = 56
x = 138 ; y = 56
x = 138 ; y = 57
x = 139 ; y = 57
x = 139 ; y = 58
x = 139 ; y = 59
```



# Gestion des événements

## L'interception des actions de l'utilisateur

### L'interface **MouseListener**

Cette interface permet de réagir aux clics de souris. Les méthodes de cette interface sont :

- `public void mouseClicked(MouseEvent e);`
- `public void mousePressed(MouseEvent e);`
- `public void mouseReleased(MouseEvent e);`
- `public void mouseEntered(MouseEvent e);`
- `public void mouseExited(MouseEvent e);`

### L'interface **WindowListener**

La méthode `addWindowListener()` permet à un objet `Frame` de générer des événements. Les méthodes de cette interface sont :

- `public void windowOpened(WindowEvent e)`
- `public void windowClosing(WindowEvent e)`
- `public void windowClosed(WindowEvent e)`
- `public void windowIconified(WindowEvent e)`
- `public void windowDeiconified(WindowEvent e)`
- `public void windowActivated(WindowEvent e)`
- `public void windowDeactivated(WindowEvent e)`

# Gestion des événements

## Listeners courants

Listeners / Event	Méthodes de listener	Générés par
ActionListener ActionEvent	actionPerformed()	AbstractButton, Button, ButtonModel, ComboBoxEditor, JComboBox, JFileChooser, JTextField, List, MenuItem, TextField, Timer
FocusListener FocusEvent	focusGained() focusLost()	Component
ItemListener ItemEvent	itemStateChanged()	AbstractButton, ButtonModel, Checkbox, CheckboxMenuItem, Choice, ItemSelectable, JComboBox, List
KeyListener KeyEvent	keyPressed() keyReleased() keyTyped()	Component

# Gestion des événements

## Listeners courants

Listeners / Event	Méthodes de listener	Générés par
MouseListener MouseEvent	mouseClicked() mouseEntered() mouseExited() mousePressed()	Component
MouseMotionListener MouseEvent	mouseDragged() mouseMoved()	Component
TextListener TextEvent	textValueChanged()	TextComponent
WindowListener WindowEvent	windowActivated() windowClosed() windowClosing() windowOpened()	Window

# Gestion des événements

## Listeners courants

Listeners / Event	Méthodes de listener	Générés par
ComponentListener ComponentEvent	componentShown() componentMoved() componentResized()	Component
ChangeListener ChangeEvent	stateChanged()	AbstractButton, BoundedRangeModel, ButtonModel, JProgressBar, JSlider, JTabbedPane, JViewport, MenuSelectionManager, SingleSelectionModel
ListDataListener ListDataEvent	contentsChanged() intervalAdded() intervalRemoved()	AbstractListModel, ListModel
ListSelectionListener ListSelectionEvent	valueChanged()	JList, ListSelectionModel

# Gestion des événements

## Exercice 1

Écrire la classe correspondante à la figure suivante :

The image shows a Java Swing window titled "Article". The window has a standard title bar with a close button (X), a maximize button (square), and a minimize button (dash). The main content area is divided into two parts. The top part is a header bar with five text labels and corresponding input fields: "Article :", "Reference :", "Prix HT :", "TVA :", and "Prix TTC :". The bottom part is a large text area labeled "Informations sur l'article :". At the bottom of the window, there are two buttons: "Créer" and "Quitter".

Article :	Reference :	Prix HT :	TVA :	Prix TTC :
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Informations sur l'article :

# Gestion des événements

## Exercice 1

```
import java.awt.GridLayout;

public class article extends JFrame implements ActionListener{

    private JLabel l1,l2,l3,l4,l5;
    private JTextField t1,t2,t3,t4,t5;
    private JTextArea ta;
    private JButton b1,b2;
    private JPanel p1,p2,p3;

    public article() {
        l1=new JLabel("Article :");
        l2=new JLabel("Reference :");
        l3=new JLabel("Prix HT :");
        l4=new JLabel("TVA :");
        l5=new JLabel("Prix TTC :");
        ta=new JTextArea("Informations sur l'article :",30,30);
        b1=new JButton("Créer");
        b2=new JButton("Quitter");
        t2=new JTextField(7);t1=new JTextField(7);
        t3=new JTextField(7);t4=new JTextField(7);t5=new JTextField(7);
        p1=new JPanel();p2=new JPanel();p3=new JPanel();
```

```
        p1.setLayout(new GridLayout(1, 10));
        p1.add(l1);p1.add(t1);p1.add(l2);p1.add(t2);p1.add(l3);p1.add(t3);p1.add(l4);p1.add(t4);p1.add(l5);p1.add(t5);
        p2.add(ta);
        p3.add(b1);p3.add(b2);

        this.add(p1,"North");
        this.add(p2, "Center");
        this.add(p3, "South");

        b1.addActionListener(this);
        b2.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setTitle("Article");
        setSize(750, 300);
        setVisible(true);
    }
    public static void main(String[] args) { new article(); }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==b1) {
            String S=new String();
            S = "Informations sur l'article :" + "\n Article :" + t1.getText() + "\n Reference :" + t2.getText()+ "\n
            ta.setText(S);
            t1.setText("");t2.setText("");t3.setText("");t4.setText("");t5.setText("");
        }
        if(e.getSource()==b2) { System.exit(0);}
    }
}
```

# Gestion des événements

## Exercice 2

Écrire la classe correspondante à la figure suivante :

The image shows a Java Swing window titled "Étudiant". On the left side of the window, there is a form with four input fields: "Nom", "Prénom", "Email", and "Filière". The "Filière" field is a dropdown menu currently showing "GL". On the right side, there is a table with four columns: "Nom", "Prénom", "Email", and "Filière". The first row of the table contains the values "Red", "Ali", "ali@gmail.com", and "SMI". Below the table, there are three buttons: "Ajouter", "Quitter", and "Supprimer".

Nom	Prénom	Email	Filière
Red	Ali	ali@gmail.com	SMI

# Gestion des événements

## Exercice 2

```
public class etudiant extends JFrame implements ActionListener{
    int i=0;
    private JLabel l1,l2,l3,l4;
    private JTextField t1,t2,t3;
    private JButton b1,b2,b3;
    private JPanel p1,p2,p3;
    private JComboBox cb;
    String[] items = {"SMI", "SMP","SMA" ,"GL" ,"SVT"};
    String[] entetes = {"Nom", "Prénom", "Email ", "Filière"};
    JTable tableau; DefaultTableModel model;

    Object[][] donnees={{ " ", " ", " ", " " },{" ", " ", " ", " " }};

    public etudiant() {
        l1=new JLabel("Nom      :");l2=new JLabel("Prénom :");
        l3=new JLabel("Email    :");l4=new JLabel("Filière:");
        cb = new JComboBox(items);

        b1=new JButton("Ajouter");b2=new JButton("Quitter");b3=new JButton("Supprimer");

        model = new DefaultTableModel(donnees, entetes);
        tableau = new JTable(model);

        t2=new JTextField(15);t1=new JTextField(15);
        t3=new JTextField(15);p1=new JPanel();p2=new JPanel();p3=new JPanel();
        p1.add(l1);p1.add(t1);p1.add(l2);p1.add(t2);p1.add(l3);p1.add(t3);p1.add(l4);p1.add(cb);
        p2.add(b1);p2.add(b2);p2.add(b3);

        p3.setLayout(new BorderLayout());
        p3.add(tableau.getTableHeader(), BorderLayout.NORTH);
        p3.add(tableau, BorderLayout.CENTER);
        this.add(p1,"Center");
        this.add(p2, "South");
        this.add(p3, "East");

        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setTitle("Etudiant");
        setSize(550, 300);
        setVisible(true);
    }

    public static void main(String[] args) {new etudiant();}
    public void addEtudiant(String[] data) {model.insertRow(i,data);i++;}
    public void removeEtudiant(int[] rows) {
        for(int i=0;i<rows.length;i++){model.removeRow(rows[i]-i);}
    }
}
```



# Gestion des événements

## Exercice 2

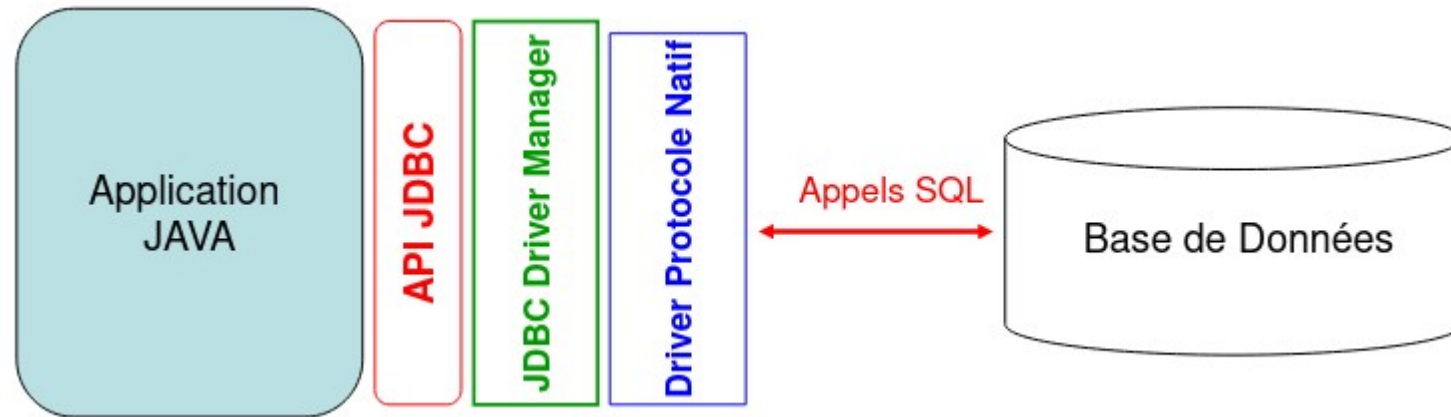
```
public void actionPerformed(ActionEvent e) {  
    // TODO Auto-generated method stub  
    String f="";  
    switch (cb.getSelectedIndex()) {  
        case 0:  
            f="SMI";  
            break;  
        case 1:  
            f="SMP";  
            break;  
        case 2:  
            f="SMA";  
            break;  
        case 3:  
            f="GL";  
            break;  
        case 4:  
            f="SVT";  
            break;  
        default:  
            f="--";  
            break;  
    }  
}
```

```
        if(e.getSource()==b1) {  
            String[] data= {t1.getText(),t2.getText(),t3.getText(),f};  
            addEtudiant(data);  
            t1.setText("");t2.setText("");t3.setText("");  
        }  
        if(e.getSource()==b2) {System.exit(0);}  
        if(e.getSource()==b3) {int[] rows = tableau.getSelectedRows();removeEtudiant(rows);}  
    }  
}
```

# Gestion des bases de données

- L'API Java Database Connectivity (JDBC) est la norme de l'industrie pour la connectivité indépendante de la base de données entre le langage de programmation Java et un large éventail de bases de données SQL et d'autres sources de données tabulaires, telles que des feuilles de calcul ou des fichiers plats. L'API JDBC fournit une API au niveau de l'appel pour l'accès à la base de données basée sur SQL.
- La technologie JDBC permet d'utiliser le langage de programmation Java pour exploiter les fonctionnalités "Write Once, Run Anywhere" pour les applications nécessitant un accès aux données de l'entreprise. Avec un pilote compatible avec la technologie JDBC, vous pouvez connecter toutes les données de l'entreprise même dans un environnement hétérogène.

# Gestion des bases de données



# Gestion des bases de données

- Les étapes suivantes configurent un environnement de développement JDBC :
  - Installez la dernière version du SDK Java SE sur votre ordinateur
  - Installez votre système de gestion de base de données (SGBD)
  - Installez un pilote JDBC du fournisseur de votre base de données
  - Créer des bases de données, des tables et remplir des tables
  - Exécutez les échantillons de test

# Gestion des bases de données

- Installez un pilote JDBC du fournisseur de votre base de données
  - Si vous utilisez MySQL, installez la dernière version du pilote JDBC pour MySQL, Connector/J.  
(<https://dev.mysql.com/downloads/connector/j/>)
  - Contactez le fournisseur de votre base de données pour obtenir un pilote JDBC pour votre SGBD.

# Gestion des bases de données

- Il existe de nombreuses implémentations possibles des pilotes JDBC. Ces implémentations sont classées comme suit :
  - Type 1 : Pilotes qui implémentent l'API JDBC en tant que mappage vers une autre API d'accès aux données, telle que ODBC (Open Database Connectivity). Les pilotes de ce type sont généralement dépendants d'une bibliothèque native, ce qui limite leur portabilité. Le pont JDBC-ODBC est un exemple de pilote de type 1.
  - Type 2 : Pilotes écrits en partie dans le langage de programmation Java et en partie en code natif. Ces pilotes utilisent une bibliothèque cliente native spécifique à la source de données à laquelle ils se connectent. Encore une fois, à cause du code natif, leur portabilité est limitée. Le pilote côté client OCI (Oracle Call Interface) d'Oracle est un exemple de pilote de type 2.
  - Type 3 : Pilotes qui utilisent un client Java pur et communiquent avec un serveur middleware à l'aide d'un protocole indépendant de la base de données. Le serveur middleware communique ensuite les requêtes du client à la source de données.
  - Type 4 : Pilotes purement Java et implémentant le protocole réseau pour une source de données spécifique. Le client se connecte directement à la source de données.
  - Vérifiez quels types de pilotes sont fournis avec votre SGBD. MySQL Connector/J est un pilote de type 4.

# Gestion des bases de données

- Installation d'un pilote JDBC :
  - L'installation d'un pilote JDBC consiste généralement à copier le pilote sur votre ordinateur, puis à ajouter son emplacement à votre chemin de classe. De plus, de nombreux pilotes JDBC autres que les pilotes de type 4 nécessitent l'installation d'une API côté client. Aucune autre configuration spéciale n'est généralement nécessaire.
  - Remarque : Le pont JDBC-ODBC doit être considéré comme une solution de transition. Il n'est pas pris en charge par Oracle. N'envisagez de l'utiliser que si votre SGBD n'offre pas de pilote JDBC Java uniquement.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC :
  - En général, pour traiter n'importe quelle requete SQL avec JDBC, vous suivez ces étapes :
    - Établir une connexion.
    - Créez un statement.
    - Exécutez la requête.
    - Traitez l'objet ResultSet.
    - Fermez la connexion.
    - Les différentes classes sont disponibles dans le package java.sql



# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC :
  - Établir une connexion : Tout d'abord, établissez une connexion avec la source de données que vous souhaitez utiliser. Une source de données peut être un SGBD, un système de fichiers hérité ou une autre source de données avec un pilote JDBC correspondant. Cette connexion est représentée par un objet Connection.
  - En règle générale, une application JDBC se connecte à une source de données cible à l'aide de l'une des deux classes :
    - DriverManager et DataSource

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : Établir une connexion :
  - DriverManager : cette classe entièrement implémentée connecte une application à une source de données, qui est spécifiée par une URL de base de données. Lorsque cette classe tente pour la première fois d'établir une connexion, elle charge automatiquement tous les pilotes JDBC 4.0 trouvés dans le chemin de classe.
  - DataSource : Cette interface est préférée à DriverManager car elle permet aux détails sur la source de données sous-jacente d'être transparents pour votre application. Les propriétés d'un objet DataSource sont définies de sorte qu'il représente une source de données particulière.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : Établir une connexion :

- DriverManager :

- La connexion à votre SGBD avec la classe DriverManager implique l'appel de la méthode DriverManager.getConnection. La méthode suivante établit une connexion à la base de données :

```
public Connection getConnection(String userName,
                               String password) throws SQLException {
    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", userName);
    connectionProps.put("password", password);
    if (this.dbms.equals("mysql")) {
        conn =
            DriverManager.getConnection("jdbc:" + this.dbms + "://" + this.serverName
+                                     ":" + this.portNumber + "/",
                                     connectionProps);
        conn.setCatalog(this.dbName);
    } else if (this.dbms.equals("derby")) {
        conn =
            DriverManager.getConnection("jdbc:" + this.dbms + ":" + this.dbName +
+                                     ";create=true", connectionProps);
    }
    return conn;
}
```

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : Établir une connexion :
  - La méthode `DriverManager.getConnection` établit une connexion à la base de données. Cette méthode nécessite une URL de base de données, qui varie en fonction de votre SGBD. Voici quelques exemples d'URL de base de données :
    - MySQL : `jdbc:mysql://localhost:3306/`, où `localhost` est le nom du serveur hébergeant votre base de données et `3306` est le numéro de port

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Création de statement** :
- Un **Statement** est une interface qui représente une instruction SQL. Vous exécutez des objets **Statement**, et ils génèrent des objets **ResultSet**, qui est une table de données représentant un ensemble de résultats de base de données. Vous avez besoin d'un objet **Connection** pour créer un objet **Statement**.
  - Par exemple, CoffeesTable.viewTable crée un objet **Statement** avec le code suivant :
  - `stmt = con.createStatement();`
  - **Il existe trois types de déclarations différentes** :
  - **Statement**: utilisée pour implémenter des instructions SQL simples sans paramètres.
  - **PreparedStatement** : (Extends Statement.) Utilisé pour précompiler des instructions SQL susceptibles de contenir des paramètres d'entrée.
  - **CallableStatement** : (étend PreparedStatement.) utilisé pour exécuter des procédures stockées qui peuvent contenir à la fois des paramètres d'entrée et de sortie.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Création de statement :**

```
public static void createDatabase(Connection connArg, String  
dbNameArg,
```

- Exemple :

- La méthode suivante permet de créer une base de données si elle n'existe pas

```
String dbmsArg) {  
  
    if (dbmsArg.equals("mysql")) {  
        try {  
            Statement s = connArg.createStatement();  
            String newDatabaseString =  
                "CREATE DATABASE IF NOT EXISTS " + dbNameArg;  
            // String newDatabaseString = "CREATE DATABASE " + dbName;  
            s.executeUpdate(newDatabaseString);  
  
            System.out.println("Created database " + dbNameArg);  
        } catch (SQLException e) {  
            printSQLException(e);  
        }  
    }  
}
```

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Exécuter des requêtes**
- Pour exécuter une requête, appelez une méthode d'exécution à partir de Statement telle que la suivante :
  - **execute** : renvoie true si le premier objet renvoyé par la requête est un objet **ResultSet**. Utilisez cette méthode si la requête peut renvoyer un ou plusieurs objets **ResultSet**. Récupérez les objets **ResultSet** renvoyés par la requête en appelant à plusieurs reprises **Statement.getResultSet**.
  - **executeQuery** : renvoie un objet **ResultSet**.
  - **executeUpdate** : renvoie un entier représentant le nombre de lignes affectées par l'instruction SQL. Utilisez cette méthode si vous utilisez des instructions SQL INSERT, DELETE ou UPDATE.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Exécuter des requêtes**

```
public static void viewTable(Connection con) throws SQLException {
    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + ", " + supplierID + ", " + price +
                               ", " + sales + ", " + total);
        }
    } catch (SQLException e) {
        JBDBTutorialUtilities.printStackTrace(e);
    }
}
```



# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **PreparedStatement** :
- Parfois, il est plus pratique d'utiliser un objet **PreparedStatement** pour envoyer des instructions SQL à la base de données. Ce type spécial d'instruction est dérivé de la classe plus générale, **Statement**, que vous connaissez déjà.
- Si vous souhaitez exécuter un objet **Statement** plusieurs fois, cela réduit généralement le temps d'exécution pour utiliser un objet **PreparedStatement** à la place.
- La principale caractéristique d'un objet **PreparedStatement** est que, contrairement à un objet **Statement**, il reçoit une instruction SQL lors de sa création.
- L'avantage est que dans la plupart des cas, cette instruction SQL est immédiatement envoyée au SGBD, où elle est compilée. Par conséquent, l'objet **PreparedStatement** contient non seulement une instruction SQL, mais une instruction SQL qui a été précompilée. Cela signifie que lorsque le **PreparedStatement** est exécuté, le SGBD peut simplement exécuter l'instruction SQL **PreparedStatement** sans avoir à le compiler au préalable.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Création d'un objet PreparedStatement**
  - Un objet PreparedStatement accepte deux paramètres d'entrée **pour le créer on :**

```
String updateString =  
    "update COFFEES set SALES = ? where COF_NAME = ?";  
//....
```

```
PreparedStatement updateSales = con.prepareStatement(updateString);
```

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Fourniture de valeurs pour les paramètres PreparedStatement :**
  - Vous devez fournir des valeurs à la place des espaces réservés de point d'interrogation (s'il y en a) avant de pouvoir exécuter un objet PreparedStatement. Pour ce faire, appelez l'une des méthodes setter définies dans la classe PreparedStatement. Les instructions suivantes fournissent les deux espaces réservés de point d'interrogation dans le PreparedStatement nommé `updateSales` :

```
updateSales.setInt(1, e.getValue().intValue());  
updateSales.setString(2, e.getKey());  
updateSales.executeUpdate();
```

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC :

## PreparedStatement :

- Exemple :

La méthode suivante, updateCoffeeSales, stocke le nombre d'euros de café vendues la semaine en cours dans la colonne SALES pour chaque type de café et met à jour le nombre total d'euros de café vendues dans la colonne TOTAL pour chaque type de café :

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek) throws SQLException {
    String updateString =
        "update COFFEES set SALES = ? where COF_NAME = ?";
    String updateStatement =
        "update COFFEES set TOTAL = TOTAL + ? where COF_NAME = ?";

    try (PreparedStatement updateSales = con.prepareStatement(updateString);
         PreparedStatement updateTotal = con.prepareStatement(updateStatement))
    {
        con.setAutoCommit(false);
        for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
            updateSales.setInt(1, e.getValue().intValue());
            updateSales.setString(2, e.getKey());
            updateSales.executeUpdate();

            updateTotal.setInt(1, e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
            updateTotal.executeUpdate();
            con.commit();
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printStackTrace(e);
        if (con != null) {
            try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
            } catch (SQLException excep) {
                JDBCTutorialUtilities.printStackTrace(excep);
            }
        }
    }
}
```

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Fermeture des connexions**
  - Lorsque vous avez fini d'utiliser un objet **Connection**, **Statement** ou **ResultSet**, appelez sa méthode **close** pour libérer immédiatement les ressources qu'il utilise.
  - Vous pouvez également utiliser une instruction **try** pour fermer automatiquement les objets **Connection**, **Statement** et **ResultSet**, qu'une exception **SQLException** ait été levée ou non. (JDBC lève une exception **SQLException** lorsqu'il rencontre une erreur lors d'une interaction avec une source de données.

# Gestion des bases de données

- Traitement des requêtes SQL avec JDBC : **Fermeture des connexions** : Une instruction de ressource automatique se compose d'une instruction **try** et d'une ou plusieurs ressources déclarées. Par exemple, la méthode `viewTable` ferme automatiquement son objet **Statement**, comme suit :

```
public static void viewTable(Connection con) throws SQLException {  
    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";  
    try (Statement stmt = con.createStatement()) {  
        ResultSet rs = stmt.executeQuery(query);  
        while (rs.next()) {  
            String coffeeName = rs.getString("COF_NAME");  
            int supplierID = rs.getInt("SUP_ID");  
            float price = rs.getFloat("PRICE");  
            int sales = rs.getInt("SALES");  
            int total = rs.getInt("TOTAL");  
            System.out.println(coffeeName + ", " + supplierID + ", " + price +  
                               ", " + sales + ", " + total);  
        }  
    } catch (SQLException e) {  
        JDBCTutorialUtilities.printSQLException(e);  
    }  
}
```