

---

# Programmation Orientée Objet Java

## Deuxième Partie

---



---

# Rappels

---



---

# Rappels :Qualité d'un Logiciel

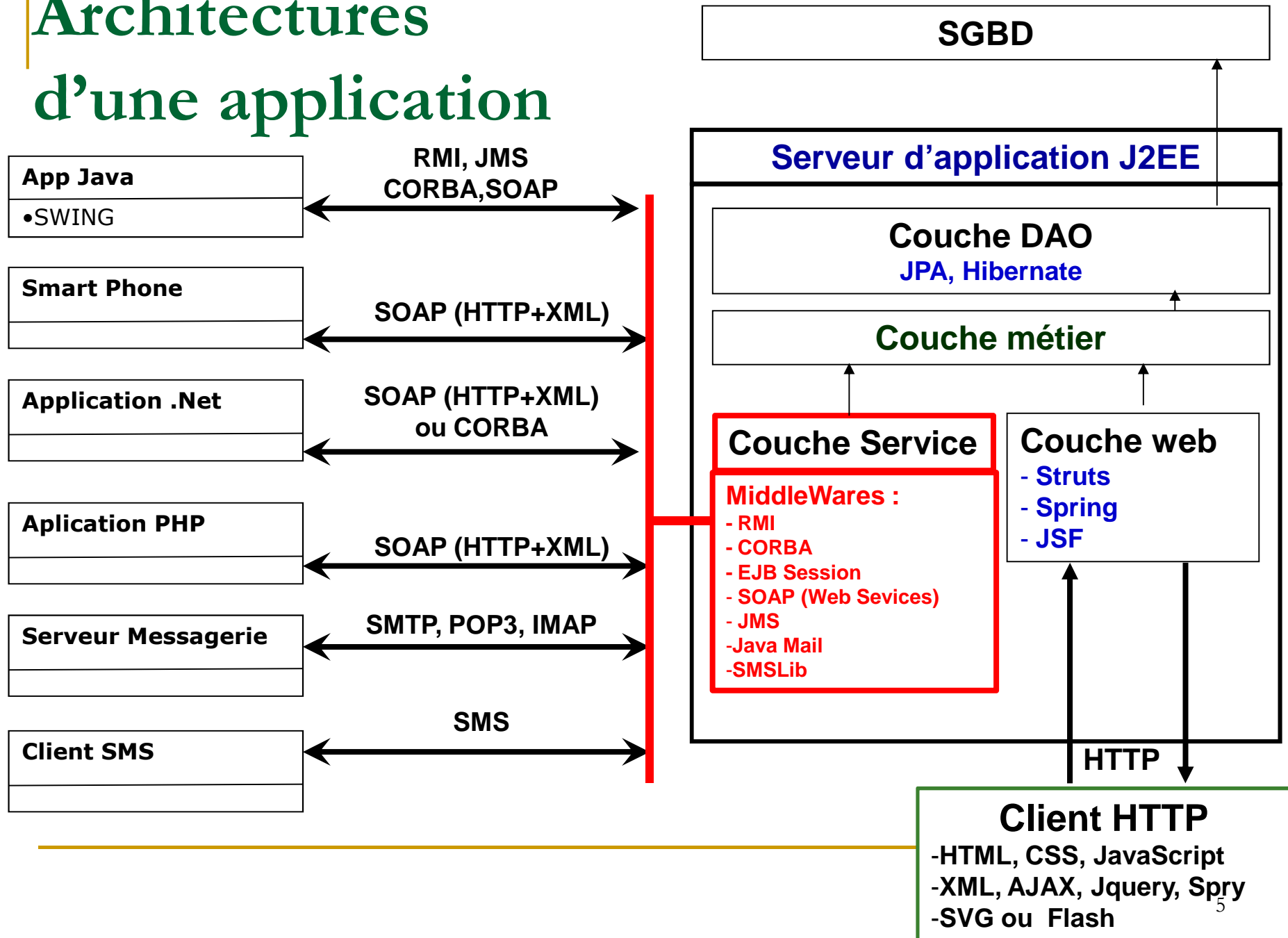
- La qualité d'un logiciel se mesure par rapport à plusieurs critères :
  - ❑ Répondre aux spécifications fonctionnelles :
    - Une application est créée pour répondre , tout d'abord, aux besoins fonctionnels des entreprises.
  - ❑ Les performances:
    - La rapidité d'exécution et Le temps de réponse
    - Doit être bâtie sur une architecture robuste.
    - Eviter le problème de montée en charge
  - ❑ La maintenance:
    - Une application doit évoluer dans le temps.
    - Doit être fermée à la modification et ouverte à l'extension
    - Une application qui n'évolue pas meurt.
    - Une application mal conçue est difficile à maintenir, par suite elle finit un jour à la poubelle.

---

# Qualité d'un Logiciel

- La qualité d'un logiciel se mesure par rapport à plusieurs critères : (Suite)
    - Sécurité
      - Garantir l'intégrité et la sécurité des données
    - Portabilité
      - Doit être capable de s'exécuter dans différentes plateformes.
    - Capacité de communiquer avec d'autres applications distantes.
    - Disponibilité et tolérance aux pannes
    - Capacité de fournir le service à différents type de clients :
      - Client lourd : Interfaces graphiques SWING
      - Interface Web : protocole http
      - Client SmartPhone
      - Téléphone : SMS
      - ....
    - Design des ses interfaces graphiques
      - Charte graphique et charte de navigation
      - Accès via différentes interfaces (Web, Téléphone, PDA, ,)
    - Coût du logiciel
-

# Architectures d'une application



---

# Qu'est ce que java?

- Langage de **programmation orienté objet** (Classe, Objet, Héritage, Encapsulation et Polymorphisme)
- Avec java on peut créer des application **multiplateformes**. Les applications java sont **portables**. C'est-à-dire, on peut créer une application java dans une plateforme donnée et on peut l'exécuter sur n'importe quelle autre plateforme.
- Java est utilisé pour créer :
  - ❑ Des applications Desktop
  - ❑ Des applets java (applications java destinées à s'exécuter dans une page web)
  - ❑ Des applications pour les smart phones (JME, Androide)
  - ❑ Des applications embarquées dans des cartes à puces (Java Card Edition)
  - ❑ Des application JEE (Java Entreprise Edition) : Les applications qui vont s'exécuter dans un serveur d'application JEE (Web Sphere Web Logic, Jboss,etc...).

---

# Programme

- Java?
- Programmation orientée Objet Java
  - Objet et Classe
  - Héritage et accessibilité
  - Polymorphisme
  - Collections
- Exceptions
- Entrées Sorties
- Application de synthèse COO (UML) et POO (JAVA).
- Interfaces graphique AWT et SWING
- Accès aux bases de données
- Notions de Design Patterns
- Threads
- Sockets
- Mapping objet relationnel avec Hibernate
- Applications distribuées avec RMI

---

# Exceptions et Entrées sorties

---





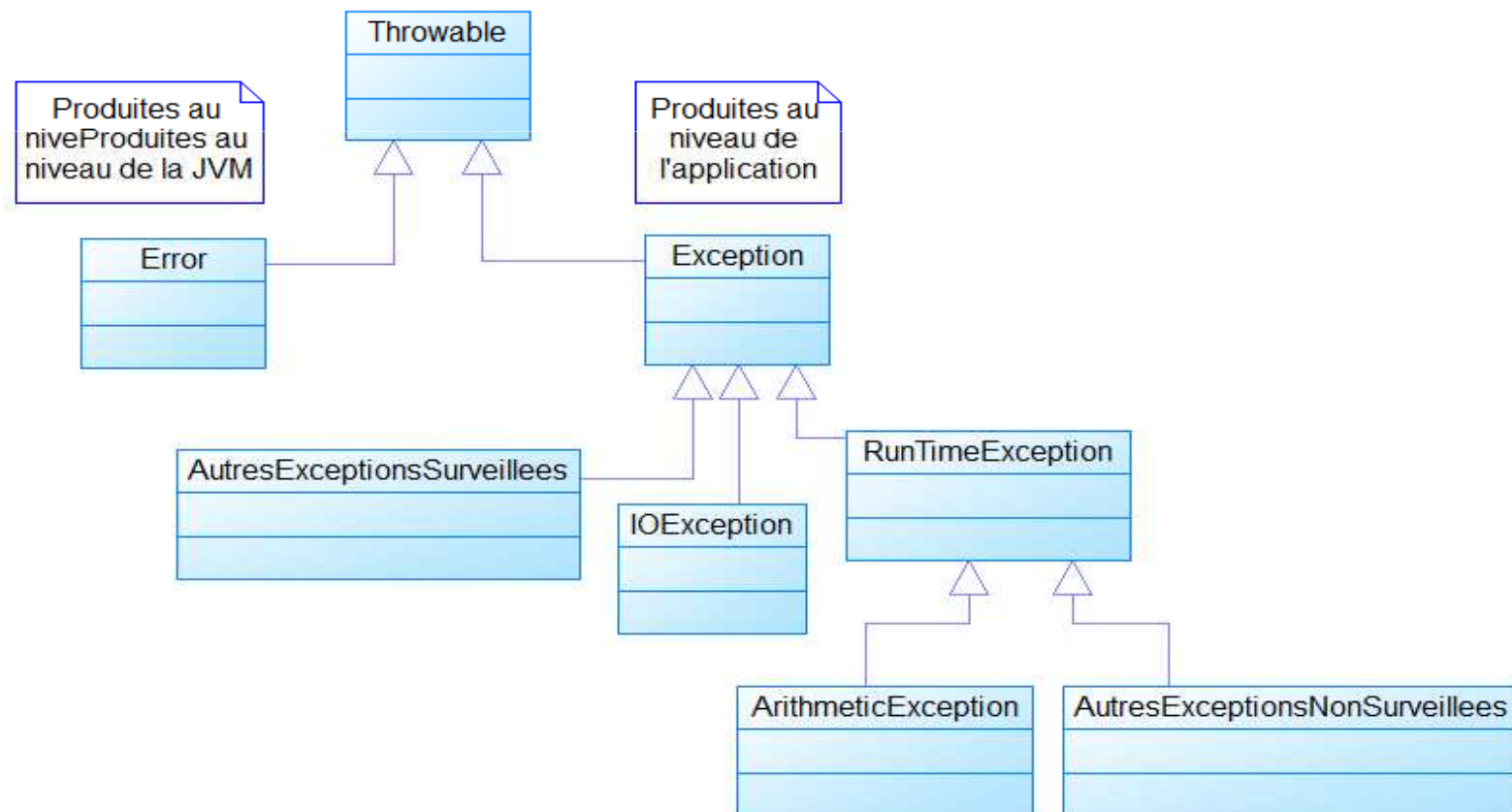
---

# Exceptions

- Souvent, un programme doit traiter des situations exceptionnelles qui n'ont pas un rapport direct avec sa tâche principale.
- Ceci oblige le programmeur à réaliser de nombreux tests avant d'écrire les instructions utiles du programme. Cette situation a deux inconvénients majeurs :
  - Le programmeur peut omettre de tester une condition ;
  - Le code devient vite illisible car la partie utile est masquée par les tests.
- Java remédie à cela en introduisant un *Mécanisme de gestion des exceptions* .
- Grâce à ce mécanisme, on peut améliorer grandement la lisibilité du code
  - En séparant
    - le code utile (Code métier)
    - de celui qui traite des situations exceptionnelles,

# Hiérarchie des exceptions

- Java peut générer deux types d'erreurs au moment de l'exécution:
  - Des erreurs produites par l'application dans des cas exceptionnels que le programmeur devrait prévoir et traiter dans son application. Ce genre d'erreur sont de type Exception
  - Des erreurs qui peuvent être générées au niveau de la JVM et que le programmeur ne peut prévoir dans son application. Ce type d'erreurs sont de type Error.



---

# Les Exceptions

En Java, on peut classer les exceptions en deux catégories :

- ❑ Les exceptions surveillées,
- ❑ Les exceptions non surveillées.
- Java oblige le programmeur à traiter les erreurs surveillées. Elles sont signalées par le compilateur
- Les erreurs non surveillées peuvent être traitées ou non. Et ne sont pas signalées par le compilateur

# Un premier exemple:

- Considérons une application qui permet de :
  - ❑ Saisir au clavier deux entiers a et b
  - ❑ Faire appel à une fonction qui permet de calculer et de retourner a divisé par b.
  - ❑ Affiche le résultat

```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=calcul(a, b);
        System.out.println("Resultat="+resultat);
    }
}
```

# Exécution

- Nous constatons que le compilateur ne signale pas le cas où b est égal à zero.
- Ce qui constitue un cas fatal pour l'application.
- Voyons ce qui se passe au moment de l'exécution

## Scénario 1 : Le cas normal

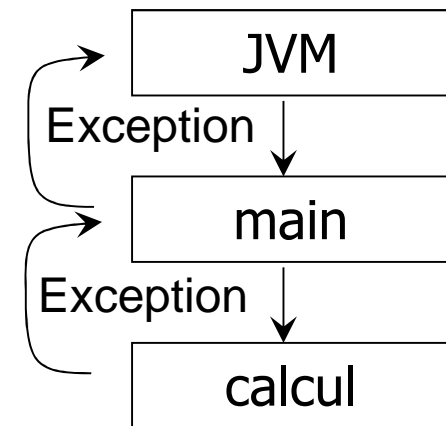
```
Donner a:12  
Donner b:6  
Resultat=2
```

## Scénario 2: cas où b=0

```
Donner a:12  
Donner b:0  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Appl.calcul(App1.java:4)  
at Appl.main(App1.java:11)
```

# Un bug dans l'application

- Le cas du scénario 2 indique que qu'une erreur fatale s'est produite dans l'application au moment de l'exécution.
- Cette exception est de type `ArithmeticException`. Elle concerne une division par zero
- L'origine de cette exception étant la méthode `calcul` dans la ligne numéro 4.
- Cette exception n'a pas été traité dans `calcul`.
- Elle remonte ensuite vers `main` à la ligne numéro 11 dont elle n'a pas été traitée.
- Après l'exception est signalée à la JVM.
- Quand une exception arrive à la JVM, cette dernière arrête l'exécution de l'application, ce qui constitue un bug fatale.
- Le fait que le message « `Resultat=` » n'a pas été affiché, montre que l'application ne continue pas son exécution normal après la division par zero



# Traiter l'exception

- Dans java, pour traiter les exceptions, on doit utiliser le bloc try catch de la manière suivante:

```
import java.util.Scanner;
public class App1 {
    public static int calcul(int a,int b){
        int c=a/b;
        return c;
    }
    public static void main(String[] args) {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Donner a:");int a=clavier.nextInt();
        System.out.print("Donner b:");int b=clavier.nextInt();
        int resultat=0;
        try{
            resultat=calcul(a, b);
        }
        catch (ArithmeticException e) {
            System.out.println("Divisio par zero");
        }
        System.out.println("Resultat="+resultat);
    }
}
```

## Scénario 1

```
Donner a:12
Donner b:6
Resultat=2
```

## Scénario 2

```
Donner a:12
Donner b:0
Divisio par zero
Resultat=0
```

# Principale Méthodes d'une Exception

- Tous les types d'exceptions possèdent les méthodes suivantes :
  - getMessage() : retourne le message de l'exception
    - `System.out.println(e.getMessage());`
    - *Réstat affiché* : / by zero
  - toString() : retourne une chaîne qui contient le type de l'exception et le message de l'exception.
    - `System.out.println(e.toString());`
    - *Réstat affiché* : java.lang.ArithmeticException: / by zero
  - printStackTrace: affiche la trace de l'exception
    - `e.printStackTrace();`
    - *Réstat affiché* :  
java.lang.ArithmeticException: / by zero  
at App1.calcul(App1.java:4)  
at App1.main(App1.java:13)



## Exemple : Générer, Relancer ou Jeter une exception surveillée de type Exception

- Considérons le cas d'un compte qui est défini par un code et un solde et sur lequel, on peut verser un montant, retirer un montant et consulter le solde.

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt)throws Exception{
        if(mt>solde) throw new Exception("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

# Utilisation de la classe Compte

- En faisant appel à la méthode retirer, le compilateur signale que cette dernière peut générer une exception de type Exception
- C'est une Exception surveillée. Elle doit être traitée par le programmeur

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2); // Le comilateur signe l'Exception
}
}
```

# Traiter l'exception

- Deux solutions :

- Soit utiliser le bloc try catch

```
try {  
    cp.retirer(mt2);  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
}
```

- Ou déclarer que cette exception est ignorée dans la méthode main et dans ce cas là, elle remonte vers le niveau supérieur. Dans notre cas la JVM.

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

# Exécution de l'exemple

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    try {
        cp.retirer(mt2);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("Solde Final="+cp.getSolde());
}
}
```

## Scénario 1

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:2000  
Solde Final=3000.0

## Scénario 2

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:7000  
Solde Insuffisant  
Solde Final=5000.0

## Exemple : Générer une exception non surveillée de type RuntimeException

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt){
        if(mt>solde) throw new RuntimeException("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```

# Utilisation de la classe Compte

- En faisant appel à la méthode retirer, le compilateur ne signale rien
- C'est une Exception non surveillée. On est pas obligé de la traiter pour que le programme soit compilé

```
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2); // Le comilateur ne signale rien
}
}
```

# Personnaliser les exception métier.

- L'exception générée dans la méthode retirer, dans le cas où le solde est insuffisant est une exception métier.
- Il est plus professionnel de créer une nouvelle Exception nommée SoldeInsuffisantException de la manière suivante :

```
package metier;  
public class SoldeInsuffisantException extends Exception {  
    public SoldeInsuffisantException(String message) {  
        super(message);  
    }  
}
```

- En héritant de la classe Exception, nous créons une exception surveillée.
- Pour créer une exception non surveillée, vous pouvez hériter de la classe RuntimeException

# Utiliser cette nouvelle exception métier

```
package metier;
public class Compte {
    private int code;
    private float solde;
    public void verser(float mt){
        solde=solde+mt;
    }
    public void retirer(float mt) throws SoldeInsuffisantException{
        if(mt>solde) throw new SoldeInsuffisantException("Solde Insuffisant");
        solde=solde-mt;
    }
    public float getSolde(){
        return solde;
    }
}
```



# Application

```
package pres;
import java.util.Scanner;
import metier.Compte;
import metier.SoldeInsuffisantException;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    try {
    cp.retirer(mt2);
    } catch (SoldeInsuffisantException e) {
    System.out.println(e.getMessage());
    }
    System.out.println("Solde Final="+cp.getSolde());
}
}
```

## Scénario 1

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0
```

## Scénario 2

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```

## Scénario 3

```
Montant à verser:azerty
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:840)
at java.util.Scanner.next(Scanner.java:1461)
at
java.util.Scanner.nextFloat(Scanner.java:2319)
at pres.Application.main(Application.java:9)
```

---

# Améliorer l'application

- Dans le scénario 3, nous découvrons qu'une autre exception non surveillée est générée dans le cas où on saisie une chaîne de caractères et non pas un nombre.
- Cette exception est de type `InputMismatchException`, générée par la méthode `nextFloat()` de la classe `Scanner`.
- Nous devrions donc faire plusieurs `catch` dans la méthode `main`

# Application

```
package pres;
import java.util.InputMismatchException;import java.util.Scanner;
import metier.Compte;import metier.SoldeInsuffisantException;
public class Application {
public static void main(String[] args) {
    Compte cp=new Compte();
    try {
        Scanner clavier=new Scanner(System.in);
        System.out.print("Montant à verser:");
        float mt1=clavier.nextFloat();
        cp.verser(mt1);
        System.out.println("Solde Actuel:"+cp.getSolde());
        System.out.print("Montant à retirer:");
        float mt2=clavier.nextFloat();
        cp.retirer(mt2);
    }
    catch (SoldeInsuffisantException e) {
        System.out.println(e.getMessage());
    }
    catch (InputMismatchException e) {
        System.out.println("Problème de saisie");
    }
    System.out.println("Solde Final="+cp.getSolde());
}
```

---

## Un autre cas exceptionnel dans la méthode retirer

- Supposons que l'on ne doit pas accepter un montant négatif dans la méthode retirer.
- On devrait générer une exception de type `MontantNegatifException`.
- Il faut d'abord créer cette nouvelle exception métier.

# Le cas MontantNegatifException

- L'exception métier MontantNegatifException

```
package metier;  
public class MontantNegatifException extends Exception {  
    public MontantNegatifException(String message) {  
        super(message);  
    }  
}
```

- La méthode retirer de la classe Compte

```
public void retirer(float mt) throws  
SoldeInsuffisantException, MontantNegatifException {  
    if (mt < 0) throw new MontantNegatifException("Montant "+mt+"  
négatif");  
    if (mt > solde) throw new SoldeInsuffisantException("Solde  
Insuffisant");  
    solde = solde - mt;  
}
```

# Application : Contenu de la méthode main

```
Compte cp=new Compte();
try {
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2);
}
catch (SoldeInsuffisantException e) {
    System.out.println(e.getMessage());
}
catch (InputMismatchException e) {
    System.out.println("Problème de saisie");
}
catch (MontantNegatifException e) {
    System.out.println(e.getMessage());
}
System.out.println("Solde Final="+cp.getSolde());
```

## Scénario 1

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:2000  
Solde Final=3000.0

## Scénario 2

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:7000  
Solde Insuffisant  
Solde Final=5000.0

## Scénario 3

Montant à verser:5000  
Solde Actuel:5000.0  
Montant à retirer:-2000  
Montant -2000.0 négatif  
Solde Final=5000.0

## Scénario 4

Montant à verser:azerty  
Problème de saisie  
Solde Final=0.0

# Le bloc finally

- La syntaxe complète du bloc try est la suivante :

```
try {  
    System.out.println("Traitement Normale");  
}  
catch (SoldeInsuffisantException e) {  
    System.out.println("Premier cas Exceptionnel");  
}  
catch (NegativeArraySizeException e) {  
    System.out.println("dexuième cas Exceptionnel");  
}  
finally{  
    System.out.println("Traitement par défaut!");  
}  
System.out.println("Suite du programme!");
```

- Le bloc **finally** s'exécute quelque soit les différents scénarios.

---

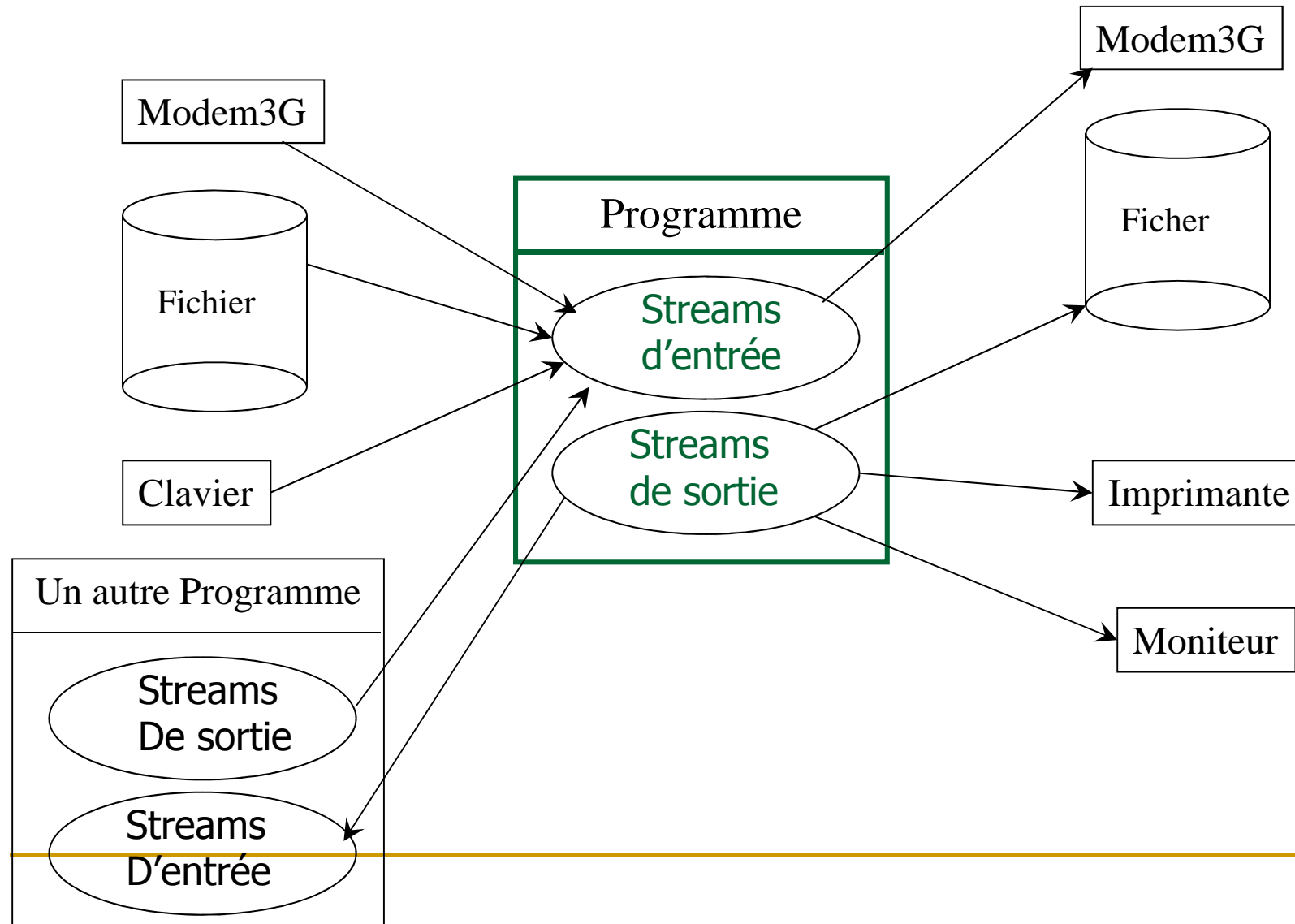
# Entrés Sorties

---





# Entrées Sorties



---

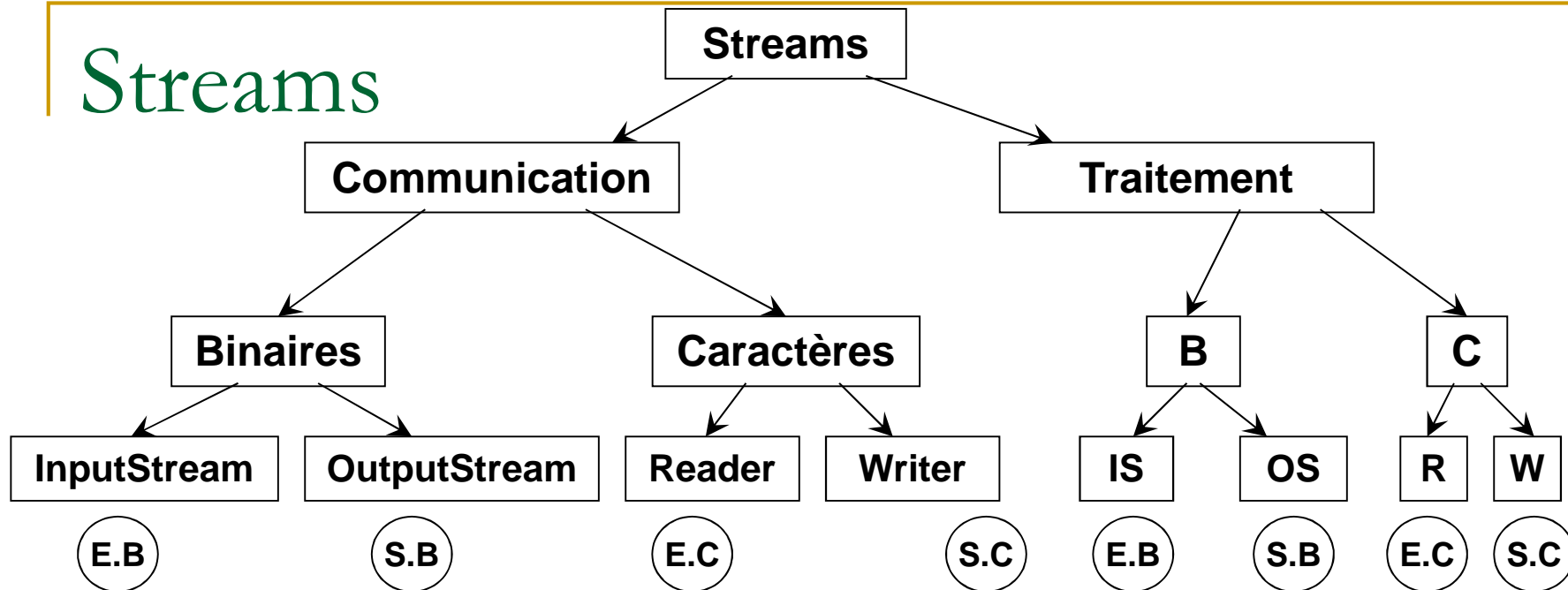
# Principe des entrées/sorties

Pour effectuer une entrée ou une sortie de données en Java, le principe est simple et se résume aux opérations suivantes :

- Ouverture d'un moyen de communication.
- Écriture ou lecture des données.
- Fermeture du moyen de communication.

En Java, les moyens de communication sont représentés par des objets particuliers appelés (en anglais) *stream*. Ce mot, qui signifie *courant*, ou *flot*

# Streams



- Il existe deux types de streams:
  - Streams de Communication: établit une liaison entre le programme et une destination.
    - Streams binaires : Exemple **FileInputStream**, **FileOutputStream**
    - Streams de caractères : Exemple **FileReader**, **FileWriter**
  - Streams de Traitement : Permet de traiter les information des streams de communication.
    - Streams binaires : ex **BufferedInputStream**, **ZipInputStream**, **ZipOutputStream**,...
    - Streams de caractères : **BufferedReader**, **BufferedWriter**, ...

---

# La classe File

- La classe File permet de donner des informations sur un fichier ou un répertoire
- La création d'un objet de la classe File peut se faire de différentes manières :
  - `File f1=new File("c:/projet/fichier.ext");`
  - `File f2=new File("c:/projet", "fichier.ext");`
  - `File f3=new File("c:/projet");`

# Principales méthodes de la classe File

<code>String getName();</code>	Retourne le nom du fichier.
<code>String getPath();</code>	Retourne la localisation du fichier en relatif.
<code>String getAbsolutePath();</code>	Idem mais en absolu.
<code>String getParent();</code>	Retourne le nom du répertoire parent.
<code>boolean renameTo(File newFile);</code>	Permet de renommer un fichier.
<code>boolean exists() ;</code>	Est-ce que le fichier existe ?
<code>boolean canRead();</code>	Le fichier est t-il lisible ?
<code>boolean canWrite();</code>	Le fichier est t-il modifiable ?
<code>boolean isDirectory();</code>	Permet de savoir si c'est un répertoire.
<code>boolean isFile();</code>	Permet de savoir si c'est un fichier.
<code>long length();</code>	Quelle est sa longueur (en octets) ?
<code>boolean delete();</code>	Permet d'effacer le fichier.
<code>boolean mkdir();</code>	Permet de créer un répertoire.
<code>String[] list();</code>	On demande la liste des fichiers localisés dans le répertoire.

# Premier Exemple d'utilisation de la classe File

- Afficher le contenu d'un répertoire en affichant si les éléments de ce répertoire sont des fichiers ou des répertoires.
- Dans le cas où il s'agit d'un fichier afficher la capacité physique du fichier.

```
import java.io.File;
public class Application1 {
public static void main(String[] args) {
    String rep="c:/"; File f=new File(rep);
    if(f.exists()){
        String[] contenu=f.list();
        for(int i=0;i<contenu.length;i++){
            File f2=new File(rep,contenu[i]);
            if(f2.isDirectory())
                System.out.println("REP:"+contenu[i]);
            else
                System.out.println("Fichier :"+contenu[i]+"
Size:"+contenu[i].length());
        }
    }
    else{
        System.out.println(rep+" n'existe pas");
    }
}}
```

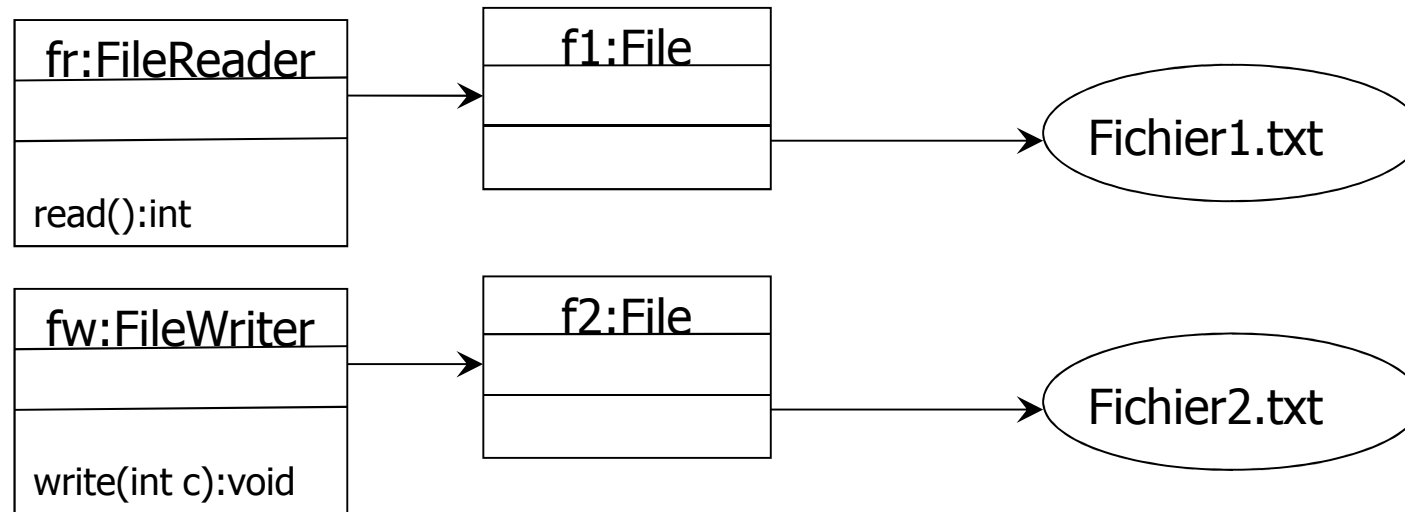
```
Fichier :aff2 (2).asm Size:12
Fichier :aff2.asm Size:8
REP:And
REP:androideProjets
REP:AP
REP:APP
Fichier :aqua_bitmap.cpp
Size:15
Fichier :autoexec.bat Size:12
```

---

## Exercice 1

- Ecrire une application java qui permet d'afficher le contenu d'un répertoire y compris le contenu de ses sous répertoires

# Lire et Écrire sur un fichier texte



```
File f1=new File("c:/Fichier1.txt");
FileReader fr=new FileReader(f1);
File f2=new File("c:/Fichier2.txt");
FileWriter fw=new FileWriter(f2);
int c;
while((c=fr.read())!=-1){
    fw.write(c);
}
fr.close();fw.close();
```



# Exemple 1 : Crypter un fichier Texte

- Considérons un fichier texte nommé « operations.txt » qui contient les opérations sur un compte.
- Chaque ligne de ce fichier représente une opérations qui est définie par :
  - le numéro de l'opération,
  - le numéro de compte,
  - la date d'opération,
  - le type d'opération (Versement ou Retrait)
  - et le montant de l'opération.
- Voici un exemple de fichier :  
`321;CC1;2011-01-11;V;4500`  
`512;CC1;2011-01-11;V;26000`  
`623;CC1;2011-01-11;R;9000`  
`815;CC1;2011-01-11;R;2500`

# Exemple 1 : Crypter un fichier Texte

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
public class App2 {
    public static void main(String[] args) throws Exception {
        File f1=new File("operations.txt");
        FileReader fr=new FileReader(f1);
        File f2=new File("operationsCryptes.txt");
        FileWriter fw=new FileWriter(f2);
        int c;
        while((c=fr.read())!=-1){
            fw.write(c+1);
        }
        fr.close();
        fw.close();
    }
}
```

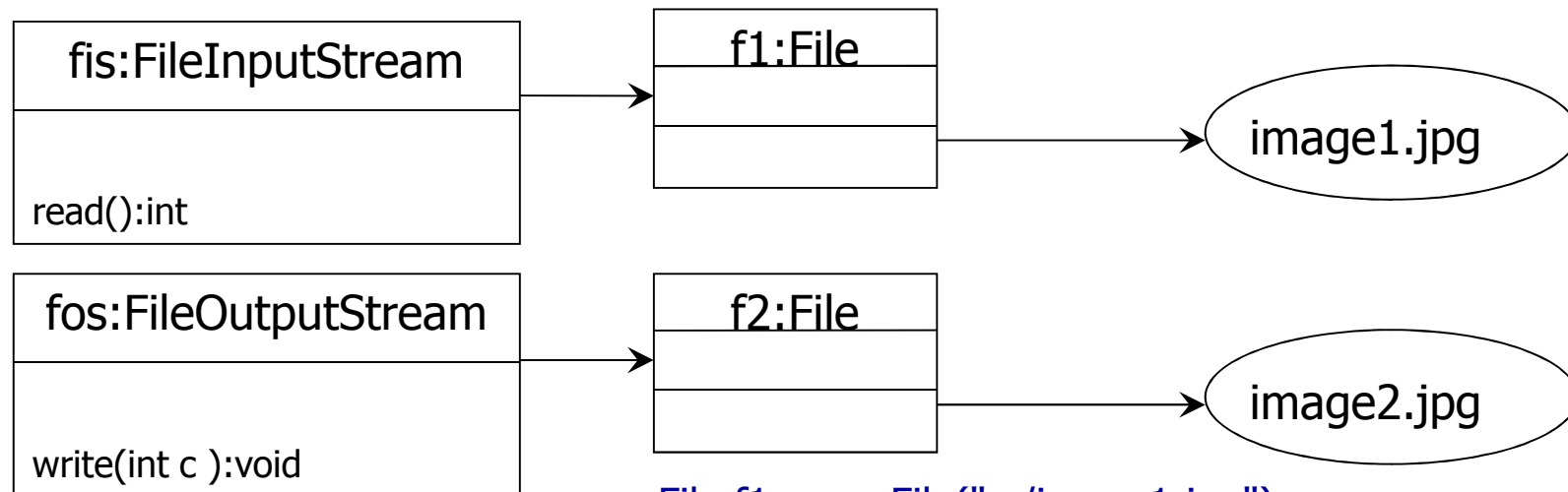
operations.txt

```
321;CC1;2011-01-11;V;4500
512;CC1;2011-01-11;V;26000
623;CC1;2011-01-11;R;9000
815;CC1;2011-01-11;R;2500
```

operationsCryptes.txt

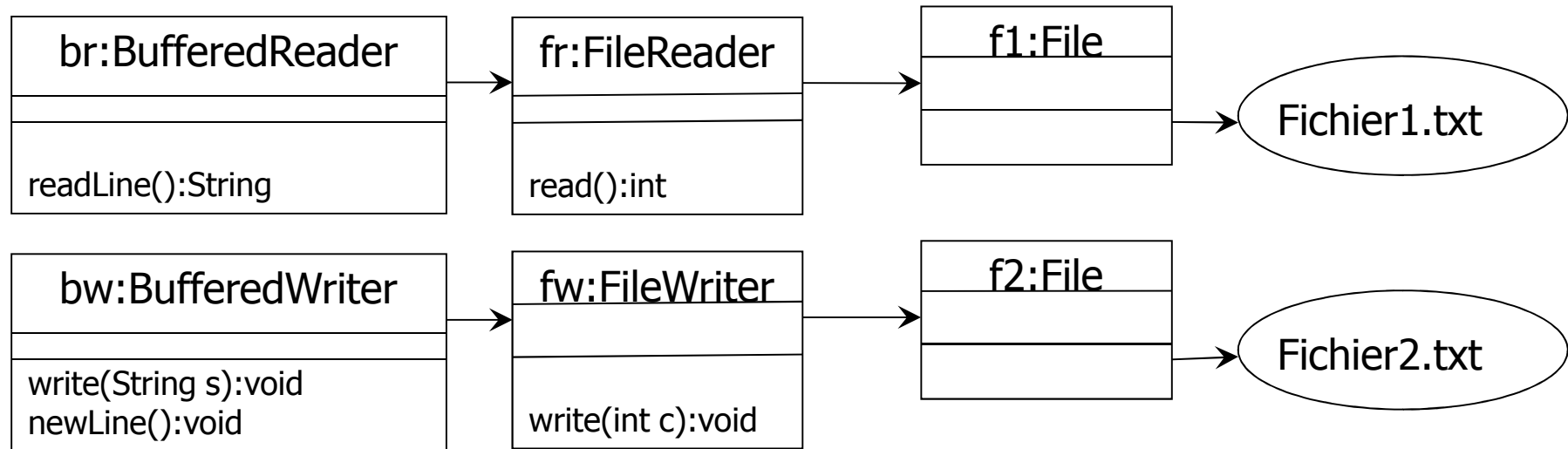
```
432<DD2<3122.12.22<W<5611623<DD2<3122.12.22<W<37111734<DD2<
3122.12.22<S<:111926<DD2<3122.12.22<S<3611
```

# Lire et Écrire sur un fichier binaire



```
File f1=new File("c:/image1.jpg");
FileInputStream fis=new FileInputStream(f1);
File f2=new File("c:/image2.jpg");
FileOutputStream fos=new FileOutputStream(f2);
int c;
while((c=fis.read())!=-1){
    fos.write(c);
}
fis.close();fos.close();
```

## Lire et Écrire sur un fichier texte ligne par ligne : Streams de traitement : BufferedReader et BufferedWriter

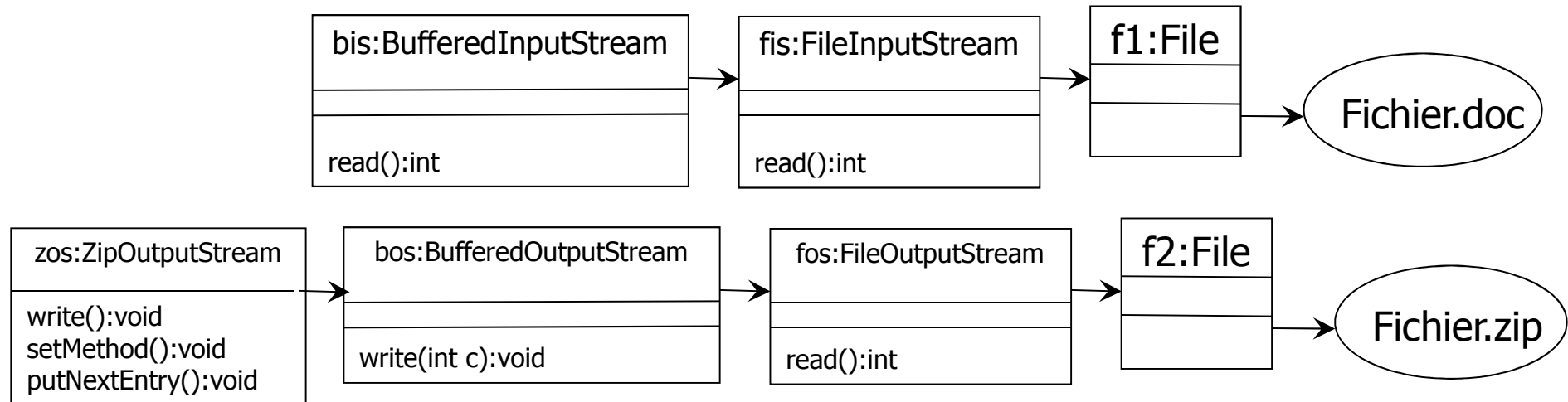


```
File f1=new File("c:/Fichier1.txt");
FileReader fr=new FileReader(f1);
BufferedReader br=new BufferedReader(fr);
File f2=new File("c:/Fichier2.txt");
FileWriter fw=new FileWriter(f2);
BufferedWriter bw=new BufferedWriter(fw);
String s;
while((s=br.readLine())!=null){
    bw.write(s);bw.newLine();
}
br.close();bw.close();
```

## Exemple 3 : Afficher le total des opérations de versements et de retraits

```
import java.io.*;
public class App3 {
public static void main(String[] args) throws Exception {
    File f=new File("operations.txt");
    FileReader fr=new FileReader(f);
    BufferedReader br=new BufferedReader(fr);
    String op;double totalVersements=0;double totalRetraits=0;
    while((op=br.readLine())!=null){
        String[] tabOp=op.split(";");
        String typeOp=tabOp[3];
        double montant=Double.parseDouble(tabOp[4]);
        if(typeOp.equals("V"))
            totalVersements+=montant;
        else
            totalRetraits+=montant;
    }
    System.out.println("Total Versement:"+totalVersements);
    System.out.println("Total Retrait:"+totalRetraits);
}
}
```

## Compression ZIP : Stream de traitement : ZipOutputStream

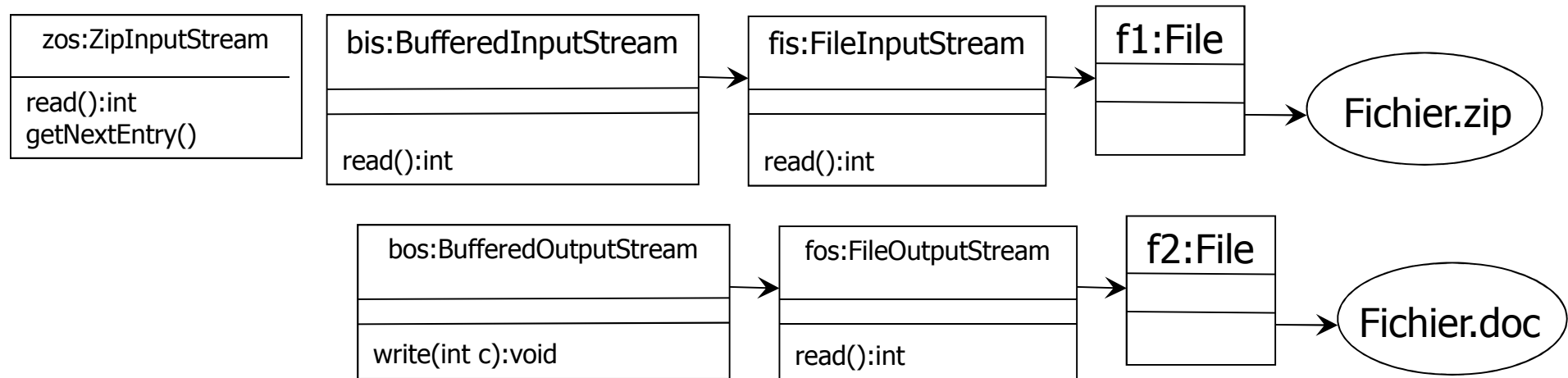


```
File f1=new File("c:/","Fichier.doc");
FileInputStream fis=new FileInputStream(f1);
BufferedInputStream bis=new BufferedInputStream(fis);
File f2=new File("Fichier.zip");
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
ZipOutputStream zos=new ZipOutputStream(bos);
zos.setMethod(ZipOutputStream.DEFLATED);
zos.putNextEntry(new ZipEntry(f1.getName()));
int c;
while ((c=bis.read())!=-1){
    zos.write(c);
}
zos.close();bis.close();
```

## Exemple 4 : Compression d'un fichier

```
import java.io.*;import java.util.zip.*;
public class App4 {
public static void main(String[] args) throws Exception {
    File f1=new File("fichier.doc");
    FileInputStream fis=new FileInputStream(f1);
    BufferedInputStream bis=new BufferedInputStream(fis);
    File f2=new File("fichier.zip");
    FileOutputStream fos=new FileOutputStream(f2);
    BufferedOutputStream bos=new BufferedOutputStream(fos);
    ZipOutputStream zos=new ZipOutputStream(bos);
    zos.setMethod(ZipOutputStream.DEFLATED);
    zos.putNextEntry(new ZipEntry(f1.getName()));
    int c;
    while((c=bis.read())!=-1){
        zos.write(c);
    }
    bis.close(); zos.close();
    System.out.println("Capacite de "+f1.getName()+" est :"+f1.length());
    System.out.println("Capacite de "+f2.getName()+" est :"+f2.length());
}}
```

# Décompression ZIP : Stream de traitement : ZipInputStream



```
File f1=new File("Fichier.zip");
FileInputStream fis=new FileInputStream(f1);
BufferedInputStream bis=new BufferedInputStream(fis);
ZipInputStream zis=new ZipInputStream(bis);
ZipEntry ze=zis.getNextEntry();
File f2=new File(ze.getName());
FileOutputStream fos=new FileOutputStream(f2);
BufferedOutputStream bos=new BufferedOutputStream(fos);
int c;
while ((c=zis.read())!=-1){
    bos.write(c);
}
zis.close();bos.close();
```



## Exemple 5 : Décompression d'un fichier zip

```
import java.io.*;import java.util.zip.*;
public class App5 {
public static void main(String[] args)throws Exception {
    File f1=new File("fichier.zip");
    FileInputStream fis=new FileInputStream(f1);
    BufferedInputStream bis=new BufferedInputStream(fis);
    ZipInputStream zis=new ZipInputStream(bis);
    ZipEntry ze=zis.getNextEntry();
    File f2=new File(ze.getName());
    FileOutputStream fos=new FileOutputStream(f2);
    BufferedOutputStream bos=new BufferedOutputStream(fos);
    int c;
    while((c=zis.read())!=-1){
        bos.write(c);
    }
    zis.close(); bos.close();
    System.out.println("Capacite de "+f1.getName()+" est :"+f1.Length());
    System.out.println("Capacite de "+f2.getName()+" est :"+f2.Length());
}}
```

---

## Exercice 2

- Créer une application java qui permet de compresser le contenu d'un répertoire y compris le contenu de ses sous répertoires
- Créer une autre application java qui permet de décompresser un fichier ZIP

# La sérialisation

- La sérialisation est une opérations qui permet d'envoyer un objet sous forme d'une tableau d'octets dans une sortie quelconque (Fichier, réseau, port série etc..)
- Les applications distribuées utilisent beaucoup ce concept pour échanger les objets java entre les applications via le réseau.
- Pour sérialiser un objet, on utiliser la méthode **writeObject()** de la classe **ObjectOutputStream**
- La désérialisation est une opération qui permet de reconstruire l'objet à partir d'une série d'octets récupérés à partir d'une entrée quelconque.
- Pour dé sérialiser un objet, on utilise la méthode **readObject()** de la classe **ObjectInputStream**.
- Pour pouvoir sérialiser un objet, sa classe doit implémenter l'interface **Serializable**
- Pour designer les attributs d'un objet qui ne doivent pas être sérialisés, on doit les déclarer **transient**

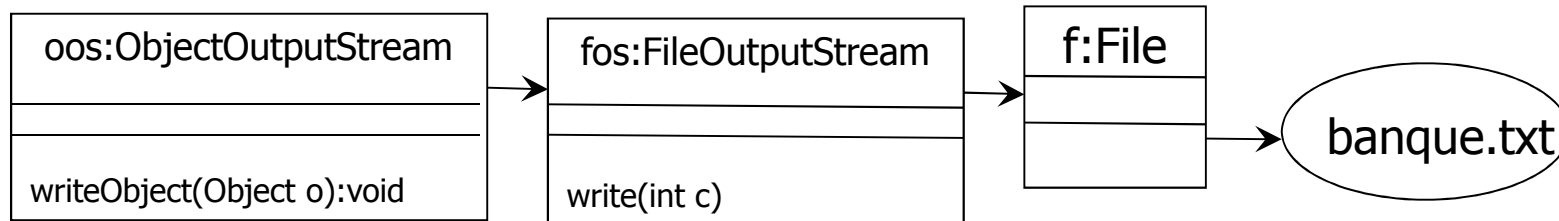
# Exemple d'une classe Serializable

```
package metier;
import java.io.Serializable;
import java.util.Date;
public class Operation implements Serializable {
    private int numeroOperation;
    private transient Date dateOperation;
    private String numeroCompte;
    private String typeOperation;
    private double montant;

    public Operation() { }

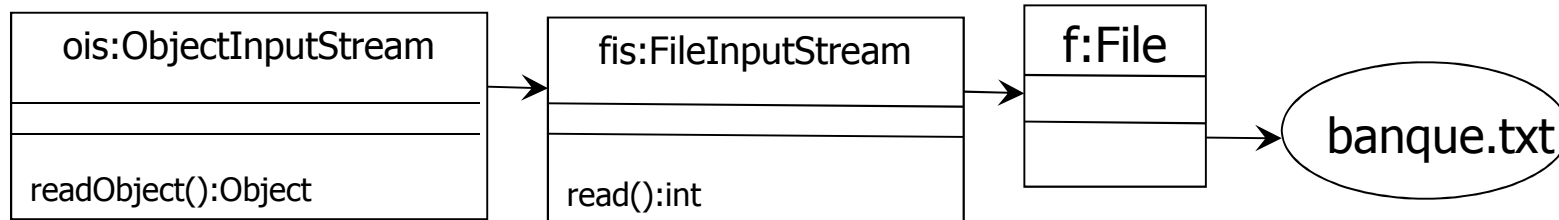
    public Operation(int numOp, Date dateOp, String numC,String to, double mt) {
        this.numeroOperation = numOp;this.dateOperation = dateOp;
        this.numeroCompte = numC;this.typeOperation = to;
        this.montant = mt;
    }
    // Getters et Setters
}
```

## Sérialisation



```
import java.io.*;import java.util.Date;
import metier.Operation;
public class Serialisation {
public static void main(String[] args) throws Exception {
    Operation op1=new Operation(1,new Date(), "CC1", "V", 40000);
    Operation op2=new Operation(2,new Date(), "CC1", "R", 6000);
    File f=new File("banque.txt");
    FileOutputStream fos=new FileOutputStream(f);
    ObjectOutputStream oos=new ObjectOutputStream(fos);
    oos.writeObject(op1);
    oos.writeObject(op2);
    oos.close();
}
}
```

## DéSérialisation



```
import java.io.*;
import metier.Operation;
public class Deserialisation {
    public static void main(String[] args) throws Exception {
        File f=new File("banque.txt");
        FileInputStream fis=new FileInputStream(f);
        ObjectInputStream ois=new ObjectInputStream(fis);
        Operation op1=(Operation) ois.readObject();
        Operation op2=(Operation) ois.readObject();
        System.out.println("Num:"+op1.getNumeroOperation());
        System.out.println("Date:"+op1.getDateOperation());
        System.out.println("Compte:"+op1.getNumeroCompte());
        System.out.println("Type:"+op1.getTypeOperation());
        System.out.println("Montant:"+op1.getMontant());
    }
}
```

---

## Exercice 3

- Créer une application java qui permet de sérialiser les objets de la classe opérations dans un fichier compressé au format zip.
- Créer une application qui permet de dé sérialiser ces objets du fichier compressé.

---

# Interfaces graphique dans java

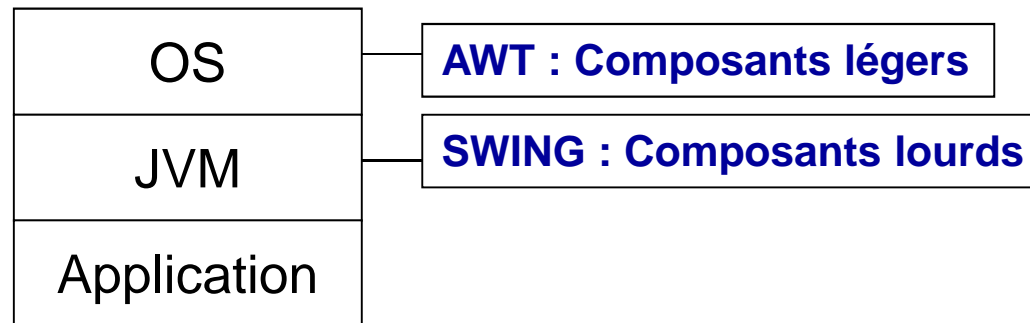
---





# Interfaces graphiques

- En java, il existe deux types de composants graphiques:
  - ❑ Composants **AWT** (Abstract Window ToolKit): Composants qui font appel aux composants graphiques de l'OS
  - ❑ Composants **SWING** : Composants écrit complètement avec java et sont indépendant de l'OS



---

# Composants AWT

Nous avons à notre disposition principalement trois types d'objets :

- ❑ Les **Components** qui sont des composants graphiques. Exemple (Button, Label, TextField...)
- ❑ Les **Containers** qui contiennent les Components. Exemple (Frame, Pannel, ....)
- ❑ Les **Layouts** qui sont en fait des stratégies de placement de Components pour les Containers. Exemple (FlowLayout, BorderLayout, GridLayout...)

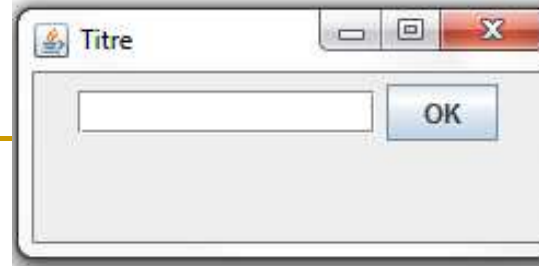
# Premier Exemple AWT et SWING

```
import java.awt.*;
public class AppAWT {
    public static void main(String[] args){
        // Créer une fenêtre
        Frame f=new Frame("Titre");
        // Créer une zone de texte
        TextField t=new TextField(12);
        // Créer un bouton OK
        Button b=new Button("OK");
        // Définir une technique de placement
        f.setLayout(new FlowLayout());
        //ajouter la zone de texte t à frame f
        f.add(t);
        // ajouter le bouton b à la frame f
        f.add(b);
        // Définir les dimensions de la frame
        f.setBounds(10,10,500,500);
        // Afficher la frame
        f.setVisible(true);
    }
}
```

```
import javax.swing.*;
import java.awt.FlowLayout;
public class AppSWING {
    public static void main(String[] args){
        JFrame f=new JFrame("Titre");
        JTextField t=new JTextField(12);
        JButton b=new JButton("OK");
        f.setLayout(new FlowLayout());
        f.add(t); f.add(b);
        f.setBounds(10,10,500,500);
        f.setVisible(true);
    }
}
```



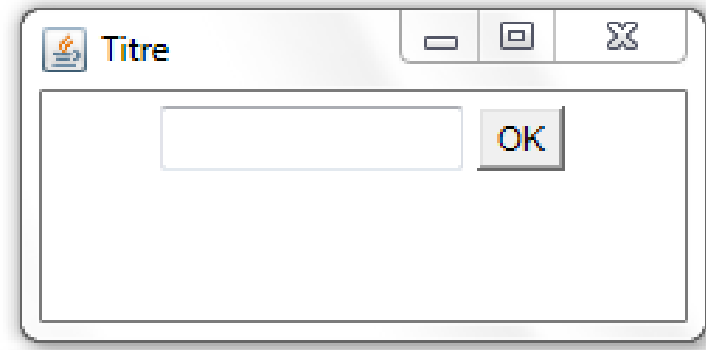
AWT



SWING

# Une autre manière plus pratique : Hérier de la classe Frame

```
import java.awt.*;
public class MaFenetreAWT extends Frame{
    // Créer une zone de texte
    TextField t=new TextField(12);
    // Créer un bouton OK
    Button b=new Button("OK");
    public MaFenetreAWT() {
        // Définir une technique de placement
        this.setLayout(new FlowLayout());
        //ajouter la zone de texte t à la frame this
        this.add(t);
        // ajouter le bouton b à la frame this
        this.add(b);
        // Définir les dimensions de la frame
        this.setBounds(10,10,500,500);
        // Afficher la frame
        this.setVisible(true);
    }
    public static void main(String[] args) {
        MaFenetreAWT f=new MaFenetreAWT();
    }
}
```



---

# Gestion des événements

- Dans java, pour q'un objet puisse répondre à un événement, il faut lui attacher un écouteur (Listener).
- Il existe différents types de Listeners:
  - ❑ WindowListener : pour gérer les événements sur la fenêtre
  - ❑ ActionListener : pour gérer les événements produits sur les composants graphiques.
  - ❑ KeyListener : pour gérer les événements du clavier
  - ❑ MouseListener : pour gérer les événements de la souris.
  - ❑ ....

---

# Gestionnaire d'événement ActionListener

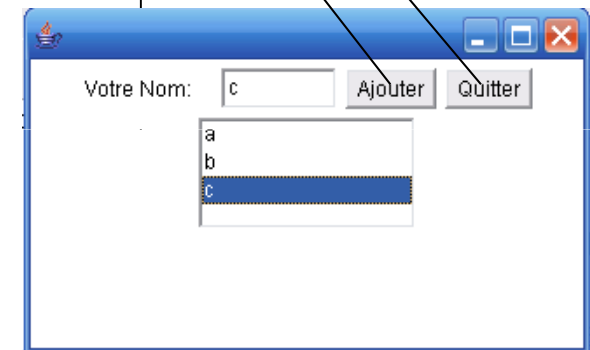
- ActionListener est une interface qui définit une seule méthode:
  - `public void actionPerformed(ActionEvent e);`
- L'événement `actionPerformed` est produit quand on valide une action par un clique ou par la touche de validation du clavier.
- Pour gérer cet événement dans une application, il faut créer une classe implémentant l'interface `ActionListener` et redéfinir la réponse aux événements utilisateur, produits dans l'interface, dans la méthode `actionPerformed`.
- La classe implémentant cette interface s'appelle un listener (écouteur) ou un gestionnaire d'événements.
- Quand on clique, par exemple, sur un bouton qui est branché sur cet écouteur, la méthode `actionPerformed` de l'écouteur s'exécute

# Gestion des événements

```
import java.awt.*;import java.awt.event.*;
public class TestEvent extends Frame implements ActionListener
{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField(12);
    Button b=new Button("Ajouter"); List liste=new List();
    Button b2=new Button("Quitter");
    public TestEvent() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        b.addActionListener(this);
        b2.addActionListener(this);
        this.setVisible(true);    }
    // Gestionnaire des événements
    public void actionPerformed(ActionEvent e) {
        // Si la source de l'événement est le bouton b
        if(e.getSource()==b){
            // Lire le contenu de la zone de texte
            String s=t.getText();
            // Ajouter la valeur de s à la liste
            liste.add(s);}
        // Si la source de l'événement est b2
        else if(e.getSource()==b2){
            // Quitter l'application
            System.exit(0);    }
    }
    public static void main(String[] args) {
        new TestEvent();
    }
}
```

: ActionListener

actionPerformed()

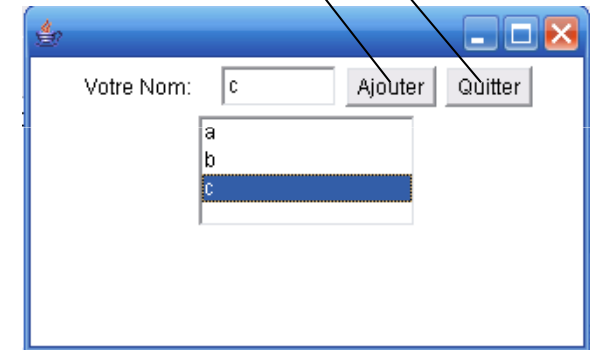


# En utilisant une classe interne

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class TestEvent2 extends Frame{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField("*****");
    Button b=new Button("Ajouter");
    List liste=new List();
    Button b2=new Button("Quitter");
    class Handler implements ActionListener{
        public void actionPerformed(ActionEvent e) {
            if(e.getSource()==b){
                String s=t.getText();
                liste.add(s);
            }
            else if(e.getSource()==b2){
                System.exit(0);
            }
        }
    }
    public TestEvent2() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        Handler h=new Handler();
        b.addActionListener(h);
        b2.addActionListener(h);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new TestEvent2();
    }
}
```

h: ActionListener

public void actionPerformed(ActionEvent e)

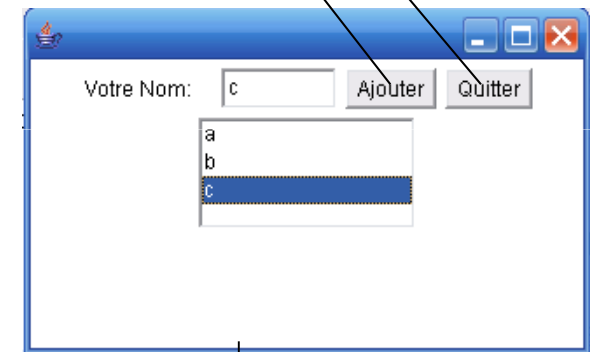




# En créant directement un objet de ActionListener

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class TestEvent extends Frame{
    Label l=new Label("Votre Nom:");
    TextField t=new TextField(12);
    Button b=new Button("Ajouter");
    List liste=new List();
    Button b2=new Button("Quitter");
    public TestEvent() {
        this.setLayout(new FlowLayout());
        this.add(l);this.add(t);this.add(b);
        this.add(b2);this.add(liste);
        this.setBounds(10,10, 400, 400);
        // Si on clique le bouton b1
        b.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent e) {
                    String s=t.getText();
                    liste.add(s);
                }
            }
        );
        // Si on clique le bouton b2
        b2.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            }
        );
        // Si on clique le bouton fermer de la fenêtre
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new TestEvent();
    }
}
```

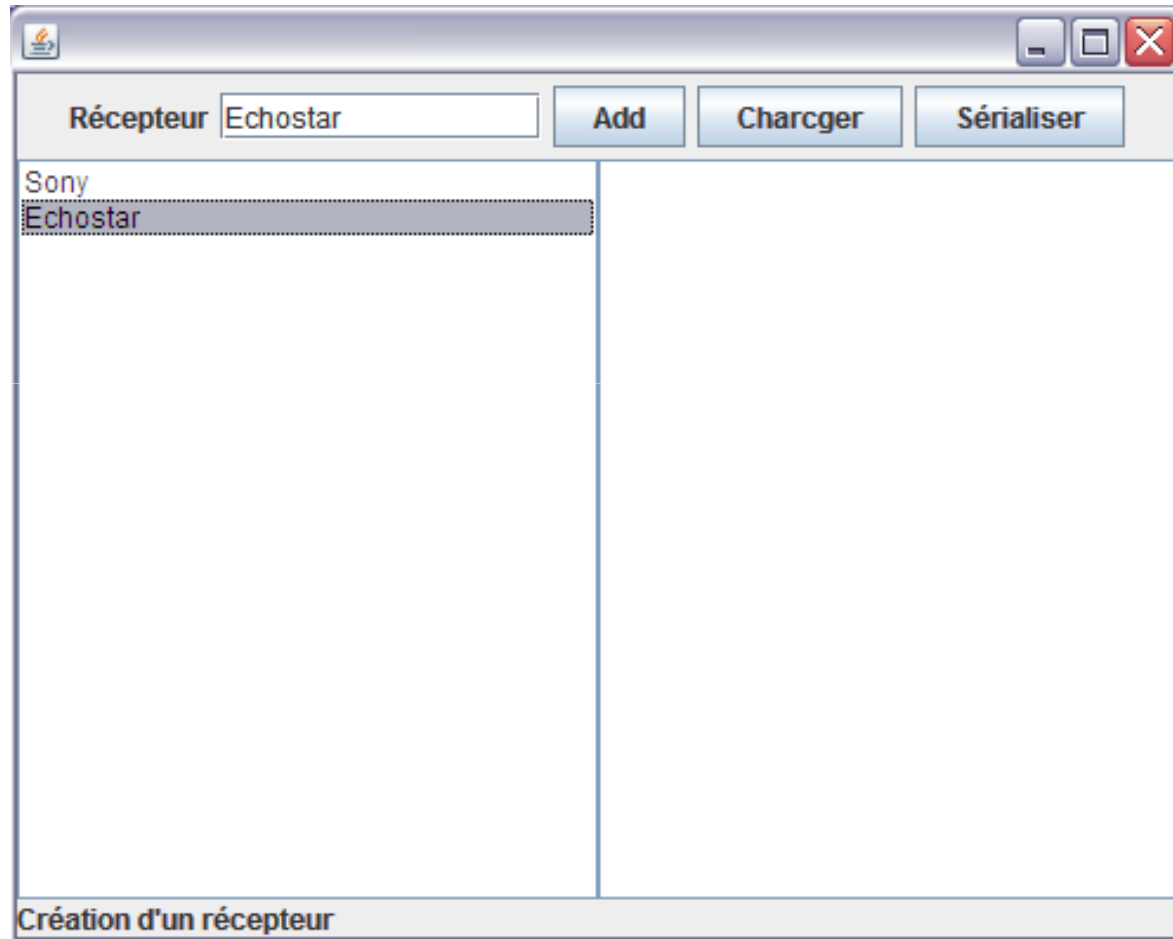
: ActionListener
public void actionPerformed(ActionEvent e)



: WindowListener
public void windowClosing(WindowEvent e)

# Exemple d'application SWING

- Interface concernant le TP entrée sorties (Récepteur)

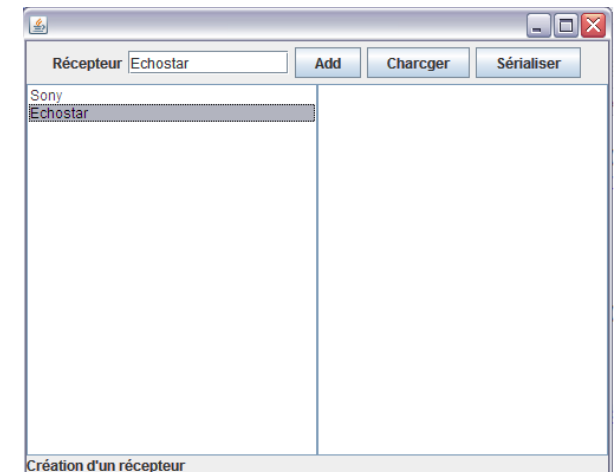


# Code de l'application : Création des composants

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

import javax.swing.*;

public class RecepteurSWING extends JFrame implements ActionListener
{
    private JLabel jl=new JLabel("Récepteur");
    private JTextField jtfl=new JTextField(12);
    private JButton jb1=new JButton("Add");
    private JButton jb2=new JButton("Charger");
    private JButton jb3=new JButton("Sérialiser");
    private List liste1=new List();
    private List liste2=new List();
    private JLabel jlErr=new JLabel("OK");
```



# Code de l'application : Constructeur

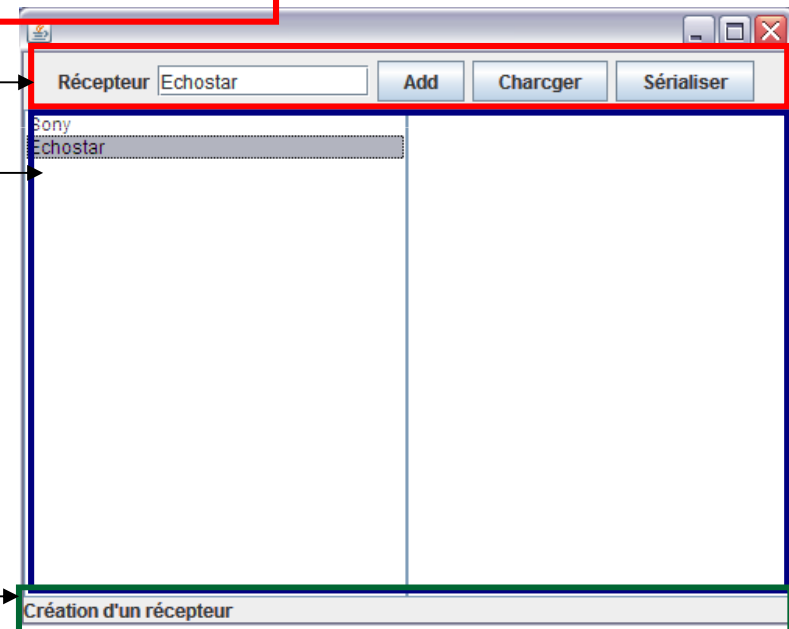
```
public RecepteurSWING(){  
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);  
    this.setLayout(new BorderLayout());
```

```
    JPanel jp1=new JPanel();  
    jp1.setLayout(new FlowLayout());  
    jp1.add(jl);jp1.add(jtff1);jp1.add(jb1);jp1.add(jb2);  
    jp1.add(jb3);  
    this.add(jp1,BorderLayout.NORTH);
```

```
    JPanel jp2=new JPanel();  
    jp2.setLayout(new GridLayout());  
    jp2.add(liste1);  
    jp2.add(liste2);  
    this.add(jp2,BorderLayout.CENTER);
```

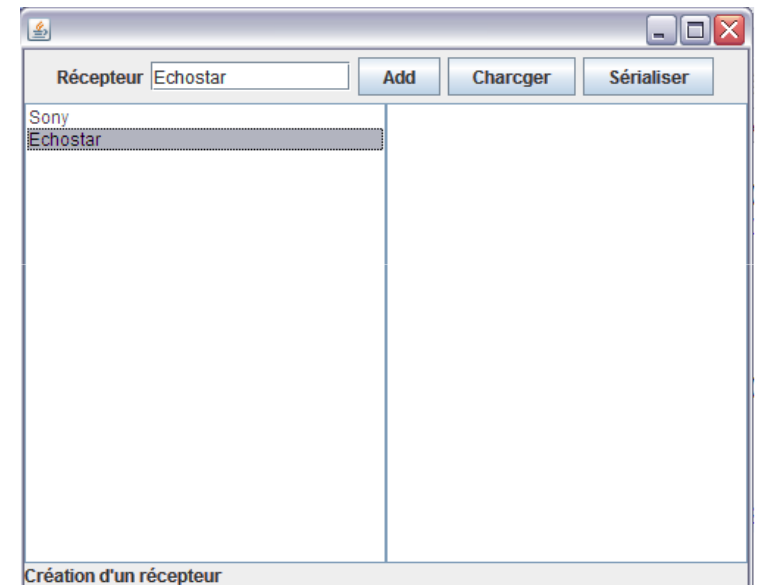
```
    this.add(jlErr,BorderLayout.SOUTH);
```

```
    jb1.addActionListener(this);  
    jb2.addActionListener(this);  
    jb3.addActionListener(this);  
    this.setBounds(10, 10, 500, 400);  
    this.setVisible(true);  
}
```



# Code de l'application : Réponse aux événements

```
public void actionPerformed(ActionEvent e) {  
    if(e.getSource()==jb1){  
        String n=jtfl.getText();  
        listel.add(n);  
        jlErr.setText("Création d'un récepteur");  
        /* A compléter */  
    }  
    else if(e.getSource()==jb2){  
        jlErr.setText("Chargement des chaines");  
        /* A compléter */  
    }  
    else if(e.getSource()==jb3){  
        /* A compléter */  
        jlErr.setText("Sérialisation");  
    }  
}  
  
public static void main(String[] args) {  
    new RecepteurSWING();  
}  
}
```



---

# Application

- On souhaite créer une application java qui permet gérer une société de transport de cargaisons transportant des marchandises. La société gère un ensemble de cargaisons. Chaque cargaison contient plusieurs marchandises. Chaque marchandise est définie par son numéro, son poids et son volume.
- Il existe deux types de cargaisons : Routière et Aérienne. Chaque cargaison est définie par sa référence et sa distance de parcours. Le cout de transport d'une cargaison est calculé en fonction du type de la cargaison.
- Une cargaison aérienne est une cargaison dont le cout est calculé selon la formule suivante :
  - $\text{cout} = 10 \times \text{distance} \times \text{poids total des marchandises}$  si le volume total est inférieur à 80000
  - $\text{cout} = 12 \times \text{distance} \times \text{poids total des marchandises}$  si le volume total est supérieur ou égal à 80000
- Une cargaison routière est une cargaison dont le cout est calculé selon la formule suivante :
  - $\text{cout} = 4 \times \text{distance} \times \text{poids total}$  si le volume total est inférieur à 380000
  - $\text{cout} = 6 \times \text{distance} \times \text{poids total}$  si le volume total est supérieur ou égale à 380000

---

# Application

- Pour chaque cargaison, on souhaite ajouter une marchandise, supprimer une marchandise, consulter une marchandise sachant son numéro, consulter toutes les marchandises de la cargaison, consulter le poids total de la cargaison, consulter le volume total de la cargaison et consulter le cout de la cargaison.
  - Cette application peut être utilisée par les clients et les administrateurs.
  - Le client peut effectuer les opérations suivantes :
    - Consulter une cargaison sachant sa référence.
    - Consulter une marchandise sachant son numéro.
    - Lire le fichier Cargaisons.
    - Consulter toutes les cargaisons.
  - L'administrateur peut effectuer toutes les opérations effectuées par le client. En plus, il peut :
    - Ajouter une nouvelle cargaison.
    - Ajouter une marchandise à une cargaison.
    - Supprimer une cargaison
    - Enregistrer les cargaisons dans un fichier.
  - Toutes les opérations nécessitent une authentification
-

# Questions :

1. Etablir un diagramme Use case UML.
2. Etablir le diagramme de classes en prenant en considération les critères suivants.
  - ❑ La classe SocieteTransport devrait implémenter les deux interfaces IClientTransport et IAdminTransport déclarant, respectivement les opérations relatives aux rôles Client et Admin.
  - ❑ Dans une première implémentation de SocieteTransport, on suppose que les cargaisons sont stockées dans une liste de type HashMap de la classe SocieteTransport.
  - ❑ Dans une deuxième implémentation, nous supposerons que les cargaisons et les marchandises sont stockées dans une base de données relationnelle.
  - ❑ L'association entre cargaison et Marchandise est bidirectionnelle.
3. Ecrire le code java des classes entités Marchandise, Cargaison, CargaisonRoutière et CargaisonAérienne
4. Ecrire le code java des deux interfaces IClientTransport et IAdminTransport
5. Créer une première implémentation java de ces deux interfaces



---

# Questions :

7. Créer une application qui permet de :
  - ❑ Créer une instance de SocieteTransport pour un administrateur.
  - ❑ Ajouter trois cargaisons routières et une cargaison aérienne à société de transport : « CR1 », « CA1 » et « CR2 »
  - ❑ Ajouter 3 marchandises à la cargaison dont la référence est CR1 (Numéros 1, 2, 3)
  - ❑ Ajouter 2 marchandises à la cargaison dont la référence est CA1 (Numéros 4,5)
  - ❑ Afficher toutes les informations concernant la cargaison CR1
  - ❑ Afficher toutes les informations concernant la marchandise 3.
  - ❑ Sérialiser les données dans le fichier « transport1.data »
8. Créer une autre application qui permet de :
  - ❑ Créer une instance de la classe SocieteTransport pour un client.
  - ❑ Charger les données à partir du fichier « transport1.data »
  - ❑ Afficher toutes les informations concernant la cargaison CA1
9. Créer une application SWING qui permet la saisie, l'ajout, la suppression, la consultation des données de l'application.

---

# Accès aux bases de données via JDBC

---

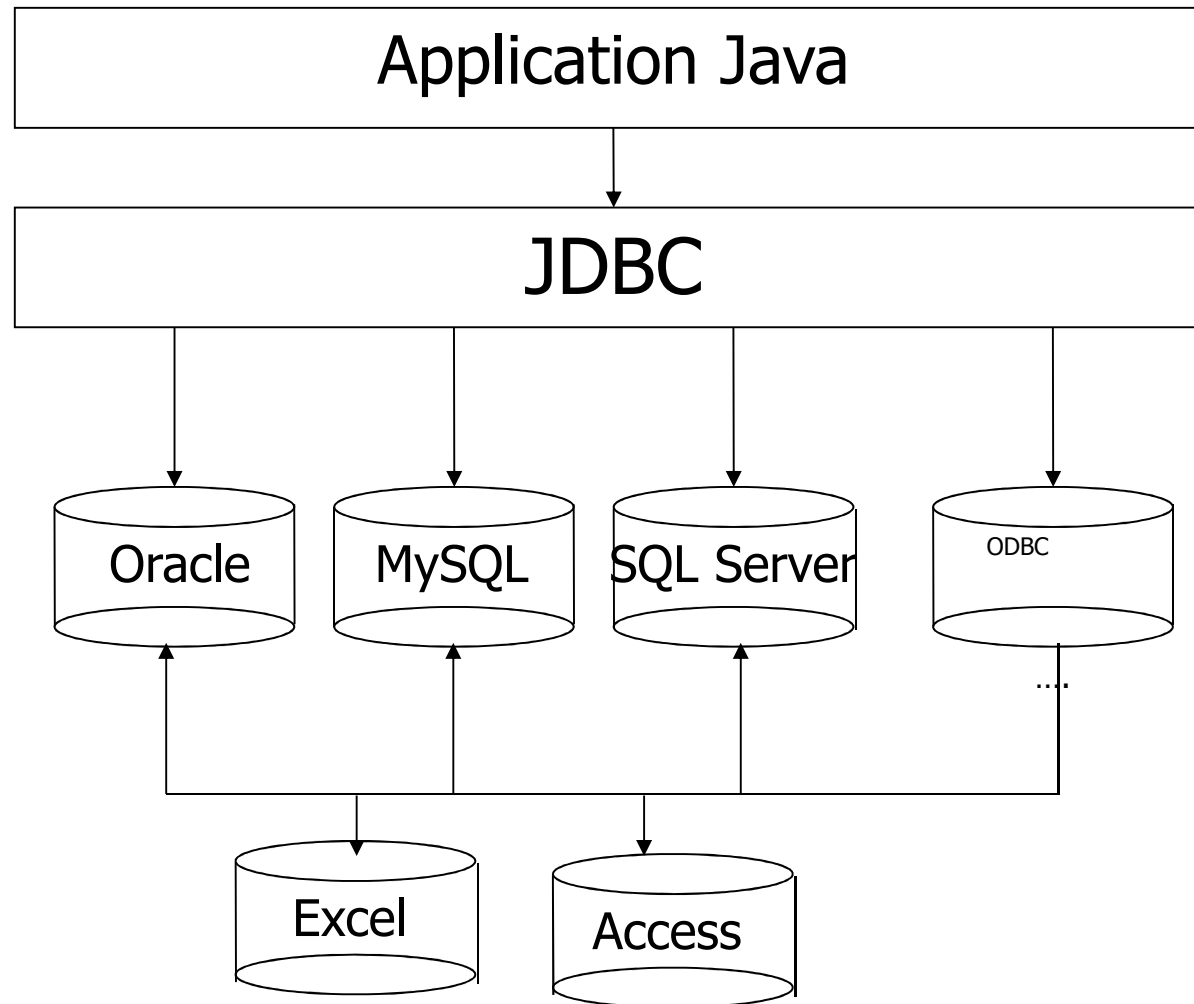


---

# Pilotes JDBC

- Pour qu'une application java puisse communiquer avec un serveur de bases de données, elle a besoin d'utiliser les pilotes JDBC (Java Data Base Connectivity)
- Les Pilotes JDBC est une bibliothèque de classes java qui permet, à une application java, de communiquer avec un SGBD via le réseau en utilisant le protocole TCP/IP
- Chaque SGBD possède ses propres pilotes JDBC.
- Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java communiquer avec n'importe quelle source de données via les pilotes ODBC (Open Data Base Connectivity)
- Les pilotes ODBC permettent à une application Windows de communiquer une base de données quelconque (Access, Excel, MySQL, Oracle, SQL SERVER etc...)
- La bibliothèque JDBC a été conçu comme interface pour l'exécution de requêtes SQL. Une application JDBC est isolée des caractéristiques particulières du système de base de données utilisé.

# Java et JDBC



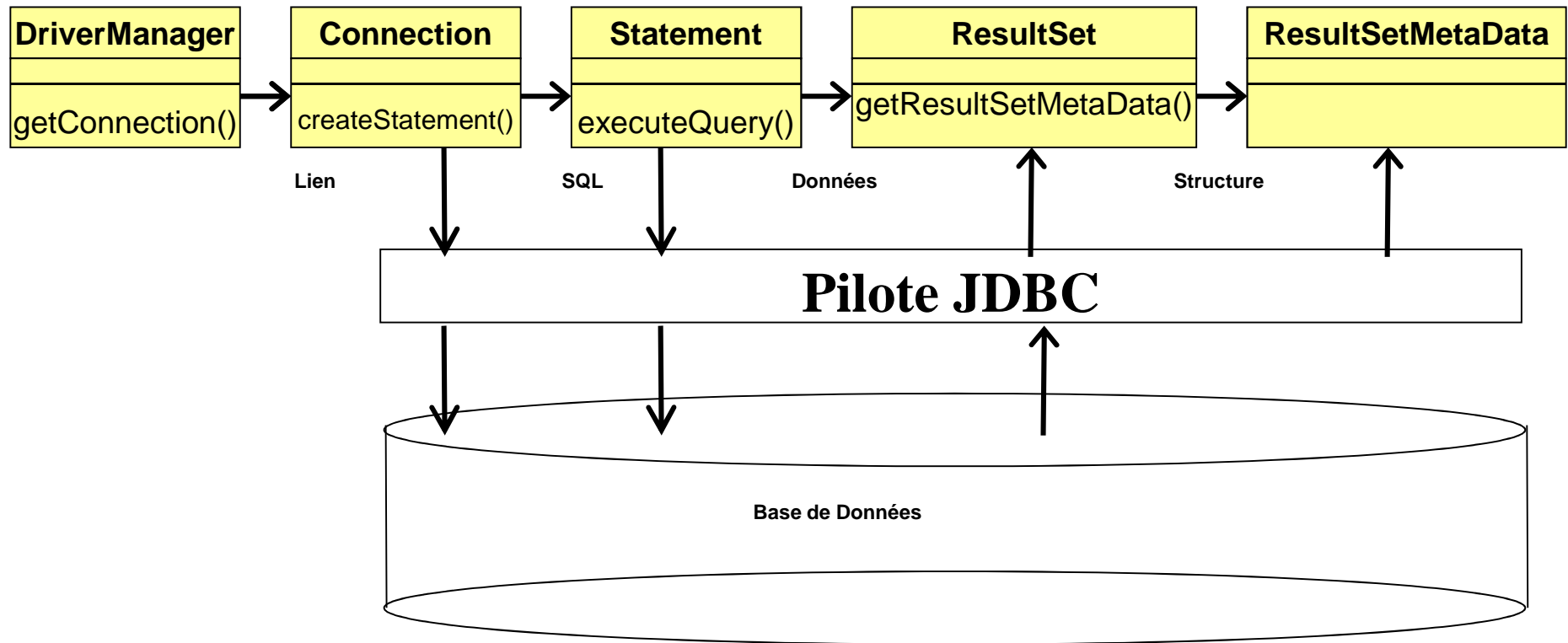
---

# Créer une application JDBC

Pour créer une application élémentaire de manipulation d'une base de données il faut suivre les étapes suivantes :

- Chargement du Pilote JDBC ;
- Identification de la source de données ;
- Allocation d'un objet **Connection**
- Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**);
- Exécution d'une requête à l'aide de l'objet Statement ;
- Récupération de données à partir de l'objet renvoyé **ResultSet** ;
- Fermeture de l'objet ResultSet ;
- Fermeture de l'objet Statement ;
- Fermeture de l'objet Connection.

# Créer une application JDBC



---

# Démarche JDBC

- Charger les pilotes JDBC :

- Utiliser la méthode `forName` de la classe `Class`, en précisant le nom de la classe pilote.

- Exemples:

- Pour charger le pilote `JdbcOdbcDriver`:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

- Pour charger le pilote jdbc de MySQL:

```
Class.forName("com.mysql.jdbc.Driver") ;
```

# Créer une connexion

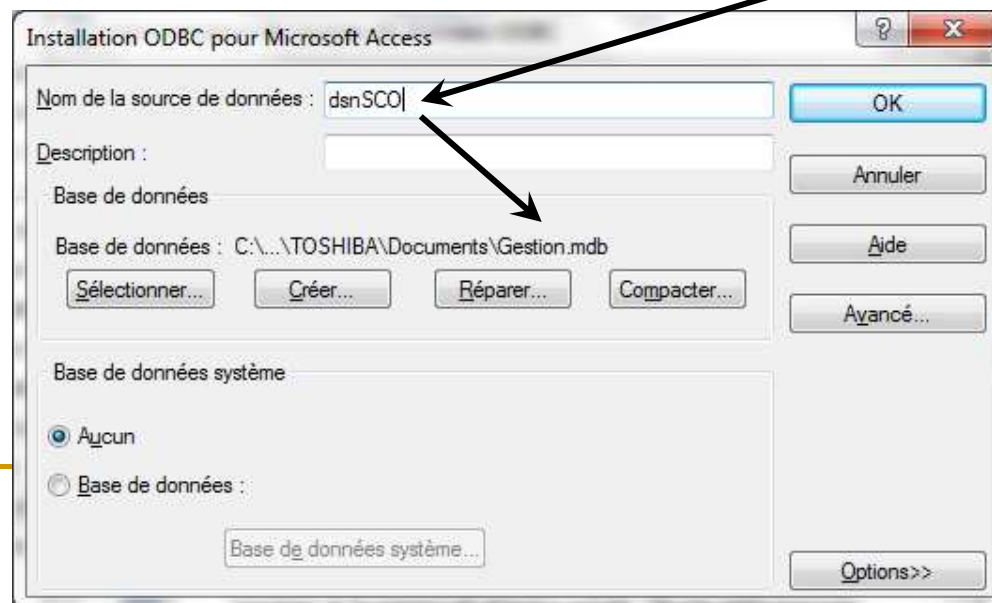
- Pour créer une connexion à une base de données, il faut utiliser la méthode statique `getConnection()` de la classe `DriverManager`. Cette méthode fait appel aux pilotes JDBC pour établir une connexion avec le SGBDR, en utilisant les sockets.

- Pour un pilote `com.mysql.jdbc.Driver` :

`Connection conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/DB", "user", "pass") ;`

- Pour un pilote `sun.jdbc.odbc.JdbcOdbcDriver` :

`Connection conn=DriverManager.getConnection("jdbc:odbc:dsnSCO", "user", "pass") ;`





## Objets Statement, ResultSet et ResultSetMetaData

- Pour exécuter une requête SQL, on peut créer l'objet Statement en utilisant la méthode createStatement() de l'objet Connection.
- Syntaxe de création de l'objet Statement

```
Statement st=conn.createStatement();
```

- Exécution d'une requête SQL avec l'objet Statement :
  - Pour exécuter une requête SQL de type select, on peut utiliser la méthode executeQuery() de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=st.executeQuery("select * from PRODUITS");
```

- Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode executeUpdate() de l'objet Statement :

```
st.executeUpdate("insert into PRODUITS (...) values(...)");
```

- Pour récupérer la structure d'une table, il faut créer l'objet ResultSetMetaData en utilisant la méthode getMetaData() de l'objet ResultSet.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

## Objet PreparedStatement

- Pour exécuter une requête SQL, on peut également créer l'objet PreparedStatement en utilisant la méthode `prepareStatement()` de l'objet Connection.
- Syntaxe de création de l'objet PreparedStatement

```
PreparedStatement ps=conn.prepareStatement("select *  
from PRODUITS where NOM_PROD like ? AND PRIX<?");
```

- Définir les valeurs des paramètres de la requête:

```
ps.setString(1, "%"+motCle+"%");
```

```
ps.setString(2, p);
```

- Exécution d'une requête SQL avec l'objet PreparedStatement :

- Pour exécuter une requête SQL de type select, on peut utiliser la méthode `executeQuery()` de l'objet Statement. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet ResultSet:

```
ResultSet rs=ps.executeQuery();
```

- Pour exécuter une requête SQL de type insert, update et delete on peut utiliser la méthode `executeUpdate()` de l'objet Statement :

```
ps.executeUpdate();
```

---

## Récupérer les données d'un ResultSet

- Pour parcourir un ResultSet, on utilise sa méthode `next()` qui permet de passer d'une ligne à l'autre. Si la ligne suivante existe, la méthode `next()` retourne `true`. Si non elle retourne `false`.
- Pour récupérer la valeur d'une colonne de la ligne courante du ResultSet, on peut utiliser les méthodes `getInt(colonne)`, `getString(colonne)`, `getFloat(colonne)`, `getDouble(colonne)`, `getDate(colonne)`, etc... colonne représente le numéro ou le nom de la colonne de la ligne courante.
- **Syntaxe:**

```
while(rs.next()){  
    System.out.println(rs.getInt(1));  
    System.out.println(rs.getString("NOM_PROD"));  
    System.out.println(rs.getDouble("PRIX"));  
}
```

## Exploitation de l'objet ResultSetMetaData

- L'objet ResultSetMetaData est très utilisé quand on ne connaît pas la structure d'un ResultSet. Avec L'objet ResultSetMetaData, on peut connaître le nombre de colonnes du ResultSet, le nom, le type et la taille de chaque colonne.
- Pour afficher, par exemple, le nom, le type et la taille de toutes les colonnes d'un ResultSet rs, on peut écrire le code suivant:

```
ResultSetMetaData rsmd=rs.getMetaData();  
// Parcourir toutes les colonnes  
for(int i=0;i<rsmd.getColumnCount();i++){  
    // afficher le nom de la colonne numéro i  
    System.out.println(rsmd.getColumnName(i));  
    // afficher le type de la colonne numéro i  
    System.out.println(rsmd.getColumnTypeName(i));  
    // afficher la taille de la colonne numéro i  
    System.out.println(rsmd.getColumnDisplaySize(i));  
}  
// Afficher tous les enregistrements du ResultSet rs  
while (rs.next()){  
    for(int i=0;i<rsmd.getColumnCount();i++){  
        System.out.println(rs.getString(i));  
    }  
}
```

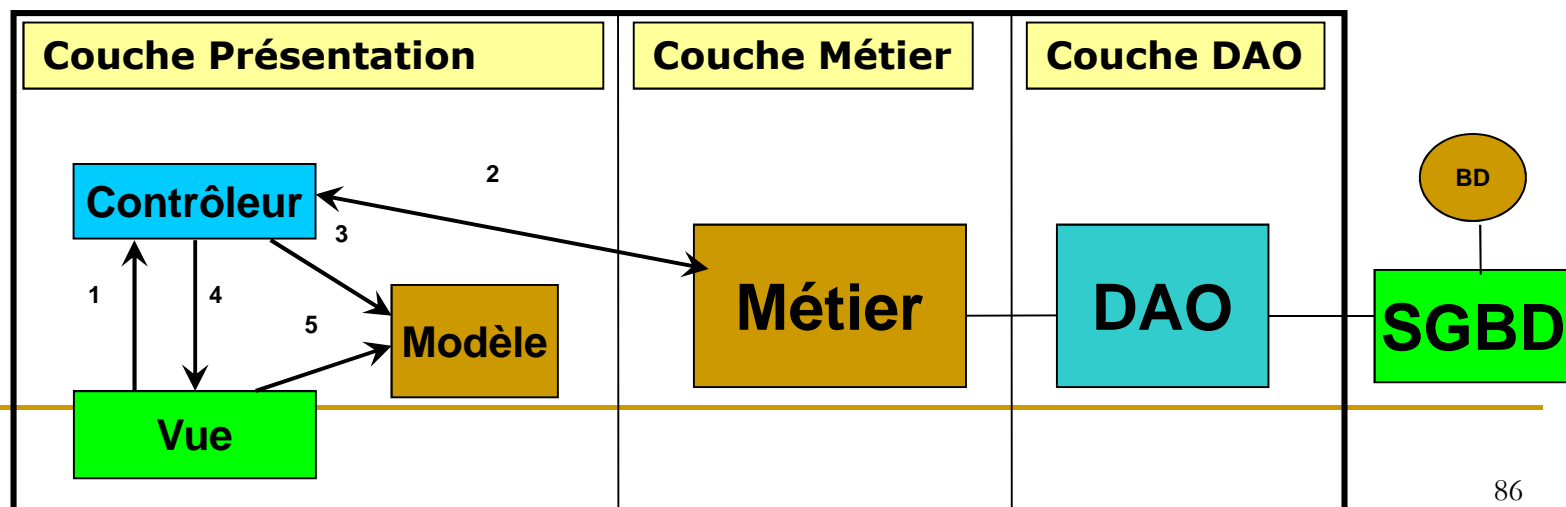
---

# Mapping objet relationnel

- Dans la pratique, on cherche toujours à séparer la logique de métier de la logique de présentation.
- On peut dire qu'on peut diviser une application en 3 couches:
  - La couche d'accès aux données: DAO
    - Partie de l'application qui permet d'accéder aux données de l'applications . Ces données sont souvent stockées dans des bases de données relationnelles .
  - La couche Métier:
    - Regroupe l'ensemble des traitements que l'application doit effectuer.
  - La couche présentation:
    - S'occupe de la saisie des données et de l'affichage des résultats;

# Architecture d'une application

- Une application se compose de plusieurs couches:
  - ❑ La couche DAO qui s'occupe de l'accès aux bases de données.
  - ❑ La couche métier qui s'occupe des traitements.
  - ❑ La couche présentation qui s'occupe de la saisie, le contrôle et l'affichage des résultats. Généralement la couche présentation respecte le pattern MVC qui fonctionne comme suit:
    1. La vue permet de saisir les données, envoie ces données au contrôleur
    2. Le contrôleur récupère les données saisies. Après la validation de ces données, il fait appel à la couche métier pour exécuter des traitements.
    3. Le contrôleur stocke le résultat de le modèle.
    4. Le contrôleur fait appel à la vue pour afficher les résultats.
    5. La vue récupère les résultats à partir du modèle et les affiche.

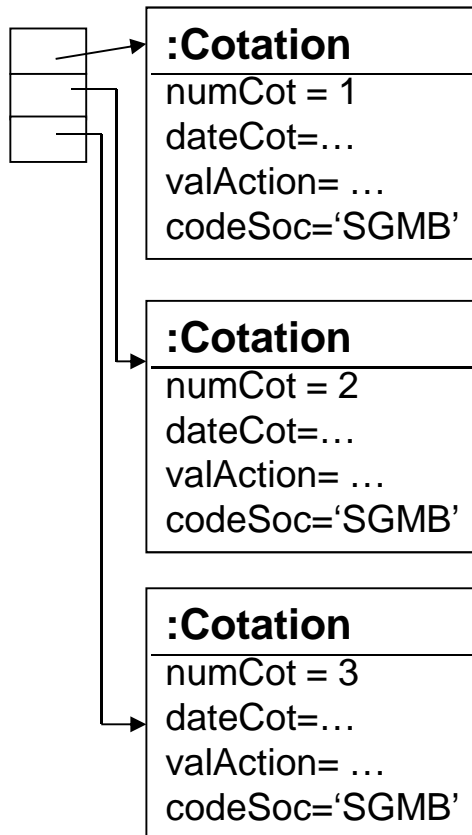


# Mapping objet relationnel

- D'une manière générale les applications sont orientée objet :
  - ❑ Manipulation des objet et des classes
  - ❑ Utilisation de l'héritage et de l'encapsulation
  - ❑ Utilisation du polymorphisme
- D'autres part les données persistantes sont souvent stockées dans des bases de données relationnelles.
- Le mapping Objet relationnel consiste à faire correspondre un enregistrement d'une table de la base de données à un objet d'une classe correspondante.
- Dans ce cas on parle d'une classe persistante.
- Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (Base de données, Fichier, etc..)

# Couche Métier : Mapping objet relationnel

cots



```
public List<Cotation> getCotations(String codeSoc){
    List<Cotation> cotations=new ArrayList<Cotation>();
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn=DriverManager.getConnection
            ("jdbc:mysql://localhost:3306/bourse_ws","root","");
        PreparedStatement ps=conn.prepareStatement
            ("select * from cotations where CODE_SOCIETE=?");
        ps.setString(1, codeSoc);
        ResultSet rs=ps.executeQuery();
        while(rs.next()){
            Cotation cot=new Cotation();
            cot.setNumCotation(rs.getLong("NUM_COTATION"));
            cot.setDateCotation(rs.getDate("DATE_COTATION"));
            cot.setCodeSociete(rs.getString("CODE_SOCIETE"));
            cot.setValAction(rs.getDouble("VAL_ACTION"));
            cotations.add(cot);
        }
    } catch (Exception e) { e.printStackTrace();}
    return(cotations);
}
```

NUM_COTATION	DATE_COTATION	VAL_ACTION	CODE_SOCIETE
1	2008-08-30 15:57:50	2093.17199826538	SGMB
2	2008-08-30 15:57:52	258.769396752267	SGMB
3	2008-08-30 15:57:52	1050.71222698514	SGMB



# Application

- On considère une base de données qui contient une table ETUDIANTS qui permet de stocker les étudiants d'une école. La structure de cette table est la suivante :

Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
ID_ET	int(11)			Non	Aucun	auto_increment
NOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
PRENOM	varchar(25)	latin1_swedish_ci		Non	Aucun	
EMAIL	varchar(25)	latin1_swedish_ci		Non	Aucun	
VILLE	varchar(200)	latin1_swedish_ci		Non	Aucun	

ID_ET	NOM	PRENOM	EMAIL	VILLE
1	A	PA	A@YAHOO.FR	casa
2	B	PB	B@YAHOO.FR	rabat
3	C	PC	C@YAHOO.FR	casa
4	BBCAAC	BBCAAC	ab@yahoo.fr	casa

- Nous souhaitons créer une application java qui permet de saisir au clavier un motclé et d'afficher tous les étudiants dont le nom contient ce mot clé.
- Dans cette application devons séparer la couche métier de la couche présentation.

# Application

- Pour cela, la couche métier est représentée par un modèle qui se compose de deux classes :
  - La classe Etudiant.java : c'est une classe persistante c'est-à-dire que chaque objet de cette classe correspond à un enregistrement de la table ETUDIANTS. Elle se compose des :
    - champs privés idEtudiant, nom, prenom, email et ville,
    - d'un constructeur par défaut,
    - des getters et setters.

Ce genre de classe c'est ce qu'on appelle un java bean.

- La classe Scholarite.java :
  - c'est une classe non persistante dont laquelle, on implémente les différentes méthodes métiers.
  - Dans cette classe, on fait le mapping objet relationnel qui consiste à convertir un enregistrement d'une table en objet correspondant.
  - Dans notre cas, une seule méthode nommée getEtudiants(String mc) qui permet de retourner une Liste qui contient tous les objets Etudiant dont le nom contient le mot clé «mc»

---

# Application

## Travail à faire :

### Couche données :

- ❑ Créer la base de données « SCOLARITE » de type MSAccess ou MySQL
- ❑ Pour la base de données Access, créer une source de données système nommée « dsnScolarite », associée à cette base de données.
- ❑ Saisir quelques enregistrements de test

### Couche métier. ( package metier) :

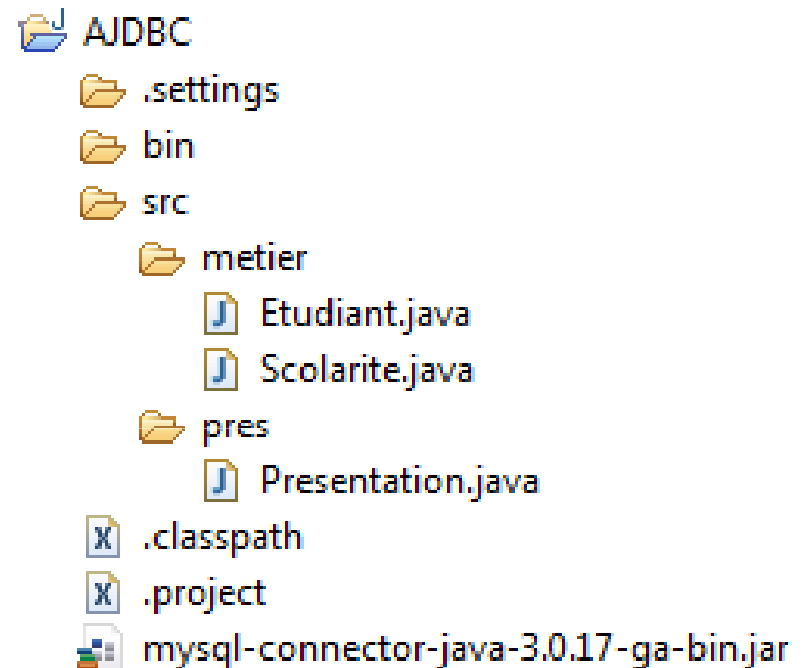
- ❑ Créer la classe persistante Etudiant.java
- ❑ Créer la classe des business méthodes Scolarite.java

### Couche présentation (package pres):

- ❑ Créer une application de test qui permet de saisir au clavier le mot clé et qui affiche les étudiants dont le nom contient ce mot clé.

## Couche métier : la classe persistante Etudiant.java

```
package metier;  
  
public class Etudiant {  
    private Long idEtudiant;  
    private String nom,prenom,email;  
    // Getters etStters  
}
```



# Couche métier : la classe Scholarite.java

```
package metier;
import java.sql.*; import java.util.*;
public class Scholarite {
    public List<Etudiant> getEtudiantParMC(String mc){
        List<Etudiant> etds=new ArrayList<Etudiant>();
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection conn=
                DriverManager.getConnection("jdbc:mysql://localhost:3306/DB_SCO","root","");
            PreparedStatement ps=conn.prepareStatement("select * from ETUDIANTS where NOM
                like ?");
            ps.setString(1,"%"+mc+"%");
            ResultSet rs=ps.executeQuery();
            while(rs.next()){
                Etudiant et=new Etudiant();
                et.setIdEtudiant(rs.getLong("ID_ET"));et.setNom(rs.getString("NOM"));
                et.setPrenom(rs.getString("PRENOM"));et.setEmail(rs.getString("EMAIL"));
                etds.add(et);
            }
        } catch (Exception e) {    e.printStackTrace(); }
        return etds;
    }
}
```

# Couche Présentation : Applications Simple

```
package pres;
import java.util.List;import java.util.Scanner;
import metier.Etudiant;import metier.Scolarite;
public class Presentation {
    public static void main(String[] args) {

        Scanner clavier=new Scanner(System.in);

        System.out.print("Mot Clé:");

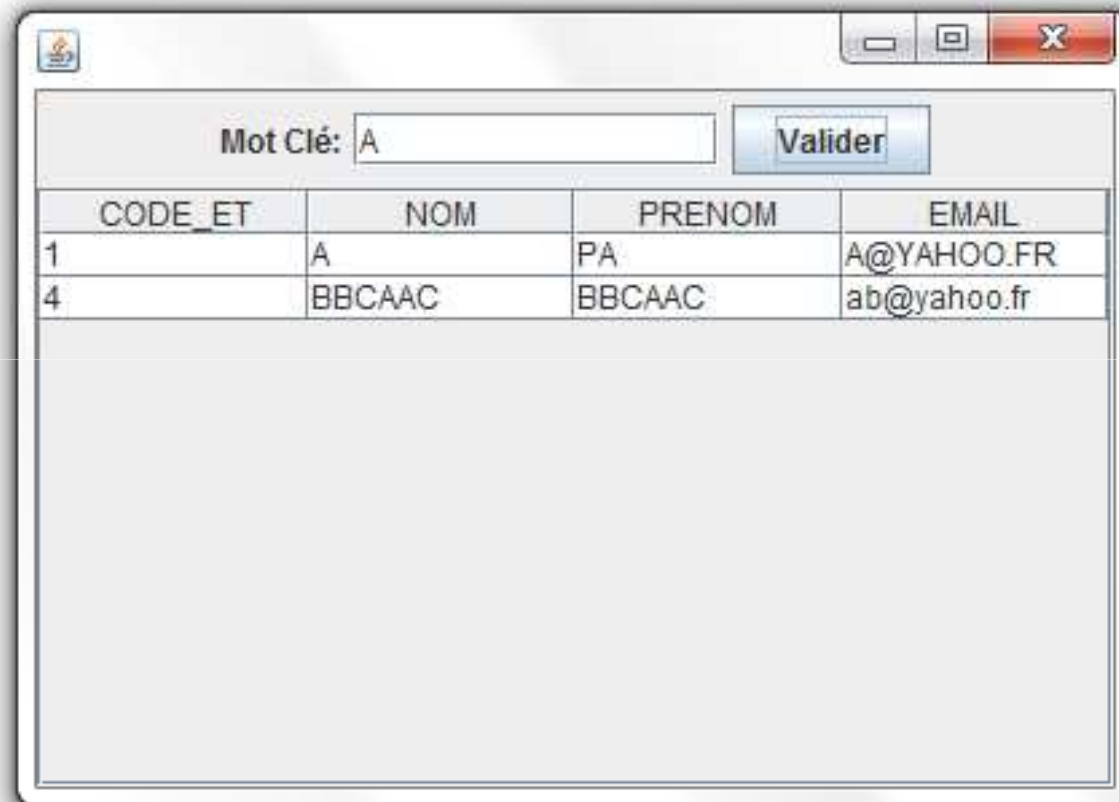
        String mc=clavier.next();

        Scolarite metier=new Scolarite();

        List<Etudiant> etds=metier.getEtudiantParMC(mc);

        for(Etudiant et:etds)
            System.out.println(et.getNom()+"\t"+et.getEmail());
    }
}
```

# Couche Présentation : Application SWING




The image shows a Java Swing application window. At the top, there is a label 'Mot Clé:' followed by a text input field containing the letter 'A'. To the right of the input field is a button labeled 'Valider'. Below this is a table with four columns: 'CODE\_ET', 'NOM', 'PRENOM', and 'EMAIL'. The table contains two rows of data. The first row has values '1', 'A', 'PA', and 'A@YAHOO.FR'. The second row has values '4', 'BBCAAC', 'BBCAAC', and 'ab@yahoo.fr'. Below the table is a large, empty rectangular area.

	CODE_ET	NOM	PRENOM	EMAIL
1		A	PA	A@YAHOO.FR
4		BBCAAC	BBCAAC	ab@yahoo.fr

# Le modèle de données pour JTable

```
package pres;
import java.util.List;import java.util.Vector;
import javax.swing.table.AbstractTableModel;
import metier.Etudiant;
public class EtudiantModele extends AbstractTableModel{
    private String[] tabelColumnNames=new
    String[]{"CODE_ET","NOM","PRENOM","EMAIL"};
    private Vector<String[]> tableValues=new Vector<String[]>();
@Override
public int getRowCount() {
    return tableValues.size();
}
@Override
public int getColumnCount() {
    return tabelColumnNames.length;
}
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    return tableValues.get(rowIndex)[columnIndex];
}
```



X

Mot Clé:

Valider

CODE_ET	NOM	PRENOM	EMAIL
1	A	PA	A@YAHOO.FR
4	BBCAAC	BBCAAC	ab@yahoo.fr

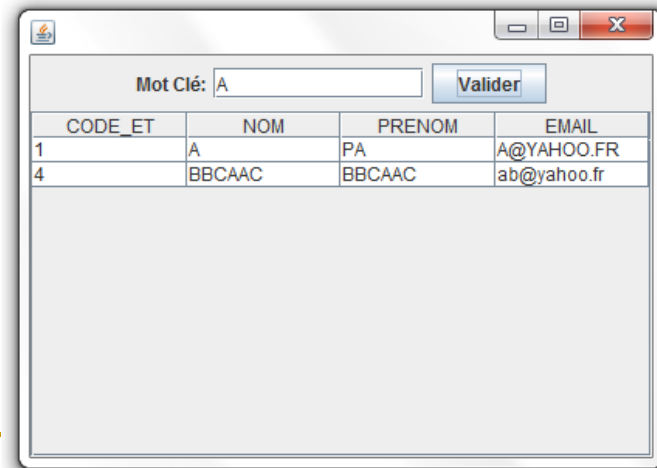


# Le modèle de données pour JTable (Suite)

```
@Override
public String getColumnName(int column) {
    return tabelColumnNames[column];
}

public void setData(List<Etudiant> etudiants){
    tableValues=new Vector<String[]>();
    for(Etudiant et:etudiants){
        tableValues.add(new String[]{
            String.valueOf(et.getIdEtudiant()),et.getNom(),et.getPrenom
            (),et.getEmail()});
    }

    fireTableChanged(null);
}
}
```



The screenshot shows a Java Swing window with a title bar. Inside, there is a text field labeled 'Mot Clé:' containing the letter 'A', followed by a 'Valider' button. Below this is a JTable with four columns: 'CODE\_ET', 'NOM', 'PRENOM', and 'EMAIL'. The table contains two rows of data.

CODE_ET	NOM	PRENOM	EMAIL
1	A	PA	A@YAHOO.FR
4	BBCAAC	BBCAAC	ab@yahoo.fr

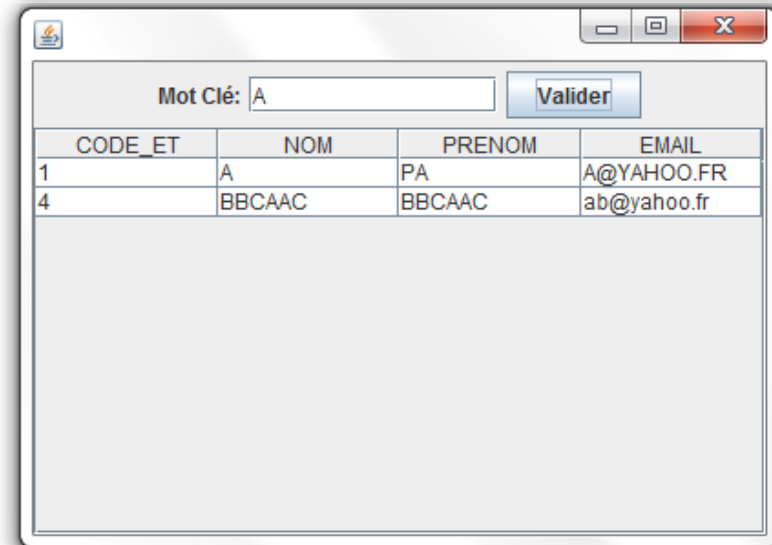
# L'appliaion SWING

```
package pres;
import java.awt.*;import java.awt.event.*;import javax.swing.*;
import metier.*;
public class EtudiantFrame extends JFrame {
    private JScrollPane jsp;

    private JLabel jLMC=new JLabel("Mot Clé:");
    private JTextField jTFMC=new JTextField(12);
    private JButton jbValider=new JButton("Valider");

    private JTable jTableEtudiants;
    private JPanel jpN=new JPanel();

    private Sclarite metier
=new Sclarite();
    private EtudiantModele model;
```



# L'application SWING (Suite)

```
public EtudiantFrame() {
    this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    this.setLayout(new BorderLayout());
    jTFMC.setTipText("Mot Clé:");
    jpN.setLayout(new FlowLayout());
    jpN.add(jLMC); jpN.add(jTFMC);
    jpN.add(jBValider);
    this.add(jpN, BorderLayout.NORTH);
    model=new EtudiantModele();
    jTableEtudiants=new.JTable(model);
    jsp=new JScrollPane(jTableEtudiants);
    this.add(jsp, BorderLayout.CENTER);
    this.setBounds(10,10,500,500);
    this.setVisible(true);
    jBValider.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String mc=jTFMC.getText();
        model.setData(metier.getEtudiantParMC(mc));
    }
    });
}

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

