



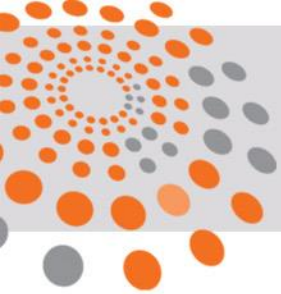
Découvrir l'Objet et programmer en Java





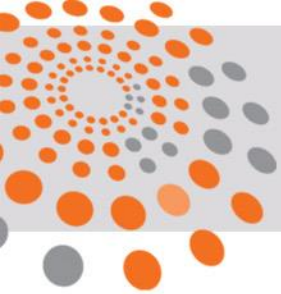
Introduction à la technologie Java





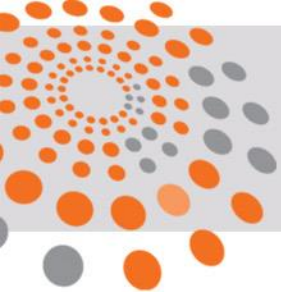
Présentation générale de java

- Java est un langage de programmation par objets développé par SUN Microsystems (James Gosling).
- La version 1.0 du langage Java apparaît en 1995 comme une solution aux limitations du html pour :
 - Créer des pages animées
 - Effectuer des contrôles de surfaces
- Initié par SUN dont la devise est « L'ordinateur, c'est le réseau », Java en garde une inclinaison pour l'ouverture et les standards.



Présentation générale de java

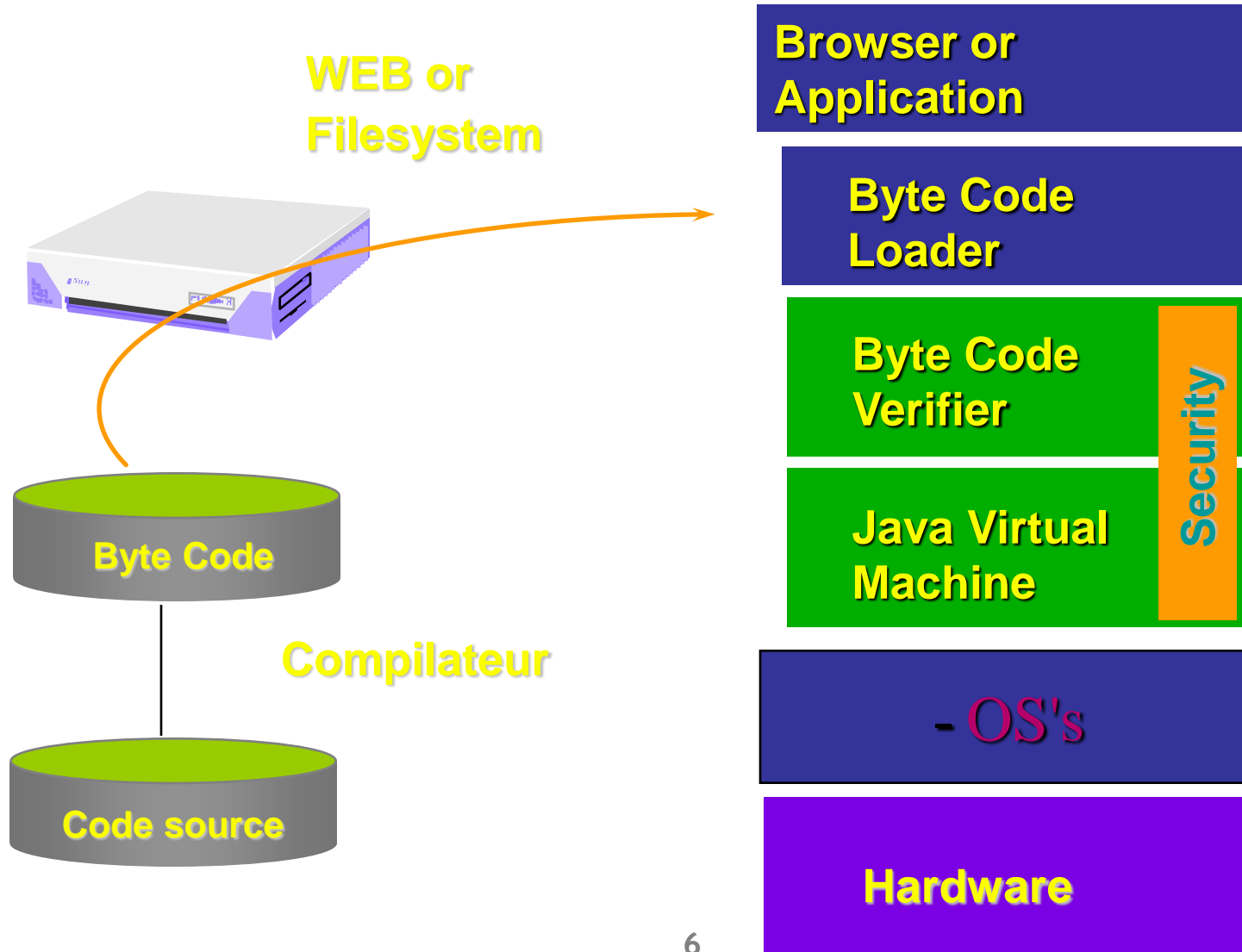
- Ses évolutions sont aujourd'hui pilotées par des processus communautaires ouverts.
- C'est un véritable langage objets qui supporte l'ensemble des mécanismes (classe, héritage, polymorphisme), portable et indépendant de toute plate-forme.
- Le langage est un bon compromis entre le langage C++ (plus simple et syntaxe proche du C++) et smalltalk (gestion dynamique).



Buts et positionnements

- A l'origine, concurrent des ActiveX et de Flash, Java a rapidement perdu la bataille du « client riche » internet (RIA) au profit de Flash.
- Ce sont ses qualités intrinsèques qui l'ont sauvé :
 - Un langage objet, fortement typé, simple d'accès,
 - Multi plateformes,
 - Doté d'une bibliothèque riche et également multi plateformes.
- C'est finalement côté serveur que Java a trouvé sa place avec des architectures élaborées pour la réalisation d'applications d'entreprise importantes.

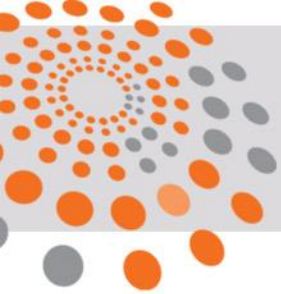
Le langage Java est portable





Les différentes plateformes Java

- Au delà du langage, le terme Java désigne l'ensemble des API (Application Programming Interface) qui constituent une plateforme de développement.
- Il existe plusieurs plateformes ainsi définies, spécialisées dans le développement d'un type donné d'applications :
 - JSE (Java SE): applications clients lourds, monopostes
 - JEE (Java EE): applications serveurs ou réparties, clients légers ou riches
 - JME (Java ME): applications pour l'informatique embarquée



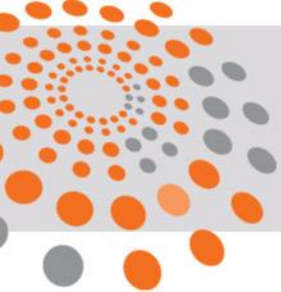
JSE (Java Platform Standard Edition)

- Il s'agit de la version « historique » de Java
- Elle comprend une riche bibliothèque d'utilitaires et d'algorithmes transversaux :
 - Structures de données telles que piles, files, tableaux dynamiques, ...
 - Tris, manipulation de chaînes de caractères, ...
- Elle intègre également deux bibliothèques graphiques :
 - AWT : historique, liée à un rendu natif
 - Swing : rendu spécifique mais consistant entre plateformes, modèle MVC
 - JavaFX: depuis Java 8, l'API java pour développer les interfaces graphiques



JEE (Java Enterprise Edition)

- Il s'agit d'une version JSE enrichie des API de toutes les briques techniques serveurs dite JEE (JEE).
- Ces API définissent en particulier les contrats de collaboration entre les composants métiers que vous développez et les environnements d'exécution dans lesquels ils seront déployés.
- Elles apportent en sus des API pour la manipulation du XML ou l'utilisation de protocoles de communication tels
 - CORBA
 - SOAP
 - ...



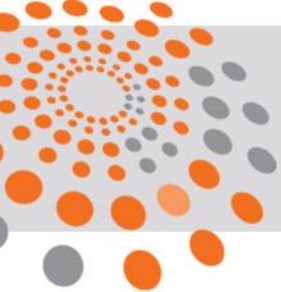
JME (Java Micro Edition)

- Une déclinaison de la plateforme Java destinée aux machines à ressources limitées pour :
 - L'informatique embarquée :
 - PDA (palm, ...)
 - Téléphones portables
 - L'électronique grand public :
 - Télévisions,
 - Fours, aspirateurs, ...
- Pour tous les matériels électroniques dotés de 128Ko de RAM ou plus et de processeurs aux fonctionnalités et aux performances limitées.



JDK et environnement d'exécution





Le JDK

- Le jdk (java Development kit) distribué gratuitement par Oracle fournit l'ensemble des outils de base pour le développement d'applications Java :
 - Un compilateur : **javac**
 - Une JVM : **java**
 - Un testeur d'applet : **appletviewer**
 - Un débogueur : **jdb**
 - Un créateur de paquet pour distribution : **jar**
 - Un système de génération de documentation : **javadoc**

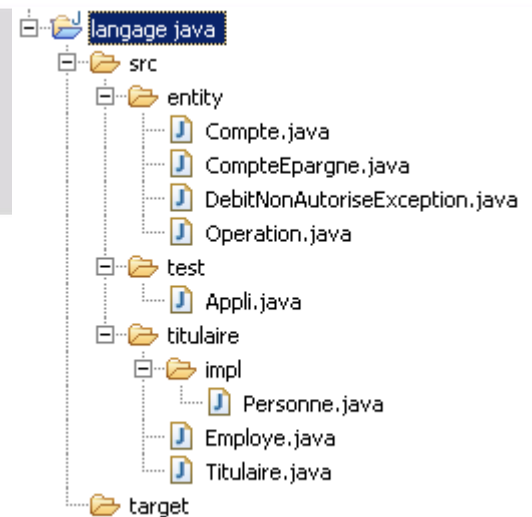


Les outils du JDK : le compilateur

- Le compilateur javac :
 - transforme le code source en byte code exécutable par la JVM sur n'importe quelle plate-forme ...
 - Pour une version donnée du JDK
- Syntaxe : `javac <options> <source files>`
 - **-g** : génère les informations de debug
 - **-verbose** : trace les activités du compilateur
 - **-classpath** <path> : répertoires des fichiers .class
 - **-sourcepath** <path> : répertoires des fichiers Java
 - **-d** <directory> : répertoires de génération des .class
 - **-source** <release> : effectue une vérification de compatibilité des sources avec une version spécifique du JDK **-target** <release> génère les .class pour une version spécifique de JVM

Le compilateur javac : mise en oeuvre

Organisation physique avant compilation



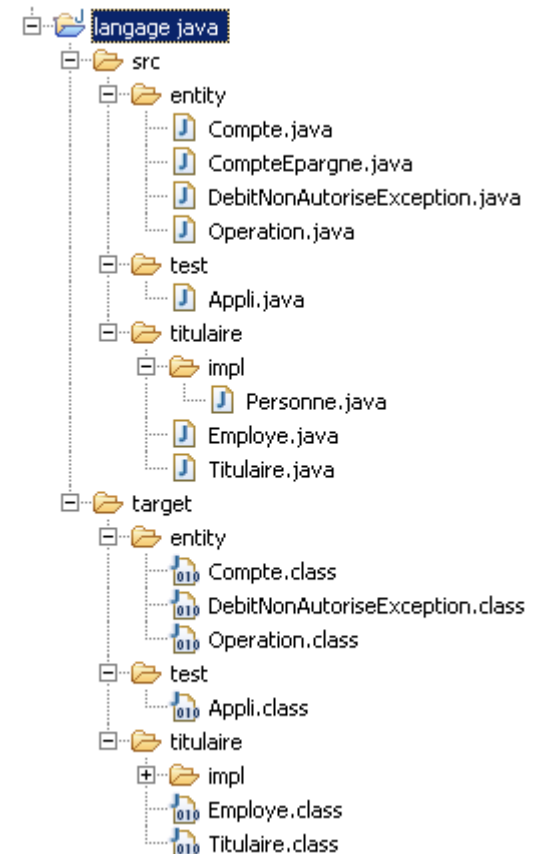
- Compilation

chdir C:\travail\langage java

javac -sourcepath src \ -d target

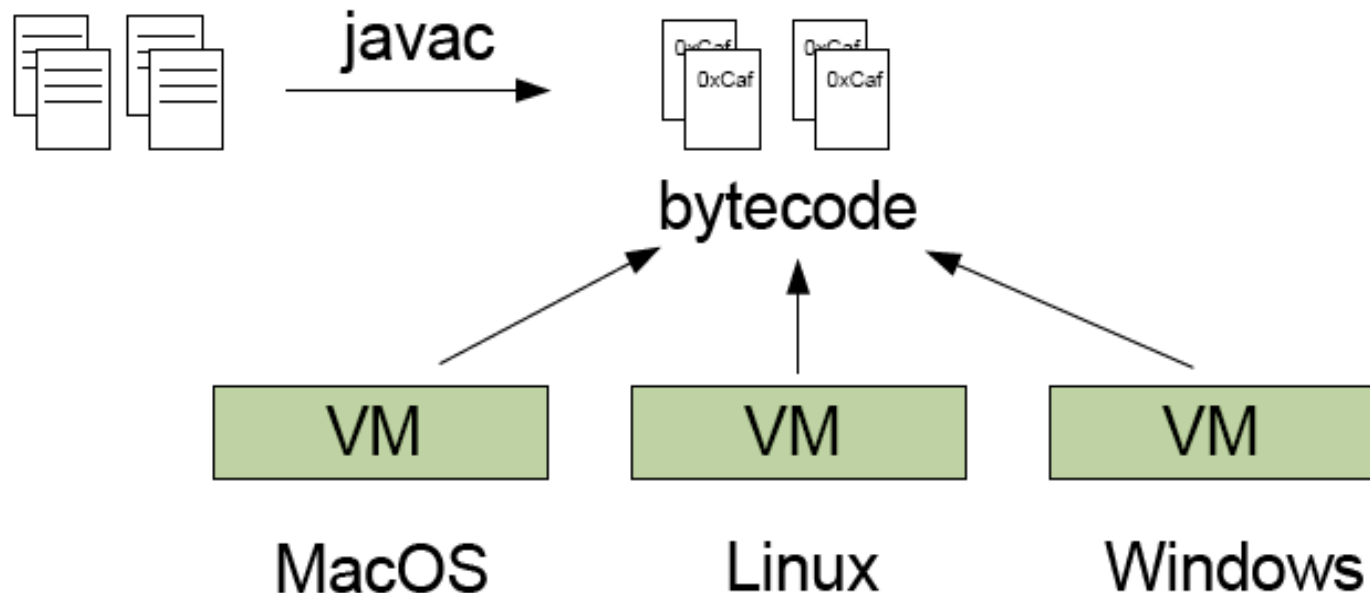
La compilation est **transitive** !

Organisation physique après compilation



Les outils du JDK : la JVM

- Le compilateur Java génère un « byte code », langage assembleur de la JVM Java. Ce code est ensuite interprété.
- La JVM assure la portabilité entre différents environnements d'exécution (OS)





La JVM : mise en oeuvre

- **Syntaxe :**

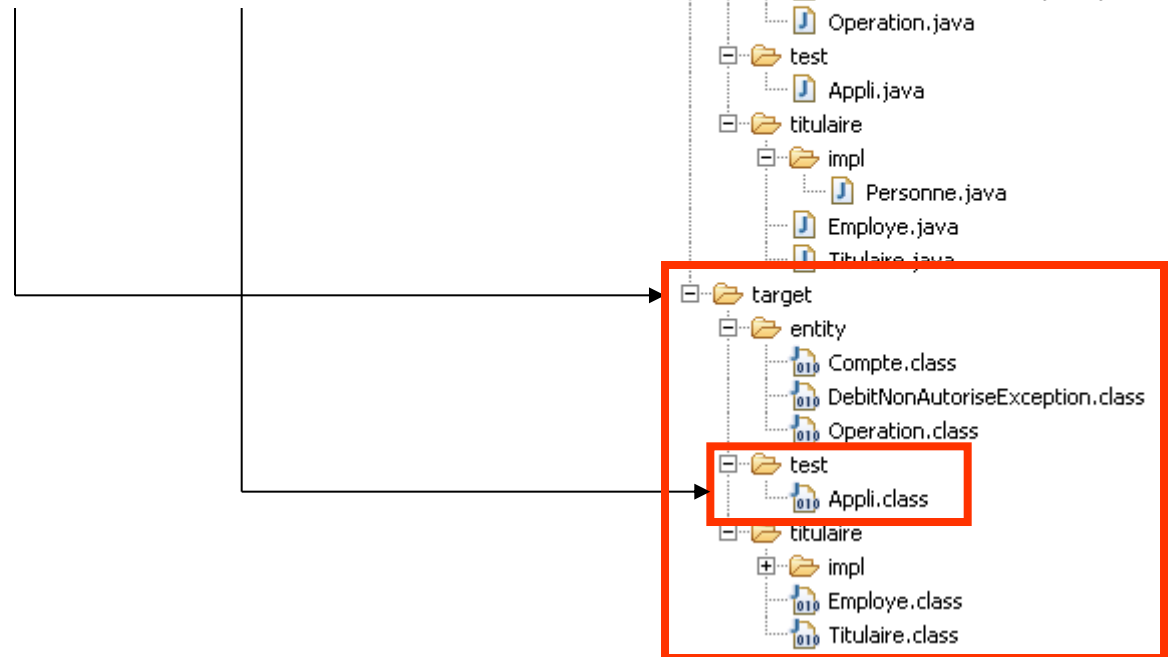
- Pour exécuter un .class : **java [-options] class[args...]**
- Pour exécuter un JAR : **java [-options] -jar jarfile [args...]**

- **Principales options**

- **-cp <class search path of directories and zip/jar files>** : fournit la liste de répertoires et d'archives (JAR et ZIP) séparés par “;” et dans lesquels rechercher les .class
- **-D<name>=<value>** : valorise une variable d'environnement
- **-version:<value>** : requière la version spécifiée
- **-showversion** : affiche la version de la JVM
- **-ea[:<packagename>... | :<classname>]** : active les assertions
- **-da[:<packagename>... | :<classname>]** : désactive les assertions

La JVM : mise en oeuvre

- Exécution de la classe Appli
 - `java -classpath target test.Appli`



- Résultat d'exécution
 - débit impossible : solde=10.0 montant demandé=100.0



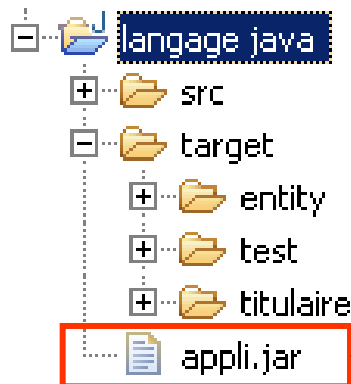
Les outils du JDK : la commande jar

- L'archiveur de classes :
 - Outil standard pour construire/lire des archives de type JAR
 - Équivalent à la notion de librairie en C/C++
- Syntaxe similaire à la comande « tar » unix
 - `jar {ctxui}[vfm0Me][fichier-jar][fichier-manifeste] [point-entrée] [-C rép] fichiers ...`
- Principales options
 - `-c` : crée une nouvelle archive
 - `-x` : extrait les fichiers nommés (ou tous les fichiers) de l'archive
 - `-u` : met à jour l'archive existante
 - `-f` : spécifie le nom du fichier archive
 - `-m` : inclut les informations de manifeste à partir du fichier de manifeste spécifié
 - `-C` : passe au répertoire spécifié et inclut le fichier suivant

La commande jar : mise en œuvre de la commande jar

Mise en œuvre

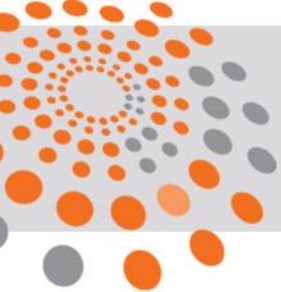
- `jar cvf appli.jar -C target\ .`



« . » indique le répertoire courant

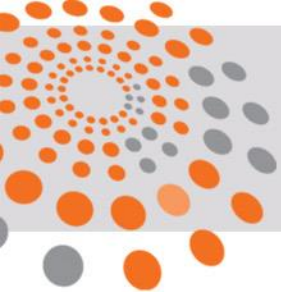
Exécution du JAR

- `java -classpath appli.jar test.Appli`
- Résultat : débit impossible : solde=10.0 montant demandé=100.0



Créer un jar exécutable

- Nécessite de créer un fichier MANIFEST définissant la classe du jar contenant le point d'entrée (le main)
- Exemple de fichier MANIFEST
Manifest-Version: 1.0
Main-Class: test.Appli
- Production du JAR
 - `jar cvfm appli.jar appli-manifest -C target\ .`
- Exécution du JAR
 - `java -jar appli.jar`



Autre outils du JDK

- L'interpréteur d'applets :
 - appletviewer permet de visualiser l'exécution d'un applet
 - Syntaxe : **jappletviewer monApplet.class**
- Le générateur de documentation :
 - JavaDoc a pour but de créer une documentation uniforme au format HTML à partir du code,
 - grâce aux commentaires prévus à cet effet (/**).
 - la documentation de l'API standard Java est de la JavaDoc ...



Premiers éléments syntaxiques

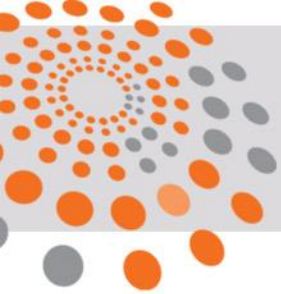




Architecture d'un programme

- Tout programme est constitué d'une classe "principale" qui doit être homographe du fichier source (*.java).
- Le point d'entrée du programme est la fonction main() de cette classe.

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



Type de base

- Entier

entier de base	int	4 octets
entier court	short	2 octets
entier long	long	8 octets
octet	byte	1 octet

- Flottant

simple précision	float	4 octets
double précision	double	8 octets

- Caractère

char 2 octets

- Booléen

boolean 1 octet



Les variables

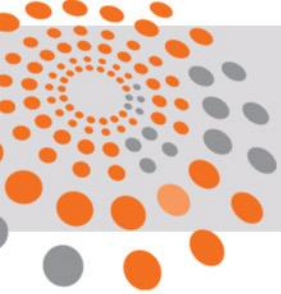
- Les variables doivent être déclarées avant d'être utilisées. Pour les déclarer, on associe un identificateur à un type de donnée particulier.

`char unCar ;`

- On peut initialiser une variable en lui affectant une valeur lors de sa déclaration.

`int unEntier = 3 ;`

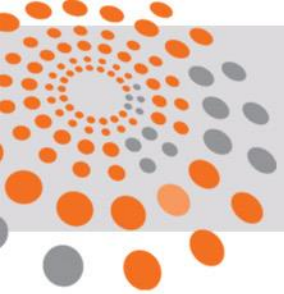
- Le qualificatif **final** permet de spécifier que le contenu d'une variable ne pourra pas évoluer (permet donc de définir des constantes).



Les littéraux

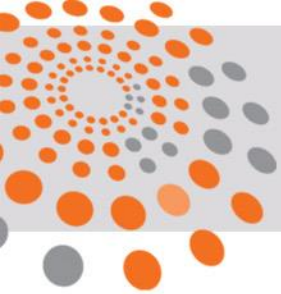
- Entières : `22, 010, 0xFF`
- Réelles: `3.1415f, 12e-3`
- Caractères : `'A', '3', '*'`
- Les chaînes de caractères : `"uneChaine", "", "A"`
- On a la possibilité de représenter les caractères spéciaux :

<i>retour arrière</i>	<code>'\b'</code>
<i>tabulation horizontale</i>	<code>'\t'</code>
<i>retour à ligne</i>	<code>'\n'</code>
<i>retour chariot</i>	<code>'\r'</code>



Les opérateurs

- Opérateurs arithmétiques :
+, -, *, /, %
- Opérateurs relationnels :
>, <, >=, <=, ==, !=
- Affectation :
=
- Opérateurs logiques :
&&, ||, !



Les opérateurs

- Opérateur d'incrémentation et de décrémentation :
 incrémentation ++
 décrémentation --
- On peut mettre ces opérateurs sous forme préfixée ou postfixée :
 int res = 0 , i = 3 ;
 res = i++ ; // i = 4 res = 3
 res = ++i ; // i = 5 res = 5
- Opérateur Conditionnel : *expr1 ? expr2 : expr3*
 int nbr1 = 3 , nbr2 = 5 , max ;
 max = (nbr1 > nbr2) ? nbr1 : nbr2 ;



Les entrées/sorties formatées

- La gestion des entrées-sorties se fait par un ensemble de fonctions standards définies dans la librairie
- Java propose une classe System qui contient des flux standards prédéfinis (in, out) nous permettant de manipuler les entrées sorties standards.
- Exemple de sortie formatée sur la console

```
class Demo {  
    public static void main ( String args [ ] ) {  
        for(int i = 0; i<args.length; i++)  
            System.out.println("args["+i+"]=" +  
args[i]);  
    }  
}
```



Les entrées/sorties formatées

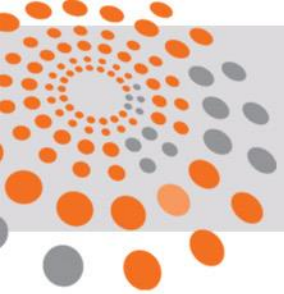
- La classe Scanner (JDK1.5) permet la lecture de d'information formatée.

```
import java.util.Scanner;

public class Demo {
    public static void main(String[] args) {
        String chaine;
        Scanner clavier = new Scanner(System.in);
        System.out.println("Entrez un chaine de caractères au clavier");
        chaine = clavier.next();
        System.out.println ( "Longueur de la chaine : " + chaine
                             + "="  + chaine.length() ) ;

        clavier.close();
    }
}
```

- La classe Scanner propose des méthodes de lecture formatée pour tous les types primitifs : nextInt, nextLong, nextFloat, nextDouble, nextBoolean, ...



L'instruction if-else

- Elle permet d'exécuter une instruction ou une suite d'instructions en fonction de l'évaluation d'une condition.

```
if ( expression )  
[ {  
    Instruction1;  
}]  
else  
[ {  
    Instruction2;  
}]
```

- On doit définir un bloc à partir du moment où il y a plus d'une instruction.
- L'instruction else n'est pas obligatoire.



Exemple

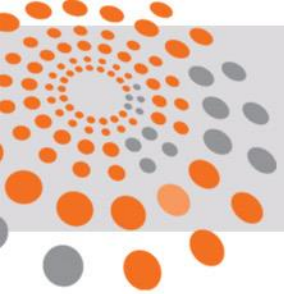
```
import java.util.Scanner;
class Demo {
    public static void main ( String args[] ) {
        int valeur1 = 0 , valeur2 = 0 , resultat = 0 ;
        Scanner  clavier = new Scanner( System.in) ;
        System.out.println("Entrer 2 entiers") ;
        valeur1 = clavier.nextInt() ;
        valeur2 = clavier.nextInt() ;
        if ( valeur1 > valeur2 )
            resultat = valeur1;
        else
            resultat = valeur2 ;
        System.out.println ( "Valeur Max : " + resultat ) ;
        clavier.close();
    }
}
```




L'instruction if-else-if

- Elle permet de choisir une séquence d'instructions parmi plusieurs en fonction de l'évaluation d'une expression.

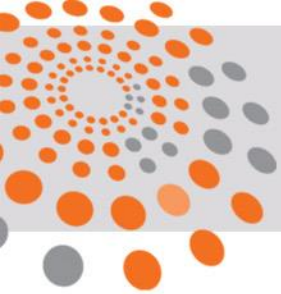
```
if (expression1)
    instruction1;
else if (expression2)
    instruction2;
else if (expression3)
    instruction3;
else
    instruction4;
```



Exemple

```
import java.util.Scanner;

class Demo {
    public static void main ( String argv[] )
    {
        Scanner  clavier = new Scanner( System.in) ;
        int jourSemaine = 0 ;
        System.out.println ( "Entrer un jour dans la semaine ( 1-7 )" ) ;
        jourSemaine  = clavier.nextInt ();
        if ( jourSemaine == 1 )  System.out.println ( "Lundi" ) ;
        else  if ( jourSemaine == 2 ) System.out.println ( "Mardi" ) ;
        ...
        else if ( jourSemaine == 7 ) System.out.println ( "Dimanche" ) ;
        else System.out.println ( "Erreur" ) ;
    }
}
```



L'instruction switch

- Elle permet de sélectionner un groupe précis d'instructions parmi plusieurs.

```
switch (expression)
{
    case constante1 :
        instruction1;
    case constante2 :
        instruction2;
    case constante_n :
        instruction_n;
    default:
        instruction;
}
```



Exemple

```
import java.util.Scanner;

class Demo {
    public static void main ( String argv[] ) {
        Scanner clavier = new Scanner( System.in) ;
        int jourSemaine = 0 ;
        System.out.println ( "Entrer un jour dans la semaine ( 1-7 )" ) ;
        jourSemaine = clavier.nextInt ();
        switch ( jourSemaine ) {
            case 1 : System.out.println ("Lundi") ;           break ;
            case 2 : System.out.println ("Mardi") ;           break ;
            ...
            case 7 : System.out.println ("Dimanche") ;        break ;
            default : System.out.println ("Erreur") ;
        }
    }
}
```



L'instruction while

- Elle permet de répéter une séquence d'instructions tant qu'une condition est vérifiée.

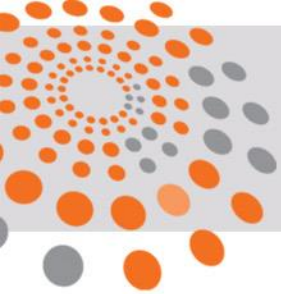
```
import java.io.*;
class Demo {
    public static void main( String args[]) {
        int nbr = 0;
        while( nbr < 10)
        {
            System.out.println ( nbr ) ;
            nbr++ ;
        }
    }
}
```



L'instruction do while

- Elle permet d'exécuter une séquence d'instructions. Mais contrairement au while elle teste sa condition d'arrêt en fin de boucle.

```
class Demo {  
    public static void main (String args[])  
    {   int nbr = 0;  
        do {  
            System.out.println ( nbr ) ;  
            nbr++ ;  
        } while ( nbr < 10 ) ;  
    }  
}
```



L'instruction for

```
for (expression1 ; expression2 ; expression3)  
    instruction;
```

```
class Demo  
{  
    public static void main (String args[])  
    {  
        for (int nbr = 0 ; nbr < 10 ; nbr++)  
        {  
            System.out.println ( nbr ) ;  
        }  
    }  
}
```



Les tableaux

- Un tableau est un ensemble d'éléments de même type. La mise en oeuvre se fait à l'aide de l'opérateur [].
- Les tableaux en java sont des objets, toute création se fait par référence à l'aide de l'opérateur new.

```
int tableauInt1[ ] ;           // ou aussi : int [ ] tableauInt1 ;
```

```
tableauInt1 = new int[4] ;
```

```
int tableauInt2[ ] = new int[4] ;
```

- Lors de la création, les éléments du tableaux sont initialisés par défaut :

0 pour les entiers et les flottants

null pour les objets

- On peut initialiser un tableau lors de sa création

```
int [ ] tab = { 10 , 20 , 30 , 40 , 50 } ;
```





Manipulation de tableaux

- Toute opération s'effectue élément par élément. L'accès à un élément se fait avec l'identificateur du tableau suivi d'un indice.

```
for (int i = 0 ; i < tableauInt1.length ; i++)  
    System.out.println ( tableauInt1[i] ) ;
```

- Les opérateurs d'affectation et d'égalité sont applicables sur les tableaux mais entraînent une manipulation de leurs références.

```
class Demo {  
    public static void main (String args [ ]) {  
        int tab1[] = { 10 , 20 , 30 } ; int tab2 [] = new int [3] ;  
        tab2 = tab1 ;  
        tab2[0] = 100 ;  
        System.out.println ( tab2[0] + " " + tab1[0] ) ;  
    }  
}
```



Manipulation de tableaux

Java fournit également un ensemble de services sur les tableaux dans la classe Arrays : tri, recherche, copie ...

```
public class Demo {  
    public static void main ( String args[] ) {  
        int tab[] = { 1 , 5 , 4 , 2 , 3 } ;  
        int pos ;  
        Arrays.sort ( tab ) ;  
        for ( int i = 0 ; i < 5 ; i++ )  
            System.out.println ( tab[i] ) ;  
        pos = Arrays.binarySearch ( tab , 3 ) ;  
        System.out.println ( "pos " + pos ) ;  
        Arrays.fill ( tab , 0 ) ;  
        for ( int i = 0 ; i < 5 ; i++ )  
            System.out.println ( tab[i] ) ;  
    }  
}
```



Tableaux Multidimensionnels

- Java donne la possibilité de créer des tableaux à plusieurs dimensions :

```
class Demo
```

```
{
```

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

```
{
```

```
    int [][] matrice1 = new int [2][3] ;
```

```
    int [][] matrice2 = { {10 , 20 , 30 } ,  
                           { 40 , 50 , 60 } } ;
```

```
    for ( int i = 0 ; i < 2 ; i++ )
```

```
        for ( int j= 0 ; j<3 ; j++ )
```

```
            System.out.println ( matrice2[i][j] ) ;
```

```
    }
```

```
}
```

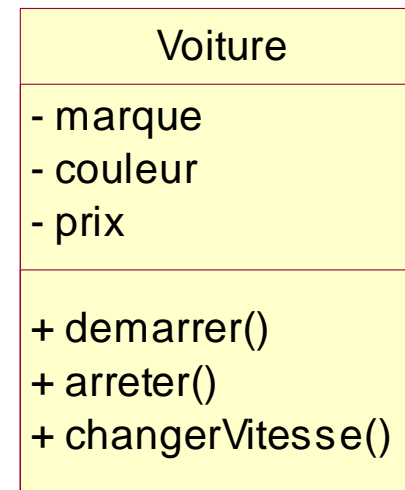


Concepts de classe et d'objet

Qu'est ce qu'une classe

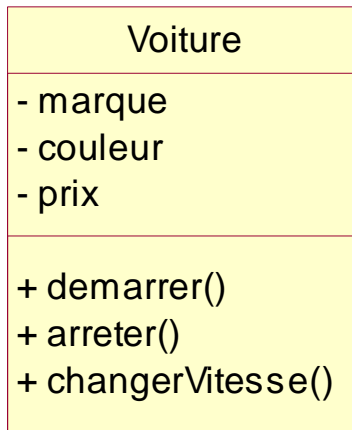
- Une classe décrit une famille d'objets, elle regroupe l'ensemble de caractéristiques communes de ses objets :
 - Les attributs contenant les valeurs propres de chaque objet,
 - Les opérations caractérisant les services proposés par les objets de la classe,
 - Et les méthodes implantant (le code) de ces opérations

Classe des voitures
(représentation UML)

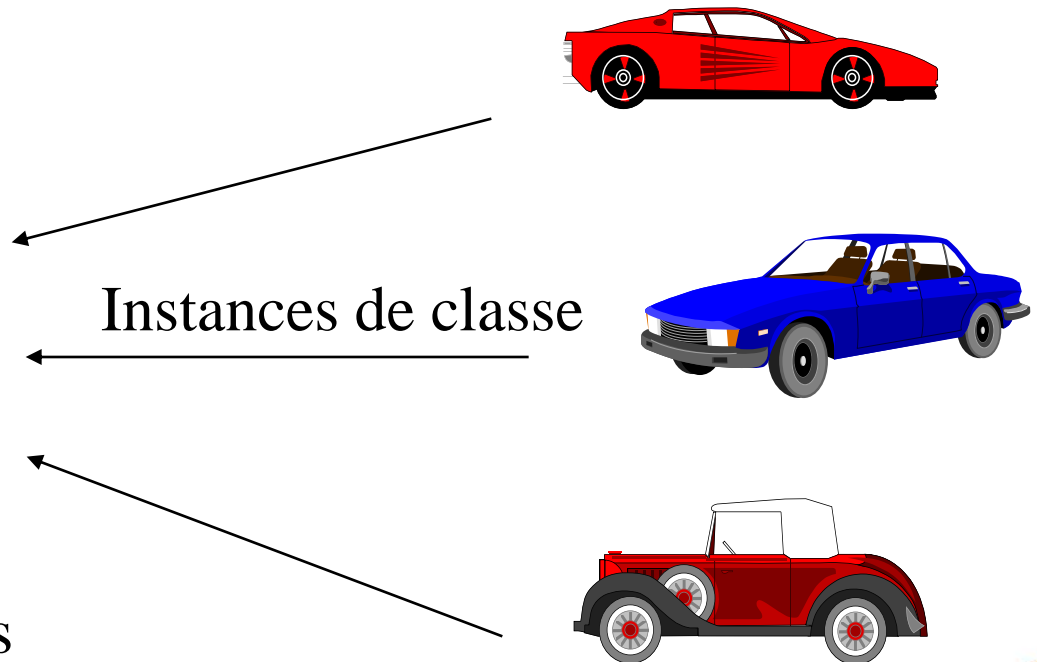


Classe et objet

- Une classe est un moule à fabriquer des objets. Un objet est une instance d'une classe :
 - Possédant des valeurs de propriétés qui lui sont propres
 - Partageant les mêmes comportements avec les autres instances de sa classe

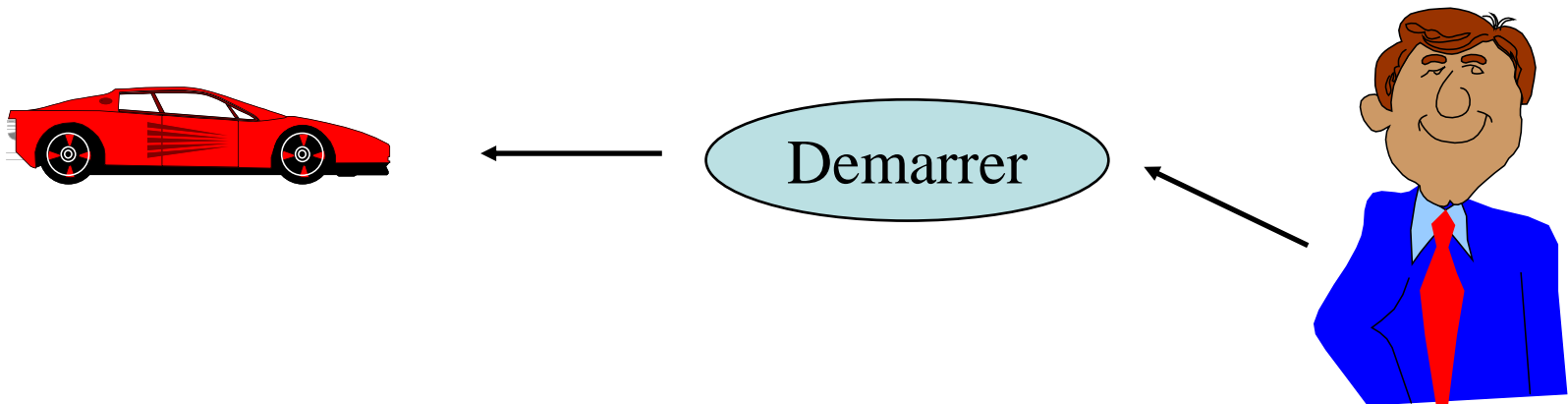


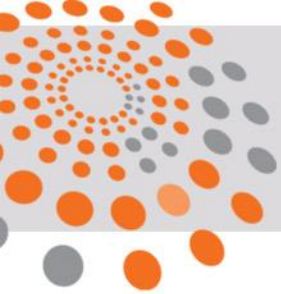
Classe des voitures



Les messages

- Un objet est une entité **encapsulée**
 - La classe est garante de cette encapsulation
- Toute manipulation d'objet s'effectue uniquement à travers les opérations (les services) définies sur sa classe





Les messages

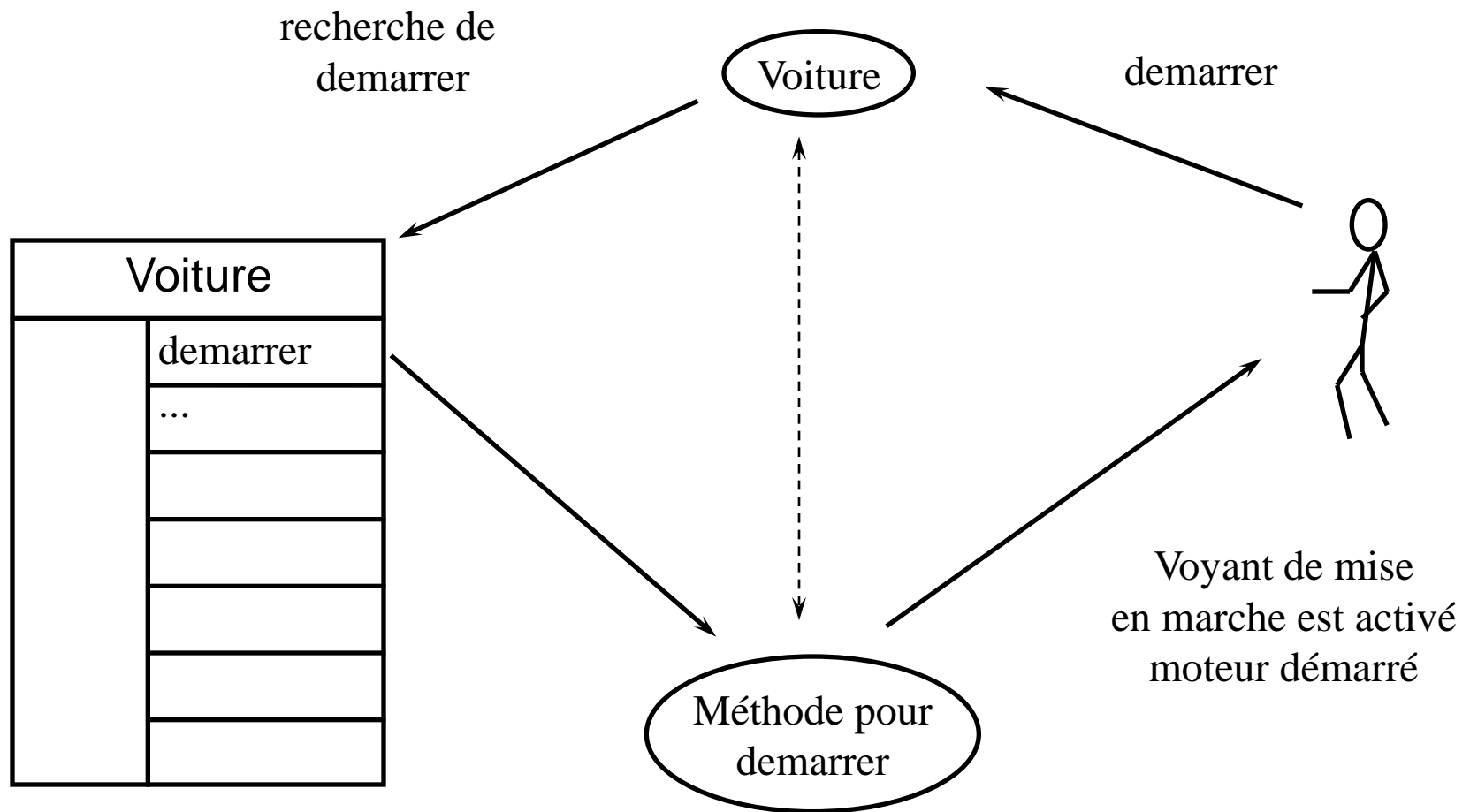
- La sollicitation de l'opération « demarrer » correspond à une demande d'exécution de ce service par l'objet
- Une telle requête est appelée « un message »
- La réception d'un message par un objet provoque la recherche de la méthode (du code) à appliquer.
- Cette recherche à lieu dans la classe.



Méthode

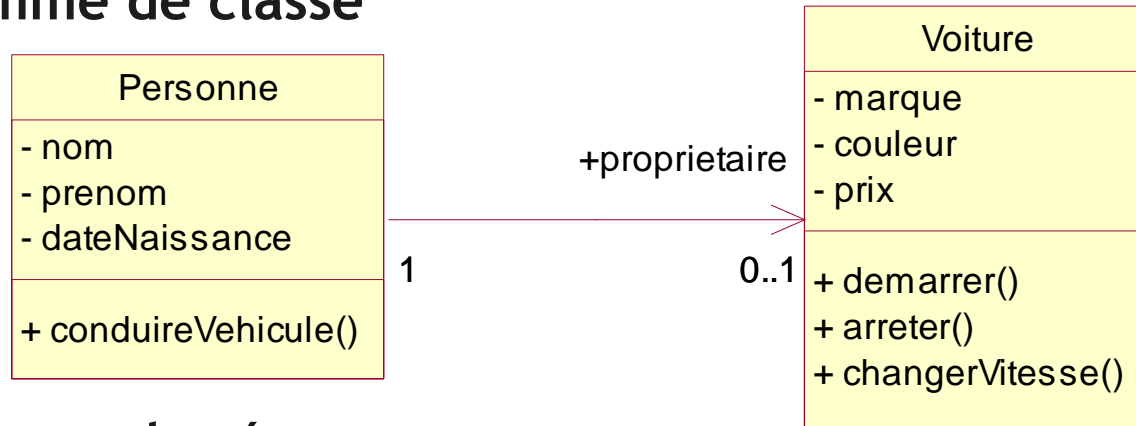
- La fonction exécutée lors de la réception du message est appelée une méthode
- Ce n'est pas à celui qui émet le message de connaître la méthode à exécuter
- C'est l'objet lui-même qui connaît la méthode adéquate pour répondre au message (encapsulation)

Recherche de méthode

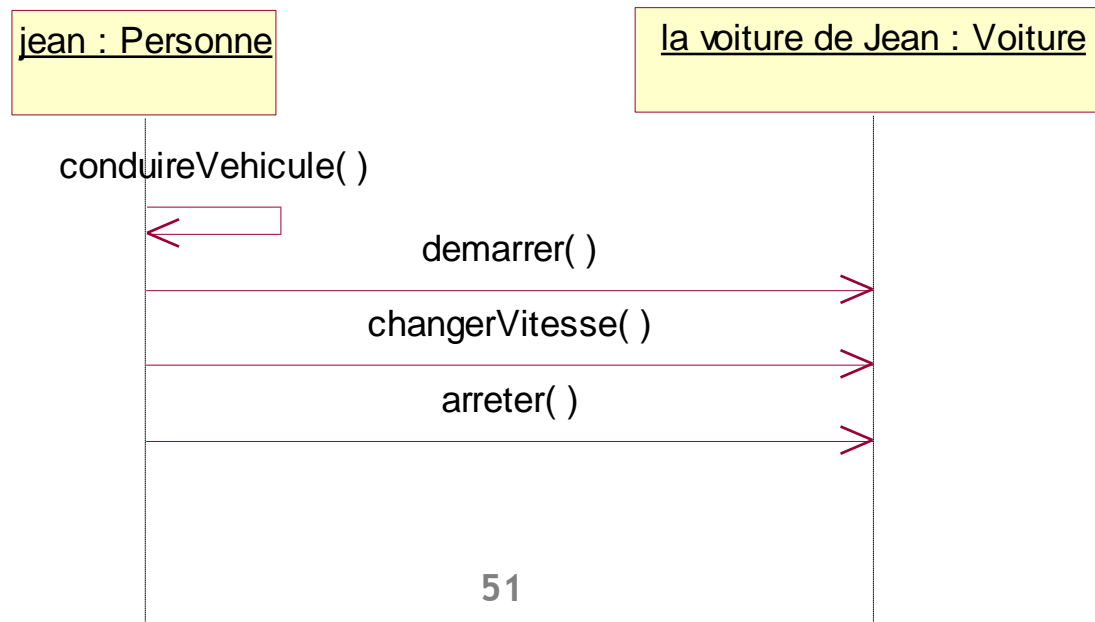


Représentation UML

- Diagramme de classe



- Diagramme de séquence





Les classes et les objets en Java





Le type classe

- Une classe est un type décrivant un groupe d'objets ayant des propriétés similaires (attributs) et des comportements communs (méthodes).
- Le mot réservé "**class**" permet de construire une classe.
- Chaque champ de la classe doit être qualifié :
 - private** : spécifie que le champ est non accessible en dehors de la classe
 - public** : spécifie que le champ est accessible en dehors de la classe
 - protected**: spécifie que le champ est accessible par les sous-classes
 - package**: est la visibilité par défaut en Java. Le champ est accessible pour toutes les classes du même package
- En général les attributs sont protégés et sont donc spécifiés « **private** ».



Les méthodes

- Elles permettent de définir des opérations sur les objets.

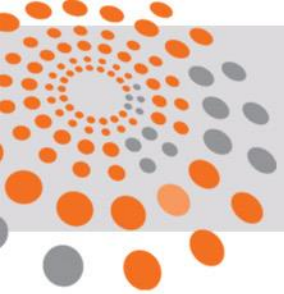
Type valeur_retour identificateur (<liste arguments>)

```
{    // Déclaration
```

```
    // Instruction
```

```
}
```

- Type valeur_retour correspond au type de la valeur retournée par la fonction.
- Liste d'arguments (type arg1 , ... , type argn) spécifie les arguments attendus pour la réalisation du traitement.
- Le type void est applicable sur les méthodes pour spécifier que la méthode ne produit pas de résultat.
- L'instruction return permet de renvoyer le résultat du traitement.



Example

```
class Point
{
    private int x;
    private int y;

    public void modifier ( int _x, int _y )
    {
        x = _x ;
        y = _y ;
    }

    public void print()
    {
        System.out.println ( " x : " + x + " y : " + y ) ;
    }
}
```



Instanciation d'une classe

- Une classe est un "moule" à partir duquel vont s'élaborer des objets.
- Toute création d'objet se fait par référence, par appel de l'opérateur **new**.

```
class Demo
{
    public static void main (String args [ ])
    {
        Point p1 = null;
        p1 = new Point () ;
        p1.modifier ( 3 , 6 ) ;
        p1.print() ;
    }
}
```

- Les méthodes s'invoquent sur une instance de la classe et à l'aide de l'opérateur '.'.



Surcharge de méthodes

- Le langage donne la possibilité de surcharger les méthodes.

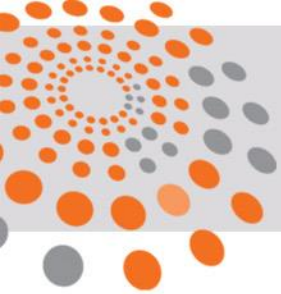
```
class Point {  
    private int x ;  
    private int y ;  
  
    public void modifier ( int _x, int _y )  
    {  
        x = _x ; y = _y ;  
    }  
  
    public void modifier ( Point p )  
    {  
        modifier(p.x, p.y);  
    }  
  
    public void print()  
    { System.out.println ( " x : " + x + " y : " + y ) ; }  
}
```



Passage d'arguments

- Le passage d'argument s'effectue toujours par valeur sur les types primitifs
- Le passage d'argument s'effectue par référence sur les objets

```
public void modifier ( Point p ) {  
    x = p.x ;      y = p.y ;  
    // p.x = 0 ;  p.y = 0 ;  
}  
  
class Demo {  
    public static void main (String args [ ] ) {  
        Point p1 = new Point () ;  
        Point p2 = new Point() ;  
        p2.modifier ( 3 , 6 ) ;  
        p2.print () ;  
        p1.modifier ( p2 ) ;  
        p2.modifier ( 30,0 ) ;  
        p1.print () ;  
        p2.print () ;  
    }  
}
```



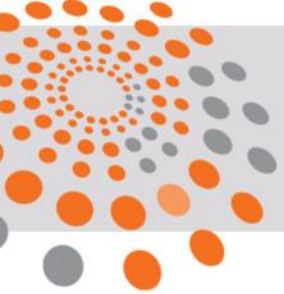
Les accesseurs

- Un objet n'est manipulable que par les méthodes spécifiées dans sa classe.
- Toute tentative de manipulation de la partie privée entraîne une erreur à la compilation.
- Pour accéder aux données il faut donc prévoir des méthodes : les accesseurs.
- Ils peuvent être en lecture ou en écriture et doivent contrôler l'intégrité des données.



Exemple

```
class Point {  
    private int x, y;  
  
    public int getX() { return x ; }  
    public int getY() { return y ; }  
  
    public void setX ( int _x )  
    {   if ( _x >=0 ) x = _x ;  
        else System.out.println ("erreur valeur négative " ) ;  
    }  
    public void setY ( int _y )  
    {   if ( _y >=0 ) y = _y ;  
        else System.out.println ("erreur valeur négative " ) ;  
    }  
    // ... autres méthodes  
}
```



Example

```
class Demo
{
    public static void main (String args [ ])
    {
        Point p1 = new Point () ;
        p1.setX ( 20 ) ;
        p1.print() ;
        p1.setX ( -5 ) ;
        p1.print() ;
    }
}
```

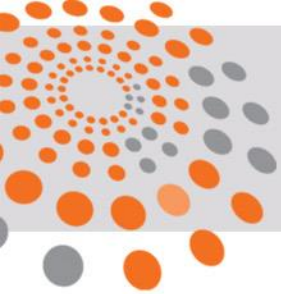


Initialisation des attributs d'une classe

- Il est souvent nécessaire d'initialiser les attributs pour donner un sens aux objets. Dans ce cas il faut prévoir une méthode à cet effet.

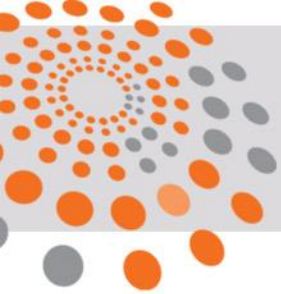
```
class Point {  
    private int x , y ;  
    public void init ( int _x , int _y )  
    {   x = _x ; y = _y ; }  
    ...  
}
```

```
class Demo {  
    public static void main (String args [ ])  
    {  
        Point p1;  
        p1 = new Point () ;  
        p1.init ( 10 , 20 ) ;  
        p1.print() ;  
    }  
}
```



Le constructeur

- Cette technique n'est guère satisfaisante, l'initialisation d'une instance nécessite l'appel explicite d'une méthode après sa création.
- Le constructeur résout ce problème.
- Un constructeur est une méthode :
 - appelée à chaque création d'instance,
 - homographe de la classe,
 - ne possède pas de type (même pas void),
 - si le constructeur contient des arguments, il faut les renseigner à la création des instances.



Exemple de constructeur

```
class Point
{
    private int x , y;
    public Point ( int _x , int _y ) {
        x = _x ; y = _y ;
    }
    // ... autres méthodes
}

class Demo
{
    public static void main (String args [ ]) {
        Point p1;
        p1 = new Point (10 , 20 ) ;
        p1.print() ;
    }
}
```




Constructeurs multiples

Il est souvent utile d'offrir plusieurs constructeurs permettant plusieurs modes d'initialisation possibles :

```
class Date {  
    private int jour, mois, annee;  
    public Date (int _jour , int _mois , int _annee) {  
        jour = _jour ; mois = _mois ; annee = _annee; }  
    public Date() { jour = 1 ; mois = 1 ; annee = 1995; }  
    public Date ( String date ) { ... }  
    // autres méthodes ...  
}
```

```
class Demo {  
    public static void main (String args[ ]) {  
        Date aujourd_hui, demain, autre ;  
        aujourd_hui = new Date (1, 2, 95 );  
        demain = new Date ( "01 02 95" );  
        autre = new Date ( );  
    }  
}
```



Initialisation des attributs

- Les attributs d'une classe peuvent être initialisés lors de la définition d'une classe mais dans ce cas :

```
class Date
{
    private int jour = 1;
    private int mois = 1;
    private int annee = 1995 ;

    // public Date() { jour = 1 ; mois = 1 ; annee = 1995; } inutile ?
    public Date() {} // et là .... c'est utile ?
    public Date (int _jour , int _mois , int _annee) {
        jour = _jour ; mois = _mois ; annee = _annee;
    }
    ...
}
```



Le garbage collector

- Tout objet est créé par un appel explicite de l'opérateur new et toute manipulation s'effectue par référence.
- Un objet conserve son emplacement mémoire tant qu'il est référencé ce qui permet de mémoriser son état.
- Dès qu'un objet n'est plus référencé, son emplacement mémoire va être récupéré.
- La récupération de la mémoire (libération) est assurée par le garbage collector.
- Le garbage collector (ramasse-miettes) est un processus dont l'exécution est indépendante de l'application qui libère la mémoire quand celle-ci n'est plus utilisée.
- Il est possible de prévoir un traitement pour la libération (méthode finalize()) qui sera appelée implicitement par le ramasse-miettes ou invoqué explicitement par le développeur.



Exemple : méthode finalize et appel du garbage collector

```
public class Date
{
    ...
    public void finalize()
    {
        System.out.println("Destruction d'une date !");
    }

    public static void main(String[]arg)
    {
        Date uneDate = new Date();

        uneDate = null;
        System.gc();      // Destruction d'une date !

    }
    ...
}
```



Affectation d'objet

- L'opérateur d'affectation est applicable sur les objets issus d'une même classe.

```
class Demo {  
    public static void main (String args [ ]) {  
        Point p1, p2;  
        p1 = new Point (10 , 20 );  
        p2 = new Point (10 , 20 );  
        p2 = p1 ;  
        p2.setX ( 50 ) ;  
        p2.print() ;  
        p1.print () ;  
    }  
}
```

- Attention, l'affectation est une affectation de référence.



Égalité entre objets

- L'opérateur d'égalité est applicable entre références d'objets de même classe.

```
class Demo {  
    public static void main (String args [ ])  
    {  
        Point p1, p2;  
        p1 = new Point (10 , 20 );  
        p2 = new Point ( 10 , 20 ) ;  
        if ( p1==p2 )  
            System.out.println ( "egalite " ) ;  
        else  
            System.out.println ( "different " ) ;  
    }  
}
```

- Attention, l'égalité s'applique sur les références des objets.
- Si on recherche une égalité de valeur, on doit redéfinir la méthode equals ().



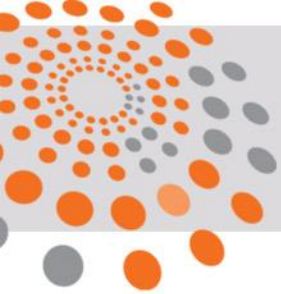
La variable this

- La variable this est un argument implicite pour chaque méthode d'instance.

```
class Demo
{
    private int i;
    public int getI()    { return i ; }
}
```

- Chaque méthode non statique contient implicitement un argument du type de la classe dans laquelle elle a été définie

```
class Demo
{
    private int i;
    public int getI ( Demo this ) { return this.i ; }
}
```



Variable, méthode, instance de classe

- Chaque instance d'une même classe possède sa propre représentation caractérisée par :
 - Des variables d'instance : ce sont les attributs d'un objet. Elles servent à mémoriser l'état d'un objet.
 - Des méthodes d'instance : méthodes s'exécutant dans le contexte d'une instance.
- On peut aussi définir :
 - Des variables de classe : ce sont des variables globales à une classe. Elles permettent de mémoriser l'état d'une classe.
 - Des méthodes de classe : méthodes s'exécutant dans le contexte de la classe (les attributs de classe)



Le qualificatif static

- Le qualificatif static permet de définir des variables et méthodes de classe.
- Les attributs qualifiés de statique n'ont qu'une représentation en mémoire, partagée par l'ensemble des instances de la classe.

```
class CptInstance {  
    private int i = 0;  
    private static int compteur = 0;  
    public CptInstance()        { compteur++ ; }  
    public int getCompteur ()    { return compteur ; }  
    public void finalize ()      { compteur-- ; }  
}  
class Demo {  
    public static void main (String args[]) {  
        CptInstance cpt = new CptInstance();  
        System.out.println(cpt.getCompteur());  
    }  
}
```

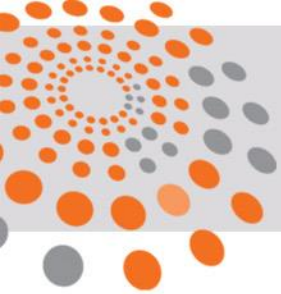


Le qualificatif static (suite)

- Le qualificatif static est également applicable sur les méthodes pour définir les méthodes de classe.
- Les méthodes de classes peuvent être accédées directement par l'identificateur de classe (sans passer par une instance).

```
class CptInstance {  
    private static int compteur = 0;  
    public CptInstance()           { compteur++ ; }  
    public static int getCompteur() { return compteur ; }  
    public void finalize ()        { compteur-- ; }  
}  
class Demo {  
    public static void main (String args[]) {  
        System.out.println (CptInstance.getCompteur() ) ;  
    }  
}
```

- Une méthode statique ne peut pas utiliser un attribut non statique.



Les classes de base Java

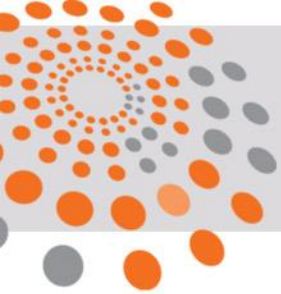
- Java définit un package `java.lang` qui est importé implicitement et qui forme le prolongement naturel du langage.
- Ce package contient un ensemble de classes de base :
 - `System` // interactions avec le system
 - `String` // implémente le type chaîne de caractères
 - Les wrappers // encapsulation des types primitifs
 - `Math` // Traitements mathématiques
 - `Class` // Classe d'un objet
 - ...



La classe String

```
class String {  
    private char value[] ;  
    private int offset ;  
    private int count ;  
  
    public String() { value = new char[0] ; }  
  
    public String ( String value )  
    {   count = value.length() ;  
        this.value = new char[count] ;  
        value.getChars(0, count, this.value, 0) ;  
    }  
  
    public String ( char value[] )  
    {   this.count = value.length ;  
        this.value = new char[count] ;  
        System.arraycopy ( value, 0, this.value, 0, count ) ;  
    }  
}
```

...



La classe String (suite)

...

```
public char charAt(int index)
{   if ((index < 0) || (index >= count))
        throw new StringIndexOutOfBoundsException(index) ;
    return value[index + offset] ;
}
```

```
public int compareTo(String anotherString) { .... }
public String concat ( String str ) { ... }
public int length() { ... }
public static String valueOf ( int i ) { .... }
public static String valueOf ( float f ) {... }
```

```
// et bien d'autres ...
}
```



Utilisation de la classe String

- L'affectation d'une constante chaîne dans une String entraîne sa création. Ce mode de création est préconisé.
- Une String ne peut pas être modifiée.

```
class Appli {  
    public static void main (String args [ ])  
    {  
        String str1 , str2 , str3 ;  
        str1 = new String ( "toto" ) ;  
        str2 = "titi" ;  
        System.out.println ( str1 + " " + str2 ) ;  
        str3 = str1.toUpperCase () ;  
        System.out.println ( str1 + " " + str3 ) ;  
    }  
}
```

- Si on veut pouvoir modifier une chaîne, il faut utiliser la classe StringBuffer.

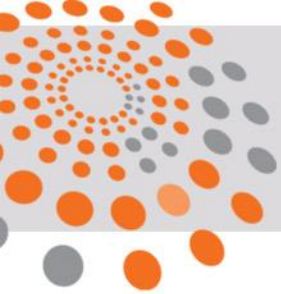


Opération sur les chaînes

- Les opérateurs =, == s'appliquent sur les références uniquement
- L'opérateur + permet la concaténation de 2 chaînes.

```
class Appli {  
    public static void main (String args [ ]) {  
        String str1 , str2 , str3 ;  
        str1 = new String ( "toto" ) ;  
        str2 = new String ( "toto" ) ;  
        str3 = str1 + str2 ;  
        if ( str1 == str2 ) System.out.println ( "egalite" ) ;  
        else System.out.println ( "different " ) ;  
    }  
}
```
- L'égalité entre 2 chaînes s'effectuera donc avec la méthode equals

```
if ( str1.equals ( str2 ) ) // traitement
```



La classe StringBuffer

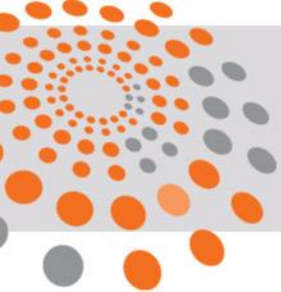
- Cette classe permet de définir des chaînes modifiables :

```
class Appli
{
    public static void main (String args [ ])
    {
        StringBuffer str1 ;
            str1 = new StringBuffer ( 20) ;
            str1.append ("Une " ) ;
            str1.append (" chaine" ) ;
            str1.setCharAt ( 5 , 'C' ) ;
            System.out.println ( str1 ) ;
        }
    }
```



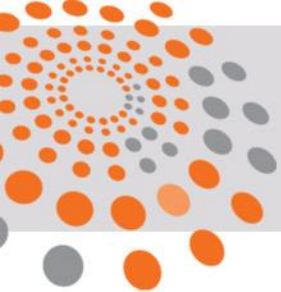

Les wrappers

- Les wrappers sont des classes qui encapsulent les types primitifs.
- Le package `java.lang` fournit un ensemble de classe wrappers :
 - Integer // implémente int de base
 - Long // implémente long de base
 - Character // implémente char de base
 - Float // implémente float de base
 - Boolean // implémente boolean de base



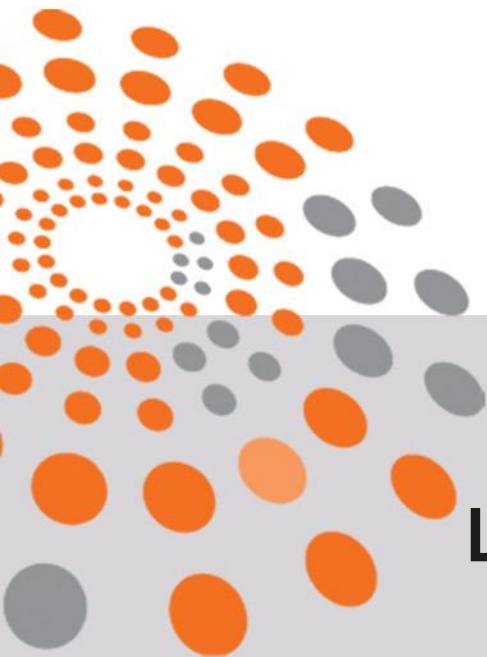
Services de base des wrappers

- Les wrappers offrent tous un ensemble de services de base :
- Un constructeur qui permet une instanciation à partir du type primitif :
 - `Integer unInteger = new Integer (10) ;`
- Un constructeur qui permet une instanciation à partir d'un objet String :
 - `Integer unInteger = new Integer ("10") ;`
- Une méthode (`typeValue`) pour fournir la valeur primitive représentée par l'objet:
 - `int entier = unInteger.intValue () ;`



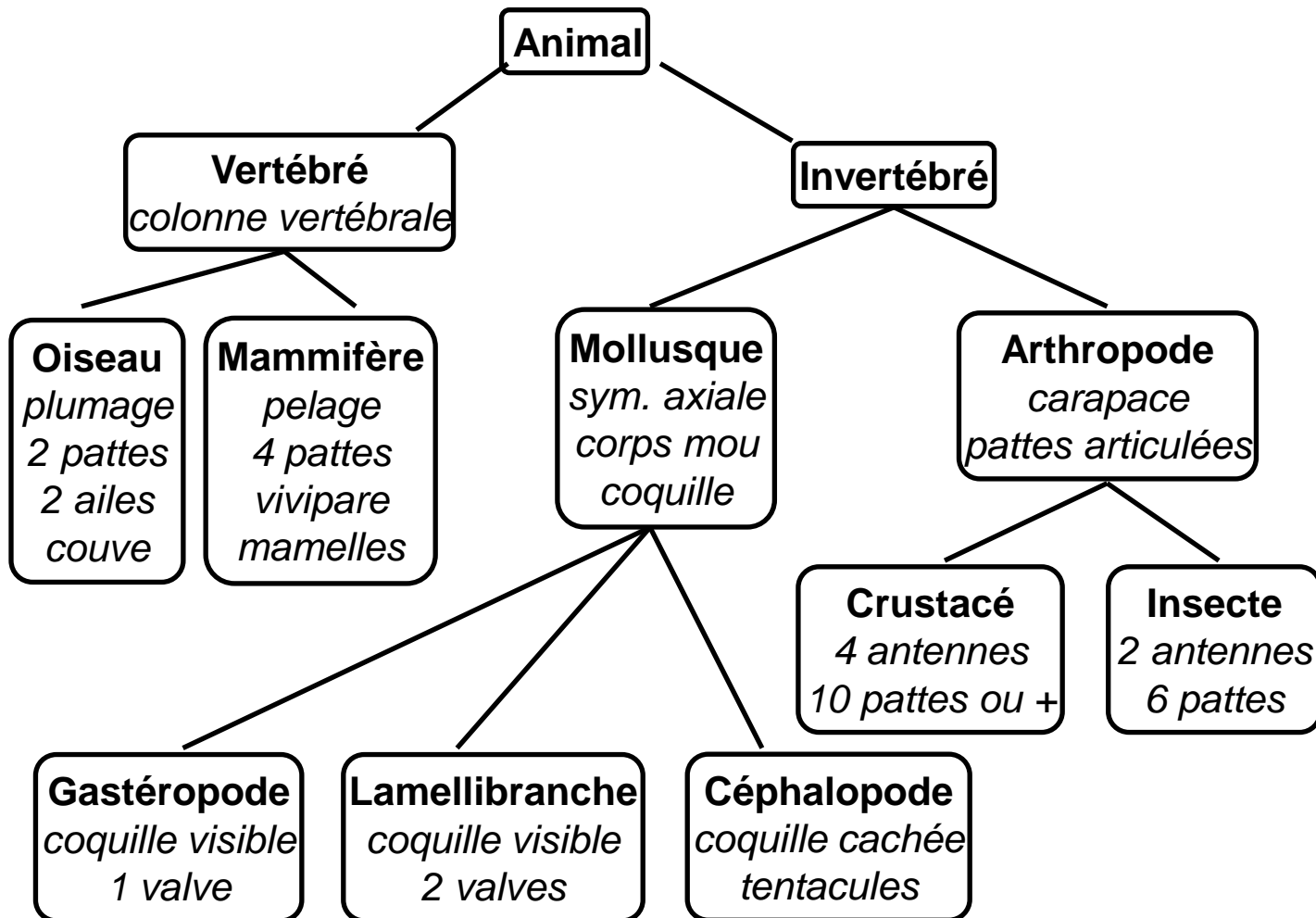
Services de base des wrappers

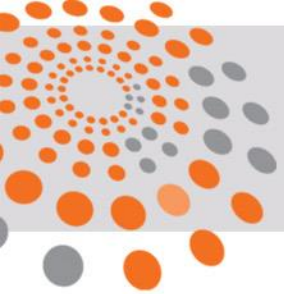
- Un ensemble de méthodes de conversions :
 - `Integer valeur = Integer.valueOf ("123");`
 - `int valeur2 = Integer.parseInt ("123");`
- Les classes `Integer`, `Long`, `Float` et `Double` définissent toutes les constantes:
 - `MAX_VALUE` valeurs Max
 - `MIN_VALUE` valeurs min
- Une méthode `equals()` pour l'égalité et une méthode `compareTo()` pour la comparaison.



Le concept d'héritage





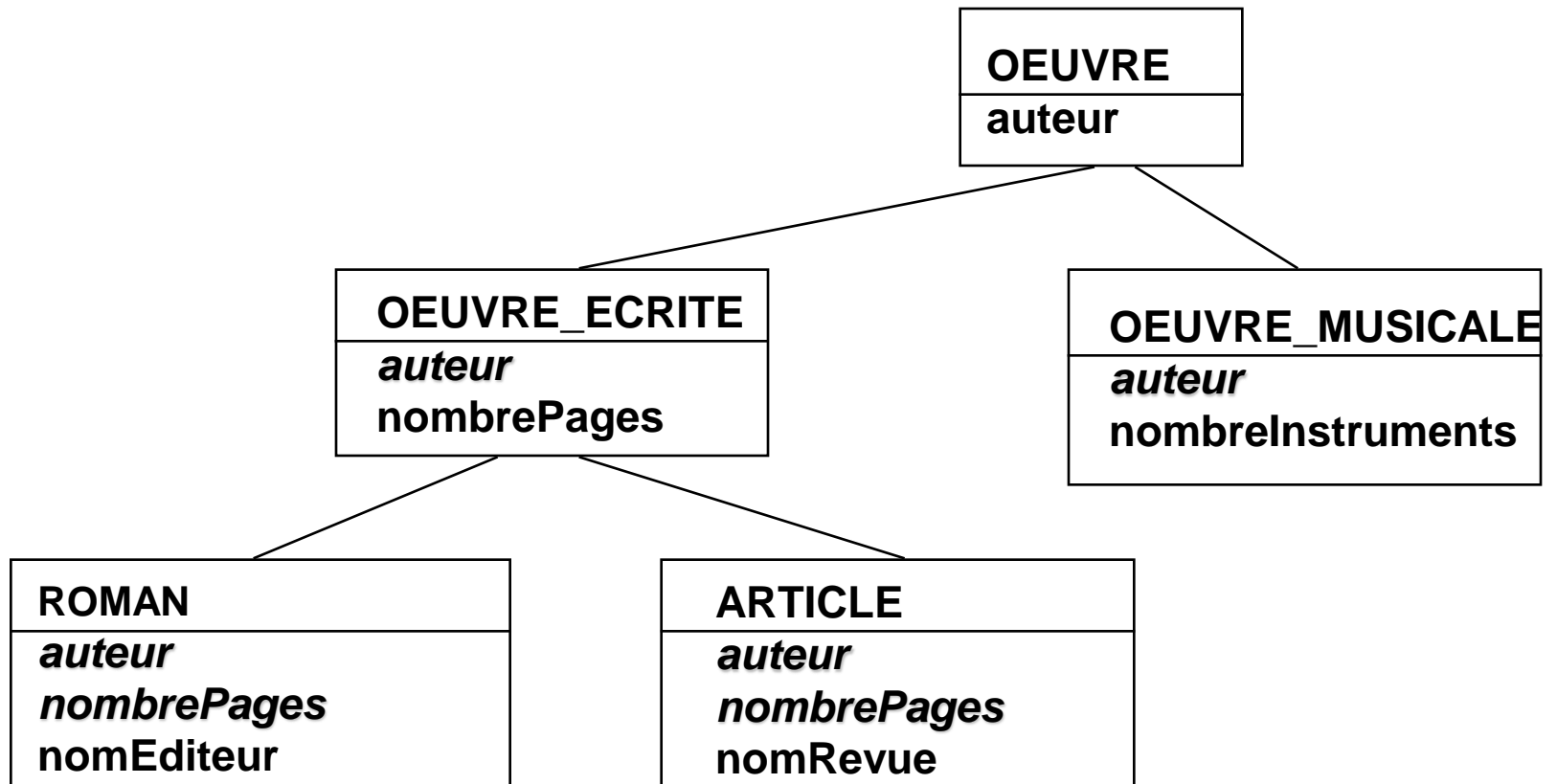


Héritage (suite)

- Tout vertébré est composé d'une colonne vertébrale
- Un oiseau est un vertébré donc il possède une colonne vertébrale
- On dit que "Oiseau" hérite de "Vertébré"

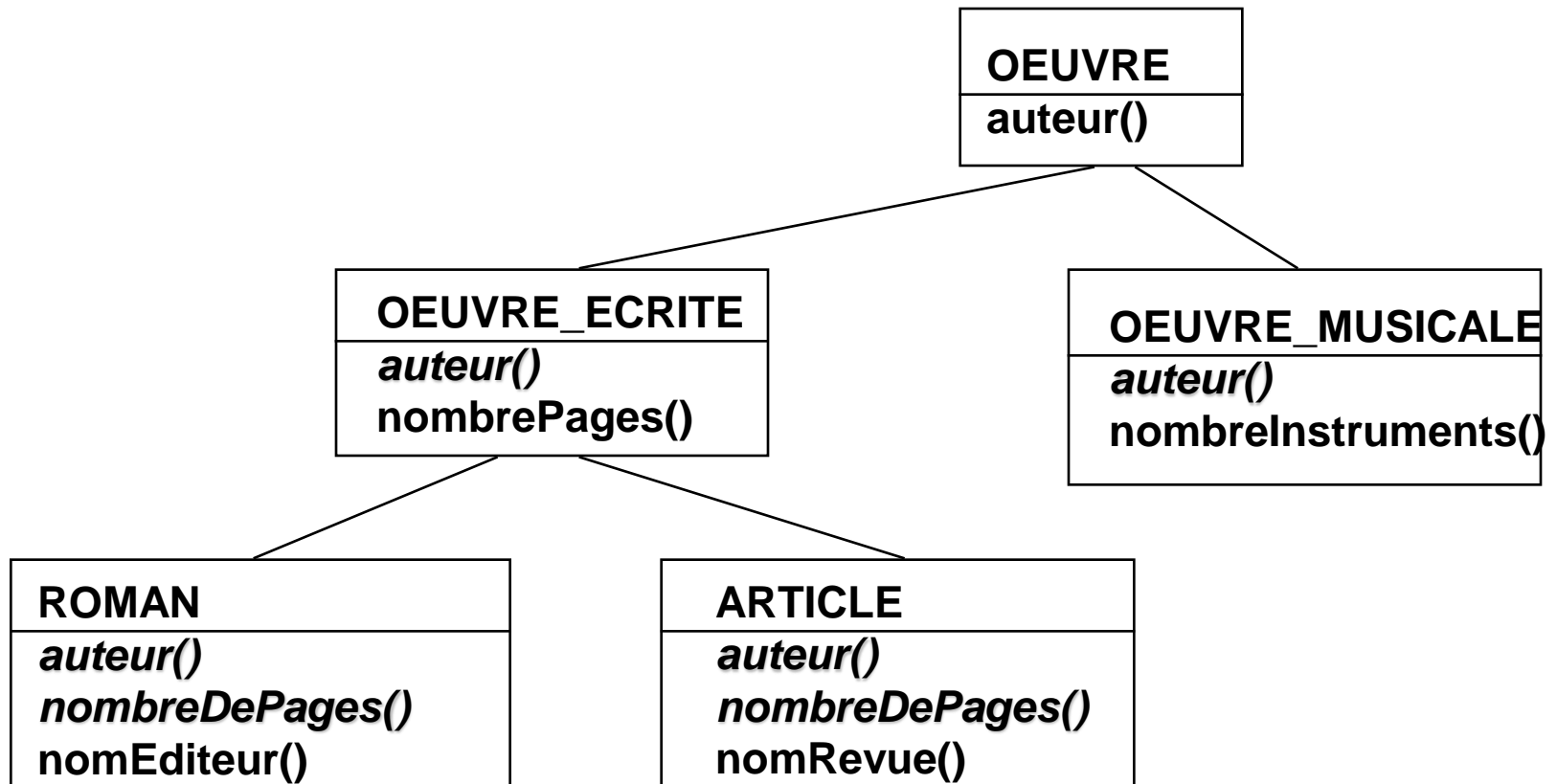
Héritage et structure

Héritage = réutilisation de structure

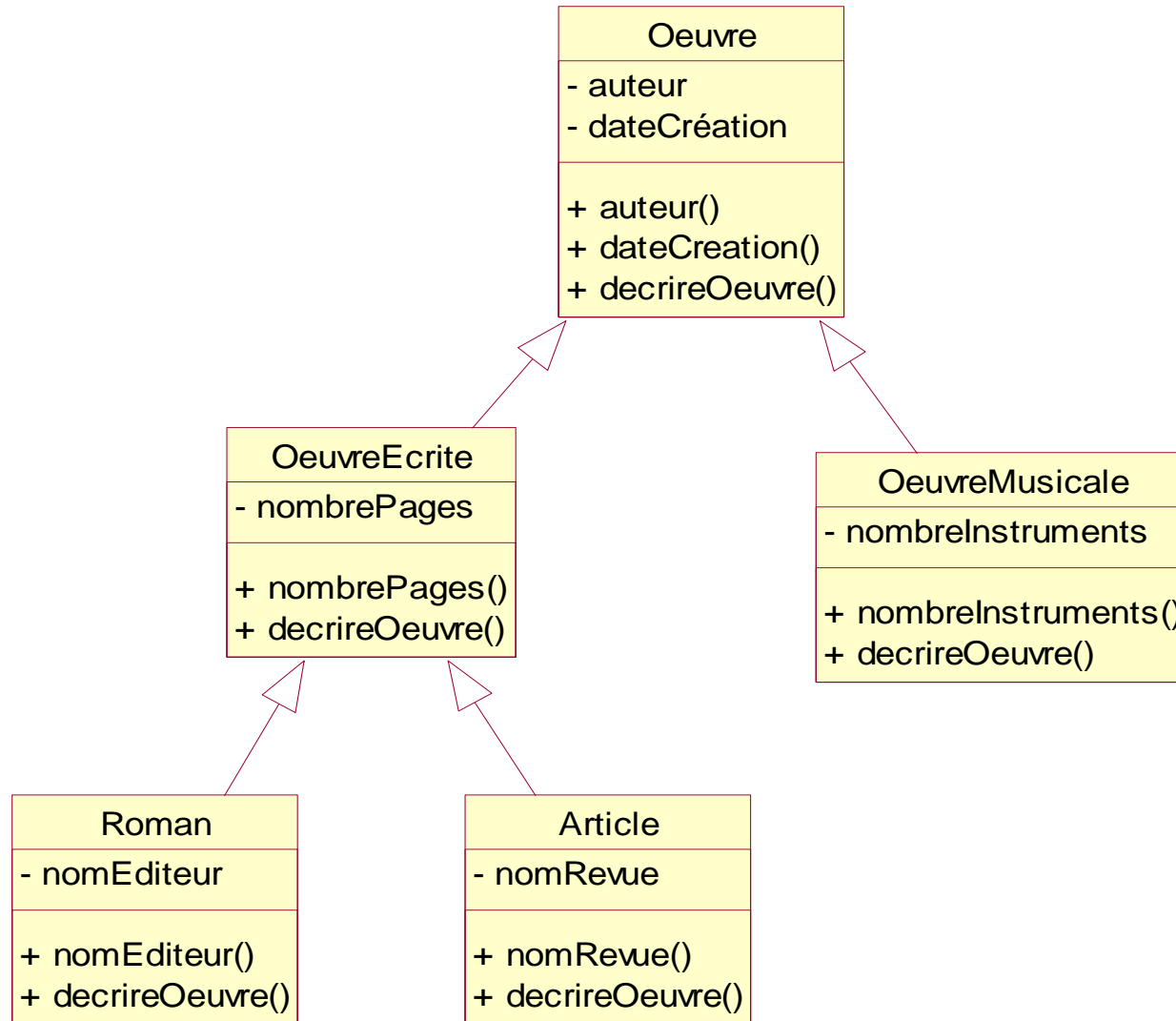


Héritage et comportement

Héritage = réutilisation de comportement



Héritage : modélisation UML



Définition d'une classe

CLASSE 1
donnée 1 donnée 2
comport 1 comport 2



CLASSE 2
<i>donnée 1</i> <i>donnée 2</i> donnée 3
<i>comport 1</i> <i>comport 2</i> comport 3



HERITAGE DES DONNEES:

DONNEES = DONNEES CLASSES
+ DONNEES SUPER CLASSES



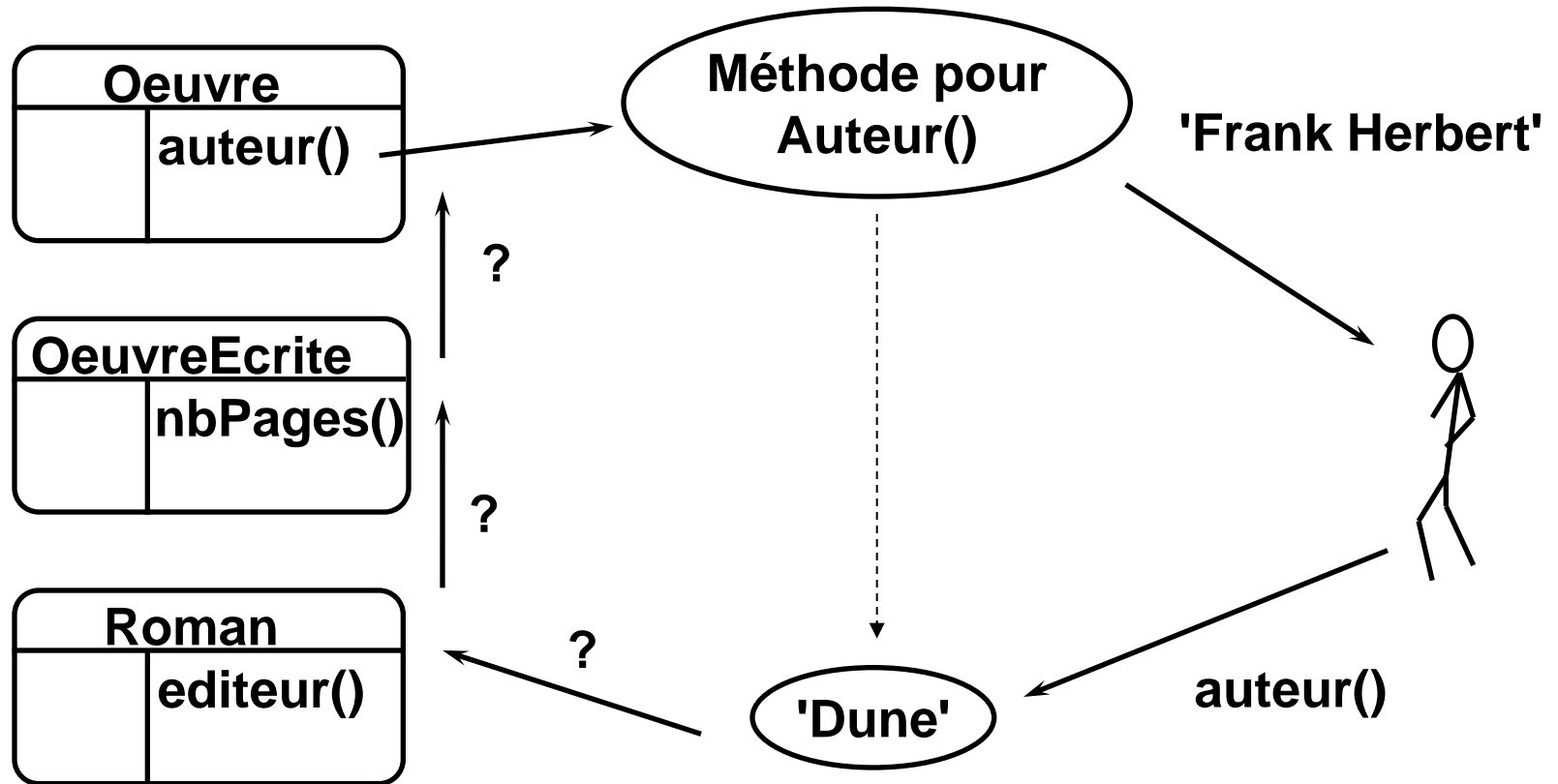
HERITAGE DES COMPORTEMENTS:

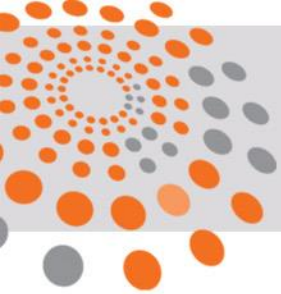
CMPTS = CMPTS SUPER CLASSES
+ CMPTS CLASSES
+ REDEFINITIONS CMPTS (éventuellement)

**Ne pas confondre héritage et
composition**

une voiture n'hérite pas de Roue et Carrosserie

Recherche de méthode



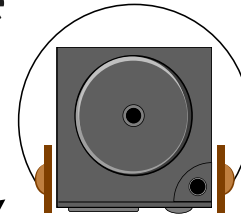
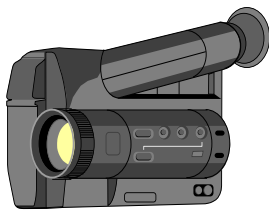


Polymorphisme

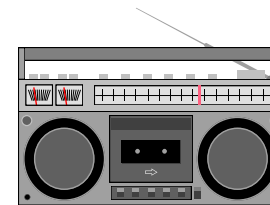
- Un lecteur laser ne fonctionne évidemment pas de la même manière qu'un lecteur de cassettes. Pourtant, leur mode d'emploi sont sensiblement voisins :
 - marche/arrêt
 - ouverture/fermeture de la porte
 - lecture/pause
 - avance/retour rapide
- sont autant d'opérations identiques exécutées différemment par chaque appareil

Le polymorphisme

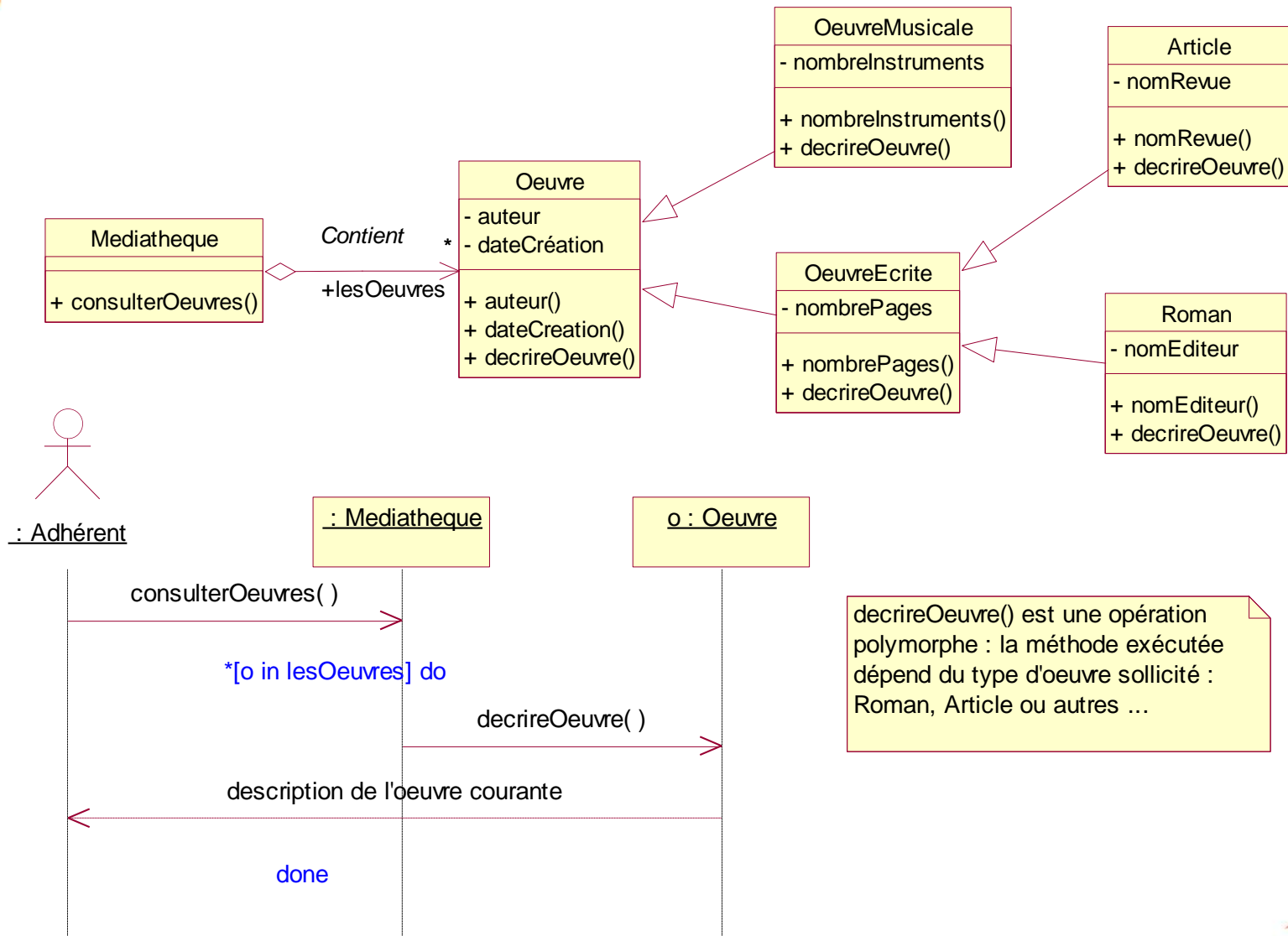
- Capacité à manipuler un ensemble d'objets à travers une interface de services homogène en faisant totalement abstraction de leur véritable nature



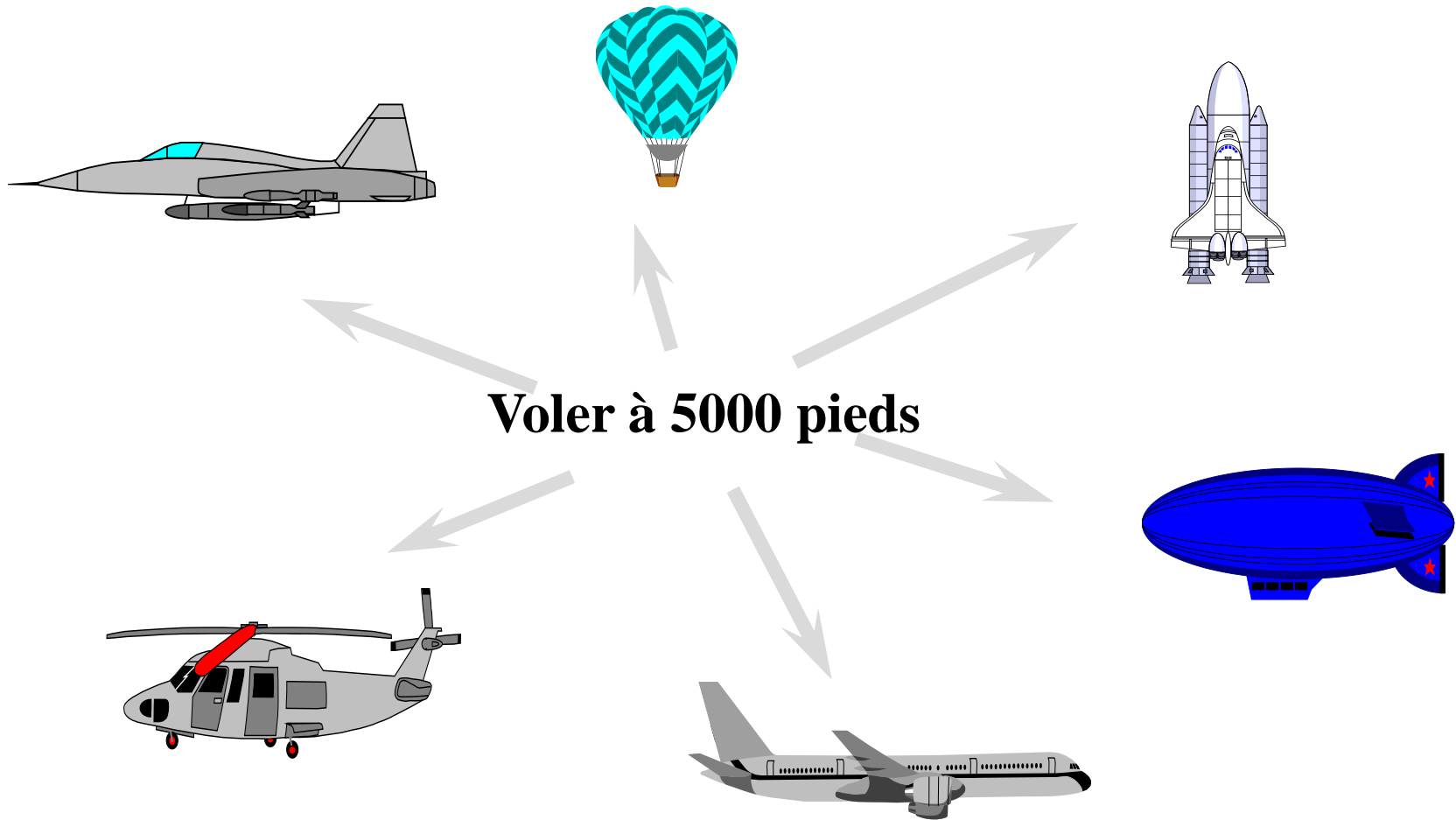
**Avance
Rapide**



Polymorphisme : modélisation UML



Le polymorphisme : autre exemple





L'héritage en Java





Héritage et classe dérivée

- L'héritage permet de créer une classe par extension ou spécialisation d'une classe existante.
- Chaque niveau (classe) amène un complément ou une spécialisation des caractéristiques du niveau supérieur auquel il est rattaché.
- Dès lors qu'une caractéristique (attribut ou méthode) est définie à un niveau elle reste valable pour tous les sous-niveaux affiliés.
- Ce mécanisme est appelé l'héritage. En java, si une classe D hérite d'une classe B, la classe D est appelée classe dérivée et B classe de base.



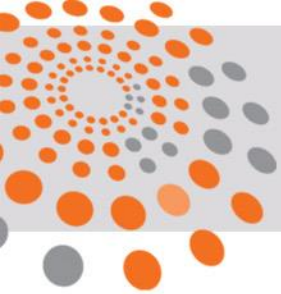
Exemple

```
public class Personne {  
    private String nom;  
    private String prenom;  
    private int age;  
  
    public Personne ()  
    {   nom = " " ; prenom = "" ; age = 0 ;   }  
  
    public Personne ( String _nom , String _prenom , int _age )  
    {   nom = _nom ; prenom = _prenom ; age = _age ; }  
  
    public void afficher()  
    {   System.out.println ( nom + " " + prenom + " " + age ); }  
  
    public void anniversaire()  
    {   age++ ; }  
}
```



Exemple (suite)

```
public class Stagiaire extends Personne {  
    private String cours;  
  
    public Stagiaire() { cours = "" ; }  
    public Stagiaire( String _nom , String _prenom , int _age ,String _cours )  
    { super ( _nom , _prenom , _age );  
      cours = _cours;  
    }  
  
    public String getCours () { return cours ; }  
    public void setCours ( String _cours ) { cours = _cours ; }  
  
    public void afficher ()  
    { super.afficher ();  
      System.out.println ("Cours : " + cours );  
    }  
}
```



Exemple (suite)

```
public class Appli
{
    public static void main (String args [ ])
    {
        Personne pers1;
        Stagiaire stag1;
        pers1 = new Personne ("Bond" , "james" , 7 );
        stag1 = new Stagiaire ( "Jones" , "Indiana" , 31 , "java" );
        pers1.afficher();
        stag1.afficher();
        pers1.anniversaire();
        stag1.anniversaire();
    }
}
```



La classe dérivée "stagiaire" (suite)

- Les caractéristiques héritées sont tributaires des attributs d'accès des membres de la classe de base.
- Les membres privés de la classe "Personne" sont inaccessibles aux méthodes de la classe "Stagiaire" :

```
class Stagiaire extends Personne {  
    public void afficher()  
    {   System.out.println ( nom + prenom + age + cours ) ;   // erreur  
    }  
}
```

- Les membres publics de la classe "Personne" sont bien-sûr accessibles aux méthodes de la classe "Stagiaire" :

```
class Stagiaire extends Personne {  
    public void afficher()  
    {   super.afficher(); System.out.println ("Cours : " , cours ) ; }  
}
```



L'attribut d'accès protected

- L'attribut d'accès protected permet de concéder un droit d'accès pour les classes dérivées.

```
class Personne
```

```
{
```

```
    protected String nom;
```

```
}
```

```
class Stagiaire extends Personne
```

```
{
```

```
    protected String cours;
```

```
    public void afficher()
```

```
    {
```

```
        System.out.println ( nom + " " + cours);
```

```
    }
```

```
}
```



Compatibilité des instances

- Pour une classe : `class Personne { ... }` on peut créer des instances :
`Personne pers1, pers2;`
`pers1 = new Personne ();`
- Pour une classe dérivée de `Personne` : `class Stagiaire extends Personne { .. }` , on peut créer de la même manière des instances :
`Stagiaire stag1 , stag2;`
`stag1 = new Stagiaire ();`
- Mais qu'en est-il de la compatibilité entre ces différents objets ?

<code>pers2 = pers1;</code>	<code>//ok</code>
<code>stag2 = stag1 ;</code>	<code>// bien sur ok</code>
<code>pers1 = stag1 ;</code>	<code>// ok compatibilité amont</code>
<code>stag2 = pers2 ;</code>	<code>// erreur types différents</code>
<code>stag2 = (Stagiaire) pers1</code>	<code>// ok conversion explicite mais ... !</code>



Exemple

```
public class Demo {  
    public static void main (String arg [ ])  
    {  
        Personne pers1 = new Personne ( "Jones" , "Indiana" , 35 );  
        Stagiaire stag1 = new Stagiaire ( "Bond", "James" , 7 , "Espionnage");  
        stag1.afficher (); // appel afficher() de la classe Stagiaire  
        pers1.afficher(); // appel afficher() de la classe Personne  
        System.out.println ( "Affectation de stag dans pers " );  
        pers1 = stag1 ; // cast non obligatoire compatibilité amont  
        pers1.afficher ();  
        stag1 = (Stagiaire)pers1;  
        stag1.afficher (); // là ça marche ... !  
        stag1 = (Stagiaire) new Personne ("Florent", "Claude", 20); // attention  
        stag1.afficher ();    }  
}
```

- La méthode appelée est toujours fonction de l'objet receveur et pas du type de référence (polymorphisme).
- L'opérateur instanceof permet de savoir si un objet appartient bien à une classe désirée : ***if(pers1 instanceof Stagiaire) { ... }***



La classe Object

- Toute classe hérite implicitement de la classe Object. L'ensemble des classes sous Java se présente donc sous la forme d'une hiérarchie.
- La classe Object possède un ensemble de méthodes qui seront applicables sur l'ensemble des classes :

```
public boolean equals ( Object obj ) ;
```

```
protected Object clone () ;
```

```
public String toString () ;
```

```
public final Class getClass () ;
```

```
protected void finalize () ;
```

```
public hashCode()
```



La classe Object

- La méthode `equals()` implémente une comparaison par défaut. Elle fait une comparaison de références (identique à l'opérateur `==`) :

```
obj1.equals(obj2) ; // true que si obj1 et obj2 désignent le même objet.
```

- La méthode `clone ()` fait une copie d'objet. Chaque classe souhaitant bénéficier de ce service doit redéfinir cette méthode sous peine de produire l'exception *CloneNotSupportedException*

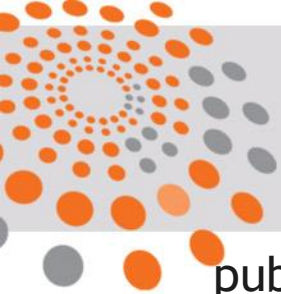
```
obj2 = obj1.clone ( );
```

- La méthode `toString ()` permet une conversion en chaîne de caractères. Elle renvoie le nom de la classe suivi du séparateur `@` lui-même suivi par la valeur de hachage de l'objet.

```
System.out.println ( obj1.toString () ) ;
```

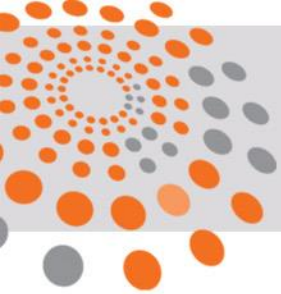
- La méthode `getClass()` renvoie un objet de la classe `Class` qui représente la classe de l'objet.

```
String nomClasse = obj1.getClass().getName () ;
```



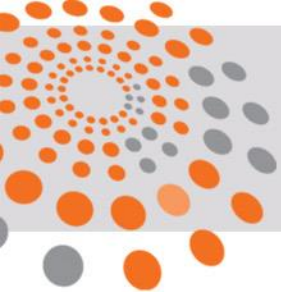
Utilisation des fonctionnalités de la classe Object

```
public class Point {  
    private int x, y;  
  
    public Point (Point p)                { this(p.x,p.y); }  
    public Point ( int _x , int _y )      { x = _x ; y = _y ; }  
    public boolean equals (Object o)      { Point p = (Point)o;  
                                           return x == p.x && y ==  
                                           p.y; }  
    public String toString()              { return "x = " + x + " y = " + y; }  
    public Object clone()                  { return new Point(x,y); }  
  
    public static void main (String args [ ]) {  
        Point p1 = new Point (10 , 20 );  
        Point p2 = new Point ( p1 ) ;  
        if ( p1.equals(p2) ) System.out.println ( p1 ) ;  
        else System.out.println ( p2) ;  
        Point px = (Point) p2.clone();  
        System.out.println ( px) ;  
    }  
}
```



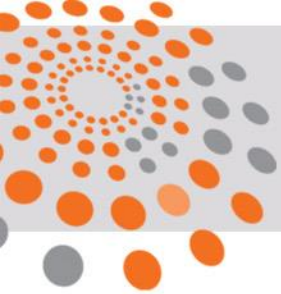
Les collections (`java.util`)

- Une collection est une entité qui gère un ensemble d'objets.
- Java fournit un framework pour les collections à partir du jdk1.2.
- Il existe trois types fondamentaux de collections
 - `List` (liste) : collection ordonnée, les objets peuvent être dupliqués
 - `Set` (ensemble) : liste sans doublon, les objets sont uniques
 - `Map` () : rassemble des paires clé/valeur
- Tous les types de collections supportent les opérations de base : ajouter, retirer, et itérer sur les objets de la liste.



Les Tableaux dynamique (ArrayList)

- Un ArrayList est un tableau dont la taille peut évoluer en fonction des besoins.
- Comme toute classe, il propose un ensemble de constructeurs :
 - ArrayList () ,
 - ArrayList (int initialCapacity),
 - ArrayList (Collection c).
- L'ajout des éléments dans un ArrayList s'effectue par une des méthodes add :
 - boolean add (Object obj) ;**
- La récupération s'effectue à l'aide de la méthode get :
 - Object get (int index) ;**



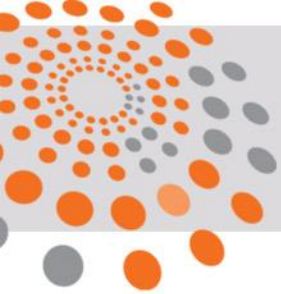
Exemple (avant JDK 1.5)

```
public class Demo
{
    public static void main (String args [ ])
    {
        ArrayList list ;
            list = new ArrayList () ;
            list.add ( "chaine1" ) ;
            list.add ( "chaine2" ) ;
            list.add ( "chaine3" ) ;
            for ( int i = 0 ; i < list.size () ; i++ )
                System.out.println ( list.get ( i ) ) ;
    }
}
```



Les génériques du JDK 1.5

- Depuis la version 1.5 du JDK, Java supporte la programmation générique.
- Les génériques sont des entités paramétrables
- Elles permettent de s'abstraire du typage des objets lors de la conception et donc de définir des comportements communs quel que soit le type des objets manipulés.
- Elles permettent :
 - d'accroître la lisibilité du code (plus de cast),
 - de renforcer la sécurité du code grâce à un meilleur du typage (vérification à la compilation).
- Premiers apports des génériques :
 - Toutes les collections de `java.util` sont maintenant des classes « génériques »



Le même exemple en JDK 1.5

```
class Demo
{
    public static void main (String args [ ])
    {
        ArrayList<String> list ;
            list = new ArrayList<String> () ;
            list.add ( "chaine1" ) ;
            list.add ( "chaine2" ) ;
            list.add ( "chaine3" ) ;
            list.add ( new Object ) ; // erreur à la compilation ... !
            for ( String s : list )
                System.out.println (s) ;
    }
}
```




Exemple : typage fort les collections

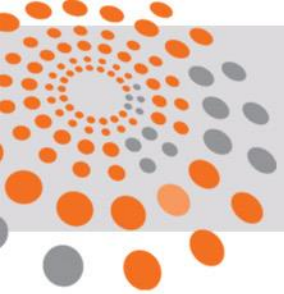
```
ArrayList<String> listeChaines = new ArrayList <String>(50);  
String valeur = null;  
for(int i = 0; i < 10; i++) {  
    valeur = "chaine" + i;  
    listeChaines.add(valeur);  
}  
for(int i=0; i< listeChaines.size(); i++;)  
    System.out.println(listeChaines.get(i));
```

- Toute tentative d'ajouts d'éléments autres que des String dans listeChaines entraînera une erreur à la compilation.
- On peut bien aussi utiliser la version simplifiée du for :

```
for(String str : listeChaines)  
    System.out.println(str);
```

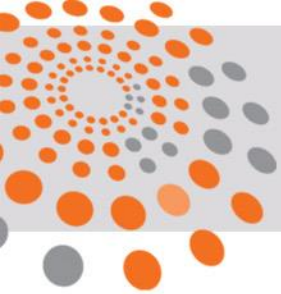


Les classes abstraites



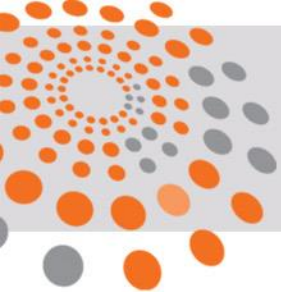
Classe abstraite

- Une des idées maîtresses de la programmation par objet consiste à construire de nouvelles classes sur la base de classes déjà existantes.
- Cette technique de programmation ne révèle toute sa puissance que si les classes de base sont génériques.
 - Ainsi, indépendantes de tout contexte spécifique, elles seront réutilisables dans un spectre plus large de situations.
- Développer une classe générique consiste à implanter des méthodes en s'appuyant sur des comportements abstraits;
 - méthodes abstraites définies par les classes dérivées.
- La définition d'une classe abstraite se fait à l'aide du qualificatif `abstract`.



Exemple

```
abstract class Vehicule {  
    private int nbrPassager ;  
    protected int nbKlm ;  
    // constructeurs + accesseurs  
    public abstract void demarrer();  
    public abstract void deplacer (int km);  
    public abstract void stopper ();  
  
    public void transporter ( int nbr, int km )  
    {   nbrPassager = nbr ;  
        this.demarrer();  
        this.deplacer( km);  
        this.stopper();  
    }  
}
```



Restrictions sur les classes abstraites

- Une classe abstraite n'est utilisable que comme classe de base par d'autres classes.
- Toute tentative de création d'objet à partir d'une telle classe provoque une erreur :

```
Vehicule unVehicule;  
unVehicule = new Vehicule (); // erreur
```
- Toute classe contenant au moins une méthode abstraite est considérée comme abstraite et définie comme telle à l'aide du qualificatif `abstract`.



Classes dérivées d'une classe abstraite

Une classe dérivée d'une classe de base abstraite doit :

- fournir une définition pour chaque méthode déclarée abstraite
- ou redéclarer ces méthodes abstraites.

```
class Bus extends Vehicule {  
    public void demarrer () { System.out.println ( "Demarrer Bus"); }  
    public void rouler () { System.out.println ( "Bus Rouler"); }  
        public void deplacer(int km) {nbKlm = km; rouler(); }  
    public void stopper () { System.out.println ( "Bus Stopper"); }  
}
```

```
public class Appli  
{  
    public static void main (String args[ ]) {  
        Bus unBus = new Bus ();  
        unBus.transporter (5, 20);  
    }  
}
```



Classes dérivées d'une classe abstraite (suite)

- Et maintenant pour un avion :

```
class Avion extends Vehicule {  
    public void decoller() { System.out.println ("l'avion décolle Bus"); }  
    public void demarrer () { this.decoller(); }  
    public void voler () { System.out.println ("Avion vole"); }  
    public void atterrir () { System.out.println ("Avion atterrit"); }  
        public void deplacer(int km) {nbKlm = km; voler(); }  
    public void stopper () { this.atterrir(); }  
}  
  
public class Appli  
{  
    public static void main (String args[ ]) {  
        Bus unBus = new Bus ();  
        Avion unAvion = new Avion ();  
        unBus.transporter (5, 20);  
        unAvion.transporter (200, 2000);  
    }  
}
```



Les interfaces

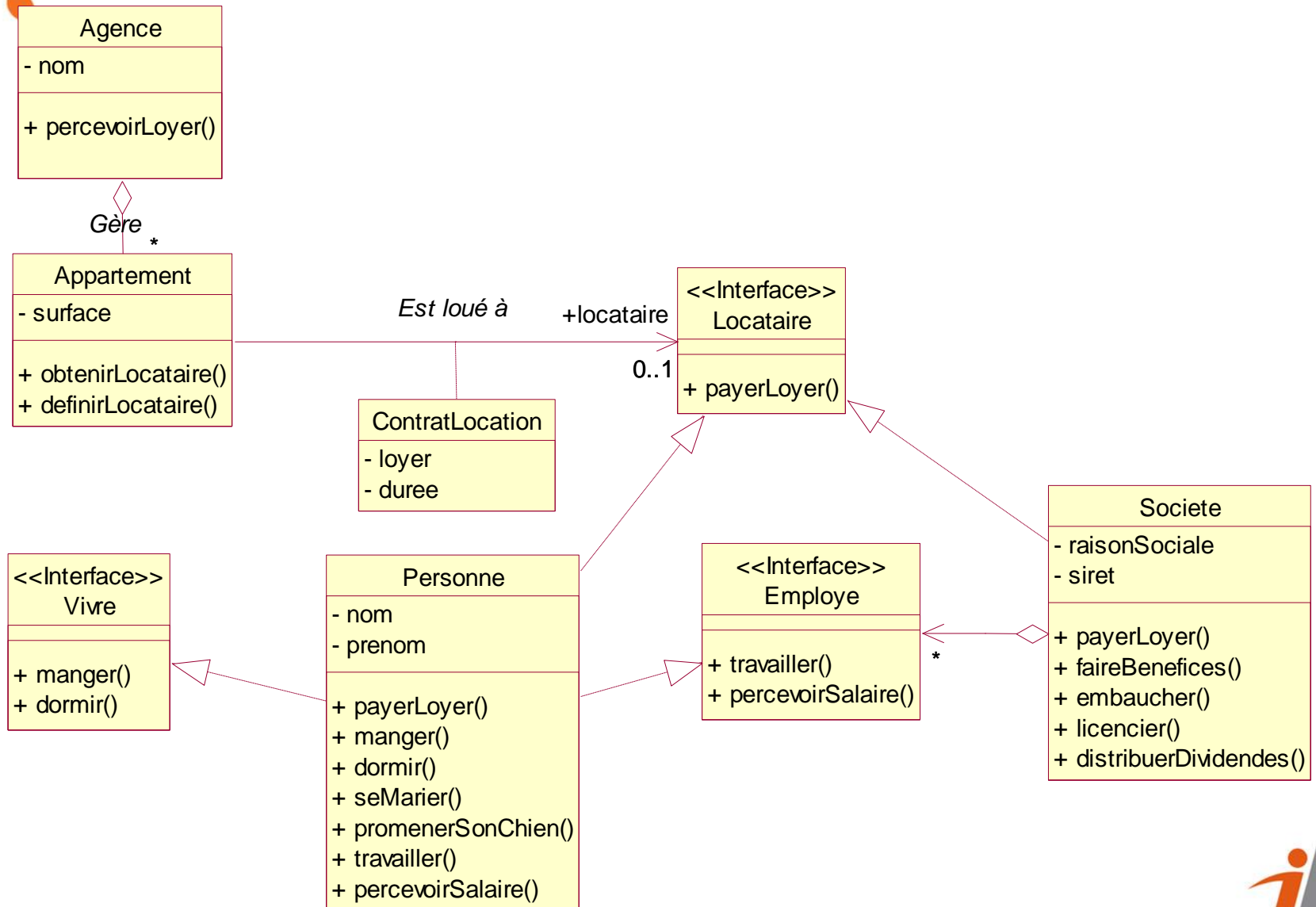




Les interfaces

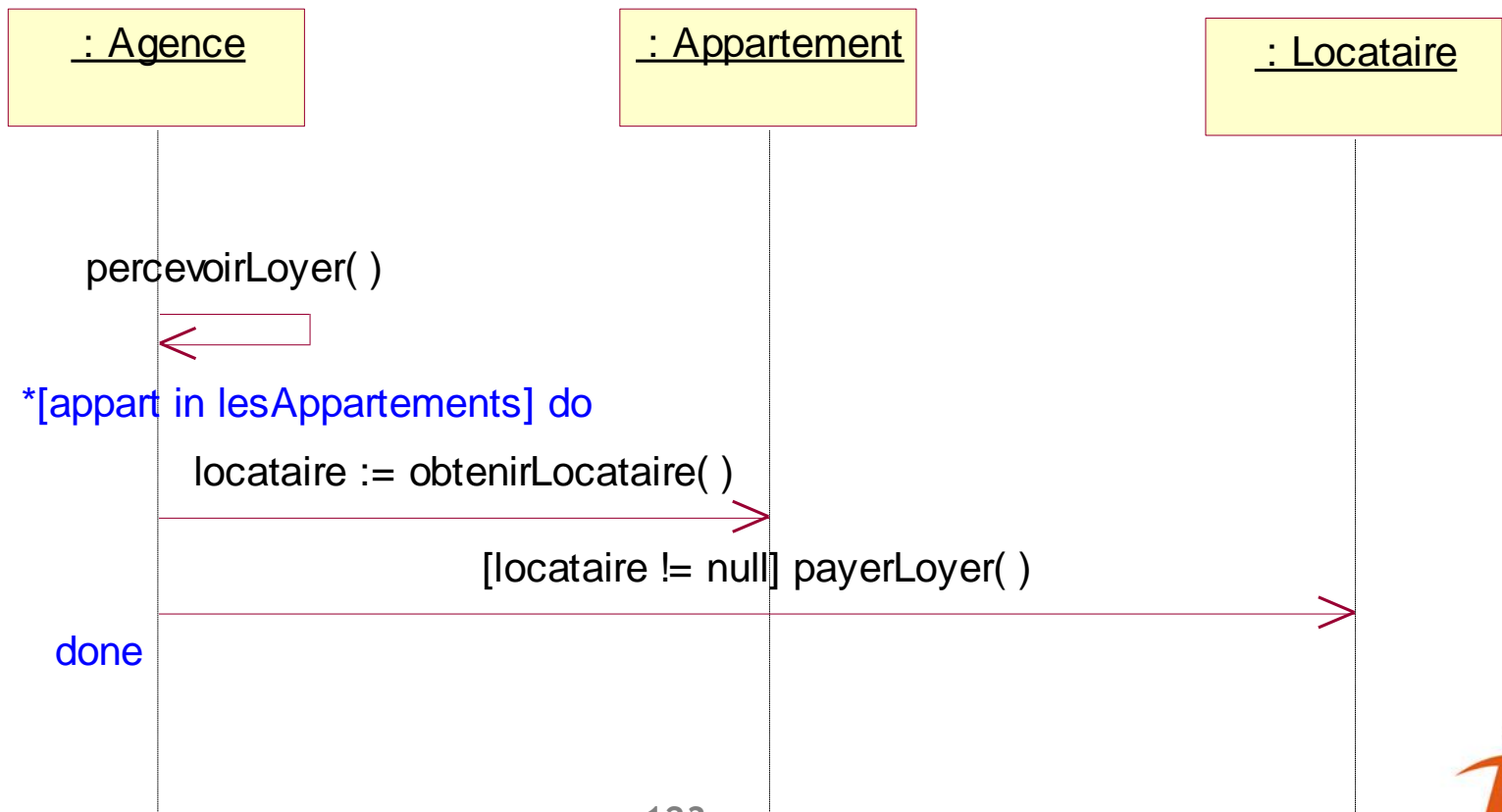
- Une interface est un contrat.
- Lorsqu'un objet client d'une interface collabore avec celle-ci, il ne connaît pas son type réel mais il sait comment le "manipuler" (quels messages il peut lui envoyer).
- Le but des interfaces est de diminuer le couplage entre deux classes.
 - L'interface permet de ne présenter au client "que ce qui l'intéresse".
- Un autre aspect est la protection des variations.
 - un objet client d'une interface peut collaborer avec n'importe quel type d'objet pourvu qu'il réalise l'interface.

Interface : exemple UML



Interface : mise en œuvre dynamique

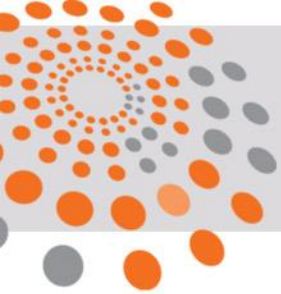
- L'objet Agence ne connaît pas la réalité du locataire (Personne ou Societe), ce qui ne l'empêche pas de le solliciter via l'interface Locataire ...





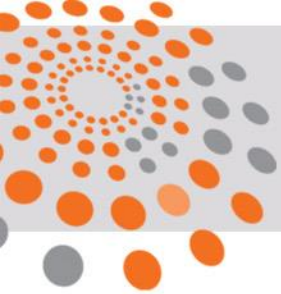
Les interfaces en Java





Les interfaces

- But des interfaces :
 - Diminuer le couplage entre classes.
 - L'interface permet de ne présenter au client « que ce qui l'intéresse »
 - Limiter les impacts sur des « Clients » lors de modifications des fournisseurs (ségrégation d'interfaces)
 - Se protéger contre des variations.
 - un objet client d'une interface peut collaborer avec n'importe quel type d'objet pourvu qu'il réalise l'interface.



Interface : exemple de mise en oeuvre

- On va définir la notion de (une interface) Musicien spécifiant les comportements que doit respecter tout musicien :
- Un guitariste étant un musicien, il devra donc respecter l'interface

```
public interface Musicien { public void jouerMusic ( ) ; }
```

```
class Guitariste implements Musicien {  
    public void jouerMusic ( ) {  
        System.out.println ( " jouer de la guitare " ) ;  
    }  
}
```

```
class Pianiste implements Musicien {  
    public void jouerMusic ( ) {  
        System.out.println ( " jouer du piano " ) ;  
    }  
}
```



Interface : exemple de mise en oeuvre

- Une classe cliente de l'interface « Musicien » :

```
import java.util.* ;  
class Orchestre  
{  
    private ArrayList<Musicien> listeMusicien = new ArrayList<Musicien> ();  
  
    public void ajouterMusicien (Musicien m )  
    { listeMusicien.add ( m ) ; }  
  
    public void jouer ( )  
    {  
        for ( int i=0 ; i <listeMusicien.size () ; i++ )  
            listeMusicien.get(i).jouerMusic () ;  
    }  
}
```



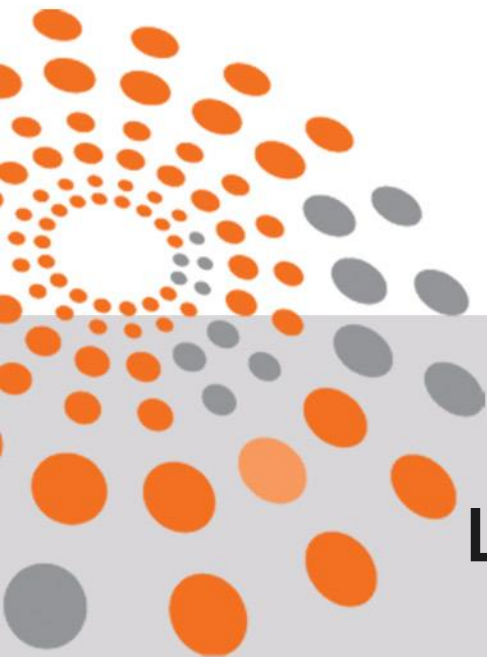
Héritage multiple

- L'héritage multiple n'est pas disponible en Java, mais une classe peut hériter d'une classe de base et implémenter une ou plusieurs interfaces :

```
class Pianiste extends Personne implements Musicien
{
    public void jouerMusic ( )
    {
        System.out.println ( " joue du Piano !" ) ;
    }
}
```

- Une classe implémentant plusieurs interfaces :

```
class Clown extends Personne implements Musicien, Acrobate, ...
```

Les packages



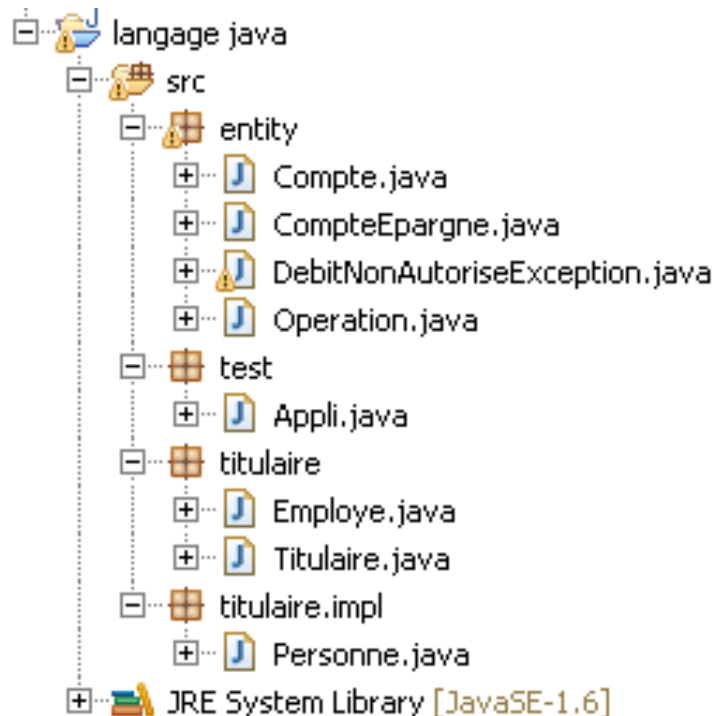


Les packages

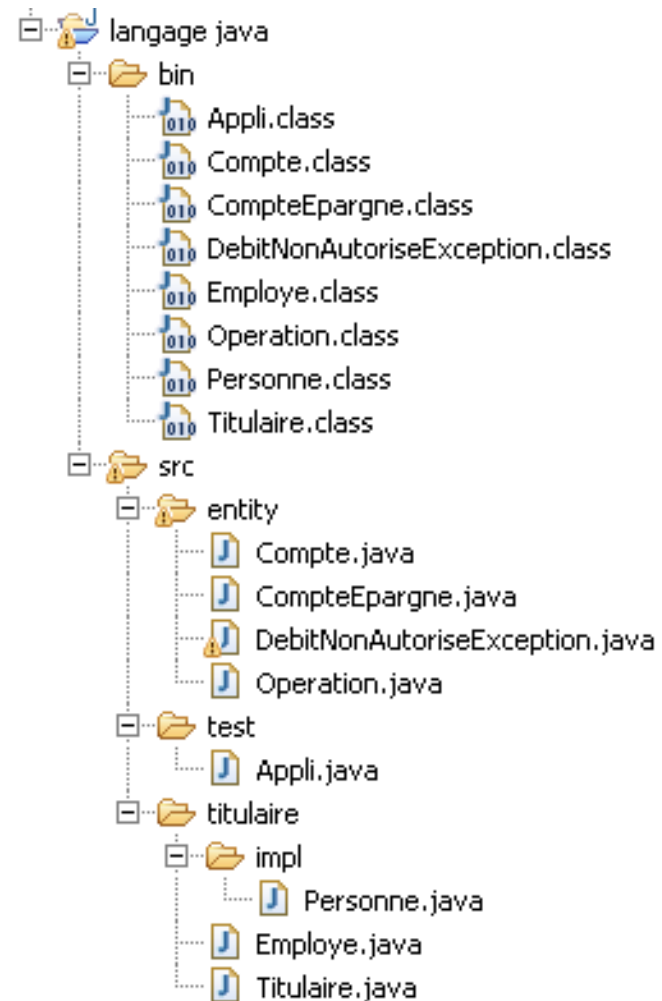
- Les packages permettent de regrouper des classes par affinité
 - Constituent l'organisation logique d'une application Java
 - Définissent des espaces de noms dans lesquels sont définies les classes
 - Constituent des entités d'encapsulation, de gestion de version, des tâches de développement, ...
- Du point de vue des sources Java, un package représente un répertoire
 - Le mot réservé « **package** » permet de définir le package d'une classe
 - Le mot réservé « **import** » permet de faire référence à une classe en dehors de son propre package

Exemple d'organisation en packages (vues Eclipse)

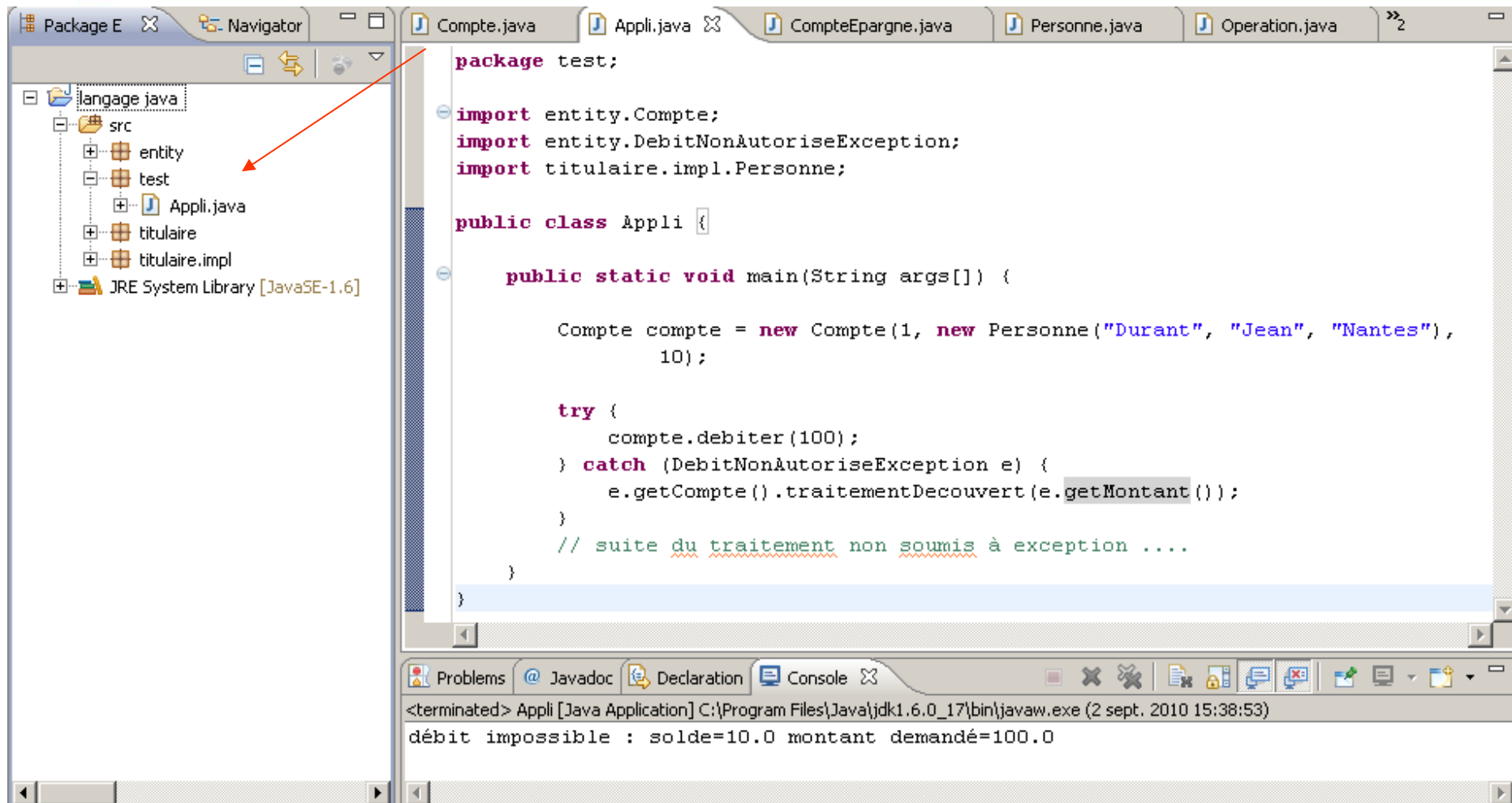
- Organisation logique



- Organisation physique



Les packages : mise en oeuvre



The screenshot shows an IDE with the following components:

- Package Explorer (left):** Displays the project structure. The 'src' folder contains 'entity', 'test', 'titulaire', and 'titulaire.impl'. The 'test' package is highlighted with a red arrow.
- Code Editor (center):** Shows the code for 'Appli.java'. The code is as follows:

```
package test;

import entity.Compte;
import entity.DebitNonAutoriseException;
import titulaire.impl.Personne;

public class Appli {

    public static void main(String args[]) {

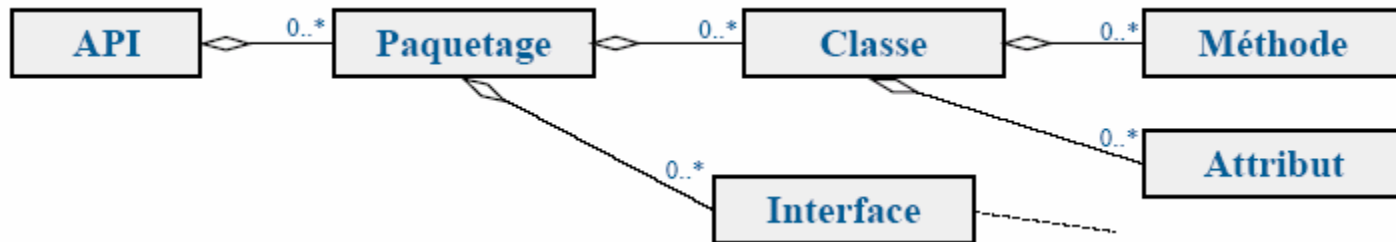
        Compte compte = new Compte(1, new Personne("Durant", "Jean", "Nantes"),
            10);

        try {
            compte.debiter(100);
        } catch (DebitNonAutoriseException e) {
            e.getCompte().traitementDecouvert(e.getMontant());
        }
        // suite du traitement non soumis à exception ....
    }
}
```
- Console (bottom):** Shows the output of the application:

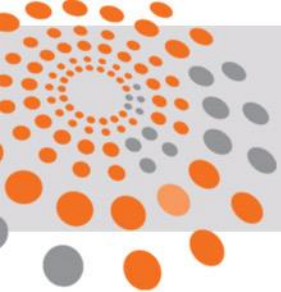
```
<terminated> Appli [Java Application] C:\Program Files\Java\jdk1.6.0_17\bin\javaw.exe (2 sept. 2010 15:38:53)
débit impossible : solde=10.0 montant demandé=100.0
```

Organisation des API standards

- Les API standards Java sont elles mêmes organisées en packages



- Quelques packages essentiels de la librairie standard (jdk1.8 (2014), 217 packages & 4240 classes)
 - java.applet : les applets sur le web
 - java.awt : interfaces graphiques, images et dessins
 - java.io : entrées / sorties
 - java.lang : chaînes de caractères, interaction avec l'OS, threads, ...
 - java.util : structures de données classiques
 - javax.swing : interfaces graphiques « light weight »
 - java.rmi : Remote Method Invocation
 - java.sql : fournit le package JDBC



JDK : Evolution

Java Development Kits	Codename	Release	Packages	Classes
Java SE 8 with JDK 1.8.0	---	2014	217	4,240
Java SE 7 with JDK 1.7.0	Dolphin	2011	209	4,024
Java SE 6 with JDK 1.6.0	Mustang	2006	203	3,793
Java 2 SE 5.0 with JDK 1.5.0	Tiger	2004	166	3,279
Java 2 SE with SDK 1.4.0	Merlin	2002	135	2,991
Java 2 SE with SDK 1.3	Kestrel	2000	76	1,842
Java 2 with SDK 1.2	Playground	1998	59	1,520
Development Kit 1.1	---	1997	23	504
Development Kit 1.0	Oak	1996	8	212

(source: [Java 8 Pocket Guide](#) book by Robert Liguori, Patricia Liguori)



Exemple : le package java.util

- Le package java.util propose des classes représentant les collections.
- Pour manipuler des collections il faut inclure le package java.util.

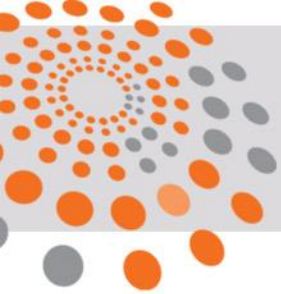
```
import java.util.* ;           // Utilisation de l'ensemble du package
import java.util.ArrayList;    // Utilisation uniquement de la class ArrayList
```

- util est un sous répertoire de java qui contient le pseudo code de l'ensemble des classes qu'il implémente :

ArrayList.class, HashMap.class,

- Le fichier ArrayList.java se présentant sous la forme suivante :

```
package java.util;
public class ArrayList { ... }
```



Les règles de nommages

- La création et l'utilisation de packages sont basées sur des règles de nommage strictes.
- Le système de nommage est basé sur l'organisation hiérarchique de ses packages.
- L'identificateur d'un package doit correspondre au répertoire ou à un chemin d'accès dans lequel se trouve la ou les classes.
- L'utilisation d'un package se fait en spécifiant le répertoire ou le chemin d'accès contenant la ou les classes que l'on veut importer.
- La partie interface d'un fichier source (class public) doit posséder le même identificateur que le fichier source.



Exemple

- Création d'un package pour la classe "Personne"

```
package ibformation.pers;  
public class Personne { ... }
```

- Le fichier `Personne.java` compilé va générer un fichier `Personne.class`.

- Ce fichier devra être mis dans le répertoire *pers* qui se trouve sous le répertoire *ibformation*.

```
import ibformation.pers.Personne;  
// ou import ibformation.pers.* ;
```

```
class Demo {  
    public static void main (String args[ ]) {  
        Personne pers1 = new Personne( "Bond" , "James" , 007 ) ;  
        pers1.print();  
        pers1.anniversaire();  
        pers1.print();  
    }  
}
```

Matérialisation des packages sous Eclipse

Java - IteratorExample.java - Eclipse SDK

File Edit Source Propager les modifications Navigate Search Project Exécuter Window Help

Vue Packages

- exosJava
 - autre
 - Revue.java
 - Revue
 - numero
 - titre
 - Revue(int, String)
 - compareTo(Object)
 - getNumero()
 - getTitre()
 - toString()
 - principal
 - CDate.java
 - CDate
 - IteratorExample.java
 - IteratorExample
- JRE System Library [jre1.5.0_02]

Navigateur

- exosJava
 - autre
 - Revue.class
 - Revue.java
 - principal
 - CDate.class
 - CDate.java
 - IteratorExample.class
 - IteratorExample.java
 - .classpath
 - .project

IteratorExample.java

```
package principal;

import java.util.*;

import autre.Revue;

class IteratorExample
{
    public static void main ( String args [ ] )
    {

        ArrayList c1 = new ArrayList ( ) ;
        for(int i =0; i<3; i++) c1.add( i, "chaine"+(i+1) );
        TreeSet c2 = new TreeSet();
        for(int i =0; i<3; i++) c2.add( "chaine"+(i+1) );

        IteratorExample.printCollection(c1);
        IteratorExample.printCollection(c2);

        TreeSet listeRevues = new TreeSet ( ) ; // les éléments

        listeRevues.add( new Revue ( 1 , "Geo" ) ) ;
        listeRevues.add( new Revue ( 2 , "Paris Match" ) )
        listeRevues.add( new Revue ( 3 , "France Football" ) )
        listeRevues.add( new Revue ( 4 , "Paris Match" ) );
        for ( Iterator i = listeRevues.iterator ( ) ; i.hasNext() ; i.next ( ) ) ;
        System.out.println ( i.next ( ) ) ;
    }
}
```

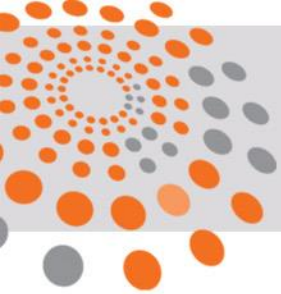
Console

```
<terminated> IteratorExample [Application Java] C:\Program Files\Java\jre1.5.0_02\bin\javaw.exe (23 nov. 200
chaine1
chaine2
```



Les collections

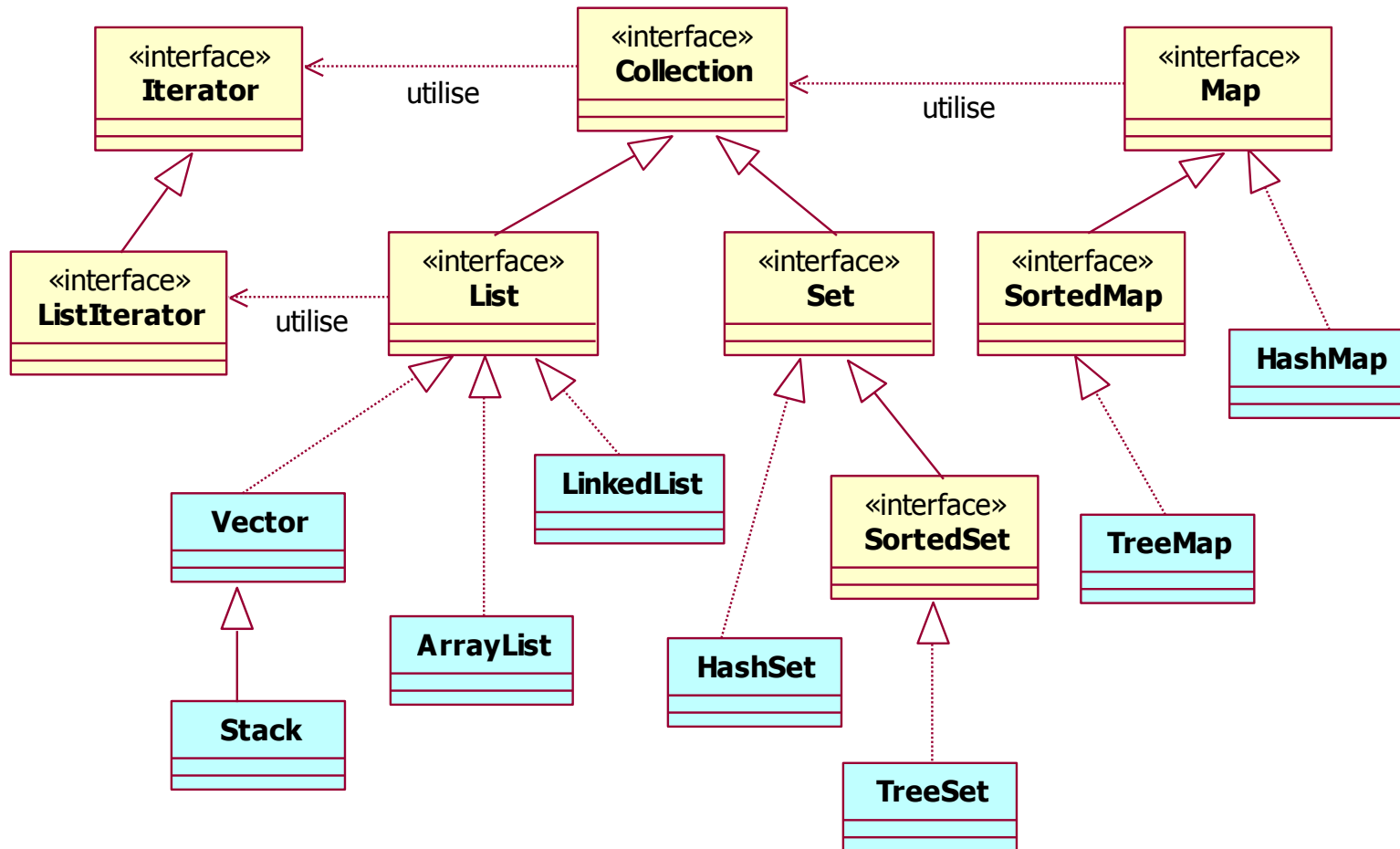




Les collections Java

- Une collection (parfois appelée *conteneur*) est un objet qui regroupe de multiples références d'objets.
- Un framework de collections est une architecture unifiée de représentation et de manipulation des collections.
- Le framework de collections java consiste en :
 - Des interfaces
 - Des collections concrètes (structures de données réutilisables).
 - Des algorithmes (fonctionnalités réutilisables)

Présentation du framework : diagramme UML





L'interface Collection

- L'interface collection est utilisée pour manipuler n'importe quelle collection concrète à partir d'une interface de services générale :

```
public interface Collection {  
    boolean add(Object o);  
    boolean addAll(Collection c);  
    void clear();  
    boolean contains(Object o);  
    boolean containsAll(Collection c);  
    boolean equals(Object o);  
    boolean isEmpty();  
    Iterator iterator();  
    boolean remove(Object o);  
    boolean removeAll(Collection c);  
    int size();  
    Object[ ] toArray();  
    ...  
}
```



L'interface List

- C'est une collection ordonnée (parfois appelée Séquence) qui peut contenir des éléments dupliqués (redondance).
- Elle vient compléter l'interface collection en ajoutant des méthodes spécifiques pour les listes.

```
public interface List extends Collection
{
    boolean add ( int index , Object o ) ; //décale vers la droite + MAJ indices
    Object remove ( int index ) ;
    Object get ( int index ) ;
    Object set ( int index , Object o ) ; //remplace
    public int indexOf ( Object o )
    public int lastIndexOf ( Object o )
    ListIterator listIterator() ;
    List subList ( int fromIndex , int toIndex ) ;
}
```



La classe ArrayList

- La classe ArrayList est implémentation la plus courante de l'interface List :

```
public class ArrayList extends AbstractList implements List,  
    RandomAccess, Cloneable, java.io.Serializable {  
    private transient Object elementData[] ;  
    private int size ;  
    public ArrayList ( int initialCapacity )  
    {    super();  
        if (initialCapacity < 0)  
            throw new IllegalArgumentException ( "Illegal Capacity: "+  
initialCapacity );  
        this.elementData = new Object[initialCapacity];  
    }  
    public ArrayList() { this ( 10 ) ; }  
    public boolean add(Object o) {  
        ensureCapacity(size + 1); // Increments modCount!!  
        elementData[size++] = o;  
        return true;  
    }  
}
```




Example

```
import java.util.* ;

class Demo
{
    public static void main (String args [ ])
    {
        List list = new ArrayList () ;
        list.add ( new Point ( 10 , 20 ) ) ;
        list.add ( new Point ( 20 , 40 ) ) ;
        list.add ( new Point ( 30 , 50 ) ) ;
        for ( Point p : list )
            p.print () ;
    }
}
```



Recherche d'un élément

- La méthode `contains ()` permet de rechercher si un élément est présent dans la collection :

```
import java.util.* ;
```

```
class Demo
```

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

```
{ List<Point> list = new ArrayList <Point> () ;
```

```
Point p1 = new Point ( 10 , 20 ) ;
```

```
list.add ( p1 ) ;
```

```
list.add ( new Point ( 20 , 40 ) ) ;
```

```
list.add ( new Point ( 30 , 50 ) ) ;
```

```
if ( list.contains ( p1 ) ) System.out.println ( "ok" ) ;
```

```
else System.out.println ( "non ok " ) ;
```

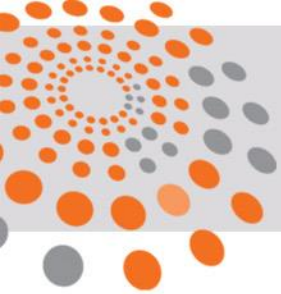
```
if ( list.contains ( new Point ( 20 , 40 ) ) )
```

```
System.out.println ( "ok" ) ;
```

```
else System.out.println ( "non ok " ) ;
```

```
}
```

```
}
```



Redéfinition de la méthode equals

- La classe Point redéfinit la méthode equals :

```
class Point
{
    ....
    public boolean equals ( Object obj )
    {
        if ( ! ( obj instanceof Point ) )
            return false ;
        return ( ( x == ( ( Point ) obj ).x ) && ( y == ( ( Point ) obj ).y ) ) ;
    }
}
```



Les itérateurs

- Un itérateur permet de parcourir une collection sans se soucier de la nature réelle du conteneur. Ainsi on développe des algorithmes de parcours constant alors que la nature du conteneur varie.
- Le parcours générique d'une collection s'effectue de la manière suivante :
 - appel de la méthode `iterator()` sur le conteneur pour renvoyer un `iterator`.
 - récupérer l'objet suivant dans la séquence grâce à sa méthode `next()`.
 - vérifier s'il reste encore d'autres objets dans la séquence via la méthode `hasNext()`.
- On a également la possibilité d'enlever le dernier élément renvoyé par l'itérateur avec la méthode `remove()`.



Example

```
import java.util.*;

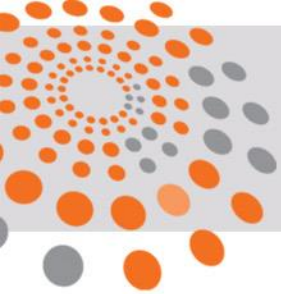
class IteratorExample {
    public static void main ( String args [ ] )
    {
        Collection<String> c1 = new ArrayList <String> ( ) ;
        for(int i =0; i<3; i++)
            c1.add( i, "chaine"+(i+1) );
        Collection c2 = new TreeSet();
        for(int i =0; i<3; i++)
            c2.add( "chaine"+(i+1) );
        IteratorExample.printCollection(c1);
        IteratorExample.printCollection(c2);
    }
    static public void printCollection(Collection<String> c) {
        for ( Iterator i = c.iterator ( ) ; i.hasNext ( ) ; )
            System.out.println ( i.next() ) ;
    }
}
```



L'interface Set

- Elle correspond à un groupe d'objets qui n'accepte pas 2 objets égaux au sens de « equals » (comme les ensembles en mathématique).
- L'interface Set n'ajoute aucune méthode à la classe Collection. Elle ne spécifie que les restrictions attendues sur les méthodes de base (unicité des objets).
- La méthode add n'ajoute pas l'élément si un élément égal (au sens du equals) est déjà dans l'ensemble (la méthode renvoie alors false)

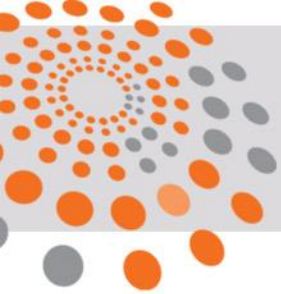
```
import java.util.* ;  
public class Demo {  
    public static void main ( String args[] ) {  
        Set s = new HashSet ( );  
        String tab[] = { "a" , "b" , "c" , "b" } ;  
        for ( int i = 0 ; i < tab.length ; i++ )  
            if ( ! s.add ( tab[i] ) )  
                System.out.println ( "duplication detectee : " + tab[i] );  
        System.out.println ( s.size() + " mots dans le set " + s );  
    }  
}
```



La classe TreeSet

- C'est un ensemble trié. Il garantit que les éléments sont rangés dans leur ordre naturel.
- Pour effectuer le tri, le TreeSet a besoin d'utiliser une méthode de comparaison.
- Une relation d'ordre sur des objets peut être définie en implémentant l'interface *Comparable*
- L'interface Comparable consiste en une seule opération :

```
public interface Comparable {  
    public int compareTo ( Object o ) ;  
}
```
- Les classes implémentant cette interface peuvent être triées automatiquement.

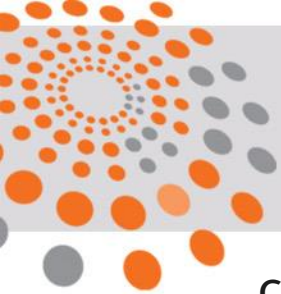


L'interface Comparator

- Une relation d'ordre naturelle n'est pas toujours suffisante :
 - Ordonner des objets à travers une autre relation d'ordre
 - Ordonner des objets qui n'implémentent pas l'interface *Comparable*
- Un comparateur (interface *Comparator*) peut être utilisé dans le cas où la relation d'ordre naturelle ne suffit pas :

```
public interface Comparator {  
    public int compare ( Object o1 , Object o2 ) ;  
}
```
- Les collections effectuant un tri sur leurs éléments peuvent être instanciées en fournissant un objet comparateur spécifique :

```
new TreeSet ( Comparator c ) ; // constructeur de TreeSet
```

Exemple d'implémentation de l'interface Comparable

```
class Revue implements Comparable
{
    private String titre ;
    private int numero ;

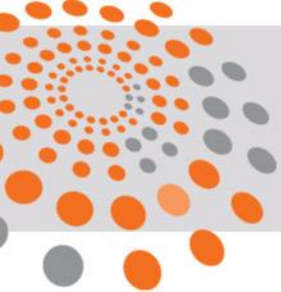
    public Revue ( int _numero , String _titre )
    { titre = _titre ; numero = _numero ; }

    public int getNumero () {    return numero ; }

    public String getTitre () {    return titre ; }

    public String toString ()
    { return titre + " " + numero ; }

    public int compareTo ( Object obj )
    { return titre.compareTo ( ((Revue ) obj ).titre); }
}
```



Exemple de mise en œuvre de l'interface Comparable via un TreeSet

```
import java.util.* ;

public class Demo
{
    public static void main ( String args[] )
    {
        Collection <Revue> listeRevues = new TreeSet <Revue> () ;
        // les éléments seront triés à partir de la relation d'ordre naturelle
        // basée sur la surcharge de l'interface Comparable

        listeRevues.add ( new Revue ( 1 , "Geo" ) ) ;
        listeRevues.add( new Revue ( 2 , "Paris Match" ) ) ;
        listeRevues.add( new Revue ( 3 , "France Football" ) ) ;
        listeRevues.add( new Revue ( 4 , "Paris Match" ) ) ;
        for ( Iterator i = listeRevues.iterator () ; i.hasNext () ; )
            System.out.println ( i.next () ) ; // utilisation de la méthode toString()
    }
}
```

L'interface Map

- L'interface Map correspond à un groupe de couples objets-clés.

```
public interface Map {  
    void clear()  
    boolean containsKey(Object clé)  
    boolean containsValue(Object valeur)  
    Set entrySet()  
    Object get(Object cle)  
    boolean isEmpty()  
    Set keySet()  
    Object put(Object clé, Object valeur)  
    void putAll(Map map)  
    void remove(Object key)  
    int size()  
    Collection values()  
};
```

retourne null si la
clé n'existe pas

- Une clé repère un et un seul objet. Deux clés ne peuvent être égales au sens de equals.

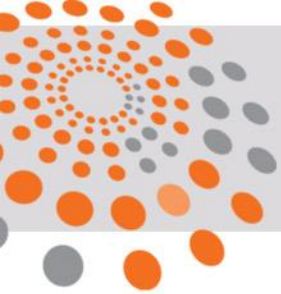


Les HashMap

- Les Classes qui implémentent l'interface Map sont :
 - **HashMap**, table de hachage garantissant un accès en temps constant
 - **TreeMap**, arbre ordonné suivant l'ordre naturel des éléments (interface Comparable) ou fonction d'un objet Comparator fourni au constructeur.

- Exemple

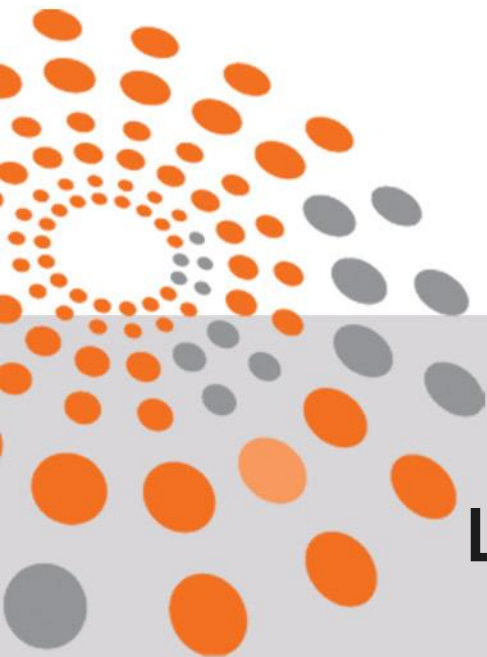
```
import java.util.* ;  
public class Demo {  
    public static void main ( String args[] )  
    {  
        Map map = new HashMap () ;  
        map.put( "Bond" , new Personne ( "Bond" , "James" , 45) ) ;  
        map.put( "Jones" , new Personne ( "Jones" , "Indiana" , 40 )) ;  
        (( Personne )map.get ( "Bond" )).afficher() ;  
    }  
}
```



Les utilitaires pour les collections : la classe Collections

- La classe Collections fournit un ensemble d'algorithmes, sous forme de méthodes statiques, permettant de travailler sur toute instance des classes implémentant l'interface Collection (tri, recherche, copie...) :

```
public class Demo
{
    public static void main ( String args[] )
    {
        String tabChaine [] = { "chaine3" , "chaine1" , "chaine2" } ;
        List listeChaine ;
        listeChaine = Arrays.asList ( tabChaine ) ;
        Collections.sort ( listeChaine ) ;
        for ( Iterator i = listeChaine.iterator () ; i.hasNext () ; )
            System.out.println (i.next());
    }
}
```



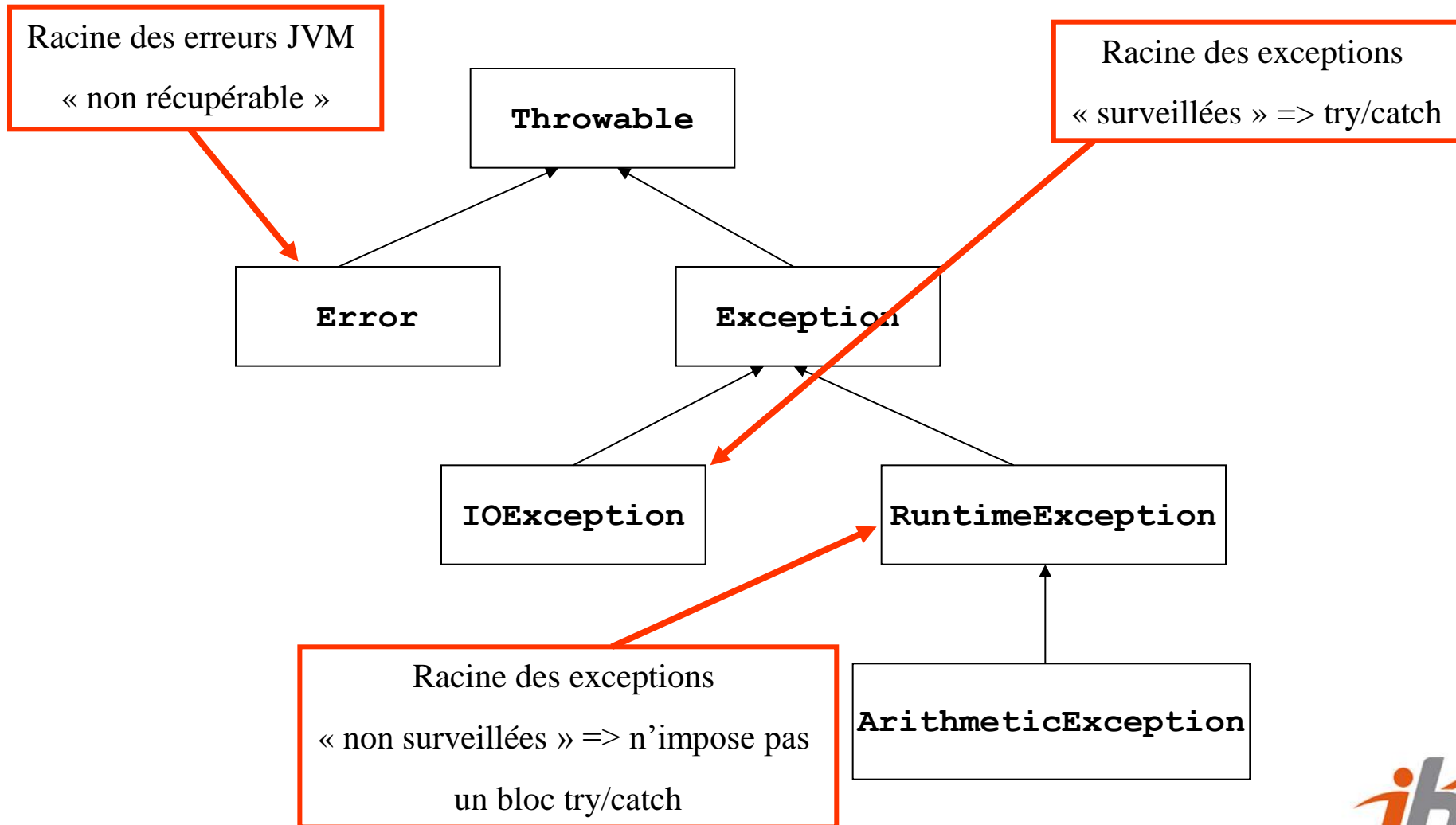
Les exceptions



Les exceptions

- Le mécanisme d'exception définit un standard pour la gestion des erreurs. Il permet de rendre la gestion des erreurs plus lisible et plus régulière.
- Lorsqu'une situation d'erreur est détectée, il faut :
 - abandonner l'action en cours,
 - signaler cette situation d'erreur,
 - exécuter les actions appropriées.
- Les erreurs peuvent être classées en deux catégories, les erreurs surveillées et les erreurs non surveillées.

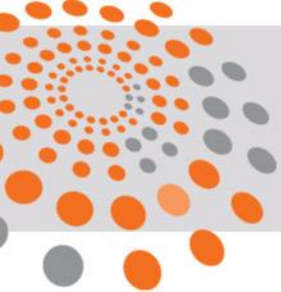
Hiérarchie standard des exceptions Java





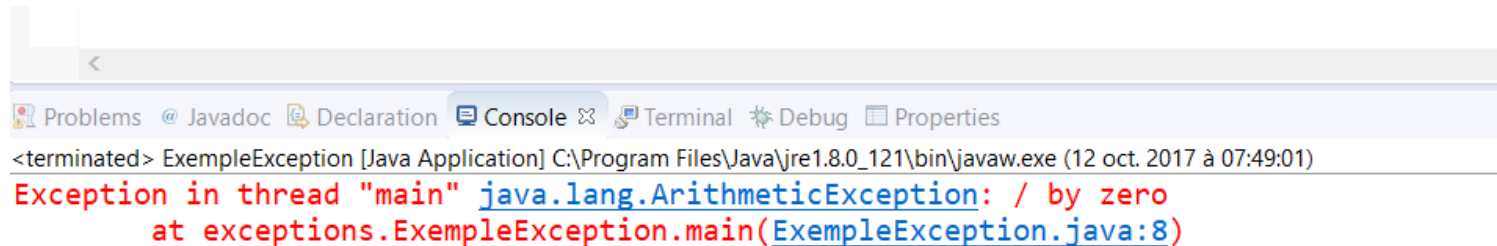
Quelques exceptions et erreurs standards en Java

- Exceptions « non surveillées » standards les plus rencontrées :
 - `java.lang.NullPointerException`
 - `java.lang.ClassCastException`
 - `java.lang.ArrayIndexOutOfBoundsException`
- Exceptions « surveillées » standards les plus rencontrées :
 - `java.sql.SQLException`
 - `Java.io.IOException`
 - `Java.lang.ClassNotFoundException`
- Les erreurs standards les plus courantes :
 - `java.lang.OutOfMemory`
 - `java.lang.StackOverflow`



Exemple de mise en œuvre d'une exception « non surveillée »

```
public class ExempleException {  
    public static void main(String[] args) {  
  
        int pay=8,payda=0;  
  
        double result=pay/payda;  
        System.out.println(result);  
    }  
}
```



The screenshot shows the 'Console' tab of an IDE. The title bar indicates the application is 'ExempleException [Java Application]' running at 'C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe' on '12 oct. 2017 à 07:49:01'. The console output shows an exception: 'Exception in thread "main" java.lang.ArithmeticException: / by zero' followed by 'at exceptions.ExempleException.main(ExempleException.java:8)'. The IDE interface includes tabs for Problems, Javadoc, Declaration, Console, Terminal, Debug, and Properties.

```
<terminated> ExempleException [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (12 oct. 2017 à 07:49:01)  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at exceptions.ExempleException.main(ExempleException.java:8)
```



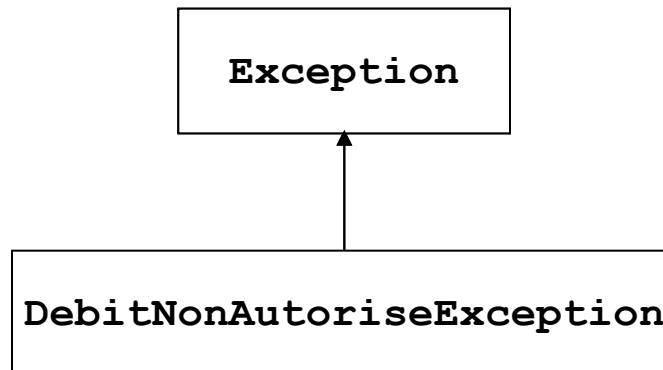
Exception « surveillée »

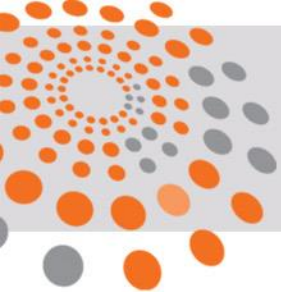
- L'appel de méthodes pouvant lever une ou des exceptions « surveillées » oblige l'appelant de les prendre en compte :
 - traiter l'exception,
 - la repropager au contexte appelant supérieur.
- Structure de code de prise en compte d'une exception

```
public void foo() {  
    try {  
        // traitement pouvant générer une Exception  
    } catch (Exception e) {  
        // traitement de l'exception  
    }  
    finally { // [optionnel]  
        // faire qq chose dans tous les cas ...  
    }  
    // suite du traitement non soumis à exception ...  
}
```

Exception « métier »

- Une exception « métier » caractérise le non respect d'une contrainte de gestion sur un objet « métier »
 - Exemple : le débit d'un montant supérieur au solde courant d'un compte est interdit
- Une exception « métier » est toujours de type « surveillée »
 - Définit par une classe d'exception spécifique,
 - Doit donner lieu à un traitement correctif adapté
- Exemple :





Lever une exception

- l'action throw (mot réservé) permet de lever une exception et de la propager à l'appelant.

```
class Voilier
{
    public void manœuvre () throws Exception
    {
        ...
        if ( probleme )
            throw new Exception ("bateau en détresse" );
        ...
    }
}
```

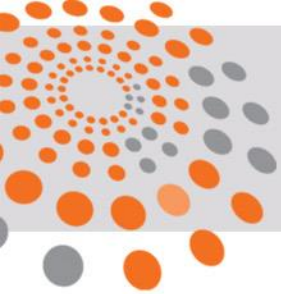


Attraper une exception

- Pour attraper une exception, il faut englober le code susceptible de la lancer dans un bloc try et traiter l'exception dans un bloc catch :

```
class Transat {  
    public static void main (String args[ ])  
    {  
        Voilier fujicolor = new Voilier();  
        try {  
            fujicolor.manoevre();  
        }  
        catch ( Exception s ) {  
            System.out.println (s.getMessage());  
        }  
    }  
}
```

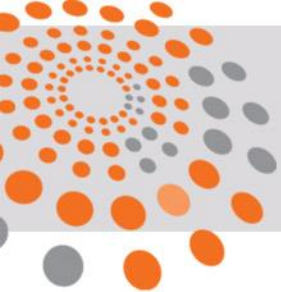
- Si aucune erreur n'intervient pendant l'exécution du bloc try, le contrôle est passé à l'instruction qui suit le dernier bloc catch.



Créer des exceptions métier

- Java offre la possibilité de créer des exceptions métier par héritage de la classe Exception :

```
class SosException extends Exception {  
    float longitude, latitude;  
    public SosException (float longitude, float latitude, String s)  
    { super(s); ... }  
}  
  
class Voilier  
{  
    public void manoeuvre () throws SosException  
    {  
        if ( probleme )  
            throw new SosException (50.34, 50.12, " bateau en détresse" );  
    }  
}
```



Spécification d'exceptions

- Lorsqu'on utilise une classe, il est fondamental de connaître les méthodes soumises à des exceptions.
- Le langage permet de spécifier les exceptions pouvant être générées lors de l'exécution des méthodes.
 - `void methode1() throws Exception1 { .. new Exception1(); ... }`
 - `void methode2() throws Exception1 , Exception2 { ... }`
- La spécification des exceptions énumère tous les types d'exceptions pouvant être levées durant l'exécution.



Traitement des exceptions

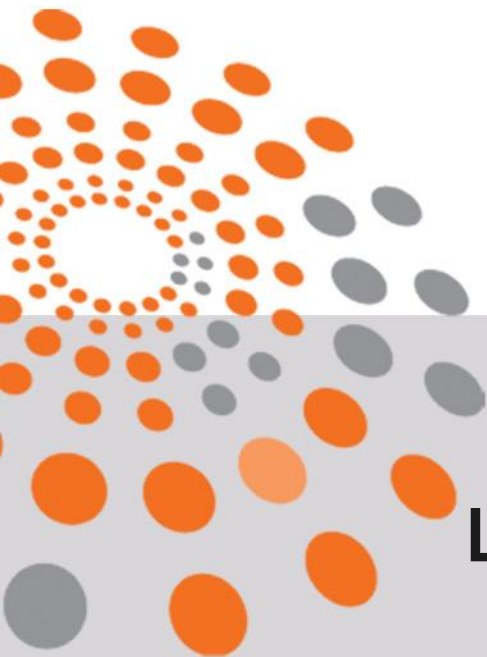
- Un bloc try est immédiatement suivi d'au moins un bloc catch.
- Le paramètre du catch indique le type d'exception traité.
- On peut définir autant de blocs catch que de types d'exceptions possibles.
- Les blocs catch sont parcourus dans leur ordre d'apparition jusqu'à concordance entre les types de leur argument et le type de l'objet exception généré.
- Un catch contrôle les exceptions de sa classe ainsi que celles de ses classes dérivées.
- Dans un bloc catch, Java offre la possibilité de relancer la dernière exception générée à l'aide du mot réservé throw.



Finaliser le traitement

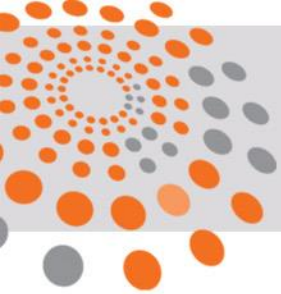
- Le langage met à disposition une instruction " finally " qui sera exécutée en sortie de bloc try, en toutes circonstances :
 - exécution du bloc try sans levée d'exception,
 - exécution du bloc try avec levée d'une exception

```
try {  
    // traitement  
}  
catch ( Exception e ) {  
    // traitement d'erreur  
}  
finally {  
    // traitement effectué en toutes circonstances, facultatif ...  
}
```



Les entrées-sorties





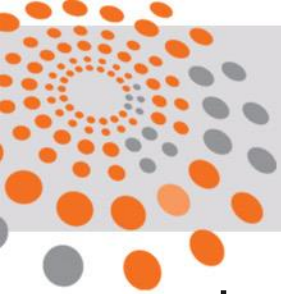
Les entrées-sorties

- Le package `java.io` fournit un ensemble de classes permettant de gérer les entrées-sorties :
 - les streams de caractères et streams binaires,
 - les streams à accès direct ou séquentiel,
 - les streams avec tampon de données.



Les entrées-sorties en mode binaire (octet)

- Les classes d'entrées-sorties en mode binaire héritent de 2 classes abstraites, `InputStream` pour les entrées et `OutputStream` pour les sorties.
- Les principales classes sont les suivantes :
 - **`FileInputStream`** , **`FileOutputStream`** : lecture ou écriture séquentiel de données dans un fichier,
 - **`BufferedInputStream`**, **`BufferedOutputStream`** : lecture ou écriture des données à l'aide d'un tampon,
 - **`PipedInputStream`**, **`PipedOutputStream`** : permet d'établir une connexion entre un stream d'entrée et un stream de sortie pour la communication inter thread
 - **`ObjectInputStream`**, **`ObjectOutputStream`** : lecture ou écriture de données représentant des objets (sérialisation)



Les entrées-sorties en mode caractère

- Les classes d'entrées-sorties en mode caractères héritent de 2 classes abstraites, **Reader** pour les entrées et **Writer** pour les sorties.
- Les principales classes sont les suivantes :
 - **FileReader, FileWriter** : lecture ou écriture séquentielle de caractères dans un fichier,
 - **BufferedReader, BufferedWriter** : lecture ou écriture de caractères à l'aide d'un tampon,
 - **PipedReader, PipedWriter** : permet d'établir une connexion entre un stream d'entrée et un stream de sortie,
 - **InputStreamReader, OutputStreamWriter** : permet la conversion d'un stream de données binaire en stream de caractères.



Exemple

```
import java.io.* ;
public class TFichier
{ public static void main ( String [ ] args )
  { char c = 0 ;
    FileReader entree = null; FileWriter sortie = null;
    try {
        entree= new FileReader ("ficSource");
        sortie = new FileWriter ("ficDesti");
        while ( ( c = ( char ) entree.read () ) != -1)
            sortie.write ( c ) ;
    } catch ( IOException e ) {
        System.out.println ( e.getMessage () ) ;
        e.printStackTrace () ;
    }
    finally {
        if(entree != null) entree.close() ;
        if(sortie != null) sortie.close();
    }
  }
}
```



Exemple

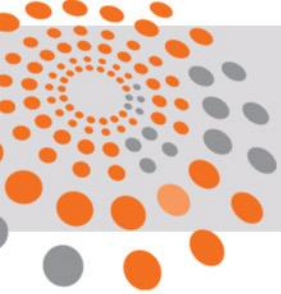
```
import java.io.* ;
class TFichier {
    public static void main ( String [ ] args ) {
        String str = null ;
        try {
            FileReader entree= new FileReader ("FichierSource.txt");
            BufferedReader in = new BufferedReader (entree);
            while ((str = in.readLine()) != null)
                System.out.println (str);
            in.close();
            entree.close();
        }
        catch ( IOException e ) {
            System.out.println ( e.getMessage () ) ;
            e.printStackTrace () ;
        }
    }
}
```




Accès direct

- La classe `RandomAccessFile` permet d'accéder directement a des données.

```
public class Demo {  
    public static void main (String args[ ]) {  
        try {  
            RandomAccessFile fic;  
            fic = new RandomAccessFile ("fic1.txt", "rw");  
            fic.seek ( 10 ) ;  
            while (fic.getFilePointer() < fic.length())  
                System.out.println (fic.readLine());  
            fic.close();  
        }  
        catch ( IOException e ) {  
            System.out.println ( e.getMessage () ) ;  
            e.printStackTrace () ;  
        }  
    }  
}
```



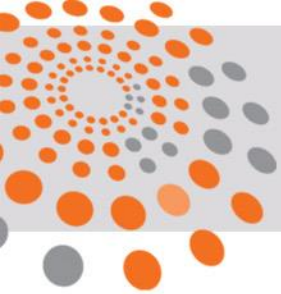
La sérialisation

- Java met a disposition des streams permettant d'enregistrer des objets sur disque. Cette opération (sérialisation) permet de rendre les objets persistants.
- Un objet voulant être sérialisé doit implémenter l'interface Serializable. Cette interface ne possède aucune méthode, elle est utilisée en tant que marqueur.
- Lorsque l'objet est sauvegardé, tous les objets auxquels il fait référence le sont aussi (ils doivent donc implémenter eux aussi l'interface Serializable).
- Si l'objet hérite d'une autre classe qui elle n'est pas sérialisable, il faut que cette classe aie un constructeur qui ne prend aucun paramètre.



Exemple

```
import java.io.* ;  
class Voiture implements Serializable  
{  
    private int nbrKlm;  
    private int carburant;  
    private Moteur moteur;  
    public Voiture (String m ) { moteur = new Moteur(m); }  
    public Moteur getMoteur() { return moteur ; }  
    public int getCarburant() { return essence; }  
    public getNbrKlm () { return nbrKlm ; }  
    public void setCarburant(int e) { essence += e; }  
    public void setNbrKlm ( int n ) { nbrKlm = n ; }  
}
```



Exemple

```
import java.io.* ;  
class Moteur implements Serializable  
{  
    private int puissance ;  
    private String marque ;  
    private static int nbInstance; // compteur de voitures ....  
    public Moteur (String _marque , int _puissance )  
    { marque = _marque ; puissance = _puissance ; }  
    public String getMarque () { return marque ; }  
    public int getPuissance { return puissance ; }  
    ...  
}
```

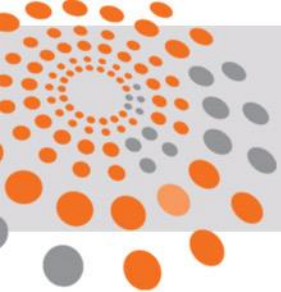
- Une exception **NotSerializableException** est générée lorsque l'on veut sérialiser un objet issue d'une classe qui n'implémente pas Serializable.



La sauvegarde d'objet

- La sérialisation d'un objet est effectuée lors de l'appel de la méthode `writeObject()` sur un objet implémentant l'interface `ObjectOutput` (`ObjectOutputStream`) :

```
void writeObject ( Object obj );
```
- Par défaut tous les champs de l'objet sont sauvegardés.
- Les attributs statiques ne sont pas sérialisés
- Chaque fois qu'un objet est sauvé dans un flux, un objet handle unique pour ce flux est également sauvé. Ce handle est attaché à l'objet dans une table de hashage.
- Lors d'une même sérialisation globale, chaque fois que l'on demande de sauver un objet déjà présent dans le flux, seul le handle est sauvé. Ceci permet de casser les circularités.



La restitution d'objet

- La désérialisation d'un objet est effectuée lors de l'appel de la méthode `readObject()` sur un objet implémentant l'interface `ObjectInput` (`ObjectInputStream`):
`Object readObject () ;`
- L'objet récupéré est une copie de l'objet sauvé.



Exemple

```
import java.io.*;
public class TSerialisation {
    public static void main (String[] args) throws Exception {
        Voiture voiture = new Voiture("V6");
        voiture.setCarburant(50); voiture.setNbrKlm (3000);
        FileOutputStream ficSortie = new FileOutputStream ("garage.ser");
        ObjectOutputStream oSortie = new ObjectOutputStream (ficSortie);
        oSortie.writeObject (voiture);
        oSortie.close();
        voiture = null; System.gc();
        FileInputStream ficEntree = new FileInputStream ("garage.ser");
        ObjectInputStream oEntree = new ObjectInputStream ( ficEntree );
        voiture = ( Voiture ) oEntree.readObject ();
        oEntree.close();
        System.out.println (voiture .getMoteur ().getMarque() +
                            voiture .getCarburant());
    }
}
```



La sérialisation : compléments

- Il est possible d'empêcher la sauvegarde de certaines données à l'aide du mot réservé transient :

```
transient int carburant
```

- La redéfinition de la méthode writeObject dans sa classe permet de sérialiser les attributs statics qui ne le sont pas implicitement :

```
private void writeObject ( ObjectOutputStream s ) throws IOException {  
    s.defaultWriteObject() ;  
    s.writeInt(nblInstance); // nblInstance est un attribut statique  
}
```

- Il est possible de définir son propre traitement de restitution en redéfinissant la méthode readObject dans sa classe :

```
private void readObject ( ObjectInputStream s ) throws IOException {  
    s.defaultReadObject();  
    nblInstance = s.readInt (); // nblInstance est un attribut int static ...  
}
```




La gestion de version

- Un objet ne peut être restitué que s'il a été sauvegardé avec la même version de l'application.
- Java propose un mécanisme permettant de "Identifier" le problème de la lecture d'un objet d'une classe qui a évolué depuis que l'objet a été sauvegardé.
- Il faut dans ce cas identifier la version de la classe:
`static final long serialVersionUID = valeurSUID`
- Une valeur est assignée automatiquement au SUID lorsque la classe est sérialisée.
- L'utilitaire `serialver` du jdk permet de récupérer le SUID.



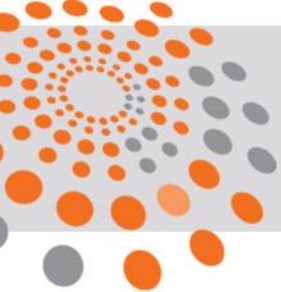
API complémentaires : JDBC, AWT et Swing





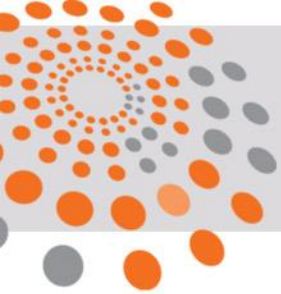
Présentation JDBC





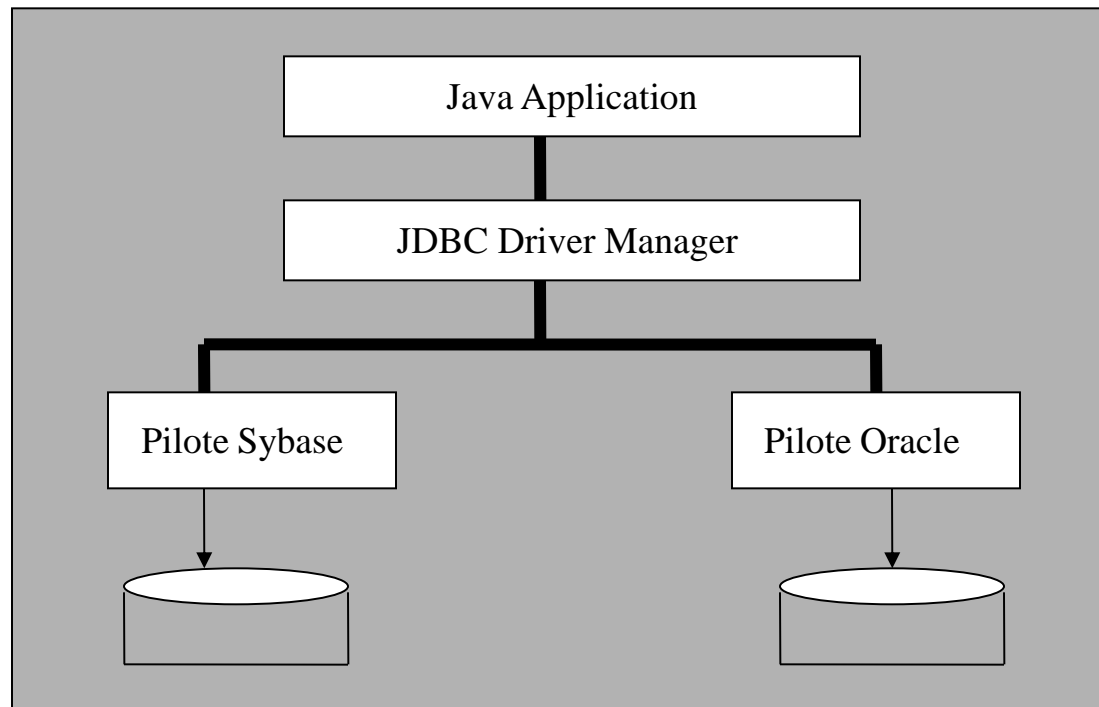
Présentation JDBC

- Acronyme : Java Data Base Connectivity
- Permet aux applications Java de communiquer avec les gestionnaires de bases de données.
- Ensemble de classes et d'interfaces permettant de réaliser des connexions vers des bases de données et d'effectuer des requêtes.
- C'est une API de bas niveau. Elle permet d'exécuter des instructions SQL et de récupérer les résultats.

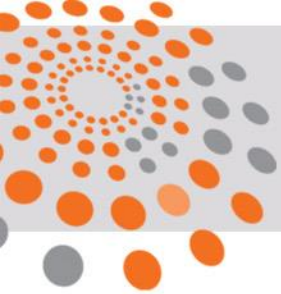


Principe de JDBC

- Toute application JDBC s'exécute sans jamais référencer l'implémentation spécifique de JDBC.

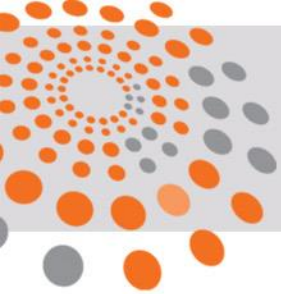


- L'accès à une base n'est possible que si la base fournit un pilote JDBC.



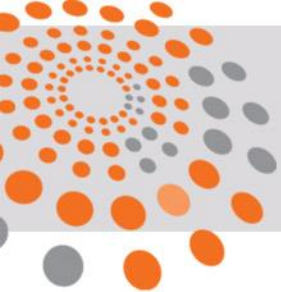
Les pilotes JDBC

- Un pilote JDBC doit fournir une implémentation de l'interface Driver.
- 4 types de pilotes :
 - **type 1** : pont entre JDBC et ODBC
 - emploi de bibliothèques natives spécifiques au SGBD
 - **type 2** : wrapper java de bibliothèques C/C++
 - emploi de bibliothèques natives spécifiques au SGBD
 - **type 3** : pilote tout java basé sur un protocole réseau à vocation universelle.
 - Nécessite un middleware pour convertir les requêtes dans le protocole SGBD requis (modèle 3 couches)
 - **type 4** : pilote tout java interagissant directement avec le SGBD
 - Utilise directement le protocole réseau du SGBD, spécifique à chaque fournisseur



Structure d'une application JDBC

- Un programme utilisant JDBC fonctionne généralement de la façon suivante :
 - Chargement du pilote ;
 - Établissement d'une connexion avec la base ;
 - Envoi de requêtes ;
 - Utilisation des données obtenues ;
 - Mise à jour des informations ;
 - Fin de connexion.

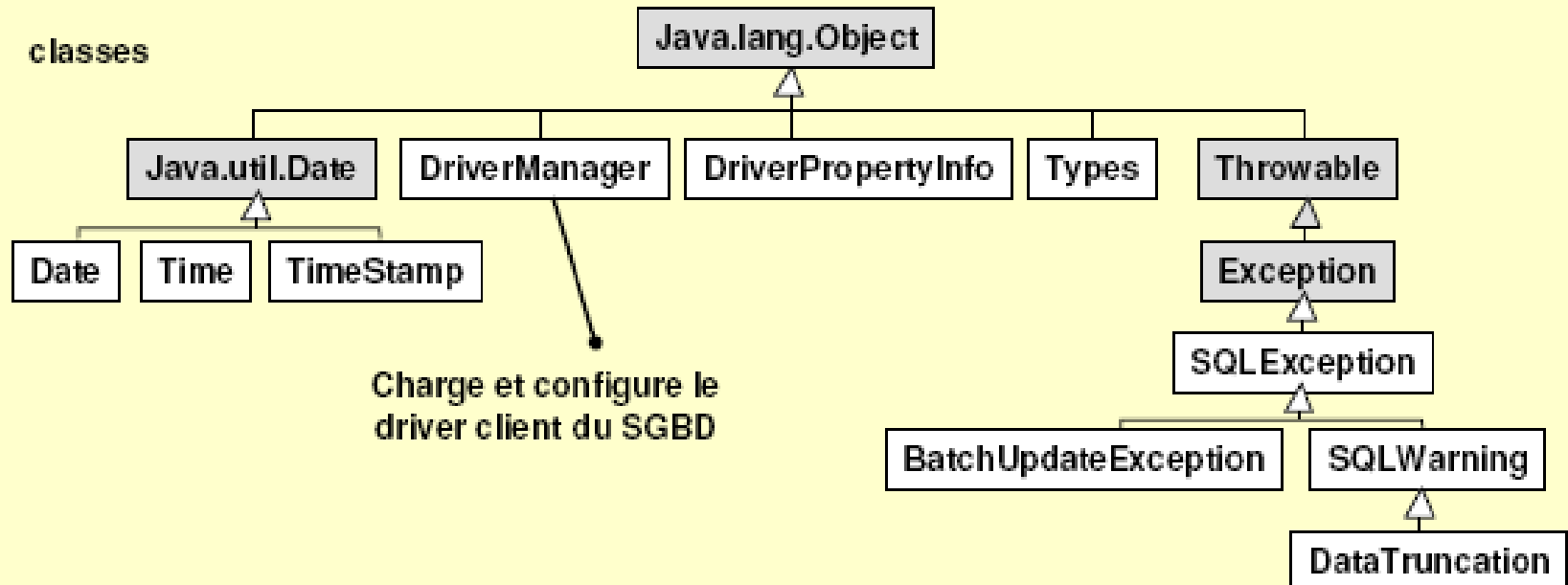


L'API JDBC

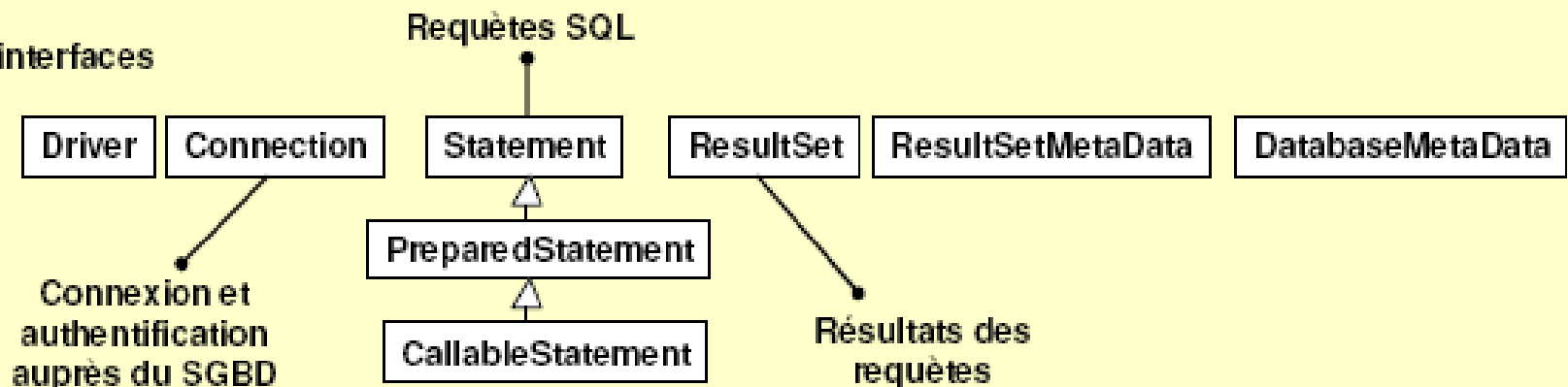
- Elle se présente sous forme d'un ensemble de classes et d'interfaces :
 - ***java.sql.DriverManager*** : prend en charge le chargement des pilotes et permet de créer de nouvelles connexions à des bases de données ;
 - ***java.sql.Connection***: représente une connexion à une base de données particulière ;
 - ***java.sql.Statement***: qui joue le rôle d'un réceptacle pour exécuter les ordres SQL à travers une connexion donnée ;
 - ***java.sql.ResultSet***: contrôle l'accès aux colonnes de résultat d'un objet Statement donné.

L'API JDBC (suite)

classes



interfaces

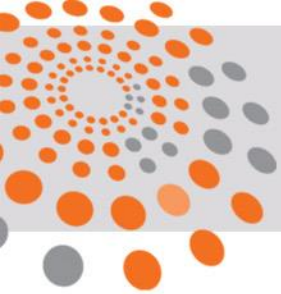




Chargement du pilote

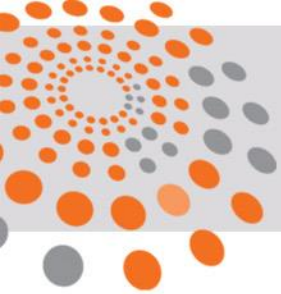
- L'utilisation de JDBC nécessite de charger le pilote de la base de données
 - Il faut connaître son "qualified name"
 - On utilise ensuite la méthode *Class.forName(...)*
- Exemple :

```
Class.forName("org.postgresql.Driver");
```
- Lors du chargement de la classe du pilote, celui-ci s'enregistre tout seul auprès de la classe **DriverManager** afin d'être disponible.
- Le driver manager maintient une liste des pilotes identifiables par leur « qualified name »



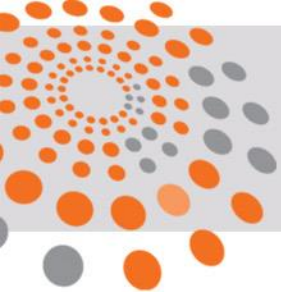
URL de connexion à une base de données

- Pour établir une **connexion** avec une base, il faut connaître son nom et lui associer un **pilote**.
- La convention de nommage retenue par JDBC est dérivée de celle utilisée pour les URL sur internet :
 - *protocole:<sousprotocole>:<complément>*
- Le protocole **JDBC** est utilisé pour distinguer les URL JDBC des autres types d'URL.
- Le **sous protocole** permet de définir le type du pilote.
 - Pas de standardisation, mais une centralisation des noms par JavaSoft afin de garantir l'unicité.



URL de connexion à une base de données (suite)

- La partie complément est utilisée pour identifier la source de données et éventuellement des éléments de configuration :
- Une **URL JDBC** identifie donc une base de données individuelle d'une façon unique spécifique au pilote
- Exemple avec un pilote Java MySQL :
`Class.forName("com.mysql.jdbc.Driver");`
- URL dépend fortement de la BD (local ou distant)



Connexion à une base de données

- L'interface ***Connection*** représente une session sur une base de données.
- Elle permet d'accéder à des méta-informations sur les tables.
- Un objet ***Connection*** sert à créer les objets permettant d'effectuer des requêtes.
- L'ouverture d'une connexion se fait par la méthode statique ***getConnection*** de la classe ***DriverManager*** :

```
Connection cx = DriverManager.getConnection(url, uid, pass);
```

- Exemple URL avec MySQL :

"jdbc:mysql://localhost:PortMySQL/NomDeVotreBaseDonnees"

- Le ***DriverManager*** va tenter de localiser le pilote correspondant à celui de L'URL en consultant sa liste et lancera une connexion sur la base en utilisant le nom d'utilisateur et le mot de passe indiqué.



Exemple

```
String driver = "com.mysql.jdbc.Driver";
String url = "jdbc:mysql://localhost/formation";
String uid = "user"; String passwd = "passwd";
Connection cx = null;

try {
    Class.forName(driver);
    cx = DriverManager.getConnection(url, uid, passwd);
    ... // faire qqChose avec la connexion ...
} catch (ClassNotFoundException e) {
    // classe du pilote introuvable
} catch (SQLException e) {
    // accès à la base refusé
} finally {
    try { if (cx != null) cx.close(); }
    catch (SQLException e) { e.printStackTrace(); }
}
```



Libération des connexions

- L'interface **Connection** expose la méthode **close()** pour fermer les connexions.
- Les appels aux méthodes **close()** doivent s'effectuer dans un bloc **finally** :

```
finally {  
    try { if (cx != null) cx.close(); }  
    catch (SQLException e) { e.printStackTrace(); }  
}
```



Rappel SQL





Requête SELECT

- Acronyme : Structured Query Language
- Le langage SQL propose un ensemble d'instructions permettant le dialogue avec une base de données.
- L'instruction ***SELECT*** permet la sélection de colonnes en fonctions de critères :

SELECT nomColonne1, ..., nomColonneN

FROM nomTable

WHERE nomColonne = valeur



Requêtes INSERT et DELETE

- L'instruction **INSERT** permet l'insertion de lignes dans une table :
INSERT INTO nomTable (nomColonne1, ..., nomColonneN)
VALUES (valeur1,...,valeurN)
- L'instruction **DELETE** permet la suppression de lignes dans une table :
DELETE FROM nomTable
WHERE nomColonne = valeur



Requêtes UPDATE et CREATE

- L'instruction **UPDATE** permet la modification de données :

UPDATE nomTable

SET nomColonne1 = valeur1, ... nomColonneN=valeurN

WHERE nomColonne = valeur

- L'instruction **CREATE** permet la création d'une table :

CREATE TABLE nomTable (

nomColonne typeColonne modificateurColonne,

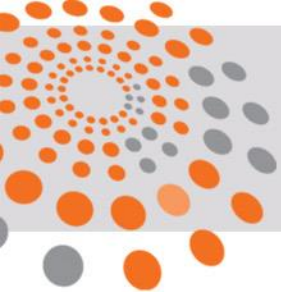
...

nomColonne typeColonne modificateurColonne)



Exécution de requêtes





L'interface Statement

- L'exécution d'une requête s'effectue à l'aide de l'interface **Statement**. Un objet Statement symbolise une requête SQL.
- Un objet **Statement** est créé par l'appel de la méthode **createStatement()** sur un objet Connection :

```
Statement stmt = cx.createStatement();
```

- Exemple :

```
String driver = "com.mysql.jdbc.Driver";  
String url = "jdbc:mysql://localhost/formation";  
String uid = "user"; String passwd = "passwd";  
Connection cx = null;  
Statement stmt = null;  
...
```



Exemple (suite)

```
try {
    Class.forName(driver);
    cx = DriverManager.getConnection(url, uid, passwd);
    stmt = cx.createStatement();
    ... // faire qqChose avec le statement ...
} catch (ClassNotFoundException e) {
    // classe du pilote introuvable
} catch (SQLException e) {
    // accès à la base refusé
} finally {
    try {
        if (stmt != null) stmt.close();
        if (cx != null) cx.close();
    }
    catch (SQLException e) { e.printStackTrace(); }
}
```



Exécution de requêtes

- L'interface ***Statement*** expose un ensemble de méthodes permettant l'exécution de requêtes SQL :

ResultSet executeQuery(String sql) *throws* SQLException

int executeUpdate (String sql) *throws* SQLException

boolean execute(String sql) *throws* SQLException

- La méthode ***executeQuery*** exécute une requête SQL restituant un ResultSet (typiquement un SELECT)
- La méthode ***executeUpdate*** renvoie le nombre de lignes affectées lors de la modification d'une table (typiquement un INSERT, un UPDATE, un DELETE) .
- La méthode ***execute*** permet l'exécution de requêtes dont on a pas de certitude quant à son type.



Exemple d'exécution de requête

```
public static void ouvrirCompte(int numero, float solde, String proprio) {
    try {
        Class.forName(driver);
        cx = DriverManager.getConnection(url, uid, pass);
        stmt = cx.createStatement();
        String requete = "INSERT INTO comptes VALUES " +
            "(" + numero + ", " + solde + ", " + proprio + ")";
        int nb = stmt.executeUpdate(requete);
        System.out.println("nb lignes mises a jour=" + nb);
    }
    catch (SQLException e) {
        // loguer l'erreur ...
        // throw e;
        throw new RuntimeException(e); }
    finally {
        try {
            if(stmt != null) stmt.close();
            if(cx != null) cx.close();
        }
        catch (Exception e) { throw new RuntimeException(e); }
    }
}
```




Exploitation des ResultSet

- Une requête de type ***SELECT*** (appel de la méthode `executeQuery`) retourne un ensemble de tuples (les lignes), sous la forme d'un objet de type ***ResultSet***.
- Un ***ResultSet*** gère en interne un curseur permettant d'accéder aux lignes du résultat.
- La méthode ***next()*** permet de gérer la progression du curseur dans le `ResultSet` :
`boolean next() throws SQLException`
- Elle renvoie vrai tant qu'il reste des lignes à consommer.



Exploitation des ResultSet (suite)

- Les valeurs des différentes colonnes sont obtenues à l'aide des méthodes "**getXxx**" où Xxx caractérise le type de la donnée souhaitée :

`xxx getXxx(int rang) // accès par indexation`

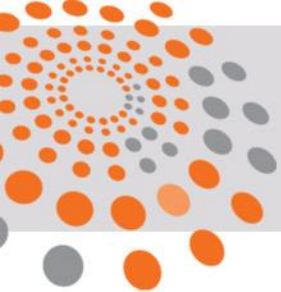
`xxx getXxx(String nom) // accès par nom`

- Les méthodes **getXxx(...)** convertissent les valeurs des types SQL en des valeurs de types Java.

- Exemple :

`int getInt(...)`

`String getString(...)`



Principales méthodes getXxx d'un ResultSet

boolean getBoolean(**int**); **boolean** getBoolean(String);

byte getByte(**int**); **byte** getByte(String);

Date getDate(**int**); Date getDate(String);

double getDouble(**int**); **double** getDouble(String);

float getFloat(**int**); **float** getFloat(String);

int getInt(**int**); **int** getInt(String);

long getLong(**int**); **long** getLong(String);

short getShort(**int**); **short** getShort(String);

String getString(**int**); String getString(String);



Exemple d'exploitation d'un ResultSet

```
public static List<Compte> getComptes () {  
    List<Compte> lesComptes = new ArrayList<Compte>();  
    String requete = "SELECT * FROM compte";  
    try {  
        Class.forName(driver);  
        cx = DriverManager.getConnection(url, uid, pass);  
        stmt = cx.createStatement();  
        ResultSet rs = stmt.executeQuery(requete);  
        while (rs.next()) {  
            int numero = rs.getInt("numero");  
            String proprio = rs.getString("proprio");  
            float solde = rs.getFloat("solde");  
            lesComptes.add(new Compte(numero, proprio, solde));  
        }  
    } catch (Exception e) { throw new RuntimeException(e); }  
    finally { ... }  
    return lesComptes;  
}
```



Aspects avancés de JDBC





Retour sur les ResultSet

- Un ResultSet permet aussi de modifier ou supprimer des lignes d'une table
- Pour cette utilisation, Il faut configurer le ResultSet par le biais de la méthode createStatement :

```
Statement createStatement(int resultSetType,  
                           int resultSetConcurrency)  
throws SQLException
```

- Les paramètres du ResultSet sont :

```
TYPE_FORWARD_ONLY (par défaut) //non scrollable et non sensible  
TYPE_SCROLL_INSENSITIVE //scrollable et non sensible aux autres  
TYPE_SCROLL_SENSITIVE //scrollable sensible aux changements  
CONCUR_READ_ONLY //ResultSet non modifiable  
CONCUR_UPDATABLE //ResultSet modifiable
```



Update à l'aide d'un ResultSet

- Modification d'un ResultSet :
 - Le ResultSet doit être créé avec la constante **CONCUR_UPDATABLE**
 - Les modification des colonnes s'effectuent à l'aide des méthodes **updateXxx()** :
 - **updateDate(int, Date)** / **updateDate(String, Date)**
 - **updateFloat(int, float)** / **updateFloat(String, float)**
 - **updateInt(int, int)** / **updateInt(String name, int x)**
 - **updateString(int, String)** / **updateString(String, String)**
 - ...
- Les modifications n'affectent que la ligne courante du ResultSet. Il faut faire appel à la méthode **updateRow()** pour synchroniser la base de données
- On peut également supprimer la ligne courante d'un ResultSet avec la méthode **deleteRow()**.



Exemple

```
public void creditorInteretsCompteEpargne() {
    try {
        Class.forName(driver);
        cx = DriverManager.getConnection(url, uid, pass);
        stmt = cx.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
        ResultSet rs = stmt.executeQuery(
            "SELECT * FROM compteEpargne");
        while (rs.next()) {
            float solde = rs.getFloat("solde");
            float taux = rs.getFloat("taux");
            rs.updateFloat("solde", solde + (solde * taux));
            rs.updateRow();
        }
    }
    catch ( SQLException e ) {
        throw new RuntimeException(e);
    }
    finally { // traitement des close() }
}
```




Les requêtes pré compilées

- Pour gagner en performance lors de l'exécution répétée de mêmes requêtes SQL, il est recommandé de les pré compiler.
- Les requêtes pré compilées sont supportées par JDBC sous la forme d'une Interface dérivée de Statement :

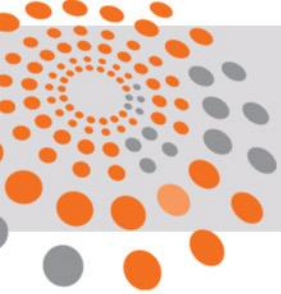
- l'interface *PreparedStatement* :

```
PreparedStatement pstmt = cx.prepareStatement(requete);
```

- Ce type de requête peut être paramétrée :

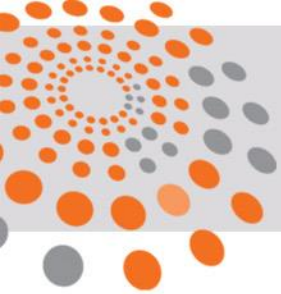
```
String requete;
```

```
requete = "UPDATE Compte SET solde = ? WHERE id = ?"
```



Paramétrage des requêtes pré compilées

- Les méthodes **setXxx()** permettent de valoriser les différents paramètres de la requête :
 - `void setInt (int parameterIndex, int i)`
 - `void setFloat (int parameterIndex, float f)`
 - `void setString (int parameterIndex, String s)`
- Une fois la requête totalement valorisée, la méthode **execute()** permet son exécution.
- Attention la valeur de l'index est « 1 based » comme beaucoup de mécanismes SQL indexés



Exemple de requête pré compilée

```
public static void updateSolde (int id , float somme) {  
    try {  
        PreparedStatement pstmt;  
        String requete = "UPDATE Compte " +  
                        "SET solde = ? " +  
                        "WHERE id = ?";  
  
        cx = DriverManager.getConnection(url, uid, pass);  
        pstmt = cx.prepareStatement(requete);  
        pstmt.setFloat(1, somme);  
        pstmt.setInt (2, id);  
        pstmt.execute();  
    }  
    catch ( SQLException e ) { // Erreur }  
    finally { // traitement des close() }  
}
```



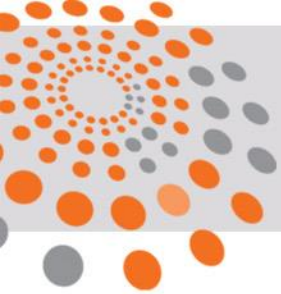
Les transactions JDBC





Gestion des transactions

- Une **transaction** représente un ensemble de requêtes devant s'exécuter de façon atomique.
- Par défaut, l'objet Connection valide (commit) à chaque requête au niveau de la base de données.
- La gestion des transactions nécessite de désactiver le mode de validation par défaut (commit) :
`void setAutoCommit(boolean mode) throws SQLException`
- L'appel d'un `setAutoCommit(false)` caractérise le début d'une transaction
- La fin d'une transaction est bornée par une des méthodes :
`void commit() throws SQLException`
`void rollback() throws SQLException`



Exemple de mise en œuvre d'une transaction JDBC

```
public static void virement(int source, float montant, int dest) {  
    try {  
        cx = DriverManager.getConnection(url, uid, pass);  
        cx.setAutoCommit(false);  
        stmt = cx.createStatement();  
        stmt.executeUpdate("UPDATE compte SET solde=solde - "  
                           + montant + " WHERE id=" + source);  
        stmt.executeUpdate("UPDATE compte SET solde=solde + "  
                           + montant + " WHERE id=" + dest);  
  
        cx.commit();  
    }  
    catch (SQLException e ) {  
        if (cx != null) try {  
            cx.rollback();  
        } catch (SQLException e1) {  
            throw new RuntimeException(e1);  
        }  
    }  
    finally { // traiter les close() }  
}
```



Transaction et concurrence d'accès

- Les **niveaux d'isolation** JDBC caractérisent la manière dont sont préservées les données accédées **concurrentement** par plusieurs transactions.
- JDBC matérialise cette isolation par la pose de verrous (en lecture ou écriture) sur les données tant que la transaction n'est pas validée.
- 5 niveaux d'isolation sont possibles, définis par des constantes de l'interface ***Connection*** :
 - ***TRANSACTION_READ_UNCOMMITTED***
 - ***TRANSACTION_READ_COMMITTED***
 - ***TRANSACTION_REPEATABLE_READ***
 - ***TRANSACTION_SERIALIZABLE***
 - ***TRANSACTION_NONE***

Niveaux d'isolation

Niveaux d'isolation	Lecture impropre	Lecture non répétable	Lecture fantôme
<i>TRANSACTION_READ_UNCOMMITTED</i>	possible	possible	possible
<i>TRANSACTION_READ_COMMITTED</i>	impossible	possible	possible
<i>TRANSACTION_REPEATABLE_READ</i>	impossible	impossible	possible
<i>TRANSACTION_SERIALIZABLE</i>	impossible	impossible	impossible

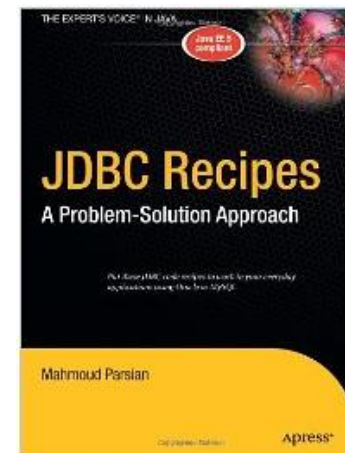
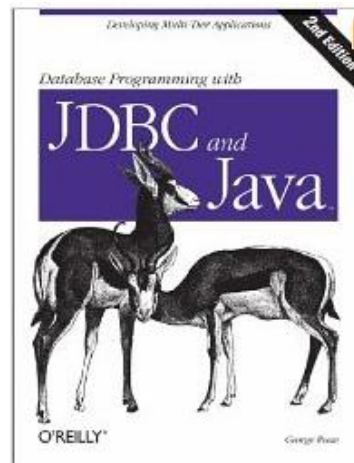
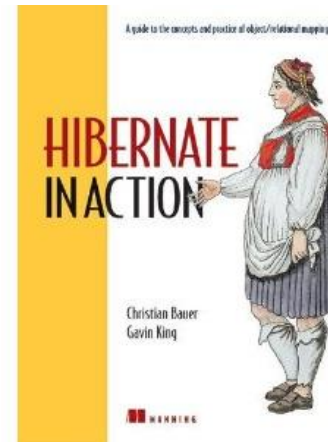
- **Lecture impropre** : ou " dirty read " : Une transaction lit des données contenant un changement non validé d'une autre transaction. Une partie des données peut se révéler fausse en fonction du commit ou du rollback de l'autre transaction.
- **Lecture non répétable** : ou " nonrepeatable reads " : Une transaction lit une donnée, une seconde transaction change la même donnée et la première transaction relit la donnée et obtient une valeur différente. La donnée a changé et est incohérente sur la transaction.
- **Lecture fantôme** : provoque une anomalie lorsqu'une même requête de votre transaction exécutée successivement produit un nombre de lignes différent (une autre transaction ajoute ou supprime des lignes dans la table)



Niveaux d'isolation (suite)

- La méthode ***getIsolationLevel()*** de la l'interface Connection permet de connaître le niveau d'isolation.
- Symétriquement, la méthode ***setIsolationLevel()*** permet de changer ce niveau.
- Attention, tous les SGBDR ne supportent pas tous les types d'isolation ...
- la méthode ***supportsTransactionIsolationLevel(int level)*** permet de savoir si le niveau d'isolation désiré est supporté par la base de données utilisée.

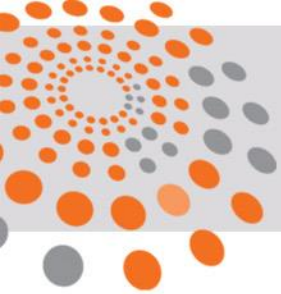
JDBC - JPA - Hibernate : documentation





Présentation de l'AWT

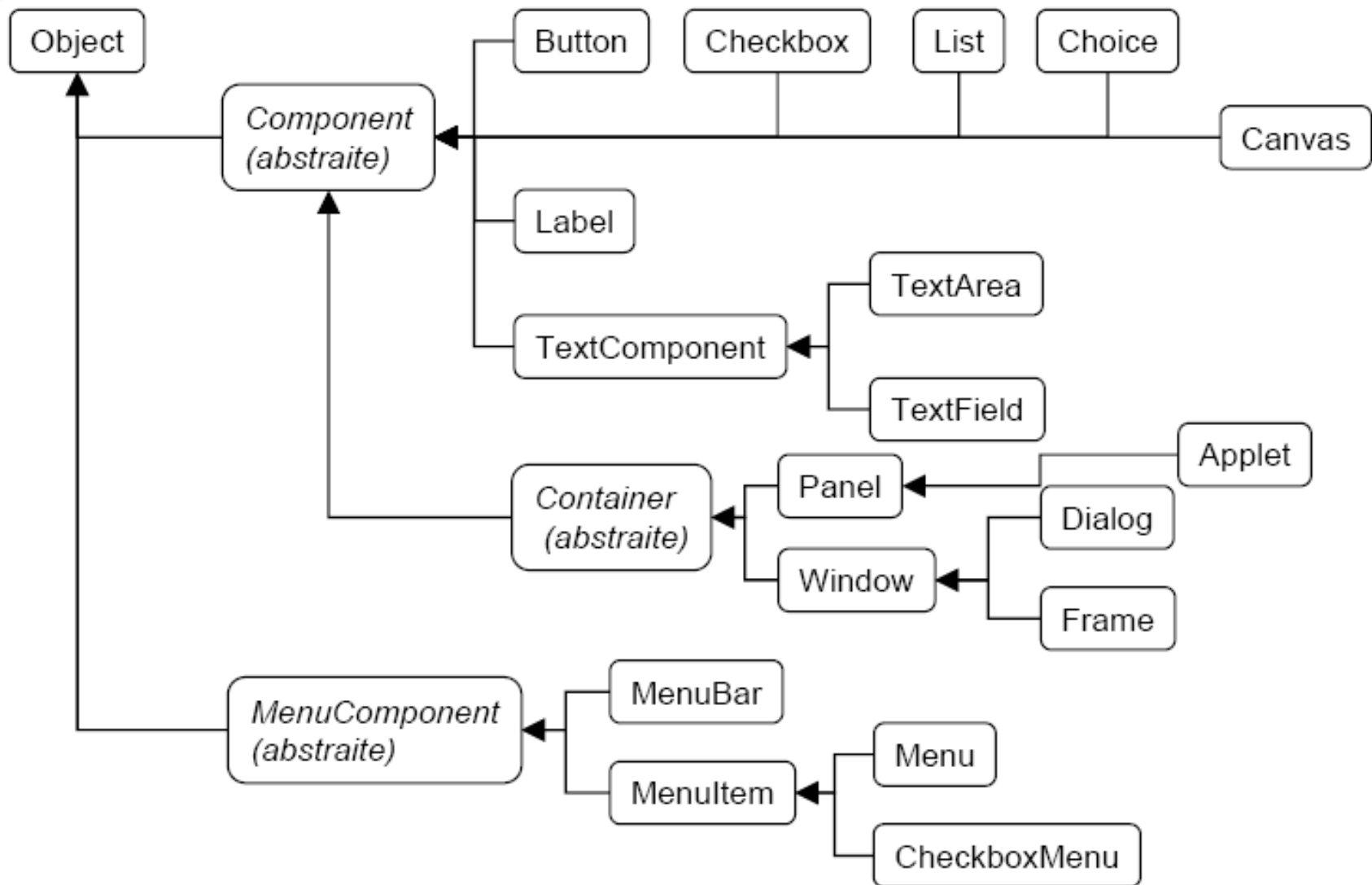


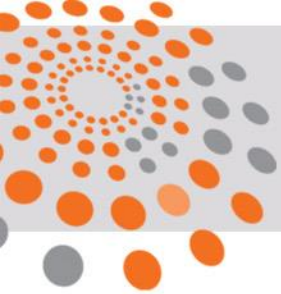


La librairie awt (package java.awt)

- Elle fournit une hiérarchie de classes permettant la réalisation d'interfaces graphiques.
- Elle met à disposition des :
 - Composants de base d'une interface (outils de fenêtrage : bouton , label, ...),
 - Composants de regroupement (panel, boite de dialogue,...),
 - Composants pour la réalisation de menus.

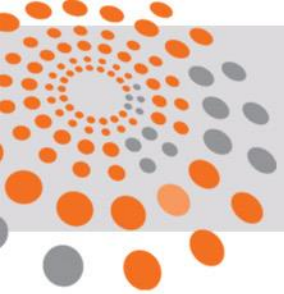
La librairie awt





Processus de conception d'une interface graphique en Java

- Créer une classe par héritage de la classe *Frame*
- Choisir et positionner le *layout manager*
- Créer les composants
- Ajouter les composants dans la classe container
- Gérer les événements



La classe Frame

- Elle permet de définir des composants (fenêtres) qui contiennent d'autres composants.
- Un composant frame est caractérisé par :
 - un nombre et un ensemble de composants
 - un moyen de disposer ses composants (layoutMgr qui est de type LayoutManager).

Disposition des composants (le layoutManager)

- Les objets containers ont un **LayoutManager** qui leur est associé et qui définit la manière dont les composants vont être disposés.
- awt fournit 6 gestionnaires de mise en page :

BorderLayout

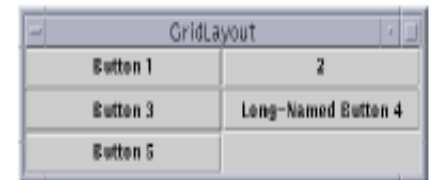
GridLayout

GridBagLayout

CardLayout

FlowLayout

BoxLayout



- La classe containers fournit des méthodes pour gérer le **LayoutManager** :
void setLayout (LayoutManager mgr), LayoutManager getLayout ()



La classe Label

- La création d'une étiquette peut se faire avec un libellé, sans libellé, en indiquant la position du libellé :

Label () , Label (String libelle)

Label (String libelle , int alignement)

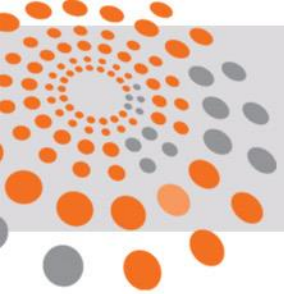
- L'alignement peut prendre trois valeurs :

LEFT, CENTER, RIGHT

- On a la possibilité de gérer le libellé et son positionnement :

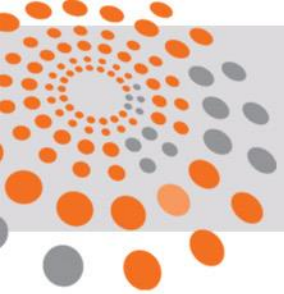
int getAlignment() , void setAlignment(int alignement)

String getText () , void setText (String libelle)



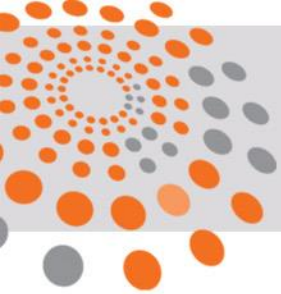
La classe Button

- Elle définit un bouton poussoir.
- La création se fait avec un libellé ou sans libellé :
 `Button ()`, `Button (String libelle)`
- On a la possibilité de gérer le libellé :
 `String getLabel ()`, `void setLabel (String libelle)`



La classe TextField

- Elle définit une zone texte monoligne.
- La création se fait avec un libellé ou sans libellé :
TextField () , TextField (String libelle)
- Les principales méthodes sont :
String getText (), void setText (String libelle)
void setEditable (boolean etat), boolean isEditable ()
String getSelectedText ()
void select (int selStart, int selEnd)



Ajout et suppression de Composants

- L'ajout d'un composant dans une classe container se fait par la méthode ***add()*** avec la possibilité de spécifier la position où l'on veut rajouter le composant :
 - Component add (Component comp)
 - Component add (Component comp, int pos)
 - Component add (String name, Component comp)
- Suppression d'un composant :
 - void remove (Component comp)
 - void removeAll ()

Exemple

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

```
class Fenetre extends Frame {  
    private Button b1 , b2, b3 , b4 , b5 ;  
    public Fenetre() {  
        setLayout( new BorderLayout() ) ;  
        b1= new Button("un"); add("North", b1) ;  
        b2 = new Button("deux"); add("East", b2) ;  
        b3 = new Button("trois"); add("South", b3) ;  
        b4 = new Button("quatre"); add("West", b4) ;  
        b5 = new Button(" cinq "); add("Center", b5) ;  
        setSize(300, 150 ); setVisible(true) ;  
    }  
}
```

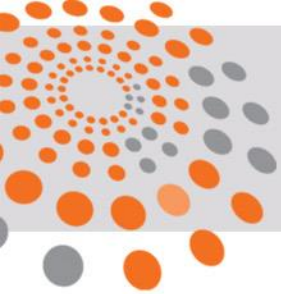
```
public static void main ( String args[] ) {  
    Fenetre appli = new Fenetre () ;  
}  
}
```





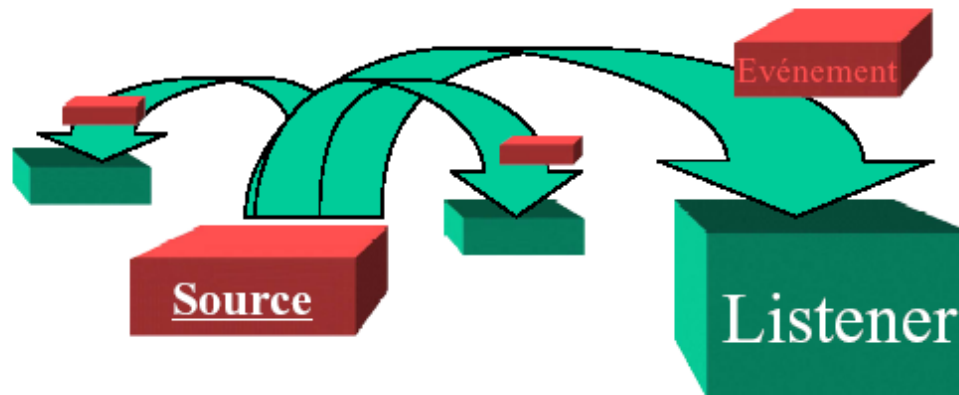
Le modèle événementiel





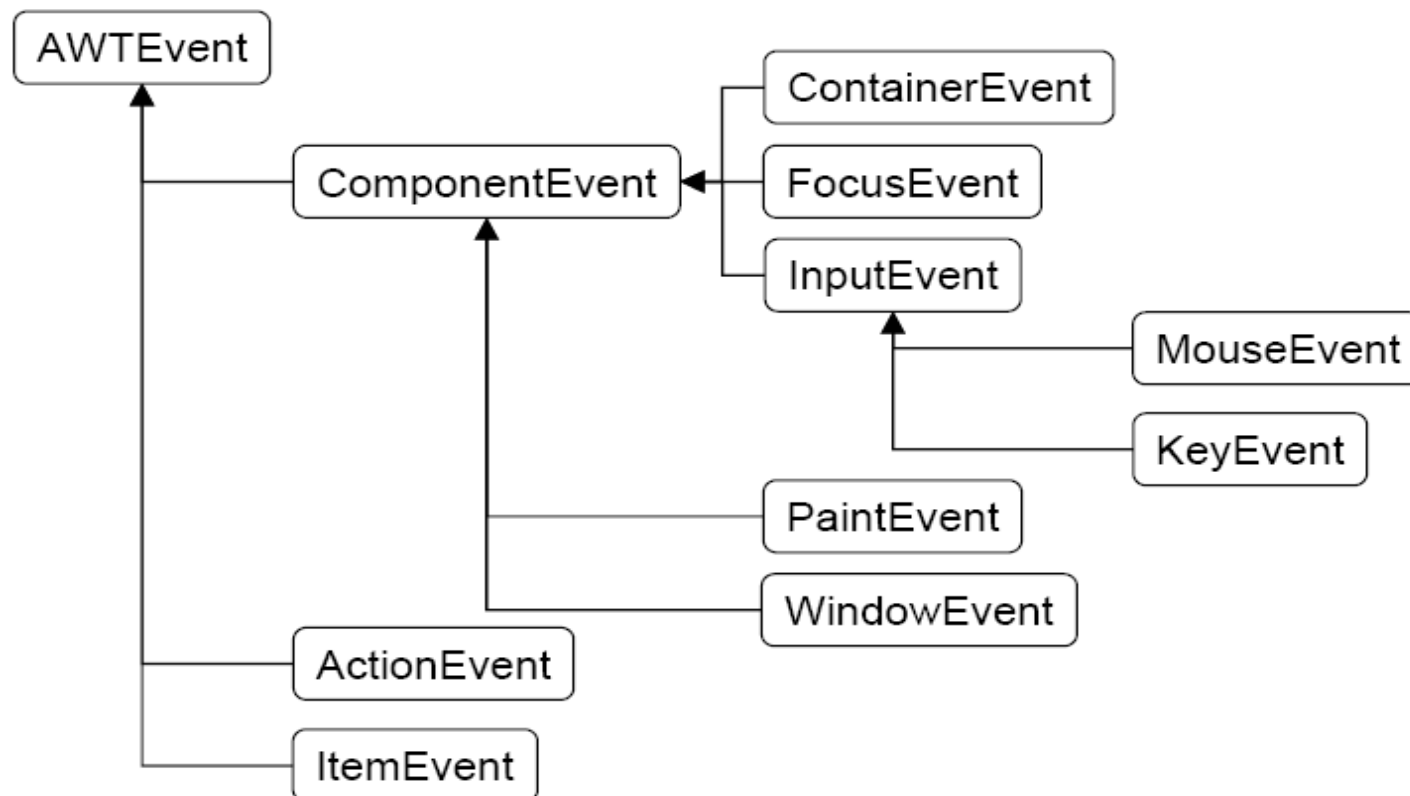
Principes du modèle événementiel (depuis JDK 1.1)

- Un modèle de programmation MVC où les échanges entre Modèle, View et Contrôleur s'effectuent sous forme d'**OBJETS EVENEMENTS**
- Un événement est propagé d'une Source vers un objet Listener en invoquant une méthode du listener et en passant une sous classe de EventObject qui définit le type d'événement qui a été engendré.



Les événements

- Pas une seule classe mais une hiérarchie de classes d'événements
- Extensible en fonction de vos propres besoins



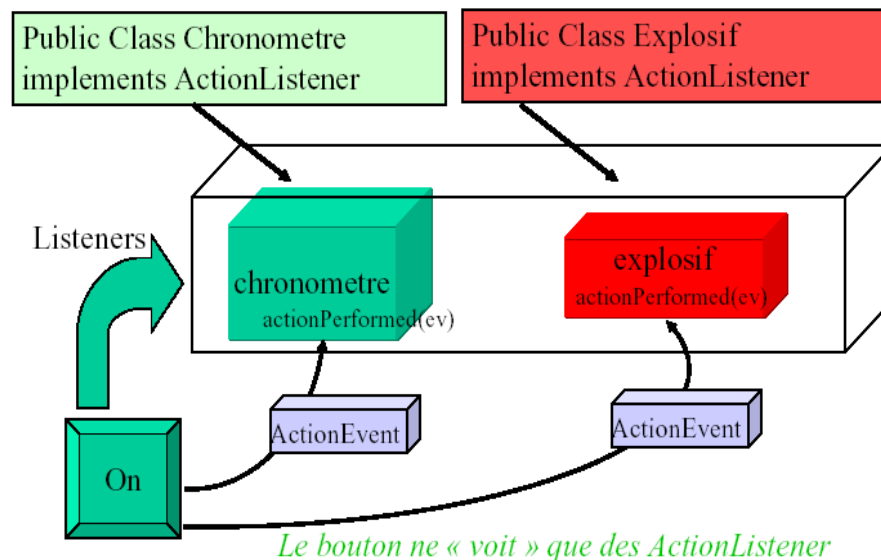


La source

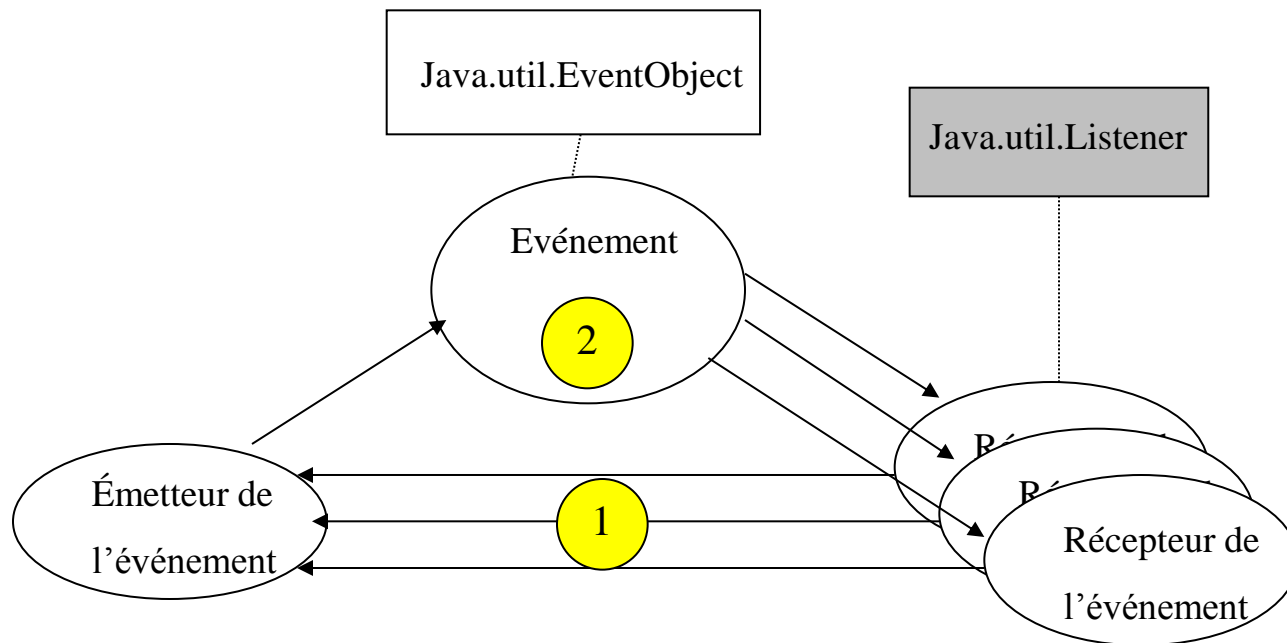
- L'objet Source émet des événements.
- La source **enregistre** ses listeners par des méthodes
set<EventType>Listener ou
add<EventType>Listener
- Toutes les sources d'événements supportent un modèle **multicast** pour l'émission : Plusieurs listeners peuvent être ajoutés pour une seule source.
- L'API n'a **aucun a priori** sur l'ordre dans lequel les événements sont délivrés.

Les listeners

- Un objet listener est un objet qui implémente (*remplit le contrat*) d'une interface spécifique.
- Cette interface (*ce contrat*) prévoit une ou plusieurs méthodes qui peuvent être invoquées par une source d'événements.
- Chaque listener reçoit une réplique identique de l'événement original.



Listener : exemple d'un ActionListener



- Un récepteur doit s'enregistrer auprès de la source en choisissant le type d'événement qui souhaite recevoir :

unBouton.addActionListener(leListener)

- Et implémenter l'interface correspondante pour recevoir les événements et les traiter :

void actionPerformed(ActionEvent e)

Listener : exemple

```
import java.awt.*;
import java.awt.event.*;
class Fenetre extends Frame implements ActionListener {
    private Button bouton1 ;
    public Fenetre () {
        FlowLayout f = new FlowLayout(); setLayout (f);
        setLayout(new FlowLayout());
        bouton1 = new Button("click me"); add(bouton1);
        bouton1.addActionListener(this);
        setSize(300, 200); setVisible(true);
    }
    public void actionPerformed(ActionEvent evt) {
        setBackground (Color.green);
    }
    public static void main ( String args[] ) {
        new Fenetre().setVisible(true);
    }
}
```



Exemple : reconnaissance de la source de l'événement

```
import java.awt.*;  
import java.awt.event.*;  
class Fenetre extends Frame implements ActionListener  
{ private Button bouton1, bouton2 ;  
  public Fenetre() {  
    setLayout(new FlowLayout());  
    bouton1 = new Button("Green"); add(bouton1);  
    bouton2 = new Button("Red"); add(bouton2);  
    bouton1.addActionListener(this);  
    bouton2.addActionListener(this);  
    setSize(300, 200); setVisible(true);  
  }  
  public void actionPerformed(ActionEvent evt) {  
    if(evt.getSource()==bouton1)  
      setBackground(Color.green);  
    else if(evt.getSource()==bouton2)  
      setBackground(Color.red);  
  }  
}
```





Les principaux types de listeners et interfaces associées (I)

- **ActionListener**

actionPerformed (ActionEvent e)	activer un composant
---------------------------------	----------------------

- **MouseListener**

mouseClicked (MouseEvent e)	click souris
mousePressed (MouseEvent e)	bouton enfoncé
mouseReleased (MouseEvent e)	bouton relâché
mouseEntered (MouseEvent e)	entrée souris sur un composant
mouseExited (MouseEvent e)	sortie souris d'un composant

- **MouseMotionListener**

mouseDragged (MouseEvent e)	déplacement souris
mouseMoved (MouseEvent e)	déplacement souris



Les principaux types de listener et interfaces associées (II)

- **KeyListener**

keyPressed (KeyEvent e)

touche enfoncée

KeyReleased (KeyEvent e)

touche relâchée

KeyTyped (KeyEvent e)

caractère entré

- **WindowListener**

windowActivated(WindowEvent e)

windowClosed(WindowEvent e)

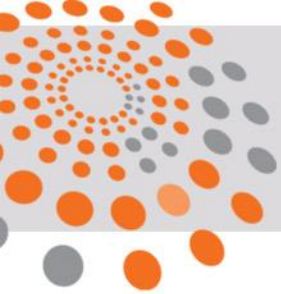
windowClosing(WindowEvent e)

windowDeactivated(WindowEvent e)

windowDeiconified(WindowEvent e)

windowIconified(WindowEvent e)

windowOpened(WindowEvent e)



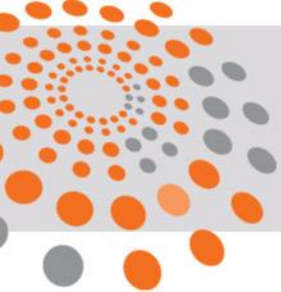
Les principaux types de listeners et interfaces associées (III)

- **TextListener**
textValueChanged(TextEvent e)
- **FocusListener**
focusGained(FocusEvent e)
focusLost(FocusEvent e)



Adaptateurs et classes internes

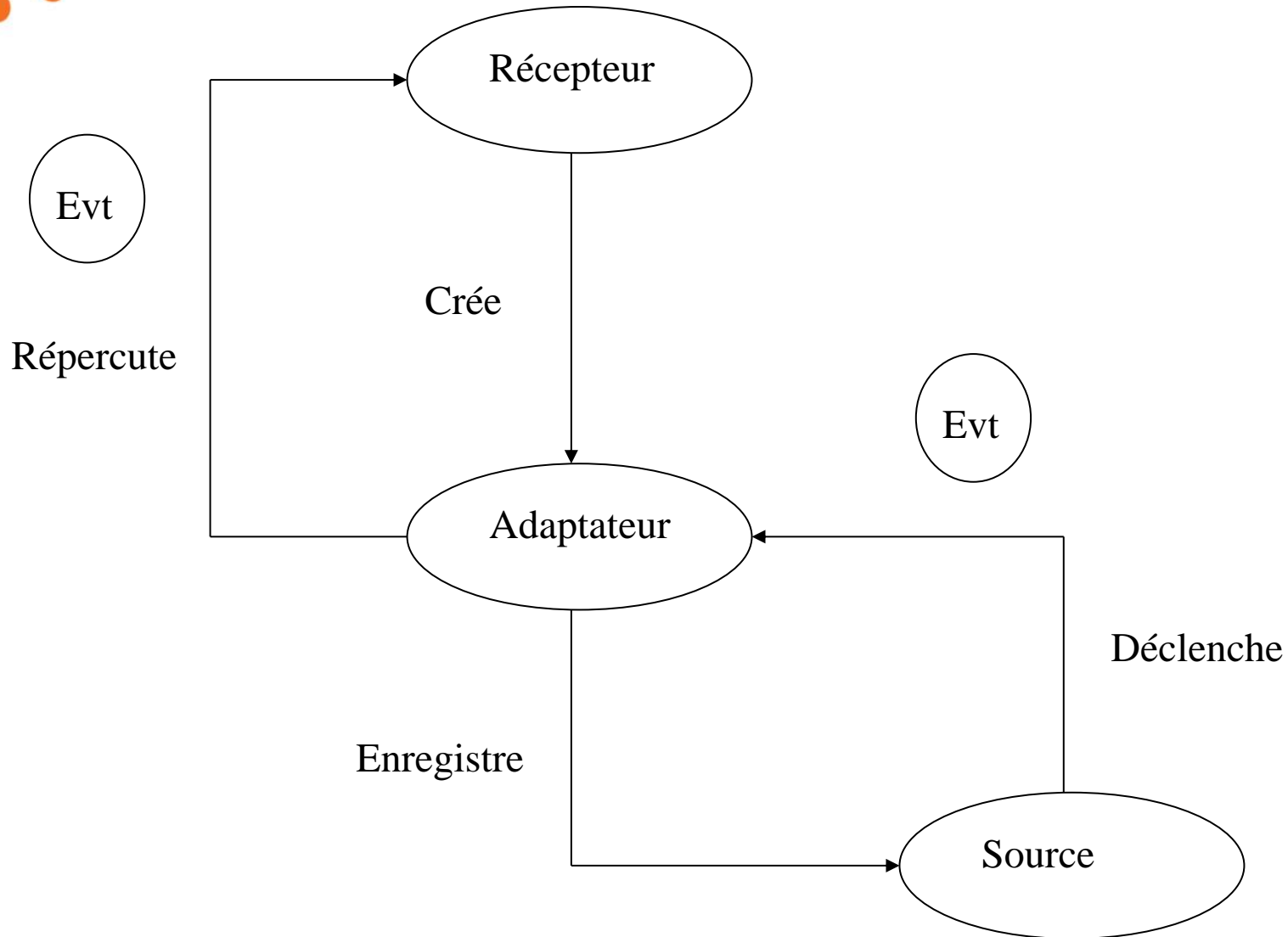




Les adaptateurs

- Les **adaptateurs** sont des classes intermédiaires qui servent de tampon entre l'objet source et le récepteur.
- Ils permettent de séparer le code de prise en compte des événements du code du récepteur.
- Le rôle d'un adaptateur est d'implémenter l'interface (ou les interfaces) des événements que le récepteur souhaite surveiller.

Les adaptateurs : cinématique





Exemple d'adaptateur (I)

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

class Fenetre extends Frame
{
    private Button bouton1 ;
    public Fenetre ()
    {
        setLayout (new FlowLayout());
        bouton1 = new Button("click me");
        add(bouton1);
        bouton1.addActionListener(new EvtFenetre(this));
        setSize(300, 200);
        setVisible(true);
    }
}
```



Exemple d'adaptateur (II)

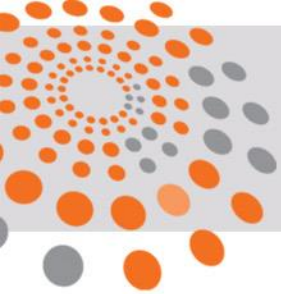
```
class EvtFenetre implements ActionListener {  
    Fenetre f ;  
    public EvtFenetre(Fenetre aFenetre) {  
        f = aFenetre ;  
    }  
  
    public void actionPerformed(ActionEvent evt) {  
        f.setBackground(Color.green);  
    }  
}  
  
public class Demo {  
    public static void main(String args[]){  
        Fenetre appli = new Fenetre();  
    }  
}
```



Notion de composant : les classes internes

- Une **classe interne** est une classe définie à l'intérieur d'une autre classe, elle possède les mêmes attributs de visibilité qu'une variable ou une méthode (public, private, etc).

```
class ClasseParente {  
    ...  
    modificateur class ClasseInterne {  
        instructions ...  
    }  
    ...  
}
```



Notion de composant : les classes internes (suite)

- Une classe interne déclarée avec le modificateur d'accès *private* implique que cette classe ne pourra être utilisée que dans sa classe parente
- Une classe interne a un accès complet sur toutes les propriétés (attributs et méthodes même *private*) de sa ou de ses classes englobantes (imbrication de classes internes).
- L'instanciation de la classe interne passe obligatoirement par une instance préalable de la classe d'inclusion. La classe parente est d'abord instanciée, puis c'est au tour de la classe interne de l'être.



Adaptateur et classe interne

```
class Fenetre extends Frame {  
    private Button bouton1 ;  
    public Fenetre() {  
        setLayout(new FlowLayout());  
        bouton1=new Button("click me "); add(bouton1);  
        bouton1.addActionListener( new EvtFenetre());  
        setSize(300, 200); setVisible(true);  
    }  
    class EvtFenetre implements ActionListener {  
        public void actionPerformed(ActionEvent evt) {  
            setBackground(Color.green);  
        }  
    }  
}
```




Utilisation des classes anonymes

```
import java.awt.*;
import java.awt.event.*;
class Fenetre extends Frame {
    private Button bouton1;
    public Fenetre () {
        setLayout(new FlowLayout());
        bouton1 = new Button("click me "); add(bouton1);
        bouton1.addActionListener(new ActionListener () {
            public void actionPerformed(ActionEvent evt) {
                setBackground(Color.green ); }
        });
        setSize (300, 200); setVisible(true) ;
    }
}
```

- Une classe anonyme a toujours pour vocation de spécialiser une classe existante ou d'implémenter une interface, ici l'interface ActionListener.



Les adaptateurs de java.awt

- L'awt fournit des adaptateurs standard pour différents types de listeners (ex: *MouseAdapter*, *WindowAdaptater*, ...).
- Ces adaptateurs sont des classes abstraites qui fournissent des traitements par défaut aux événements du listener associé (ces traitements consistent souvent à ignorer l'événement).
- Utiliser un adaptateur java.awt nécessite de le sous-classer et de redéfinir les méthodes (les événements) que vous souhaitez traiter spécifiquement.

...

```
this.addWindowListener(new java.awt.event.WindowAdapter() {  
    public void windowClosing(java.awt.event.WindowEvent e) {  
        System.out.println("windowClosing()");  
        dispose();  
    }  
});
```



AWT : Exemple de synthèse

Exemple de synthèse

- Un convertisseur Francs / Euros



The image shows a software window titled "Conversion Francs <-> Euros" with a blue title bar and standard Windows window controls (minimize, maximize, close). The window has a yellow background and contains two columns of input fields and buttons. The left column is for Francs, showing a value of 982.5 and a button labeled "Francs vers Euros". The right column is for Euros, showing a value of 150 and a button labeled "Euros vers Francs". The button for Euros is highlighted with a dashed border.

Montant en Francs	Montant en Euros
982.5	150
Francs vers Euros	Euros vers Francs

Un outil pour faciliter la conception d'interfaces graphiques Plugin Eclipse : Visual Editor (=> WindowBuilder ou JFormDesigner)

Java - Calculatrice.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Conversion Francs <-> Euros

Montant en Francs Montant en Euros

Francs vers Euros Euros vers Francs

```
if(sens == true) {
    valFranc = Float.parseFloat(textFrancs.getText());
    valEuro = valFranc / 6.55f;
    textEuros.setText(String.valueOf(valEuro));
} else {
    valEuro = Float.parseFloat(textEuros.getText());
    valFranc = valEuro * 6.55f;
    textFrancs.setText(String.valueOf(valFranc));
}

public static void main(String args[] ) {
    Calculatrice c = new Calculatrice();
} // @jve:decl-index=0:visual-constraint="10,10"
```

Outline

import declarations

Calculatrice

- serialVersionUID : long
- labelFranc : Label
- labelEuros : Label
- textFrancs : TextField
- textEuros : TextField
- buttonFranc : Button
- buttonEuros : Button
- Calculatrice()
- initialize()
- new WindowAdapt
- ... windowClosing
- getTextFrancs()
- getTextEuros()
- getButtonFranc()
- new ActionListener
- ... actionPerformed
- getButtonEuros()
- new ActionListener
- ... actionPerformed
- conversionFrancEuro(bc
- main(String[])

Java Beans

- this-"Conversion Francs <->..."
- ... windowClosing
- labelFranc-"Montant en Francs"
- labelEuros-"Montant en Euros"
- textFrancs
- textEuros
- buttonFranc-"Francs vers Euros"
- ... actionPerformed
- buttonEuros-"Euros vers Francs"
- ... actionPerformed

Problems Javadoc Declaration Search Console Variables Breakpoints Properties

Calculatrice [Java Application] C:\Program Files\Java\jdk1.5.0_06\bin\javaw.exe (27 févr. 07 23:20:08)

Writable Smart Insert 134 : 15 In Sync



Code généré par Visual Editor (1)

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

```
public class Calculatrice extends Frame {  
    private static final long serialVersionUID = 1L;  
    private Label labelFranc = null;  
    private Label labelEuros = null;  
    private TextField textFrancs = null;  
    private TextField textEuros = null;  
    private Button buttonFranc = null;  
    private Button buttonEuros = null;
```

```
    public Calculatrice() {  
        super();  
        initialize();  
        setVisible(true);
```

```
}
```

```
...
```



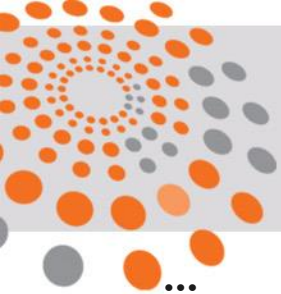
Code généré par Visual Editor (2)

```
private void initialize() {  
    labelEuros = new Label();  
    labelEuros.setBounds(new Rectangle(165, 40, 117, 29));  
    labelEuros.setAlignment(Label.CENTER);  
    labelEuros.setText("Montant en Euros");  
    labelFranc = new Label();  
    labelFranc.setBounds(new Rectangle(15, 40, 117, 29));  
    labelFranc.setAlignment(Label.CENTER);  
    labelFranc.setText("Montant en Francs");  
    this.setLayout(null);  
    this.setSize(300, 162);  
    this.setBackground(Color.orange);  
    this.setForeground(Color.darkGray);  
    this.setResizable(false);  
    this.setTitle("Conversion Francs <-> Euros");  
    ...  
}
```



Code généré par Visual Editor (3)

```
private void initialize() {  
    ...  
    this.add(labelFranc, null);  
    this.add(labelEuros, null);  
    this.add(getTextFrancs(), null);  
    this.add(getTextEuros(), null);  
    this.add(getButtonFranc(), null);  
    this.add(getButtonEuros(), null);  
    this.addWindowListener(  
        new java.awt.event.WindowAdapter() {  
            public void windowClosing(java.awt.event.WindowEvent e) {  
                dispose();  
            }  
        });  
}
```

Code généré par Visual Editor (4)

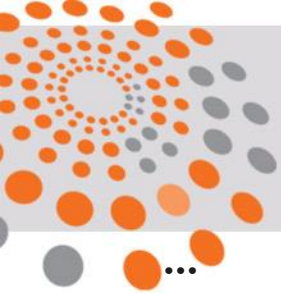
```
...  
private TextField getTextFrancs() {  
    if (textFrancs == null) {  
        textFrancs = new TextField();  
        textFrancs.setBounds(new Rectangle(15, 70, 116, 26));  
    }  
    return textFrancs;  
}  
  
private TextField getTextEuros() {  
    if (textEuros == null) {  
        textEuros = new TextField();  
        textEuros.setBounds(new Rectangle(165, 70, 116, 26));  
    }  
    return textEuros;  
}  
...
```



Code généré par Visual Editor (5)

```
private Button getButtonFranc() {  
    if (buttonFranc == null) {  
        buttonFranc = new Button();  
        buttonFranc.setBounds(new Rectangle(15, 110, 116, 31));  
        buttonFranc.setLabel("Francs vers Euros");  
        buttonFranc.addActionListener(  
            new java.awt.event.ActionListener() {  
                public void actionPerformed(java.awt.event.ActionEvent e) {  
                    conversionFrancEuro(true);  
                }  
            });  
    }  
    return buttonFranc;  
}
```

```
private Button getButtonEuros() { ... }
```



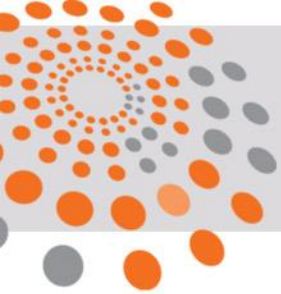
La fonction de conversion (6)

```
private void conversionFrancEuro(boolean sens) {  
    float valEuro = 0, valFranc = 0;  
    if(sens == true) {  
        valFranc = Float.parseFloat(textFrancs.getText());  
        valEuro = valFranc / 6.55f;  
        textEuros.setText(String.valueOf(valEuro));  
    }  
    else {  
        valEuro = Float.parseFloat(textEuros.getText());  
        valFranc = valEuro * 6.55f;  
        textFrancs.setText(String.valueOf(valFranc));  
    }  
}  
  
public static void main(String args[] ) {  
    Calculatrice c = new Calculatrice();  
}  
}
```



Présentation des swing





Les JFC (Java Foundation Classes jdk1.2)

- Librairie graphique s'appuyant sur la librairie graphique d'origine de Java : l'AWT.
- Elle se divise en quatre groupes :
 - **Swing** : ensemble de composants légers, développés par JavaSoft sur la base de composants IFC de netscape.
 - **Java 2D API** : gestion complète du dessin 2D (texte élaboré, images, couleurs , composition, ...) .
 - **Java Accessibility** : ensemble de classes pour gérer des périphériques de communication destinés essentiellement aux handicapés (lecteurs braille...).
 - **Drag and Drop** : gestion du glisser-déposer généralisé.

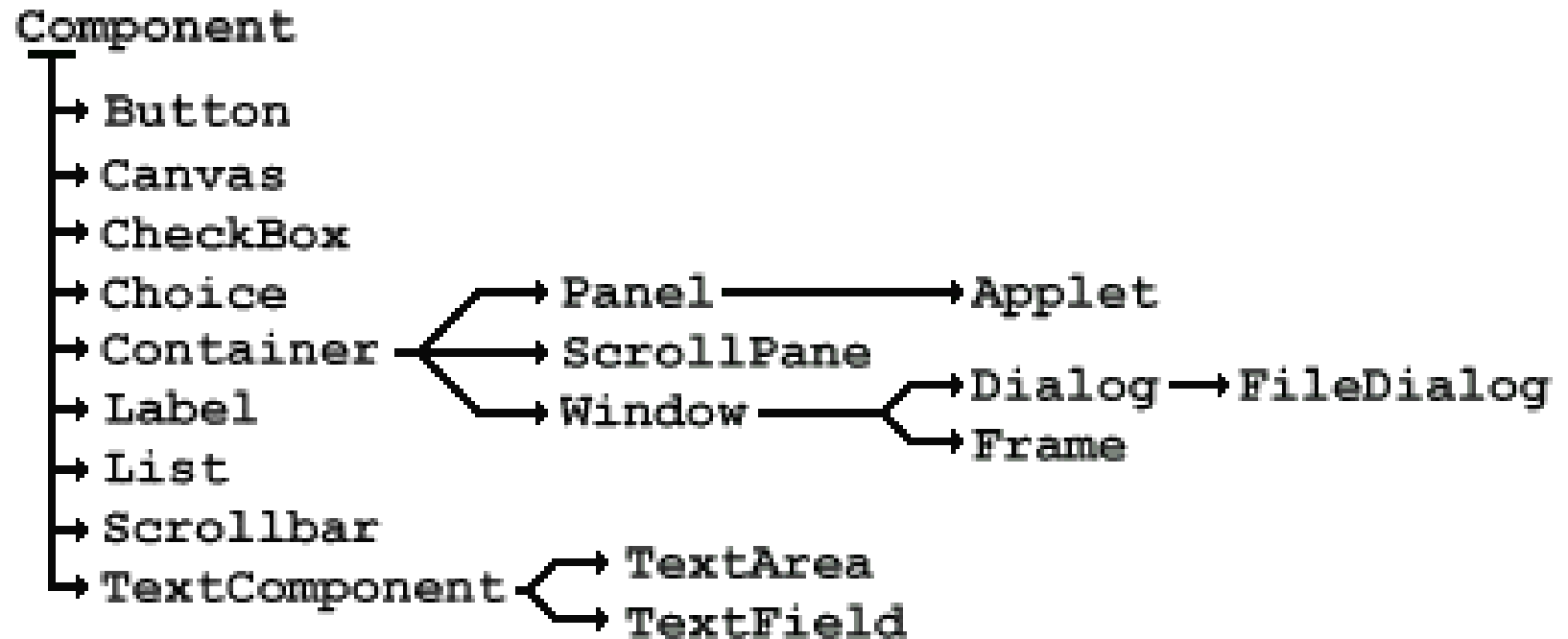


Les swing

- Ils sont basés sur le nouveau modèle de composants « légers » qui consiste à dessiner directement les composants en 100% Java sans utiliser les primitives graphiques de bas niveau des systèmes hôtes.
- Ils redéfinissent l'ensemble des composants de base de l'awt (*JButton*, *JLabel*, *JTextField* ...) et amène des nouvelles fonctionnalités et des composants de plus haut niveau (onglets, feuille de calcul ...).
- Ils utilisent le modèle événementiel du jdk1.1.
- Ils offrent de meilleures performances notamment en ce qui concerne la mémoire occupée.
- Il est possible de dessiner des contrôles transparents, ...
- Il est possible de changer dynamiquement le « look and feel » de l'interface et de créer son propre look and feel .

Hiérarchies de classe AWT

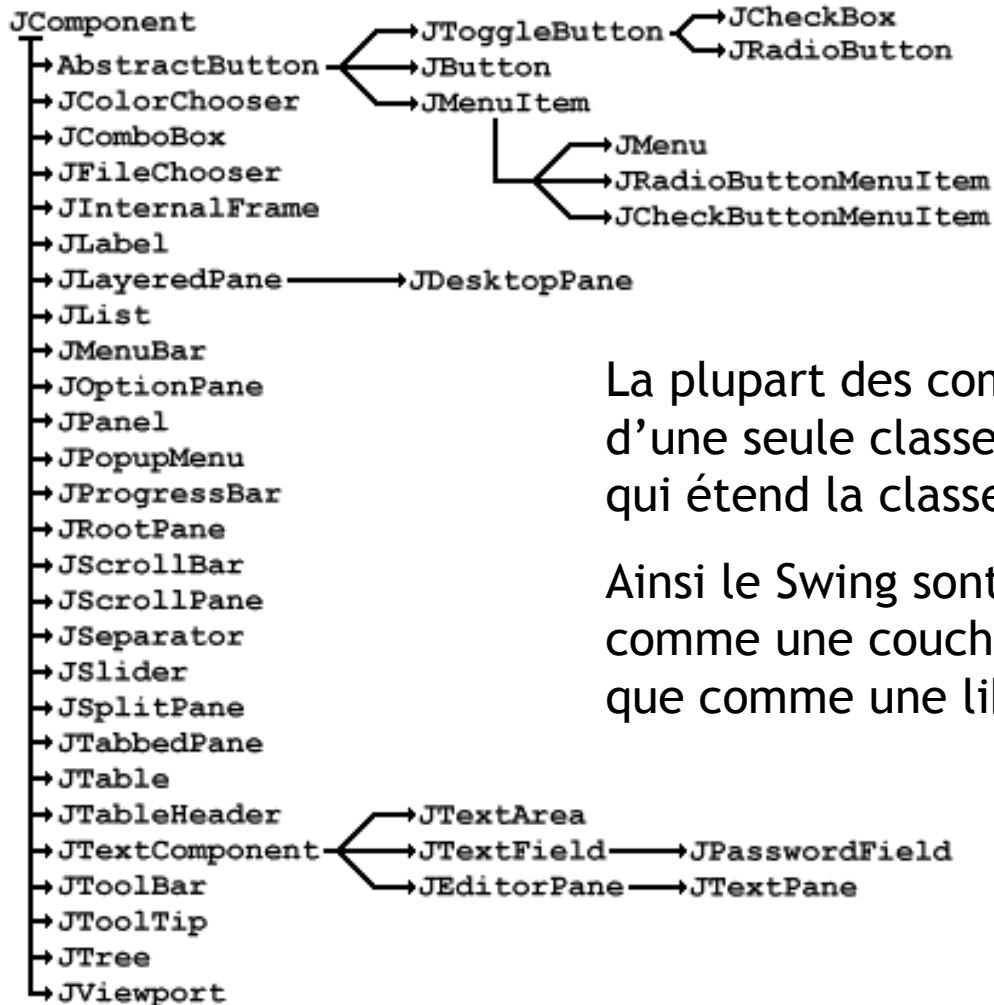
- Hiérarchie de l'awt (vue partielle)



- La plupart des composants AWT dérivent de `java.awt.Component`. A noter que les menus ne font pas partie de cette hiérarchie.

Hiérarchies de classe AWT

- Hiérarchie des Swing (vue partielle)



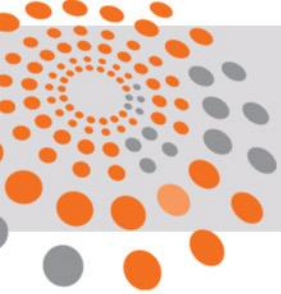
La plupart des composants Swing dérivent d'une seule classe parente : `JComponent` qui étend la classe `Container` de l'AWT.

Ainsi le Swing sont plus à considérer comme une couche au dessus de l'AWT que comme une librairie la remplaçant.

Exemples de référence

- <http://java.sun.com/docs/books/tutorial/uiswing/components/example-swing>

Source Files	Image & Other Files	Where Described
ButtonDemo.java	right.gif , middle.gif , left.gif	How to Use Buttons ...
CheckBoxDemo.java	All of the images in the images/geek directory.	How to Use Buttons ...
ColorChooserDemo.java		How to Use Color Choosers
ColorChooserDemo2.java CrayonPanel.java	red.gif , yellow.gif , green.gif , blue.gif	How to Use Color Choosers
ComboBoxDemo.java	Bird.gif , Cat.gif , Dog.gif , Rabbit.gif , Pig.gif	How to Use Combo Boxes
ComboBoxDemo2.java		How to Use Combo Boxes
CustomComboBoxDemo.java	Bird.gif , Cat.gif , Dog.gif , Rabbit.gif , Pig.gif	How to Use Combo Boxes
DialogDemo.java CustomDialog.java	middle.gif	How to Use Dialogs
...



Swing : un exemple avec des radiobox

- C'est un ensemble de "***ToggleButton***" à choix exclusif.
- La création d'une ***radiobox*** s'effectue de la façon suivante :
 - Création d'un JPanel
 - Création d'un ButtonGroup
 - Création des JRadioButton
 - Ajout des JRadioButton dans la ButtonGroup
 - Ajout des JRadioButton dans le JPanel
- Le changement d'état des boutons est géré en implémentant l'interface ***ItemListener***.



Example

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

>>

class TestRadio extends JFrame
{
    private ButtonGroup bg ;
    private JRadioButton bGreen, bRed, bYellow ;
    private JPanel pColor , pSurface ;
    public TestRadio () {
        buildPanelColor(); buildPanelSurface(); addEvent();
        setSize(400, 200); show();
    }
    public void buildPanelSurface () {
        pSurface = new JPanel();
        this.getContentPane().add("Center", pSurface);
    }
}
```



Example

```
public void buildPanelColor() {  
    pColor = new JPanel() ;  
    this.getContentPane().add("South", pColor);  
    pColor.setBorder(new TitledBorder("Couleur"));  
    pColor.setLayout(new GridLayout(1, 3));  
    bg = new ButtonGroup();  
    bGreen = new JRadioButton("Green");  
    bg.add(bGreen);  
    pColor.add(bGreen);  
    bRed = new JRadioButton("Red", true);  
    bg.add(bRed);  
    pColor.add(bRed);  
    bYellow = new JRadioButton("Yellow");  
    bg.add(bYellow);  
    pColor.add(bYellow);  
}
```

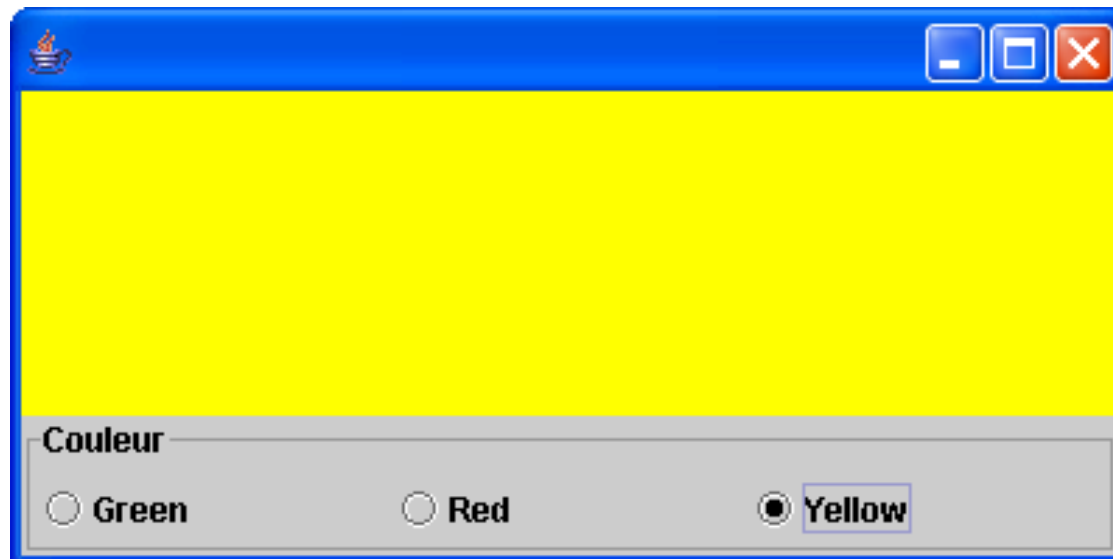


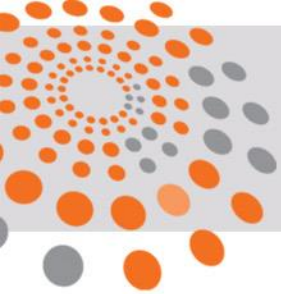
Example

```
public void addEvent() {  
    ColorItemListener evtColor = new ColorItemListener();  
    bGreen.addItemListener(evtColor);  
    bRed.addItemListener(evtColor);  
    bYellow.addItemListener(evtColor);  
}  
private class ColorItemListener implements ItemListener {  
    public void itemStateChanged(ItemEvent e) {  
        JToggleButton source = (JToggleButton)e.getItem();  
        String text = source.getText();  
        if(text.equals("Green"))  
            pSurface.setBackground(Color.green);  
        else if(text.equals("Red"))  
            pSurface.setBackground(Color.red);  
        else if(text.equals("Yellow"))  
            pSurface.setBackground(Color.yellow);  
    }  
}
```

Exemple

```
class Demo
{
    public static void main (String args []) {
        TestRadio t = new TestRadio();
    }
}
```





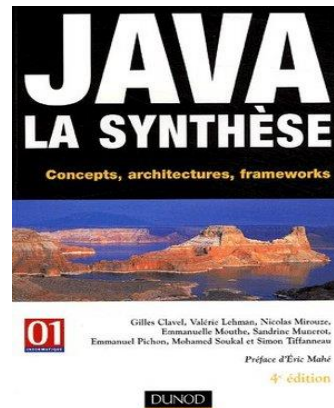
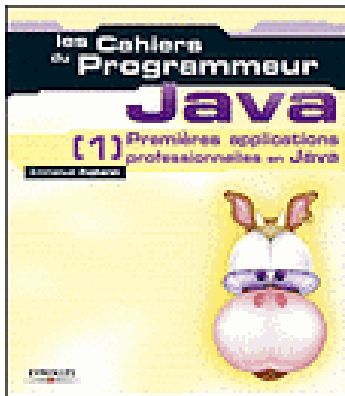
Exercice : IHM Swing

The screenshot shows a Java Swing window titled "gestion des comptes" with a blue title bar and standard window controls (minimize, maximize, close). The window contains the following elements:

- N° du compte:** A text field containing the number "2".
- valider:** A button located to the right of the account number field.
- Valeur du solde:** A text field displaying "-390.0".
- Montant de l'opération:** A text field containing "200".
- crédit / débit:** Two buttons located below the operation amount field.
- Liste des opérations du compte:** A text area displaying a list of transactions:
 - CREDIT jeudi 1 mars 2007
 - CREDIT jeudi 1 mars 2007
 - DEBIT jeudi 1 mars 2007
 - CREDIT jeudi 1 mars 2007
 - CREDIT jeudi 1 mars 2007

Pour aller plus loin ... bibliographie

- Littérature très abondance, pour débiter et à titre d'exemples :



- Sites web
 - <http://java.sun.com/docs/books/tutorial/> : tutoriels en ligne en anglais
 - <http://java.sun.com/> : le site de référence
 - <http://www.javaworld.com/> : revue en ligne consacrée à Java
 - <http://java.developpez.com/cours/> : tutoriels java en français
 - <http://www.laltruiste.com> : autres tutoriels en français