Une introduction à Java

Jean-François Remm

(Jean-Francois.Remm@ujf-grenoble.fr)

Duke is saying ... 

... Hello

Plan du cours

- **Introduction**
- Langage
- Entrée/Sortie
- Threads
- Interface graphique
- Applet
- Perspective - Conclusion

Objectifs

- Beaucoup de bruit autour du langage Java
- Savoir programmer en JAVA
- Savoir programmer des Applets
- Surtout, utiliser Java comme langage de référence

Environnement

- Outils logiciels : J2SE (*Java 2 Stadard Edition*) ex-JDK (*Java Development Kit*) :
  - compilateur (*javac*),
  - debugger (*jdb*),
  - testeur d'applets (*appletviewer*)
  - archiveur (*jar*)
  - machine virtuelle (*java*)
  - ...
- Documentations
  - spécification du langage : `http://doc.src/docs/JavaLang`
  - tutorial `http://doc.src/docs/JavaTutorial`
  - apis (*Application Programming Interface*) `http://doc.src/docs/JavaApi`
  - le site de référence `http://java.sun.com`
- Livres
  - *Java in a Nutshell* (O'Reilly)
  - Java facile (Marabout)

Vue d'ensemble

Java est un langage :

– **Simple, Orienté objet, Distribué, Interprété, Robuste, Sûr, Neutre, Portable, Haute Performance, Multi-Thread, Dynamique.**

Plan du cours

- Introduction
- **Langage**
- Entrée/Sortie
- Threads
- Interface graphique
- Applet
- Perspective - Conclusion

Langage : Plan

- **Généralités**
- Compilation&exécution
- Expressions, contrôle ...
- Classes&Objets
- Héritage
- Exceptions
- Encapsulation
- Interfaces
- Divers

Généralités

Java est un **langage à objets** ou **langage de classes**.

On y retrouve les paradigmes objets classiques :

– classe et objet

– encapsulation

– héritage

– polymorphisme

On trouve aussi des notions plus nouvelles :

– paquetage (*package*)

– interface

– exception.

Langage : Généralités

- langage de classe à **héritage simple**
- syntaxe proche de C et C++
- typage **fort** des variables
- deux sortes de types :
  1. types “simples” ou primitifs $\neq$ objets :
     `int` (entier), `short` (entier court), `long` (entier long), `float` (réel), `double` (réel long), `boolean` (booléen), `char` (caractère) et `byte` (octet)
  2. types “références” : objets et tableaux, par défaut à la valeur `null`.
- une donnée de type simple est manipulée par **valeur**
- une donnée de type référence est manipulé par **référence**

Entiers

- tous les entiers sont signés
- codés en compléments à deux
- détail :

type	défaut	taille	valeurs possibles
`byte`	0	8 bits	-128⟶127
`short`	0	16 bits	-32768⟶32767
`int`	0	32 bits	-2147483648⟶2147483647
`long`	0	64 bits	-9223372036854775808⟶9223372036854775807

Réels

- codage selon la norme IEEE 754
- détail :

type	défaut	taille	valeurs possibles
`float`	0.0	32 bits	$1.40239846 * 10^{-45} \rightarrow 3.40282347 * 10^{+38}$
`double`	0.0	64 bits	$4.9406564584124654 * 10^{-324}$
			$\rightarrow 1.7976931348623157 * 10^{+308}$

Booléens

- les booléens ne sont QUE des booléens
- détail :

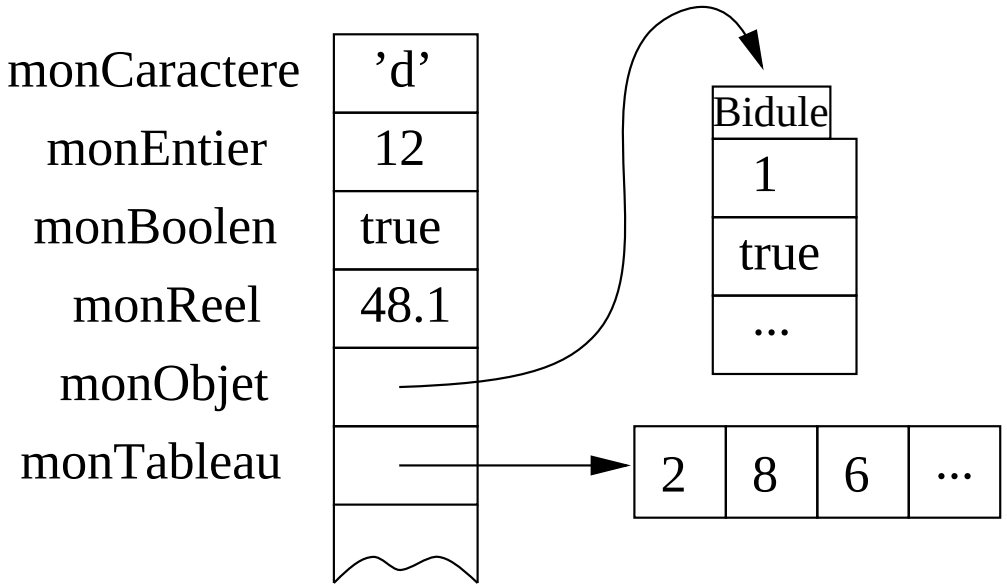
type	défaut	taille	valeurs possibles
`boolean`	`false`	1 bit	`true` `false`

Caractères

- les caractères sont codés en unicode
- détail :

type	défaut	taille	valeurs possibles
`char`	`\u0000`	16 bit	`\u0000` → `\uFFFF`

Schéma mémoire



Langage : Plan

- Généralités
- **Compilation&exécution**
- Expressions, contrôle ...
- Classes&Objets
- Héritage
- Exceptions
- Encapsulation
- Interfaces
- Divers

Compilation

- le code source java est saisi dans des fichiers avec un suffixe `.java`.
- On compile ces fichiers (appel à `javac`) et on obtient autant de fichier `.class` que de classes définies dans le fichier `.java`. Ces fichiers `.class` portent le nom des classes.

Exécution

- l'exécution (interprétation) se fait par appel de `java`
- À l'appel de la machine virtuelle, on donne en paramètre le NOM D'UNE CLASSE (et non un fichier)
- le `.class` correspondant doit pouvoir être trouvé dans la liste des répertoires de recherche contenu dans la variable d'environnement `CLASSPATH`.
- chargement dans la JVM et exécution débutant par la méthode `main` :
  public static void main(String[] args)

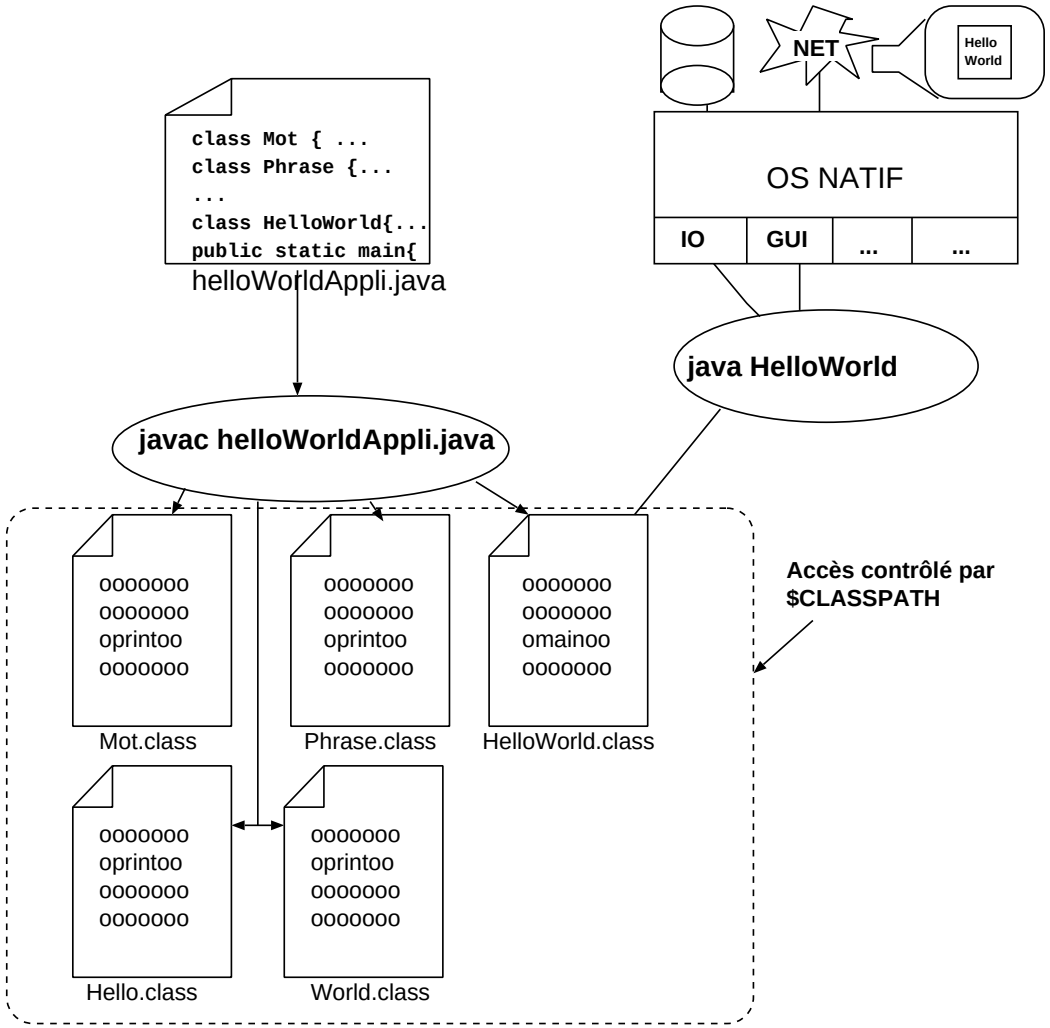
Premier exemple

```
class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

Exemple

```
$ javac HelloWorld.java
$ java HelloWorld
Hello World !
```

Compilation&exécution



Langage : Plan

- Généralités
- Compilation & exécution
- **Expressions, contrôle ...**
- Classes & Objets
- Héritage
- Exceptions
- Encapsulation
- Interfaces
- Divers

Variables

- variables **déclarées** et **typées**
- principe de déclaration : `nomType nomVariable;`
- exemple :

```
public class DemoVariable
{
    public static void main (String[] args)
    {
        // déclaration d'un entier i
        int i;

        // déclaration d'une chaine de caractère
        String s;

        // déclaration et initialisation d'un réél
        double d = 3.23;

        // déclaration de deux entier
        int x,y;
    }
}
```

Opérateurs

- les opérateurs sont classés par ordre de priorité
  1. ++ : incrémentation, -- : décrémentation, +, - : opérateurs unaires, ! : négation logique (type) : transtypage
  2. * : multiplication, / : division, % : modulo
  3. + : addition, - : soustraction, + : concaténation
  4. >, >=, <, <=, instanceof
  5. == : égalité, != : différence
  6. && : ET logique
  7. || : OU logique
  8. = : affectation, *=, /=, +=

Opérateurs

```
public class DemoOperateur
{
    public static void main (String[] args)
    {
        int i = 3;
        int j;

        j = i+7;
        System.out.println(j);

        System.out.println("la valeur de j est : " + j);

        System.out.println(i>j);
    }
}
```

```
10
la valeur de j est : 10
false
```

Transtypage

- sert à convertir d'un type vers un autre
- Attention : il FAUT que la conversion soit possible
- Exemple :

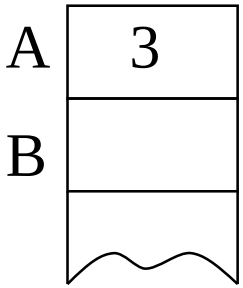
Transtypage

```
public class SimpleCastDemo
{
    public static void main(String args[])
    {
        double d1 = 12.0;
        double d2 = 2.54;
        //-> int a1 = d1;
        int a1 = (int)d1;
        //-> int a2 = d2;
        int a2 = (int) d2;
        System.out.println(a1 + " " + a2);
    }
}
```

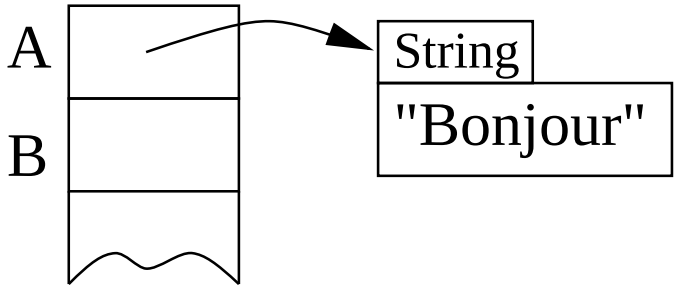
```
12 2
```

```
SimpleCastDemoB.java:7: possible loss of precision
found   : double
required: int
int a1 = d1;
                 ^
SimpleCastDemoB.java:8: possible loss of precision
found   : double
required: int
int a2 = d2;
                 ^
2 errors
```

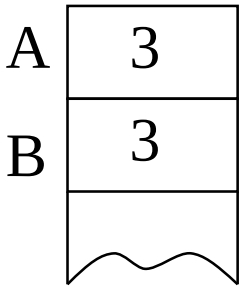
Affectations : `b=a ;`



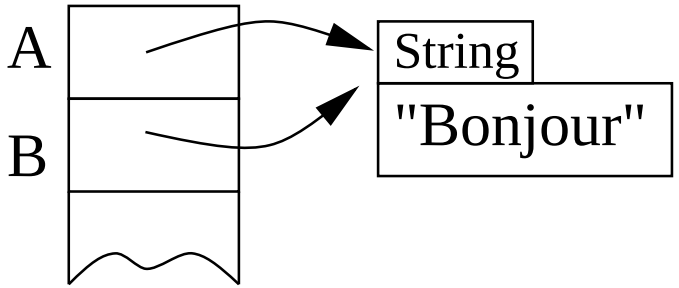
(a) Vue 1



(b) Vue 1



(c) Vue 2



(d) Vue 2

Incrémentation – décrémentation

```
import java.io.*;

public class DemoIncDec
{
  public static void main(String args[])
  {
    int x = 0;
    int y = 0;

    x++;
    ++y;

    System.out.print("x == "+x);
    System.out.println("      y == "+y);

    System.out.print("x++ == "+ x++);
    System.out.println("      ++y == "+ ++y);

    System.out.print("x == "+x);
    System.out.println("      y == "+y);

    x = x++;
    y = --y;

    System.out.print("x == "+x);
    System.out.println("      y == "+y);
  }
}
```

```
x == 1  y == 1
x++ == 1  ++y == 2
x == 2  y == 2
x == 2  y == 1
```

Conditionnelle : `if/else`

- Syntaxe :

```
if (condition)
   instruction1
[else
   instruction2]
```

- condition est une expression à **valeur booléenne**
- Exemple :

```
int i=10;
if (i<20)
{
  i = 14;
  System.out.println("vrai");
}
else
  System.out.println("faux");
```

Boucle : `while`

- syntaxe :

```
while (condition)
   instruction
```

- `condition` est une expression à **valeur booléenne**
- Exemple :

```
int i=10;
while (i<20)
{
  System.out.println(i);
  i = i + 5;
}
```

Boucle : do/while

- syntaxe :

```
do
{
  instructions
}
while (condition);
```

- condition est une expression à **valeur booléenne**
- Exemple :

```
int i=10;
do
{
  System.out.println(i);
  i = i + 5;
}
while (i<20);
```

Boucle : `for`

- syntaxe :

```
for(initialisation ; condition ; suite)
    instruction
```

$\Leftrightarrow$

```
initialisation
while (condition)
{
  instruction
  suite
}
```

- Exemple :

```
int i,j;
for (i=0, j=9;   // initialisation
     i<j && j>2; // test de continuation
     i++, j--)
  System.out.println(i + " " + j);
```

Aiguillage : `switch/case`

- Syntaxe :

```
switch (expression)
{
    case val1:
        instructions1
    case val2:
      ...
    default:
        instructionsD
}
```

- `expression` à valeur entière (`byte`, `short`, `int`, `long`) ou caractère (`char`)
- Exemple :

```
switch (i)
{
  case 1:
    System.out.println("1");
  default:
    System.out.println("defaut");
}
```

Langage : Plan

- Généralités
- Compilation& exécution
- Expressions, contrôle ...
- **Classes&Objets**
- Héritage
- Exceptions
- Encapsulation
- Interfaces
- Divers

“Philosophie” Java

- tout programme Java est constitué d'au moins une définition de classe
- rien n'existe hors de ces définitions de classes
- un “programme principal” n'est qu'une méthode (particulière) d'une classe : la méthode `main`

Classes&Objets

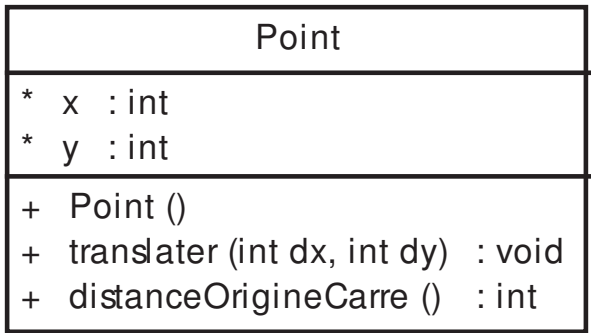
CLASSE :

- description STATIQUE d'une famille d'objets ayant même structure et même comportement
- "plan de construction"

OBJET (ou instance) :

- entité créée DYNAMIQUEMENT, en respectant les plans de construction donnés par sa classe
- existe lorsque le programme s'exécute

Exemple


Point
* x : int
* y : int
+ Point ()
+ translater (int dx, int dy) : void
+ distanceOrigineCarre () : int

Définir une classe

- décrire l'**ÉTAT** (la structure) d'une instance :
  - Variables d'instances (ou propriétés) : données caractérisant l'état d'une instance à un moment donné
- décrire son **COMPORTEMENT** :
  - Méthodes : fonctions spécialisées dans la manipulation des instances
  - Constructeurs : méthodes spéciales construisant les objets

Définir une classe

- une classe (publique) `nomClasse` se saisit dans un fichier `nomClasse.java`
- la syntaxe de définition de la classes est la suivante :

```
[public | private] class nomClasse [extends autreClass]
                                   [implements nomInterface]
{
   ...
}
```

```
public class Point
{

}
```

Définir l'état

- l'état est décrit par les variable d'instance (variable caractérisant une instance)
- toute déclaration de variable en dehors d'une méthode ⟶ variable d'instance
- une déclaration de variable dans une méthode ⟶ variable locale à la méthode

---

```
public class Point
{
    int x;
    int y;
}
```

Définir le comportement

- un comportement est décrit par des **méthodes**
- définition méthode = **signature** + **corps**

Définir la signature

- **signature** = nom + nombre et types des paramètre + type de retour
- TOUJOURS un type de retour; `void` éventuellement
- TOUJOURS des parenthèses
- TOUJOURS un type pour le paramètre formel

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

    public int distanceOrigineCarre()
    {
      ...
    }

    public void translater(int dx, int dy)
    {
      ...
    }
}
```

Définir le corps

- codage des actions
- accès aux **paramètres** : info "entrante"
- accès aux **variables d'instances** : état consultable et modifiable
- deux cas :
  1. type de retour $\neq$ `void` : renvoie la valeur de retour avec `return` : info "sortante"
  2. type de retour $=$ `void` : PAS de `return`

```
...

public int distanceOrigineCarre()
{
    return (x*x+y*y);
}

public void translater(int dx, int dy)
{
    x = x + dx;
    y = y + dy;
}

...
```

Définir le constructeur

- cas particulier de comportement
- porte **toujours** le nom de la classe
- **pas** de type de retour
- **pas** de `return`
- si nous ne l'écrivons pas, existence d'un constructeur par défaut (sans paramètres) qui initialise les variables d'instance aux valeurs par défaut.

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

    public Point()
    {
        x = 0;
        y = 0;
    }
```

Classe `Point`

```
public class Point
{
  // Variables d'instances

  int x;
  int y;


  // Constructeur

  public Point()
  {
    x=0;
    y=0;
  }

  // Méthodes


  public void translater( int dx, int dy )
  {
    x = x + dx;
    y = y + dy;
  }

  public int distanceOrigineCarre()
  {
    return (x*x+y*y);
  }
}
```

Utiliser une classe

- on écrit une autre classe avec une méthode `main`

```
public class TestPoint
{
    public static void main(String[] args)
    {
        ...
    }
}
```

Création d'objets

- correspond à une allocation de mémoire
- crée dynamiquement une nouvelle **instance** de la classe
- s'occupe des **initialisations** du nouvel objet
- se fait à l'aide du mot clé `new`
- en appelant un constructeur qui a le **même nom** que la classe
- exemple :

```
Date aujourdhui = new Date();
Random alea = new Random(10l);
Point p1 = new Point();
```

- seule exception : `String s = "bonjour";` en fait c'est un raccourci syntaxique

Accès à une variable d'instance

- une variable d'instance dépend d'une **instance** particulière
- utilisation de la notation pointée
- sur le format `monInstance . laVI`
- **jamais** de `()`
- exemple :

```
Point pt;
pt = new Point();
pt.x = 3;
```

- bien souvent on évite d'accéder "directement" à une variable d'instance → encapsulation des données

Utilisation de méthodes

- une méthode s'utilise sur une **instance** particulière
- utilisation de la notation pointée
- sur le format `monInstance . laMethode(paramEventuel1, ...)`
- **toujours** des `()`
- fournir les éventuels **paramètres** en **nombre exact** et **correctement typés**
- exemple :

```
aujourdhui.getMinutes();
s.charAt(2);
```

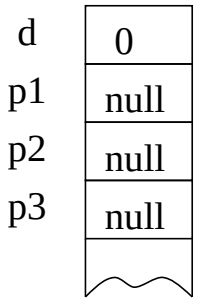
- attention : l'objet DOIT avoir été créé préalablement

Utilisation de la classe `Point`

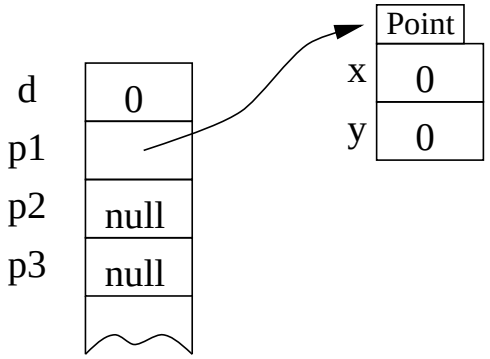
```
public class TestPoint
{
  public static void main(String args[])
  {
    Point p1, p2 ,p3;
    int d;
    // 1
    // instanciation du point p1
    p1 = new Point(); // 2
    // translatation du point désigné par p1 :
    p1.translater(2,3); // 3
    // calcul dans d de la distance carre a l'origine
    d = p1.distanceOrigineCarre(); // 4
    // exemple de consultation de variables d'instances
    d = p1.x + p1.y; // 5
    // instanciation et manipulation d'un deuxième point :
    p2 = new Point();
    p2.translater(p1.y,6);
    d = p2.distanceOrigineCarre(); // 6
    // signification de l'affectation
    p3 = p2;
    p3.translater(1,6); // 7
    // perdre l'acces α une instance
    p1 = null; // 8
    // affectation d'un variable contenant null
    p2 = p1; //9
  }
}
```

Classes&Objets : exemple

```
Point p1, p2 ,p3;

int d;

// Vue 1

// instanciation du point p1

p1 = new Point();

// Vue 2
```



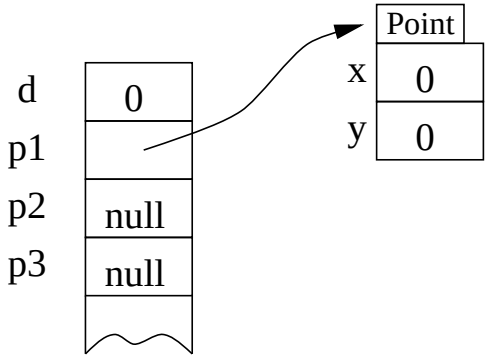
(e) Vue 1



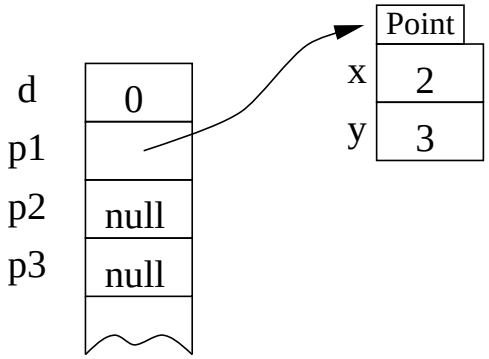
(f) Vue 2

Classes&Objets : exemple

```
// Vue 2

// translatation du point d=E9sign=E9 par p1 :

p1.translater(2,3);

// Vue 3
```



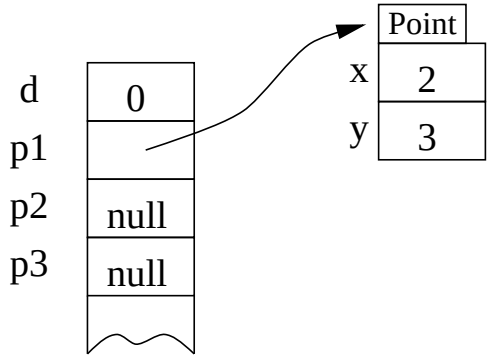
(g) Vue 2



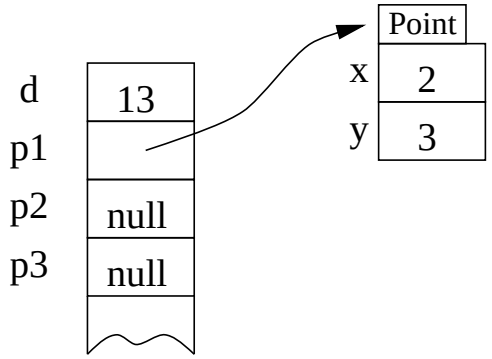
(h) Vue 3

Classes&Objets : exemple

```
// Vue 3

// calcul dans d de la distance carre a l'origine

d = p1.distanceOrigineCarre();

// Vue 4
```



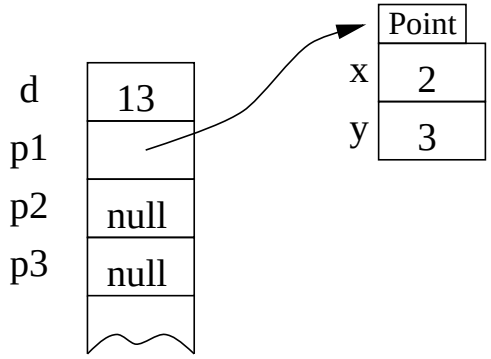
(i) Vue 3



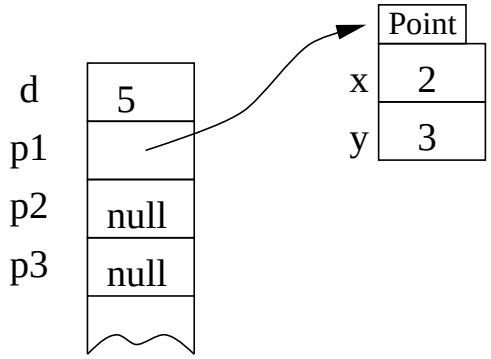
(j) Vue 4

Classes&Objets : exemple

```
// Vue 4

// exemple de consultation de variables d'instances

d = p1.x + p1.y;

// Vue 5
```



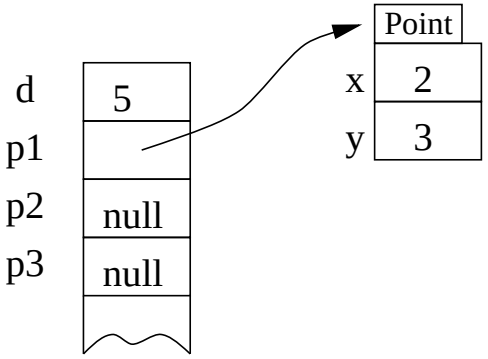
(k) Vue 4



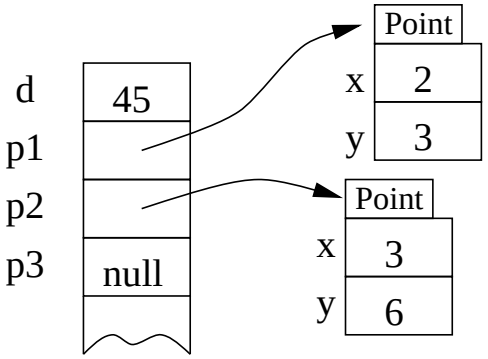
(l) Vue 5

Classes&Objets : exemple

```
// Vue 5

// instanciation et manipulation d'un deuxième point :

p2 = new Point();

p2.translater(p1.y,6);

d = p2.distanceOrigineCarre();

// Vue 6
```



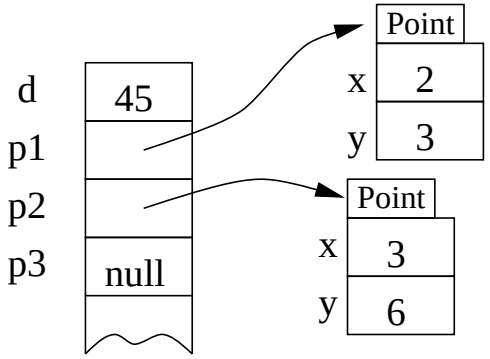
(m) Vue 5



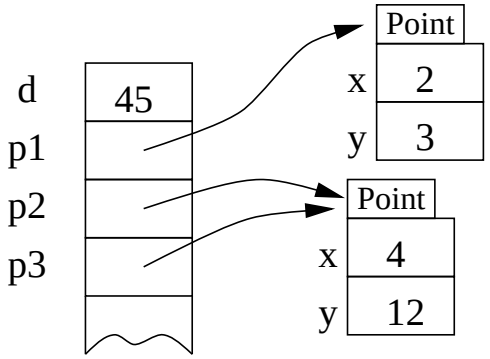
(n) Vue 6

Classes&Objets : exemple

```
// Vue 6

// signification de l'affectation

p3 = p2;

p3.translater(1,6);

// Vue 7
```



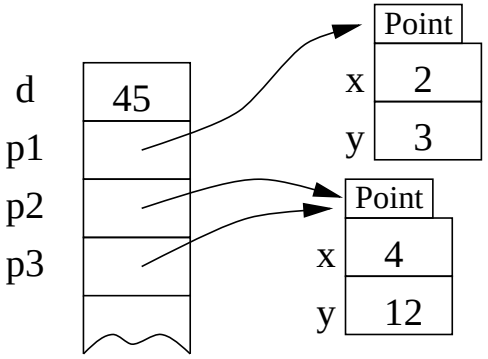
(o) Vue 6



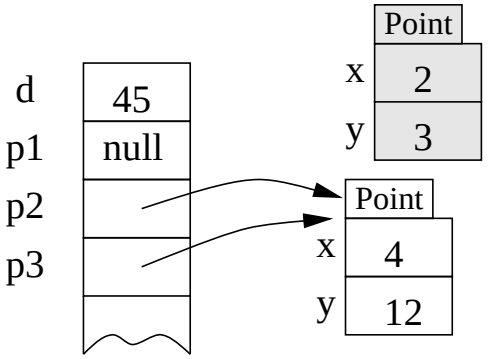
(p) Vue 7

Classes&Objets : exemple

```
// Vue 7

// perdre l'acces α une instance

p1 = null ;

// Vue 8
```



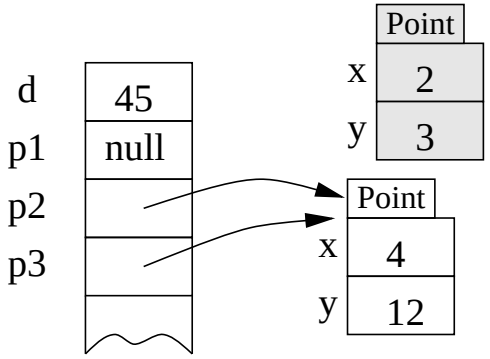
(q) Vue 7



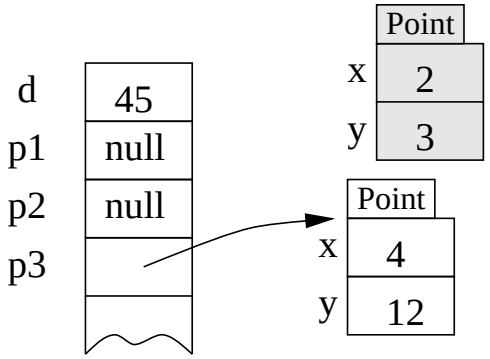
(r) Vue 8

Classes&Objets : exemple

```
// Vue 8

// affectation d'un variable contenant null

p2 = p1;

// Vue 9
```



(s) Vue 8



(t) Vue 9

Surcharge

- Principe selon lequel des opérations de même nom s'appliquent à des arguments de types différents.
- exemple : opérateur + surchargé : addition de nombres et concaténation de chaînes de caractères

Surcharge

- Il est possible de **surcharger** une **méthode** (pas les opérateurs)
- des méthodes surchargées sont des méthodes avec :
  1. même nom
  2. nombre de paramètres ou types des paramètres $\neq$
- exemple :
  1. int bidule(int x) // 1 paramètre
  2. int bidule(int x, int y) // 2 paramètre $\Rightarrow$ nombre de paramètres $\neq$ méthode 1 : autorisé
  3. int bidule(int x, float y) // 2 paramètres, type des paramètres $\neq$ méthode 2
  4. double bidule(int x) // NON ! même nombre et type des paramètres méthode 1
- appel:
  - ????.bidule(1) $\longrightarrow$ methode 1
  - ????.bidule(10,2) $\longrightarrow$ methode 2
  - ????.bidule(10,2.5) $\longrightarrow$ methode 3

this

- correspond dans une méthode d'instance à l'objet courant - l'instance sur laquelle se fait l'opération
- correspond dans un constructeur à l'objet courant - l'instance en cours de création
- comme **première** instruction d'un constructeur $\Leftrightarrow$ appel d'un autre constructeur
- exemple :

```
class Point
{
    int x,y;

    public Point(int x, int y)
    {this.x = x; this.y = y;}

    public Point()
    {this(0,0);}

    public int translater(int dx, int dy)
    {this.x = this.x + dx; this.y = this.y + dy;}
}
```

String

- les chaînes de caractères sont des objets `String`
- une `String` est **non modifiable** ⇒ recréation d'une nouvelle chaîne pour une modification
- création : `String maChaine = "Bonjour";`
- Beaucoup de méthodes utilitaires de conversion, formatage
- longueur : `length()`
- concaténation : opérateur `+` ou méthode `concat()`
- Seule classe avec un opérateur
- Notion de *string conversion* pour prendre en compte les expressions avec des `String` et des `+`
- ...

StringBuffer

- chaînes de caractères **modifiables**
- ajout de caractères `:append()`
- insertion de caractères `:insert()`
- **...**

Object

- ∃ classe `Object`
- méthodes : `toString()`, `equals()`, `clone()`, ...
- détail après l'héritage

Instanciation et tableaux

- Les tableaux sont des objets... (`java.lang.Object`)
- Déclaration : `T[] nomTab` ou `T nomTab[]`
- **pas** de dimensionnement à la déclaration
- Types possibles : tous les types java ; simples ou références
- Création et dimensionnement par :
  - `T[] unTableau = new T[20];`
  - `T unTableau[] = new T[20];`
  - expression d'initialisation statique : `String[] tabStr= {"un", "deux" };`
- Accès :

```
int [] tabInt = new int[10]
tabInt[1] = 12;
System.out.println(tabInt[1]);
```

Tableaux

- variable d'instance `public final :length`
- **Pas** de redimensionnement dynamique
- indexe :
  - notation `:[ expression ]`,
  - premier = 0
  - type `:int`
  - Validité vérifiée à l'exécution (`ArrayIndexOutOfBoundsException`)
- Copie :par `public clone`
- `char[]` **n'**est **pas** `String`

Tableaux multi-dimensionnels

- Tableau multi-dimensionnel ↔ tableau de tableaux
- Déclaration : `type nomTab []...[];`
- Dimensionnement à l'exécution
- Exemples :

```
int troisD [][][] = new int[3][5][2];
int deuxD [][] = new int[2][];
deuxD[0] = new int[3];
deuxD[1] = new int[2];
int tab [][] = {{1,2},{3,4,5},{6}};
```

Classes&Objets

- Les classes existent en tant qu'objet. L'appel de la méthode `getClass()` sur n'importe quelle instance renvoie une référence sur objet de classe `Class`.
- Plus exactement, la JVM crée un objet de classe `Class` pour chaque classe chargée.
- La description des classes est donc disponible à l'exécution.
  Exemple : nom de la classe par la méthode `getName()`

```
System.out.println("le classe de " + obj +
                   " est " + obj.getClass().getName());
```

- Mais il n'est pas possible de créer de nouvelles classes.
- Applications : debuggeurs, interpréteur, inspecteurs, *browser* de classes

Variables de classe

$\neq$ variable d'instance

$=$ variable partagée et accessible par n'importe quelle instance

$=$ variable partagée et accessible indépendamment d'une instance

$\Leftrightarrow$ c'est une variable de classe

Déclaration

- dans une classe (ça devient une variable de cette classe)
- en dehors de toute méthode
- clause **static**
- exemple :

```
public class MaClasse
{
   static int maVariable = 0;
...
```

Initialisation

- trois solutions :
  1. lors de la déclaration :

```
public class MaClasse
{
   static int maVariable = 0;
...
```

  2. dans un initialiseur statique.

```
public class MaClasse
{
   static int maVariable;

   static
   {
     maVariable = 0;
   }
...
```

  3. ailleurs mais c'est moyen

- l'initialisation se fait **UNE seule fois** au chargement de la classe

Variables de classe

```
class DemoStaticVar
{
    int varInstance = 0;
    static int varClasse = 0;
}

public class SimpleStaticDemo
{
    public static void main(String[] args)
    {
        DemoStaticVar o1 = new DemoStaticVar();
        o1.varInstance++;
        o1.varClasse++;
        System.out.println(o1.varClasse + " " + o1.varInstance);
        DemoStaticVar o2 = new DemoStaticVar();
        o2.varInstance++;
        o2.varClasse++;
        System.out.println(o1.varClasse + " " + o1.varInstance);
        System.out.println(o2.varClasse + " " + o2.varInstance);
        System.out.println(DemoStaticVar.varClasse); // MIEUX
        DemoStaticVar.varClasse++;
        System.out.println(DemoStaticVar.varClasse);
    }
}
```

```
1 1
2 1
2 1
2
3
```

Variables de classe

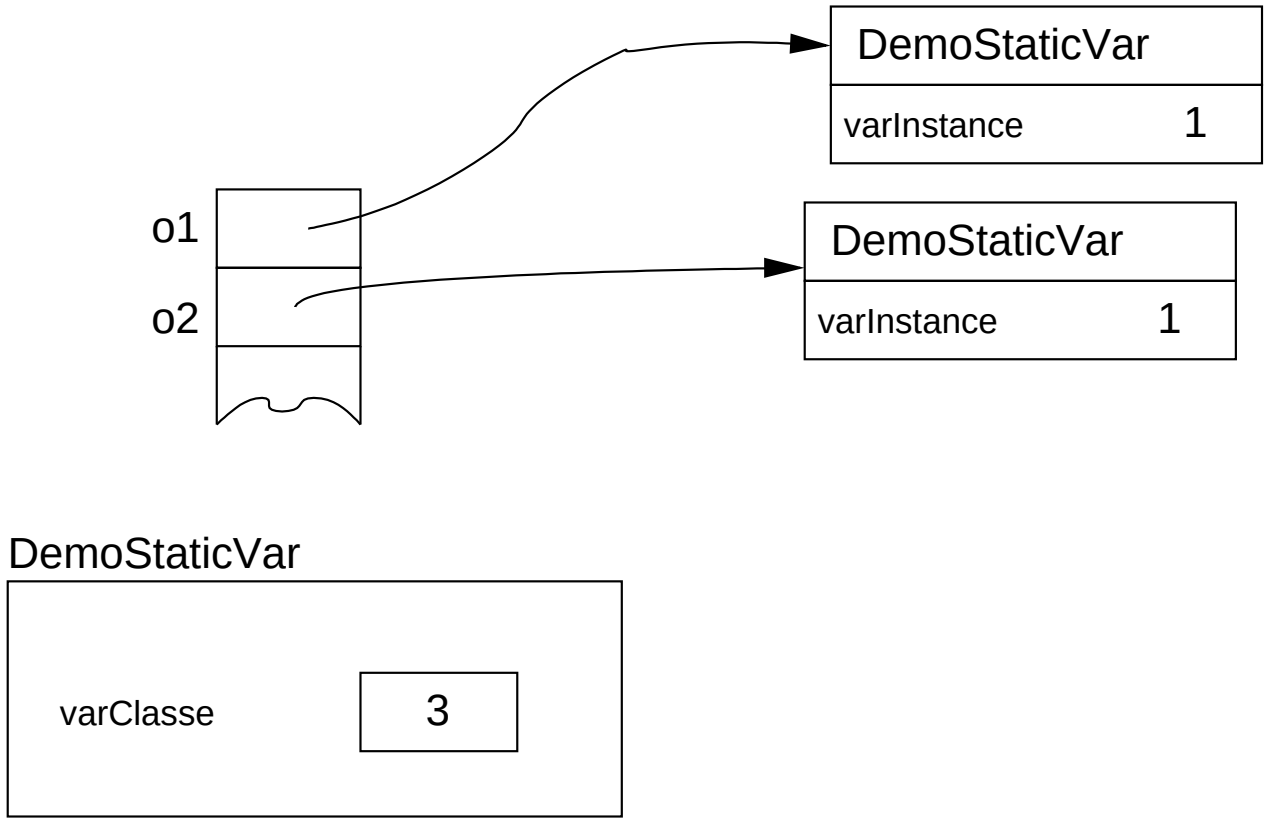


FIG. 1 – Schéma mémoire

Méthodes de classe

- déclarées par "`static`"
- méthodes qui peuvent être appelées indépendamment d'une instance.
-
- **PAS** de `this`
- **PAS** d'accès aux variables d'instances
- trois "sortes" de méthodes de classes :

  1. fonctions. Exemple : `Math.sqrt(10);`
  2. "symétrisation" de méthode d'instance de comparaison. Exemple :

     ```
     public boolean plusGrand(Personne autre)
     public static boolean plusGrand(Personne p1, Personne p2)
     ```

  3. manipulations de variables de classe

Méthodes de classe

```
class DemoStaticMeth
{
    static int varClasse=0;

    public int methInst()
    {
        return (varClasse);
    }
    public static int methClasse()
    {
        return (varClasse);
    }
}

public class SimpleStaticDemo2
{
    public static void main(String[] args)
    {
        DemoStaticMeth o1 = new DemoStaticMeth();
        System.out.println(o1.methInst());
        System.out.println(o1.methClasse());
        //// System.out.println(DemoStaticMeth.methInst());
        System.out.println(DemoStaticMeth.methClasse());
    }
}
```

```
0
0
0
```

```
SimpleStaticDemo2B.java:22: non-static method methInst() cannot be referenced fro
 System.out.println(DemoStaticMeth.methInst());
                                           ^
1 error
```

Constantes

- le modifieur "final" déclare qu'une variable est constante.
- cette variable doit être initialisée.
- exemple de la classe Integer de `java.lang.Integer` :

```
class Integer extends Number {
    public static final int MIN_VALUE = 0x80000000;
    public static final int MAX_VALUE = 0x7fffffff;
}
```

Langage : Plan

- Généralités
- Compilation & exécution
- Expressions, contrôle ...
- Classes & Objets
- **Héritage**
- Exceptions
- Encapsulation
- Interfaces
- Divers

Héritage : idée

- Simple (NON MULPTIPLE)
- Syntaxe : `class A extends B {....}`
- Sémantique :
  - A est une sorte de B
  - ou encore : un objet de classe A est aussi un objet de classe B
  - le contraire est bien entendu faux
- Classe "top level" : `Object`
  - Implicitement : `class A {...` $\Leftrightarrow$ `class A extends Object {...`
  - pas de super-classe
  - méthodes utilisables sur tous les objets

Héritage : mécanisme

- c'est l'équivalent d'une recopie du code de B dans A
- TOUTES les variables et méthodes de B sont accessibles dans A
- Ajout de variable(s) d'instance dans l'état d'un objet de la nouvelle classe
- Ajout de fonctionnalités à la super classe directe (ici B) $\Leftrightarrow$ ajout de nouvelles méthodes
- Modification de **l'implantation** de certaines fonctionnalités de la super classe $\Leftrightarrow$ redéfinition de méthodes (dans la sous classe, même signature, code $\neq$)
- **Pas** de modification de ce que doit faire une instance de la nouvelle classe en tant qu'instance de la super classe (pas de modification du "contrat" offert par la super classe)

Héritage : vocabulaire

- `class A extends B {....}`
- A hérite de B
- A étends B
- A spécialise B
- A est une sous classe de B
- B est LA superclasse de A
- A est un B
- B n'est pas un A

Héritage : exemple

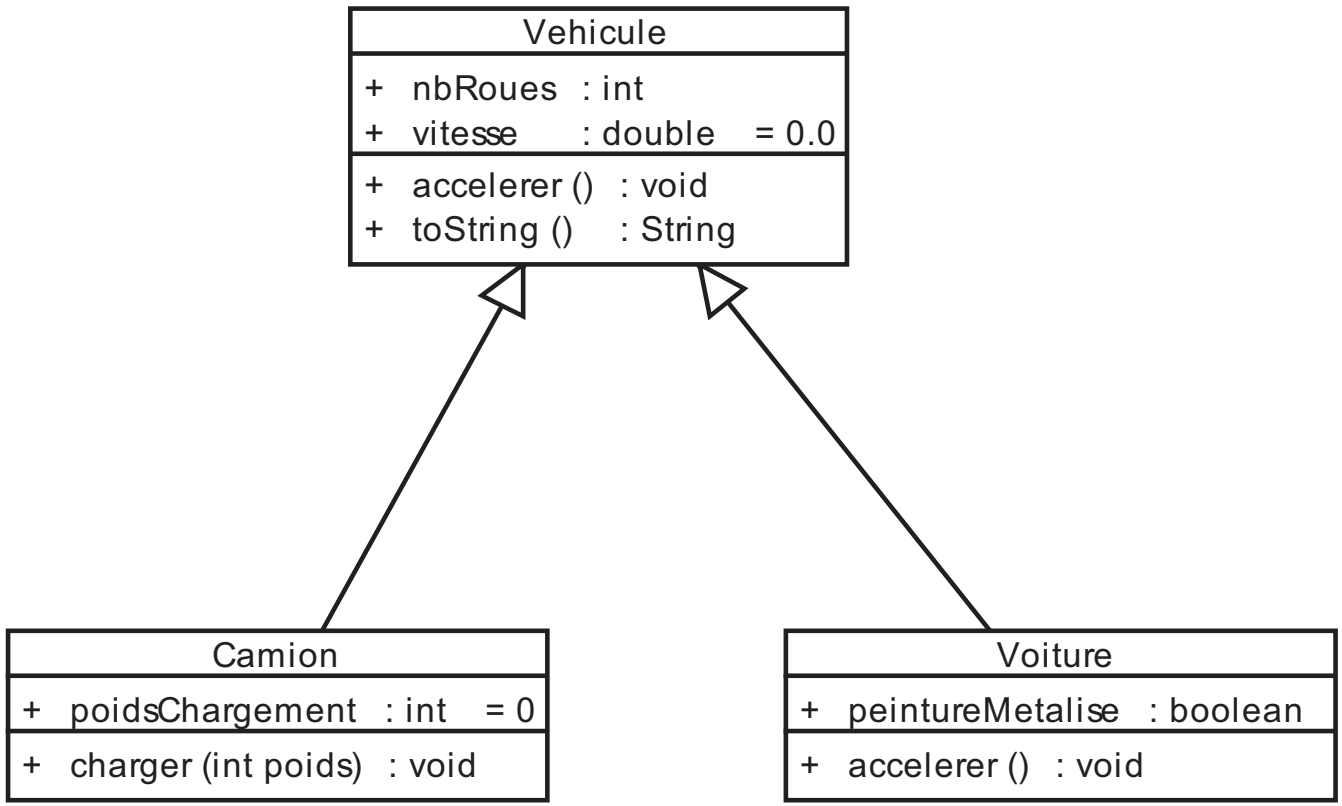


FIG. 2 – Graphe d'héritage

Héritage : exemple

```
public class Vehicule
{
    public int nbRoues;
    public double vitesse = 0.0;
    public void accelerer()
    {
        vitesse = vitesse + 10.0;
    }
    public String toString()
    {
        return (this.getClass().getName() +
                " vitesse " +vitesse);
    }
}
```

```
public class Voiture extends Vehicule
{
    /** ajout variable d'instance */
    public boolean peintureMetalise;
    /** redéfinition méthode accelerer*/
    public void accelerer()
    {
        vitesse = vitesse + 20.0;
    }
}
```

```
public class Camion extends Vehicule
{
    /** ajout variable d'instance */
    public int poidsChargement = 0;
    /** ajout méthode charger*/
    public void charger(int poids)
    {
        poidsChargement = poidsChargement + poids;
    }
}
```

Class Vehicule

java.lang.Object
  |
  +--**Vehicule**

Direct Known Subclasses:
Camion, Voiture

---

public class **Vehicule**
extends Object

Field Summary	
int	**nbRoues**
double	**vitesse**

Constructor Summary
Vehicule()

Method Summary	
void	**accelerer**()
String	**toString**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

nbRoues

```
public int nbRoues
```

---

vitesse

```
public double vitesse
```

Constructor Detail

Vehicule

```
public Vehicule()
```

Method Detail

accelerer

```
public void accelerer()
```

---

toString

```
public String toString()
```

Overrides:
toString in class Object

Class Camion

```
java.lang.Object
   |
   +--Vehicule
         |
         +--Camion
```

public class **Camion**
extends Vehicule

Field Summary

int	**poidsChargement**

Fields inherited from class Vehicule

nbRoues, vitesse

Constructor Summary

Camion()

Method Summary

void	**charger**(int poids) méthode charger

Methods inherited from class Vehicule

accelerer, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

poidsChargement

```
public int poidsChargement
```

Constructor Detail

Camion

```
public Camion()
```

Method Detail

charger

```
public void charger(int poids)
```

méthode charger

Class Voiture

```
java.lang.Object
   |
   +--Vehicule
         |
         +--Voiture
```

public class **Voiture**
extends Vehicule

Field Summary

boolean	**peintureMetalise**

Fields inherited from class Vehicule

nbRoues, vitesse

Constructor Summary

Voiture()

Method Summary

void	**accelerer**() méthode accelerer

Methods inherited from class Vehicule

toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

peintureMetalise

```
public boolean peintureMetalise
```

Constructor Detail

Voiture

```
public Voiture()
```

Method Detail

accelerer

```
public void accelerer()
```

méthode accelerer
Overrides:
accelerer in class Vehicule

Héritage et instanciation

```
public class TestVehicule
{
  public static void main(String[] args)
    {
      Vehicule saisPas, vehicule2, vehicule3;
      Voiture saxo;
      Voiture clio;
      Camion daf;

      // instanciations
      daf = new Camion();
      saxo = new Voiture();
      saisPas = new Vehicule();
      vehicule2 = new Voiture();
      vehicule3 = new Camion();
      //// clio = new Vehicule();

      System.out.println("saisPas " + saisPas);

      saisPas.accelerer();
      System.out.println("saisPas " + saisPas);

      System.out.println("daf " + daf);
      daf.accelerer();
      System.out.println("daf " + daf);

      System.out.println("saxo " + saxo);
      saxo.accelerer();
      System.out.println("saxo " + saxo);

      System.out.println("vehicule2 " + vehicule2);
      vehicule2.accelerer();
      System.out.println("vehicule2 " + vehicule2);

      //// vehicule3.charger();
    }
}
```

Héritage et instanciation

```
TestVehiculeB.java:16: incompatible types
found   : Vehicule
required: Voiture
       clio = new Vehicule();
              ^
TestVehiculeB.java:34: cannot find symbol
symbol  : method charger()
location: class Vehicule
       vehicule3.charger();
                 ^
2 errors
```

```
saisPas Vehicule vitesse 0.0
saisPas Vehicule vitesse 10.0
daf Camion vitesse 0.0
daf Camion vitesse 10.0
saxo Voiture vitesse 0.0
saxo Voiture vitesse 20.0
vehicule2 Voiture vitesse 0.0
vehicule2 Voiture vitesse 20.0
```

Héritage : `super`

- mot clé réservé
- équivalent de this
- sert à faire référence à la superclasse
- super() en première instruction fait appel au constructeur de la superclasse

Masquage de variables

```
class A
{ char x = 'a'; }

class B extends A
{ char x = 'b'; }

class C extends B
{ char x = 'c';
  public String bidon()
  {
    StringBuffer temp = new StringBuffer();
    temp.append("x de C " + x);
    temp.append("\nx de C " + this.x);
    temp.append("\nx de B " + super.x);
    temp.append("\nx de B " + ((B)this).x);
    temp.append("\nx de A " + ((A)this).x + "\n");
    //temp.append("\nx de A "+ super.super.x);
    return (new String(temp));
  }
}

public class TestMasquage
{
  public static void main(String args[])
  {
    C test = new C();
    System.out.println(test.bidon());
  }
}
```

```
x de C c
x de C c
x de B b
x de B b
x de A a
```

Héritage et constructeurs

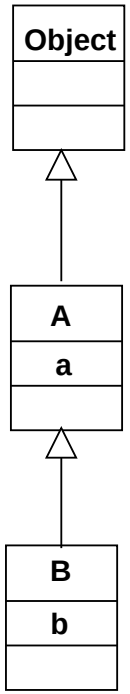
- Idée : l'état initial d'un objet est défini par l'appel de constructeurs. En cas de construction par héritage il faut une **discipline** d'appel.
- En java : **construction TOP-DOWN** : le code utile d'un constructeur de la superclasse sera exécuté **avant** celui du constructeur de la classe considérée.

Héritage et constructeurs : Mise en œuvre

Dans un constructeur d'une classe A :

- L'appel `super(...)` est utilisable en première ligne et invoque un constructeur de la superclasse ;
- puis initialisation des variables d'instance introduites par A ;
- Et exécution du reste du code du constructeur considéré
- Par défaut si pas d'appel à `this(..)` ou `super(...)` (en première ligne d'un constructeur) le compilateur introduit implicitement un appel à `super()`

Constructeur et héritage : Exemple




```
Object(){
  CODE 1
}
```

```
int a = 10;
A(int i){
  CODE 2
}
```

appel implicite à super()

```
int b = 5;
B(){
  super(2);
  CODE 3
}
B(int i){
  this();
  b=i;
  CODE 4
}
```

appel explicite à super()

```
            Appel de
            B(100) :

  this() => >B()
 super(2) => >>A(2)
  super() => >>>Object()
             >>>>CODE 1
exec init => >>>a=10;
             >>>CODE 2
exec init => >>b=5;
             >>CODE 3
             >b=100;
             >CODE 4
```

Constructeur et héritage : Exemple

- Remontée *Bottom-up* des appels à `super(...);`
- Mais exécution *Top-down* des initialisations et du code des constructeurs.

Héritage : remarques diverses

- Redéfinition d'une méthode $\Rightarrow$
  - l'accès ne doit pas être réduit
  - les exceptions spécifiées doivent être un sous ensemble de sous classes de celles indiquées dans la super classe
  - Idée : conservation du contrat de la superclasse.

Object

- Classe mère de toutes les classes
- Maintenant intéressant de détailler ses méthodes

toString

- pour obtenir une description textuelle d'un objet
- signature :

```
public String toString()
```

- retourne par défaut :

```
getClass().getName() + "@" + Integer.toHexString(hashCode());
```

- TRÈS conseillé de (re)définir cette méthode dans toute classe

Test d'égalité

- pour comparer deux objets
- signature :

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

- retourne par défaut :

```
(this == obj)
```

- on peut (re)définir cette méthode dans toute classe

“Destructeur”

- Rappel : mécanisme de ramasse-miette (*Garbage Collector*)
- $\Rightarrow$ pas de vrai destructeur
- $\exists$ méthode appelée par le ramasse-miette avant désalocation des ressource :

  `void finalize() throws Throwable`
- sert à libérer sockets, descripteur de fichiers, ...

```
public class Finalize
{
    public void finalize() throws Throwable
    {
        super.finalize();
        // code de finalization de cet objet
    }
}
```

“Destructeur”

```
public class Finalize2
{
    public Finalize2()
    {
        System.out.println("Contructeur");
    }

    public void finalize() throws Throwable
    {
        super.finalize();
        System.out.println("Finalise");
    }

    public static void main(String[] args)
    {
        Finalize2 f = new Finalize2();
        f = null;
        System.gc(); // pour appeler explicitement le Garbage Collector
    }
}
```

Gestion de threads

- `public final native void notify()`
- `public final native void notifyAll()`
- `public final native void wait(long timeout) throws InterruptedException`
- `public final void wait() throws InterruptedException`
- `public final void wait(long timeout, int nanos) throws InterruptedException`

Le reste

- `public final native Class getClass()`
- `public native int hashCode()`
- `protected native Object clone() throws CloneNotSupportedException`

Classe abstraite

- Pour définir une classe forcément superclasse d'une autre pour être utile
- Autrement dit : doit être sous classée pour être utile
- Classe **incomplète**. Non instanciable.
- Syntaxe `abstract class A ....`
  - Défaut : **NON** `abstract`
- On peut parfaitement sous classer une classe `abstract` par une autre classe `abstract`

Classe finale

- Ne peut pas être superclasse d'une autre
- Syntaxe : `final class A ...{...}`
- Sémantique : si un objet est une sorte de A alors c'est un A
- Défaut : **NON** `final`

Méthode abstraite

- Déclaration d'une méthode sans l'implanter.
- Syntaxe : `abstract Type nomMeth(..);` dans le corps d'une classe (disons `A`)
  - Défaut : **NON** `abstract`
- Sémantique : on définit ce que doit faire un `A` mais en fait c'est en considérant une sorte plus spécialisée que l'on obtiendra une action réelle. Ex :
  - méthode `getSurface()` dans une classe `Polygone`
  - base*hauteur/2 pour un `Triangle`, cote*cote pour un `Carre`, ...
- méthode `abstract` $\Rightarrow$ la classe doit être déclarée en `abstract`. En effet la méthode ne pourra être implantée qu'en sous classant `A`. Implicitement `A` est donc abstraite : et bien disons-le.
- incompatible avec `private`, `static`, `final`

Méthode finale

- Une sous classe ne pourra pas redéfinir la méthode
- Syntaxe : `final Type nomMeth(..){CODE}` dans le corps d'une classe (disons A)
  - Défaut : **NON** `final`
- Sémantique : on fixe une fois pour toute un comportement de la sorte A
- Incompatible avec `abstract`

Langage : Plan

- Généralités
- Compilation & exécution
- Expressions, contrôle ...
- Classes & Objets
- Héritage
- **Exceptions**
- Encapsulation
- Interfaces
- Divers

Exception : généralités

- Conditions exceptionnnelles détournant le programme de son exécution normale
- deux possibilités :

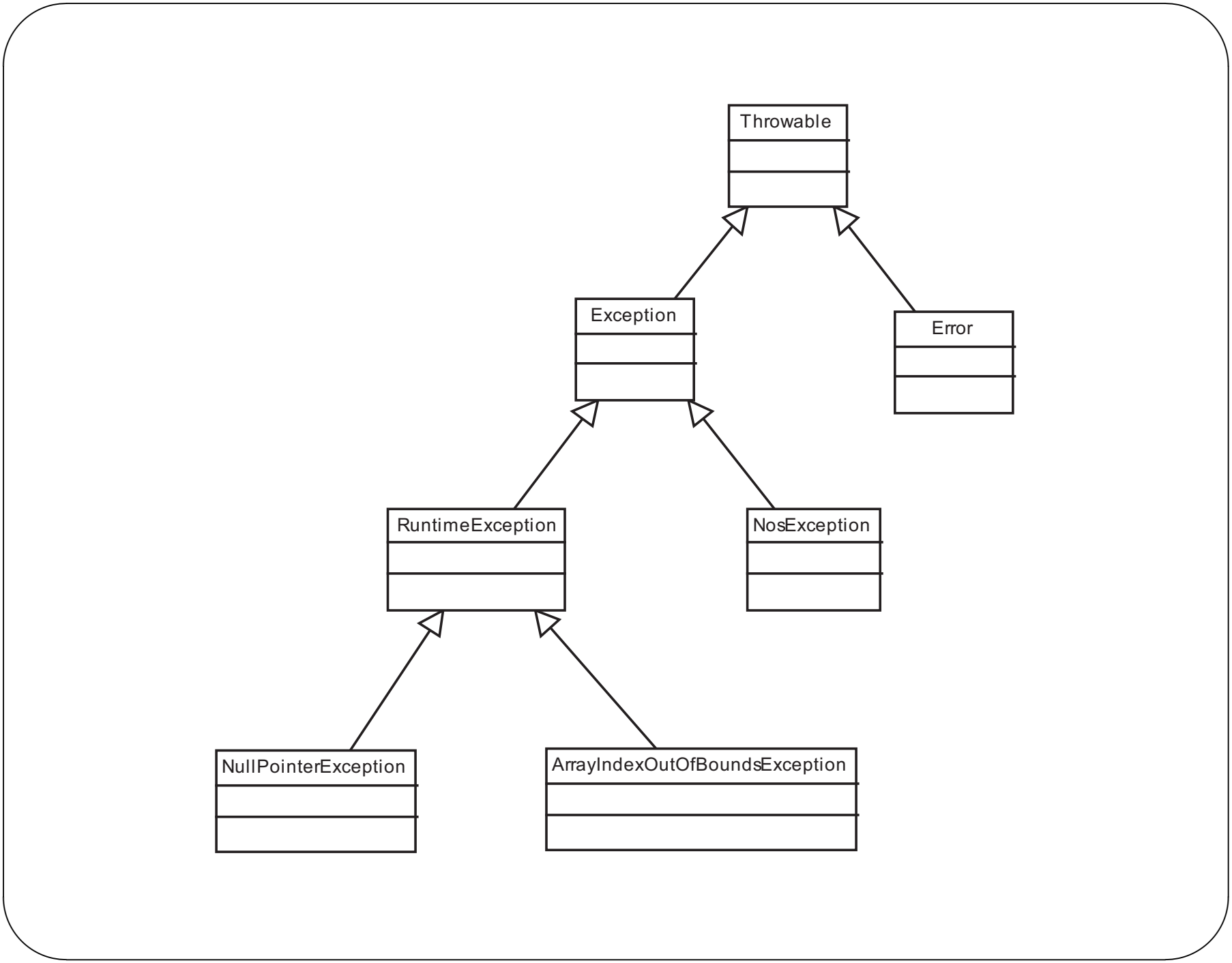
  lancer une exception = la signaler : clause `throw`

  capturer une exception = la traiter : clause `try/catch/finally`

- Spécification dans le profil des méthodes

Exceptions : Objets

- Les exceptions sont des objets : classe `Throwable` :
  - sous-classes à partir de `Exception` pour utilisateur
  - sous-classe à partir de `Error` pour JVM, système


Throwable
Exception
Error
RuntimeException
NosException
NullPointerException
ArrayIndexOutOfBoundsException

Exception : capture

```
public class Exception1
{
  public static void main (String args[])
  {
    try
      {
          String s = null;
          s.toUpperCase();
      }
    catch (NullPointerException e)
      {
        System.out.println("exception : " + e.getMessage());
        e.printStackTrace();
      }
  }
}
```

```
exception: null
java.lang.NullPointerException
        at Exception1.main(Exception1.java:8)
```

Exception : capture

- comment savoir ce qui peut provoquer des exceptions ?
- regarder les clauses `throws` des signatures des méthodes !

Socket

```
public Socket(String host,
              int port)
       throws UnknownHostException,
              IOException
```

Creates a stream socket and connects it to the specified port number on the named host.

Parameters:

`host` - the host name.
`port` - the port number.

Throws:

`UnknownHostException` - if the IP address of the host could not be determined.
`IOException` - if an I/O error occurs when creating the socket.

Exception : capture

- Que ce passe-t-il si on omet de capturer les exceptions signalés dans les clauses `throws` ?

```
import java.net.*;

public class DemoNoTry
{
    public static void main(String args[])
    {
        //// Socket sk = new Socket("TURING-PDC", 1025);
    }
}
```

```
DemoNoTry.java:7: unreported exception java.net.UnknownHostException;
must be caught or declared to be thrown
        Socket sk = new Socket("TURING-PDC", 1025);
                    ^
1 error
```

Exception : définition

- on définit un objet qui hérite de `Exception`
- on écrit les constructeurs

```
class DivisionParZeroException extends Exception
{
  public DivisionParZeroException(String s)
  {super(s);}
}
```

Exception : lancement

```
public class Exception2
{
  public int div(int a, int b) throws DivisionParZeroException
  {
    if (b==0)
      throw new DivisionParZeroException("Division par Zero");
    else
      return(a/b);
  }// fin div

  public static void main (String args[])
  {
    Exception2 demo = new Exception2();

    for(int i=1;i>=0;i--)
      {
        System.out.println(" i = " + i);
        try
          {demo.div(1,i);}
        catch (DivisionParZeroException e)
          {System.out.println("Exception : " + e.getMessage());}
        catch (Exception e)
          {System.out.println("Autre exception");}
        finally
          {System.out.println("Finally");}
      }
  }// fin main
}// fin class Exception2
```

Exception : lancement

```
 i = 1
Finally
 i = 0
Exception: Division par Zero
Finally
```

- obligation de déclarer les exceptions que peut générer une méthode
- se fait par la clause `throws`
- on déclare tout SAUF les `RuntimeException`

Exception : capture ; solutions

Deux solutions à l'appel d'une méthode avec une clause `throws UneException` :

- capturer et traiter (`try/catch/finally`)
- repasser l'exception à l'appelant : `throws UneException`

```
public class Exception3
{
    public static void main (String args[]) throws DivisionParZeroException
    {
        Exception2 demo = new Exception2();
        for(int i=1;i>=0;i--)
        {
            System.out.println(" i = " + i);
            demo.div(1,i);
        }
    }// fin main
}// fin class Exception3
```

Exception : avantages

- Avantages :
  - Séparer cas correct/traitement des erreurs $\Rightarrow$ "l'algo apparait mieux"
  - Le code pour "filtrer" : + simple, + clair
    - diminution du code jusqu'à -400% du code rapport à des choses à la ==-1 UNIX
- Mais il ne faut pas rêver :
  - Prévoir ce qu'on veut traiter (mettre des `try`) !
  - Code pour traiter les erreurs (blocs des `catches`) !
  - Prendre certaines (mauvaises ?) habitudes ...

`throws` et héritage

- **Problème :** Si deux déclarations de méthode de classe (ou interface) différentes mais de même signature se cachent, ou recouvrent (overridding) : Que dire de leurs éventuelles clauses `throws` ?
- **Idée :** Respect du contrat de la superclasse.
- **Dans ce cas = :** Du code écrit pour prendre en compte l'appel et les exceptions levées pour la méthode de la superclasse, doit **aussi** fonctionner si la méthode effectivement atteinte est celle de la sous classe.
- **Conséquence** La clause `throws` de la méthode de la sous classe ne doit lever que des exceptions individuellement **sous classe** de celles spécifiées dans la superclasse.

Langage : Plan

- Généralités
- Compilation & exécution
- Expressions, contrôle ...
- Classes & Objets
- Héritage
- Exceptions
- **Encapsulation**
- Interfaces
- Divers

Encapsulation : idée

- **Objet (instance) = état + opérations (méthodes)**
- état = ensemble des variables d'instance.
- l'état d'une instance doit être **uniquement accédé par celle-ci**
- l'utilisateur d'un objet ne peut modifier cet état que via certaines opérations "légales" ou "publiques"

Chaîne de caractères : String

interface "encapsulée" de l'objet

creation(s)

length()

charAt()

toUpperCase()

```
private char value[];
private int offset;
private int count;
```

Utilisateur
= tout code qui ne s'exécute pas pour l'instance considérée

programmeur (de la classe) :
- **données concrêtes**
- **manipulation privée de ces données**

Paquetages

- notion introduite en Ada $\simeq$ 1980
- ensemble de classes "travaillant" sur le même domaine
- toutes les classes font parties d'un paquetage (un seul)
- les paquetages permettent de :
  1. regrouper syntaxiquement des classes proches conceptuellement
  2. définir des niveaux de protections pour les variables ou méthodes
  3. référencer chaque champ (variable ou méthode) par un nom complet:

     ```
     nomPaquetage.nomClasse.nomVariable
     nomPaquetage.nomClasse.nomMethode()
     ```

  4. Générer des espaces de nommage pour éviter les conflits de noms.

Paquetages

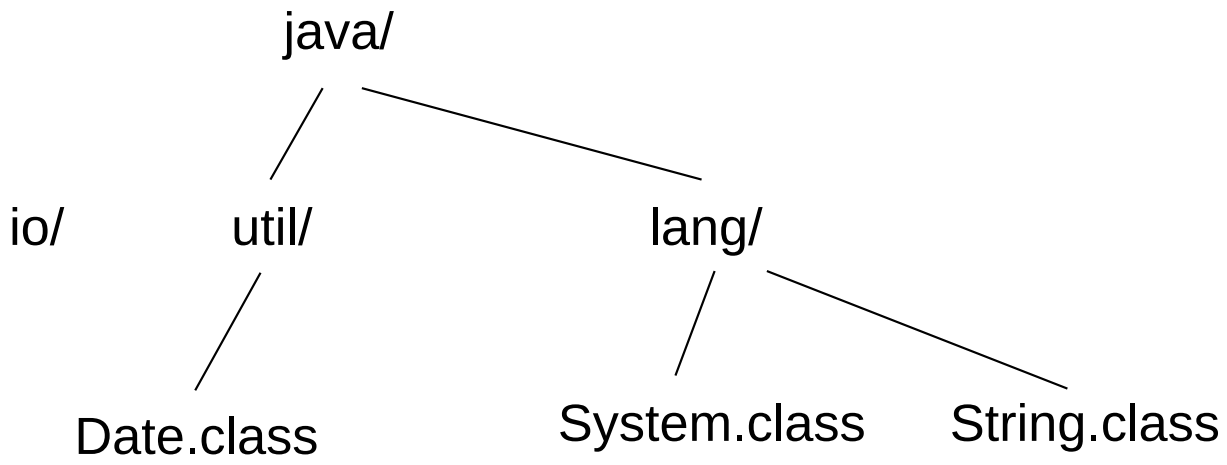
- le nom d'un paquetage peut être composé :

  `p1.p2.p2`

- le chemin complet du fichier sera :

  `p1/p2/p2`

- paquetages de l'API Java



Instruction package

- indique le paquetage auquel appartient le code du fichier
- Syntaxe : `package NomPaquetage;`

$\Rightarrow$ le nom complet d'une classe du fichier est abrégé `nomPaquetage.NomClasse`.

- en **première ligne** d'un fichier source
- en cas d'omission $\Leftrightarrow$ paquetage par défaut

Instruction `import`

- permet de faire référence à des classes à partir de leur noms "abrégés"
- Syntaxe : `import nomPaquetage.NomClasse;` permet d'utiliser directement le nom `NomClasse` dans la suite du fichier `.java`
- exemple : `import java.util.Date;`
- `import nomPaquetage.*;` : permet d'utiliser directement tout nom de classe `public` de `nomPaquetage` dans la suite du fichier `.java`
- exemple : `import java.io.*;`

Encapsulation : Accès aux classes

- une définition de classe peut être préfixée par le modifieur `public`
- une classe d'un paquetage est accessible par toutes les autres du paquetage
- une classe déclarée publique est accessible par les classes des autres paquetages
- une classe non déclarée publique n'est pas accessible par les classes des autres paquetages

Encapsulation : Accès aux classes

```
package PaquetageA;

public class Publique { ; }
```

```
package PaquetageA;

class Privee { ; }
```

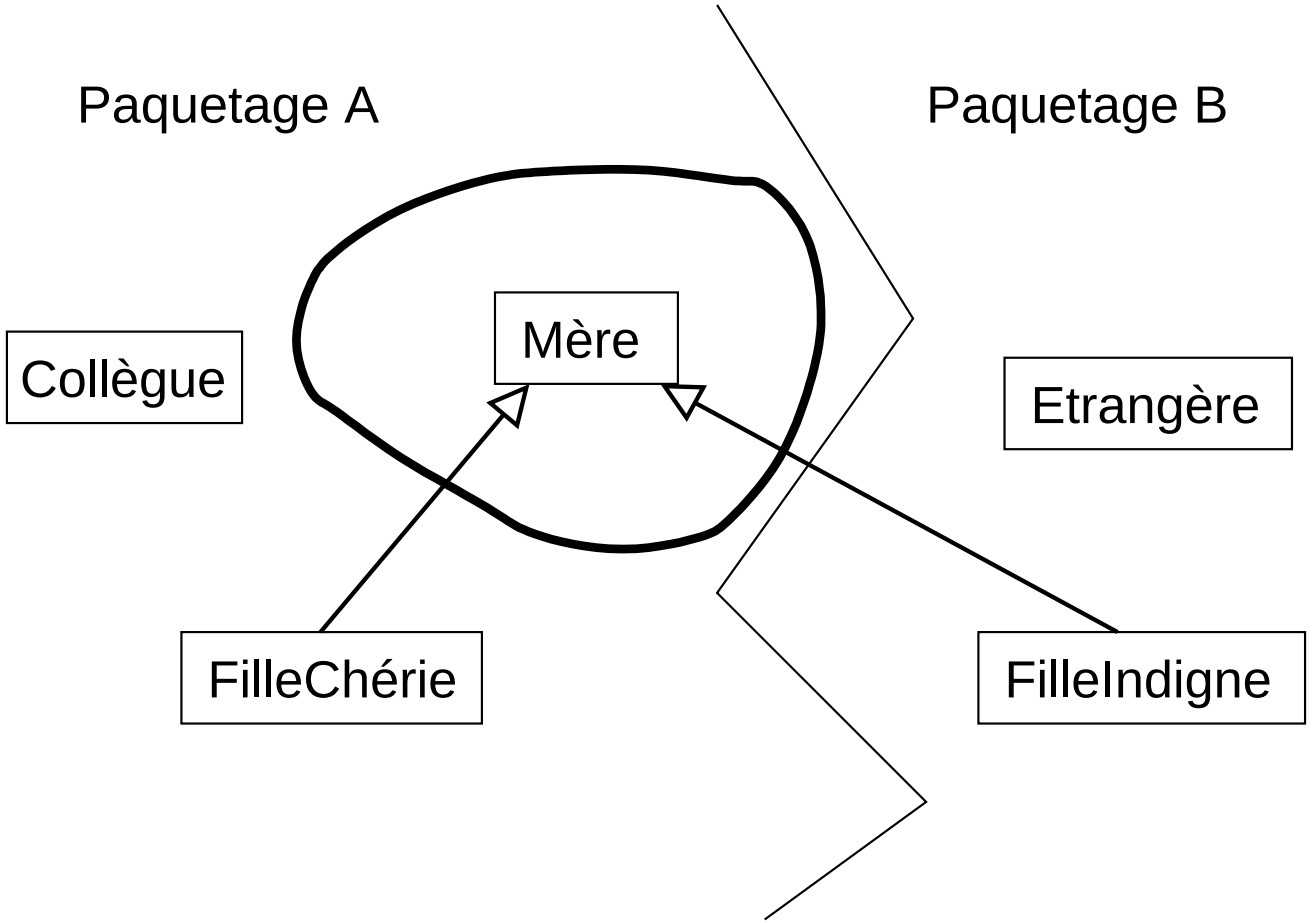
```
package PaquetageB;
import PaquetageA.*;

class Client
{
  Publique pu;
  //// Privee pr;
}
```

```
ClientB.java:7: PaquetageA.Privee is not public in PaquetageA; cannot be accessed from outside package
   Privee pr;
   ^
1 error
```

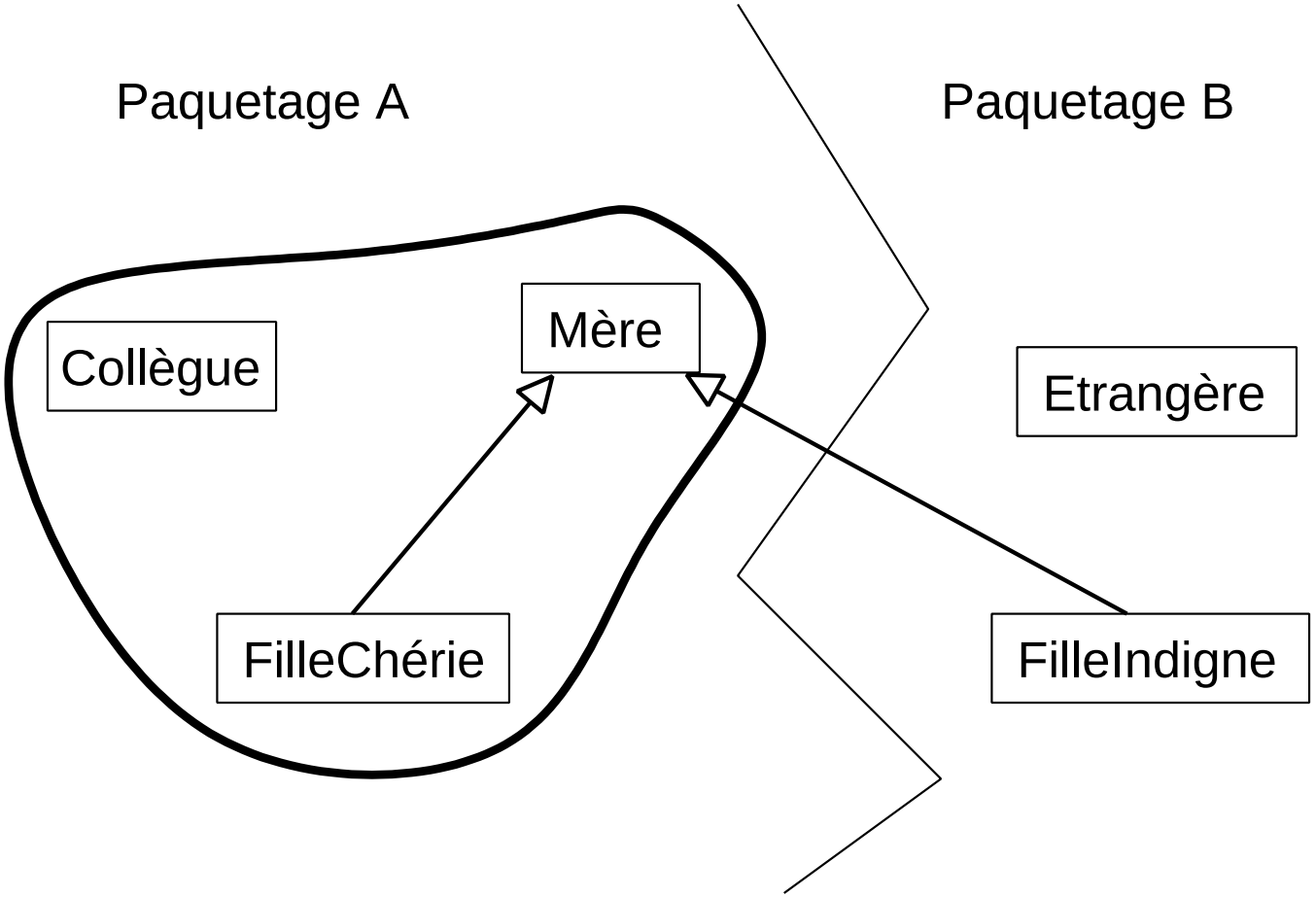
Encapsulation : Accès aux variables et méthodes

- une définition de méthode ou une déclaration de variable peut être préfixée par `:public`, `protected` ou `private`
- en fait il existe **4** niveaux de protections (3 + défaut) :
  1. `public`
  2. `protected`
  3. défaut
  4. `private`

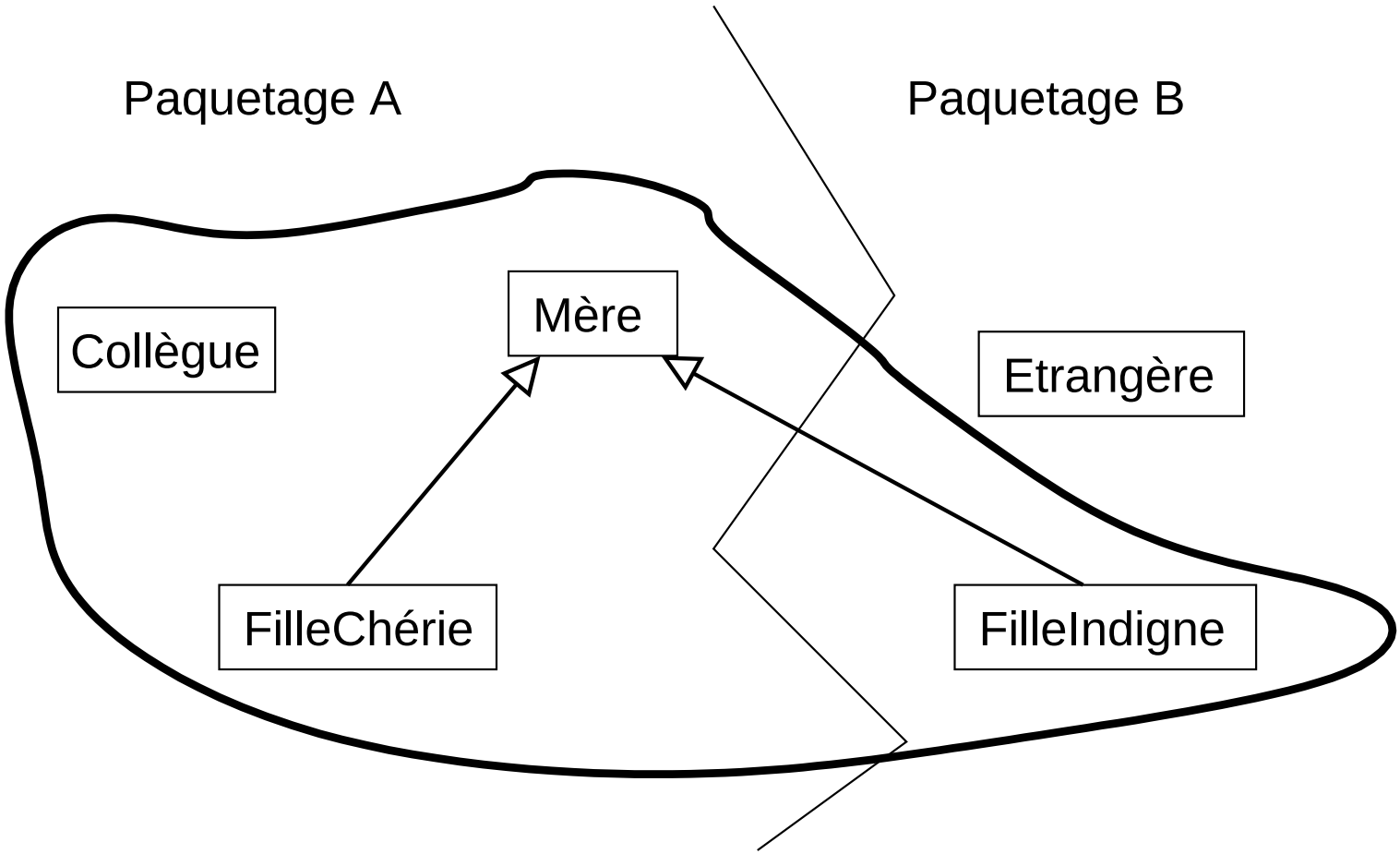
Encapsulation : Accès `private`



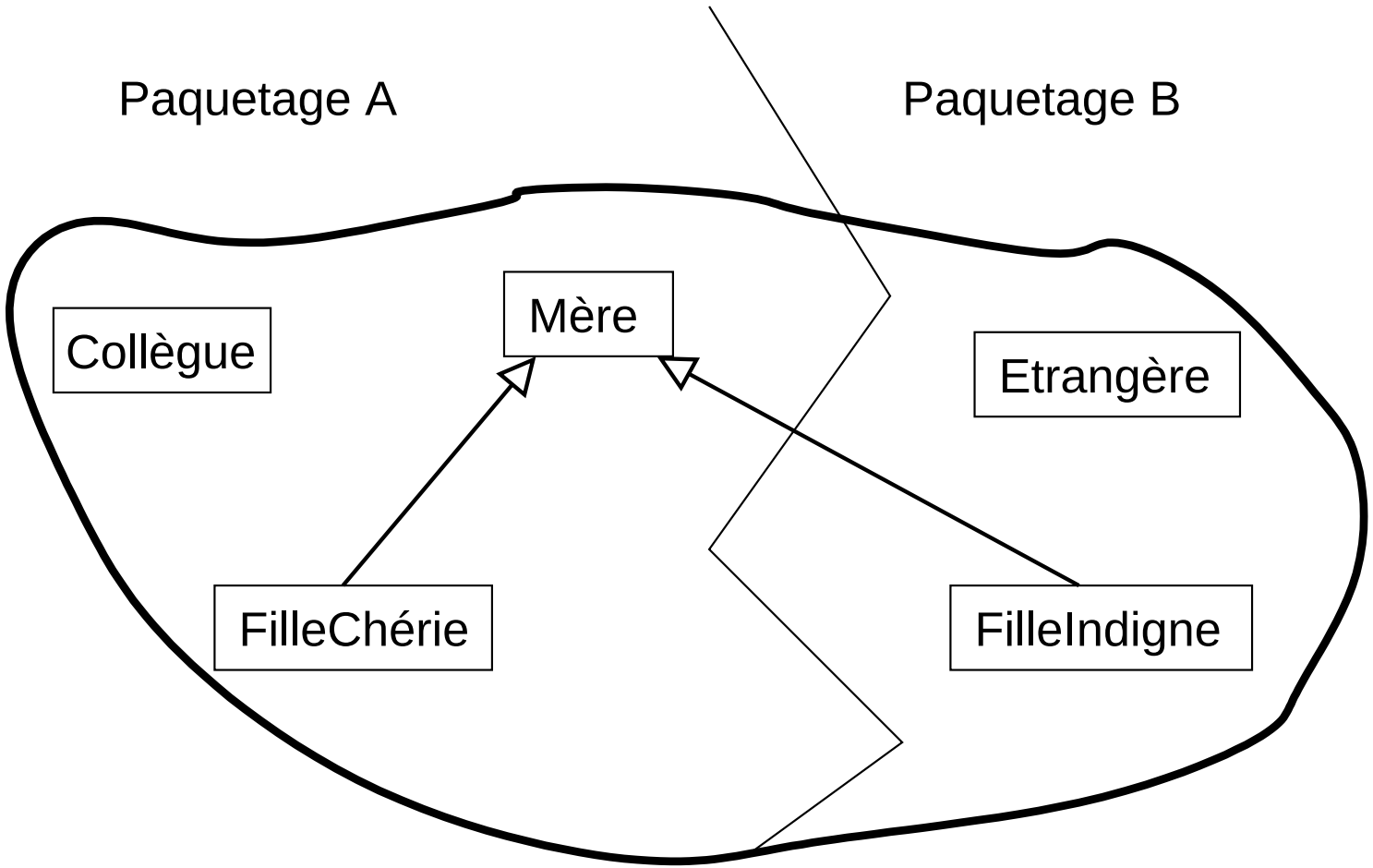
Encapsulation : Accès par défaut



Encapsulation : Accès `protected`



Encapsulation : Accès `public`



Encapsulation : Accès par une sous classe du même paquetage

```
package PaquetageA;

public class Mere
{
  private int attPrivate = 0;
  int attDefaut = 0;
  protected int attProtected = 0;
}
```

```
package PaquetageA;

class FilleCherie extends Mere
{
  void accesMere(Mere m)
  {
      //// m.attPrivate = 0;
    m.attDefaut = 0;
    m.attProtected = 0;
  }

  void accesFilleCherie()
  {
      //// attPrivate = 0;
    attDefaut = 0;
    attProtected = 0;
  }
}
```

```
FilleCherieB.java:7: attPrivate has private access in PaquetageA.Mere
       m.attPrivate = 0;
         ^
FilleCherieB.java:14: attPrivate has private access in PaquetageA.Mere
```

```
        attPrivate = 0;
        ^
2 errors
```

Encapsulation : Accès par une classe du même paquetage

```
package PaquetageA;


public class Mere
{
  private int attPrivate = 0;
  int attDefaut = 0;
  protected int attProtected = 0;
}
```

```
package PaquetageA;

class Collegue
{
  void accesMere(Mere m)
  {
      //// m.attPrivate = 0;
    m.attDefaut = 0;
    m.attProtected = 0;
  }
}
```

```
CollegueB.java:7: attPrivate has private access in PaquetageA.Mere
       m.attPrivate = 0;
         ^
1 error
```

Encapsulation : Accès par une sous classes d'un autre paquetage

```
package PaquetageB;
import PaquetageA.*;
class FilleIndigne extends Mere
{
  void accesMere(Mere m)
  {
      //// m.attPrivate = 0;
      //// m.attDefaut = 0;
      //// m.attProtected = 0;
  }
  void accesFilleIndigne()
  {
      //// attPrivate = 0;
      //// attDefaut = 0;
      attProtected = 0;
  }}
```

```
FilleIndigneB.java:7: attPrivate has private access in PaquetageA.Mere
       m.attPrivate = 0;
         ^
FilleIndigneB.java:8: attDefaut is not public in PaquetageA.Mere; cannot be accessed from outside package
       m.attDefaut = 0;
         ^
FilleIndigneB.java:9: attProtected has protected access in PaquetageA.Mere
       m.attProtected = 0;
         ^
FilleIndigneB.java:13: attPrivate has private access in PaquetageA.Mere
       attPrivate = 0;
       ^
FilleIndigneB.java:14: attDefaut is not public in PaquetageA.Mere; cannot be accessed from outside package
```

```
        attDefaut = 0;
        ^
5 errors
```

Encapsulation : Accès par une classe d'un autre paquetage

```
package PaquetageA;

public class Mere
{
  private int attPrivate = 0;
  int attDefaut = 0;
  protected int attProtected = 0;
}
```

```
package PaquetageB;

import PaquetageA.*;

class Etrangere
{
  void accesMere(Mere m)
  {
      //// m.attPrivate = 0;
      //// m.attProtected = 0;
      //// m.attDefaut = 0;
  }
}
```

```
EtrangereB.java:8: attPrivate has private access in PaquetageA.Mere
       m.attPrivate = 0;
         ^
EtrangereB.java:9: attProtected has protected access in PaquetageA.Mere
       m.attProtected = 0;
         ^
EtrangereB.java:10: attDefaut is not public in PaquetageA.Mere; cannot be accessed from outside package
       m.attDefaut = 0;
```

```
         ^
3 errors
```

Encapsulation : Résumé des accès

	private	défaut	protected	public
Mere	O	O	O	O
FilleChérie	N	O	O	O
Collegue	N	O	O	O
FilleIndigne	N	N	R	O
Etragere	N	N	N	O

Encapsulation : mise en œuvre Java

- Problème : langage de **classes**
- Règles de programmation : *Data Hiding*
  - ecrire des méthodes d'accès aux variables d'instance
  - les utiliser même si l'utilisateur est une instance de la même classe
- Contrôle d'accès aux champs et méthodes d'une classe :
  - mettre en `private` ou `protected` toute variable d'instance
  - mettre en `public` l'interface "légale"
  - mettre en `private` les méthodes "utilitaires" pour l'implémentation de l'interface légale
- Attention en Java l'accès par défaut n'est **pas** `private`

Langage : Plan

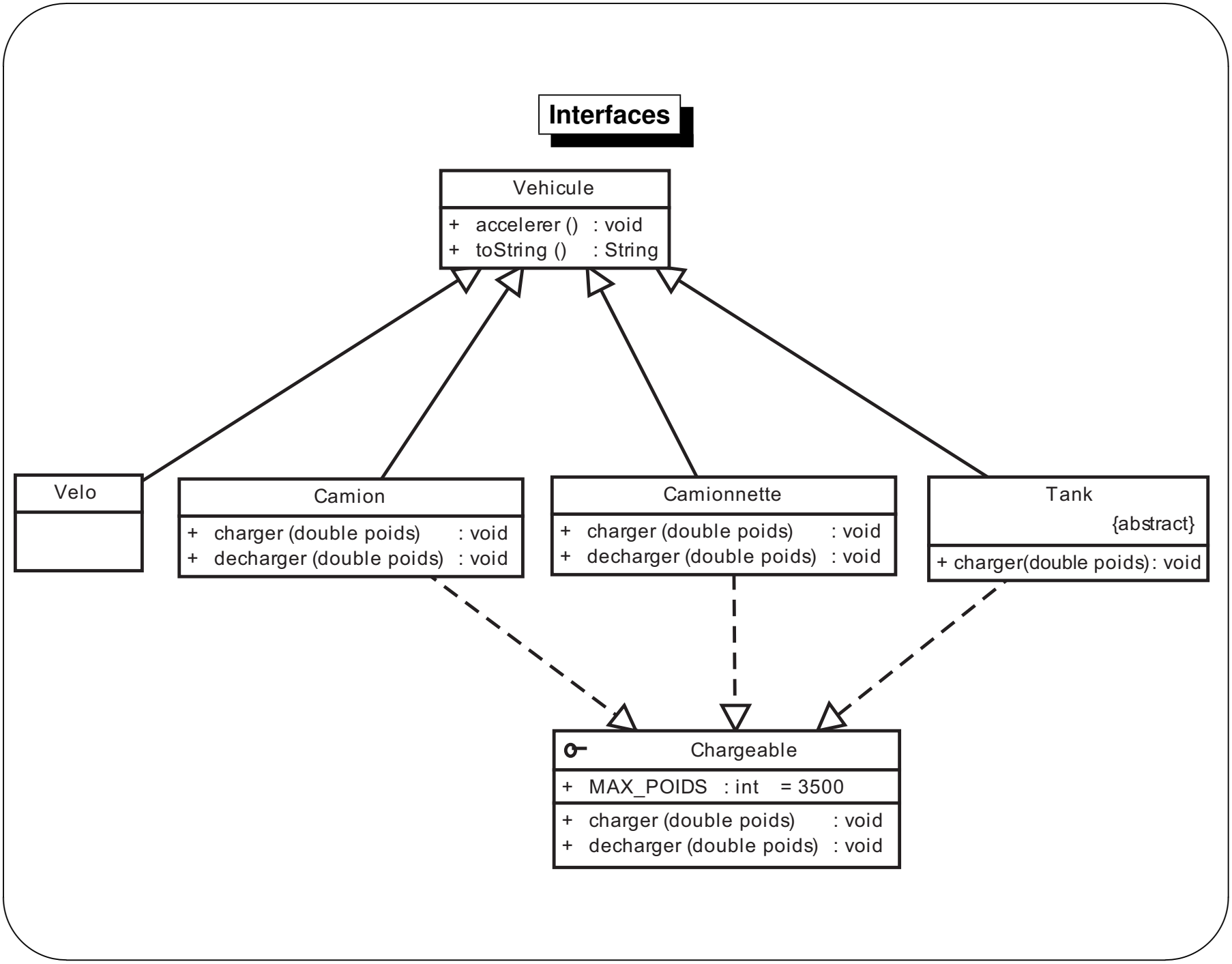
- Généralités
- Compilation & exécution
- Expressions, contrôle ...
- Classes & Objets
- Héritage
- Exceptions
- Encapsulation
- **Interfaces**
- Divers

Interfaces

- notion apparue dans Objective C
- une interface spécifie un comportement attendu
- une classe **implante** l'interface
- interface = forme restreinte de classe abstraite
  - Elle ne peut être instanciée.
  - Pas de constructeurs
  - Pas de méthodes non abstraites

  $\Longrightarrow$ Toute méthode est publique et abstraite (même sans déclaration explicite)
  - Pas de variables d'instances

  $\Longrightarrow$ Toute variable est constante (de classe) (même sans utilisation de `final static`)


Interfaces
Vehicule
+ accelerer () : void
+ toString () : String
Velo
Camion
+ charger (double poids) : void
+ decharger (double poids) : void
Camionnette
+ charger (double poids) : void
+ decharger (double poids) : void
Tank
{abstract}
+ charger(double poids): void
Chargeable
+ MAX_POIDS : int = 3500
+ charger (double poids) : void
+ decharger (double poids) : void

Exemple

```
public interface Chargeable
{
    public int MAX_POIDS = 3500;

    public void charger(double poids);
    public void decharger(double poids);
}
```

```
public class Camion extends Vehicule implements Chargeable
{
  public double poidsChargement = 0.0;
  /** méthode charger */
  public void charger(double poids)
  {
    poidsChargement = poidsChargement + poids;
  }
  /** méthode décharger */
  public void decharger(double poids)
  {
    poidsChargement = poidsChargement - poids;
  }
}
```

Interfaces&classes

- héritage simple
- par contre, implantation possible de plusieurs interfaces.
- implémentation de toutes les méthodes définies dans les interfaces.
- sinon classe abstraite.

```
//-> public class Tank extends Vehicule implements Chargeable
public abstract class Tank extends Vehicule implements Chargeable
{
    /** méthode charger */
    public void charger(double poids)
    {
        // corps de la méthode de chargement
    }

    /** pas de déchargement */
}
```

```
TankB.java:1: TankB should be declared abstract; it does not define decharger(double) in TankB
public class TankB extends Vehicule implements Chargeable
       ^
1 error
```

Interfaces&Types

- on peut manipuler une interface comme un type ordinaire

```
public class TestVehicule
{
    public void transvaser(Chargeable c1, Chargeable c2)
    {
        //
    }

    public static void main(String[] args)
    {
        Chargeable[] transporteur = new Chargeable(2);
        transporteur[0] = new Camion();
        transporteur[1] = new Camionnette();
        for (int i=0; i<transporteur.length;i++)
            {
                transporteur[i].charger(100);
            }
    }

}
```

Héritage d'interfaces

- Une interface peut hériter d'une autre interface

```
public interface ChargeableConstante
{
    public int MAX_POIDS = 3500;
}
```

```
public interface Chargeable2 extends ChargeableConstante
{
    public void charger(double poids);
    public void decharger(double poids);
}
```

Héritage multiple d'interfaces

- on peut faire de l'héritage multiple d'interfaces

```
public interface ChargeableConstante
{
    public int MAX_POIDS = 3500;
}
```

```
public interface ChargeableMethode
{
    public void charger(double poids);
    public void decharger(double poids);
}
```

```
public interface Chargeable3 extends ChargeableConstante, ChargeableMethode
{
}
```

Interfaces standards (utiles)

- `Cloneable`
- `Serializable,Externalizable` $\Longrightarrow$ `E/S`
- `Runnable` $\Longrightarrow$ `Threads`
- `Comparable,java.util.Comparator`

Cloneable

- interface sans méthode
- marqueur pour autoriser l'utilisation (et la redéfinition) `Object.clone()`
- sinon `CloneNotSupportedException`

Exemple

```
public class Clone_1 implements Cloneable
{
    CloneChamps champs;
    int entier;

    public Clone_1(int i)
    {
        champs = new CloneChamps(i);
        entier = i;
    }

    public Object clone() throws CloneNotSupportedException
    {
        return ((Clone_1)super.clone());
    }

    public String toString()
    {
        return (champs + " " + entier);
    }
}
```

```
public class CloneChamps
{
    public int val;

    public CloneChamps(int i)
    {
        val = i;
    }

    public String toString()
    {
        return (" * " + val);
    }
}
```

Exemple

```
public class Clone_2 implements Cloneable
{
    CloneChamps champs;
    int entier;

    public Clone_2(int i)
    {
        champs = new CloneChamps(i);
        entier = i;
    }

    public Object clone()
    {
        Clone_2 resultat = null;
        try
        {
            resultat = (Clone_2) super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        resultat.champs = new CloneChamps(champs.val);
        resultat.entier = entier;
        return resultat;
    }

    public String toString()
    {
        return(champs + " " + entier);
    }
}
```

Exemple

```
public class TestClone
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Clone_1 c1, c1c;
        Clone_2 c2, c2c;

        c1 = new Clone_1(1);
        c2 = new Clone_2(2);

        System.out.println(c1);
        System.out.println(c2);

        c1c = (Clone_1)c1.clone();
        c2c = (Clone_2)c2.clone();

        System.out.println(c1c);
        System.out.println(c2c);

        c1c.champs.val = 3;
        c2c.champs.val = 3;

        System.out.println(c1);
        System.out.println(c2);

    }
}
```

```
* 1 1
* 2 2
* 1 1
* 2 2
* 3 1
* 2 2
```

Comparable / Comparator

- permet de définir un ordre total sur des objets
- `Comparable` de manière interne à la classe ordonnée
- `Comparator` de manière externe à la classe ordonnée
- `Comparable` :

```
public int compareTo(Object o)
```

- `Comparator` :

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

Exemple

```
public class Etudiant implements Comparable<Etudiant>
{
    private double note;
    private String nom;

    public Etudiant(String nom, double note)
    {
        this.nom = nom;
        this.note = note;
    }

    public int compareTo(Etudiant e)
    {
        int res = -1;
        if (note==e.note)
            res = 0;
        else if (note > e.note)
            res = 1;
        return(res);
    }

    public String toString()
    {
        return(nom + " " + note);
    }
}
```

Exemple

```
import java.util.Arrays;

public class TestEtudiant
{
    public static void main (String[] args)
    {
        Etudiant[] etus = new Etudiant[10];
        etus[0] = new Etudiant("Etudiant0",2);
        etus[1] = new Etudiant("Etudiant1",1);
        etus[2] = new Etudiant("Etudiant2",18);
        etus[3] = new Etudiant("Etudiant3",4);
        etus[4] = new Etudiant("Etudiant4",13.5);
        etus[5] = new Etudiant("Etudiant5",14);
        etus[6] = new Etudiant("Etudiant6",12);
        etus[7] = new Etudiant("Etudiant7",13.5);
        etus[8] = new Etudiant("Etudiant8",12.5);
        etus[9] = new Etudiant("Etudiant9",20);
        for (int i=0; i<etus.length;i++)
            System.out.println(etus[i]);
        System.out.println("");
        Arrays.sort(etus);
        for (int i=0; i<etus.length;i++)
            System.out.println(etus[i]);
    }
}
```

Conclusions

- La notion d'interface est nouvelle
- Permet (d'une certaine manière) de lever certaines ambigüités dans l'utilisation de l'héritage pour modéliser une application.
- Utilisée intensivement dans tous les packages java et surtout dans les packages graphiques.
- Permet de spécifier des **services attendus** sur des classes
- Attention : une interface ne peut "grandir"

Langage : Plan

- Généralités
- Compilation & exécution
- Expressions, contrôle ...
- Classes & Objets
- Héritage
- Exceptions
- Encapsulation
- Interfaces
- **Divers**

Types ?

- Java type fortement ses variables, retours de méthode et expressions :
  - Types primitifs
  - Types références (classe ou interface)
- **Les types primitifs ne sont pas des objets**
- "Wrapper" types primitifs → classe (`java.lang`)

Types primitfs

nom	taille	codage
byte	8	cpl à 2
short	16	cpl à 2
int	32	cpl à 2
long	64	cpl à 2
char	16	unicode
float	32	IEEE 754
double	64	IEEE 754
boolean	ràf	true/false

- Unicode : voir www.unicode.org

Wrappers

- Nom du type primitive mais **Capitalisé**
  - `java.lang.Double` pour `double`, ...
  - Sauf `java.lang.Integer` pour `int`
- Wrappers plus "abstrait" : Ex : `Number`
- Support de plein de méthodes de conversion/formatage
- Support de "constantes" liées au type
  - Ex :

```
public final
class Integer extends Number {
    /**
     * The minimum value an Integer can have.
       The lowest minimum value an
     * Integer can have is 0x80000000.
     */
    public static final int   MIN_VALUE = 0x80000000;
.......
```

Plan du cours

- Introduction
- Langage
- **Entrée/Sortie**
- Threads
- Interface graphique
- Applet
- Perspective - Conclusion

Entrée/Sortie

- un programme échange de l'information par :
  - des écritures
  - des lectures
- ces échanges se font :
  - en mémoire
  - dans des fichiers ordinaires
  - dans des fichiers spéciaux ($\longrightarrow$ clavier, écran, imprimante, ...)

Entrées

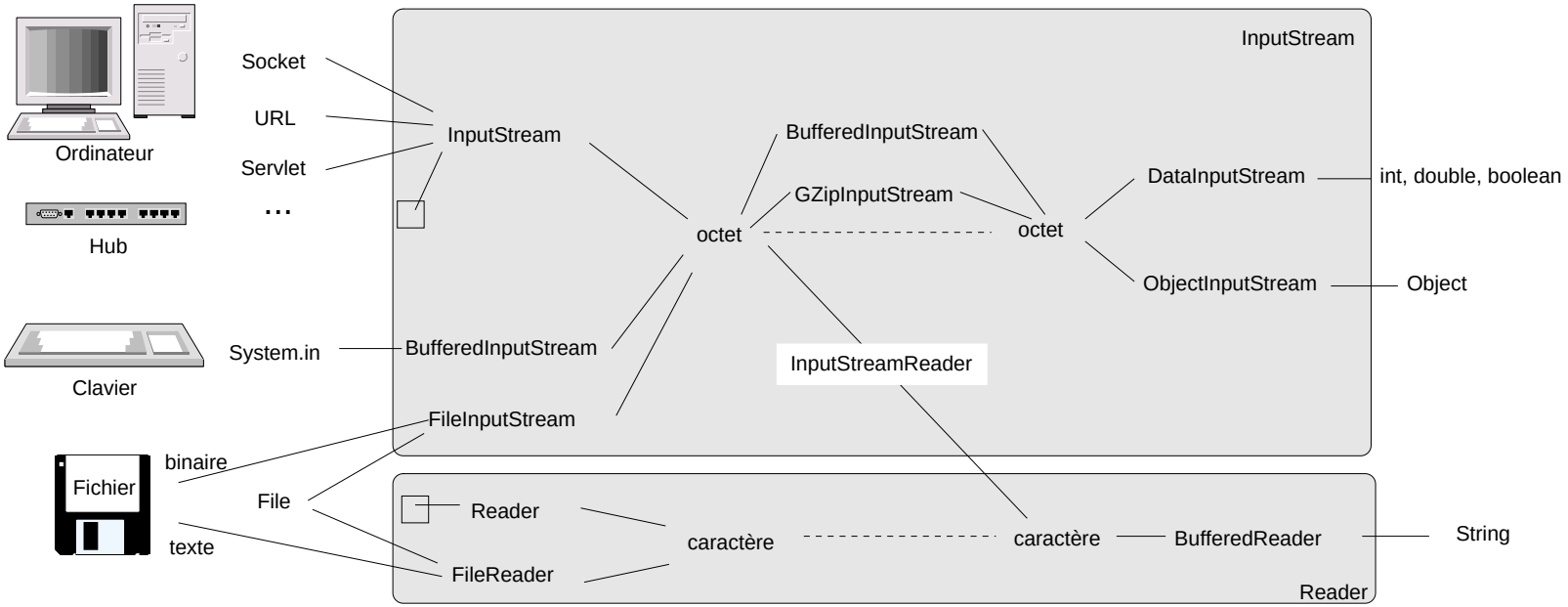


FIG. 3 – Flux d'entrées

Sorties

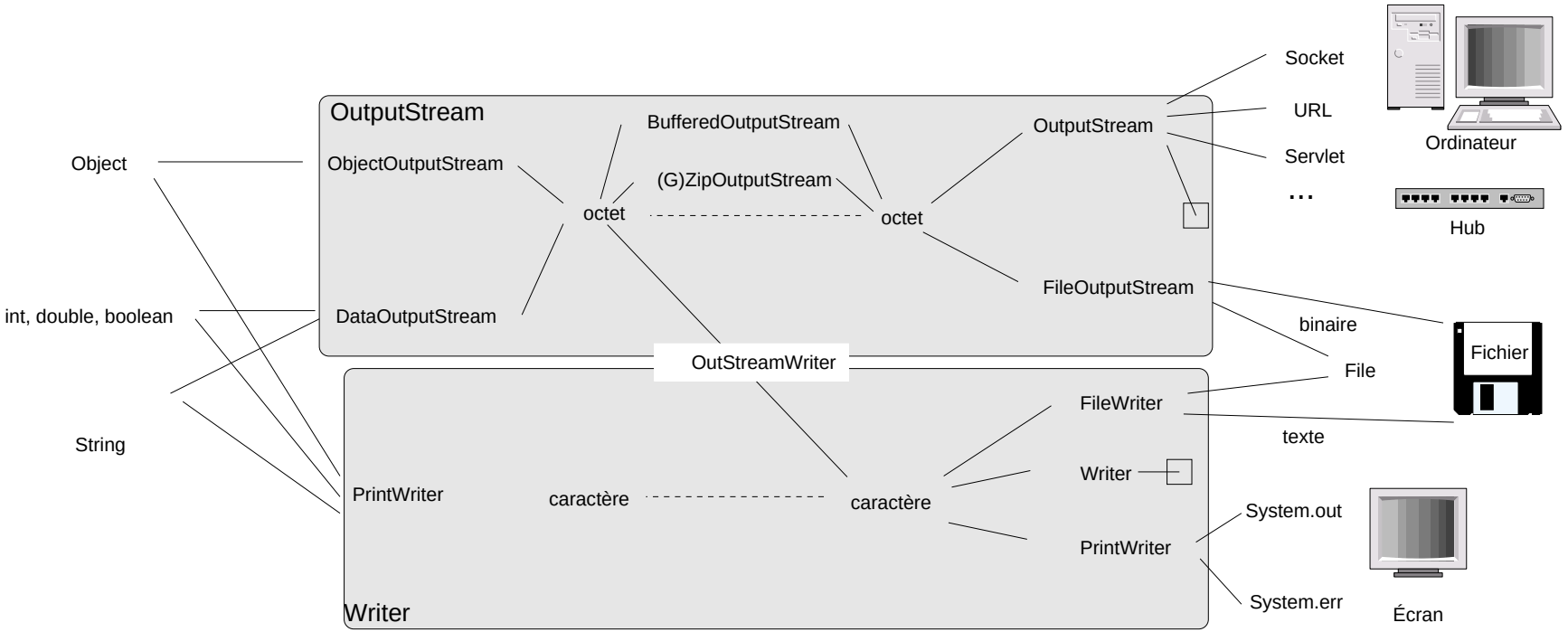


FIG. 4 – Flux de sorties

Flots – principe de lecture/écriture

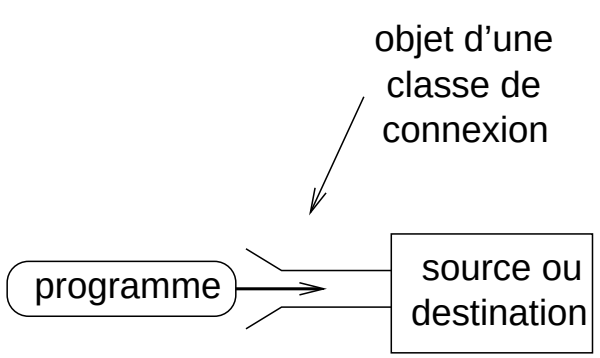
- programme ↔ **stream** ou flot ↔ source ou destination
- (pseudo)algo :

```
ouvrir le flot
tant que (il y a encore de l'information)
   écrire ou lire
fin tant que
fermer le flot
```

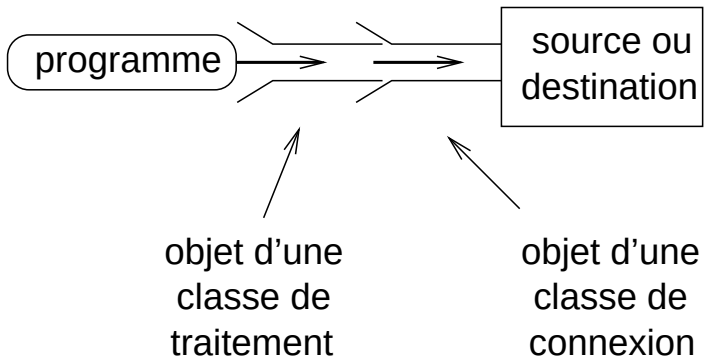
Entrée/Sortie

Deux types d'objets :

1. classe de connexion (directe).
   - lecture/écriture de caractères et octets.
   - méthodes `int read()` et `void write(int c)`
   - exemple : `FileInputStream`, `StringBufferInputStream`, `FileReader`, ...
2. classe de filtrage (traitement, connexion indirecte).
   - lecture/écriture de tout type simple ou objet, avec cache, ...
   - méthodes `writeObjet`, `writeInt`, `readLine`, ...



(a) Connexion directe



(b) Connexion indirecte

Lecture connexion directe

```
TrucReader in = new TrucReader(???);
// ou TrucInputStream in = new TrucInputStream(???);
// ou TrucInputStream in = obj.methode(); (renvoyant un TrucInputStream)
int c;
while ((c = in.read()) != -1)
{
    System.out.println((char)c);
}
in.close();
```

Écriture filtrée

```
TrucWriter out = new TrucWriter(); // connexion
BiduleWriter b = new BiduleWriter(out); // filtrage
b.writeXXX(val);
b.close();
out.close();
```

Exemple

```
import java.net.URL;
import java.io.*;

public class DemoIO_1
{
    public static void main(String[] args) throws IOException
    {
        String line;
        URL url = new URL("http ://doc.src");
        BufferedReader b
            = new BufferedReader(new InputStreamReader(url.openStream ()));
        while ((line = b.readLine()) != null)
            System.out.println(line);
    }
}
```

Entrée/Sortie standard

- Sorties :
  - sortie standard : `System.out`
  - sortie erreur standard : `System.err`
  - `final static PrintStream` de la classe `java.lang.System`
  - méthodes : `print(String)` et `println(String)`
  - Rappel : on dispose de la *string conversion* et de `Object.toString()`
- Entrée
  - entrée standard : `System.in`
  - `final static InputStream` de la classe `java.lang.System`

Lecture ?

```
import java.io.*;

public class DemoIO_2
{
    public static void main(String[] args) throws IOException
    {
        String line="";
        try
            {
                BufferedReader in =
                    new BufferedReader(new InputStreamReader(System.in));
                line = in.readLine();
            }
        catch (IOException e)
            {
                System.err.println("Erreur entree/sortie");
                System.exit(0);
            }
        System.out.println(line);
    }
}
```

Fichier

- classe `File`. Cette classe est dotée de méthodes de :
  - construction : à partir du nom d'un fichier (en pouvant spécifier son répertoire)
  - tests : d'écriture `canWrite()`, de lecture `canRead()`, `isFile()` fichier ou `isDirectory()` répertoire, ...
  - informatives : `getName()` nom, `getPath()` nom complet du fichier, `length()` longueur du fichier, `list[Files]()` fichiers d'un répertoire, ...
  - de manipulation : `createNewFile()`, `delete()`, `mkdir()`, ...
- écriture et lecture sont réalisés à l'aide d'objets de connexion à ces fichiers.

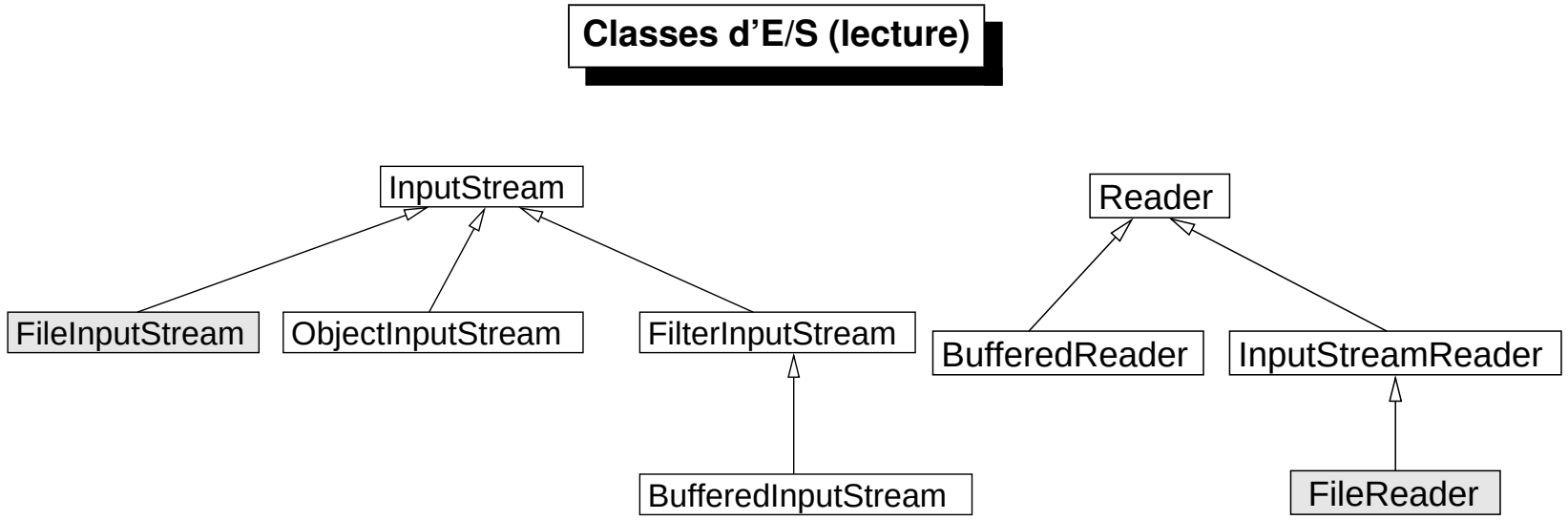
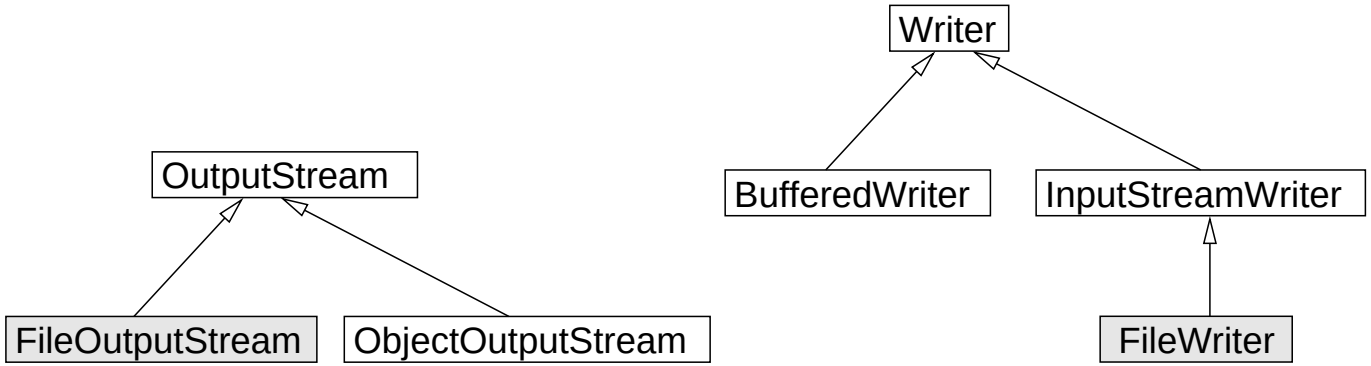


FIG. 5 – Lecture (simplifié)

Classes d'E/S (écriture)



(a) flots d'octets (b) flots de caractères

FIG. 6 – Écriture (simplifié)

Plan du cours

- Introduction
- Langage
- Entrée/Sortie
- **Threads**
- Interface graphique
- Applet
- Perspective - Conclusion

Rappels

- multitâche : exécution de plusieurs tâches en même temps
- une tâche ou processus
  - comportement donné par un programme
  - dispose de son espace mémoire
  - est dynamique (naît, s'exécute, ...) $\neq$ programme (statique)
- les sytèmes d'exploitation sont multitâches :
  1. coopératif : un processus prend la main et la rend volontairement !
  2. préemptif : l'ordonnanceur donne et reprend la main aux processus
- le multitâche peut être :
  1. réel : machine multiprocesseur
  2. simulé : machine monoprocesseur
- multitâche simulé obtenu en allouant des temps d'exécutions petit par rapport à notre perception
- un programme peut être multitâche
- deux manières :
  1. processus : contexte lourd, espace mémoire séparé
  2. thread (fil d'exécution) : processus léger, en particulier espace mémoire partagé

Cadre d'utilisation/Utilité

- exemples d'utilisations
  - serveur réseaux
  - gestion des intéractions utilisateurs
  - ...
- utilité :
  - machine multiprocesseur : repartir une tâche sur plusieurs processeurs
  - machine monoprocesseur : mieux traiter des tâches interactives

Vue d'ensemble

- Thread = flot d'instructions séquentielles
- éventuellement **plusieurs** Threads dans un programme

⟶ Multitâche

- un thread est un fil d'exécution
- le contrôle d'un thread se fait à l'aide d'un objet `Thread`
- le fil d'exécution se trouve dans la méthode `run()` du `Thread`
- la classe `Thread` dispose par défaut d'une méthode `run()` ne faisant rien

Thread actif

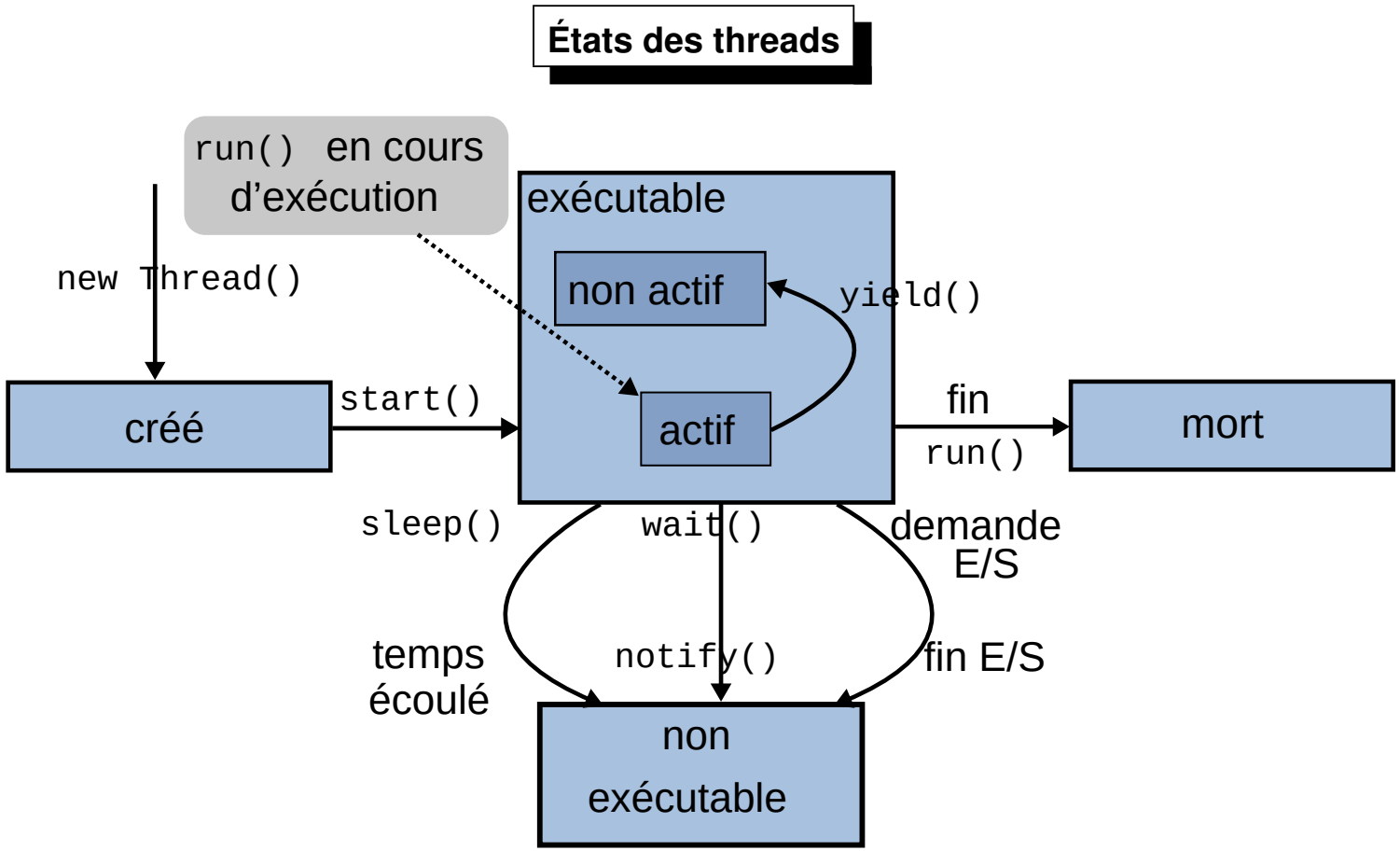
```
public class AllThread
{
    public static void main (String args[])
    {
        ThreadGroup parent =     Thread.currentThread().getThreadGroup();
        while(parent.getParent()!=null)
        {
            parent = parent.getParent();
        }
        }

        public static affThread(ThreadGroup tg)
        {
            int nbGrp = tg.activeGroupCount();
            int nbThread = tg.activeCount();
            ThreadGroup[] groupes = new ThreadGroup[nbGrp];


        }
}
```

Thread actif (GUI)

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

public class AllThreadGui extends JFrame implements ActionListener
{
    public AllThreadGui(String titre)
    {
        super(titre);
        JButton b = new JButton("GO");
        b.addActionListener(this);
        add(b);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
        for (Thread t: Thread.getAllStackTraces().keySet())
            System.out.println(t);
    }

    public static void main(String args[])
    {
        AllThreadGui gui = new AllThreadGui("AllThreadGui");
    }
}
```

Plan

- État
- Création
- Ordonnancement
- Synchronisation


États des threads
run() en cours d'exécution
exécutable
non actif
yield()
new Thread()
créé
start()
actif
fin
run()
mort
sleep()
wait()
demande E/S
temps écoulé
notify()
fin E/S
non exécutable

Création d'un `Thread`

- pour créer SON thread, et lui fournir `run()` deux méthodes :
  1. `run()` "interne" :
     - sous classer la classe `Thread`
     - redéfinir la méthode `run()` avec SON code
     - utiliser le contructeur ⇐ `Thread([String nom])`
  2. `run()` "externe" :
     - implémenter l'interface `Runnable`
     - définir la méthode `run()`
     - utiliser le contructeur ⇐ `Thread(Runnable cible,[String nom])`

Exemple

```
public class SimpleThread extends Thread
{
    public SimpleThread(String str)
    {
        super(str);
    }

    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            System.out.println(i + " " + getName());
            try
            {
                sleep((long)(Math.random() * 1000));
            }
            catch (InterruptedException e) {}
        }
        System.out.println(getName() + " fini !");
    }

    public static void main (String[] args)
    {
        new SimpleThread("Un").start();
        new SimpleThread("Deux").start();
    }
}
```

Exemple

```
public class ComplicatedThread implements Runnable
{
    private Thread leThread = null;
    private String name;
    public void start()
    {
        if (leThread == null)
            {
                leThread = new Thread(this, name);
                leThread.start();
            }
    }
    public ComplicatedThread(String str)
    {
        name = str;
    }
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            System.out.println(i + " " + name + " "
                               + leThread.getName() + " "
                               + Thread.currentThread().getName());
            try
            {
                Thread.currentThread().sleep((long)(Math.random() * 1000));
            }
            catch (InterruptedException e) {}
        }
        System.out.println(name + " fini !");
    }
    public static void main (String[] args)
    {
        new ComplicatedThread("Un").start();
        new ComplicatedThread("Deux").start();
    }
}
```

```
0 Un
0 Deux
1 Deux
2 Deux
1 Un
2 Un
3 Deux
4 Deux
3 Un
5 Deux
6 Deux
7 Deux
4 Un
8 Deux
5 Un
9 Deux
6 Un
7 Un
Deux fini !
8 Un
9 Un
Un fini !
```

Priorités

- Pour assurer le partage d'une CPU entre plusieurs threads →
- Ordonnancement des threads basé sur des priorités fixes (et non flottantes !).
- À un instant donné, si plusieurs threads sont dans l'état "runnable", celui dont la priorité est la plus élevée passe dans l'état running.
- Si plusieurs threads sont au même niveau de priorité, alors l'ordonnanceur les prend les uns après les autres.
- La priorité d'exécution **n'est pas garantie** → Ne doit pas servir pour implanter une exclusion mutuelle.

Priorités

Un thread s'exécute jusqu'à ce que :

- Un thread de priorité plus élevée passe dans l'état "runnable"
- Il redonne la main : yield(), sleep() ...
- Sur les systèmes supportant le temps partagé, son quota de temps est expiré.

À la création un thread à la même priorité que son père.

Priorités : Example

```
class RaceTest {
  final static int NUMRUNNERS = 2;
  public static void main(String[] args) {
    SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i] = new SelfishRunner(i); runners[i].setPriority(2); }
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i].start(); }
  }
}
class SelfishRunner extends Thread {
  public int tick = 1;
  public int num;
  SelfishRunner(int num) { this.num = num; }
  public void run() {
    while (tick < 400000) { tick++; if ((tick % 50000) == 0) { System.out.println("Thread #" + num + ", tick = " + tick); }}
  }
}
```

Synchronisations

- Les synchronisation sont basés sur les moniteurs de HOARE [**?**]
- Une section critique, plutôt matérialisée par une méthode en java est déclarée comme `synchronized`

```
public class TestProdCons
{
    public static void main(String[] args)
    {
        int prod = 0;
        int cons = 0;

        if (args.length != 2)
          System.exit(0);
        try
        {
          prod = Integer.parseInt(args[0]);
          cons = Integer.parseInt(args[1]);
        }
        catch (NumberFormatException e)
        {
          System.out.println(" un entier !!!!");
        }
        Tampon c = new Tampon();
        Producteur p1 = new Producteur(c, prod);
        Consommateur c1 = new Consommateur(c, cons);

        p1.start();
        c1.start();
    }
}
```

```
public class Producteur extends Thread
{
    private Tampon tampon;
    private int number;

    public Producteur(Tampon t, int number)
    {
        tampon = t;
        this.number = number;
    }

    public void run()
    {
        for (int i = 0; i < 10; i++)
            {
                System.out.print("Producteur " + " put : ");
                tampon.put(i);
                System.out.println(i);
                try {
                    sleep((int)(Math.random() * number * 100));
                } catch (InterruptedException e) { }
            }
    }
}
```

```
public class Consommateur extends Thread
{
    private Tampon tampon;
    private int number;

    public Consommateur(Tampon t, int number)
    {
        tampon = t;
        this.number = number;
    }

    public void run()
    {
        int value = 0;
        for (int i = 0; i < 10; i++)
        {
            System.out.print("Consommateur " + " get : ");
            value = tampon.get();
            System.out.println(value);
                try {
                    sleep((int)(Math.random() * number * 100));
                } catch (InterruptedException e) { }

        }
    }
}
```

```
import java.util.LinkedList;
public class Tampon
{
    final static int N = 5;
    private LinkedList l;
    private int compteur;

    public Tampon()
    {
        l = new LinkedList();
        compteur = 0;
    }

    public synchronized int get() {
        int val;
        if (compteur == 0)
        {
          System.out.println(" get bloquant... ");
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        compteur--;
        notifyAll();
        val = ((Integer) l.getFirst()).intValue();
        l.removeFirst();
        return (val);
    }

    public synchronized void put(int value) {
        while (compteur == N) {
            try {
                System.out.println(" put bloquant... ");

                wait();
            } catch (InterruptedException e) { }
        }
        l.addLast(new Integer(value));
        compteur++;
        notifyAll();
    }
}
```

Moniteurs

- Un moniteur est généralement associé à une donnée et fonctionne comme un verrou sur cette donnée.
- Quand un thread acquiert le moniteur d'une donnée, les autres threads sont verrouillés et ne peuvent ni lire ni modifier cette donnée.
- En java, UN moniteur unique est associé à chaque objet qui a au moins une méthode "synchronized".
- Si une méthode de classe est déclarée comme "synchronized" alors un moniteur est associé à l'instance de la classe `Class` qui représente la classe de la méthode statique considérée, dans la JVM.

Moniteurs

2 méthodes (atomiques) sont associées à l'utilisation des moniteurs (et donc ne peuvent être utilisées que dans le corps de méthodes "synchronized") :

wait S'endormir dans le moniteur → un autre thread peut donc entrer.

notify Réveiller un thread endormi dans le moniteur ...

Objectif : un seul thread actif dans le moniteur

Programmes multi-threads

JAVA ne prévient :

- ni les deadlocks
- ni la famine
- C'est donc à la responsabilité du programmeur d'utiliser des modèle de programmation (producteur/consommateurs etc...) pour éviter ces problèmes.

Plan du cours

- Introduction
- Langage
- Entrée/Sortie
- Threads
- **Interface graphique**
- Applet
- Perspective - Conclusion

Interface graphique : Plan

- Principes généraux
- Composants graphiques
  - Composants (primitifs)
  - Conteneur
  - Gestionnaires de répartition
  - Menus
- Gestion des événements
  - classes internes

Vocabulaire

- CLI : *command line interface* interface utilisateur **texte**
- GUI : *graphical user interface* interface utilisateur **graphique**. Permet des actions souris et clavier.
- *widget* : composant graphique (boutton, fenêtre, champs de saisie texte, ...)

Historique

- JDK 1.0 : AWT *Abstact Window Toolkit*
- JDK 1.1 : révision du modèle de gestion des événements
- projet swing des JFC (*Java Foundation Classes*). Contenu :
  1. jeu de composant
  2. plusieurs *Look-and-Feel*
  3. 2D API : graphiques 2D
  4. accessibilité, glisser-déposer
  5. internationalisation
  6. ...
- JDK 1.2 : Swing integré (`javax.swing`)
- Cohabitation de deux APIs de GUI : AWT et Swing :-(
- Compatibilité ascendante $\Rightarrow$ choix discutables

Look-and-Feel

(a) Motif

(b) Windows

(c) Métal

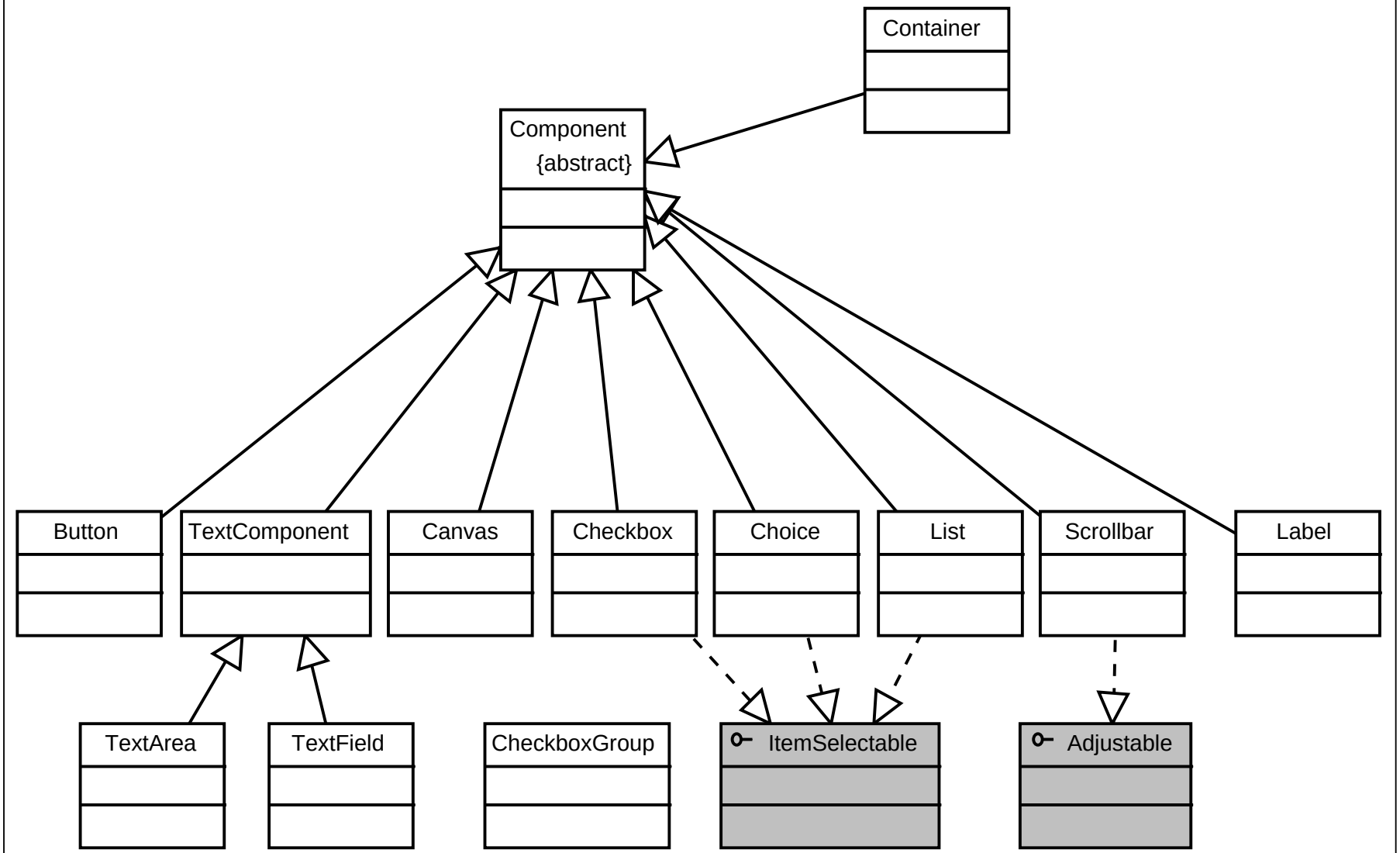
(d) Métal "Complet"

FIG. 7 – *Look-and-Feel*

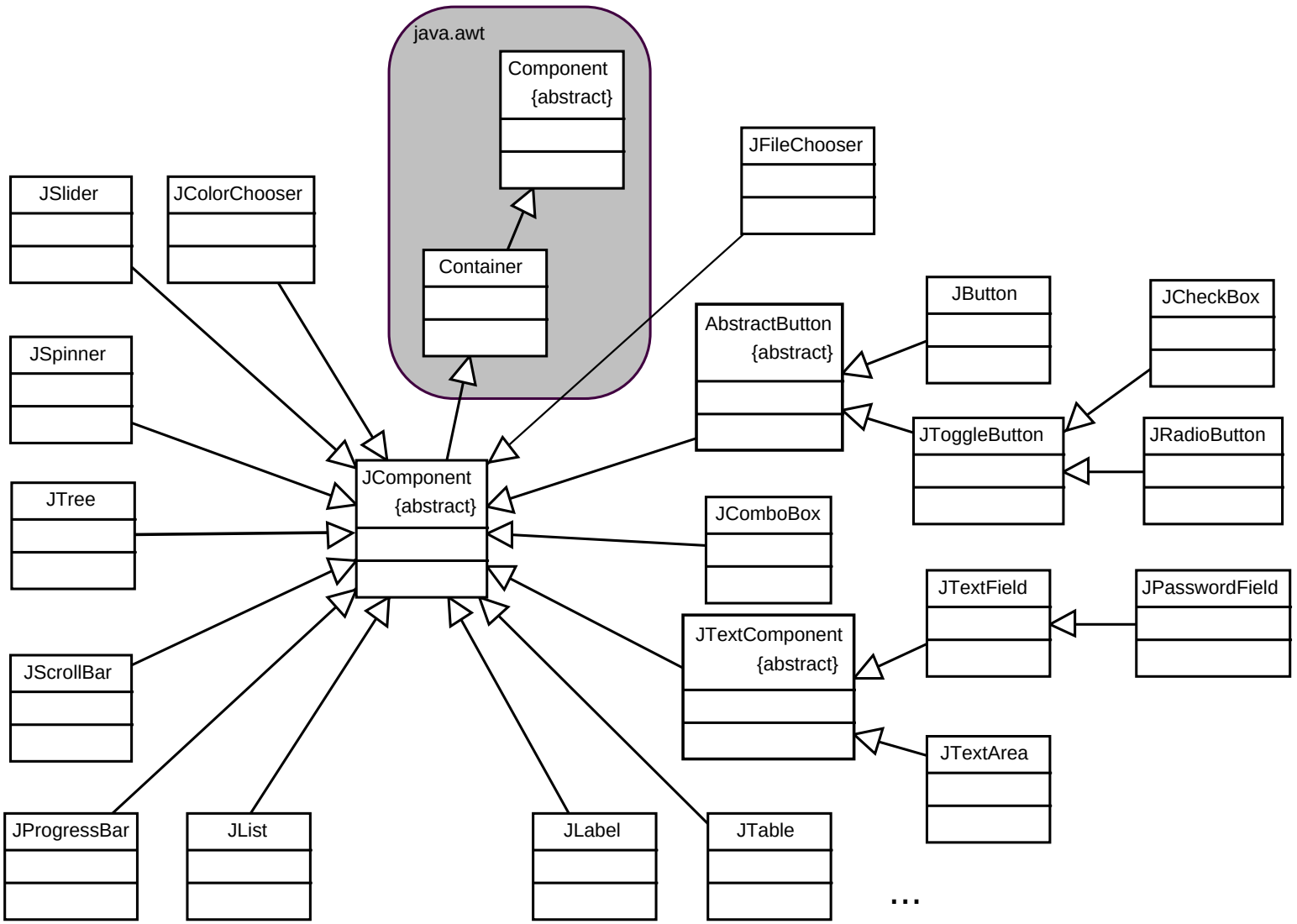
Principe

- **Composant** : objet pouvant être dessiné et interagir avec l'utilisateur.
  - objets graphiques primitifs : label, boutton ...
  - **Conteneur** : des "répertoires" de composants.
- **disposition** régie par un gestionnaire de répartition (`LayoutManager`)
- une **action** sur un Composant déclenche des (objets) **évenements**
- la **gestion de l'évenement** est déléguée à un `EventListener`
- Menus traités à part

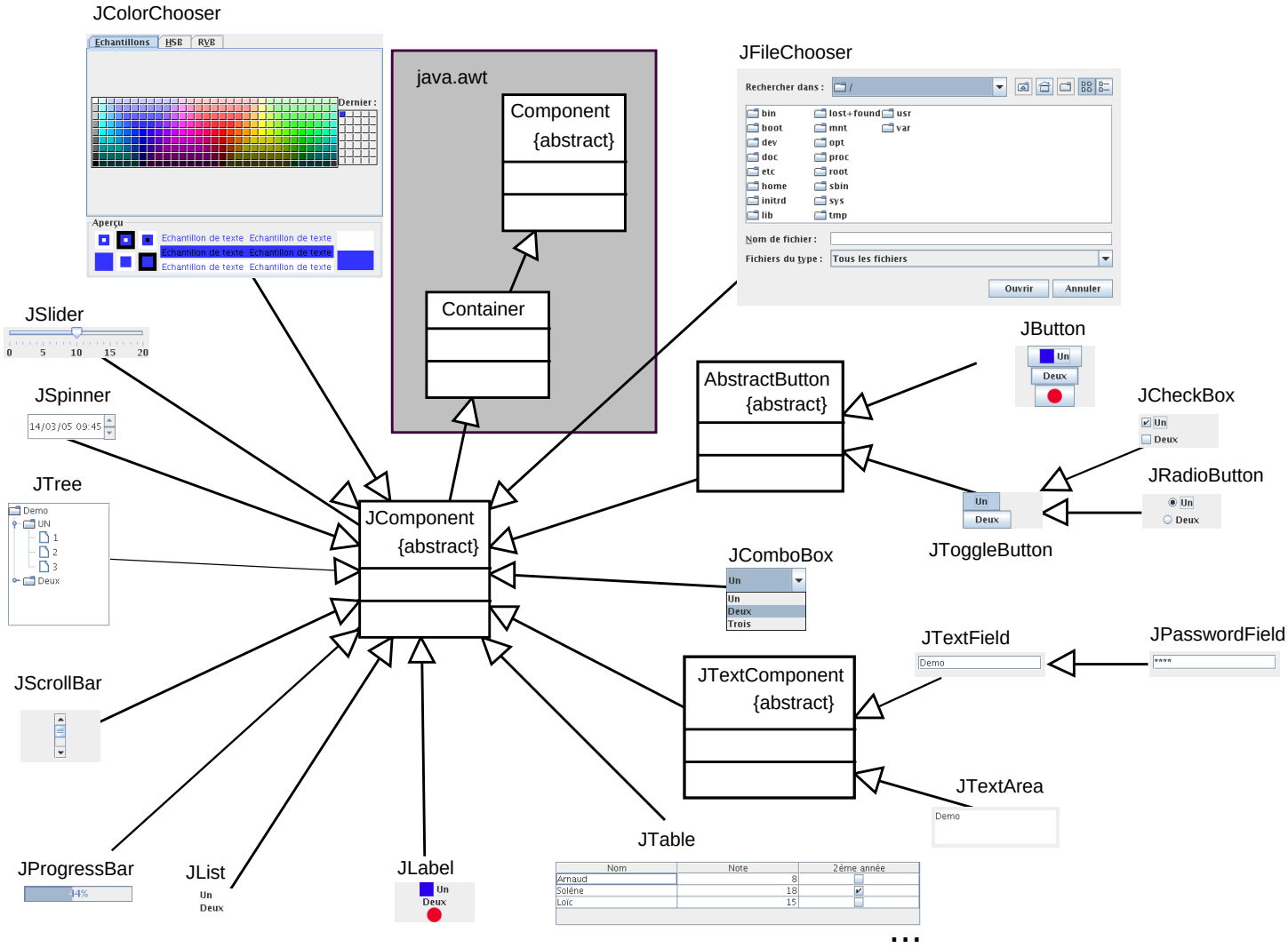
Hiérarchie des classes AWT



Hiérarchie des classes Swing



Hiérarchie des classes Swing



(J)Component

- Les composants sont dans un arbre d'objets.
- Les `Containers` constituent les nœuds
- Les objets primitifs les feuilles.
- Chaque nœud peut instancier un *layout* de façon à organiser le placement des objets qu'il contient.
- Un `(J)Component` dispose des méthodes suivantes :

  fixer son État `setEnable(),setVisible(),setBackground(),setFont(),...`

  Information `isEnabled(),isShowing(),getBackground(),getFont(),...`

  Taille `set/getPreferedSize set/getMaximumSize set/getMinimumSize`

Arborescence de composants

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

public class Arborescence
{
  public static void main(String[] args)
  {
    // Création du "Top Level Container"
    JFrame f = new JFrame("Arborescence");
    // Association d'un gestionnaire de répartition
    f.setLayout(new FlowLayout());
    // création et ajout d'un premier composant
    JButton b = new JButton("Go !");
    f.add(b);
    // création et ajout d'un deuxième composant
    JComboBox l = new JComboBox();
    l.addItem("item 1");
    l.addItem("item 2");
    f.add(l);
    // Dimensionnement et affichage
    f.setSize(300,150);
    f.setVisible(true);
  }
}
```

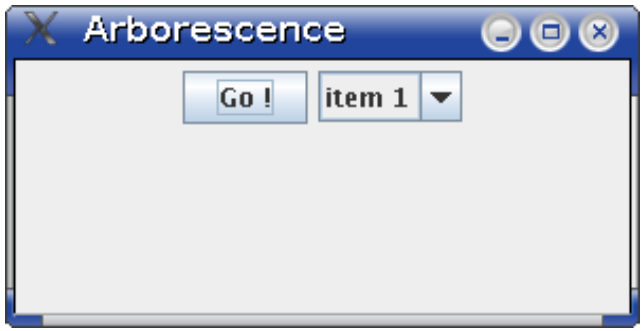


FIG. 8 – Arborescence de composants

Classes graphiques de base

- `JButton,JCheckbox,JLabel,JComboBoxList,JScrollbar,JTextField,...`
- "de base" : ne peuvent pas contenir d'autres composants graphiques

Composants primitifs

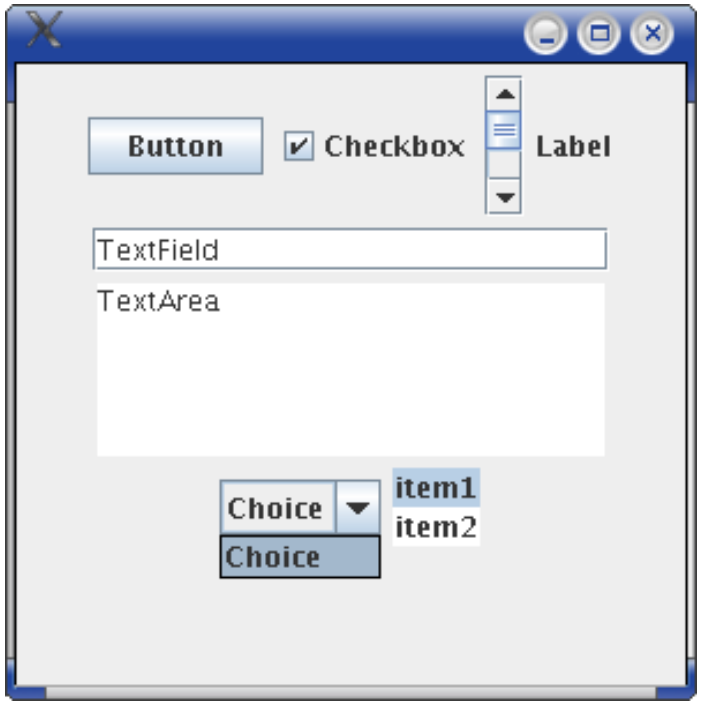


FIG. 9 – Bouton, liste, boite à cocher ...

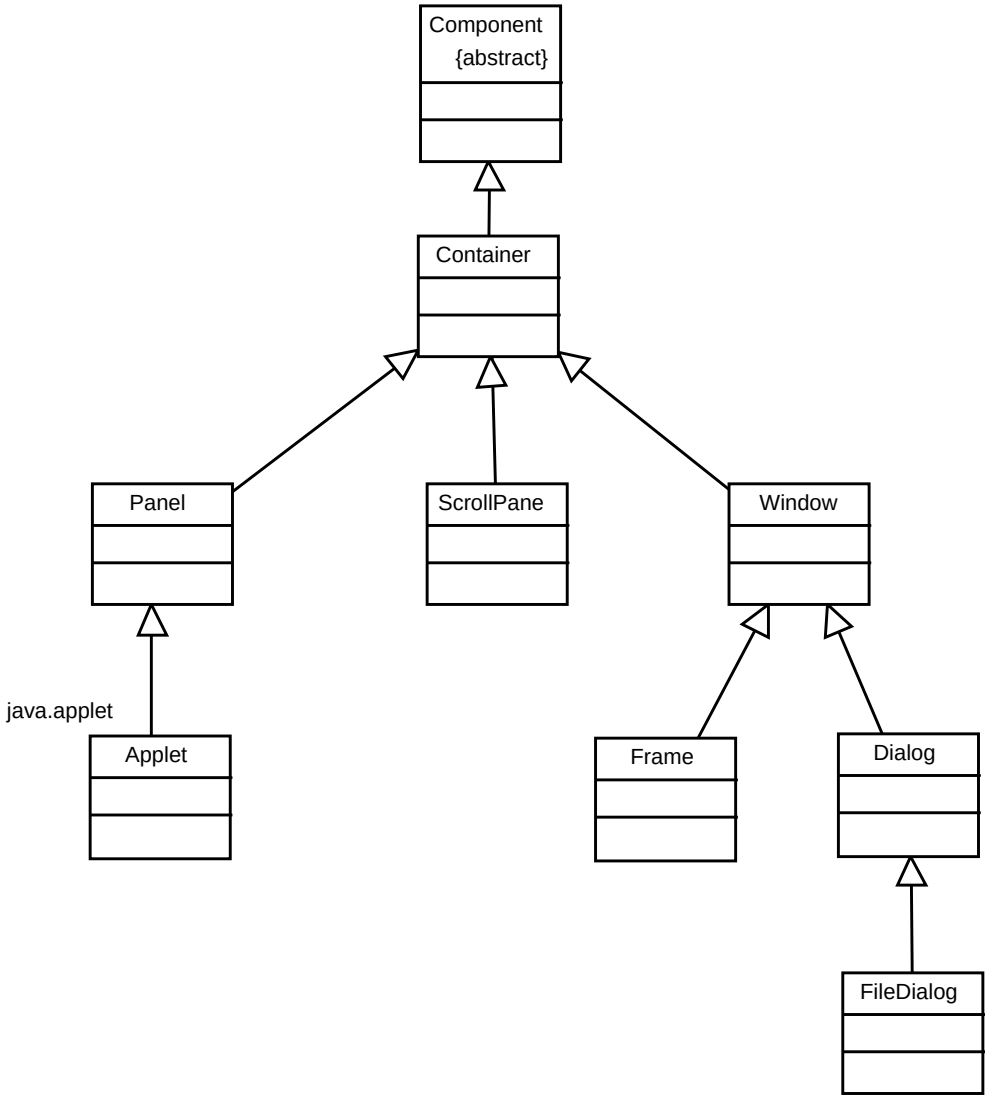
Composants primitifs

```
package gui;

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

public class Demo
{
  public static void main(String args[])
  {
    JFrame f = new JFrame();
    Container c = f.getContentPane();
    c.setLayout(new FlowLayout());
    JButton b = new JButton("Button");
    c.add(b);
    JCheckBox cbox = new JCheckBox("Checkbox");
    c.add(cbox);
    JScrollBar sb = new JScrollBar();
    c.add(sb);
    JLabel lab = new JLabel("Label");
    c.add(lab);
    JTextField tf = new JTextField("TextField",20);
    c.add(tf);
    JTextArea ta = new JTextArea("TextArea",5,20);
    c.add(ta);
    JComboBox co = new JComboBox();
    co.addItem("Choice");
    c.add(co);
    String[] items = {"item1", "item2"};
    JList l = new JList(items);
    c.add(l);
    f.setSize(300,300);
    f.setVisible(true);
  }
}
```

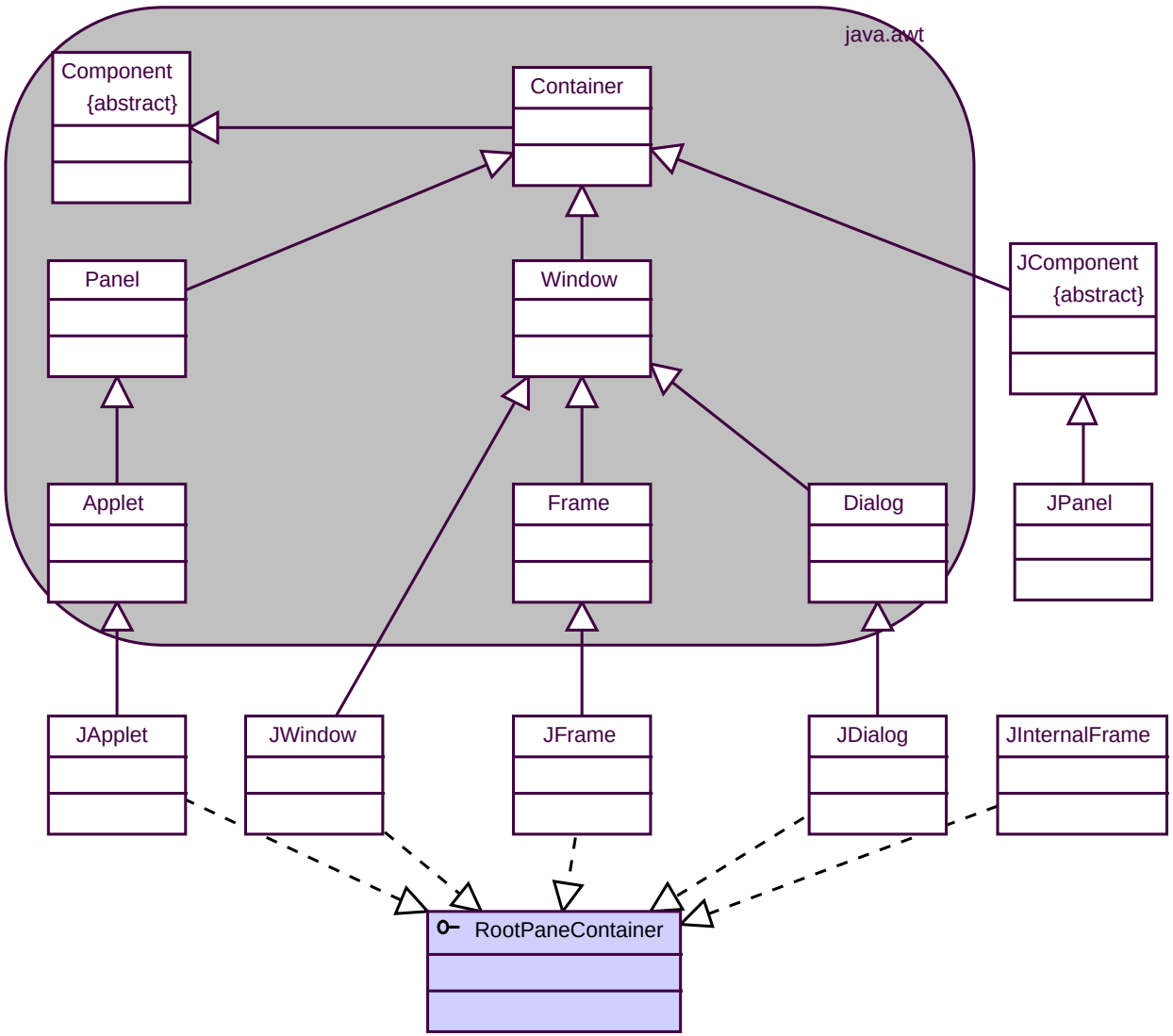
Container et Layout

- Un Container se comporte comme un répertoire d'objets graphiques.
- Un *layout* organise le placement des objets graphiques contenus dans un Container.
- Layout et Container sont clairement séparés :
  - possibilité de définir des nouveaux Layout sans définir de nouveau Container (interface LayoutManager[2])
  - Possibilité pour un Container de changer de Layout à l'exécution.

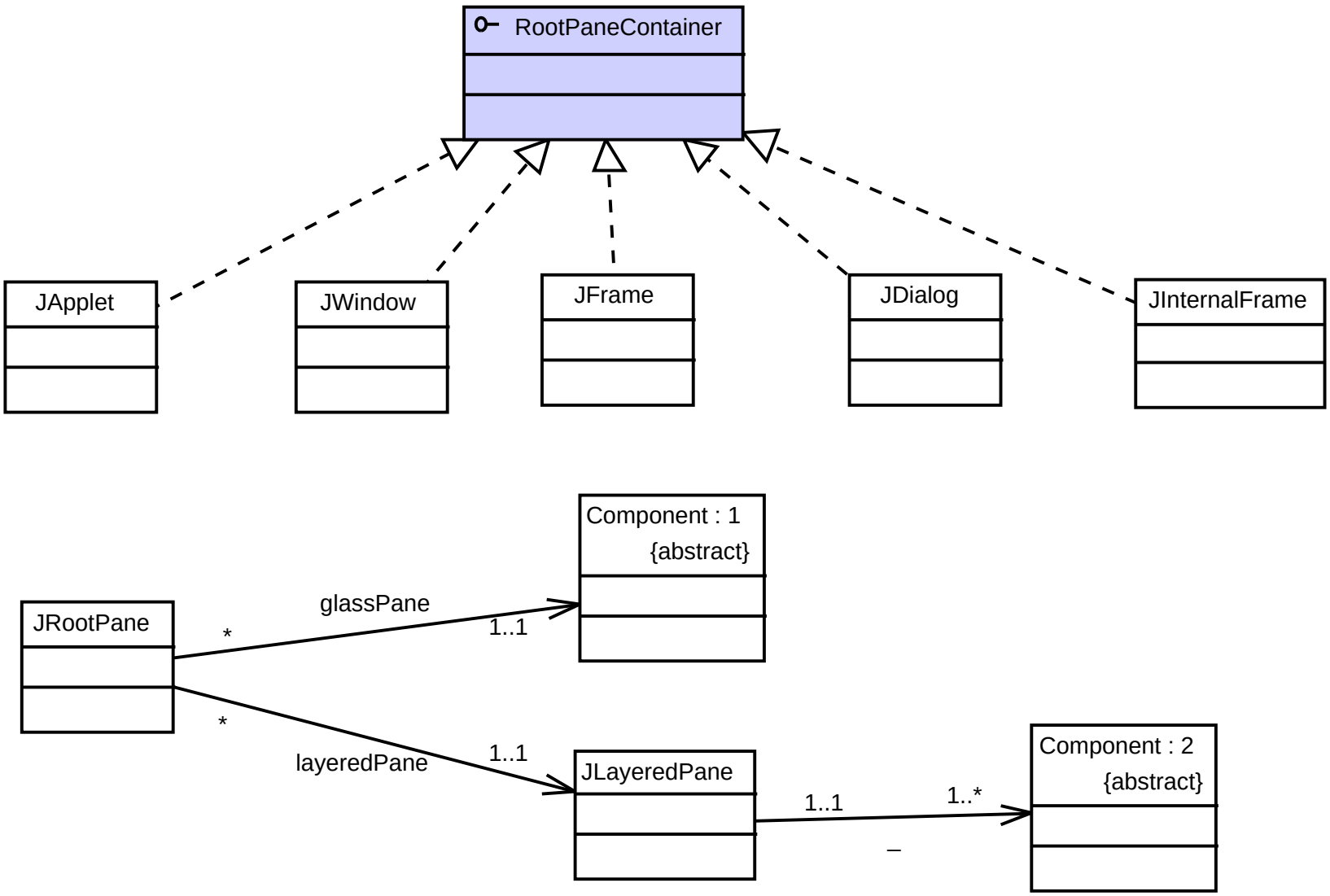
Container Awt



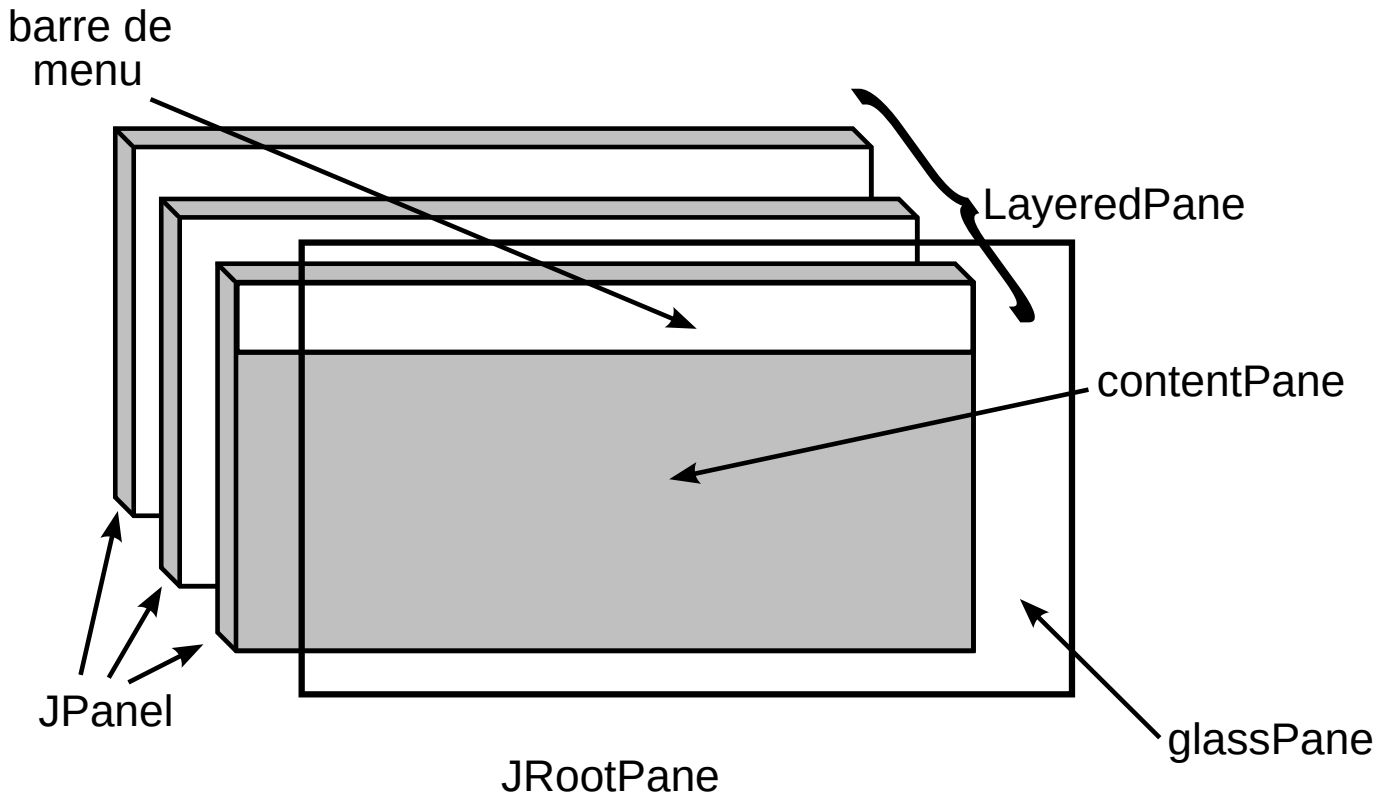
Container Swing



Container Swing (Détail)



Container Swing (Détail)



Container

2 catégories :

- Ceux qui n'ouvrent pas de nouvelles fenêtres : `(J)Panel, (J)ScrollPane, (J)Applet`
- Ceux qui ouvrent de nouvelles fenêtres : `(J)Dialog, (J)Frame (J)Window.`

Dialog

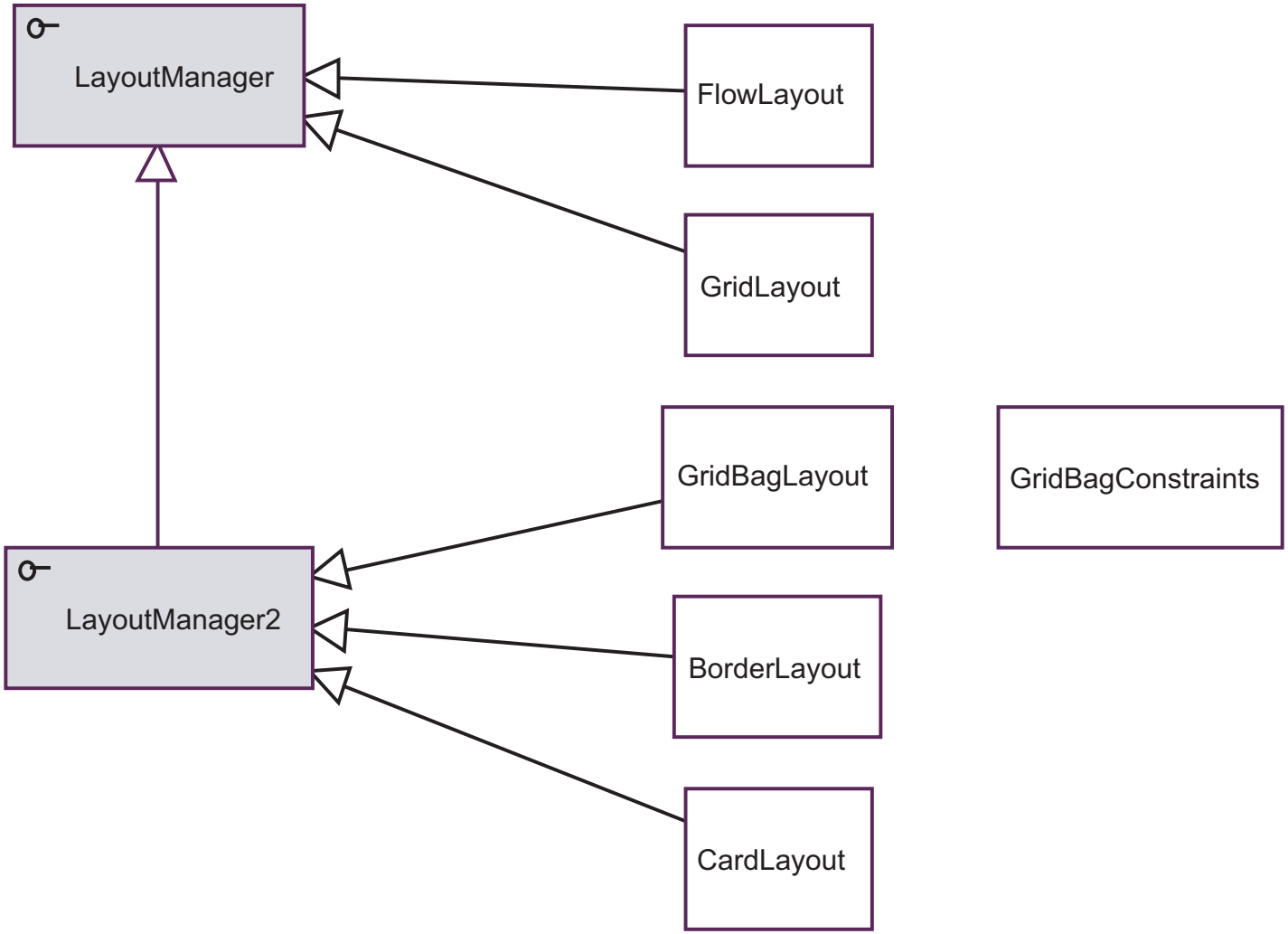
```
package gui;
import java.awt.*;
import javax.swing.*;

public class DialogDemo
{
    public static void main(String[] args)
    {
        JFrame f = new JFrame("DialogDemo");
        JOptionPane.showMessageDialog(f,"Information");
        JOptionPane.showMessageDialog(f,"Erreur", "Titre",
                                      JOptionPane.ERROR_MESSAGE);
        JOptionPane.showConfirmDialog(f,"Confirmation", "Titre",
                                      JOptionPane.YES_NO_OPTION);
        String entree = JOptionPane.showInputDialog("Valeur : ");
        JLabel l = new JLabel(entree,JLabel.CENTER);
        f.add(l);
        f.setSize(200,100);
        f.setVisible(true);
    }
}
```

(a) Dia

FIG. 10 – Exemple de diabgue

Layout

Pour tout `Container`, il est possible de choisir un *layout* pour organiser le placement des objets graphiques qu'il contient :



Container

- association conteneur/gestionnaire de répartition :
  1. à la création

```
JPanel panel = new JPanel(new BorderLayout());
```

  2. sur un conteneur existant

```
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
```

- ajout de composant :

```
leConteneur.add(unComposant, eventuellementUneContrainte);
```

BorderLayout

- définit 5 zones :
  1. nord : NORTH ou PAGE_START
  2. sud : SOUTH ou PAGE_END
  3. ouest : WEST ou LINE_START
  4. est : EAST ou LINE_END
  5. centre : CENTER ou par défaut
- les composants se dimensionnent à l'espace disponible
- l'espace résiduel va prioritairement au centre
- non respect des tailles
- par défaut pour les (J)Frame

Border

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

public class LayoutBorderDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f = new JFrame("Border");
        Container c = f.getContentPane();
        c.setLayout(new BorderLayout());

        c.add(new JButton("NORTH ou PAGE_START"),
              BorderLayout.NORTH);
        c.add(new JButton("SOUTH ou PAGE_END"),
              BorderLayout.SOUTH);
        c.add(new JButton("<html>WEST <br>ou <br>LINE_ST
              BorderLayout.WEST);
        c.add(new JButton("<html>EAST <br>ou <br>LINE_END</html>"),
              BorderLayout.EAST);
        c.add(new JButton("<html>CENTER <br>ou <br>rien</html>"),
              BorderLayout.CENTER);

        f.setSize(300,200);
        f.setVisible(true);
    }
}
```

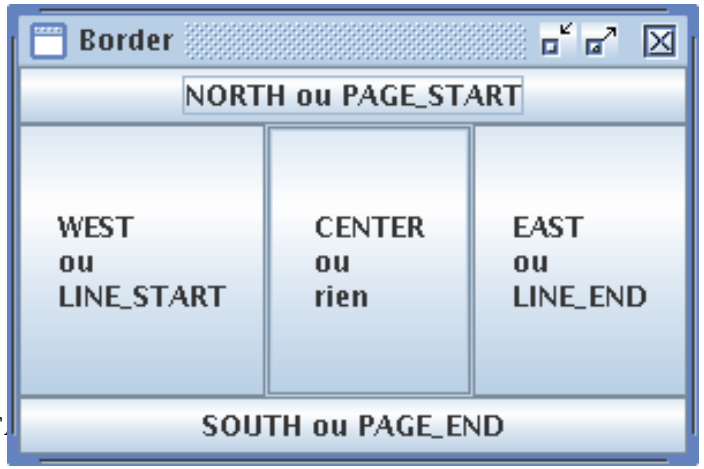


FIG. 11 – BorderLayout

FlowLayout

- par défaut pour les (J)Panel
- place les composants en ligne
- nouvelle ligne quand l'espace est insuffisant
- respecte la taille préférée des composants
- possibilité d'un espacement entre les composants

Flow

```
package gui;

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

public class LayoutFlowDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f = new JFrame("Flow");
        Container c = f.getContentPane();
        c.setLayout(new FlowLayout());

        c.add(new JButton("Bouton 1"));
        c.add(new JButton("2"));
        c.add(new JButton("Bouton 3"));
        c.add(new JButton("Long nom de Bouton 4"));
        c.add(new JButton("Bouton 5"));
        f.setSize(200,200);
        f.setVisible(true);
    }
}
```

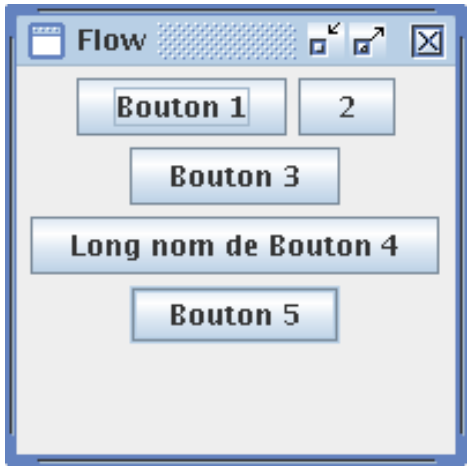


FIG. 12 – FlowLayout

GridLayout

- place les composants dans une grille de cellule
- toutes les cellules ont la même taille
- nombre de ligne et de colonne donné
- ou 0 = autant qu'il est nécessaire
- ne respecte pas la taille préférée des composants

Grid

```
package gui;

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

public class LayoutGridDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f = new JFrame("Grid");
        Container c = f.getContentPane();
        c.setLayout(new GridLayout(0,2));
        // 0 signifie n'importe quel nombre de ligne

        c.add(new JButton("Bouton 1"));
        c.add(new JButton("2"));
        c.add(new JButton("Bouton 3"));
        c.add(new JButton("Long nom de Bouton 4"));
        c.add(new JButton("Bouton 5"));

        f.setSize(300,200);
        f.setVisible(true);
    }
}
```

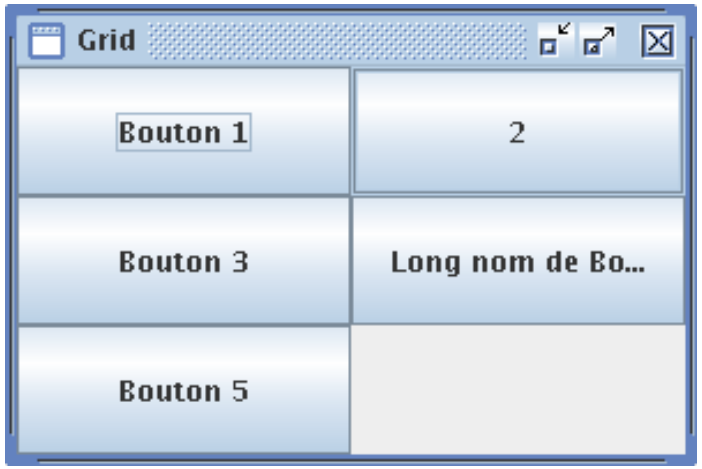
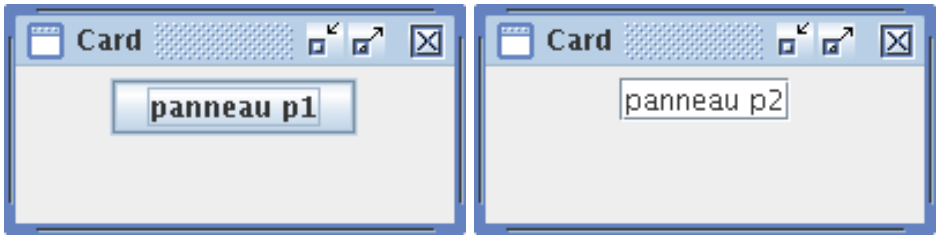


FIG. 13 – GridLayout

Card

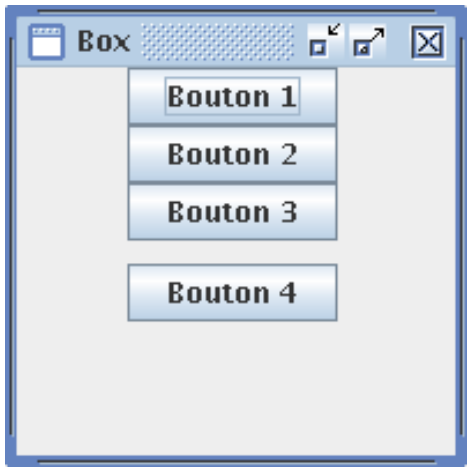
```
package gui;

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

public class LayoutCardDemo extends JFrame implements ActionListener
{
    final static String NOM1 = "panneau1";
    final static String NOM2 = "panneau2";
    JButton b1;
    JTextField t2;

    public LayoutCardDemo(String titre)
    {
        super(titre);
        getContentPane().setLayout(new CardLayout());
        JPanel p1 = new JPanel();
        p1.add(b1 = new JButton("panneau p1"));
        b1.addActionListener(this);
        JPanel p2 = new JPanel();
        p2.add(t2 = new JTextField("panneau p2"));
        t2.addActionListener(this);
        getContentPane().add(p1,NOM1);
        getContentPane().add(p2,NOM2);
        setSize(200,100);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
      String nom = (e.getSource() == b1)? NOM2 : NOM1;
      ((CardLayout)this.getContentPane().getLayout()).show(
      this.getContentPane(),nom);
    }

    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        LayoutCardDemo f = new LayoutCardDemo("Card");
    }
}
```

Card



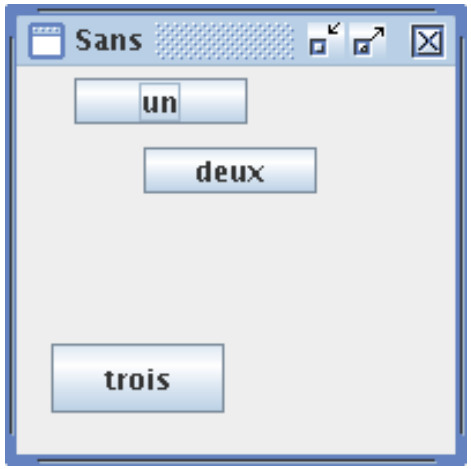
Box

```
package gui;

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

public class LayoutBoxDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f = new JFrame("Box");
        Container c = f.getContentPane();
        c.setLayout(new BoxLayout(c,BoxLayout.Y_AXIS));
        JButton b1 = new JButton("Bouton 1");
        b1.setAlignmentX(Component.CENTER_ALIGNMENT);
        c.add(b1);
        JButton b2 = new JButton("Bouton 2");
        b2.setAlignmentX(Component.CENTER_ALIGNMENT);
        c.add(b2);
        JButton b3 = new JButton("Bouton 3");
        b3.setAlignmentX(Component.CENTER_ALIGNMENT);
        c.add(b3);
        c.add(Box.createRigidArea(new Dimension(0,10)));
        JButton b4 = new JButton("Bouton 4");
        b4.setAlignmentX(Component.CENTER_ALIGNMENT);
        c.add(b4);
        f.setSize(200,200);
        f.setVisible(true);
    }
}
```

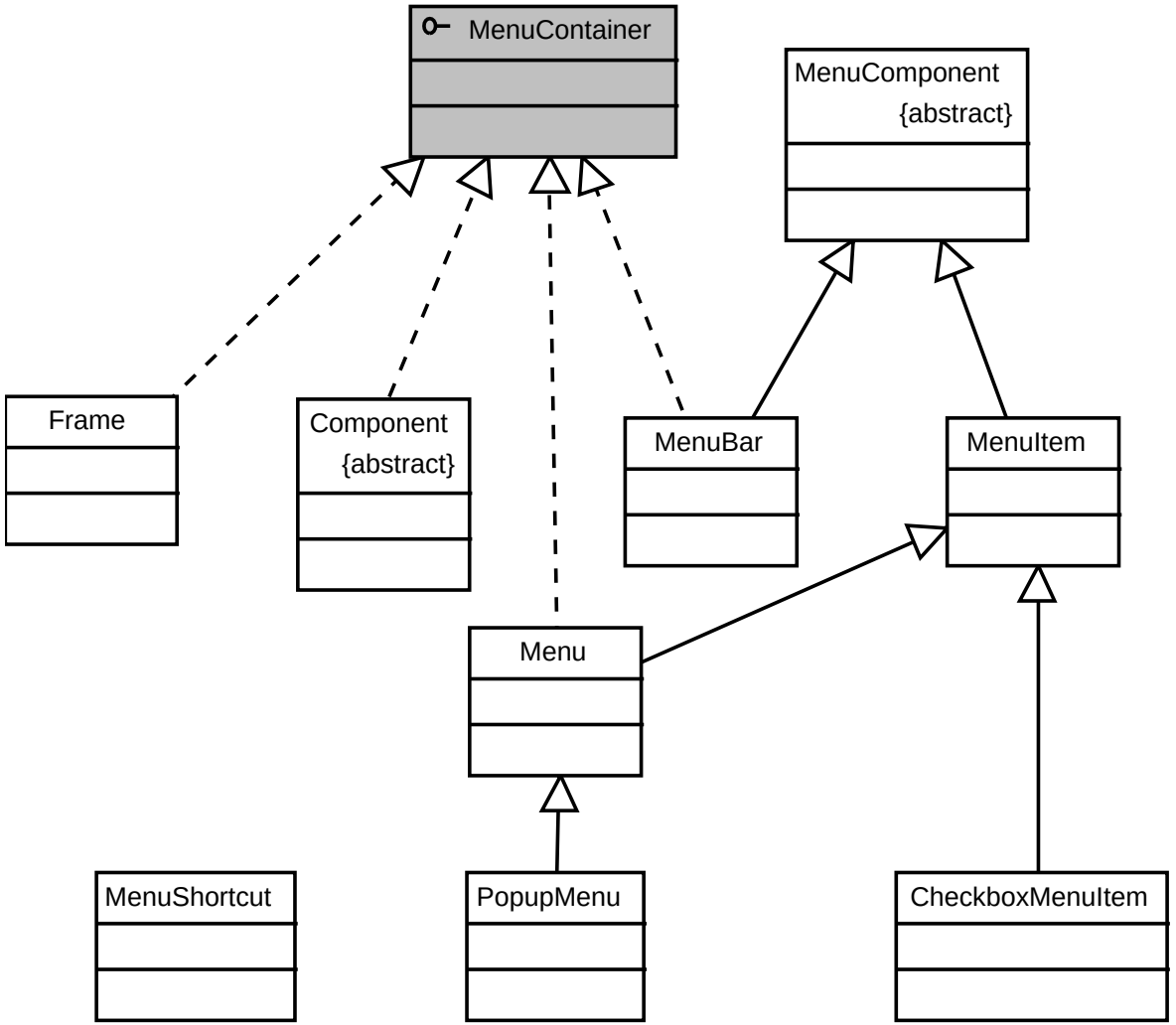


Sans Layout

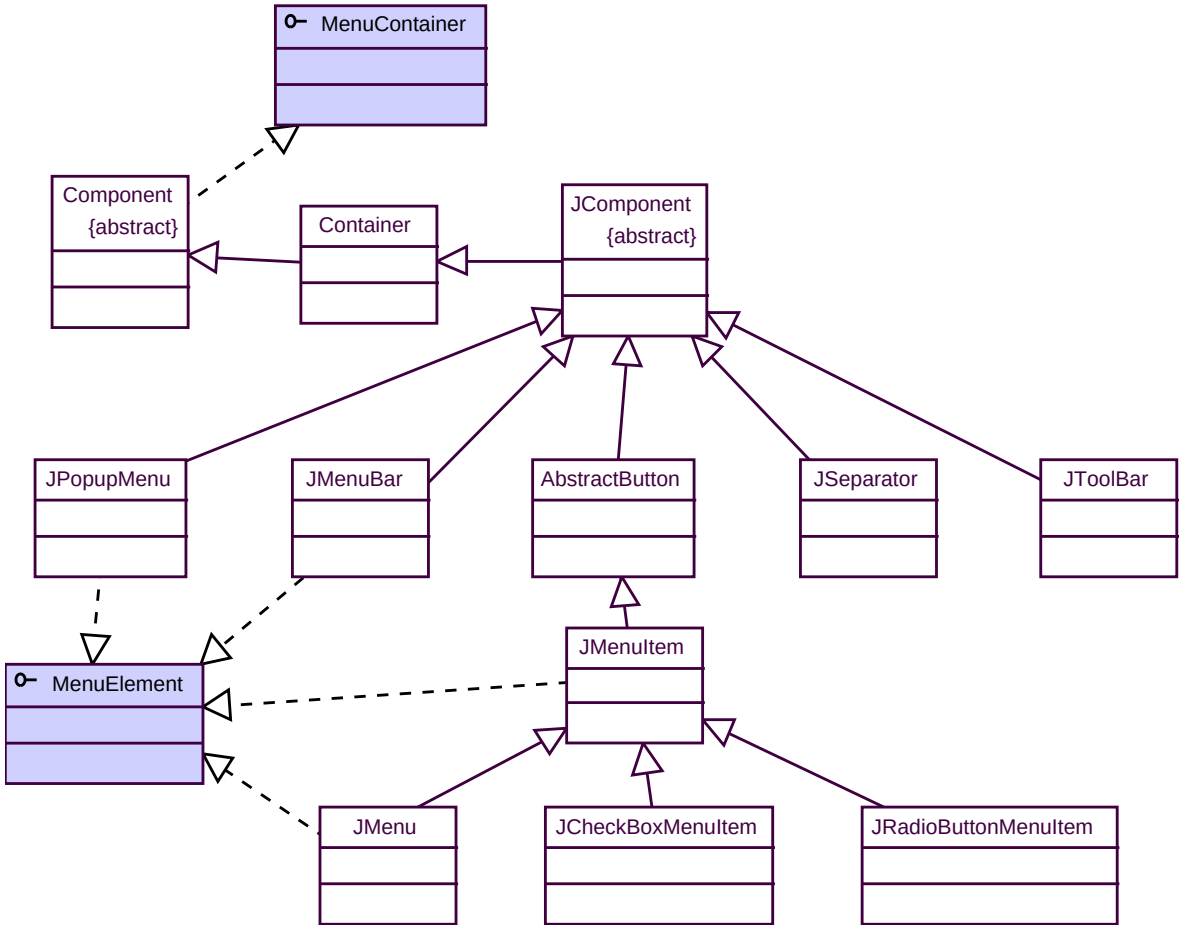
```
package gui;

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

public class LayoutSansDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JButton b1, b2, b3;
        JFrame f = new JFrame("Sans");
        f.setLayout(null);
        b1 = new JButton("un");
        f.add(b1);
        b2 = new JButton("deux");
        f.add(b2);
        b3 = new JButton("trois");
        f.add(b3);
        Insets insets = f.getInsets();
        b1.setBounds(25 + insets.left, 5 + insets.top, 75, 20);
        b2.setBounds(55 + insets.left, 35 + insets.top, 75, 20);
        b3.setBounds(15 + insets.left, 120 + insets.top, 75, 30);
        f.setSize(200,200);
        f.setVisible(true);
    }
}
```



Menu (AWT)


MenuContainer
MenuComponent
{abstract}
Frame
Component
{abstract}
MenuBar
MenuItem
Menu
MenuShortcut
PopupMenu
CheckboxMenuItem

Menu (Swing)


MenuContainer
Component
{abstract}
Container
JComponent
{abstract}
JPopupMenu
JMenuBar
AbstractButton
JSeparator
JToolBar
MenuElement
JMenuItem
JMenu
JCheckBoxMenuItem
JRadioButtonMenuItem

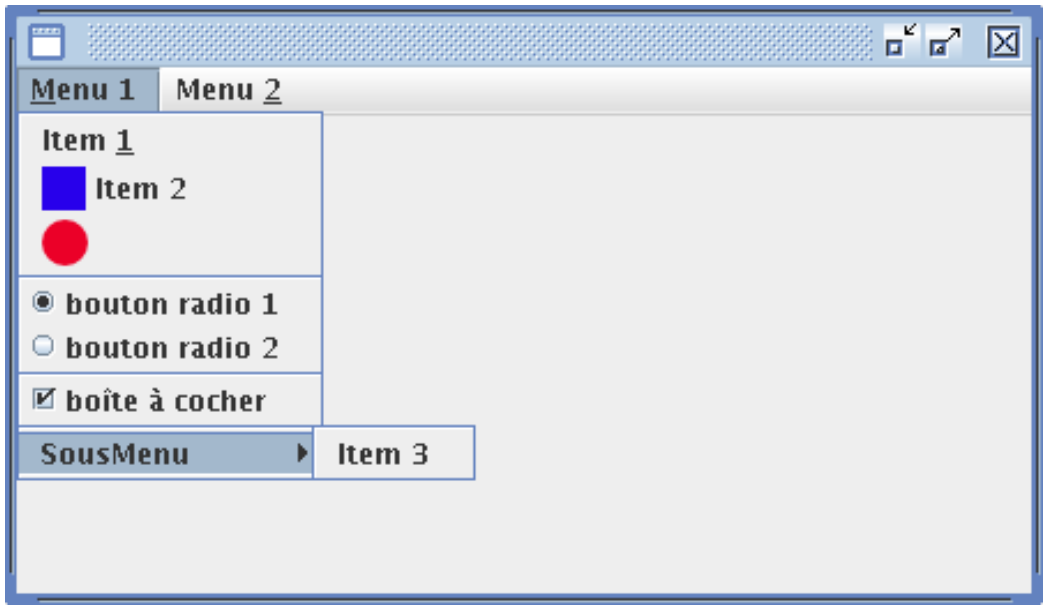
Composants des menus

- Classes séparées
- Une (J)Frame peut avoir une barre de menu
- Une barre de menu contient des menus
- Un menu peut avoir des sous-menus.
- Un menu est composé d'items de menu.
- On peut associer des racourcis claviers aux menus

Menus

```
package gui;
import java.awt.event.KeyEvent;
import javax.swing.*;

public class JMenuDemo
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f = new JFrame("JMenuDemo");
        JMenuBar barre = new JMenuBar();
        f.setJMenuBar(barre);
        JMenu menu = new JMenu("Menu 1 ");
        menu.setMnemonic(KeyEvent.VK_M);
        barre.add(menu);
        JMenuItem itemMenu = new JMenuItem("1", KeyEvent.VK_1);
        menu.add(itemMenu);
        itemMenu = new JMenuItem("2",
                new ImageIcon("gui/rect.png"));
        menu.add(itemMenu);
        ButtonGroup group = new ButtonGroup();
        JRadioButtonMenuItem br =
            new JRadioButtonMenuItem("bouton radio 1");
        group.add(br);
        menu.add(br);
        br = new JRadioButtonMenuItem("bouton radio 2",true);
        group.add(br);
        menu.add(br);
        JCheckBoxMenuItem cb =
            new JCheckBoxMenuItem("boîte à cocher",true);
        menu.add(cb);
        menu.addSeparator();
        JMenu sousMenu = new JMenu("SousMenu");
        itemMenu = new JMenuItem("3");
        sousMenu.add(itemMenu);
        menu.add(sousMenu);
        menu = new JMenu("Menu 2");
        menu.setMnemonic(KeyEvent.VK_2);
        barre.add(menu);
        f.setSize(250, 200);
        f.setVisible(true);
    }
}
```

Menus


Menu 1
Menu 2
Item 1
Item 2
bouton radio 1
bouton radio 2
boîte à cocher
SousMenu
Item 3

Gestion des événements : modèle par délégation

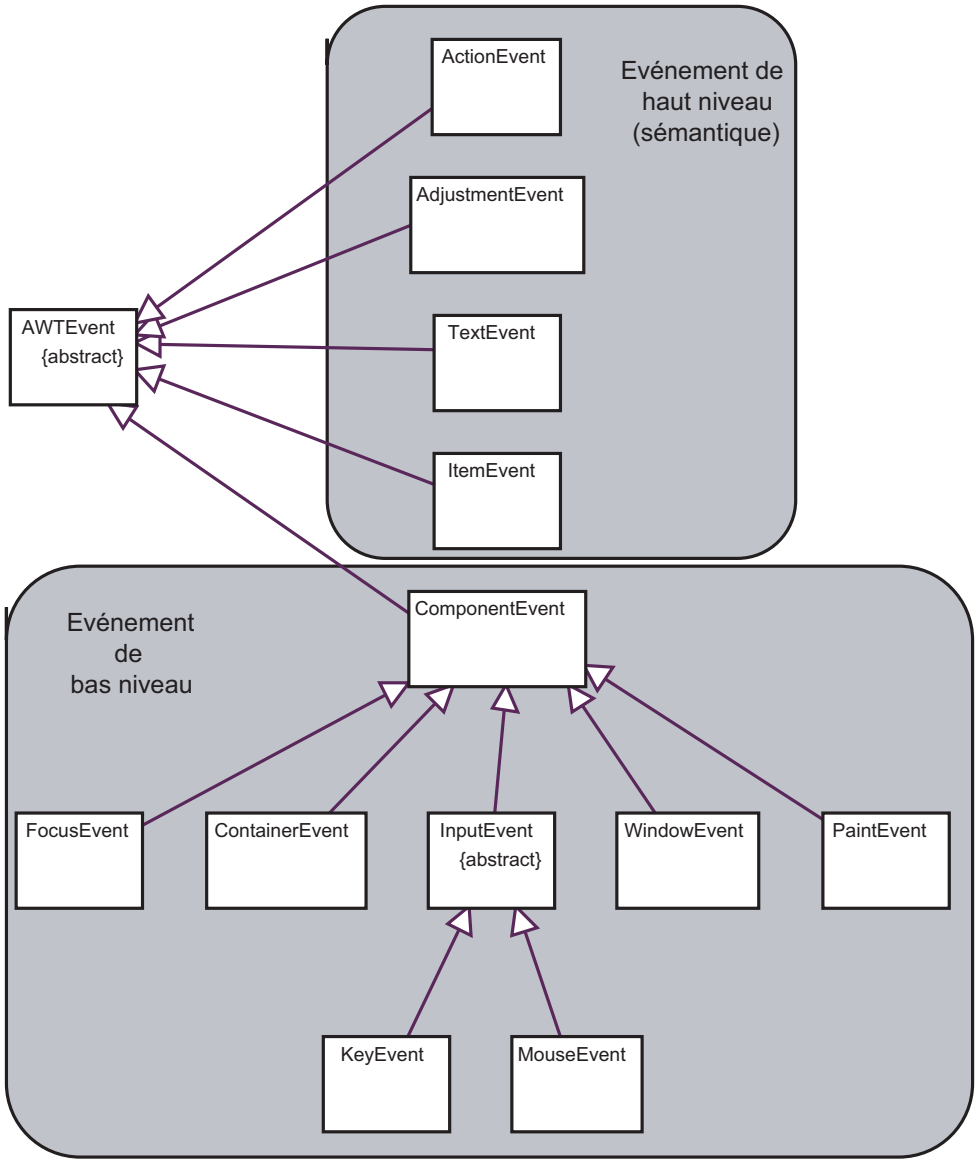
Gestion des événements :

- associer un comportement à une action
- action : clic sur un bouton, passage de souris, saisie, ...

Modèle par délégation :

- une action sur un `Component` génère un évenement (`AWTEvent`)
- cet événement est propagé d'une SOURCE (*Event Source*) vers une ou plusieurs DESTINATIONS (*Event Listeners*).
- une destination (*listener*) est un objet qui implémente une ou plusieurs interfaces de la hiérarchie d'interface `EventListener`.
- une source est un objet de l'interface graphique (bouton, Checkbox,...).

Modèle par délégation



Modèle par délégation

Événements de bas niveau :

- `ComponentEvent` : *resized, moved ...*
- `ContainerEvent` : *added, removed*
- `FocusEvent` : *focus gained, focus lost*
- `KeyEvent` : *key-pressed, key-released*
- `MouseEvent` : *mouse-down, mouse-move ...*
- `WindowEvent` : *activated, closed ...*

Modèle par délégation

Événements sémantiques :

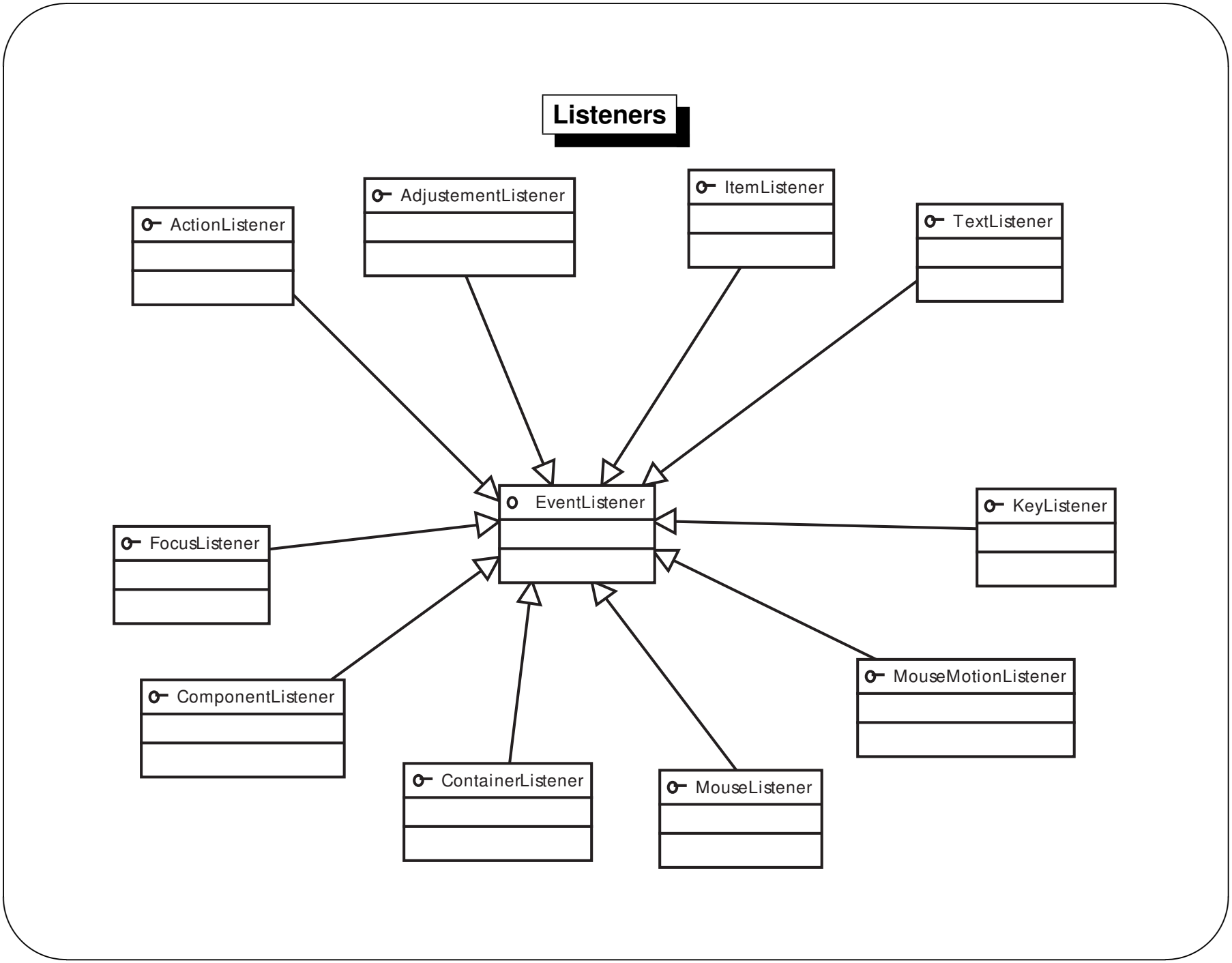
- `ActionEvent` : faire une commande
- `AdjustementEvent` : une valeur a été ajustée
- `ItemEvent` : l'état d'un item a changé
- `TextEvent` : l'état d'une zone de saisie de texte (`TextComponent, TextArea`) a changé

Gestion des événements

- L'utilisateur peut éventuellement rajouter ses propres type d'événements dans ce modèle.
- Gestion d'une queue d'événements gérée par un thread qui assure la distribution des événements aux objets.
- Tous les événements générés sont donc placés dans cette queue.

`java.awt.EventQueue :`

```
public static EventQueue getEventQueue()

public synchronized void postEvent(AWTEvent e)

public synchronized AWTEvent getNextEvent()

public synchronized AWTEvent peekEvent()

public synchronized AWTEvent peekEvent(int eventID)
```


Listeners
AdjustementListener
ItemListener
ActionListener
TextListener
EventListener
KeyListener
FocusListener
MouseMotionListener
ComponentListener
ContainerListener
MouseListener

Listeners

- Un `Listener` ne consomme pas d'événement (read-only pour le listener), il en prend note seulement.
- Une classe peut décider d'implémenter un ou plusieurs de ces listeners auquel cas il lui faut implanter toutes les méthodes définies dans ces interfaces.
- Exemple :

```
public interface KeyListener extends EventListener
{
        public void keyTyped(KeyEvente);
        public void keyPressed(KeyEvente);
        public void keyReleased(KeyEvente);
}
```

Sources

Destination de bas niveau :

Component	addComponentListener(ComponentListener l) addFocusListener(FocusListener l) addKeyListener(KeyListener l) addMouseMotionListener(MouseMotionListener l)
Container	addContainerListener(ContainerListener l)
Dialog	addWindowListener(WindowListener l)
Frame	addWindowListener(WindowListener l)

Ajouter des destinations pour des événements sémantiques :

Button	addActionListener(ActionListener l)
Choice	addItemListener(ItemListener l)
Checkbox	addItemListener(ItemListener l)
CheckboxMenuItem	addItemListener(ItemListener l)
List	addActionListener(ActionListener l) addItemListener(ItemListener l)
MenuItem	addActionListener(ActionListener l)
Scrollbar	addAdjustmentListener(AdjustmentListener l)
TextComponent	addTextListener(TextListener l)
TextField	addActionListener(ActionListener l)

Modèle par délégation

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

class Traitement0 implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("c'est le bouton ");
    }
}

public class DelDemo0
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f= new JFrame();
        f.setLayout(new FlowLayout());

        Traitement0 trait=new Traitement0();

        JButton go=new JButton("Go !");
        go.addActionListener(trait);

        f.add(go);
        f.pack();
        f.setVisible(true);
    }
}
```

Modèle par délégation

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Traitement implements ActionListener, FocusListener
{
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() instanceof JButton)
            System.out.println("c'est le bouton ");
        else
            System.out.println("c'est le champs de texte ");
    }
    public void focusGained(FocusEvent e)
    {
        System.out.println("je t'ai eu");
    }
    public void focusLost(FocusEvent e)
    {
        System.out.println("je t'ai perdu");
    }
}
public class DelDemo1
{
    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f= new JFrame();
        f.setLayout(new FlowLayout());
        Traitement trait=new Traitement();
        JButton go=new JButton("Go !");
        go.addActionListener(trait);
        JTextField tf= new JTextField(20);
        tf.addActionListener(trait);
        tf.addFocusListener(trait);
        f.add(go);
        f.add(tf);
```

```
        f.pack();

        f.setVisible(true);
    }
}
```

Modèle par délégation

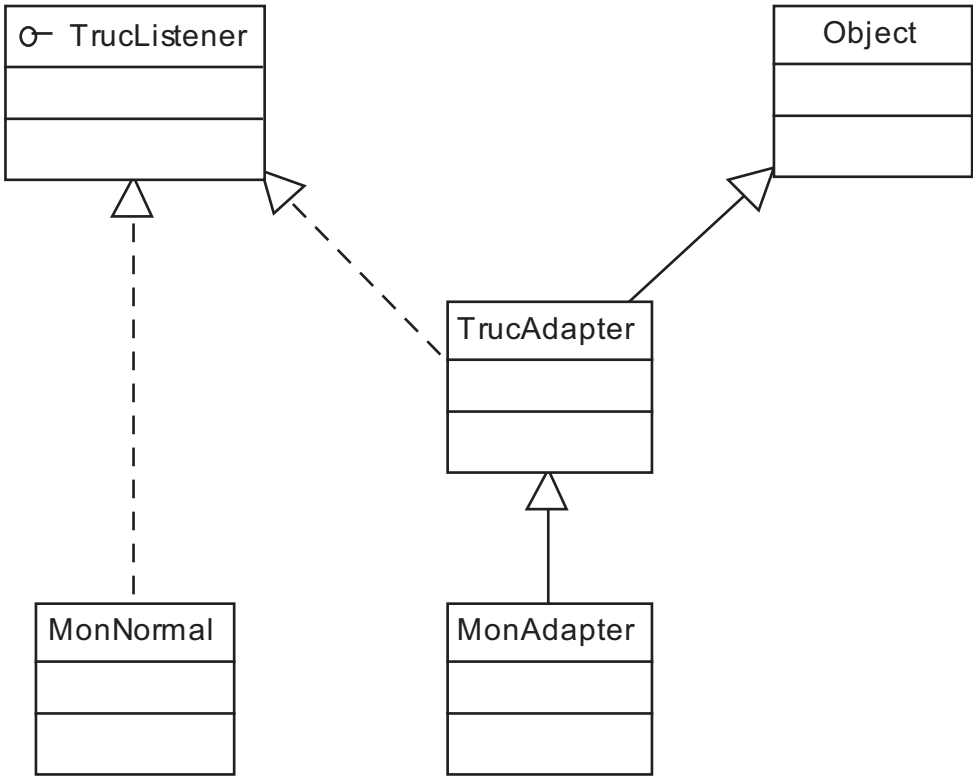
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class DelDemo2 extends JFrame
{
    JButton go;
    JTextField tf;
    JLabel l;

    class Traitement implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            l.setText(tf.getText());
        }
    }

    public DelDemo2()
    {
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Traitement trait=new Traitement();
        go =new JButton("Go !");
        go.addActionListener(trait);

        tf= new JTextField(20);
        tf.addActionListener(trait);

        l = new JLabel("                              ");
        add(go);
        add(tf);
        add(l);
        pack();
        setVisible(true);
    }

    public static void main(String[] args)
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        DelDemo2 demo = new DelDemo2();
    }

}
```

Les classes *Adapter*

Un des problèmes :

- Une classe implémentant une interface ***listener*** doit a priori implanter toutes les méthodes définies dans l'interface $\Rightarrow$ un peu pénible parfois ...
- Les **classes** ***adapter*** proposent une implantation par défaut de toutes les méthodes définies dans les interfaces des ***listeners*** de bas-niveau.
- L'utilisateur définit alors sa classe d'application comme sous-classe de la classe ***adapter*** et surcharge juste la méthode qui l'intéresse.

Les classes *Adapter*



les classes *Adapters*

```
package gui;

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

class ClavierListener implements KeyListener
{
    public void keyTyped(KeyEvent e)
    {
        System.out.print(e.getKeyChar());
    }

    public void keyPressed(KeyEvent e)
    {
        System.out.print(" *");
    }

    public void keyReleased(KeyEvent e)
    {
        System.out.print("* ");
    }

}

public class SansAdapter
{
    public static void main(String args[])
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f= new JFrame();
        ClavierListener cla = new ClavierListener();
        f.addKeyListener(cla);
        f.setSize(100,100);
        f.setVisible(true);
    }
}
```

les classes *Adapters*

```
package gui;

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

class ClavierAdapter extends KeyAdapter
{
    public void keyTyped(KeyEvent e)
    {
        System.out.print(e.getKeyChar());
    }
}
```

```
public class Adapter
{
    public static void main(String args[])
    {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame f= new JFrame();
        ClavierAdapter cla = new ClavierAdapter();
        f.addKeyListener(cla);
        f.setSize(100,100);
        f.setVisible(true);
    }
}
```

Avantages du nouveau modèle

- Séparation application/UI
- Pas besoin de sous-classer les composant graphiques
- Seuls les événements "écouté" sont créés à l'exécution (filtrage)

Graphique 2D

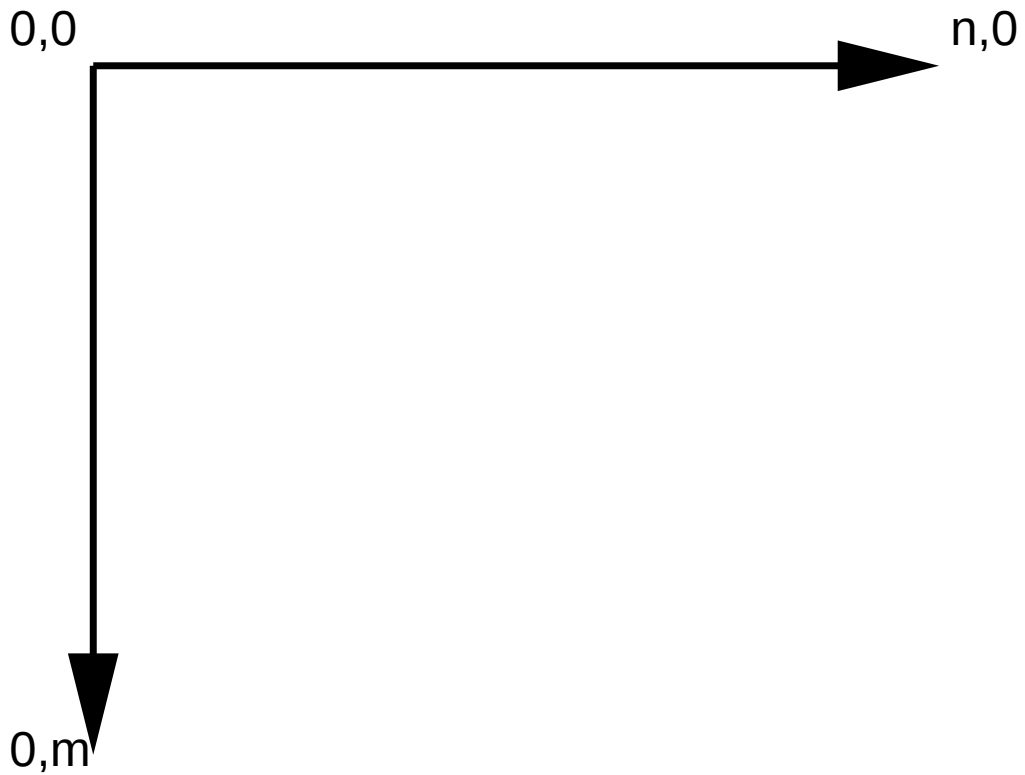
- composant (`Component` et sous classes) : objet **graphique**
- representation à l'écran → `paint(Graphics g)`
- `paint` peut être déclenchée de deux manières :
  1. par le système qui fournit le contexte graphique
  2. par l'utilisateur → `repaint()`

Graphique 2D : principe

- sous classer un `Component` (`Canvas` qui est fait pour ça)
- redéfinir la méthode `paint(Graphics g)`
- utiliser l'api de `Graphics` qui permet à une application de faire du dessin 2d : texte, image, rectangle, lignes ...

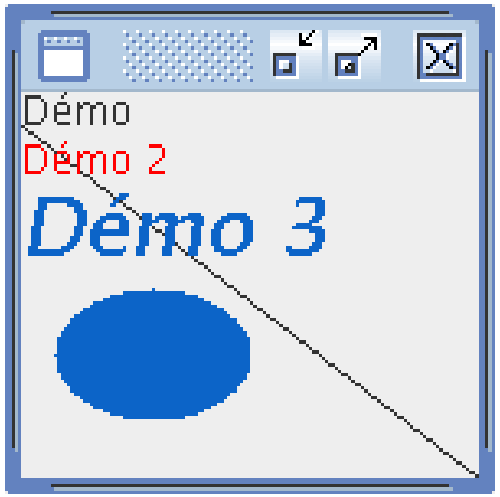
Système de coordonnées

- "local" au composant



Graphique 2D

```
import java.awt.*;

class MonCanvas extends Canvas
{
    public void paint(Graphics g)
    {
        Dimension d = getSize();
        g.drawLine(0, 10, d.width, d.height);
        // x1, y1, x2, y2
        g.drawString("Démo", 0, 10);
        g.setColor(Color.RED);
        g.drawString("Démo 2", 0, 25);
        g.setColor(new Color(12,100, 200));
        g.setFont(new Font("SansSerif",
                           Font.ITALIC+Font.BOLD,24));
        g.drawString("Démo 3", 0, 50);
        g.fillOval(10,60,60,40);
        System.out.println("PAINT");
    }
}

public class SimpleGraphicDemo
{
    public static void main(String args[])
    {
        Frame f= new Frame();
        f.add(new MonCanvas());
        f.setSize(150,150);
        f.setVisible(true);
    }
}
```



Graphique 2D

- Graphics limité $\longrightarrow$ Graphics2
- apiGraphics2 plus riche
- principe :

```
public void paint(Graphics g)
{
   Graphics2D g2 = (Graphics2D) g;
   g2. ...
```

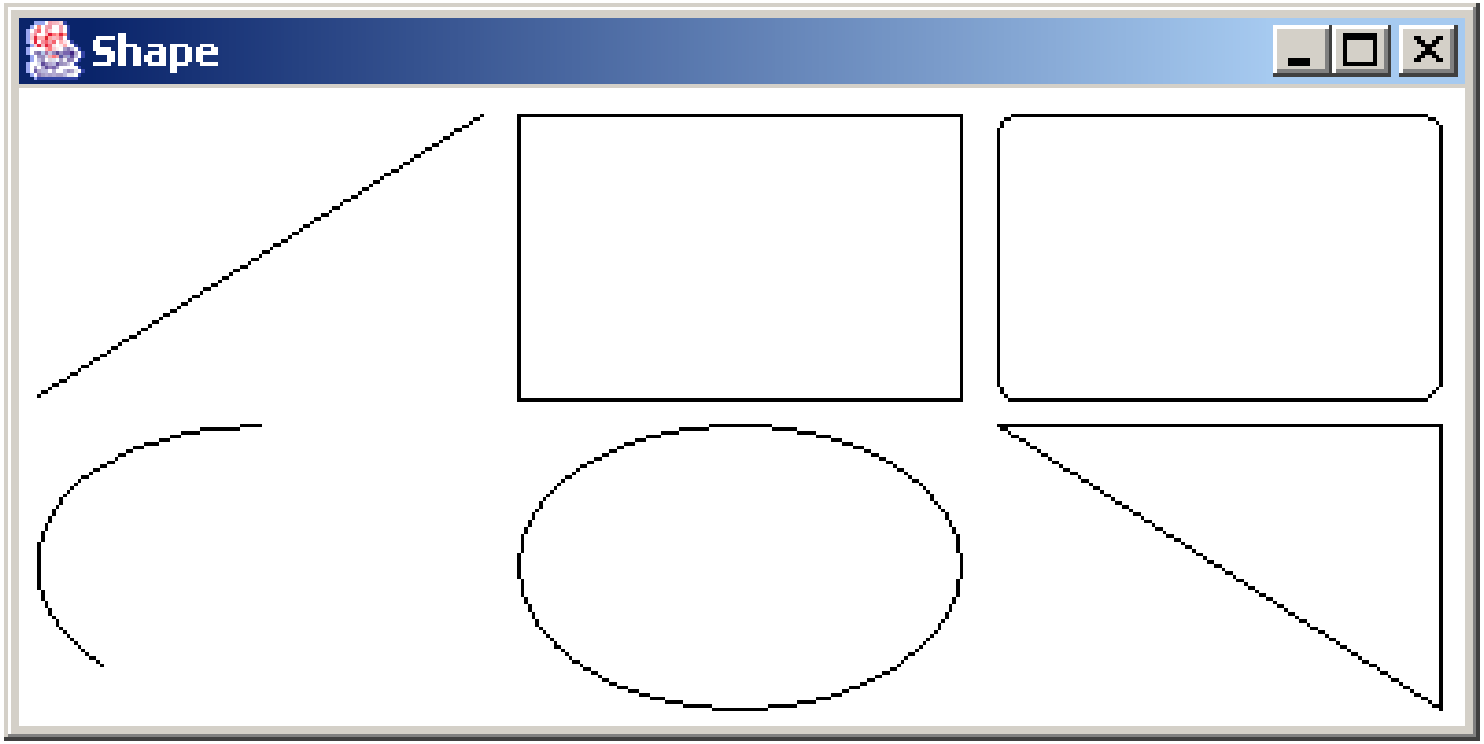
Forme

- interface `Shape`
- package `java.awt.geom` : implantation par `CubicCurve`, `Arc2D`, `Ellipse2D`, `GeneralPath`, `Line2D`, `Rectangle2D`, `RoundRectangle2D`
- méthode `draw` pour les dessiner

Graphique 2D

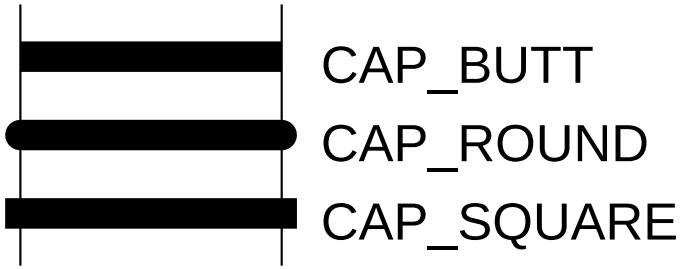
```
package gui;

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

public class ShapeDemo extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Dimension d = getSize();
        int x = 5;
        int y = 7;
        int largeur = d.width / 3 - 2*x;
        int hauteur =    d.height / 2 - y - 2;

        Graphics2D g2 = (Graphics2D) g;
        g2.draw(new Line2D.Double(x, y+hauteur-1,x + largeur, y));
        x += largeur+10;
        g2.draw(new Rectangle2D.Double(x, y, largeur, hauteur));
        x += largeur+10;
        g2.draw(new RoundRectangle2D.Double(x, y,largeur,hauteur,10, 10));
        x = 5;
        y += hauteur+7;
        g2.draw(new Arc2D.Double(x,y,largeur,hauteur,90, 135,Arc2D.OPEN));
        x += largeur+10;
        g2.draw(new Ellipse2D.Double(x, y,largeur,hauteur));
        x += largeur+10;
        GeneralPath triangle = new GeneralPath();
        triangle.moveTo(x,y);
        triangle.lineTo(x+largeur,y+hauteur);
        triangle.lineTo(x+largeur,y);
        triangle.closePath();
        g2.draw(triangle);
    }

    public static void main(String[] args)
    {
        JFrame f = new JFrame("Shape");
        f.getContentPane().add(new ShapeDemo());
        f.setSize(250,200);
        f.setVisible(true);
    }
}
```

Shape



Trait – contour

- interface `Stroke` implanté par `BasicStroke`
- permet de définir le style des lignes
- méthode `setStroke` pour appliquer à un `Graphics2D`
- les attributs du style sont les suivants :
  1. épaisseur, pointillé
  2. terminaison



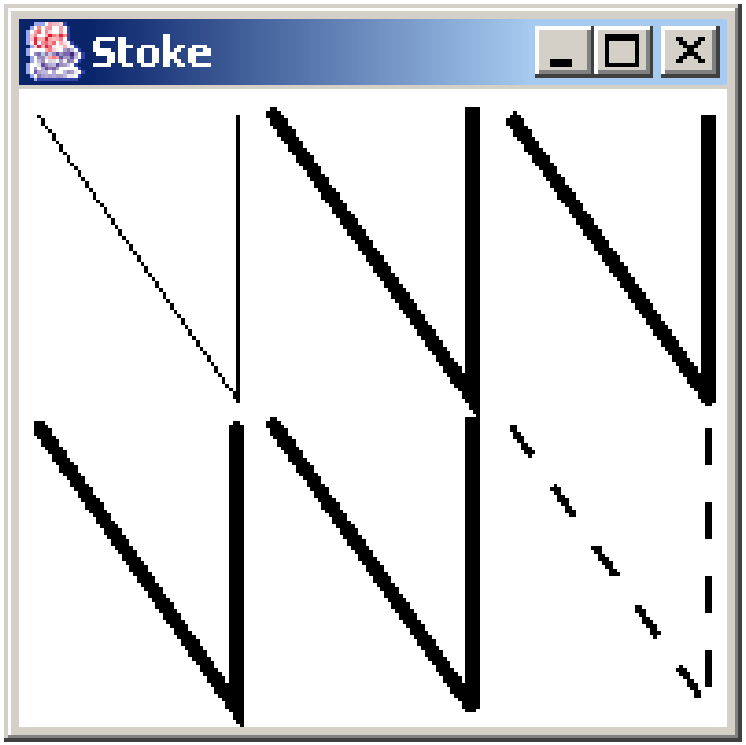
  3. jonction :

Graphique 2D

```
package gui;

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

public class StrokeDemo extends JPanel
{
    public void paint(Graphics g)
    {
        Dimension d = getSize();
        int x = 5;
        int y = 7;
        int largeur = d.width / 3 - 2*x;
        int hauteur = d.height / 2 - y - 2;

        Graphics2D g2 = (Graphics2D) g;

        g2.setStroke(new BasicStroke());
        g2.draw(ligne(x,y,largeur,hauteur));
        x += largeur+10;
        g2.setStroke(new BasicStroke(4));
        g2.draw(ligne(x,y,largeur,hauteur));
        x += largeur+10;
        g2.setStroke(new BasicStroke(4,BasicStroke.CAP_BUTT,
                                     BasicStroke.JOIN_ROUND));
        g2.draw(ligne(x,y,largeur,hauteur));
        x = 5;
        y += hauteur+7;
        g2.setStroke(new BasicStroke(4,BasicStroke.CAP_ROUND,
                                     BasicStroke.JOIN_MITER));
        g2.draw(ligne(x,y,largeur,hauteur));
        x += largeur+10;
        g2.setStroke(new BasicStroke(4,BasicStroke.CAP_SQUARE,
                                     BasicStroke.JOIN_BEVEL));
        g2.draw(ligne(x,y,largeur,hauteur));
        x += largeur+10;
        float[] p = {10.0f};
        g2.setStroke(new BasicStroke(2,BasicStroke.CAP_BUTT,
                                     BasicStroke.JOIN_MITER,
                                     10.0f,p, 0.0f));
        g2.draw(ligne(x,y,largeur,hauteur));
    }

    public GeneralPath ligne(int x, int y, int largeur, int hauteur)
    {
```

```
        GeneralPath ligne = new GeneralPath();
        ligne.moveTo(x,y);
        ligne.lineTo(x+largeur,y+hauteur);
        ligne.lineTo(x+largeur,y);
        return(ligne);
    }

    public static void main(String[] args)
    {
        JFrame f = new JFrame("Stroke");
        f.getContentPane().add(new StrokeDemo());
        f.setSize(250,200);
        f.setVisible(true);
    }
}
```

Stroke



Remplissage

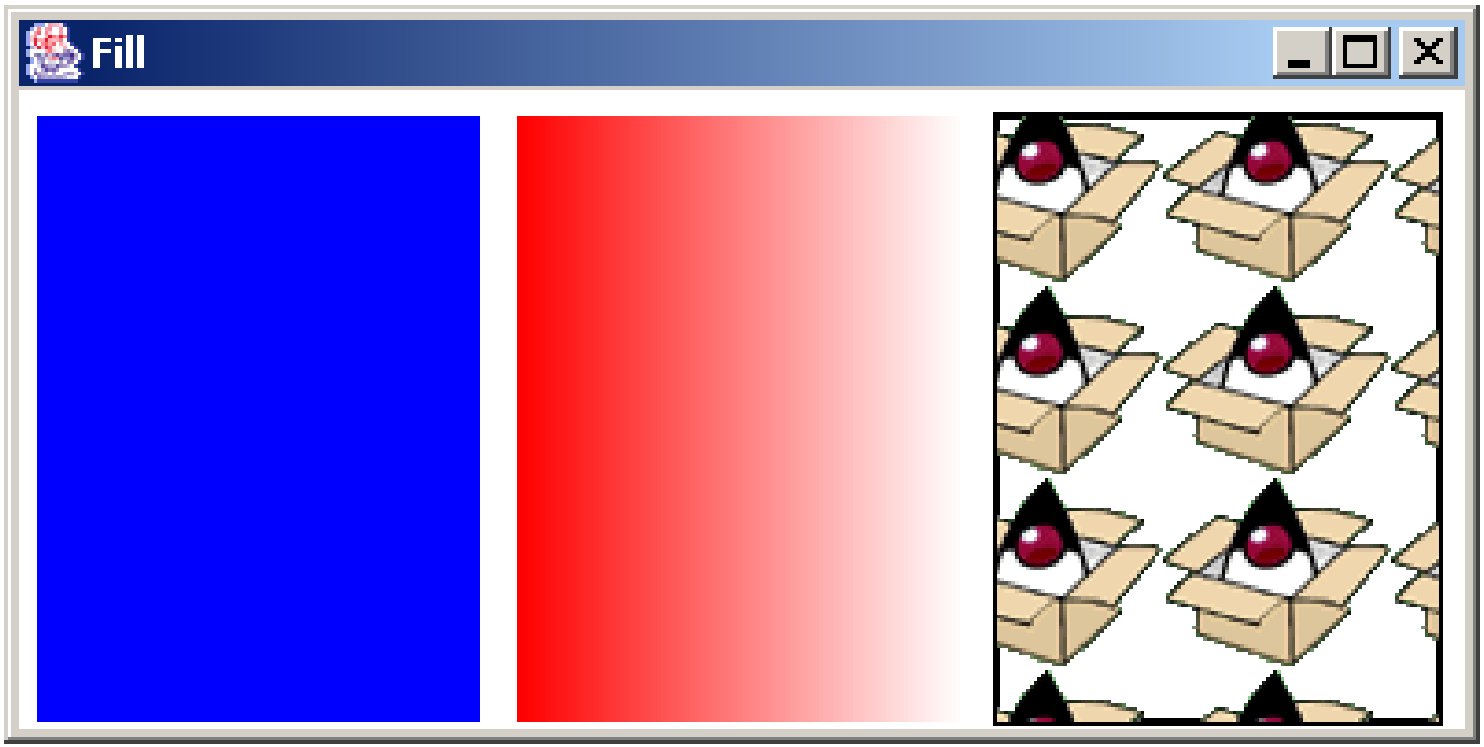
- interface `Paint`
- implanté par implanté par `Color`, `GradientPaint`, `TexturePaint`
- permet de définir la couleur de remplissage des formes
- méthode `setPaint` pour appliquer à un `Graphics2D`

Graphique 2D

```
package gui;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.*;

public class FillDemo extends JPanel
{
    public void paint(Graphics g)
    {
        Dimension d = getSize();
        int x = 5;
        int y = 7;
        int largeur = d.width / 3 - 2*x;
        int hauteur =    d.height - y - 2;

        Graphics2D g2 = (Graphics2D) g;

        g2.setPaint(Color.BLUE);
        g2.fill(new Rectangle2D.Double(x, y, largeur, hauteur));
        x += largeur+10;
        GradientPaint coul = new GradientPaint(x,y,Color.RED,
                                               x+largeur, y,Color.WHITE);
        g2.setPaint(coul);
        g2.fill(new Rectangle2D.Double(x, y, largeur, hauteur));
        x += largeur+10;

        g2.setStroke(new BasicStroke(2));
        g2.setPaint(Color.BLACK);
        g2.draw(new Rectangle2D.Double(x, y, largeur, hauteur));
        File f = new File("gui","box.gif");
        BufferedImage monImage = null;
        try {monImage = ImageIO.read(f);}
        catch (IOException ioe) {}
        TexturePaint coul2 = new TexturePaint(monImage,
                                    new Rectangle2D.Double(0,0,62,52));
        g2.setPaint(coul2);
        g2.fill(new Rectangle2D.Double(x, y, largeur, hauteur));

    }

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

```
        JFrame f = new JFrame("Fill");
        f.getContentPane().add(new FillDemo());
        f.setSize(300,200);
        f.setVisible(true);
    }
}
```

Fill



Graphique 2D

- interface `Composite` et méthode `setComposite` : recouvrement de deux objets
- interface `Transform` et méthode `setTransform` : transformation (rotation, translation, mise à l'échelle, ...)
- méthode `setClip` : définition de la partie visible d'une forme
- classe `Font` et méthode `setFont` : définition des polices
- ...

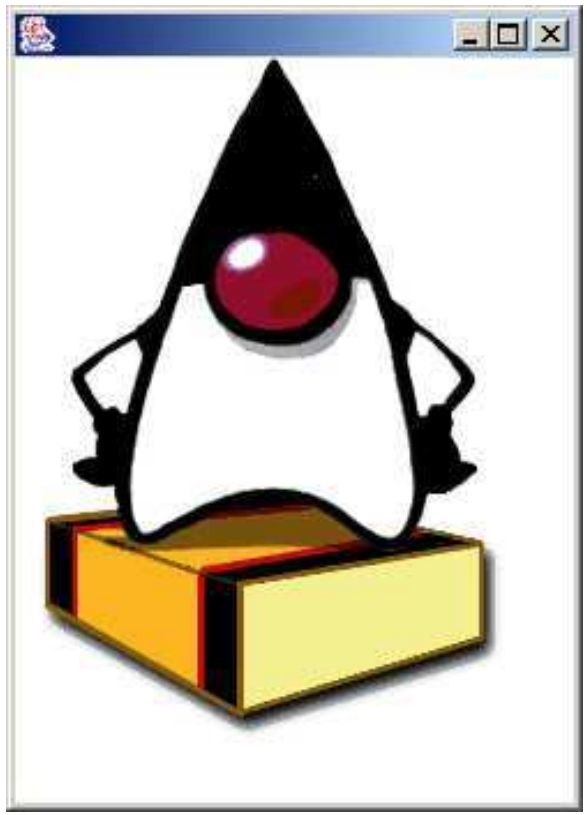
Images

- à l'origine une classe `Image`
- une méthode `drawImage`
- maintenant : `BufferedImage` : modèle de couleur + données
- Éventuellement `BufferedImageOp` pour faire des transformations sur l'image.

Images

```
package gui;

import javax.swing.*;
import java.awt.*;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.*;

class MonCanvasI extends JPanel
{
    BufferedImage monImage;

    public MonCanvasI() throws IOException
    {
        File f = new File("gui","duke.gif");
        monImage = ImageIO.read(f);
    }

    public void paint(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        g2.drawImage(monImage,null,0,0);
    }
}

public class ImageDemo
{
    public static void main(String[] args) throws IOException
    {
        JFrame f= new JFrame();
        f.add(new MonCanvasI());
        f.setSize(250,350);
        f.setVisible(true);
    }
}
```

Images



Plan du cours

- Introduction
- Langage
- Entrée/Sortie
- Threads
- Interface graphique
- **Applet**
- Perspective - Conclusion

Applet

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.

- programme java non autonome
- embarqué dans une autre application
- modèle de diffusion :
  - chargées par le réseau
  - présentées via une page HTML
  - exécutées dans un navigateur

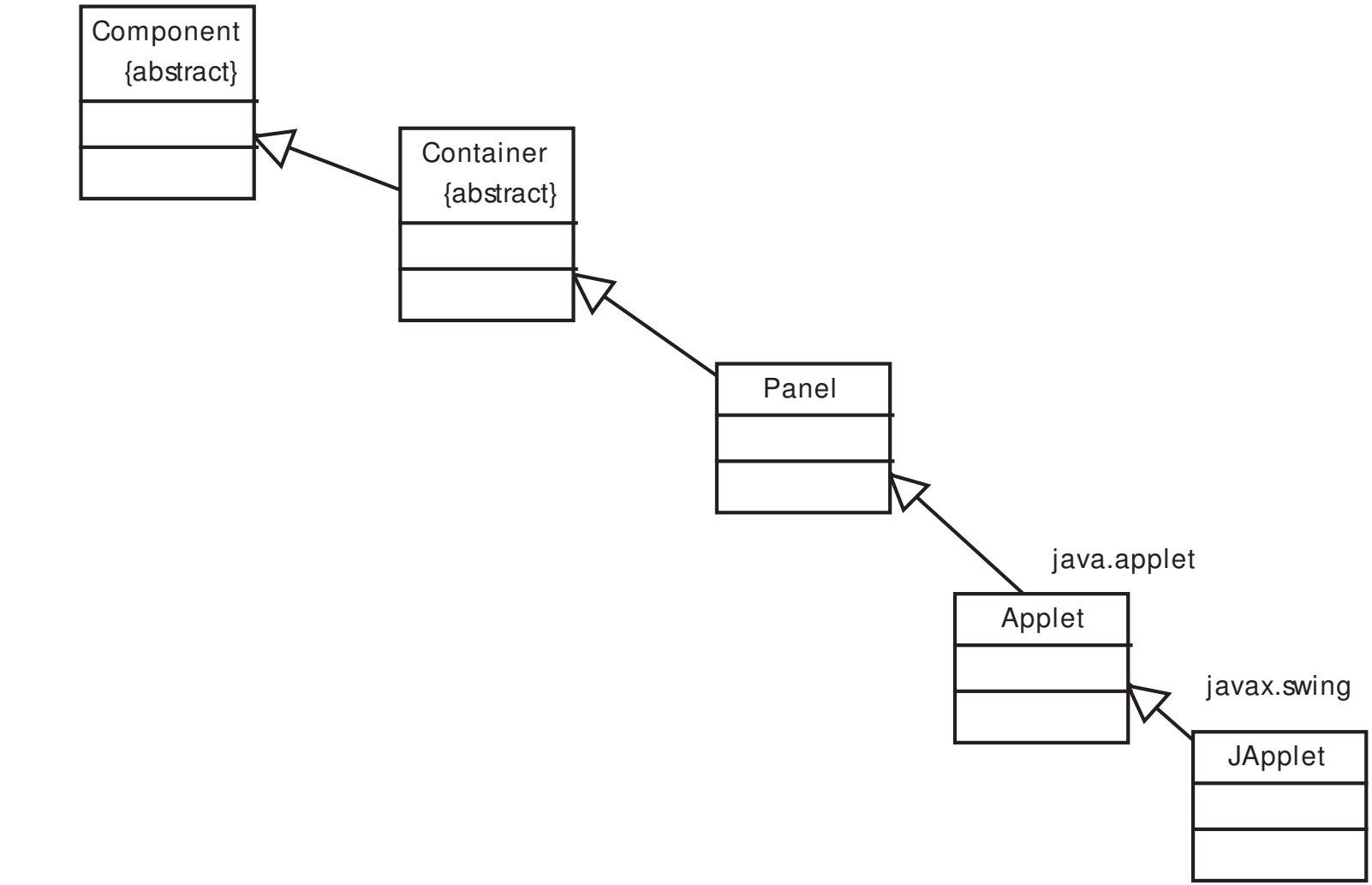
Applet

Une applet est une forme **restreinte** d'application

- Forcément un GUI → AWT ou Swing
- Forcément une SOUS-CLASSE de
  1. pour l'AWT : `java.applet.Applet` (une applet EST une `Applet`)
  2. pour Swing : `javax.swing.JApplet` (sous classe de `java.applet.Applet`)
- des restrictions pour la sécurité
  1. pas de lecture de certaines propriétés système
  2. pas d'accès au système de fichier local
  3. pas de connexions réseaux sauf vers le serveur de provenance
- un cycle de vie particulier de l'applet

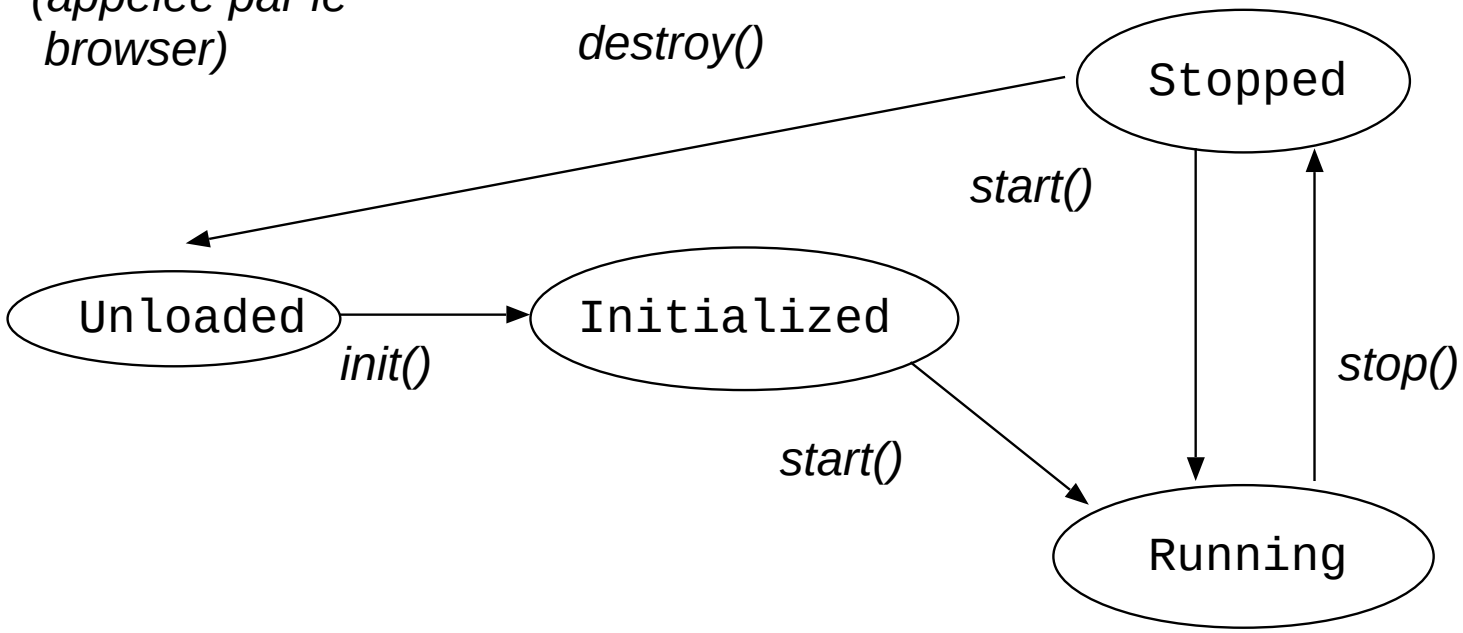
(J)Applet

C'est un composant graphique !



Cycle de vie d'une applet

Méthodes de la classe Applet (appelée par le browser)



Exemple

```
package applet;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class AppletUn extends JApplet
{
    JButton bt1;
    JButton bt2;
    JLabel l;

    public void init()
    {
        setLayout(new FlowLayout());
        bt1 = new JButton("Go 1 !");
        add(bt1);
        bt2 = new JButton("Go 2 !");
        add(bt2);
        l = new JLabel("                    ",JLabel.CENTER);
        add(l);
        Trait tt = new Trait();
        bt1.addActionListener(tt);
        bt2.addActionListener(tt);
    }

    class Trait implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            if (e.getSource() == bt1)
                l.setText("Re-Initialisation");
            else if (e.getSource()== bt2)
                l.setText("Ca marche");
            else
                System.out.println("???");
        }
    }
}
```

Applet et HTML

```
<html>
  <head>
    <title>Applet</title>
  </head>
  <body>
    <h1>Applet</h1>
    <applet code="applet.AppletUn.class"
      codebase=".."
      width="300" height="300">
      Commentaire
    </applet>
  </body>
</html>
```

```
<html>
  <head>
    <title>Applet</title>
  </head>
  <body>
    <h1>Applet</h1>
    <object codetype="application/java"
      classid="java:applet.AppletUn"
      codebase=".."
      width="300" height="300">
      Commentaire.
    </object>
  </body>
</html>
```

Exemple

```
package applet;
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class AppletCycle extends JApplet
{
    DefaultListModel lesAppels;

    public AppletCycle()
    {
        setLayout(new FlowLayout());
        lesAppels = new DefaultListModel();
        lesAppels.addElement("Constructeur");
        JList jl = new JList(lesAppels);
        JScrollPane jsp = new JScrollPane(jl);
        jsp.setPreferredSize(new Dimension(200, 150));
        add(jsp);
    }
    public void paint(Graphics g)
    {
        lesAppels.addElement("paint()");
    }
    public void init()
    {
        lesAppels.addElement("init()");
    }
    public void start()
    {
        lesAppels.addElement("start()");
    }
    public void stop()
    {
        lesAppels.addElement("stop()");
    }
    public void destroy()
    {
        lesAppels.addElement("destroy()");
    }
}
```

Passage de paramètres : déf.

```
<html>
  <head>
    <title>Demo Param&egrave;tres Applet</title>
  </head>
  <body>
      <h1>Passage de param&egrave;tres &agrave; une Applet</h1>
    <object  codetype="application/java" codebase=".."
      classid="java:applet.AppletParam" width=320 height=100>
      <param name=texte value="premiere inclusion">
    </object>
    <object  codetype="application/java" codebase=".."
      classid="java:applet.AppletParam" width=320 height=100>
      <param name=texte value="deuxieme inclusion">
    </object>
    <object  codetype="application/java" codebase=".."
      classid="java:applet.AppletParam" width=320 height=100>
    </object>
  </body>
</html>
```

Passage de paramètres : récupération

```
package applet;
import javax.swing.*;

public class AppletParam extends JApplet
{
  String aAfficher;

  public void init()
  {
    aAfficher = getParameter("TEXTE");
    if (aAfficher == null)
      aAfficher = "Il faut un parametre";
    JLabel jl = new JLabel(aAfficher,JLabel.CENTER);
    add(jl);
  }
}
```

Applets et communication

- Une applet peut communiquer avec les autres applets de la même page HTML (et issu du même serveur)
- Une applet peut communiquer avec le browser (dans la limite des moyens du browser)
- Une applet peut communiquer avec le serveur fournissant l'applet.

Applet vs Applet

- la méthode `getAppletContext()` de la classe `Applet` renvoie un objet `AppletContext`
- la méthode `getApplet()` de la classe `AppletContext` renvoie un objet `Applet`

Applet vs Applet

```
<html>
  <head>
    <title>Demo Comm Applets</title>
  </head>
  <body>
      <h1>Demo communication d'Applets</h1>
      <object name="comm1" codetype="application/java" codebase=".."
      classid="java:applet.AppletComm1" width=100 height=100>
Votre butineur ne permet pas l'ex&eacute;cution d'applets
      </object>
      <object name="comm2" codetype="application/java" codebase=".."
      classid="java:applet.AppletComm2" width=100 height=100>
Votre butineur ne permet pas l'ex&eacute;cution d'applets
      </object>
  </body>
</html>
```

Applet vs Applet

```
package applet;
import javax.swing.*;

public class AppletComm1 extends JApplet
{
  JLabel label;

  public AppletComm1()
  {
    label = new JLabel("Initial",JLabel.CENTER);
  }

  public void init()
  {
    add(label);
  }
}
```

Applet vs Applet

```
package applet;
import javax.swing.*;
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class AppletComm2 extends JApplet implements ActionListener
{
    JButton un;
    JButton deux;

    public void init()
    {
        setLayout(new FlowLayout());
        un = new JButton("Un");
        deux = new JButton("Deux");
        add(un);
        un.addActionListener(this);
        add(deux);
        deux.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        String mess;
        if (e.getSource()==un)
            mess="Final";
        else if (e.getSource()==deux)
            mess="Initial";
        else
            mess="Pas Normal";
        Applet ap = this.getAppletContext().getApplet("comm1");
        ((AppletComm1)ap).label.setText(mess);
    }
}
```

Applet vs Applet

```
package applet;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import java.net.MalformedURLException;

public class AppletGo extends JApplet
{
    JButton go;

    public void init()
    {
        setLayout(new FlowLayout());
        go = new JButton("Go");
        add(go);
        Trait tt = new Trait();
        go.addActionListener(tt);
    }

    class Trait implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            try
            {
                getAppletContext().showDocument(
                    new URL("http ://www.google.fr"));
            }
            catch (MalformedURLException mue)
            {
                ;
            }
        }
    }
}
```

Applet vs Serveur

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

public class AppletCommServeurBof extends JApplet implements ActionListener
{
    JComboBox c;
    Connection cnx;
    JTextArea ta;

    public void init()
    {
        setLayout(new BorderLayout());
        c= new JComboBox();
        String url = "jdbc :postgresql ://127.0.0.1 :5432/db";

        try {
            Class.forName ("org.postgresql.Driver");
            cnx = DriverManager.getConnection(url,"db","db*");
            Statement stmt = cnx.createStatement();
            ResultSet rs = stmt.executeQuery ("SELECT * FROM fournisseur");

            while (rs.next ())
            {
                c.addItem (rs.getString("frs_nom"));
            }
            rs.close();
            stmt.close();
        }
        catch (Exception e)
        {
            System.err.println("Erreur " + e.getMessage());
        }
        JButton b = new JButton ("GO !");
        b.addActionListener(this);
        ta = new JTextArea(10,20);
        add (c, BorderLayout.NORTH);
        add (b,BorderLayout.SOUTH);
        add (ta,BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e)
```

```
{
    try
    {
        PreparedStatement pstmt = cnx.prepareStatement(
        "SELECT art_num, art_nom, frs_nom" +
        " FROM Fournisseur,Article" +
        " WHERE frs_num=art_frs AND frs_nom = ?");
        pstmt.setString(1,(String)c.getSelectedItem());
        ResultSet rs = pstmt.executeQuery();
        String ress = "";
        while (rs.next())
            {
                // On lit et affiche toutes les colonnes
                ress += rs.getString("art_num");
                ress += "\t";
                ress += rs.getString("art_nom");
                ress += "\n";
            }

        // On termine la requête et la connexion
        rs.close();
        pstmt.close();
        ta.setText(ress);
    }
// on traite les éventuelles exceptions
    catch (Exception ee)
    {
        System.err.println("Erreur " + ee.getMessage());
    }
}
```

Applet vs Serveur

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import java.net.URLConnection;
import java.io.*;
import java.util.Vector;

public class AppletGeniale extends JApplet implements ActionListener
{
    JComboBox cb;
    JButton bt;
    JList jl;

    public void init()
    {
        setLayout(new FlowLayout());
        cb = new JComboBox();
        try
            {
                URL url = new URL(getDocumentBase(),"AppletFrs.jsp");
                BufferedReader rep = new BufferedReader(
                                        new InputStreamReader (url.openStream ()));
                String s = null;
                while ((s=rep.readLine())!=null)
                    {
                        if (s.length() != 0)
                            cb.addItem (s);
                    }
            }
        catch (IOException ioe)
            {
                bt.setText("PB ! ! !");
            }

        bt = new JButton("OK");
        jl = new JList();
        add(cb);
        add(bt);
        add(jl);
        bt.addActionListener(this);
    }
```

```
public void actionPerformed(ActionEvent e)
{
    go();
}

public void go()
{
    try
        {
            URL url = new URL(getDocumentBase(),"AppletArt.jsp");
            URLConnection connexion = url.openConnection();
            connexion.setDoOutput(true); // on poste des données

            PrintWriter req = new PrintWriter(connexion.getOutputStream(),true);
            req.print("frs="+((String)cb.getSelectedItem()).trim());}
            req.flush();
            BufferedReader rep = new BufferedReader(
                                 new InputStreamReader(connexion.getInputStream()));
            Vector<String> v = new Vector<String>();
            String s = null;
            while ((s=rep.readLine())!=null)
                {
                    if (s.length()!= 0)
                        v.add(s);
                }
            jl.setListData(v);

        catch (IOException ioe)
            {
                bt.setText("PB 2!!!");
            }
        }
```

Restrictions

- Pas de méthodes natives
- Pas d'appels au système de fichier local
- Pas de mise en place de connexions réseau sauf avec le serveur d'origine
- Pas de lancement de programmes
- Pas d'accès aux propriétés du système
- Les fenêtres ouvertes par une applet sont différentes des autres ...

Conclusions

- Une applet est une forme restreinte d'application permettant son execution dans un browser.
- Permet la diffusion à large echelle... mais d'APPLET, pas d'APPLICATIONS !

Plan du cours

- Introduction
- Langage
- Entrée/Sortie
- Threads
- Interface graphique
- Applet
- **Perspective - Conclusion**

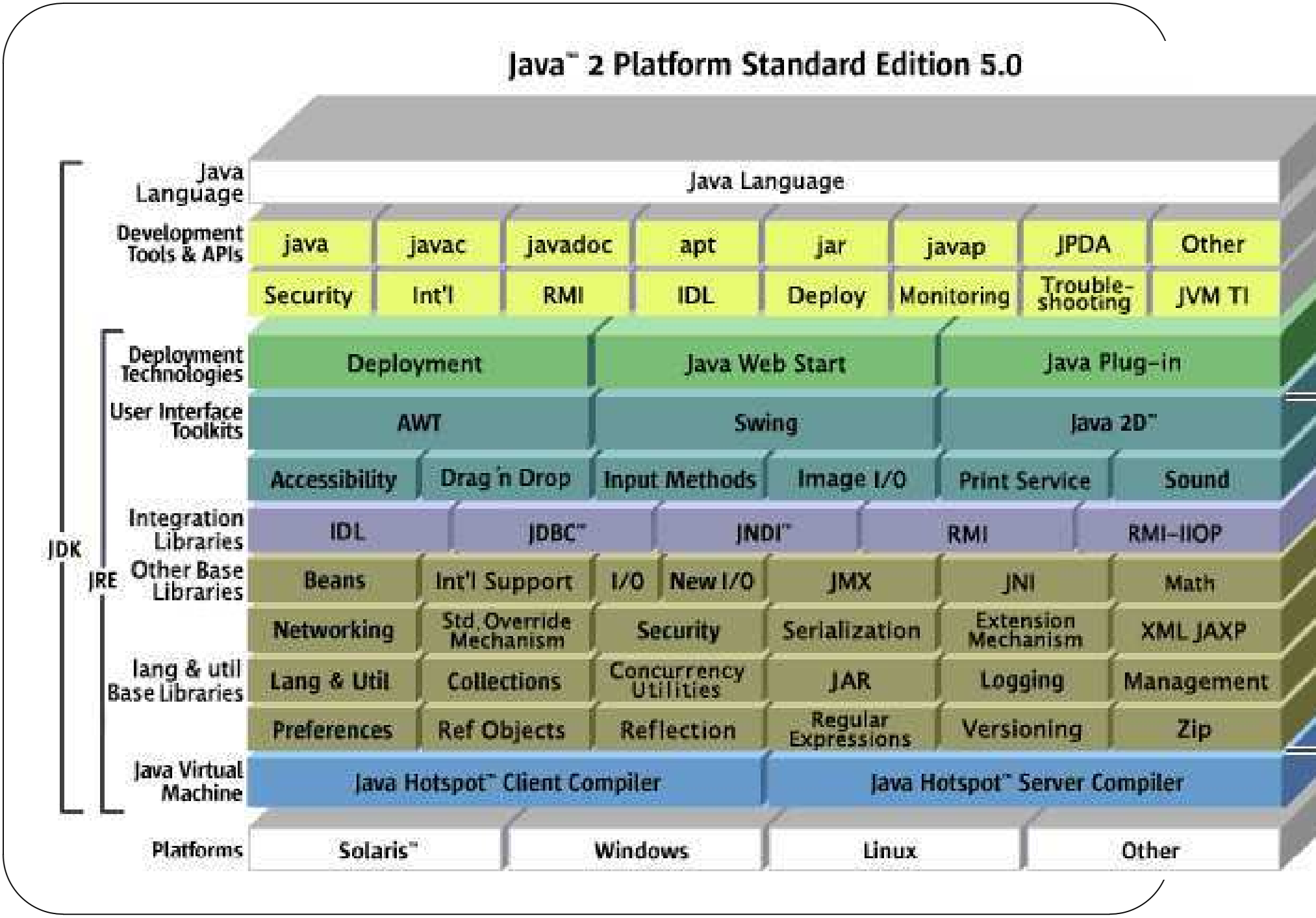


FIG. 14 – Récapitulatif

En perspectives

- J2SE
  - JDBC, RMI, JavaBeans, JavaIDL ..
  -
- des extensions standard
  - Java Advanced Imaging
  - Java Communications
  - Java Cryptography Extension
  - Java Data Object (JDO)
  - Java Naming and Directory Interface (JNDI)
  - Java Secure Socket Extension (JSSE)
  - Java Speech
  - Java 3D
  - Java Media Framework
- Java Telephony
- Java Entreprise Edition (J2EE)
  - Enterprise JavaBean
  - JavaBeans Activation Framework
  - Java Servlet et JavaServer Page
  - Java Message Service
  - Java Transaction

  - JavaMail
  - XML (JAXB JAXP JAXR JAX-RPC
- Java Micro Edition (J2ME)
- Java Card
- ...

JDBC

Java DataBase Connectivity :

- Écrire des applications bases de données complétement en JAVA.
- API pour envoyer des ordres SQL à n'importe quelle base de données RELATIONNELLE !

RMI

Remode Method Invocation :

- La version java des RPC ...
- Mise en oeuvre assez simple, : en fait un programme s'exécutant sur une JVM, peut prendre une référence sur objet instancié dans une autre machine java.
- Le système prend en charge tous les problèmes de sérialisation/déserialisation ...
- un système de nommage permet de connaitre les "remote object" instancié dans une JVM ...

JavaBeans

- le slogan : "Write once, run everywhere", en fait : un modèle de composants logiciels.
- l'objectif : assembler ces composants (quasiment des mini-applications) pour construire une application, voir son environnement de travail...
- Les composants logiciels sont assemblés de la même manière que la statique d'une interface graphique.

JavaBeans

Services attendus des composants :

- Déclaration d'interface et découverte à l'exécution des interfaces des autres composants ...
- Attributs
- Gestion d'événements
- Persistence (de l'assemblage des composants)
- support pour des constructeur d'APPLICATIONS
- Packaging

Servlet – JSP

- modules java s'exécutant du coté **serveur**
- équivalent des applets coté serveur (serv-let)
- à l'origine déstinés à JavaServer (ex Jeeves)
- intégrées dans la plupart des serveur web (apache)
- destinées à remplacer les CGI:
  - plus facile
  - plus rapide $\longrightarrow$ compilation
  - connexions bases de données standardisées (via JDBC)

$\longrightarrow$ tomcat

Conclusion générale

- Beaucoup plus qu'un langage ...
- une nouvelle alternative pour la construction et le déploiement d'applications complexes sur un réseau à large échelle.
- Le champ d'application est énorme. À priori, on peut trouver une machine java dans n'importe quoi :
  - un téléphone mobile
  - un équipement réseau
  - un network computer
  - un serveur de base de données ...
  - sa machine à laver? le distributeur de café ?

Conclusion générale

Les points forts :

- Indépendance par rapport à l'architecture
- Tout le confort moderne pour pour construire des applications complexes (AWT, RMI, JDBC, Beans ...) toujours en restant indépendant d'une architecture.
- Deploiement sur des réseaux à large échelle traité de manière globale (sécurité, maintenance ...)

Conclusion générale

Les points difficiles :

- Un JDK pas encore stabilisé ... des bugs nombreux ...
- L'indépendance vis à vis de l'architecture n'est pas encore acquise (voir initiative 100 pour 100 pure java)
- Encore des problèmes de performance (mais les JIT arrivent ...)

Conclusion générale

- Évidemment, les grands acteurs de l'informatique ont pris la mesure des bouleversements que peuvent générer une telle informatique.
- Une activité sans pareille autour de java : Corel a porte sa suite bureautique sous java, Notes est proposé en java, IBM annonce smartsuite en Java, Oracle intègre java dans sa version 8 ...
- Une nouvelle façon de déployer un système informatique ?? une alternative à *bip* ??

Plan du cours

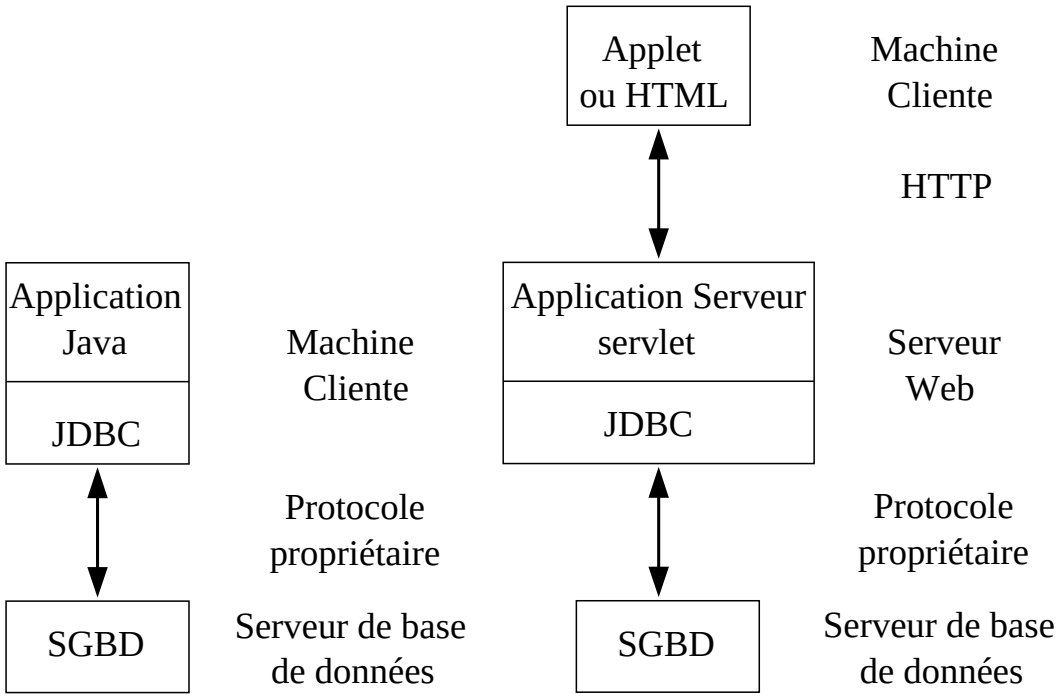
- JDBC

Java DataBase Connectivity

JDBC est la solution java d'accès aux bases de données. C'est une API d'accès aux bases de données :

- indépendante de la base de donnée
- de bas niveau $\Rightarrow$ SQL
- basée sur les travaux de l'X/Open SQL CLI (Call level Interface) (aussi à l'origine d'ODBC)
- regroupée dans le paquetage `java.sql` constitué essentiellement d'interfaces

Utilisation de JDBC



(a) Architecture 2 machines (b) Architecture 3 machines

FIG. 15 – Utilisation de JDBC

Pilotes

- le choix d'une API constituée d'interfaces ⇒
  1. pour utiliser un SGBD précis, il faut disposer d'un ensembles de classes qui implantent les interfaces du paquetage `java.sql.`
  2. pour le développeur, écriture d'un code générique, quelque soit le SGBD
  3. pour les éditeurs de produits SGBD, possibilité de fournir des classes dédiées à leur système
- Le cœur est une classe `Driver`

```
public class Driver implements java.sql.Driver
```

Types de pilotes

1. pont JDBC-ODBC : permet la connexion à la base via ODBC. Exemple : sur une base Access. Ce pibte est livré en "standard" avec Java
2. pibte faisant appel à des méthodes *natives* (écrites dans un autre langage que Java ; souvent en C). Exemple : oracle, ...
3. pibte pour travailler avec un service *middleware*. Exemple : Interbase, produit Symantec, ...
4. pibte tout Java faisant appel au protocole réseau du SGBD. Exemples : pibte pour les bases MySql, Msql, oracle, ...

Principe

L'utilisation de JDBC se fait en trois temps :

1. établir une connexion avec la base de donnée
2. envoyer des commandes SQL
3. traiter les éventuels résultats

Établissement d'une connexion

- connaître le nom (url) de la source de données
- associer et utiliser le pilote de SGBD adéquat $\Rightarrow$ charger la classe `Driver` appropriée
- obtenir une connexion en utilisant le "bon" pilote

Identification de la source

- source de donnée = pseudo url d'une base de donnée
- syntaxe proche d'un url sur la toile
- format : `jdbc:sous-protocole:url` ou
  `jdbc:sous-protocole://www.serveur.ex:port/nomSourceDeDonnée`
- le `sous-protocole` dépend du SGBD
- exemple :
  `jdbc:obbc:Exemple // url jamais distant`
  `jdbc:msql://192.168.168.197:1114/Exemple`
  `jdbc:mysql://Linux:3306/test_ex`
  `...`

Retour sur le CLASSPATH

- lorsque l'on veut charger des classes (fichiers `.class`) dans la JVM, celles-ci sont recherchés dans la variable `CLASSPATH` :
  - "vrais" répertoires contenant des fichiers `.class`
  - archives zip contenant des fichiers `.class`
  - archives spécifiques à java : jar *Java ARchives*. Ces fichiers peuvent contenir en plus des fichiers `.class` des images, des fichiers d'aides, ... et peuvent être signés.
- exemples :
  1. unix : chemins séparés par `:`, séparations des répertoires par des `/`

     `/usr/local/java/lib/classes.zip:/usr/local/mysql_jdbc/mysql.jar`
  2. windows : chemins séparés par `;`, séparations des répertoires par des `\`

     `C:\Internet\jdk\lib\classes.zip;C:\Internet\jsdk\lib\jsdk.jar`

Chargement d'une classe dans la JVM

- deux solutions :
  1. Déclaration d'une variable de la classe. Exemple :

```
String s;
java.util.Hashtable ht;
...
```

     Vérification de l'existence de la classe à la compilation et à l'exécution

  2. Utilisation de la méthode

```
public static native Class forName(String className)
        throws ClassNotFoundException;
```

     Exemple :

```
Class.forName("String");
```

     Vérification de l'existence de la classe à l'exécution

Chargement du pilote

- chargement de la classe du pibte en connaissant son nom
- Class.forName("nom Pibte")
- exemples :

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("com.imaginary.sql.msql.MsqlDriver");
Class.forName ("org.gjt.mm.mysql.Driver")
Class.forName ("twz1.jdbc.mysql.jdbcMysqlDriver");
...
```

Obtention d'une connexion

- La classe DriverManager est une classe gestionnaire de pilotes
- une "demande" de connexion se fait par la méthode `getConnection()` en spécifiant la source de donnée
- le pilote approprié est utilisé
- exemple :

```
String source = "jdbc:odbc:Exemple";
Connection conn = DriverManager.getConnection(source);
```

Envoi de commandes SQL

- Se fait par l'intermédiaire de l'objet `Statement` (ou une de ses deux sous-classes `PreparedStatement` ou `CallableStatement`)
- obtention de l'instance de Statement:

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

- les commandes SQL sont transmise à l'aide des méthodes `executeUpdate()` et `executeQuery()`

```
int nbLignesTouch = stm.executeUpdate("INSERT INTO fournisseurs
VALUES ('F06','LES STYLOS DIVISES')");
ResultSet rs = stm.executeQuery("SELECT * FROM fournisseurs");
```

Traitement des résultats

- le résultat d'une interrogation est stocké dans un `ResultSet`
- on accède aux résultats ligne par ligne
- dans une ligne chaque champ est accessible directement
- le `ResultSet` est doté de méthode d'accès au données : `getString()`, `getInt()`, ... et d'un méthode `next()` pour changer de ligne
- exemple :

```
while (r.next())
{
  int i = r.getInt("a");
  String s = r.getString(2);
  float f = r.getFloat("c");
  System.out.println("ligne = " + i + " " + s + " " + f);
}
```

Valeur NULL

- Attention NULL $\neq$ null
- NULL valeur SQL
- méthode wasNull() pour tester la NULLité
- exemple :

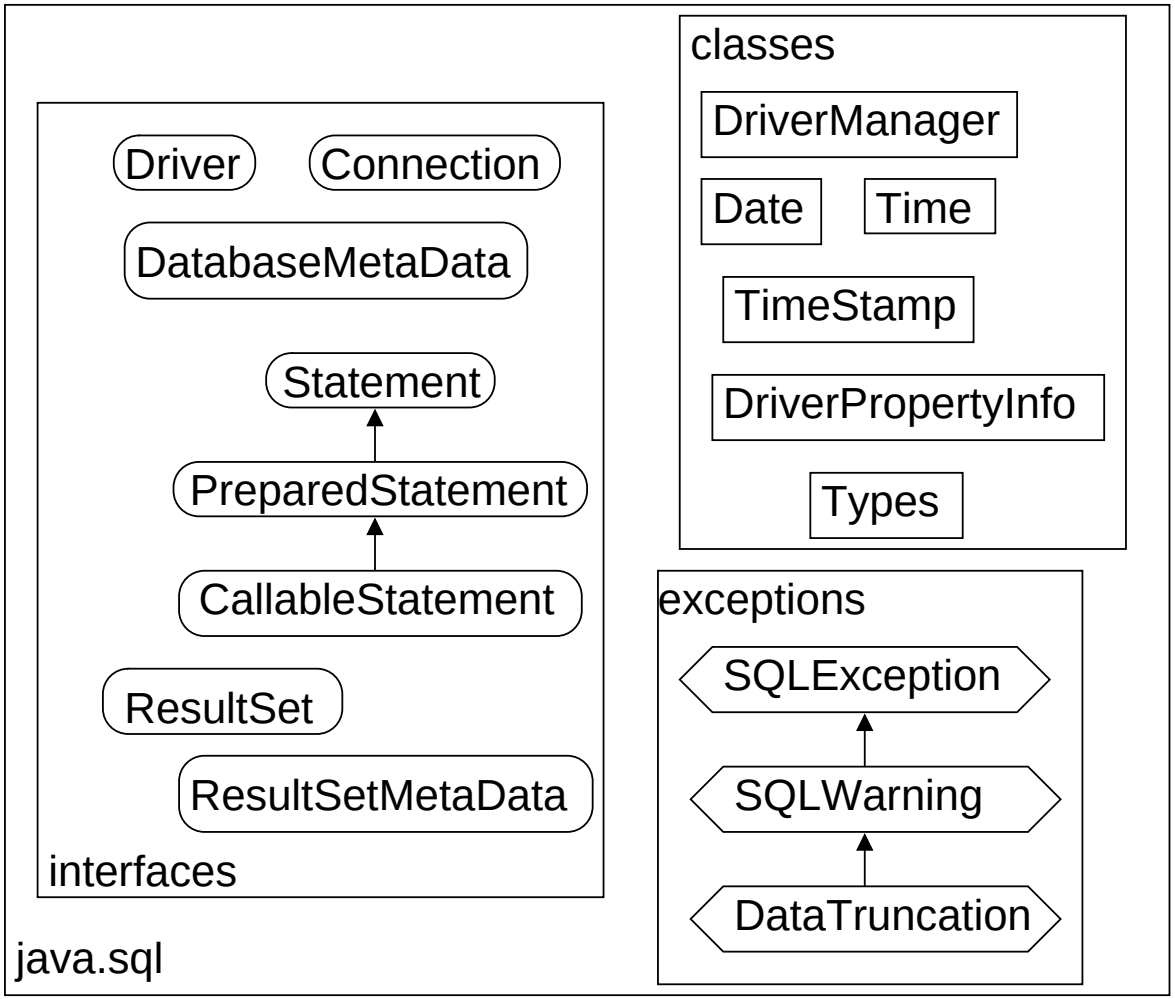
```
// rs de type ResultSet
String s = rs.getString(1);
if (rs.wasNull())
  System.out.println("s est nul");
...
```

Correspondance types Java – type SQL

type SQL	type Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long

type SQL	type Java
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Paquetage `java.sql`



Exemple récapitulatif

```
import java.sql.*;

public class SimpleJdbcDemo
{
    public static void main (String args[]) throws Exception
    {
        String url = "jdbc :postgresql ://sioux :5432/db";
        Class.forName ("org.postgresql.Driver");

        Connection cnx = DriverManager.getConnection(url,"db","db*");
        Statement stmt = cnx.createStatement();
        ResultSet rs = stmt.executeQuery ("SELECT * FROM fournisseur");

        while (rs.next ())
        {
            System.out.print(rs.getString(1));
            System.out.println(rs.getString("frs_nom"));
        }
        rs.close();
        stmt.close();
        cnx.close();
    }
}
```

Exemple

```
import java.sql.*;

public class SimpleJdbcDemo2
{
    public static void main (String args[])
    {
        // L'URL représentant la source de données
        String url = "jdbc :postgresql ://sioux :5432/db";

        try {
            // On charge le driver MySql
            Class.forName ("org.postgresql.Driver");

            // On réalise une connexion à la source de données
            Connection cnx = DriverManager.getConnection(url,"db","db*");
            // Création d'un objet Statement permettant d'exécuter du code SQL
            Statement stmt = cnx.createStatement();
            // Envoi de la requète et recupération du résultat
            ResultSet rs = stmt.executeQuery ("SELECT * FROM article");
            // utilisation de l'objet ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData ();
            int ncols = rsmd.getColumnCount ();

            while (rs.next ())
```

```
                {
                    // On lit et affiche toutes les colonnes
                    for (int i=1; i<=ncols; i++) {
                        if (i > 1) System.out.print("\t");
                        System.out.print(rs.getString(i));
                    }
                    System.out.println("");
                }
            //    On termine la requète et la connexion
                rs.close();
                stmt.close();
                cnx.close();
            }
            // on traite les éventuelles exceptions
            catch (SQLException ex) {
                System.err.println("Erreur SQL");
                System.err.println(ex.getMessage());
                ex.printStackTrace();
            }
            catch (ClassNotFoundException c) {
                System.err.println("Nom de classe invalide");
                System.err.println(c.getMessage());
                c.printStackTrace();
            }
        }
    }
```

Exemple PreparedStatement

```
import java.sql.*;
public class SimpleJdbcDemo3
{
    public static void main (String args[]) throws Exception
    {
        String url = "jdbc :postgresql ://sioux :5432/db";
        Class.forName ("org.postgresql.Driver");

        Connection cnx = DriverManager.getConnection(url,"db","db*");
        PreparedStatement pstmt =
            cnx.prepareStatement("SELECT * FROM article WHERE art_frs= ?");
        String frs = Clavier.readString();
        pstmt.setString(1,frs);
        ResultSet rs = pstmt.executeQuery ();
        while (rs.next ())
        {
            System.out.print(rs.getString(1));
            System.out.println(rs.getString("art_nom"));
        }
        rs.close();
        pstmt.close();
        cnx.close();
    }
}
```

Servlet

- application Java qui s'exécute du coté serveur
- équivalent écrit en java d'un programme CGI
- API standard Java
- nécessite un conteneur de servlets

Conteneur

- appelé aussi moteur de servlet
- Rôle :
  - traiter les requêtes clientes
  - instancier, initialiser la servlet
  - lui passer deux objets : `Request` et `Response`
  - renvoyer le résultat au client
- Deux types :
  1. intégré dans un serveur Web : mode autonome
  2. module supplémentaire d'un serveur Web

Avantages

- facilité de mise en œuvre
- performances : une servlet est un programme compilé alors que l'essentiel des CGI se font dans des langages interprété
- intégration avec toutes API Java (JDBC $\longrightarrow$ accès à des bases de données facile)
- portabilité...

Utilisation des servlets

- réponse à un formulaire
- écriture d'un document dynamiquement
- redirection de requêtes (dissimulation de l'adresse d'une machine sensible, équilibre de charge sur plusieurs serveurs, ...)
- synchronisation de requêtes concurrentes (conférences on-line)
- ...
- base de tout service web java

Interface `Servlet`

- classe "mère" de l'API
- toute servlet DOIT implanter cette interface
- méthodes :
  1. `init(ServletConfig)` : initialisation de la servlet
  2. `getServletConfig()` : paramètres d'initialisation
  3. `getServletInfo()` : information textuelle sur la servlet
  4. `service(ServletRequest,ServletResponse)` : traitement
  5. `destroy()` : déchargement de la servlet

Cycle de vie d'une servlet

Les servlet ont le cycle de vie suivant :

1. chargement et initialisations par le serveur : par la méthode `init()` `init()` n'est jamais réutilisé, sauf en cas de rechargement de la servlet pas le serveur
2. traitement de requêtes clientes : par la méthode `service(ServletRequest, ServletResponse.` A chaque requête cliente correspond un thread.
3. déchargement de la servlet (souvent à l'arrêt du programme serveur) précédé de l'exécution de `destroy()`

Interaction avec des clients

1. `ServletRequest`, communications du client vers le serveur,
2. `ServletResponse`, communications du serveur vers le client

Interface `ServletRequest`

- fournit des informations clientes : nom des paramètres transmis (`getParameterNames()`), valeurs des paramètres (`getParameter(String)`, `getParameterValues(String)`), nom de la machine cliente (`getRemoteHost()`, ...
- fournit un canal de communication `ServletInputStream` pour obtenir des données du client comme dans le cas d'une méthode `POST`

Interface `ServletResponse`

- fournit à la servlet le moyen de répondre au client.
- permet de positionner le type MIME des données (`setContentType(String)`) et leur longueur `setContentLength(int)`
- fournit un canal de communication vers le client, `ServletOutputStream getOutputStream()` ou `PrintWriter getWriter()` par lequel le serveur peut transmettre des données.

Paquetage Servlet

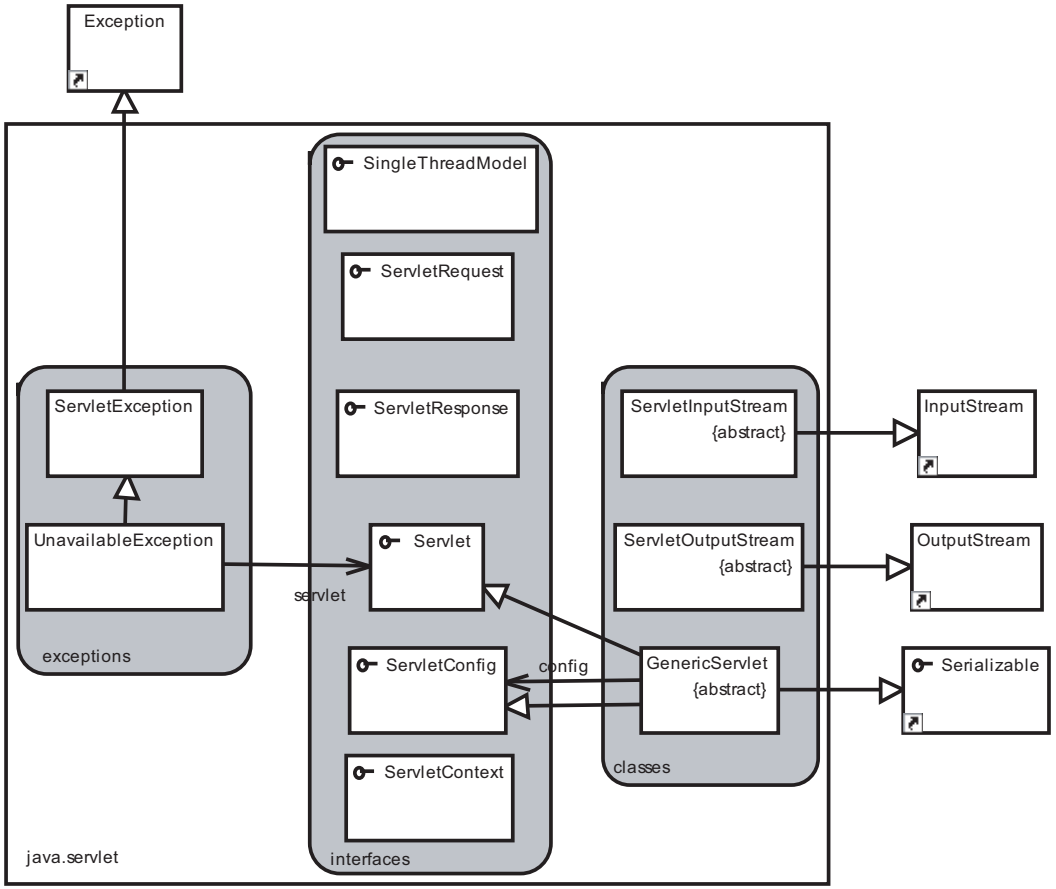


FIG. 16 – Le paquetage `java.servlet`

Outils logiciels

- tomcat est l'implantation de référence des servlets
- contient en particulier :
  1. un moteur de servlet (module et autonome)
  2. des exemples
  3. deux applications de configuration
  4. l'api des classes à utiliser
  5. les bibliothèques des paquetages des servlets.

Mise en œuvre

- implanter directement l'interface `Servlet`
- sous-classer GenericServlet ⇒ définir `service`)
- sous classer `HttpServlet`.

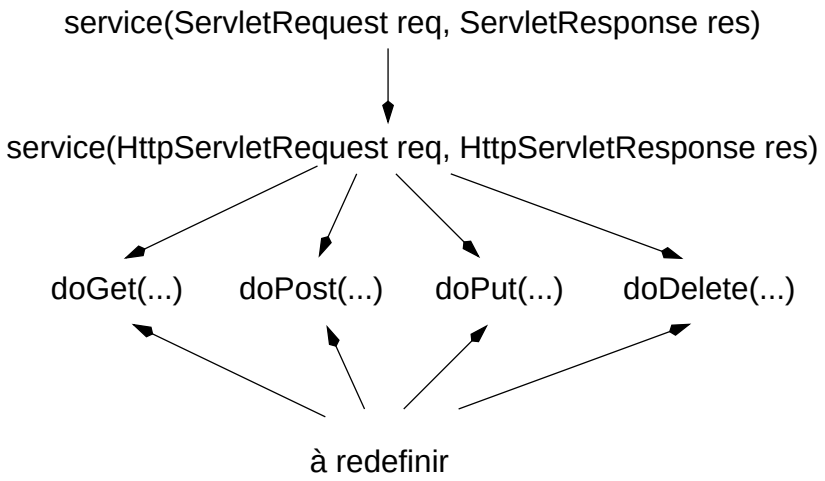


FIG. 17 – Spécialisation de `service`

Mise en œuvre

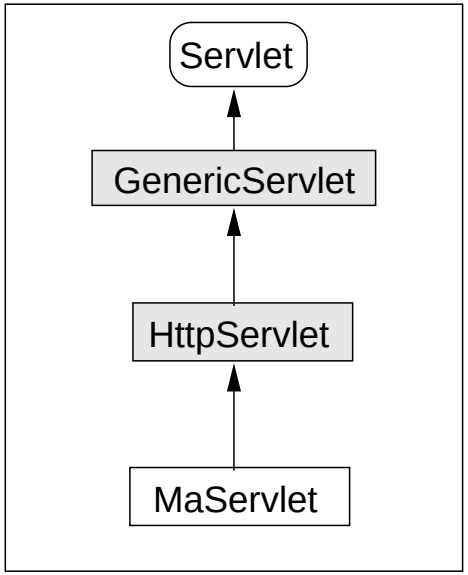


FIG. 18 – Mécanisme

Premier exemple

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;

public class SimpleServletDemo extends HttpServlet
{
    public void doGet (
        HttpServletRequest request,
        HttpServletResponse response
    ) throws ServletException, IOException
    {
        PrintWriter out;
        String titre = "Simple Servlet Demo ";

        response.setContentType("text/html");
        out = response.getWriter();

        out.println(
           "<HTML>\n <HEAD>\n <TITLE>");
        out.print(titre);
        out.println(
           "</TITLE>\n </HEAD>\n <BODY>");
        out.println(
           "    <H1>" + titre + "</H1>");
        out.println("On =E9crit la page");
        out.println("     </BODY>\n</HTML>");
        out.close();
    }
}
```

Exemple avec formulaire

```
<html>
  <head>
    <title>Simple Form Servlet Demo</title>
  </head>

  <body>
    <h1>Simple Form Servlet Demo</h1>
    <form action="http://127.0.0.1:8080/servlet/SimpleFormServletDemo">
      <input type="text" name="demo">
      <input type="submit" value="Go !">
    </form>
  </body>
</html>
```

Exemple avec formulaire

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;

public class SimpleFormServletDemo extends HttpServlet
{
    public void doGet (
        HttpServletRequest request,
        HttpServletResponse response
    ) throws ServletException, IOException
    {
        PrintWriter out;
        String titre = "Simple Form Servlet Demo ";
        response.setContentType("text/html");

        out = response.getWriter();

        out.println("<HTML>\n    <HEAD>\n  <TITLE>");
        out.print(titre);
        out.println("</TITLE>\n   </HEAD>\n <BODY>");
        out.println("       <H1>" + titre + "</H1>");
        out.println("Vous avez saisi : <h2>"
                + request.getParameter("demo") + "</h2>");
        out.println("      </BODY>\n</HTML>");
        out.close();
    }
}
```

Lancement d'une servlet

- pas de méthode universelle
- souvent : `http://nomMachine:8080/servlet/nomServlet`

Gestion des exceptions

- Attention : lors de la gestion des exceptions, les sorties habituelles à l'écran ne sont pas possibles.
- C'est une bonne idée (au moins en phase debogage) de "rediriger" les erreur sur la page de sortie.

```
catch (MonException e)
{
  out.println (e.getMessage());
  e.printStackTrace(out);
}
```

Conclusion

- programmation serveur facile
- tous les avantages de java
- bonnes performances
- se développe de plus en plus